



Escola d'Enginyeria de Telecomunicació i
Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO FINAL DE GRADO

TÍTULO DEL TFG: Programación de un drone Tello Edu

TITULACIÓN: Doble titulación de grado en ingeniería de sistemas aeroespaciales e ingeniería de sistemas de telecomunicaciones

AUTOR: Jaime Jaume Busquets

DIRECTOR: Miguel Valero Garcia

FECHA: 28/01/2022

Título: Programación de un drone Tello Edu

Autor: Jaime Jaume Busquets

Director: Miguel Valero Garcia

Fecha: 28/01/2022

Resumen

Los drones son una herramienta que cada vez está más presente en nuestro día a día y nos ofrecen una infinidad de posibilidades que ayudarán al desarrollo de muchos sectores de trabajo. Pero, ¿conocemos toda la tecnología que hay detrás de estas pequeñas aeronaves?

El objetivo de este proyecto es explorar toda esta tecnología mediante un drone programable como el Tello EDU y documentarla de tal manera que cualquier persona con nociones básicas de programación pueda desarrollar aplicaciones web y móviles relacionadas con el control de una y múltiples aeronaves, inteligencia artificial o procesado de imágenes.

De esta manera, este proyecto va a poder ser usado educativamente para enseñar e introducir a personas dentro del sector de los drones y la programación, de una manera amigable, entretenida y llamativa, ya que se va a explicar como crear aplicaciones como el reconocimiento facial y seguimiento de éste, control del drone mediante gestos manuales o creaciones de coreografías para múltiples drones.

Además, durante todo el trabajo está recogida mi experiencia, diferentes consejos y conclusiones acerca de cada una de las diferentes aplicaciones desarrolladas, sirviendo como base para poder crear funciones más complejas por parte de futuros alumnos. Asimismo, todos los códigos usados estarán a disposición mediante GitHub y se incorporarán demostraciones visuales haciendo uso de Youtube.

Title: Programming a drone Tello EDU

Author: Jaime Jaume Busquets

Director: Miguel Valero Garcia

Date: 28/01/2022

Overview

Drones are devices which are increasingly present in our everyday lives. Drones offer us endless possibilities that can help the development of many working sectors. However, do we really know all the technology behind these small aircraft?

The objective of this project is to explore all this technology through a programmable drone such as the Tello EDU; by documenting it in such a way that anyone with a basic knowledge of programming will be able to develop a web or mobile app app that allows the user to fly one or multiple aircrafts at the same time. As well as using artificial intelligence or image processing.

Hence, this project will be available for educational purposes in order to teach and introduce people to the drone and programming sector, in a friendly, entertaining and attractive way. The project will address different methodologies for creating apps such as facial tracking, controlling the drone through manual gestures or creating choreographies including multiple drones.

Lastly, the user will find different advices, thoughts and conclusions about the different apps that have been developed based on my personal experience. This will serve as the basis for future students to create more complex functions. At the same time, every single code used during this project will be available through GitHub; as well as visual tutorials will be included via YouTube.

ÍNDICE

INTRODUCCIÓN	1
CAPÍTULO 1. DRONE TELLO EDU	2
Información general del Tello EDU	2
Características técnicas del Tello EDU	3
Diagrama componentes aeronave	3
Indicador de estado de la aeronave	3
Especificaciones técnicas	4
SDK 2.0	5
App oficial Tello EDU	5
Modos de vuelo	6
CAPÍTULO 2. ENTORNO PC	7
Instalación del entorno de trabajo	7
Desarrollo de funcionalidades	8
Movimientos Básicos	8
Transmisión de video	9
Control desde teclado y toma de imágenes	11
Mapeo del movimiento	14
Reconocimiento facial y seguimiento	21
Control del drone mediante las manos	24
CAPÍTULO 3. MODO ENJAMBRE	34
Configuración del escenario	34
Control desde teclado del enjambre	36
Coreografías entre drones	37
Cuadrado	38
Círculo	40
Flips	41
Modo rebote	42
CAPÍTULO 4. ENTORNO TELÉFONO MÓVIL	47
Estudio del método de desarrollo	47
IONIC	47
Kivy	48
Elección de framework	48
Generación de la aplicación mediante Kivy	49
Kivy Launcher	49
Buildozer	49

Instalación del entorno de trabajo	50
Aplicación básica para el control del Tello EDU	52
Desarrollo del código	52
Generación de la aplicación móvil con Buildozer	62
Instalación de la aplicación en el dispositivo móvil	64
Configuración de planes de vuelo	64
Visualización imagen del drone	73
Planteamiento de desarrollo propuesto	74
Error de visualización de imagen en dispositivo móvil	76
Modo Enjambre	78
CONCLUSIÓN	84
BIBLIOGRAFÍA	86
ANEXOS	88
Anexo 1. Instrucciones de uso repositorio GitHub	88

INTRODUCCIÓN

Este proyecto está centrado en el estudio de las posibilidades ofrecidas por el drone Tello EDU con un propósito experimental y educacional. El tello EDU es una pequeña aeronave programable de DJI con unas características técnicas limitadas en cuanto a precisión y duración de batería, pero presenta las condiciones perfectas para poder explorar las diferentes capacidades de un drone mediante la programación.

Las diferentes maneras de afrontar algunas de estas capacidades se pueden encontrar en internet de manera muy dispersa y a veces poco entendible. Así que, utilizando la experiencia adquirida durante el doble grado en cuanto a síntesis de información y programación, se va a buscar documentar de manera clara y explicativa cada uno de los aspectos trabajados añadiendo mi experiencia personal, dificultades encontradas y conclusiones extraídas.

Para empezar, se va a explicar la manera de poder realizar misiones básicas desde el ordenador. Entre ellas se encuentra como pilotar el drone mediante teclado, conseguir visualizar la imagen captada por la aeronave en directo o tomar fotografías. También se desarrollarán aplicaciones más complejas como el mapeo del movimiento del drone, el reconocimiento fácil y el seguimiento de éste o el control de la aeronave mediante gestos manuales.

Seguidamente, se abordará el control de múltiples aeronaves, es decir, se documentará cómo realizar la conexión a un enjambre de drones, como controlar su movimiento y la manera de crear diferentes interacciones entre ellos.

Por último, se va a explicar paso a paso cómo crear una aplicación Android que permita la conexión y el control de uno o varios drones. Además, se implementará la posibilidad de configurar planes de vuelo para las aeronaves y se documentará la imposibilidad encontrada a la hora de visualizar la imagen captada por el Tello EDU.

La memoria se divide en diferentes capítulos de manera que la información está ordenada dependiendo del aspecto trabajado, haciendo que la documentación deseada sea fácilmente accesible. Dicho de otro modo, se puede acceder rápidamente a las características de Tello EDU acudiendo al capítulo 1, al desarrollo de aplicaciones en el ordenador mediante el capítulo 2, lo trabajado referente a los enjambres con el capítulo 3 o la creación de la aplicación móvil siguiendo el capítulo 4.

Además, en el anexo ([Anexo 1. Instrucciones de uso repositorio GitHub](#)) se encontrará una pequeña descripción del repositorio de GitHub [1], donde se encuentra todo el código creado durante el trabajo, y en el vídeo de Youtube [2], se pueden encontrar las demostraciones visuales de las aplicaciones desarrolladas más complejas.

CAPÍTULO 1. DRONE TELLO EDU

A continuación, se va a realizar una pequeña descripción de las principales características del drone Tello EDU. Esta información está extraída principalmente del manual de instrucciones de la aeronave (ver [3]) y de las descripciones que ofrecen los vendedores.

Este capítulo servirá para familiarizarse con la aeronave y sus principales especificaciones. Además, se dará a conocer su kit de desarrollo para el software (SDK), que su uso será vital durante todo el proyecto, y la aplicación oficial de DJI con la cual se puede pilotar el drone y realizar diferentes modos de vuelo.

1.1. Información general del Tello EDU

Tello EDU es un pequeño cuadricóptero (ver Fig. 1.1) fabricado por Shenzhen Ryze Technology, con procesadores de Intel y control de vuelo incorporado por DJI. Este drone es la versión mejorada del drone Tello, en el cual no se permitía el modo Swarm y su SDK era más limitado.

Esta aeronave es idónea para poder desarrollar infinidad de aplicaciones de manera educativa gracias a la facilidad que ofrece para ser programado mediante diferentes lenguajes de programación (Scratch, Python y Swift...)

Actualmente, el precio de esta aeronave es de 159 € en la web oficial de DJI. Este precio incluye la aeronave, 4 hélices de recambio, los protectores de las hélices, 1 batería, 1 cable micro USB, 1 herramienta de desinstalación de hélices y 4 bases para realizar misiones.



Fig. 1.1 Drone Tello EDU

1.2. Características técnicas del Tello EDU

En el siguiente apartado se van a documentar las características técnicas del Tello EDU. Se darán a conocer sus componentes, los indicadores de estado que ofrece su LED y las diferentes especificaciones de la aeronave, la cámara y la batería.

1.2.1. Diagrama componentes aeronave

En la Fig. 1.2 se pueden observar los diferentes componentes que posee el Tello EDU. Me gustaría destacar que el sistema de posicionamiento visual hace referencia a una cámara, de características técnicas inferiores a la frontal, que puede ser ciertamente útil en algunos casos.

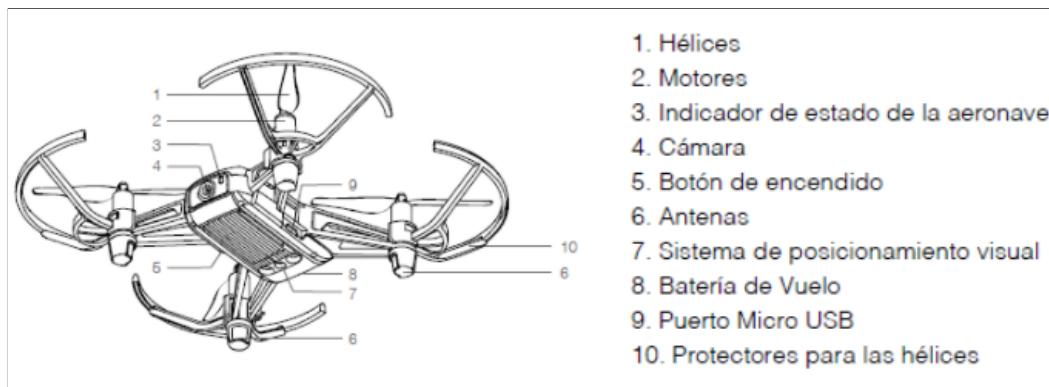


Fig. 1.2 Diagrama componentes Tello EDU

1.2.2. Indicador de estado de la aeronave

El indicador de estado del Tello EDU consiste en un LED ubicado en la parte frontal de la aeronave, junto a la cámara delantera. Éste nos ofrece información acerca de la condición del drone y su batería dependiendo del color y patrón de repetición. En la Tabla 1.1 se definen los diferentes indicadores de estado.

Tabla 1.1. Estados de la aeronave según comportamiento del indicador

	Color	Patrón	Estado aeronave
Estados normales	Alterna entre rojo, verde y amarillo	Parpadea	Encendido y realizando pruebas de autodiagnóstico

	Verde	Parpadea dos veces de forma periódica	Sistema de posicionamiento visual activado
	Amarillo	Parpadea lentamente	Sistema de posicionamiento visual no disponible, la aeronave se encuentra en modo Atti (Attitude mode)
Estado de carga	Azul	Luz fija	La carga ha finalizado
	Azul	Parpadea lentamente	Cargando
	Azul	Parpadea rápidamente	Error de carga
Estados de advertencia	Amarillo	Parpadea rápidamente	Pérdida de señal del control remoto
	Rojo	Parpadea lentamente	Advertencia de nivel
	Rojo	Parpadea rápidamente	Batería baja crítica
	Rojo	Luz fija	Error crítico

1.2.3. Especificaciones técnicas

En cuanto a las especificaciones técnicas, en la Tabla 1.2 se pueden observar las referentes a la aeronave, la cámara o la batería.

Tabla 1.2. Especificaciones técnicas

Aeronave	Modelo	TLW004
	Peso	87 g
	Dimensiones	98x92,5x41 mm
	Velocidad máxima	28,8 km/h (8 m/s)
	Tiempo de vuelo máx.	13 minutos
	Distancia de vuelo máx.	100 m
	Altura de vuelo máx.	30 m
	Intervalo de temperatura de funcionamiento	De 0°C a 40°C
	Intervalo de frecuencias de funcionamiento	De 2,4 a 2,4835 GHz
	Transmisor (PIRE)	20 dBm (FCC)
		19 dBm (CE)
		19 dBm (SRRC)
Cámara	Tamaño imagen máximo	2592x1936
	Formato de imagen	JPG

	Modos de grabación de vídeo	HD: 1280x720 30p
	Formato de vídeo	MP4
Batería	Capacidad	1100 mAh
	Voltaje	3,8 V
	Tipo de batería	LiPo
	Energía	4,18 Wh
	Peso neto	25 ± 2 g
	Intervalo de temperatura de carga	de 5°C a 45°C
	Potencia de carga máx.	10 W

1.3. SDK 2.0

El Tello Edu cuenta con un Kit de Desarrollo para el Software que sirve de base para la creación de diferentes aplicaciones y funcionalidades. Al igual que el manual de instrucciones comentado anteriormente, también existe un manual de uso del SDK 2.0 (ver [4]) donde se explica el funcionamiento de la aeronave y cómo controlarla mediante todo tipo de comandas.

Durante todo el trabajo se va a hacer uso de estas comandas ofrecidas por el SDK, por lo que más adelante se entrará en detalle en ellas.

1.4. App oficial Tello EDU

El Tello EDU puede ser controlado mediante la aplicación llamada Tello. Esta app está desarrollada por Shenzhen RYZE Tech Co. Ltd. y gracias a ella se puede pilotar fácilmente la aeronave, así como sacar partido a todas las funcionalidades que ésta ofrece.

La interfaz principal (Fig. 1.3) ofrece los dos joysticks necesarios para pilotar el drone, botones para iniciar el despegue, elegir entre los diferentes modos de vuelo, acceder a la configuración, ver los archivos multimedia, seleccionar fotografía o vídeo o para tomar fotos o iniciar la grabación, y finalmente, también se puede observar cierta información acerca de la batería, la señal, la posición y velocidad de la aeronave.

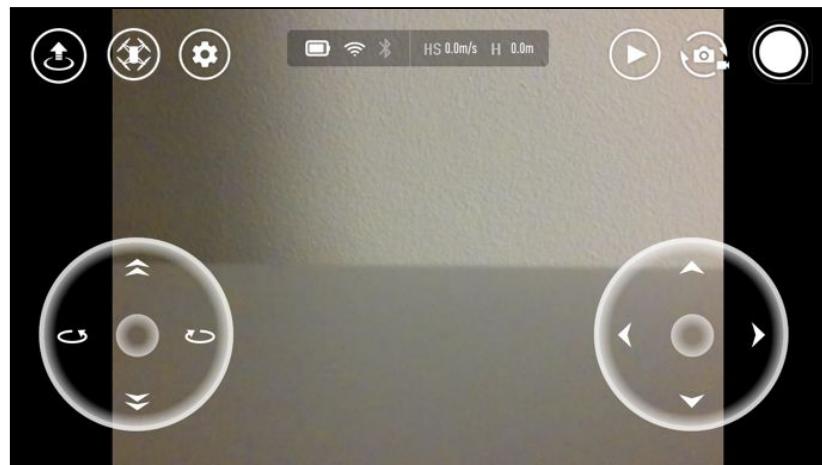


Fig. 1.3 Interfaz principal APP Tello

1.4.1. Modos de vuelo

La aplicación oficial de Tello EDU ofrece diferentes modos de vuelo con distintas funcionalidades como se puede observar en la Fig. 1.4. Posteriormente, se va a dar uso a algunas de estas capacidades que ofrece el drone en las aplicaciones creadas, y además, se desarrollarán nuevos modos de vuelo como ya se ha comentado en la introducción, los cuales podrían incorporarse a la app oficial sin problema.



Fig. 1.4 Modos de vuelo y sus descripciones

CAPÍTULO 2. ENTORNO PC

Para empezar, cabe destacar que el trabajo ha sido desarrollado en Windows, ya que es el sistema operativo de mi ordenador personal y es el sistema en el que mejor me desenvuelvo.

Por otro lado, se ha usado el lenguaje de programación Python, el cual ya había usado durante mis estudios y además ofrece un amplio abanico de posibilidades para sacar partido al Tello EDU.

En este capítulo se encontrará la información necesaria para instalar el entorno de trabajo, asimismo, se desarrollarán diferentes funcionalidades del drone de manera muy descriptiva. Además, recordar que están disponibles demostraciones visuales (ver [2]) de las aplicaciones creadas más complejas y todos los códigos utilizados (ver [1]).

Me gustaría destacar, que para el desarrollo de muchas de las funcionalidades que se trabajarán en el entorno PC, me fueron de gran ayuda una serie de cursos (ver [5]) que ofrecen un gran aprendizaje acerca de Tello EDU, procesado y desarrollo de imagen e inteligencia artificial. De esta manera, he podido experimentar su funcionamiento familiarizándome con el drone y la programación que hay detrás para poder crear aplicaciones propias en los siguientes apartados. Las funcionalidades trabajadas serán explicadas desde mi punto de vista, ofreciendo los problemas que me he encontrado a la hora de ponerlas en práctica, planteando posibles mejoras y corrigiendo errores encontrados en los propios cursos.

2.1. Instalación del entorno de trabajo

Como he comentado anteriormente, el lenguaje de programación usado durante el trabajo es Python, por lo que es primordial la correcta instalación de éste. La versión que se compatibiliza mejor con el Tello EDU es la 3.7 y desde la página web oficial de Python se puede acceder a la descarga.

Por otro lado, también es aconsejable la descarga de algún IDE (Entorno de Desarrollo Integrado) para así poder trabajar fácil y cómodamente con Python. Además, estos ofrecen la posibilidad de descargar e importar librerías necesarias para el proyecto de manera sencilla. En mi caso hice uso de la versión gratuita de PyCharm.

Finalmente, otro requisito importante es la instalación de la librería djitellopy, donde todas las comandas que hacíamos mención en el punto [1.3 SDK 2.0](#) están incorporadas. Mediante su Github (ver [6]), se puede observar la información necesaria acerca de esta librería y las posibilidades que nos ofrece. Más adelante, dependiendo de las diferentes funcionalidades que se desarrolleen, se tendrán que tener en cuenta otras librerías.

2.2. Desarrollo de funcionalidades

Como se ha comentado anteriormente, estas primeras funcionalidades son reproducciones experimentales de diferentes cursos ofrecidos en [5], en concreto uno subido a Youtube (ver [7]) me ha sido de gran ayuda. Mediante ello, se va a obtener un conocimiento acerca de la programación detrás del Tello EDU que posteriormente será usado para generar aplicaciones propias.

Además, se va a realizar una conclusión de cada apartado documentando las dificultades encontradas, ofreciendo diferentes consejos, relacionando conceptos estudiados durante el doble grado y sugiriendo posibles cambios que mejorarían la funcionalidad.

2.2.1. Movimientos Básicos

Para empezar, es importante conocer la manera de conectar el drone al ordenador. Para ello, simplemente hay que encenderlo, conectarnos desde el PC a su conexión Wi-Fi y realizar la comanda `connect()`. En mi caso, siempre pido que me indique la batería del drone mediante la comanda `get_battery()`, ya que es una información vital a la hora de trabajar con él, y además, nos aseguramos de que la conexión se ha realizado con éxito al recibir respuesta.

Seguidamente, se debe conocer que para despegar se usa la comanda `takeoff()` y para aterrizar `land()`.

Por último, falta por indicar las comandas de movimiento básicas. Por un lado, podemos indicar la dirección de movimiento o de giro y los centímetros o grados deseados utilizando comandas como `move_up(100)`, `move_right(30)`, `move_forward(40)`, `rotate_clockwise(50)` y sus comandas opuestas.

Por otro lado, se puede usar la comanda `send_rc_control()`, donde es necesario indicar las velocidades de todas las componentes de movimiento. De esta manera, podemos indicar, por ejemplo, que se mueva a 50 cm/s hacia delante (`send_rc_control(0, 50, 0, 0)`). Si por el contrario queremos que se mueva hacia atrás, deberíamos enviar la comanda (`send_rc_control(0, -50, 0, 0)`). En la Fig. 2.1, se pueden observar cómo se realizan estas comandas para cada dirección.

Este segundo modo resulta más interesante y recomiendo su uso, ya que se podrá ligar al control mediante teclado del drone, simulando los joysticks que vimos en la interfaz de la aplicación Tello (Fig. 1.4). El código de este apartado se puede apreciar en la Fig. 2.1.

```
from djitellopy import tello
from time import sleep

me = tello.Tello()
me.connect()
print(me.get_battery())

me.takeoff()
""" Go forward/backward"""
me.send_rc_control(0, 50, 0, 0)
sleep(2)
"""Go right/left"""
me.send_rc_control(30, 0, 0, 0)
sleep(2)
"""Rotate clockwise/anticlockwise (yaw)"""
me.send_rc_control(0, 0, 0, 30)
sleep(2)
"""Go up/down """
me.send_rc_control(0, 0, 30, 0)
sleep(2)
me.send_rc_control(0, 0, 0, 0)
me.land()
```

Fig. 2.1 Código movimientos básicos

2.2.2. Transmisión de video

Para poder obtener en nuestro ordenador la imagen captada por el drone, se va a hacer uso de la librería opencv-python, que es una de las librerías que mejor funciona para el procesado de imagen.

Una vez conectados al drone como se ha indicado anteriormente, la comanda necesaria para empezar a transmitir es streamon(), así como streamoff() para finalizar. Además, se va a hacer uso de la comanda get_frame_read().frame, que nos ofrece el frame recibido por el drone actualizado en todo momento, es decir, la imagen que queremos observar. Cabe destacar que hay que añadir un pequeño delay de milisegundos (cv2.waitKey(1)), de no ser así, no podríamos ver la imagen.

Por último, se hará uso de opencv para tratar esta imagen. En mi caso, recomiendo usar la función cv2.resize(), ya que de no ser así la ventana donde podemos observar la imagen mediante cv2.imshow(), es muy grande y puede que nos afecte en el procesado de ésta dependiendo de la capacidad del ordenador. En la Fig. 2.2 se puede encontrar el código comentado.

```

from djitellopy import tello
import cv2

me = tello.Tello()
me.connect()
print(me.get_battery())

me.streamon()

while True:
    img = me.get_frame_read().frame
    img = cv2.resize(img,(360,240))
    cv2.imshow("Image", img)
    cv2.waitKey(1)

```

Fig. 2.2 Código captura de imagen

Por otro lado, en el tutorial no se menciona como acceder a las imágenes ofrecidas por la cámara inferior comentada en el apartado [1.2.1 Diagrama componentes aeronave](#). Para ello, se debe hacer uso de la comanda `set_video_direction()`, como se puede observar en la Fig. 2.3. Es importante saber que se requiere al menos el firmware v02.05.01.17 instalado en el Tello EDU.

```

from djitellopy import tello
import cv2

me = tello.Tello()
me.connect()
print(me.get_battery())

"Cámara frontal (0) o inferior (1)"
me.set_video_direction(1)
me.streamon()

while True:
    img = me.get_frame_read().frame
    img = cv2.resize(img, (320, 240))
    cv2.imshow("Image", img)
    cv2.waitKey(1)

```

Fig. 2.3 Código para cámara inferior

Cabe destacar que como ya se ha comentado anteriormente, esta cámara posee una calidad inferior a la frontal y proporciona una imagen en escala de grises con una resolución de 320x240 píxeles. Además, existe un error en el frame proporcionado que hace que la imagen se vea como en el Fig. 2.4.

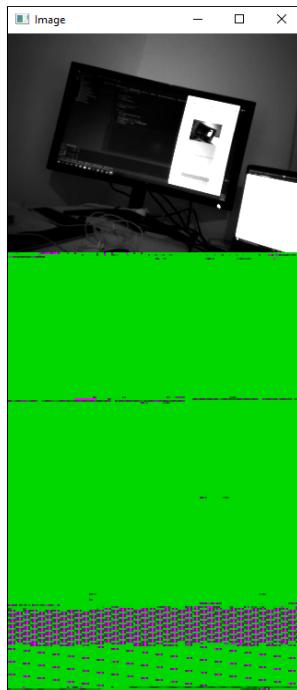


Fig. 2.4 Frame recibido por cámara inferior

Tratando la matriz de la imagen se podría llegar a eliminar la banda verde, pero como durante el trabajo necesitamos la mayor calidad de imagen posible, únicamente se va a tratar con la cámara frontal, la cual funciona correctamente.

2.2.3. Control desde teclado y toma de imágenes

Para el desarrollo de este apartado se ha seguido con el tutorial mencionado anteriormente (ver [7]), es decir, se ha reproducido a modo experimental esta aplicación y se va a proceder a hacer una explicación descriptiva acerca de ella.

El control de la aeronave así como la posibilidad de poder tomar fotos son funciones esenciales para cualquier aplicación de control de drones. Para poder desarrollar estas funciones, se va a hacer uso de la librería pygame, así como las mencionadas anteriormente. Primeramente, el objetivo es crear un módulo donde incluir todo el procesado del teclado para poder utilizarlo siempre que se deseé más adelante con facilidad.

A la hora de crear el módulo, hay que tener en cuenta que pygame se basa en el desarrollo de juegos, por lo que es necesario tener una ventana de juego siempre disponible para que se detecten las teclas presionadas del teclado. Mediante la función `pygame.display.set_mode()` e indicando el tamaño deseado para la ventana, se crea.

Posteriormente, es necesario crear una función que indique si existe una tecla presionada. Para comprobar si esto está sucediendo, se hace uso de pygame.key.get_pressed(). Es importante saber que se debe seguir un formato para cada tecla presionada en esta librería. Por ejemplo, si la tecla “UP” está presionada, se debe tratar como K_UP.

```
import pygame

def init():
    pygame.init()
    """Window Game creation"""
    win = pygame.display.set_mode((400, 400))

def getKey(keyName):
    ans = False
    for eve in pygame.event.get(): pass
    keyInput = pygame.key.get_pressed()
    myKey = getattr(pygame, 'K_{}'.format(keyName))
    if keyInput[myKey]:
        ans = True
    pygame.display.update()
    return ans
```

Fig. 2.5 Módulo procesado de teclado

Una vez creado el módulo como se puede observar en la Fig. 2.5, ya se puede usar para generar una pequeña aplicación que permita controlar el drone utilizando el teclado y tomar fotos.

Inicialmente, se deben importar todas las librerías necesarias, además del nuevo módulo, y realizar la conexión a la aeronave junto con el inicio de retransmisión de video.

A continuación, se va a crear una función donde se definirá la velocidad de movimiento que se desee (50 cm/s en este caso) y dependiendo de la tecla presionada, se devolverán los parámetros de velocidad en dirección izquierda o derecha (lr), delante o detrás (fb), arriba o abajo (ud) y giro en sentido horario o antihorario (yv). Por ejemplo, si se presiona la tecla “LEFT”, el parámetro lr deberá ser -50 y todos los demás 0.

Una vez obtenidos estos parámetros, sólo hay que aplicar la comanda que se utilizó en [2.2.1. Movimientos Básicos](#) de la siguiente manera: send_rc_control(lr, fb, ud, yv).

Hay que destacar que este modo de aplicar el movimiento es interesante y eficiente, ya que se pueden presionar varias teclas al mismo tiempo sin que suponga ningún problema, porque en la comanda send_rc_control(), se tienen que definir las velocidades en las 4 direcciones siempre.

Por otro lado, hay que incorporar que al pulsar la tecla “q” se realice la comanda land() y mediante la tecla “e” takeoff().

Para terminar, presionando la tecla “z”, se va a realizar la función de opencv-python imwrite(f'Resources/Images/{time.time()}.jpg', img). Es decir, se van a tomar imágenes de la retransmisión en vídeo y se guardarán en el directorio Resources/Images (creado anteriormente) en formato jpg utilizando como nombre de archivo el tiempo en que se tomó la foto. De esta manera, nunca se van a sobrescribir las imágenes al ser tomadas en diferentes instantes y tener nombres diferentes.

Es importante dejar un pequeño delay después de presionar la tecla “z” para tomar fotos, ya que de no ser así, se van a guardar una gran cantidad de fotografías iguales. La codificación en torno a esta funcionalidad se encuentra en la Fig. 2.6.

```
from djitellopy import tello
import KeyPressModule as kp
import time
import cv2

kp.init()
me = tello.Tello()
me.connect()
print(me.get_battery())
global img
me.streamon()

def getKeyboardInput():
    lr, fb, ud, yv = 0, 0, 0, 0
    speed = 50

    if kp.getKey("LEFT"): lr = -speed
    elif kp.getKey("RIGHT"): lr = speed

    if kp.getKey("UP"): fb = speed
    elif kp.getKey("DOWN"): fb = -speed

    if kp.getKey("w"): ud = speed
    elif kp.getKey("s"): ud = -speed

    if kp.getKey("a"): yv = -speed
    elif kp.getKey("d"): yv = speed

    if kp.getKey("q"): me.land(); time.sleep(3)
    if kp.getKey("e"): me.takeoff()

    if kp.getKey('z'):
        cv2.imwrite(f'Resources/Images/{time.time()}.jpg', img)
        time.sleep(0.3)

    return [lr, fb, ud, yv]

while True:
    vals = getKeyboardInput()
    me.send_rc_control(vals[0], vals[1], vals[2], vals[3])
    img = me.get_frame_read().frame
    img = cv2.resize(img, (360, 240))
    cv2.imshow("Image", img)
    cv2.waitKey(1)
```

Fig. 2.6 Código control desde teclado y toma de fotos

Desde mi punto de vista, el desarrollo de esta funcionalidad no ha sido muy complejo, me ha ayudado a entender la eficacia que nos aportan los módulos creados y la manera en que proceder para dar comandas al drone dependiendo de las teclas presionadas. En contraparte, la necesidad de tener una ventana de juego de pygame únicamente para que detecte las teclas presionadas me ha parecido poco eficiente. Además, teniendo en cuenta que uno de los objetivos finales es la creación de una aplicación móvil para manejar el Tello EDU, es inviable el uso de esta librería. En el capítulo relacionado con el entorno móvil se verá la dinámica a seguir para poder trabajar sin una ventana de juego.

2.2.4. Mapeo del movimiento

La siguiente aplicación que queremos desarrollar es el mapeo de todo el movimiento realizado por nuestro drone en 2D. Esta aplicación también está extraída del curso mencionado anteriormente (ver [7]), por lo que se va a proceder, al igual que en el apartado anterior, a hacer una revisión experimental de la aplicación aportando documentación y dando una conclusión final.

También, se va a realizar un estudio propio acerca de la precisión en la velocidad lineal y angular del Tello EDU y se corregirá un error cometido en el tutorial. Esta equivocación en el código propuesto mapeaba erróneamente el movimiento del drone al no haber tenido en cuenta los movimientos diagonales que involucran más de una tecla pulsada al mismo tiempo.

Para empezar, se va realizar una pequeña explicación teórica de cómo vamos a afrontar la codificación de esta funcionalidad.

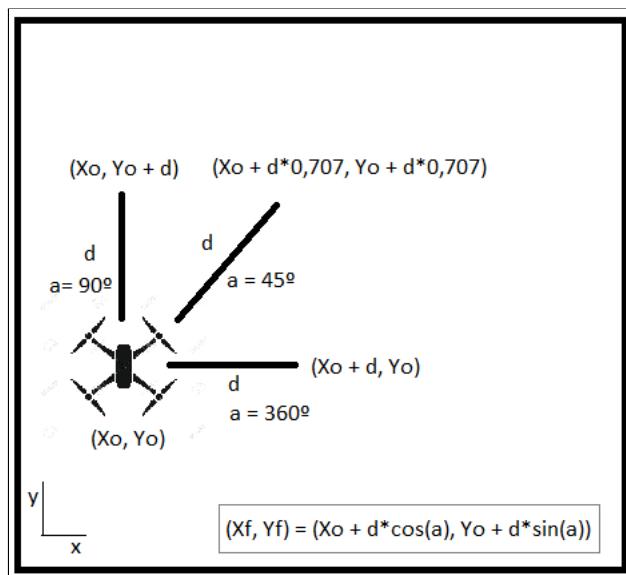


Fig. 2.7 Teoría detrás del mapeo

Como se puede observar en la Fig. 2.7, nuestra aeronave siempre va a partir de una posición inicial (X_0 , Y_0), específicamente de donde se ha despegado. Teniendo en cuenta que se puede fijar la velocidad lineal y angular de nuestro drone y que se programa cada cuanto tiempo se va a chequear la posición, realizar el producto de ambas constantes ($v*t$) nos ofrecerá la distancia recorrida ("d") o los grados girados en el intervalo de tiempo programado.

Al ya conocer estas constantes, mediante trigonometría y siempre teniendo en cuenta los valores de "x" e "y" anteriores (como se indica en la Fig. 2.7), se va a poder obtener la posición del cuadricóptero en todo momento.

Una vez realizado el estudio teórico de la metodología a seguir, se va a explicar su posterior codificación.

Para poder desarrollar el código, es necesario importar las librerías numpy y math a las mencionadas anteriormente. También será necesario el uso del módulo y prácticamente la totalidad del código creado en [2.2.3. Control desde teclado y toma de imágenes](#), ya que se busca seguir moviendo el drone usando el teclado y que este movimiento quede mapeado.

A continuación, va a ser necesario fijar la velocidad lineal y angular, así como el intervalo de tiempo deseado. Teóricamente, hasta ahora se había visto que si se usa la comanda, por ejemplo, send_rc_control(0, 0, -50, 0), el drone desciende a 50 cm/s. En la práctica, esta velocidad nunca es constante, ya que el Tello EDU no es muy preciso, por lo que si hacemos los cálculos de posición con una velocidad teórica, en la práctica el drone no estará ubicado donde indica el mapa.

Por ello, se va a realizar un pequeño estudio buscando el valor más real posible de velocidad. Se fijará la velocidad teórica del drone a 15 cm/s y se le ordenará que avance a esa velocidad durante 10 segundos. Posteriormente, se medirá la distancia real que ha recorrido en ese tiempo obteniendo una velocidad real (distancia/tiempo). Este proceso se repetirá 10 veces, por la poca precisión del Tello EDU y finalmente se promediarán todos los resultados.

Tabla 2.1. Estudio de velocidad lineal real

Distancia recorrida en 10 s [cm]	Velocidad Real [cm/s]
160	16
135	13,5
120	12
137	13,7
119	11,9
206	20,6
245	24,5
138	13,8

144	14,4
245	24,5

Una vez realizado el estudio, se ha obtenido un valor promediado de 16,5 cm/s. A primera instancia, parece un resultado real relativamente parecido a los 15 cm/s teóricos, pero si calculamos su desviación típica se obtiene un valor de 11,42 cm, es decir, de media el resultado será 11,42 centímetros erróneo. Además, hay casos en que la velocidad real es casi 10 cm/s superior a la teórica. En el tutorial seguido, el valor promedio es de 11,7 cm/s, por lo que también se puede apreciar la poca precisión del drone comparando los resultados finales de ambos estudios.

Para la velocidad angular, se va a realizar el mismo procedimiento fijando la velocidad angular teórica a 50 °/s, pero en este caso, se medirá el tiempo que tarda el drone en dar una vuelta completa. De esta manera, no se tendrá que medir el ángulo de giro realizado, que es una medición aproximada y poco precisa. El valor promedio obtenido fue 9,98 segundos, con una desviación típica muy pequeña, es decir, todos los resultados son iguales. Este valor implica una velocidad angular real de 36 °/s aproximadamente, bastante diferente a los 50 °/s teóricos. En el caso del tutorial, los resultados obtenidos son iguales a los realizados en este caso.

Tabla 2.2. Estudio de velocidad angular real

Tiempo en completar 360° [s]	Velocidad Real [°/s]
9,95	36,18
10	36
9.98	36,07
9,83	36,62
10,03	35,89
9,91	36,32
9,94	36,21
10,01	35,95
10	36
9,97	36,1

El siguiente paso es indicar los valores de “a” para cada tecla. Por ejemplo, al pulsar la tecla “RIGHT”, indicaremos que “a” es 360° ya que no tenemos componente vertical ($\sin(360^\circ) = 0$), por otro lado, si por ejemplo pulsamos la tecla “DOWN”, el drone deberá retroceder, por lo que “a” tiene que ser igual a 270°, siendo la componente “y” negativa y obteniendo lógicamente un valor inferior en el sumatorio de la posición vertical.

Finalmente, mediante la librería opencv-python, crearemos pequeños círculos rojos para cada punto recorrido y un círculo verde para la posición actual.

Además, incorporaremos los valores de “x” e “y” en la ubicación en que nos encontramos. Es interesante saber que en opencv las representaciones de color son en BGR, por lo que el color rojo sera (0, 0, 255).

En las siguientes figuras, Fig. 2.8, y Fig. 2.9, se podrá observar el código completo de la aplicación con comentarios explicativos y una muestra de la interfaz que simula nuestro mapeo.

```
from djitellopy import tello
import KeyPressModule as kp
import numpy as np
import cv2
import math
from time import sleep

#####
#PARAMETERS #####
fSpeed = 165/10 #Velocidad lineal real
aSpeed = 36 #Velocidad angular real
interval = 0.25 #Intervalo de tiempo

dInterval = fSpeed * interval #Distancia recorrida durante un intervalo
aInterval = aSpeed * interval #Grados girados durante un intervalo
x, y = 500, 500 #x0 y0
a = 0
yaw = 0

kp.init()
me = tello.Tello()
me.connect()
print(me.get_battery())

points = [(0, 0), (0, 0)]

def getKeyboardInput():
    lr, fb, ud, yv = 0, 0, 0, 0
    aspeed = 50 #Velocidad angular teórica
    speed = 15 #Velocidad lineal teórica
    global x, y, yaw, a
    d = 0
    if kp.getKey("LEFT"):
        lr = -speed
        d = dInterval
        a = 180

    elif kp.getKey("RIGHT"):
        lr = speed
        d = -dInterval
        a = 360

    if kp.getKey("UP"):
        fb = speed
        d = -dInterval
        a = 90

    elif kp.getKey("DOWN"):
        fb = -speed
        d = -dInterval
        a = 270

    if kp.getKey("w"):
        ud = speed
    elif kp.getKey("s"):
        ud = -speed

    if kp.getKey("a"):
        yv = -aspeed
        yaw -= aInterval

    elif kp.getKey("d"):
```

```

yv = aspeed
yaw += alinterval

if kp.getKey("q"): me.land()
if kp.getKey("e"): me.takeoff()

sleep(interval)
#Sumatorio de las posiciones
a += yaw
x += int(d * math.cos(math.radians(a)))
y += int(d * math.sin(math.radians(a)))

return[ir, fb, ud, yv, x, y]

def drawPoints(img, points):
    for point in points:
        cv2.circle(img, point, 5, (0, 0, 255), cv2.FILLED) #Círculos rojos para cada punto

    cv2.circle(img, points[-1], 8, (0, 255, 0), cv2.FILLED) #Círculo verde para posición actual
    cv2.putText(img, f'({(points[-1][0]- 500)/100},{(points[-1][1]- 500)/100})m',
               (points[-1][0]+10, points[-1][1]+30), cv2.FONT_HERSHEY_PLAIN, 1, (255, 0, 255), 1)

while True:
    vals = getKeyboardInput()
    me.send_rc_control(vals[0], vals[1], vals[2], vals[3])

    img = np.zeros((1000, 1000, 3), np.uint8) #Creación imagen negra (mapa) como matriz de 0
    # y valores irán de 0 a 256 (2^8) como integers sin signo.
    if (points[-1][0] != vals[4] or points[-1][1] != vals[5]): #Solo dibuja si se mueve
        points.append((vals[4], vals[5]))
    drawPoints(img, points)
    cv2.imshow("Output", img)
    cv2.waitKey(1)

```

Fig. 2.8 Código aplicación mapeo



Fig. 2.9 Interfaz mapeo

Para finalizar, me gustaría comentar que he encontrado un pequeño error en esta funcionalidad. A la hora de querer hacer volar el drone en diagonal, dos teclas deben ser pulsadas, por ejemplo, “UP” y “LEFT”. Así que, con el modo en que está desarrollado el código (ver Fig. 2.8) sólo se definiría un único valor de “a”. Es decir, el drone volaría en diagonal pero en el mapeo aparecería

volando hacia delante o hacia la izquierda. Para corregirlo, es necesario añadir los ángulos para cada una de las posibles diagonales como se muestra en la Fig. 2.10. Asimismo, otros pequeños cambios serán necesarios para el correcto funcionamiento de la aplicación en cuanto a los movimientos en diagonal. El código completo se encuentra disponible en GitHub [1] y en el inicio del video de Youtube (ver [2]) se encontrará la demostración visual del correcto funcionamiento de esta aplicación.

```

if kp.getKey("LEFT") and kp.getKey("UP"):
    a = 135
    fb = speed
    lr = -speed
    d = dInterval
elif kp.getKey("RIGHT") and kp.getKey("UP"):
    a = 45
    fb = speed
    lr = speed
    d = dInterval
elif kp.getKey("LEFT") and kp.getKey("DOWN"):
    a = 225
    fb = -speed
    lr = -speed
    d = dInterval
elif kp.getKey("RIGHT") and kp.getKey("DOWN"):
    a = 315
    fb = -speed
    lr = speed
    d = dInterval

```

Fig. 2.10 Definición de ángulos para diagonales

Por otro lado, aunque el vídeo del tutorial seguido tiene cerca de dos millones de visualizaciones, no he podido encontrar ningún comentario que exponga el error cometido, por lo que procedí a enviar un correo electrónico al autor del tutorial informando de la equivocación y sugiriendo el cambio mencionado. Su equipo de soporte lo anotó y me lo agradeció (Fig. 2.11). Asimismo, procedí a dejar un comentario en el video (Fig. 2.12), de este modo, el error quizás podrá ser corregido o servirá de ayuda a futuros visualizadores del tutorial.

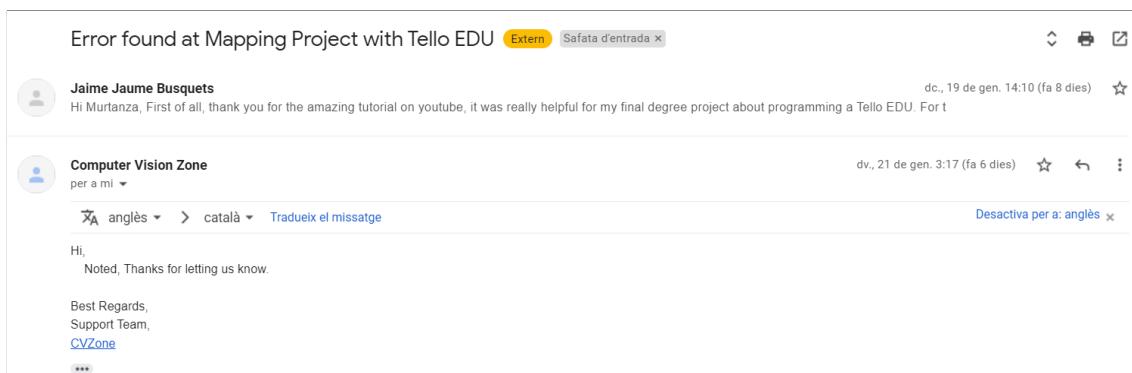


Fig. 2.11 Email de contestación del autor

Jaume Busquets hace 7 días (editado)
Hi Murtanza,

First of all, thank you for this amazing tutorial, it was really helpful for my final degree project so I want to coment you an error I found at the Mapping activity.

If you are pressing for example LEFT and UP at the same time, the tello is moving in diagonal but the mapping not. Thats because It only takes one value of "a", 180 or 90. The function getKeyboardInput() should be like that:

```

def getKeyboardInput():
    lr, fb, ud, yv = 0, 0, 0, 0
    aspeed = 50
    speed = 15
    global x, y, yaw, a
    d = 0
    if kp.getKey("LEFT") and kp.getKey("UP"):
        a = 135
        fb = -speed
        lr = -speed
        d = dInterval
    elif kp.getKey("RIGHT") and kp.getKey("UP"):
        a = 45
        fb = speed
        lr = speed
        d = dInterval
    elif kp.getKey("LEFT") and kp.getKey("DOWN"):
        a = 225
        fb = -speed
        lr = -speed
        d = dInterval
    elif kp.getKey("RIGHT") and kp.getKey("DOWN"):
        a = 315
        fb = -speed
        lr = speed
        d = dInterval
    else:
        if kp.getKey("LEFT"):
            lr = -speed
            d = dInterval
            a = 180
        elif kp.getKey("RIGHT"):
            lr = speed
            d = dInterval
            a = 360
        if kp.getKey("UP"):
            fb = speed
            d = dInterval
            a = 90
        elif kp.getKey("DOWN"):
            fb = -speed
            d = dInterval
            a = 270
        if kp.getKey("w"):
            ud = speed
        elif kp.getKey("s"):
            ud = -speed
        if kp.getKey("a"):
            yv = -aspeed
            yaw += aInterval
        elif kp.getKey("d"):
            yv = aspeed
            yaw -= aInterval
    if kp.getKey("q"): me.land()
    if kp.getKey("e"): me.takeoff()
    sleep(interval)

    a += yaw
    x += int(d * math.cos(math.radians(a)))
    y += int(-1*d * math.sin(math.radians(a)))

    return[lr, fb, ud, yv, x, y]

```

Thank you for your amazing work, I hope my comment could help you or others to get a more robust application.

Fig. 2.12 Comentario en Youtube exponiendo el error

Para dar por terminado este apartado, me gustaría comentar que esta función me ha parecido muy interesante, ya que se puede extrapolar a poder chequear todo el recorrido que ha realizado tu aeronave en un mapa. Además, esta dinámica posibilita el conocimiento del punto de despegue, por lo que a pesar de la poca precisión, se podría llegar a programar un modo Return to Home (RTH) como en los drones más profesionales. Este modo es vital en el uso de estas aeronaves, ya que en caso de pérdida de cobertura, el drone volverá a su punto de despegue.

Más adelante, se hará uso de este sistema para crear una aplicación para los enjambres de drones, ya que nos ayudará a conocer si dos drones van a entrar en colisión.

2.2.5. Reconocimiento facial y seguimiento

La siguiente funcionalidad que se ha desarrollado consiste en el reconocimiento facial y posterior seguimiento de éste, es decir, se va a programar que nuestro drone nos reconozca la cara y nos siga. Esta aplicación también está extraída del curso (ver [7]), por lo que se seguirá con la misma dinámica que hasta ahora.

Primero, como ya se ha hecho en el apartado anterior, se va a explicar la teoría que hay detrás del código, para que así sea más fácil la compresión de éste.

Mediante la librería opencv-python, se pueden reconocer caras y marcarlas con un rectángulo, más adelante se entrará en detalle en cómo hacerlo. Como se puede ver en la Fig. 2.13, si la cara está más alejada, el área del correspondiente rectángulo será más pequeña, y si este área es inferior a un valor establecido se le indicará al drone que avance. Si por el contrario el área es superior al valor establecido, se le indicará que retroceda. En el caso que la aeronave procese un área que se encuentre entre el límite de áreas inferior y superior, no se moverá.

Esto puede desencadenar en un sistema inestable y no parar de realizar movimientos hacia delante y hacia atrás al no conseguir que el área esté dentro del rango deseado, por eso, se implementará un controlador PD que tendrá en cuenta el error de área actual y anterior para evitar el balanceo del drone.

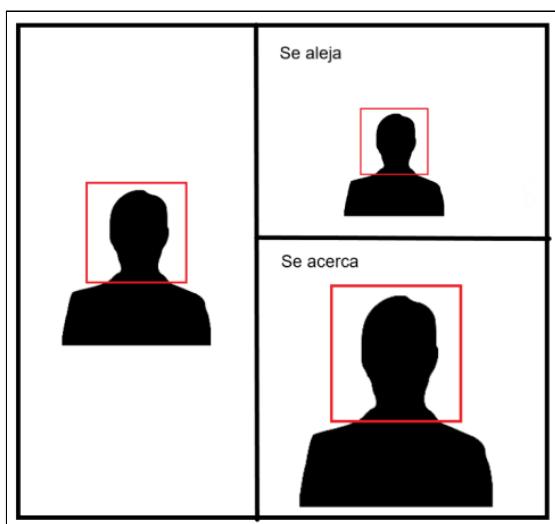


Fig. 2.13 Teoría movimiento tras reconocimiento facial

Por otro lado, como se puede observar en la Fig. 2.14, si la cara no está centrada en la imagen, se le ordenará al drone realizar movimientos de guiñada hasta que ésta lo esté. En este caso, también implementaremos un controlador PD para evitar un continuo movimiento de guiñada al no conseguir centrar la cara.

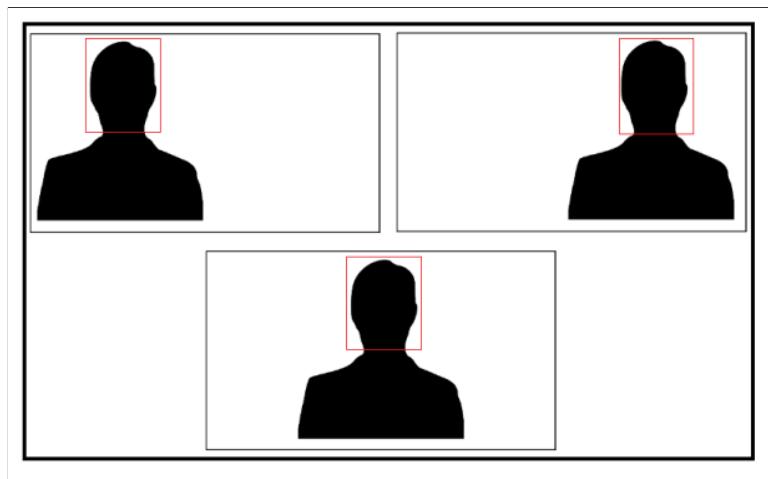


Fig. 2.14 Teoría movimiento guiñada tras reconocimiento facial

Para el correcto funcionamiento del código, va a ser necesario importar las librerías que se vienen utilizando hasta ahora, opencv-python, numpy y djitellopy. Posteriormente, se definirá el ancho y alto de la imagen, que nos servirá para conocer cuando la cara está centrada, el rango de áreas del rectángulo, para que el drone sepa cuándo y cómo moverse dependiendo de la proximidad a la cara, y los valores para los controladores PD, que han sido fijados buscando el mejor rendimiento mediante prueba y error.

Para la detección facial, haremos uso del algoritmo de Viola-Jones, que tiene una probabilidad de detección verdadera del 99,9% y tiene un coste computacional muy bajo. Este método consiste en la división de la imagen en subregiones de tamaños diferentes y gracias a una serie de clasificadores en cascada, determina si esta subregión pertenece a una cara o no. Además, mediante el uso de opencv, se puede implementar a nuestro código fácilmente. Para ello, es necesario importar el código de los controladores en cascada, que se puede encontrar en el Github de opencv (ver [8]).

Posteriormente, mediante opencv se marcarán las caras detectadas con rectángulos rojos y su punto central con un punto verde. Pero el tello EDU únicamente se fijará en la cara con mayor área, es decir, la más cercana. En la Fig. 2.15, se puede observar el código de la función que acabamos de describir y en la Fig. 2.16 el resultado de ésta.

```

def findFace(img):
    faceCascade = cv2.CascadeClassifier("Resources/haarcascade_frontalface_default.xml")
    imgGray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = faceCascade.detectMultiScale(imgGray, 1.2, 8) #Detecta objetos y los retorna como
    una lista de rectángulos.

    myFaceListC = [] #Creación lista de centros de caras
    myFaceListArea = [] #Creación lista de áreas

    for (x, y, w, h) in faces:
        cv2.rectangle(img, (x, y), (x+w, y+h), (0, 0, 255), 2) #Rectángulo rojo que marque la cara
        cx = x + w/2
        cy = y + h/2
        area = w*h
        cv2.circle(img, (cx, cy), 5, (0, 255, 0), cv2.FILLED) #Círculo verde que marque el centro de la
    cara
        myFaceListC.append([cx, cy])
        myFaceListArea.append(area)
    if len(myFaceListArea) != 0:
        i = myFaceListArea.index(max(myFaceListArea)) #Únicamente queremos la cara más
    cercana
        return img, [myFaceListC[i], myFaceListArea[i]]
    else:
        return img, [[0, 0], 0]

```

Fig. 2.15 Código función de detección facial

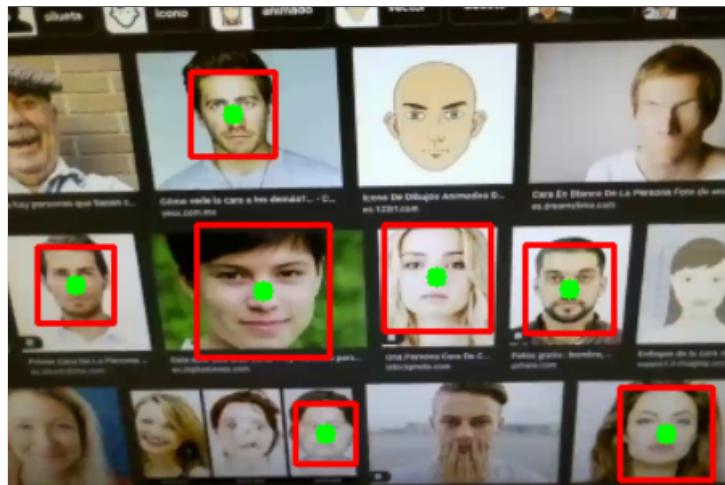


Fig. 2.16 Resultado código detección facial

Seguidamente, se programará la función para el seguimiento de la cara por parte del drone (Fig. 2.17). Para el correcto funcionamiento de ésta, es necesario pasarle ciertos inputs: los referentes al drone para que se puedan enviar comandas, la información de área y centro de ésta, la anchura de nuestra imagen para saber si la cara está centrada, los parámetros de los controladores PD y los valores de error anteriores.

```

def trackFace(me, info, w, pid, pid2, pError, pErrorFB):
    area = info[1]
    x, y = info[0]
    fb = 0
    ud = 0

    "Cálculo de la yaw speed dependiendo del error de centrado actual y anterior"
    error = x - w/2 #Que lejos esta el centro de la cara del centro de la imagen
    speed = pid[0] * error + pid[1]*(error - pError)
    speed = int(np.clip(speed, -50, 50)) #min. fijado a -50 y max. a 50

    "Cálculo de la fb speed dependiendo del error de area actual y anterior"
    if area < fbRange[0]:
        errorFB = fbRange[0] - area
    elif area > fbRange[1]:
        errorFB = fbRange[1] - area
    else:
        errorFB = 0

    speedFB = pid2[0] * errorFB + pid2[1]*(errorFB - pErrorFB)
    speedFB = int(np.clip(speedFB, -50, 50))

    if x == 0: #Si no hay cara detectada
        speed = 0
        error = 0
        speedFB = 0

    me.send_rc_control(0, speedFB, 0, speed)
    return error, errorFB

```

Fig. 2.17 Código función de seguimiento

Finalmente, llamaremos en bucle a las funciones necesarias para el correcto funcionamiento de la aplicación. El código completo se encuentra en GitHub [1] y la demostración visual se puede apreciar a partir del minuto 0:22 en [2].

Como conclusión, me gustaría comentar que gracias a esta aplicación he conocido el algoritmo de Viola-Jones y su funcionamiento, así como el método de procesar imágenes mediante opencv. También he podido comprobar experimentalmente la necesidad de aplicar controladores PD, que había estudiado durante la asignatura de Control y Guiaje. Sin ellos el drone entraba en un bucle de inestabilidad y el objetivo de la aplicación era inalcanzable.

Por otro lado, he podido ver la programación que hay detrás de los modos de vuelo de seguimiento de los drones profesionales. Estos modos se basan en reconocer a personas y seguirlos, es decir, la misma dinámica que la aplicación trabajada.

2.2.6. Control del drone mediante las manos

En este apartado se va a desarrollar la detección de manos y sus diferentes dedos para posteriormente controlar el drone mediante ello. El modo de detectar manos y dedos está basado en uno de los cursos (ver [5]), y a partir

de ello, vamos a desarrollar un método propio para poder controlar el drone con gestos manuales.

Esta aplicación se basará en la librería mediapipe (página web de mediapipe [9]), que ofrece diferentes posibilidades de procesado de imagen que se pueden implementar a nuestro drone.

Esta librería tiene la función de poder detectar 21 “landmarks” de una mano, como se puede observar en la Fig. 2.18. Con esta información, se van a ordenar diferentes comandos a nuestra aeronave.

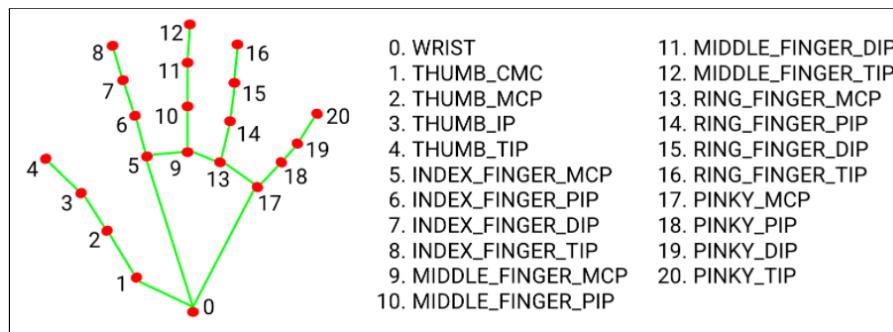


Fig. 2.18 Landmarks detectables en la mano

Primero de todo, se va a codificar un módulo con las diferentes funciones a utilizar, así como se hizo para el control mediante teclado del drone. De esta manera, siempre que se quiera incorporar esta aplicación, únicamente habrá que importar este módulo y llamar a sus funciones.

En este módulo, se necesitará importar las librerías de mediapipe, opencv-python y math y se definirán los parámetros necesarios para la correcta detección: modo de imagen estática, máximo número de manos detectables y umbrales de detección y seguimiento. Además, añadiremos algunos atributos que serán necesarios en las funciones que desarrollaremos más adelante.

El modo de imagen estática se basa en la constante detección de manos utilizando el umbral de confianza de detección. Es decir, si detecta una mano con una seguridad del 60% y nuestro umbral indicado es 50%, se ha encontrado una mano. Por otro lado, si no hacemos uso de este modo, una vez se detecte una mano, se dejará de procesar la imagen para buscar más (en el caso que ya se hayan detectado el máximo de manos detectables indicado) y se pasará al seguimiento de dicha mano usando el umbral de “trackeo”.

En este segundo caso, se ahorra una gran cantidad de cálculo computacional al ya no tener que seguir detectando manos, por lo que el código es más eficiente. Cabe destacar que si el parámetro de “trackeo” en algún momento es inferior al umbral, será necesario volver a detectar la mano. En la Fig. 2.19 se puede observar la definición de todos estos parámetros mencionados.

```

def __init__(self, mode=False, maxHands=2, detectionCon=0.5, minTrackCon=0.5):
    """
    mode:
        Static Image Mode = False -> Si ya ha detectado, pasa a trackear y deja de detectar hasta
        que mintrackCon sea menor al especificado.
        Static Image Mode = True -> Nunca deja de detectar.
    maxHands: Máx. número de manos a detectar
    detectionCon: Valor mínimo de confianza de detección.
    minTrackCon: Valor mínimo de confianza de trackeo
    """
    self.mode = mode
    self.maxHands = maxHands
    self.detectionCon = detectionCon
    self.minTrackCon = minTrackCon

    self.mpHands = mp.solutions.hands
    self.hands = self.mpHands.Hands(self.mode, self.maxHands,
                                   self.detectionCon, self.minTrackCon)

    self.mpDraw = mp.solutions.drawing_utils
    self.tipIds = [4, 8, 12, 16, 20] #Landmarks de los extremos de cada dedo (FingersUp)
    self.fingers = []
    self.lmList = []

```

Fig. 2.19 Código definición atributos

A continuación, vamos a pasar a la función de detección de manos. Para empezar, es importante saber que la función de detección de manos de mediapipe funciona en modo de color BGR y nos retorna el tipo de mano (derecha o izquierda) y cada uno de los landmarks detectados indicando su id (como en la Fig. 2.18) y un ratio de posición “x” e “y” sobre la imagen.

En esta función se va a crear una lista de todas las manos detectadas, donde cada una de ellas poseerá una lista de todos los landmarks, información del rectángulo que rodeará la mano, las coordenadas del centro de la mano y la tipología de ésta. Además se va a poder decidir si la información de landmarks, sus conexiones y el rectángulo alrededor de la mano se quieren incorporar a la imagen. Para finalizar, también hay añadido un flag que nos ayudará a contrarrestar la captura de imagen en modo espejo de nuestra cámara. Sin este flag y con nuestra cámara en modo espejo, se erraría a la hora de definir el tipo de mano (izquierda o derecha).

En las Figuras 2.20 y 2.21 se podrá observar el código de la función y el resultado de ésta.

```

def findHands(self, img, draw=True, flipType=True):
    "img: Imagen donde buscar las manos, solo funciona en modo BGR"
    "draw: ¿Queremos que se dibuje la detección de manos en la imagen?"
    "flipType: Si nuestra cámara captura en modo espejo o no"
    imgRGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB) #Conversión a BGR
    self.results = self.hands.process(imgRGB) #Procesado de imagen para encontrar manos
    allHands = []
    h, w, c = img.shape
    if self.results.multi_hand_landmarks: #¿Se ha detectado alguna landmark?
        for handType,handLms in zip(self.results.multi_handedness, self.results.
        multi_hand_landmarks):
            myHand, mylmList, xList, yList = {}, [], [], []
            for id, lm in enumerate(handLms.landmark): #Obtenemos el id de cada lm y su posición
                px, py = int(lm.x * w), int(lm.y * h) #Pasamos de % de imagen a pixeles
                mylmList.append([px, py])
                xList.append(px)
                yList.append(py)

            # Rectángulo alrededor de la mano
            xmin, xmax = min(xList), max(xList)
            ymin, ymax = min(yList), max(yList)
            boxW, boxH = xmax - xmin, ymax - ymin
            bbox = xmin, ymin, boxW, boxH
            cx, cy = bbox[0] + (bbox[2] // 2), \
                      bbox[1] + (bbox[3] // 2)

            myHand["lmList"] = mylmList
            myHand["bbox"] = bbox
            myHand["center"] = (cx, cy)

            if flipType:
                if handType.classification[0].label == "Right":
                    myHand["type"] = "Left"
                else:
                    myHand["type"] = "Right"
            else: myHand["type"] = handType.classification[0].label
            allHands.append(myHand)

    ## draw
    if draw:
        self.mpDraw.draw_landmarks(img, handLms, self.mpHands.HAND_CONNECTIONS)
        cv2.rectangle(img, (bbox[0] - 20, bbox[1] - 20),
                     (bbox[0] + bbox[2] + 20, bbox[1] + bbox[3] + 20),
                     (255, 0, 255), 2) #Draw del rectángulo
        cv2.putText(img, myHand["type"], (bbox[0] - 30, bbox[1] - 30),cv2.
        FONT_HERSHEY_PLAIN,
                    2, (255, 0, 255), 2) #Marcar derecha o izquierda de la mano
    if draw:
        return allHands, img
    else:
        return allHands

```

Fig. 2.20 Código función detección manos

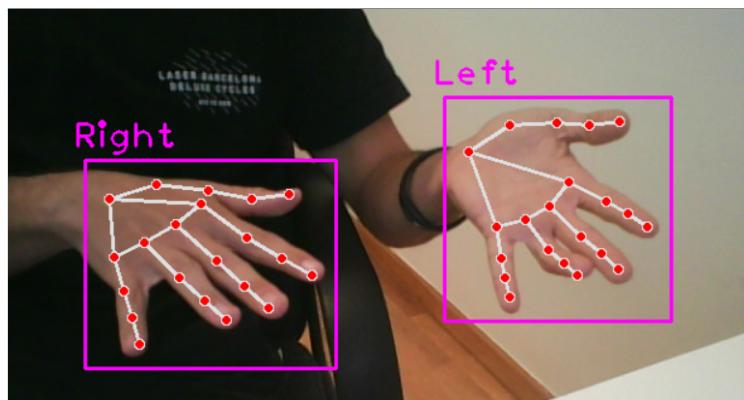


Fig. 2.21 Resultado función detección manos

Una vez ya se han detectado correctamente las manos, se debe desarrollar una función capaz de contar cuántos dedos tenemos levantados. Esta función, diferenciará entre el dedo pulgar y los demás, y nos retornará una lista con los dedos que estén abiertos.

Para saber si un dedo está levantado, hay que fijarse en los landmarks de los extremos de los dedos (ids 4, 8, 12, 16 y 20), como ya se había visto en la definición de atributos de la Fig. 2.19. En el caso del dedo pulgar, al cerrarse hacia el centro de la mano, se comparará la posición “x” del landmark 4 con la posición “x” del landmark 3. Para los demás dedos se seguirá el mismo procedimiento pero comparando la coordenada “y”, ya que estos dedos se cierran hacia abajo. En la Fig. 2.22 se puede observar lo descrito anteriormente y en la Fig. 2.23 se encuentra el código.

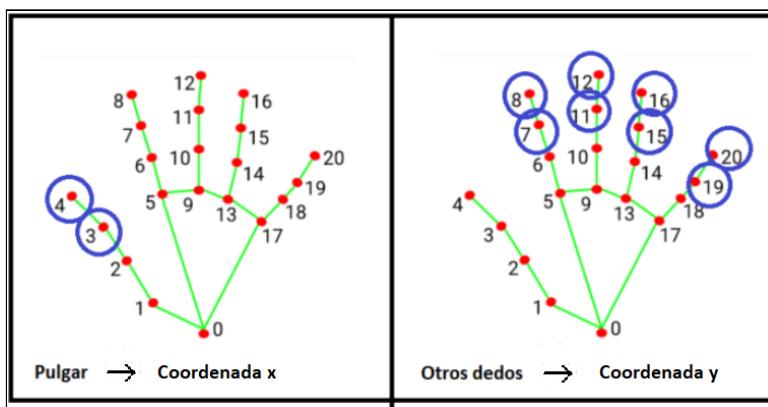


Fig. 2.22 Teoría para contar dedos levantados

```

def fingersUp(self, myHand):
    "Recordar: self.tipIds = [4, 8, 12, 16, 20]"
    myHandType = myHand["type"]
    myLmList = myHand["lmList"]
    if self.results.multi_hand_landmarks:
        fingers = []
        # Pulgar
        if myHandType == "Right":
            if myLmList[self.tipIds[0]][0] > myLmList[self.tipIds[0] - 1][0]:
                fingers.append(1)
            else:
                fingers.append(0)
        else:
            if myLmList[self.tipIds[0]][0] < myLmList[self.tipIds[0] - 1][0]:
                fingers.append(1)
            else:
                fingers.append(0)

        # Otros 4 dedos
        for id in range(1, 5):
            if myLmList[self.tipIds[id]][1] < myLmList[self.tipIds[id] - 2][1]:
                fingers.append(1)
            else:
                fingers.append(0)
    return fingers

```

Fig. 2.23 Código para contar dedos levantados

Por último, se va a crear una función que mida la distancia entre dos landmarks indicados. Para ello, se calculará la diferencia entre las coordenadas “x” e “y” de cada punto, y haciendo la hipotenusa mediante la función hypot de la librería math, se obtendrá la distancia. Finalmente, si se desea, se incorporarán marcas visuales en los landmarks seleccionados y se dibujará la línea donde se ha calculado la distancia, junto con su punto medio. En la Fig. 2.24 se puede ver el resultado visual de obtener la distancia entre dos índices y en la Fig. 2.25 el código.

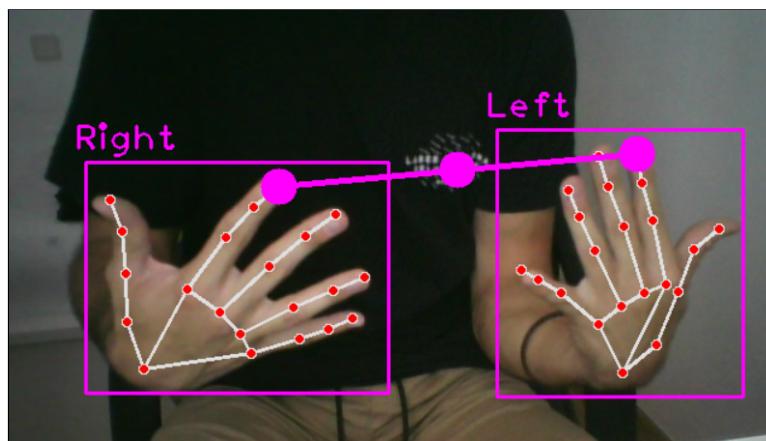


Fig. 2.24 Visual del cálculo de distancia

```
def findDistance(self, p1, p2, img=None):
    x1, y1 = p1
    x2, y2 = p2
    cx, cy = (x1 + x2) // 2, (y1 + y2) // 2
    length = math.hypot(x2 - x1, y2 - y1) #Hipotenusa
    info = (x1, y1, x2, y2, cx, cy)
    "Draw?"
    if img is not None:
        cv2.circle(img, (x1, y1), 15, (255, 0, 255), cv2.FILLED)
        cv2.circle(img, (x2, y2), 15, (255, 0, 255), cv2.FILLED)
        cv2.line(img, (x1, y1), (x2, y2), (255, 0, 255), 3)
        cv2.circle(img, (cx, cy), 15, (255, 0, 255), cv2.FILLED)
        return length, info, img
    else:
        return length, info
```

Fig. 2.25 Código para cálculo de distancia

Una vez ya creado el módulo siguiendo el tutorial, se va a hacer uso de éste para controlar el drone. El objetivo será poder indicar a la aeronave mediante las manos si acercarse o alejarse, elevarse o descender, moverse a derecha o izquierda, aterrizar o cuándo tomar una foto. De esta manera, por ejemplo, nos podremos tomar una foto ajustando la posición del drone como deseemos. En la Tabla 2.3 se van a describir los gestos manuales necesarios para controlar el cuadricóptero.

Tabla 2.3. Gestos manuales para controlar el drone

Gesto	Descripción	Comanda
Manos abiertas acercándose	El drone se acercará a nosotros	me.send_rc_control (0, speed, 0, 0)
Manos abiertas separándose	El drone se alejará de nosotros	me.send_rc_control (0, -speed, 0, 0)
Manos abiertas moviéndose hacia la izquierda	El drone se moverá hacia nuestra izquierda	me.send_rc_control (speed, 0, 0, 0)
Manos abiertas moviéndose hacia la derecha	El drone se moverá hacia nuestra derecha	me.send_rc_control (-speed, 0, 0, 0)
Únicamente dedos índices abiertos de las dos manos	El drone ascenderá	me.send_rc_control (0, 0, speed, 0)
Dedos índice y corazón abiertos de las dos manos	El drone descenderá	me.send_rc_control (0, 0, -speed, 0)
Dedos pulgar y meñique abiertos de las dos manos y juntando los pulgares	El drone tomará una foto a los 5 segundos	cv2.imwrite(f'Resources/Images/{time.time()}.jpg', img)
Alguna mano con los dedos índice, corazón y anular abiertos	El drone aterrizará	me.land()

En cuanto al código, se utilizarán las funciones creadas en el módulo para conocer en todo momento las manos que estén levantadas con sus respectivos dedos. Igualmente, será importante conocer la distancia entre las manos y sus ubicaciones para poder recrear los movimientos que se sugieren en la Tabla 2.3.

En la Fig. 2.26 se muestra el código completo con comentarios que explican su funcionamiento. También, éste se encuentra disponible en el GitHub [1] y su demostración visual se encuentra a partir del minuto 0:45 del video de Youtube [2].

```

import cv2
from HandsTrackingModule import HandDetector
from djitellopy import tello
import time

"Parámetros"
w, h = 720, 480
lengths = []
centers = []
speed = 40
tiempoFoto = 0.0
Stop = False #Flag para detener el drone

"Conexión al Tello"
me = tello.Tello()
me.connect()
print(me.get_battery())
me.streamoff()
me.streamon()
me.takeoff()

detector = HandDetector(mode=False, minTrackCon=0.8, detectionCon=0.8, maxHands=2)

while True:
    img = me.get_frame_read().frame
    hands, img = detector.findHands(img) #Encontrar manos

    if hands and not Stop:
        #Hand 1
        hand1 = hands[0]
        lmList1 = hand1["lmList"] #List of 21 Landmark points
        centerPoint1 = hand1["center"] #Center of the hand cx, cy

        fingers1 = detector.fingersUp(hand1) #Dedos levantados
        "Aterrizar"
        if fingers1 == [0, 1, 1, 1, 0]:
            me.land()
        else:
            pass

        if len(hands) == 2:
            # Hand 2
            hand2 = hands[1]
            lmList2 = hand2["lmList"]
            centerPoint2 = hand2["center"]

            fingers2 = detector.fingersUp(hand2)

            length, info = detector.findDistance(centerPoint1, centerPoint2) #Para acercar/alejar/
            derecha/izquierda
            lengthPhoto, inf = detector.findDistance(lmList1[4], lmList2[4]) #Para tomar foto

            lengths.append(length)
            centers.append(info[4]) #componente x del centro

        "Movimiento delante/detrás y derecha/izquierda"
        if len(lengths) > 1 and fingers1 == [1, 1, 1, 1, 1] and fingers2 == [1, 1, 1, 1, 1]:
            if abs(lengths[len(lengths)-1] - lengths[len(lengths)-2]) >= \
                abs(centers[len(centers) - 1] - centers[len(centers) - 2]): #Solo quiero que haga 1 de
            los dos mov.
                if lengths[len(lengths)-1] < lengths[len(lengths)-2] - 2:

```

```

        me.send_rc_control(0, speed, 0, 0)
    elif lengths[len(lengths)-1] > lengths[len(lengths)-2] + 2:
        me.send_rc_control(0, -speed, 0, 0)
    else:
        me.send_rc_control(0, 0, 0, 0)
    else:
        if centers[len(centers)-1] < centers[len(centers)-2] - 2:
            me.send_rc_control(-speed, 0, 0, 0)
        elif centers[len(centers)-1] > centers[len(centers)-2] + 2:
            me.send_rc_control(speed, 0, 0, 0)
        else:
            me.send_rc_control(0, 0, 0, 0)
    else:
        pass

    "Movimiento arriba/abajo"
    if fingers1 == [0, 1, 0, 0, 0] and fingers2 == [0, 1, 0, 0, 0]:
        me.send_rc_control(0, 0, speed, 0)
    elif fingers1 == [0, 1, 1, 0, 0] and fingers2 == [0, 1, 1, 0, 0]:
        me.send_rc_control(0, 0, -speed, 0)
    else:
        pass

    "Tomar foto a los 5 segundos"
    if lengthPhoto < 20 and fingers1 == [1, 0, 0, 0, 1] and fingers2 == [1, 0, 0, 0, 1]:
        Stop = True
        tiempoFoto = int(time.time()) #momento del gesto
    else:
        pass
    else:
        me.send_rc_control(0, 0, 0, 0)
    else:
        me.send_rc_control(0, 0, 0, 0)

    if int(time.time()) == tiempoFoto + 5:
        cv2.imwrite(f'Resources/Images/{time.time()}.jpg', img)
        time.sleep(1)
        Stop = False

    else:
        pass

    img = cv2.resize(img, (w, h))
    cv2.imshow("Image", img)
    cv2.waitKey(1)

```

Fig. 2.26 Código control mediante gestos manuales

Para terminar, me gustaría exponer que esta aplicación me ha servido para poder crear mi primera funcionalidad propia a partir de todo lo aprendido durante este capítulo. A partir de otros TFGs y TFMps proporcionados por el profesor, había podido ver cómo otros alumnos aplicaban movimiento a drones a partir de posturas corporales, por lo que me pareció interesante usar una dinámica parecida para gestos manuales e interacciones entre las manos.

Además, a parte de todo el código detrás de la detección de manos y dedos, he podido aplicar todo lo referente al movimiento del drone, procesado de imágenes y toma de fotos aprendido en los apartados anteriores.

CAPÍTULO 3. MODO ENJAMBRE

Mediante el Tello EDU también existe la posibilidad de trabajar en modo “Swarm”, pudiendo realizar múltiples funcionalidades con un número ilimitado de drones. Para ello, se necesitan realizar unas configuraciones previas en cada una de las aeronaves y conocer el modo de enviar comandas al enjambre o a drones específicos de éste.

En este capítulo, se va a enseñar de manera detallada cómo poder trabajar con enjambres formados por Tello EDUs, y además, se describirá cómo hacer diferentes movimientos coordinados entre drones.

En mi caso, por cuestiones logísticas, el enjambre estará formado por dos aeronaves, es decir, el número mínimo para ser considerado enjambre, pero aún así, todas las funciones se pueden extraer a enjambres con mayor número de dispositivos.

Por otro lado, por el momento no es posible obtener la imagen capturada por los drones en el modo “Swarm”. La única solución posible es obtener un adaptador para poder conectar dos conexiones Wi-Fi al ordenador al mismo tiempo. Por falta de éste, el capítulo se centrará en el propio movimiento del enjambre y las interacciones entre diferentes drones.

3.1. Configuración del escenario

Anteriormente, como ya se ha visto, para realizar la conexión entre el ordenador y el drone es necesario conectarse al Wi-Fi de éste. Este modo es llamado “Station Mode” y consiste en que el drone actúa como emisor de la red Wi-Fi. En el caso de múltiples cuadricópteros, este proceso es inviable ya que nuestro ordenador únicamente se puede conectar a la red Wi-Fi de uno de ellos.

Por esta razón, el escenario cambia (ver Fig. 3.1) y va a ser necesario el uso de un router. Este modo es el “Access Point Mode” y consiste en conectar, tanto el PC, como las aeronaves, a un router que será el encargado de transmitir las comandas deseadas a todo el enjambre.

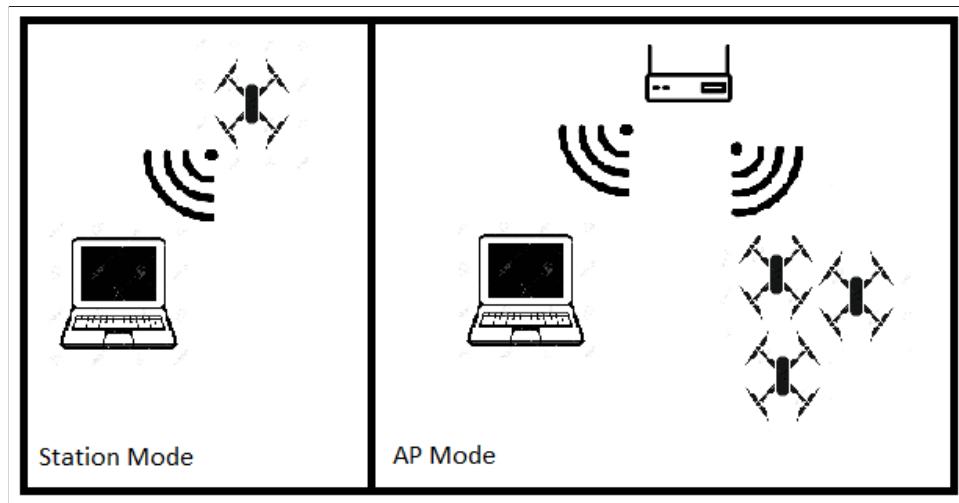


Fig. 3.1 Escenario individual y escenario Swarm

Para la correcta configuración del nuevo escenario se debe ajustar cada drone en “AP Mode”. Para ello, es necesario conectarse individualmente a cada uno de ellos y realizar la comanda de la librería dgitellopy connect_to_wifi(), donde se le indicará el nombre de la red Wi-Fi y su contraseña (ver Fig. 3.2).

```
from dgitellopy import tello
from dgitellopy import TelloSwarm

me = tello.Tello()
me.connect()
print(me.get_battery())

ssid = "MIWIFI_dd6r"
password = "XXXXXXXXXX"

me.connect_to_wifi(ssid, password)
```

Fig. 3.2 Código conexión a red Wi-fi

Cabe destacar que una vez realizada la comanda, el drone se reiniciará automáticamente y pasará a estar en “AP Mode”. Para devolver el drone a “Station Mode”, basta con encender el drone y seguidamente mantener el botón de apagado durante unos 10 segundos.

A continuación, es necesario conocer la IP que el router ha asociado a cada uno de los drones. En mi caso, me conecté a la configuración de mi router y revisando su DHCP pude obtener las IPs asignadas. Seguidamente, ya es posible realizar la conexión a todo el enjambre (ver Fig. 3.3) y poder enviar comandas a cada uno de ellos a la vez.

```

from djitellopy import TelloSwarm

swarm = TelloSwarm.fromIps([
    "████████.1.151",
    "████████.1.152",
])

swarm.connect()
swarm.takeoff()

# run in parallel on all tellos
swarm.move_up(100)

swarm.land()
swarm.end()

```

Fig. 3.3 Código conexión al enjambre

Por último, me gustaría comentar que en caso de tener la posibilidad de conectar el PC a dos redes Wi-Fi, el método para poder obtener la imagen captada por alguno de ellos es hacer una combinación entre el AP mode y el Station mode. Es decir, todos los drones del enjambre deberán estar en modo Access Point excepto uno, que se encontrará en Station mode.

Después, el ordenador se conectará al router para poder tener el control del enjambre, y al drone individual en Station Mode, para obtener su imagen.

La solución encontrada no es del todo eficiente ya que posteriormente, al querer enviar comandas a todo el enjambre, habrá que tener en cuenta que el Tello EDU en Station mode no las va a recibir, por lo que habrá que especificar una comanda únicamente para éste.

3.2. Control desde teclado del enjambre

Así como se desarrolló en [2.2.3. Control desde teclado y toma de imágenes](#) para un solo Tello, en este apartado se van a describir las comandas necesarias para poder controlar todo un enjambre a partir del teclado. La dinámica será la misma, aunque en este caso la comanda send_rc_control() se deberá enviar a todo el enjambre.

Para empezar, se va a importar el módulo KeyPressModule.py que se había desarrollado anteriormente y se puede observar en la Fig. 2.5. Después, es necesario crear un nuevo archivo .py donde importar este módulo y las librerías de TelloSwarm y Time. Posteriormente, se va a definir el enjambre mediante las IPs que el router ofrezca a cada drone, como ya se ha explicado en el apartado anterior, y se va a iniciar el módulo y la conexión al enjambre. Como siempre, se va a revisar la batería de cada una de las aeronaves, y por último, se utilizará el mismo proceso que en el apartado 2.2.3 para controlar un único

drone, pero en este caso, enviando la comanda de movimiento a todo el enjambre mediante: swarm.send_rc_control(). En la Fig. 3.4 se puede observar el código completo descrito.

```
from dgitellopy import TelloSwarm
import KeyPressModule as kp
import time

swarm = TelloSwarm.fromIps([
    "192.168.1.54",
    "192.168.1.57",
])
kp.init()
swarm.connect()
for tello in swarm:
    print(tello.get_battery())

def getKeyboardInput():
    lr, fb, ud, vv = 0, 0, 0, 0
    speed = 50

    if kp.getKey("LEFT"): lr = -speed
    elif kp.getKey("RIGHT"): lr = speed

    if kp.getKey("UP"): fb = speed
    elif kp.getKey("DOWN"): fb = -speed

    if kp.getKey("w"): ud = speed
    elif kp.getKey("s"): ud = -speed

    if kp.getKey("a"): vv = -speed
    elif kp.getKey("d"): vv = speed

    if kp.getKey("q"): swarm.land(); time.sleep(3)
    if kp.getKey("e"): swarm.takeoff()

    return [lr, fb, ud, vv]

while True:
    vals = getKeyboardInput()
    swarm.send_rc_control(vals[0], vals[1], vals[2], vals[3])
    time.sleep(0.001)
```

Fig. 3.4 Código control del enjambre

Me gustaría destacar que a causa de la poca precisión de Tello EDU, los movimientos indicados por teclado a todo el enjambre no serán ejecutados de la misma manera. Por esa razón, se puede concluir en que se pueden pilotar varios drones al mismo tiempo pero con una precisión muy mala, llegando al punto de poder ocasionar colisiones entre ellos.

3.3. Coreografias entre drones

En este apartado, ya que como se ha comentado anteriormente no se puede trabajar sobre la imagen capturada por los drones, se van a desarrollar diferentes movimientos por parte del enjambre de manera coordinada.

Cabe destacar que las “coreografías” están creadas desde cero en torno a un enjambre formado por dos aeronaves. Aunque el enjambre sea con el número

mínimo de drones, visualmente es muy llamativo y la metodología aprendida se puede extrapolar a enjambres mayores.

Antes de empezar, se van a describir los diferentes procedimientos para enviar comandas a todo el enjambre y que se realicen de manera secuencial o paralela, asimismo, la manera de enviar comandas a un único drone del enjambre será explicada.

En el caso de querer enviar comandas que se realicen al mismo tiempo por todo el enjambre, se puede hacer uso de, por ejemplo, `swarm.move_up()`. Por otro lado, si queremos que se realicen a la vez dos comandas distintas, se puede hacer uso de `swarm.parallel(lambda i, tello: tello.send_rc_control(25*(-1)**i), 0, 0, 0))`, donde el primer drone de la lista de IPs ($i=0$), se movería a la derecha haciendo `send_rc_control(25, 0, 0, 0)` y el segundo drone ($i=1$) mediante `send_rc_control(-25, 0, 0, 0)`, a la izquierda al mismo tiempo. Recordar que $**$ en Python implica una exponencial.

Por el contrario, si queremos realizar comandas secuencialmente, es decir, que hasta que no termine la comanda el primer drone, no inicie el segundo, se puede hacer uso de, por ejemplo, `swarm.sequential(lambda i, tello: tello.land())`, donde las aeronaves aterrizarían una a una.

En el caso de únicamente querer enviar comandas a un drone en específico, se hace uso de, por ejemplo, `swarm.tellos[0].takeoff()`, es decir, sólo el primer drone introducido en la lista de IPs despegará.

Una vez ya conocidos los procedimientos para enviar comandas a los drones, se va a proceder a desarrollar diferentes coreografías haciendo uso de ellos.

3.3.1. Cuadrado

Para empezar, se va a programar un movimiento que formará un cuadrado (ver Fig. 3.5). Los movimientos verticales de los drones se van a hacer uno a uno, mientras que los desplazamientos horizontales se realizan al mismo tiempo.

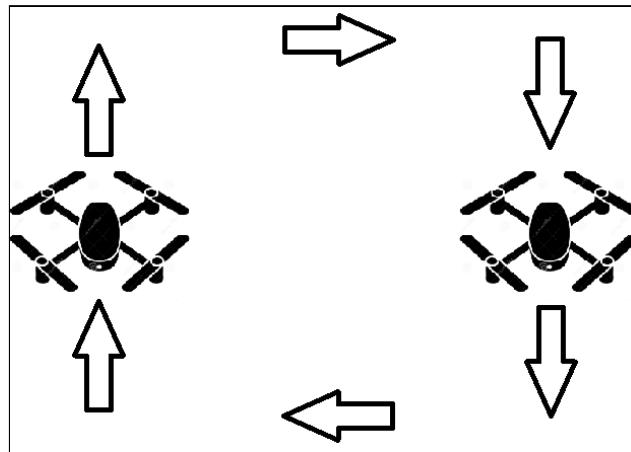


Fig. 3.5 Coreografía en cuadrado

Después de entender el objetivo, se van a programar las comandas pertinentes para poder llegar a ello. Como se puede observar en la Fig. 3.6, se hace uso de las diferentes metodologías mencionadas en la introducción del capítulo.

```

swarm.takeoff()

"Primer drone de la lista de Ips colocado a la derecha"
swarm.tellos[0].send_rc_control(0, 0, -25, 0) #Drone der baja
time.sleep(2)
swarm.tellos[1].send_rc_control(0, 0, 25, 0) #Drone izq sube
time.sleep(2)

swarm.parallel(lambda i, tello: tello.send_rc_control(25*(-1)**i, 0, 0, 0)) #movs laterales
time.sleep(4)

swarm.tellos[0].send_rc_control(0, 0, 25, 0) #Drone der sube
swarm.tellos[1].send_rc_control(0, 0, 0, 0) #Drone izq para
time.sleep(4)

swarm.tellos[0].send_rc_control(0, 0, 0, 0) #Drone der para
swarm.tellos[1].send_rc_control(0, 0, -25, 0) #Drone izq baja
time.sleep(4)

swarm.parallel(lambda i, tello: tello.send_rc_control(-25*(-1)**i, 0, 0, 0)) #movs laterales
time.sleep(4)

swarm.tellos[0].send_rc_control(0, 0, -25, 0) #Drone der baja
swarm.tellos[1].send_rc_control(0, 0, 0, 0) #Drone izq para
time.sleep(2)

swarm.tellos[1].send_rc_control(0, 0, 25, 0) #Drone izq sube
swarm.tellos[0].send_rc_control(0, 0, 0, 0) #Drone der para
time.sleep(2)

swarm.land()

```

Fig. 3.6 Función cuadrado

Cabe destacar que es importante que el primer drone de la lista de IPs (recordemos Fig. 3.3) esté situado a la derecha (si los estamos observando de frente). Además, aconsejo que los drones no estén ubicados en la misma línea horizontal para que no se desestabilicen al pasar uno por encima del otro. Igualmente, los Tello EDU deberán estar separados 1 metro aproximadamente.

A partir del minuto 1:28 en el vídeo (ver [2]) se puede apreciar la demostración visual y en cuanto al código, también estará disponible en el GitHub [1].

3.3.2. Círculo

Para conseguir que los drones formen un círculo de un metro de radio, se va a hacer uso de una nueva comanda que nos proporciona el SDK 2.0. Esta comanda es `curve_xyz_speed(x1, y1, z1, x2, y2, z2, speed)`, y consiste en proporcionar dos puntos y una velocidad al drone para que mediante ello realice una curva.

Partiendo de que el drone siempre empieza en la ubicación (0, 0, 0) y el sistema de referencia es el que se puede observar en la Fig. 3.7, ya que siempre se mantendrá la misma altura (coordenada z=0), podemos completar un círculo mediante dos comandas.

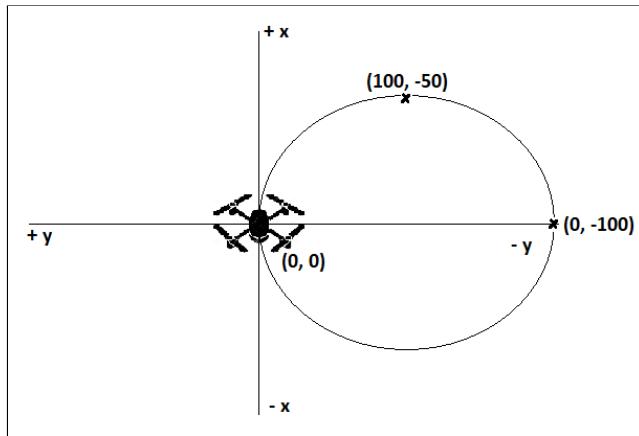


Fig. 3.7 Sistema de referencia para `curve_xyz_speed()`

Así que, utilizando el esquema de la figura anterior, la primera comanda para realizar medio círculo sería `curve_xyz_speed(100, -50, 0, 0, -100, 0, 25)` suponiendo una velocidad de 25 cm/s, y la segunda, `curve_xyz_speed(-100, 50, 0, 0, 100, 0, 25)` para terminar de completar la circunferencia. En el caso del segundo drone, las comandas serían las mismas pero permutando el orden. Entre las dos comandas es aconsejable incorporar un pequeño delay, ya que de no ser así, entran en conflicto ambas. Recordando lo explicado

anteriormente, se va a utilizar el envío de comandas paralelamente como se puede observar en la Fig. 3.8.

```
def Circulo():
    swarm.parallel(lambda i, tello:
        tello.curve_xyz_speed(100*((-1)**i), -50*((-1)**i), 0, 0, -100*((-1)**i), 0, 25))
    time.sleep(0.2)
    swarm.parallel(lambda i, tello:
        tello.curve_xyz_speed(-100 * ((-1) ** i), 50 * ((-1) ** i), 0, 0, 100 * ((-1) ** i), 0, 25))
```

Fig. 3.8 Función círculo

Es importante recordar, que para que el círculo se complete perfectamente, los drones deben encontrarse inicialmente a un metro de distancia. En este caso, la demostración visual aparece a partir del minuto 1:55 del vídeo [2].

3.3.3. Flips

En este apartado, se va a buscar que los drones realicen movimientos paralelos hacia delante y hacia atrás terminando con flips. El SDK 2.0 de DJI nos ofrece la posibilidad de poder realizar flips en diferentes direcciones (flip_forward(), flip_left(...)). Como se puede observar en la Fig. 3.9, no es un código muy complejo, pero el resultado visual, encontrado a partir del minuto 2:22 en [2], si llama la atención.

```
swarm.takeoff()
swarm.move_up(30)

swarm.parallel(lambda i, tello: tello.send_rc_control(0, 25*((-1)**i), 0, 0))
time.sleep(3)
swarm.tellos[0].send_rc_control(0, 0, 0, 0)
swarm.tellos[1].send_rc_control(0, 0, 0, 0)
swarm.tellos[0].flip_back()
swarm.tellos[1].flip_forward()
time.sleep(1)
swarm.parallel(lambda i, tello: tello.send_rc_control(0, -25*((-1)**i), 0, 0))
time.sleep(3)
swarm.tellos[0].send_rc_control(0, 0, 0, 0)
swarm.tellos[1].send_rc_control(0, 0, 0, 0)
swarm.tellos[0].flip_forward()
swarm.tellos[1].flip_back()
time.sleep(1)
swarm.land()
```

Fig. 3.9 Función flips

Hay que recordar que los Tello EDU no poseen una gran batería, por lo que realizar flips supone un gasto de energía extra y el tiempo de vuelo será reducido enormemente.

3.3.4. Modo rebote

Este último modo va a buscar que los drones se ubiquen dentro de un cuadrado imaginario y empiecen a rebotar contra sus paredes y entre ellos mismos. En la Fig. 3.10 se puede ver la dinámica propuesta.

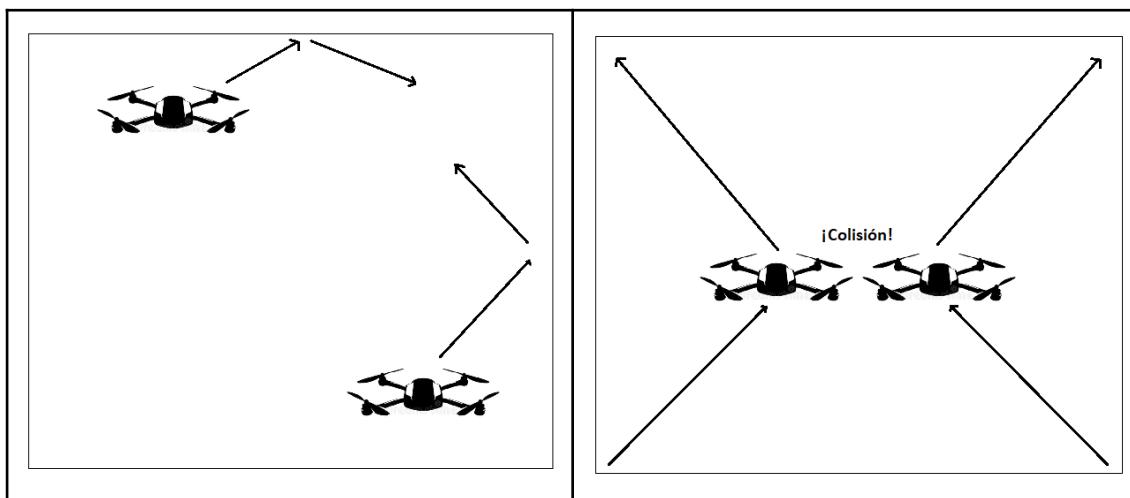


Fig. 3.10 Visualización modo rebote

Para ello, se va a hacer uso de la misma dinámica que se trabajó en el apartado [2.2.4 Mapeo del movimiento](#), ya que es importante saber la posición de los drones en todo momento para así poder aplicar los rebotes pertinentes.

Para empezar, va a ser necesario definir ciertos parámetros como la longitud del lado del cuadrado, las posiciones iniciales de los drones, la velocidad... En la Fig. 3.11, se puede apreciar la definición de todos éstos junto con una breve descripción de ellos.

```
#####
PARAMETERS #####
speed = 15
lado = 150 #Lado en cm del cuadrado
margen = 5 #margen de 5 cm antes de llegar a pared
x, y = margen + 0.5, margen + 0.5 #Posición inicial del drone1 en el cuadrado
x2, y2 = 149.5 - margen, margen + 0.5 #Posición inicial del drone2 en el cuadrado
Distancia = x2 - x
Colision = False
Inicio = True
points = [(x, y)] #Lista de posiciones drone1
points2 = [(x2, y2)] #Lista de posiciones drone2

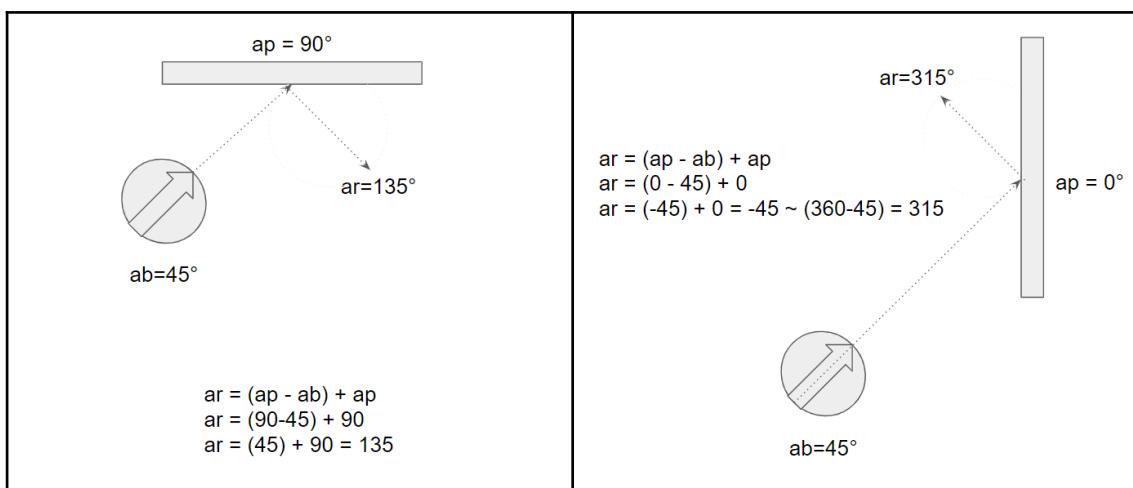
fSpeed = 16.5 #Velocidad lineal real
interval = 0.25 #Intervalo de tiempo

dInterval = fSpeed * interval #Distancia recorrida durante un intervalo

lr, fb = speed, speed
lr2, fb2 = -speed, speed
a = 45 #ángulo de mov. inicial drone1
a2 = 135 #ángulo de mov. inicial drone2
```

Fig. 3.11 Definición parámetros

Una vez definidos los parámetros fundamentales, se va a crear una función encargada de indicar el movimiento correspondiente a los drones. Antes de todo, se deberá comprobar si va a existir una colisión, y en ese caso, se cambiará 180° en la dirección de cada drone. El siguiente paso, será ver si va a haber un choque contra alguna de las paredes del cuadrado. De ser así, se van a aplicar cambios en la dirección del drone para que los rebotes sean reales en torno a ángulos. En la Fig. 3.12, se pueden apreciar dos ejemplos de cómo calcular el ángulo de rebote en las paredes laterales y frontales.

**Fig. 3.12** Teoría de rebotes

Para terminar esta función (ver Fig. 3.13), se deberá crear un sumatorio de la posición mediante trigonometría y se retornarán los valores de este sumatorio junto con los valores de velocidad y ángulo. De este modo, se conocerá la posición, la velocidad y el ángulo de movimiento de los drones del enjambre en todo momento.

```
"""Movimiento"""
def Movimiento(Colision):
    global x, y, lr, fb, lado, a, pared
    d = dInterval
    pared = False
    if Colision:
        "Si hay Colisión, el drone va a girar 180°"
        print("Colision!!")
        lr = -lr
        fb = -fb
        a = a + 180
    else:
        "Si no esta cerca de pared sigue"
        if (0 + margen) < x < (lado - margen) and (0 + margen) < y < (lado - margen):
            pass
        else:
            pared = True
            print("Pared en drone1")
            "Se aplican los angulos de rebote dependiendo de la pared"
            if x <= 0 + margen:
                lr = speed
                a = (90 - a) + 90
            elif x >= lado - margen:
                lr = -speed
                a = (90 - a) + 90
            if y <= 0 + margen:
                fb = speed
                a = (0 - a)
            elif y >= lado - margen:
                fb = -speed
                a = (0 - a)

            # Sumatorio de las posiciones
            x += (d * math.cos(math.radians(a)))
            y += (d * math.sin(math.radians(a)))
            x = round(x, 2) #Redondeamos a 2 decimales
            y = round(y, 2)

    return [lr, fb, x, y, a, pared]
```

Fig. 3.13 Función de movimiento

Seguidamente, se va a añadir una función (ver Fig. 3.14) que calcule la distancia entre las dos aeronaves para poder comprobar si va a suceder una colisión entre ellas. Utilizando la fórmula para calcular la distancia entre dos puntos en un mismo plano (basada en pitágoras), ya que los drones están volando a la misma altitud, es fácil obtenerla.

```
def DistanciaDrones(points, points2):
    "Fórmula distancia entre dos puntos de un plano"
    distancia = math.sqrt((points2[-1][0] - points[-1][0])**2 + (points2[-1][1] - points[-1][1])**2)

    return distancia
```

Fig. 3.14 Distancia entre drones

Para finalizar, es necesario crear un bucle, como en la Fig. 3.15, que vaya comprobando todos los aspectos comentados anteriormente y envíe las comandas correspondientes a los drones en cada caso.

```
while True:
    vals = Movimiento(Colision)
    vals2 = Movimiento2(Colision)

    if vals[5] or vals2[5] or Colision or Inicio:
        swarm.tellos[0].send_rc_control(vals[0], vals[1], 0, 0)
        swarm.tellos[1].send_rc_control(vals2[0], vals2[1], 0, 0)

    else:
        pass

    Inicio = False

    points.append((vals[2], vals[3]))
    points2.append((vals2[2], vals2[3]))
    Distancia = round(DistanciaDrones(points, points2), 2)

    print("Pos drone1: " + str(points[-1]) + " Pos drone2: " + str(points2[-1]) + " Distancia: " + str(Distancia) + " Colision: " + str(Colision) + " RC1: (" + str(vals[0]) + ", " + str(vals[1]) + ")" + " RC2: (" + str(vals2[0]) + ", " + str(vals2[1]) + ")")

    if Distancia <= 30:
        Colision = True
    else:
        Colision = False

    sleep(interval)
```

Fig. 3.15 Detección de colisiones y respectivo movimiento

Este apartado me ha servido para corroborar la poca precisión que hay detrás de los movimientos del Tello EDU, ya que para el correcto funcionamiento de esta coreografía es necesario un conocimiento muy preciso de la posición de los drones en todo momento para que se puedan ejecutar los rebotes adecuadamente.

Además, el error que puede existir de posición es acumulativo así como se está ejecutando la coreografía, por lo que llega un momento en que las posiciones

reales y teóricas del drone disciernen y la detección de una posible colisión es errónea.

Por otro lado, he descubierto que en ocasiones existe una pérdida de paquetes que hace que las comandas no sean enviadas a las aeronaves. Por ejemplo, en algunos casos se detectaba una pared y el Tello rebotaba correctamente, y en otros casos con las mismas condiciones, el drone seguía su curso sin aplicar el rebote pertinente. Pienso que esta situación puede ser debida al sobrecalentamiento de la aeronave.

También, he podido aprender que todos los Tello EDU no vuelan de la misma manera. Es decir, aplicando una comanda a ambos con la misma velocidad teórica, el movimiento real final no era igual en cada aeronave.

Por todas estas razones, como ya he comentado anteriormente es inviable el desarrollo de esta coreografía. Aún así, todo el código trabajado es correcto en cuanto a la teoría detrás del cálculo de la posición, aplicación de los rebotes y detección de colisiones, por lo que está disponible en GitHub [1].

CAPÍTULO 4. ENTORNO TELÉFONO MÓVIL

En este capítulo se va a describir el proceso necesario para crear una aplicación móvil con las funcionalidades desarrolladas en los capítulos anteriores.

Para empezar, se va a realizar un estudio entre los diferentes modos para poder llegar al objetivo de la manera más eficiente. Seguidamente, se explicará paso a paso el proceso para crear y diseñar aplicaciones en Android mediante Python. Por último, se va a indicar de manera muy descriptiva un tutorial para crear una aplicación básica para controlar el Tello EDU, y posteriormente, se le irán añadiendo funcionalidades más complejas.

Me gustaría comentar que algunas de las funcionalidades finales para la aplicación no han podido tener éxito debido a diferentes causas. Aún así, está todo documentado de tal manera que futuros alumnos puedan continuar con mi trabajo y puedan buscar diferentes soluciones para conseguir alcanzar dichas funcionalidades en el entorno móvil.

4.1. Estudio del método de desarrollo

Existen diferentes maneras de poder llegar a desarrollar una aplicación móvil, y entre ellas, me llamaron la atención los frameworks IONIC y Kivy. Por ello, se va a realizar un estudio con sus ventajas y desventajas enfocadas al proyecto para así poder tomar la mejor opción.

4.1.1. IONIC

IONIC framework es un SDK de código abierto creado en 2013 para desarrollar aplicaciones multiplataforma, es decir para IOS, android o entorno web, utilizando un único código. Este framework se basa en tecnología HTML, CSS y JS y actualmente lidera el sector de desarrollo de aplicaciones multiplataforma.

Presenta un gran número de ventajas a parte de las ya mencionadas:

- Framework amigable y fácil de usar.
- Integró con Angular, React y Vue, frameworks donde habitualmente se trabaja para el desarrollo de aplicaciones móviles.
- Bien documentado y apoyado por una comunidad activa.

Por otro lado, también presenta algunas desventajas:

- Peor rendimiento que aplicaciones nativas.

- Limitaciones en aplicaciones que dependan de FPS.
- Poca compatibilidad con Python.

4.1.2. Kivy

Kivy es uno de los frameworks más utilizados en Python para el desarrollo de aplicaciones móviles. Es de código abierto y multiplataforma, al igual que IONIC, y viene con soporte nativo para muchos dispositivos de entrada multi-touch, una creciente biblioteca de widgets y su diseño se basa en la creación personalizada de aplicaciones de manera rápida y sencilla.

Presenta las siguientes ventajas:

- Excelente estabilidad.
- Compatibilidad con Python.
- Gran cantidad de widgets para un amplio abanico de posibilidades en la creación de la app.

En cambio, sus desventajas son:

- Peor rendimiento que aplicaciones nativas.
- Poco amigable.
- Falta de apoyo por la comunidad y escasez de documentación en comparación con otros frameworks.

4.1.3. Elección de framework

Después de haber recopilado información acerca de estos dos posibles frameworks para el desarrollo de la aplicación móvil, se tomó la decisión de usar Kivy.

A pesar de que IONIC es un framework más amigable y con mucha información acerca de su uso, no destaca por su compatibilidad con Python. Además, la app va a depender de los FPS, ya que deseamos incorporar retransmisión de video y la necesidad del procesamiento de éste. Es decir, IONIC tiene limitaciones de rendimiento en este caso.

Dicho de otro modo, básicamente se elige Kivy por delante de IONIC por su compatibilidad con Python, lenguaje que se ha usado durante todo el proyecto, y su mejor rendimiento frente a retransmisiones de vídeo.

4.2. Generación de la aplicación mediante Kivy

Para empezar, se va a indicar los procesos necesarios para poder implementar la aplicación en el teléfono móvil. Cabe destacar que se va a hacer uso de un dispositivo con Android como sistema operativo, ya que el proceso será más fácil por las características de éste.

A través de Kivy, existen dos maneras de poder llevar a cabo este proceso:

4.2.1. Kivy Launcher

Este método es el más simple y rápido a la hora de cargar nuestra aplicación en Android. La aplicación Kivy Launcher es descargable desde GooglePlay y una vez instalada permite lanzar aplicaciones Kivy desde el dispositivo móvil.

Su funcionamiento se basa en adjuntar los archivos (.py, .kv, imágenes, etc) en un lugar específico de la memoria del teléfono sin necesidad de realizar ninguna compilación. La app reconocerá estos archivos y permitirá el uso de la aplicación adjuntada.

Para usar Kivy Launcher se debe:

- Llamar al módulo principal de la app main.py.
- Crear un archivo de texto con la configuración de la app, llamado android.txt y que tiene que contener como mínimo:
 - title = "Nombre de la app"
 - author = "Nombre del autor"
 - orientation = "Orientación de la aplicación en el teléfono" (portrait/landscape)
- Buscar la carpeta /sdcard/kivy/ en la memoria del dispositivo móvil y crear dentro otra carpeta con el nombre de la aplicación. Dentro de esta carpeta se deben pegar todos los archivos necesarios: módulos (.py), archivos de kivy language (.kv), archivos de configuración (.txt), iconos, videos, imágenes...
- Finalmente, al abrir la app Kivy Launcher aparecerá una lista de aplicaciones. Al seleccionar la adjuntada, deberá correr sin problemas si todo el proceso se ha realizado correctamente.

4.2.2. Buildozer

Buildozer es una herramienta que tiene como objetivo empaquetar aplicaciones móviles fácilmente. Automatiza todo el proceso de compilación y descarga los requisitos previos como python-for-android, Android SDK, NDK, etc.

El problema de este método es la imposibilidad de realizarlo mediante Windows. Es necesario ejecutarlo desde un sistema operativo Linux, pero hoy en día, haciendo uso de Google Collab, se puede ejecutar fácilmente sin la necesidad de máquinas virtuales.

En mi caso, he creado un código con los pasos a seguir descritos en Google Collab [10]. Ahí, se deben adjuntar los archivos necesarios para que funcione la aplicación en el repositorio “archivos”, es decir, el mismo proceso que en Kivy Launcher pero sin la necesidad del archivo de texto de configuración (android.txt).

A continuación, se van a ir realizando las comandas ya escritas en el código hasta llegar a “!buildozer init”. Ésta va a crear un archivo llamado buildozer.spec, donde se van a tener que configurar las especificaciones necesarias para que la aplicación funcione correctamente. Por ejemplo, habrá que indicar las librerías utilizadas en la aplicación para que se tengan en cuenta a la hora de la compilación.

Una vez configuradas las especificaciones, ya se podrá iniciar la creación del .apk. Este proceso es bastante duradero, dependiendo de la app puede llegar a tardar más de media hora. Una vez finalizado, en la carpeta bin, ubicada en archivos, se va a encontrar el .apk que deberemos instalar en el Android.

Es muy importante saber que en el caso de querer realizar cambios en la app, basta con actualizar el nuevo main situado en el repositorio de archivos y volver a realizar la última comanda !buildozer -v android debug. Esta segunda generación del .apk actualizado durará segundos, por lo que va a hacer más dinámico el proceso de trabajar con buildozer, ya que de no ser así, se habría que esperar unos 40 minutos entre cada .apk generado debido a actualizaciones en el main.

Posteriormente, se va a realizar un tutorial más visual y explicativo de como realizar todo este proceso. El objetivo será la creación de una aplicación simple para controlar el Tello EDU. ([4.4 Aplicación básica para el control del Tello EDU](#))

4.3. Instalación del entorno de trabajo

Desde mi experiencia, estos dos procesos comentados anteriormente son relativamente complicados, ya que un mínimo error imposibilita la inicialización de la app. Por esa razón, es de vital importancia la correcta configuración de las especificaciones y poder conocer qué errores se están cometiendo para así poder corregirlos.

En mi caso, siempre hice uso de Buildozer, ya que mediante el .spec que se genera, es más intuitivo y fácil poder configurar las apps correctamente. Además, como se ha comentado anteriormente, únicamente el proceso de

generación del primer .apk es muy duradero, por lo que es prácticamente igual de dinámico el trabajo con Buildozer y Kivy Launcher.

Personalmente, hice uso de Android Studio (web oficial en [11]) para poder conocer los errores que estaba cometiendo. Para que este software reconozca el dispositivo móvil, es necesario configurarlo en modo desarrollador y activar el modo de depuración USB. En la siguiente página web (ver [12]) se describe el proceso a seguir. Una vez configurado, al conectar el teléfono móvil por USB al ordenador, Android Studio debería reconocerlo y permitirnos ver el Logcat donde se podrán apreciar los posibles errores cometidos..

Es importante saber que se debe autorizar el Logcat de Python en el archivo de especificaciones que se genera en Buildozer. De no ser así, no aparecería la información que deseamos en el Logcat de Android Studio. Para autorizarlo, basta con descomentar la línea donde aparece: “android.logcat_filters = *:S python:D”.

De ahora en adelante, a través del Logcat (ver Fig. 4.1) se va a poder observar el registro de mensajes de nuestro Android, es decir, se podrá observar si existen errores y en qué consisten.

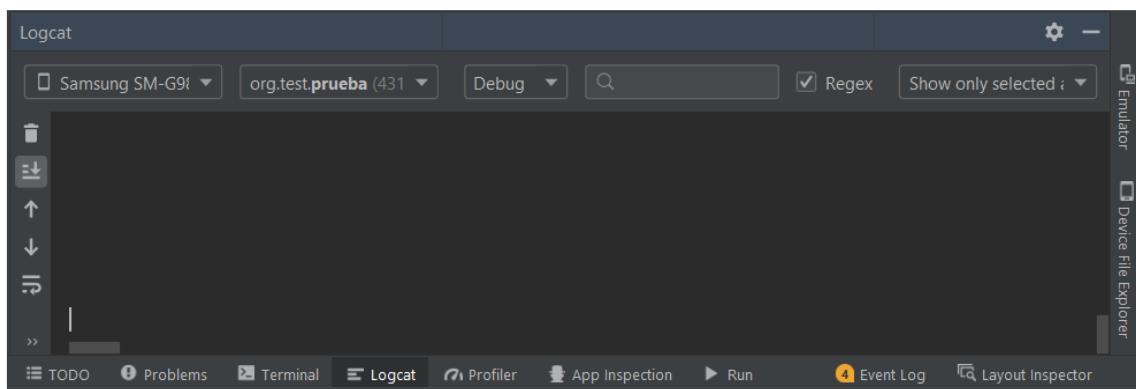


Fig. 4.1 Logcat Android Studio

Desde la experiencia obtenida realizando el trabajo, la mayoría de errores que suelen existir al principio son relacionados a los requisitos de librerías que necesita la aplicación. Por ejemplo, si la app creada necesita numpy, esta librería deberá aparecer en el archivo de especificaciones. En el siguiente apartado, se explicará cómo configurar el archivo de especificaciones correctamente para poder generar una aplicación simple para controlar el Tello EDU.

4.4. Aplicación básica para el control del Tello EDU

Este apartado tiene como objetivo la creación de una aplicación básica para poder controlar el drone desde el teléfono móvil mediante Kivy. Con ella, se va a poder aterrizar y despegar cuando se deseé y realizar todos los movimientos necesarios para pilotar la aeronave correctamente. Además, como siempre, se va a comprobar el nivel de batería del Tello en el inicio de la conexión.

Cabe destacar que este apartado también va a servir de introducción a Kivy, ya que se van a apreciar algunas de sus funcionalidades. Asimismo, se podrá observar de manera más explicativa y visual el desarrollo para generar el .apk mediante Buildozer.

Por último, hay que mencionar que gracias a la compatibilidad de Kivy con Python y su funcionalidad multiplataforma, se podrá seguir haciendo uso de Python y del IDE PyCharm al igual que en el entorno PC. [2.1 Instalación entorno de trabajo](#). De esta manera, se va a desarrollar la app en el entorno PC y posteriormente se instalará en el Android.

4.4.1. Desarrollo del código

Antes de todo, es importante mencionar que Kivy posee un lenguaje de diseño (archivos .kv) similar a HTML o CSS. Éste es el responsable de diseñar y agregar widgets a la pantalla, pero no maneja ninguna lógica. Por eso, en la aplicación se va a hacer uso del main.py, donde se ejecutará toda la lógica de la aplicación, y un my.kv donde se estructurará el diseño de ésta.

Es importante conocer que el nombre del archivo de lenguaje Kivy debe estar todo en minúsculas y coincidir parcialmente con el inicio del nombre de la clase principal donde se ubica el método de construcción. Por ejemplo, la clase principal de la app será llamada MyApp, por lo que el archivo de lenguaje Kivy deberá ser llamado my.kv. Más adelante se podrá entender con más facilidad juntamente con el código.

Hecha la aclaración, para empezar se va a instalar la librería Kivy. Para la creación de cualquier aplicación haciendo uso de un archivo de lenguaje Kivy, va a ser necesario importar de kivy.app “App”, y de kivy.lang “Builder”. A continuación, se va a codificar la clase principal mencionada anteriormente MyApp() y se creará el archivo de lenguaje my.kv en la carpeta del proyecto donde también está ubicado el main.py. El código de este proceso se puede observar en la Fig. 4.2.

```

"Para la creación de la app"
from kivy.app import App
from kivy.lang import Builder

"Archivo de lenguaje Kivy"
kv = Builder.load_file("my.kv")

class MyApp(App):
    def build(self):
        self.Title = "Control Básico Tello EDU"
        return kv

if __name__ == '__main__':
    MyApp().run()

```

Fig. 4.2 Código creación de App en Kivy

El siguiente paso es importar Screen y ScreenManager ya que se va a desarrollar una app con dos ventanas, la de conexión al drone y la de control de la aeronave. El proceso es simple, se debe crear una clase para cada una de las ventanas donde se ubicarán las funciones necesarias de cada una de ellas. También será necesario crear una clase encargada de la interacción entre las dos ventanas haciendo uso de ScreenManager. En la Fig. 4.3 se muestra el proceso descrito anteriormente.

```

"Para la creación de la app"
from kivy.app import App
from kivy.lang import Builder
"Desarrollo de diferentes ventanas"
from kivy.uix.screenmanager import Screen, ScreenManager

"Interacción entre ventanas"
class WindowManager(ScreenManager):
    pass

"Conexión al Tello EDU"
class ConexionWindow(Screen):
    pass

"Control del Tello EDU"
class ControlWindow(Screen):
    pass

"Archivo de lenguaje Kivy"
kv = Builder.load_file("my.kv")

class MyApp(App):
    def build(self):
        self.Title = "Control Básico Tello EDU"
        return kv

if __name__ == '__main__':
    MyApp().run()

```

Fig. 4.3 Código creación de ventanas

Por otro lado, ya se puede empezar a configurar el archivo de lenguaje Kivy. El código estará dividido por ventanas ya que cada una de ellas tendrá un diseño

diferente, pero para empezar, hay que llamar a la clase WindowManager e indicarle las ventanas que tendrá la aplicación (ConexionWindow y ControlWindow). Posteriormente, se creará un espacio para diseñar cada una de las ventanas y se les incorporará un nombre que hará referencia a cada una de ellas dentro del archivo .kv. Es decir, funcionará como un ID dentro del archivo de lenguaje al que hay que llamar cuando nos referimos a una ventana en específico. El código debe construirse en estructura de árbol como se puede apreciar en la Fig. 4.4.

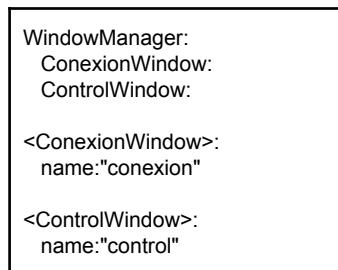


Fig. 4.4 Creación de ventanas en .kv

Una vez estructuradas las ventanas de la aplicación, se va a empezar a desarrollar el diseño de la ConexionWindow. Se trata de un diseño simple basado en un BoxLayout de 2 filas (vertical), compuesto por un Label en la fila superior y un Button en la inferior. El Label hará la función de título, es decir, mostrará el texto: “Conexión Tello EDU” y el Button será el que iniciará la conexión con el drone. Siguiendo la estructura de árbol, el código y diseño de ConexionWindow será así:



Fig. 4.5 Diseño simple ventana conexión

Como se puede observar, no es un diseño muy atractivo, por lo que se va a modificar para dar a la ventana un mejor aspecto.

Para empezar, nos interesa que el título únicamente esté en la parte superior de la ventana, por lo que añadiendo un “size_hint: 1, 0.15” al Label, se le indica que el tamaño que ocupa en la coordenada X es del 100% pero en la Y solo del 15%. Predeterminadamente, como se ha visto en la Fig. 4.5, el Label y el Button ocupan el 50% cada uno dentro del BoxLayout, pero si ahora el Label en la coordenada Y ocupa el 15%, el Button ocupará el otro 85%.

Por otro lado, se va a cambiar el fondo del Label a un gris claro. Para ello se añade un rectángulo antes del texto mediante “canvas.before” con el mismo tamaño que el Label y se le indica el código rgb: 0.7 ,0.7, 0.7. Además, la letra del texto se cambiará a negra mediante “color: 0, 0, 0”, ya que el código rgb del negro es el (0, 0, 0).

El código de la ventana ConexionWindow en el archivo de lenguaje Kivy y el aspecto visual final se podrá observar en la Fig. 4.6.

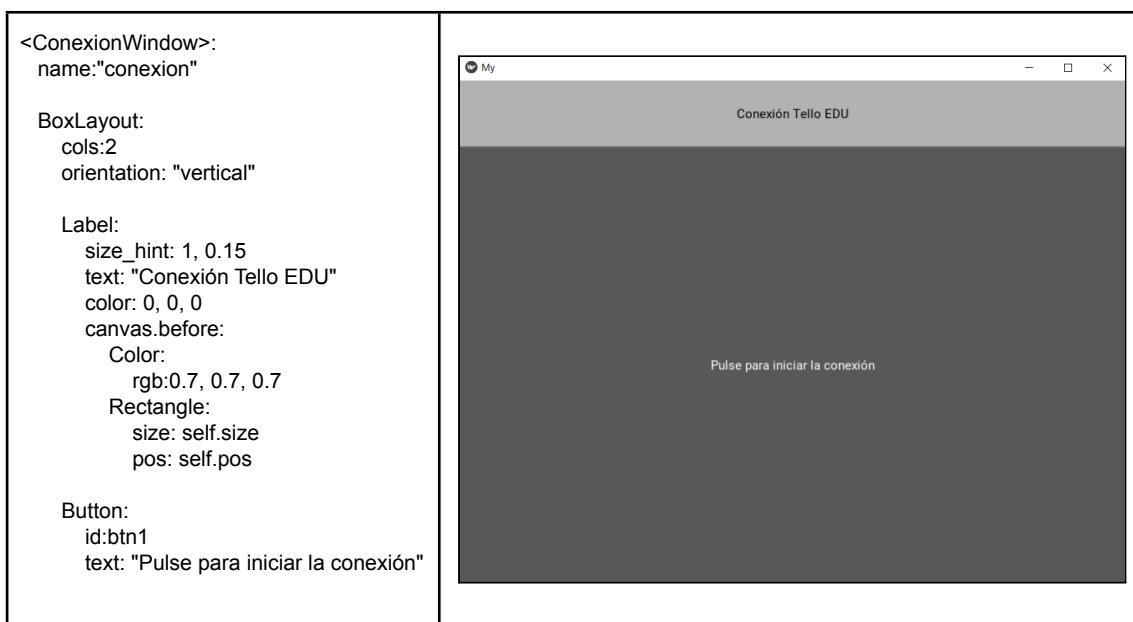


Fig. 4.6 Diseño más atractivo ventana conexión

El próximo paso será incorporar la conexión al drone y la transición a la ventana de control al presionar el botón. Por ello, es necesario importar la librería de djitellopy, pero en este caso, se va a importar de manera diferente a la usada hasta ahora. Para este nuevo modo, es necesario descargar los archivos de Github de la librería (ver [6]) y adjuntarlos al proyecto como se puede ver en la Fig. 4.7.

La razón por la que se va a importar como si fuera un módulo es para evitar problemas de incompatibilidad a la hora de generar el .apk mediante Buildozer. Utilizando el anterior método, Buildozer no reconocía la librería y no podía generar la aplicación. Asimismo, señalar que habrá que adjuntar toda la carpeta referente al módulo tellopy en los archivos de Buildozer también.

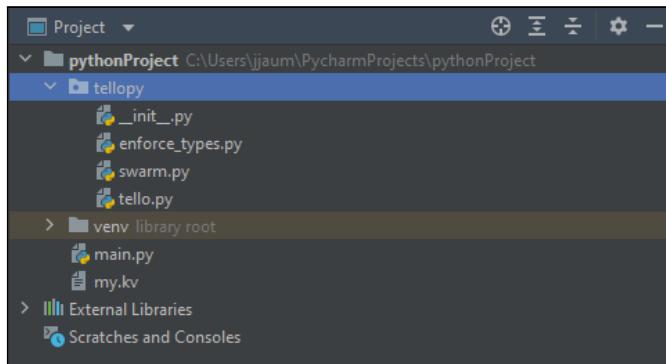


Fig. 4.7 Archivos djitellopy importados

Seguidamente, se va a crear una función llamada pressedConexion() dentro de la clase ConexionWindow para realizar las acciones pertinentes al pulsar el botón de conectar.

Como se ha mencionado anteriormente, es interesante conocer el nivel de batería del drone antes de iniciar un vuelo, por lo que se hará uso del widget PopUp para dar a conocer esta información. El widget PopUp crea una ventana emergente y dentro de esta ventana se va a configurar un BoxLayout compuesto por un Label y un Button. Para ello, será necesario importar todos estos widgets (BoxLayout, Label, PopUp y Button) ya que hasta ahora únicamente se había hecho uso de ellos en el archivo de lenguaje.

El código de la función se basa en la conexión al drone y la creación del PopUp con la información deseada. En Python, no se hace uso de la programación en árbol al igual que en el archivo de lenguaje, pero los widgets se configuran de manera parecida.

Únicamente hay que conocer que para agregar un widget dentro de otro (por ejemplo meter un Label en un BoxLayout) se hace uso de “add_widget()”. También, es interesante comentar que al PopUp se le indica “auto_dismiss = False” para que no se cierre la ventana emergente al pulsar en cualquier lugar fuera de ella, ya que esto debe ocurrir cuando el botón sea pulsado. Por esta razón, se le indica al Button “on_press = pop.dismiss”.

Para terminar con el botón de conexión, falta la configuración del archivo de lenguaje Kivy. Al botón con id:btn1 añadido, habrá que indicarle que al ser pulsado (on_press) llame a la función pressedConexion() ubicada en root (la función está ubicada en la ventana ConexionWindow de main.py y estamos en

la rama de esta misma ventana en my.kv) y cuando ya haya sido pulsado (on_release), se visualice la ventana de control de la aeronave y haga la transición entre ventanas hacia la derecha.

En la Fig. 4.8 se pueden observar los códigos del main.py y del botón de conexión en el .kv, así como el diseño del PopUp.

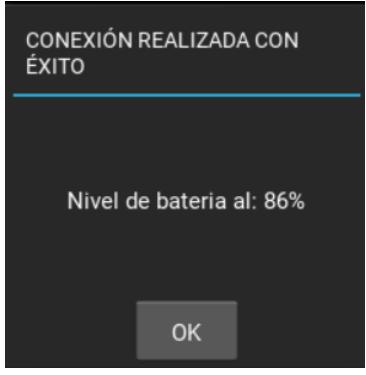
	<pre>Button: id:btn1 text: "Pulse para iniciar la conexión" on_press: root.pressedConexion() on_release: app.root.current = "control" root.manager.transition.direction = "right"</pre>
<pre>"Para la creación de la app" from kivy.app import App from kivy.lang import Builder "Desarrollo de diferentes ventanas" from kivy.uix.screenmanager import Screen, ScreenManager "Función PopUp" from kivy.uix.popup import Popup from kivy.uix.label import Label from kivy.uix.boxlayout import BoxLayout from kivy.uix.button import Button "Para el Tello EDU" from tellopy import tello from time import sleep me = tello.Tello() "Interacciones entre ventanas" class WindowManager(ScreenManager): pass "Conexión al Tello EDU" class ConexionWindow(Screen): def pressedConexion(self): me.connect() content = BoxLayout(orientation="vertical") pop = Popup(title="CONEXIÓN REALIZADA CON ÉXITO", content=content, size_hint=(None, None), size=(250, 250), auto_dismiss=False) texto = Label(text="Nivel de batería al: " + str(me.get_battery()) + "%") boton = Button(text="OK", size_hint=(0.3, 0.3), pos_hint={"center_x": 0.5, "center_y": 0.5}, on_press=pop.dismiss) content.add_widget(texto) #Se añade el Label al BoxLayout content.add_widget(boton) #Se añade el botón al BoxLayout pop.open()</pre>	

Fig. 4.8 Códigos y visualización del botón de conexión

A continuación, se va a desarrollar el diseño para la ventana de control. Primero de todo, se debe añadir un BoxLayout, donde en la parte superior de

éste se ubicarán los botones de salir y aterrizar/despegar. Para aterrizar y despegar se va a hacer uso de un ToggleButton, éste se caracteriza por tener dos estados, normal y down, por lo que se podrá usar para cumplir con las dos funciones. El botón de salir volverá a la ventana de conexión haciendo una transición hacia la izquierda, y por el momento, se va a dejar sin funcionalidad al ToggleButton, ya que únicamente se está realizando el diseño. El código de la ControlWindow y su visualización se pueden apreciar en la Figura 4.9.

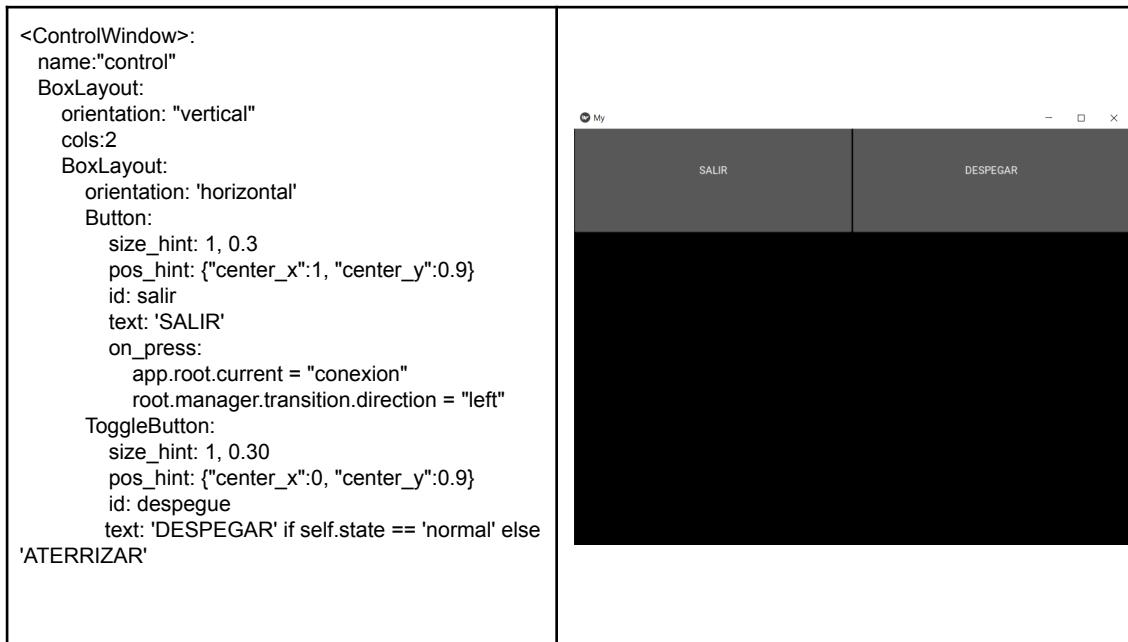


Fig. 4.9 Desarrollo diseño ventana de control

En la fila inferior del BoxLayout debe añadirse otro BoxLayout horizontal de dos columnas, donde se ubicarán los botones para controlar el drone. Estas dos columnas van a ser iguales y consistirán en 4 botones en cada una de ellas.

Para poder ubicar estos 4 botones correctamente en cada columna, se hará uso de otro BoxLayout vertical de 3 columnas, donde en la primera fila estará ubicado un botón de arriba, en la segunda fila un BoxLayout horizontal con 2 botones de izquierda y derecha, y por último, en la tercera fila un botón de abajo. Cabe destacar que habrá que ajustar los centrados y tamaños de los botones y BoxLayouts haciendo uso de “pos_hint” y “size_hint” para que todo quede atractivo a la vista. En las siguientes Figuras 4.9 y 4.10 se encuentra el código y el diseño de la ventana.

```
<ControlWindow>
    name:"control"
    BoxLayout:
        orientation: "vertical"
        cols:2
        BoxLayout:
            orientation: 'horizontal'
            Button:
                size_hint: 1, 0.3
                pos_hint: {"center_x":1, "center_y":0.9}
                id: salir
                text: 'SALIR'
                on_press:
                    app.root.current = "conexion"
                    root.manager.transition.direction = "left"
            ToggleButton:
                size_hint: 1, 0.30
                pos_hint: {"center_x":0, "center_y":0.9}
                id: despegue
                text: 'DESPEGAR' if self.state == 'normal' else 'ATERRIZAR'
        BoxLayout:
            cols:2
            orientation: "horizontal"
            BoxLayout:
                size_hint: 0.4, 0.7
                pos_hint: {"center_x":0, "center_y":0.4}
                orientation: "vertical"
                Button:
                    size_hint: 0.5, 1
                    id: arriba
                    pos_hint: {"x":0.25}
                    text: "^^"
                BoxLayout:
                    Button:
                        id: yawiz
                        text: "<"
                    Button:
                        id: yawder
                        text: ">"
                Button:
                    size_hint: 0.5, 1
                    id: abajo
                    pos_hint: {"x":0.25}
                    text: "v"
            BoxLayout:
                orientation: "vertical"
                size_hint: 0.4, 0.7
                pos_hint: {"center_x":1, "center_y":0.4}
                Button:
                    size_hint: 0.5, 1
                    id: delante
                    pos_hint: {"x":0.25}
                    text: "^^"
                BoxLayout:
                    Button:
                        id: izquierda
                        text: "<"
                    Button:
                        id: derecha
                        text: ">"
                Button:
                    size_hint: 0.5, 1
                    id: atras
                    pos_hint: {"x":0.25}
                    text: "v"
```

Fig. 4.10 Desarrollo diseño ventana de control



Fig. 4.11 Visual ventana de control

Para finalizar, es necesario configurar los botones añadidos en la ventana de control. Para llamar a cada uno de estos botones creados en el archivos de lenguaje, se hace uso de “self.ids.(id del botón)” y para saber si están siendo presionados, es necesario comprobar si su estado es “down”. Es decir, se hará uso de condicionales con la comanda “self.ids.(id del botón).state == “down”.

En el caso de los botones superiores de salir y despegar/aterrizar será sencillo, ya que únicamente habrá que comprobar el estado del botón y realizar las comandas de `takeoff()` y `land()` en cada caso. Las funciones creadas serán `DespegaAterriza()` y `Desconexion()` y deben ser llamadas desde el archivo .kv haciendo uso de `on_state` para el `ToggleButton`, ya que es un botón que depende del estado en que se encuentre, y `on_press` para el botón de salir.

Por otro lado, para los 8 botones de movimiento, se va a crear una función llamada `Movimiento()` basada en la función `getKeyboardInput()` del [2.2.3 Control desde teclado y toma de imágenes](#). Ahí se comprobaba si las teclas del teclado eran pulsadas y dependiendo de ello se daba un valor fijado de “speed” a las diferentes direcciones de velocidad de la aeronave. Siguiendo la misma estructura, se van a crear 4 variables (`lr`, `fb`, `ud`, `yv`) definidas a 0 y dependiendo del estado de cada uno de los botones, se le va a añadir una velocidad determinada a cada una de ellas. Finalmente, se va a realizar la comanda `send_rc_control(lr, fb, ud, yv)`.

Por último, al igual que en los botones superiores, es necesario configurar los botones en el archivo de lenguaje Kivy. En este caso, en cada botón relacionado con el movimiento del drone va a ser necesario añadir `on_press: root.Movimiento()` y `on_release: root.Movimiento()`.

A primera vista, parece redundante, pero es de vital importancia añadir los dos eventos. Mediante el siguiente ejemplo se podrá entender fácilmente. Si se pulsa el botón de adelante, la función Movimiento() lo detectará y la variable fb será igual a speed, por lo que se enviará send_rc_control(0, speed, 0, 0) y el drone avanzará. Si se desea detener este movimiento, se dejará de presionar el botón en la app, pero no hay programado nada que indique al drone que debe detenerse. Por ello, se incorpora el evento on_release, que hace referencia a cuando el botón ya ha sido presionado y ya ha vuelto a estado normal. Así que, cuando se desee dejar de avanzar hacia delante, se dejará de presionar ese botón y el evento on_release llamará de nuevo a la función Movimiento(), que no detectará ningún botón presionado y enviará la comanda send_rc_control(0, 0, 0, 0), por lo que el drone se detendrá.

El código de la clase ControlWindow está ubicado en la Fig. 4.12. Por otro lado, los códigos completos del main.py y del my.kv estarán ubicados en [1].

```
speed = 30

"Control del Tello EDU"
class ControlWindow(Screen):

    def DespegaaTerriza(self):
        if self.ids.despegue.state == "down":
            me.takeoff()
            sleep(2)
        else:
            me.land()
            sleep(2)

    def Desconexion(self):
        me.land()
        sleep(2)
        quit()

    def Movimiento(self):
        lr, fb, ud, yv = 0, 0, 0, 0

        if self.ids.arriba.state == "down":
            ud = speed
        elif self.ids.abajo.state == "down":
            ud = -speed
        if self.ids.yawiz.state == "down":
            yv = -speed
        elif self.ids.yawder.state == "down":
            yv = speed
        if self.ids.delante.state == "down":
            fb = speed
        elif self.ids.atras.state == "down":
            fb = -speed
        if self.ids.izquierda.state == "down":
            lr = -speed
        elif self.ids.derecha.state == "down":
            lr = speed

        me.send_rc_control(lr, fb, ud, yv)
```

Fig. 4.12 Funciones para la ventana de control

4.4.2. Generación de la aplicación móvil con Buildozer

Una vez desarrollado el código de la aplicación y haber comprobado su correcto funcionamiento en el entorno web, el objetivo es la generación del archivo .apk para posteriormente realizar la instalación de la app en el teléfono móvil.

El primer paso es acceder al link de Google Collab [10], que ya se había comentado anteriormente, y adjuntar todos los archivos referentes a la aplicación.

Es decir, se deberán adjuntar el main.py y el my.kv, y además, al no poder importar carpetas directamente, se tendrá que crear una carpeta llamada tellopy, y dentro de ésta, ubicar todos los archivos necesarios. En definitiva, se deben contener los mismos archivos que en la Fig. 4.13.

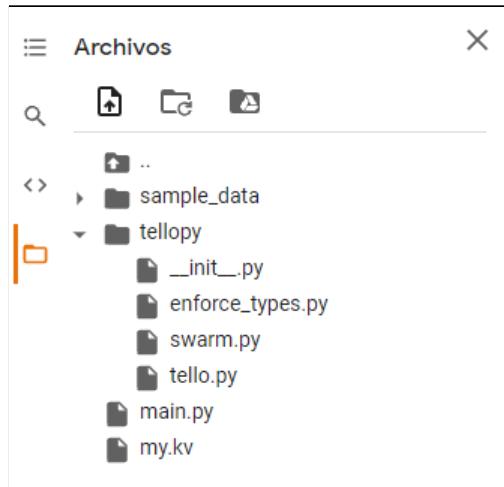


Fig. 4.13 Archivos necesarios en Buildozer

Seguidamente, se irán ejecutando las comandas ya configuradas una a una. Para ello, únicamente es necesario clicar al botón de play ubicado a la izquierda de cada comanda y esperar al visto verde, que nos indica que ya se ha completado la ejecución.

Una vez ejecutada la comanda “!buildozer init”, la cual hay que confirmar su ejecución escribiendo “y” a la pregunta “Are you sure you want to continue [y/n]?", se habrá generado un archivo buildozer.spec. Para poder visualizarlo, es necesario clicar en el botón de actualizar archivos y para poder editarlo, es tan simple como hacer doble click sobre él.

Este archivo hace referencia a las especificaciones necesarias para generar el archivo .apk de la aplicación, por lo que es de gran importancia editararlo

correctamente. Cabe destacar que en este apartado sólamente se van a indicar los cambios necesarios para el funcionamiento de la app, ya que es un archivo de 330 líneas donde se encuentra un gran número de especificaciones a las que entrar en detalle pero que van a ser irrelevantes para nuestra aplicación.

Para empezar, en la línea 4 es posible indicar el título deseado para la app. Por ejemplo, se le va a indicar “Tello EDU Basic Control”. A continuación, en la línea 39 se deben añadir las librerías requeridas para la app, es decir, a parte de python3 y kivy (que ya están indicadas predeterminadamente), es necesario incluir: tellopy (la carpeta creada en archivos), opencv y numpy, ya que éstos dos últimos son requeridos para el funcionamiento de tellopy.

Después, en la línea 55 se va a indicar orientation = landscape, para que la aplicación sólo funcione horizontalmente, y además, se va a descomentar la línea 88 añadiendo diferentes permisos para la app (android.permissions = INTERNET, ACCESS_NETWORK_STATE, WRITE_EXTERNAL_STORAGE), que por ejemplo, posibilitan el uso de la conexión Wi-Fi requerida para enlazarse al drone.

Por último, también se va a descomentar la línea 216 (“android.logcat_filters = *:S python:D”), como ya se había comentado en el apartado [4.3. Instalación del entorno de trabajo](#), para así poder comprobar el Logcat de la aplicación mediante Android Studio y poder observar si existe algún problema mientras se está haciendo uso de ésta. El archivo builddozer.spec editado se va a poder encontrar en [1].

Al ya haber configurado las especificaciones, se va a poder ejecutar la última comanda “!builddozer -v android debug”, encargada de generar el archivo .apk. Como ya se comentó anteriormente, esta comanda puede llegar a tardar más de media hora en ser ejecutada completamente. Si por alguna razón aparece algún error en la creación de la app, después de corregirlo se debe ejecutar la comanda “!builddozer android clean”, para posteriormente poder volver a realizar la generación del archivo .apk. Es decir, después de la comanda de limpieza, únicamente sería necesario volver a ejecutar “!builddozer -v android debug”.

Finalmente, en la carpeta bin va a aparecer el archivo .apk de la aplicación. Después de descargarlo, se va a inicializar AndroidStudio (configurados como se indica en [4.3. Instalación del entorno de trabajo](#)) y se va a instalar la app en el Android.

Una vez instalada, se va a inicializar y comprobar que no existe ningún crasheo. En el caso que éste existiera, se debe revisar el Logcat de Android Studio para conocer de dónde proviene el error y poder corregirlo.

4.4.3. Instalación de la aplicación en el dispositivo móvil

Para poder instalar la aplicación en el dispositivo móvil, se debe enviar el archivo .apk al teléfono mediante correo electrónico, google drive o cualquier herramienta disponible, y al intentar instalarla, se va a solicitar un permiso específico que tiene que ser configurado en ajustes. El proceso de configuración es el siguiente: Ajustes/Configuración > Aplicaciones > (Seleccionar aplicación usada para el envío del .apk) > Instalar apps desconocidas. Como se puede apreciar en la Fig. 4.14, se ha conseguido instalar la aplicación correctamente en el dispositivo móvil.

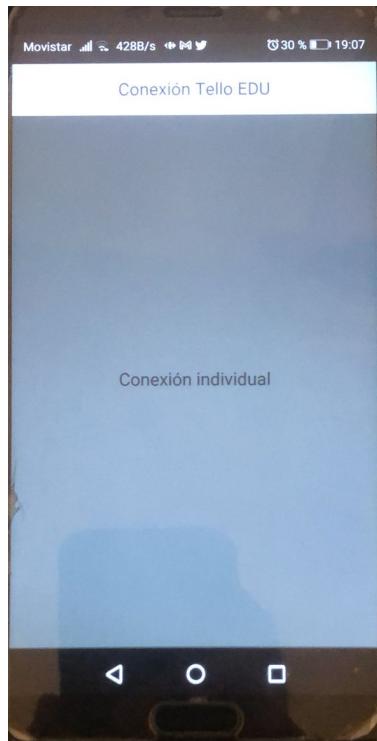


Fig. 4.14 Visualización aplicación en dispositivo móvil

A partir del minuto 2:51 en el vídeo de Youtube [2], se encuentra una demostración visual del control del Tello EDU. Cabe destacar que la interfaz de la app es la final, ya que los vídeos fueron grabados al terminar el proyecto y los siguientes apartados ya están incorporados.

4.5. Configuración de planes de vuelo

Una vez creada la aplicación básica para poder controlar el Tello EDU desde el dispositivo móvil, se va a añadir un apartado para poder configurar planes de vuelo. Es decir, se va a abrir la posibilidad de poder enviar una serie de

comandas ordenadas cronológicamente al drone para que posteriormente éste las realice de manera autónoma.

Para empezar, se va a desarrollar el diseño de una nueva pantalla desde donde poder crear el plan de vuelo. Por ello, se debe crear una nueva clase en el main.py llamada PlanVuelo, donde se codificarán todas las funciones necesarias para el funcionamiento de esta nueva pantalla. Además, como ya se observó anteriormente, se deberá añadir esta nueva pantalla en el WindowManager del archivo my.kv (recordar Fig. 4.4), y obviamente, un apartado <PlanVuelo>: donde crear todo el diseño de la pantalla.

El siguiente paso es crear un acceso a esta nueva pantalla, que será un botón ubicado en la parte superior de la pantalla de control. Es decir, añadiremos un botón entre SALIR y DESPEGAR/ATERRIZAR como se puede observar en la parte del código de diseño de la ControlWindow en la Fig. 4.15. La visualización del diseño estará disponible en la Fig. 4.16

```
BoxLayout:  
    orientation: 'horizontal'  
    Button:  
        size_hint: 1, 0.3  
        pos_hint: {"center_x":1, "center_y":0.9}  
        id: salir  
        text: 'SALIR'  
        on_press:  
            app.root.current = "conexion"  
            root.manager.transition.direction = "left"  
        on_release: root.Desconexion()  
    Button:  
        size_hint: 1, 0.3  
        pos_hint: {"center_x":0.5, "center_y":0.9}  
        id:PlanVuelo  
        text: 'PLAN DE VUELO'  
        on_press:  
            app.root.current = "plan"  
            root.manager.transition.direction = "left"  
    ToggleButton:  
        size_hint: 1, 0.30  
        pos_hint: {"center_x":0, "center_y":0.9}  
        id: despegue  
        text: 'DESPEGAR' if self.state == 'normal' else 'ATERRIZAR'  
        on_state: root.DespegaAterriz()
```

Fig. 4.15 Código acceso a plan de vuelo



Fig. 4.16 Botón de acceso a plan de vuelo

A continuación, se va a proceder a diseñar la pantalla PlanVuelo. Cabe destacar que el name de esta pantalla será “plan”, como ya se había podido intuir en la configuración del botón de la Fig. 4.15 y su diseño va a constar principalmente de un BoxLayout vertical de 5 filas con un color de fondo gris claro. Para este primer paso es necesario el siguiente código:

```
<PlanVuelo>:
    name: "plan"
    BoxLayout:
        cols:5
        orientation:"vertical"
        canvas.before:
            Color:
                rgb:0.80, 0.80, 0.80
            Rectangle:
                size: self.size
                pos: self.pos
```

Fig. 4.17 Diseño pantalla PlanVuelo

En la primera de las cinco filas se va a crear un nuevo BoxLayout formado por dos Buttons y un Label. El primer botón tendrá la función de volver a la pantalla de control y el segundo abrirá una ventana PopUp, como la de la Fig. 4.8, donde se ubicará la información necesaria acerca del funcionamiento del plan de vuelo. Para la creación de esta ventana emergente, volveremos al main.py y dentro de la clase PlanVuelo se creará la función info().

Esta función será exactamente igual a la función pressedConexion() creada en la Fig. 4.8 pero añadiendo al Label la característica text_size=(self.width,

None). Esta característica hará que el texto se adecue al tamaño de pantalla del dispositivo, y será necesario porque estamos hablando de varias líneas de información. En la Fig. 4.18 se encuentra el código de la función descrita.

```
"Configuración Plan de Vuelo"
class PlanVuelo(Screen):

    def info(self):
        content = BoxLayout(orientation="vertical")
        pop = Popup(title="INFORMACIÓN", content=content, size_hint=(1, 0.5), auto_dismiss=False)
        texto = Label(text="Para crear el plan de vuelo indique por orden cronológico las comandas que desea enviar al drone."
                     " Tenga en cuenta que la primera comanda será despegar y la última aterrizar."
                     " El rango de distancias es de 20-500 cm y el rango de rotaciones es de 1-360º",
                     text_size=(self.width, None))
        boton = Button(text="OK", size_hint=(0.3, 0.3),
                      pos_hint={"center_x": 0.5, "center_y": 0.5}, on_press=pop.dismiss)
        content.add_widget(texto) # Se añade el Label al BoxLayout
        content.add_widget(boton) # Se añade el botón al BoxLayout
        pop.open()
```

Fig. 4.18 Función PopUp de información

Al ya tener la función info() se procede a codificar el diseño de esta primera fila del BoxLayout principal. Se hará uso de otro BoxLayout donde se añadirán los dos Buttons y el Label como se puede ver en la Fig. 4.19.

Por otro lado, el aspecto visual final de este BoxLayout secundario y del PopUp se encuentra en la Fig. 4.20. (Actualmente el diseño no sería como el de la Fig. 4.20 ya que aún faltan por diseñar las otras 4 filas del BoxLayout principal).

```
BoxLayout:
    size_hint: 1, 0.10
    Button:
        size_hint: 0.2, 1
        id: volver
        text: 'VOLVER'
        on_press:
            app.root.current = "control"
            root.manager.transition.direction = "right"
    Button:
        size_hint: 0.2, 1
        id: info
        text: 'INFO'
        on_press:root.info()
    Label:
        text: "CREACIÓN DEL PLAN DE VUELO"
        color: 0, 0, 0
```

Fig. 4.19 Diseño pantalla Plan de Vuelo



Fig. 4.20 Diseño PopUp de información

Seguidamente, añadiremos a la fila dos del BoxLayout principal un Label. Mediante este Label, se van a poder visualizar las comandas que se le van indicando al drone, así el usuario va a tener una experiencia más amigable con la aplicación al conocer en todo momento las comandas configuradas. Su codificación en cuanto al diseño se encuentra en la Fig. 4.21 y más adelante, se explicará la manera en que actualizar esta lista de comandas.

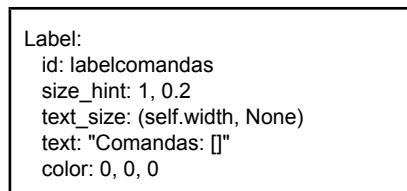


Fig. 4.21 Diseño Label de comandas

La tercera fila estará compuesta por un GridLayout de 6 columnas. Las cuatro primeras filas estarán formadas por la siguiente secuencia de widgets: Label, TextInput, Button, Label, TextInput, Button. De esta manera, se podrán programar los 8 movimientos posibles del drone. En la Fig. 4.22, se podrá ver el código del diseño de la primera de las cuatro filas del GridLayout.

```
GridLayout:  
cols:6  
Label:  
    text: "Avanza."  
    text_size: self.width, None  
    color: 0, 0, 0  
    size_hint:0.20, 1  
  
TextInput:  
    id:plandelantetexto  
    size_hint:0.15, 1  
    multiline: False  
    input_filter: "int"  
  
Button:  
    text: "Añadir"  
    id: plandelante  
    size_hint:0.15, 1  
    on_press:root.Comandas()  
  
Label:  
    text: "Retrocede."  
    text_size: self.width, None  
    color: 0, 0, 0  
    size_hint:0.20, 1  
  
TextInput:  
    size_hint:0.15, 1  
    id:plandetrastexto  
    multiline: False  
    input_filter: "int"  
  
Button:  
    text: "Añadir"  
    size_hint:0.15, 1  
    id: plandetas  
    on_press:root.Comandas()
```

Fig. 4.22 Diseño primera fila GridLayout

Como ya se puede observar, los botones activarán una función llamada Comandas(), que más adelante será explicada, y los TextInput solo permitirán escribir números en ellos.

Por el momento, se explicarán las dos últimas filas del BoxLayout principal, que cada una de ellas estará formada por un BoxLayout de dos botones. Estos botones tendrán la función de indicar el despegue y aterrizaje, de eliminar la última comanda configurada, en el caso que el usuario se haya equivocado, y por último, iniciar el plan de vuelo. El código del diseño se encuentra en la Fig. 4.23.

```

BoxLayout:
    orientation: "horizontal"
    size_hint: 1, 0.2
    Button:
        text: "Despegar"
        id:plandespegue
        on_press:root.Comandas()

    Button:
        text: "Aterrizar"
        id:planaterrizaje
        on_press:root.Comandas()

BoxLayout:
    orientation: "horizontal"
    size_hint: 1, 0.2
    Button:
        text: "Eliminar última comanda"
        id: planelimina
        on_press:root.Comandas()

    Button:
        text: "Iniciar vuelo"
        id: planvuela
        on_press:root.Comandas()

```

Fig. 4.23 Diseño últimas filas BoxLayout principal

Finalmente ya se ha diseñado toda la pantalla referente al plan de vuelo y se puede apreciar su aspecto en la Fig. 4.24.

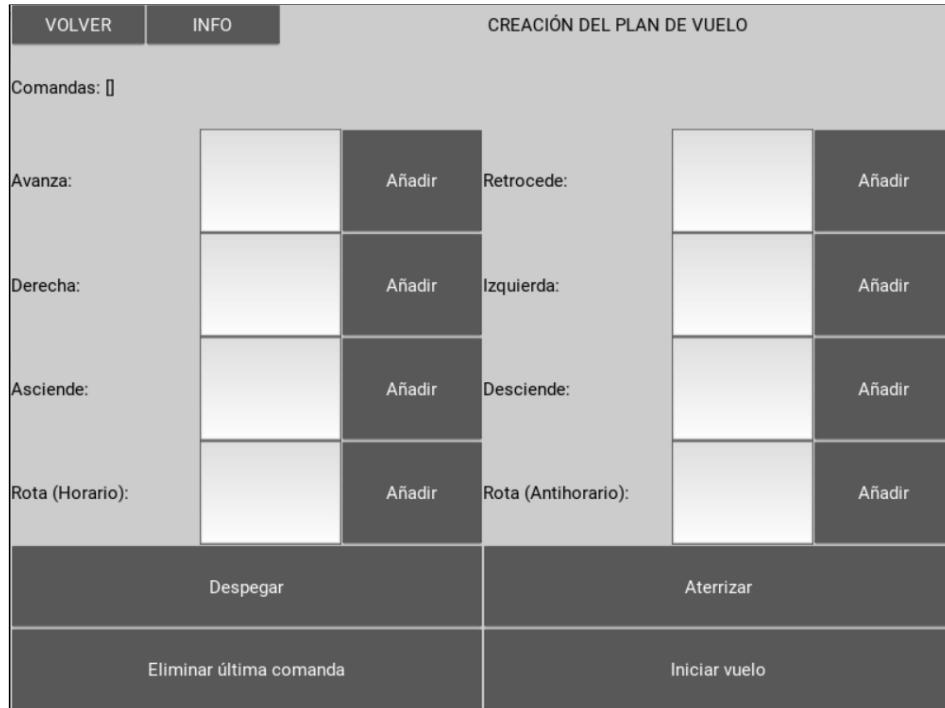


Fig. 4.24 Diseño pantalla Plan de Vuelo

El último paso para tener terminado este apartado es la creación de la función Comandas(). Esta función se va a encargar de ir actualizando la lista de comandas y posteriormente enviarlas al drone una a una en el momento preciso.

Para empezar, se va a crear un archivo de texto llamado Comandas.txt y mediante “comandas = open("Comandas.txt", 'w')” se va a asegurar que este .txt esté vacío siempre al iniciar la aplicación. Después, dentro de la función Comandas() se va a iniciar con comandas = open("Comandas.txt", 'a'), ya que siempre que se vaya a llamar a esta función será para añadir.

A continuación, se va a programar cada uno de los botones para que proceda a añadir en el archivo de texto la comanda específica configurada mediante el TextInput. En la Fig. 4.25, se podrá observar el código únicamente para el botón de movimiento hacia delante y para el botón de rotar de manera horaria (los demás siguen la misma dinámica).

```

if self.ids.plandelante.state == "down":
    if 20 <= int(self.ids.plandelantetexto.text) <= 500:
        comanda = "forward " + self.ids.plandelantetexto.text
        self.ids.plandelantetexto.text = "" #TextInput vacio al haber pulsado
        comandas.write(comanda + "\n")
    else:
        self.ids.plandelantetexto.text = "Error"

if self.ids.planhorario.state == "down":
    if 1 <= int(self.ids.planhorariotexto.text) <= 360:
        comanda = "cw " + self.ids.planhorariotexto.text
        self.ids.planhorariotexto.text = ""
        comandas.write(comanda + "\n")
    else:
        self.ids.planhorariotexto.text = "Error"

```

Fig. 4.25 Programación de botones

Como se ha podido observar, está programado para que solo deje añadir valores dentro de los rangos indicados en el botón de información (recordar Fig. 4.20). Si se cumple este rango, la variable comanda tomará un valor que posteriormente será enviado al archivo de texto.

Para ver la estructura que deben seguir todas las comandas, se va a simular un plan de vuelo con todos los movimientos posibles y se va a observar la forma final del archivo Comandas.txt en la Fig. 4.26.

```

takeoff
forward 50
right 60
up 40
cw 180
back 30
left 20
down 20
ccw 300
land

```

Fig. 4.26 Estructura de las comandas

En el caso que se desee eliminar la última comanda, se deberá cambiar a modo lectura mediante “comandas = open("Comandas.txt", 'r')”, extraer un listado de todas las filas, volver al modo escritura, y por último, volver a escribir todas las comandas de la lista extraída anteriormente exceptuando la última posición. En la Fig. 4.27 se va a poder apreciar el proceso de eliminación de la última comanda de mejor manera.

```

if self.ids.planelimina.state == "down":
    comandas = open("Comandas.txt", 'r') #Cambiamos a modo lectura
    lineas = comandas.readlines() #Lista con todas las lineas
    comandas = open("Comandas.txt", 'w') #Cambiamos a modo escritura
    linea = lineas[-1] #Seleccionamos última linea
    lineas.remove(linea) #La eliminamos
    for linea in lineas:
        comandas.write(linea) #Reescribimos todas las otras lineas

```

Fig. 4.27 Proceso para eliminar última comanda

Para que el Label de comandas diseñado en la Fig. 4.21 se vaya actualizando, habrá que configurar que al llamar a la función Comandas(), después de revisar si se ha añadido o eliminado algo en el plan de vuelo, se haga una lectura del archivo de texto, se cree una lista con todas las comandas y se actualice el texto del Label. Tanto el código como el aspecto visual para esta función se puede encontrar en la Fig. 4.28.

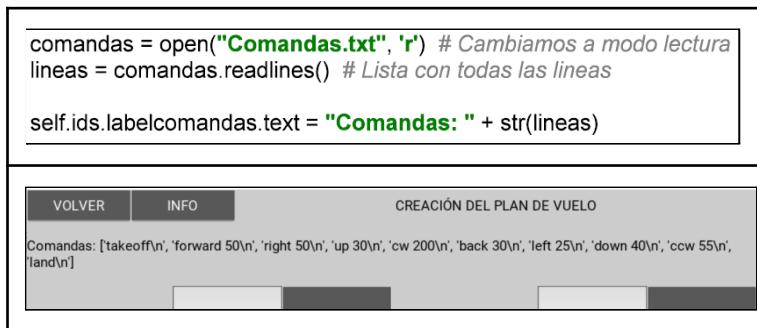


Fig. 4.28 Lista de comandas

Por último, falta configurar como van a ser enviadas las comandas del plan de vuelo. El objetivo es ir enviando todas las comandas individualmente en el momento adecuado. Es decir, se procederá a enviar la siguiente orden cuando se haya terminado la comanda anterior.

Para conocer si la comanda anterior ha sido realizada por completo, se va a ir comprobando la velocidad de la aeronave en los 3 ejes de dirección. Cuando las 3 velocidades sean 0, significa que el drone está parado y ya ha realizado la comanda, por lo que se le pondrá enviar la siguiente. Asimismo, es importante saber que el Tello EDU después de despegar necesita al menos 8 segundos para estabilizarse y poder empezar a realizar comandas. El código para realizar este proceso se encuentra en la Fig. 4.29.

```
if self.ids.planvuela.state == "down":
    comandas = open("Comandas.txt", 'r')
    lineas = comandas.readlines()
    i = 0
    while i < len(lineas):
        comanda = lineas[i].rstrip()
        if comanda == "takeoff":
            me.send_command_with_return(comanda)
            sleep(8) # Esperar a que realice el despegue correctamente
            i = i + 1
        elif me.get_speed_x() <= 0 and me.get_speed_y() <= 0 and me.get_speed_z() <= 0:
            print("Esta parado, envio comanda")
            me.send_command_with_return(comanda)
            i = i + 1
            sleep(0.5) #Para cuando se pare no envie 2 comandas a la vez
        else:
            pass
```

Fig. 4.29 Proceso de envío de comandas

Para finalizar, siguiendo el mismo procedimiento que hasta ahora, se podrá encontrar la demostración visual del plan de vuelo en Youtube [2] a partir del minuto 3:29 y el código completo en GitHub [1].

4.6. Visualización imagen del drone

Este apartado tenía el objetivo de incorporar la visualización de las imágenes tomadas por el Tello EDU en directo en la aplicación. En un principio, su funcionamiento parecía correcto ya que en el entorno web se ejecutaba todo a la perfección. Aún así, a la hora de testearlo en la aplicación, nunca se llegó a conseguir el propósito.

De todos modos, se va a explicar detalladamente la manera en que se había propuesto el desarrollo de esta funcionalidad y se va a documentar todo lo investigado acerca de la imposibilidad encontrada.

Mi objetivo con este apartado es poder ayudar a aquellos que se encuentren con la misma dificultad, ofreciendo todo el conocimiento obtenido después de haber invertido una gran cantidad de tiempo intentando solucionar dicho problema.

4.6.1. Planteamiento de desarrollo propuesto

Para poder visualizar la imagen captada por el drone en el entorno web, se va a usar un bucle donde se vayan actualizando los frames enviados por la aeronave siguiendo la misma dinámica que en los apartados anteriores del capítulo 2.

Para empezar, dentro del código de diseño de la pantalla Control Window, se va a añadir el widget Image antes del BoxLayout principal. De esta manera, la imagen aparecerá como fondo en la pantalla. El código del diseño se puede observar en la Fig. 4.30.

```
<ControlWindow>:
    name:"control"
    on_enter: root.clock()

    Image:
        id: video

    BoxLayout:
```

Fig. 4.30 Código de diseño para la imagen

Como ya se puede apreciar en la Fig. 30, se ha configurado que al entrar en la pantalla de control se ejecute la función clock (on_enter: root.clock()). El propósito de esta función es hacer que se repita otra función cada cierto periodo de tiempo. Esto es posible gracias a widget clock de kivy, que obviamente se debe importar previamente, y mediante “Clock.schedule_interval(self.video, 0.1)”, se va a ejecutar una función video() cada 0.1 segundos.

Por último, falta configurar la función video. Anteriormente, se obtenía el frame y posteriormente mediante cv2.imshow se visualizaba. En este caso, no es posible realizar este proceso, ya que la información del frame llega en forma matricial y Kivy no puede tratar estos datos. Por esa razón, se va a crear un archivo jpg mediante cv2.imwrite, que será cargado como imagen y actualizado constantemente. No hay que olvidar que para que se puedan obtener los frames de la aeronave es necesaria la comanda streamon() que se añadirá en la función pressedConexion() (Fig. 4.8). El código de la función clock() y video() aparece en la Fig. 4.31.

```

def clock(self): #Función para actualizar el frame del video
    Clock.schedule_interval(self.video, 0.1)

def video(self, dt):
    img = me.get_frame_read().frame
    cv2.imwrite('Imagen.jpg', img) #matrix to jpg
    self.ids.video.source = 'Imagen.jpg'
    self.ids.video.reload() #Cargar nueva imagen

```

Fig. 4.31 Funciones clock y video

Finalmente, se va a añadir la característica background_color: 0, 0, 0, 0.2 a los botones encargados de dirigir el drone. Esta característica hará que los botones sean “transparentes” y posibiliten ver la imagen por completo. el aspecto visual de la aplicación en el entorno web es el siguiente:

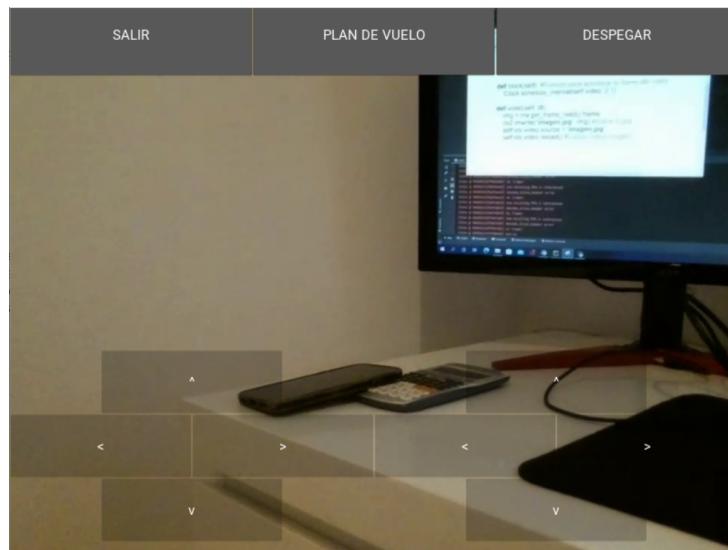


Fig. 4.32 Aspecto visual video en aplicación web

Otro método encontrado para poder cargar imágenes en Kivy, es haciendo uso de texturas. Después de importarlas mediante “from kivy.graphics.texture import Texture”, haciendo uso del código de la Fig. 4.33, se podrán visualizar las imágenes ofrecidas por el drone al igual que usando el método anterior.

De hecho, se podría decir que este método es más interesante y eficiente ya que no implica tener que crear un archivo “Imagen.jpg” y sobreescribirlo constantemente.

```

def clock(self): #Función para actualizar el frame del video
    print("Entro al clock")
    Clock.schedule_interval(self.video, 1/30)

def video(self, dt):
    img = me.get_frame_read().frame
    buffer = cv2.flip(img, 0).tobytes()
    tex = Texture.create(size=(img.shape[1], img.shape[0]), colorfmt='bgr')
    tex.blit_buffer(buffer, colorfmt='bgr', bufferfmt='ubyte')
    self.ids.video.texture = tex

```

Fig. 4.33 Alternativa video con texturas

4.6.2. Error de visualización de imagen en dispositivo móvil

A la hora de cargar la app en el Android con el planteamiento propuesto anteriormente, se producía un crasheo. El error que aparecía en el Logcat de Android Studio era el siguiente: Fatal signal 7 (SIGBUS), code 1 (BUS_ADRALN).

El siguiente paso fue investigar acerca de éste para tratar de corregirlo pero en internet hay mucha información dispersa y general acerca de él, que no clarifican nada útil para nuestro caso.

Por esa razón, procedí a comentar la situación y el error por StackOverflow y por el foro de Tellopilots, ya que en mi caso, estas plataformas me habían ayudado durante todo el desarrollo del trabajo. El link a mis consultas se puede encontrar en [13] y [14] y en las Fig. 4.34 y Fig. 4.35 se pueden apreciar.

The image shows a Stack Overflow post with the following details:

- Title:** Fatal signal 7 (SIGBUS), code 1 (BUS_ADRALN). on TelloEDU while I get_frame_read().frame on android app created with buildozer
- Upvotes:** 0
- Content:**

I am creating an android app using kivy and buildozer to manage the TelloEDU drone and everything is working fine until im trying to get the image. On my computer, when i ask for get_frame_read().frame after a couple seconds and some:

```
[h264 @ 00000272a264c700] non-existing PPS 0 referenced
[h264 @ 00000272a264c700] non-existing PPS 0 referenced
[h264 @ 00000272a264c700] decode_slice_header error
[h264 @ 00000272a264c700] no frame!
```

The image start flowing.

I think that is the problem then on my android. Just at the moment that I am asking for the image, it crashes with "A/libc: Fatal signal 7 (SIGBUS), code 1 (BUS_ADRALN)".

That is the code to upload the frames:

```
def clock(self):
    Clock.schedule_interval(self.video, 0.001)

def video(self, dt):
    img = me.get_frame_read().frame
    cv2.imwrite("Imagen.jpg", img)
    self.ids.video.source = "Imagen.jpg"
    self.ids.video.reload()
```

Can someone help me with this? Or maybe have some ideas about it? Do you think the problem of the crash is h264?

Ty

Fig. 4.34 Consulta Stackoverflow

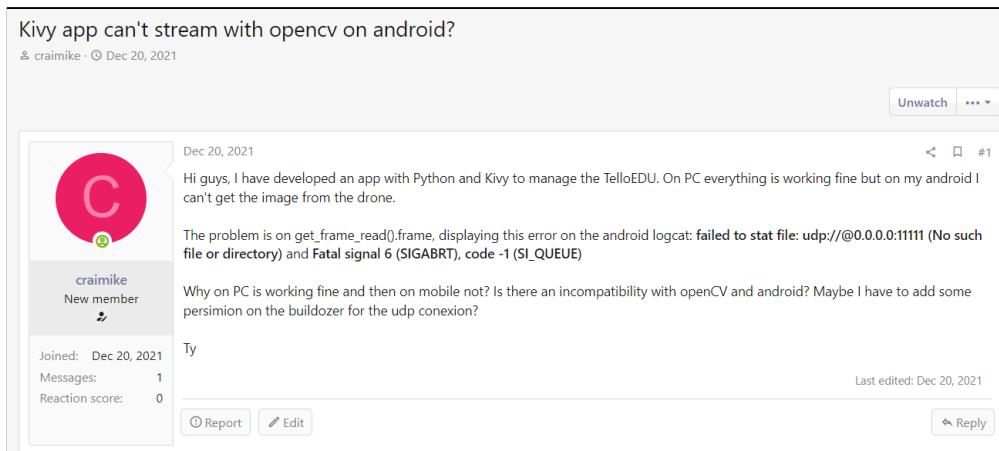


Fig. 4.35 Consulta foro Tellopilots

Por desgracia no obtuve respuesta, y por esa razón seguí investigando por mi cuenta ya que posiblemente no estaba haciendo la pregunta adecuadamente. Era la primera vez que planteaba dudas en estas plataformas y después de esta experiencia, recomendaría siempre incorporar el código completo usado para que la gente pueda entrar en contexto.

El siguiente avance fue descubrir que el error provenía del `cv2.imwrite("imagen.jpg", img)` (Fig. 4.31), ya que la variable `img` estaba vacía. Gracias a ello pude situar el error, que procedía del “`img = me.get_frame_read().frame`”.

Posteriormente, me di cuenta de que `cv2.VideoCapture()` nunca conseguía leer los frames del Tello EDU, por lo que sospeche acerca de necesitar incorporar permisos adicionales de Android en buildozer para realizar conexiones udp, como ya preguntaba en el foro de Tellopilots.

Esta teoría fue revocada rápidamente al intentar sin éxito capturar mediante `cv2.VideoCapture(0)` la imagen ofrecida por la misma cámara del dispositivo móvil. Dicho de otra manera, el error no estaba relacionado con el Tello EDU.

Después de informarme acerca de ello, aprendí que existe una incompatibilidad entre la función `VideoCapture` de Opencv y Android. Es decir, era inviable obtener el objetivo haciendo uso de esta librería.

A pesar de ello, encontré en algunas webs (ver [15] y [16]) que se había hecho una actualización a principios de 2021 la cual compatibiliza `VideoCapture` con Android. En estas se comenta que es necesaria al menos la versión 4.5.2 de Opencv y Android ndk mínimo de 24. Aún así, aplicando en las especificaciones de Buildozer `opencv==4.5.2`, `android.minapi = 24` y `android.ndk_api = 24` seguía sin funcionar.

El código planteado junto con la posibilidad de aplicar el seguimiento facial está disponible en GitHub [1], ya que en el entorno web funciona correctamente.

Aún así, al estar subida la última versión de la app, todo lo relacionado con la imagen está comentado para que la aplicación funcione correctamente en Android sin ningún crasheo.

4.7. Modo Enjambre

En este apartado se va a incorporar en la aplicación la posibilidad de conectarse a un enjambre de drones y realizar las diferentes coreografías que se habían ya programado en el capítulo 3. El proceso para poder realizar la conexión no es especialmente sencillo, por lo que se intentará crear una interfaz amigable.

Para empezar, es necesario importar la librería referente a Tello Swarm, y a continuación, en la pantalla ConexionWindow se añadirá un botón para poder elegir entre conexión individual o de múltiples drones. En la Fig. 4.36 se puede observar el código de diseño del botón y el aspecto final de la pantalla de conexión.

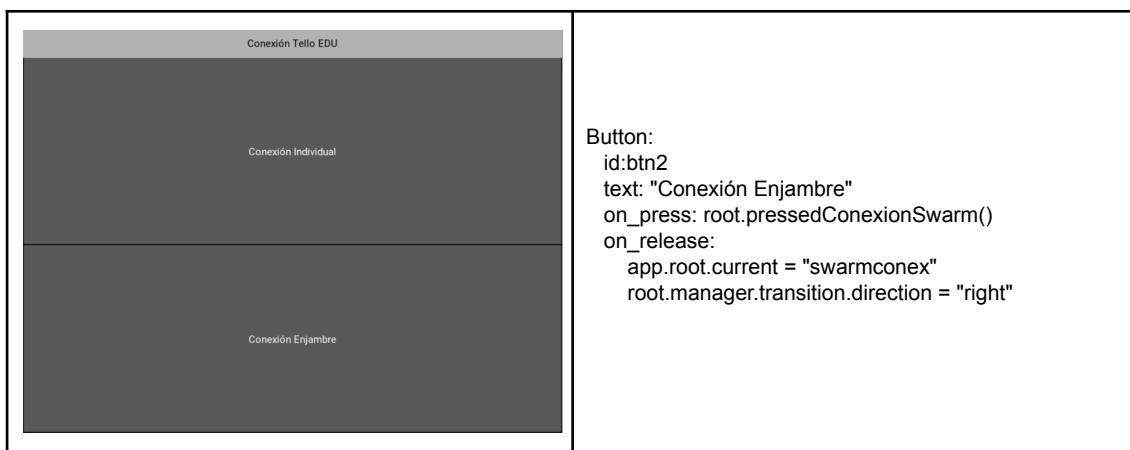


Fig. 4.36 Botón conexión a enjambre

Posteriormente, como ya se puede intuir en la anterior figura, se creará una nueva pantalla llamada SwarmConexion, con name: swarmconex, y se añadirá la función pressedConexionSwarm() que será llamada al pulsar el botón. Esta función, será exactamente igual a la referente a la conexión individual, pero se aprovechará el PopUp para indicar las instrucciones del proceso de conexión a seguir. En la Fig. 4.37 se puede visualizar dicha función.

```

def pressedConexionSwarm(self):
    content = BoxLayout(orientation="vertical")
    pop = Popup(title="CONEXIÓN MODO AP", content=content, size_hint=(1, 0.5),
    auto_dismiss=False)
    texto = Label(text="Debe conectarse individualmente a cada drone y indicar los
    parámetros de la red WiFi para realizar la conexión. "
    "Después, es necesario indicar la IP que el router ha asignado a cada drone
    ."
    "Si ya ha realizado el proceso con todos los drones, clique en CONEXIÓN
    ENJAMBRE REALIZADA",
    text_size=(self.width, None))
    boton = Button(text="OK", size_hint=(0.3, 0.3),
    pos_hint={"center_x": 0.5, "center_y": 0.5}, on_press=pop.dismiss)
    content.add_widget(texto) # Se añade el Label al BoxLayout
    content.add_widget(boton) # Se añade el botón al BoxLayout
    pop.open()

```

Fig. 4.37 Función de conexión a enjambre

Una vez ya ofrecida la información necesaria para realizar la conexión al usuario, se va a acceder a la pantalla SwarmConexion. En la Fig. 4.38 se encuentra su aspecto visual y en la Fig. 4.39 el código de diseño del archivo .kv.

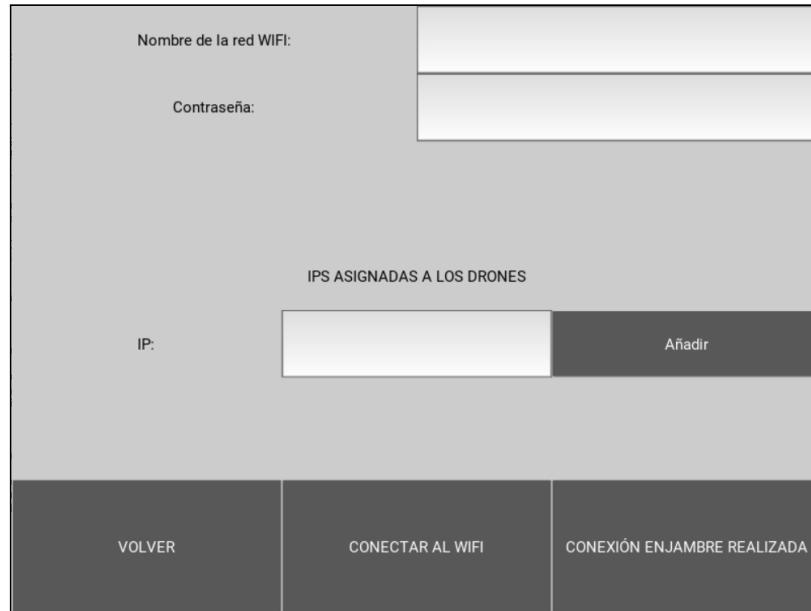


Fig. 4.38 Aspecto visual pantalla SwarmConexion

```

<SwarmConexion>
    name: "swarmconex"
    BoxLayout:
        spacing: 100
        cols:3
        orientation:"vertical"
        canvas.before:
            Color:
                rgb:0.80, 0.80, 0.80
            Rectangle:
                size: self.size
                pos: self.pos
        GridLayout:
            cols:2
            Label:
                text: "Nombre de la red WIFI:"
                color: 0, 0, 0
            TextInput:
                id:idwifi
                multiline: False
            Label:
                text: "Contraseña:"
                color: 0, 0, 0
            TextInput:
                id:passwifi
                multiline: False

        BoxLayout:
            orientation: "vertical"
            Label:
                text: "IPS ASIGNADAS A LOS DRONES"
                color: 0, 0, 0
            BoxLayout:
                orientation: "horizontal"
                Label:
                    text: "IP:"
                    color: 0, 0, 0
                TextInput:
                    id:ip
                    multiline: False
                Button:
                    text: "Añadir"
                    on_press:root.SetIps()
        BoxLayout:
            orientation: "horizontal"
            Button:
                text: "VOLVER"
                on_press:
                    app.root.current = "conexion"
                    root.manager.transition.direction = "left"
            Button:
                text: "CONECTAR AL WIFI"
                id: wificonex
                on_press:root.APConexion()
            Button:
                text: "CONEXIÓN ENJAMBRE REALIZADA"
                id: swarmconex
                on_release:
                    app.root.current = "swarmcontr"
                    root.manager.transition.direction = "right"

```

Fig. 4.39 Código diseño de pantalla SwarmConexion

Como se puede apreciar, el objetivo es que el usuario vaya conectándose individualmente a cada drone y cambiando su modo a Acces Point aportando el

nombre y contraseña del Wi-Fi. Una vez todos los drones están configurados, hay que ir añadiendo las IPs que el router ha asociado a cada aeronave.

Para que este proceso funcione correctamente se deben implementar dos nuevas funciones. Una encargada de cambiar el modo a cada aeronave, y la otra, de crear un archivo de texto que recoja todas las IPs añadidas. En la Fig. 4.40 se pueden apreciar ambas funciones.

```
"Conexion drones en modo AP al router"
class SwarmConexion(Screen):
    def APConexion(self):
        me.connect()
        ssid = self.ids.idwifi.text
        password = self.ids.passwifi.text
        print(ssid, password)
        me.connect_to_wifi(ssid, password)

        content = BoxLayout(orientation="vertical")
        pop = Popup(title="Modo AP", content=content, size_hint=(1, 0.3), auto_dismiss=False)
        texto = Label(text="Modo Access Point establecido, conéctese al siguiente drone y
realice el mismo proceso")
        boton = Button(text="OK", size_hint=(0.3, 0.3),
                      pos_hint={"center_x": 0.5, "center_y": 0.5}, on_press=pop.dismiss)
        content.add_widget(texto) # Se añade el Label al BoxLayout
        content.add_widget(boton) # Se añade el botón al BoxLayout
        pop.open()

    def SetIps(self):
        Ips = open("Ips.txt", 'a')
        Ip = self.ids.ip.text
        self.ids.ip.text = "" # TextInput vacío al haber pulsado
        Ips.write(Ip + "\n")
```

Fig. 4.40 Funciones pantalla SwarmConexion

Cabe destacar que al igual que se hizo en el apartado [4.5 Configuración de planes de vuelo](#), se debe crear un archivo llamado “Ips.txt” en el proyecto. También, al iniciar la aplicación se deberá realizar la comanda “Ips = open("Ips.txt", 'w')” para que dicho archivo de texto se vacíe.

Para terminar, se va a diseñar la última pantalla necesaria, SwarmControl. Esta consta de seis botones, para indicar al drone cuando aterrizar o despegar, o cuando realizar alguna de las cuatro coreografías disponibles. Su aspecto visual está disponible en la Fig. 4.41 y su codificación en la Fig. 4.42.



Fig. 4.41 Diseño pantalla SwarmControl

```
<SwarmControl>:
    on_enter: root.pressedConexion()
    name: "swarmcontr"
    BoxLayout:
        spacing: 200
        orientation: "vertical"
        BoxLayout:
            orientation: "horizontal"
            size_hint: (1, 0.3)
            Button:
                text: "DESPEGAR"
                id: desp
                on_press:root.Despegue()
            Button:
                text: "ATERRIZAR"
                id: atr
                on_press:root.Aterrizaje()

        BoxLayout:
            orientation: 'vertical'
            Label:
                text: "COREOGRAFIA"
        BoxLayout:
            orientation: "horizontal"
            Button:
                text: "CUADRADO"
                id: cuadrado
                on_press:root.Cuadrado()
            Button:
                text: "CIRCULO"
                id: circulo
                on_press:root.Circulo()
            Button:
                text: "FLIPS"
                id: flips
                on_press:root.Flips()
            ToggleButton:
                text: "REBOTE"
                id: rebote
                on_press:root.BucleRebote()
```

Fig. 4.42 Código diseño pantalla SwarmControl

Como se puede apreciar en el código de diseño, al entrar en SwarmControl se va a ejecutar la comanda `pressedConexion()` (ver Fig. 4.43), que se encargará de realizar la conexión al enjambre mediante el archivo de texto creado. Por otro lado, cada uno de los botones diseñados para coreografías llamará a una función de [3.3 Coreografía drones](#). Recordar que el código completo de la aplicación está disponible en GitHub [1] y la demostración visual del control del enjambre en Youtube [2] a partir del minuto 4:47.

```
"Control del Enjambre"
class SwarmControl(Screen):

    def pressedConexion(self):
        Ips = open("Ips.txt", 'r')
        Ipsstr = [linea.rstrip() for linea in Ips]
        print(Ipsstr)

        self.Enjambre = swarm.TelloSwarm.fromIps(Ipsstr)
        self.Enjambre.connect()

        for tello in self.Enjambre:
            print(tello.get_battery())
```

Fig. 4.43 Función conexión al enjambre

Para finalizar, me gustaría comentar que para que la aplicación sea más robusta, en este apartado se debería crear una función que devuelva el drone a su posición inicial después de realizar una coreografía. El correcto desarrollo de estas interacciones programadas entre drones depende del punto de partida de las aeronaves, por lo que sería interesante.

De todos modos, al existir poca precisión en los movimientos generados por los drones en modo enjambre, es inviable realizar un seguimiento de la posición así como se desarrolló en el apartado [2.2.4. Mapeo del movimiento](#). La solución propuesta es aterrizar el enjambre después de cada coreografía realizada para recolocar los drones en su punto de despegue.

CONCLUSIÓN

Aún recuerdo mi primer día en el campus de la EETAC quedándome fascinado con las pruebas que se realizaban con un drone octocóptero. Desde ese momento supe que quería enfocar mi futuro en el sector de los RPAS y este proyecto ha sido un pequeño paso hacia él.

Desde el comienzo, el desarrollo de mi trabajo de fin de grado ha sido una experiencia llena de aprendizaje. Mi principal objetivo era explorar la programación que hay detrás de un drone y las diferentes funcionalidades que pueden ofrecer, y además de ello, he podido aprender acerca de Python, inteligencia artificial, procesado y desarrollo de imagen, Kivy y Android.

Recomendaría a todo el mundo realizar proyectos basados en prácticas experimentales como en mi caso, la satisfacción de obtener un correcto funcionamiento práctico después de una gran dedicación llega a ser adictivo y te impulsa a avanzar día a día con motivación.

Gracias a esta motivación, he podido llegar a crear una aplicación para controlar mi propio drone, lo cual era impensable inicialmente debido a la falta de experiencia en torno a ello. Asimismo, he obtenido la capacidad de detectar y solucionar un error cometido en uno de los mejores tutoriales sobre programación con Tello EDU que he encontrado. A pesar de ser un video con millones de visualizaciones, no pude identificar a nadie que se percatara de ello, por lo que decidí informar personalmente al autor obteniendo agradecimientos por su parte.

Por otro lado, hay algunos aspectos del proyecto que debido a diferentes dificultades encontradas, como la poca precisión en cuanto a los movimientos del Tello EDU o la incompatibilidad de OpenCv y Android, no he podido llevarlos a cabo con éxito. Aún así, he podido extraer mis conclusiones y documentar todo el proceso para que cualquier persona pueda continuar con el trabajo realizado. De hecho, aunque no disponga de más tiempo para seguir con el proyecto, mi intención es intentar buscar diferentes soluciones para lograr los objetivos deseados durante los próximos meses.

Otro de los objetivos que he tenido presente ha sido buscar el desarrollo de una memoria muy descriptiva, visual y dinámica. De este modo, este trabajo puede servir educativamente para introducir a gente con conocimientos básicos de programación a un sector en auge como el de los drones. En mi caso, me hubiera encantado poder conocer esta tecnología durante mi enseñanza obligatoria, bachillerato o universidad. Si al menos una persona puede conocer prematuramente hacia dónde quiere dirigir su futuro gracias a este trabajo, estaré satisfecho.

Para terminar, me gustaría destacar una de las conclusiones personales extraídas durante este proyecto. Si te apasiona algo, dedicate a ello, porque las

dificultades y obstáculos que te encuentres se convertirán en motivación y retos para seguir adelante.

BIBLIOGRAFÍA

- [1] Jaime Jaume Busquets - *Librería TelloEDU-programing* (2021) - [Online]
<https://github.com/JaimeJaumeBusquets/TelloEDU-programming>
- [2] Jaime Jaume Busquets - *Demostraciones visuales TFG Jaime Jaume Busquets - Programación de un Tello EDU* (2021) - [Online]
<https://www.youtube.com/watch?v=4exZv-AZrDQ>
- [3] Ryze Tech - *Tello Manual del usuario v1.0* (2018) - [Online]
https://dl-cdn.ryzerobotics.com/downloads/Tello/201806mul/Tello%20User%20Manual%20V1.0_ES.pdf
- [4] Ryze Tech - *Tello Guia de usuario para SDK 2.0 v1.0* (2018) - [Online]
<https://dl-cdn.ryzerobotics.com/downloads/Tello/Tello%20SDK%202.0%20User%20Guide.pdf>
- [5] Vizdx LLC - *Robotics and AI Computer Vision* (2019) - [Online]
<https://www.computervision.zone/>
- [6] Damià Fuentes Escoté - *Librería DJITelloPy* (2021) - [Online] [GitHub](#) - [damiafuentes/DJITelloPy](#)
- [7] Murtaza's Workshop - Robotics and AI - *Drone Programming With Python Course | 3 Hours | Including x4 Projects | Computer Vision* (2021) - [Online] <https://www.youtube.com/watch?v=LmEcyQnfpDA>
- [8] Opencv - *haarcascade_frontalface_default.xml* (2013) - [Online]
https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml
- [9] GOOGLE LLC - *Web libreria Mediapipe* (2020) - [Online]
<https://mediapipe.dev/>
- [10] Jaime Jaume Busquets - *Código Buildozer Google Collab* (2021) - [Online]
https://colab.research.google.com/drive/1CrO9vgwwUzUH9cPZp_W3vzwC_UlIFzkN#scrollTo=Vz7YvKFZbCoX
- [11] Google Developers - *Página oficial Android Studio* (2013) - [Online]
<https://developer.android.com/studio>
- [12] Antonio Leiva - *Cómo activar la Depuración USB en tu teléfono Android* (2021) - [Online] <https://devexperto.com/como-activar-la-depuracion-usb/>
- [13] Jaime Jaume Busquets - *Consulta en Stackoverflow* (2021) - [Online]
<https://es.stackoverflow.com/questions/503385/fatal-signal-7-sigbus-code-1-bus-adraln-on-telloedu-while-i-get-frame-read>

- [14] Jaime Jaume Busquets - *Consulta foro Tellopilots* (2021) - [Online]
<https://tellopilots.com/threads/kivy-app-cant-stream-with-opencv-on-android.6209/>
- [15] Opencv - *Android NDK camera support* (2021) - [Online]
<https://github.com/opencv/opencv/pull/19597>
- [16] Serge Platonov - *Respuesta en StackOverflow* (2021) - [Online]
<https://stackoverflow.com/questions/69157558/cannot-open-camera-using-cv2-videocapture0-in-android-phones/69445715#69445715>

ANEXOS

Anexo 1. Instrucciones de uso repositorio GitHub

Este repositorio contiene el código desarrollado durante mi Trabajo de Final de Grado. La memoria de dicho proyecto está disponible en el repositorio como MemoriaTFGJaimeJaumeBusquets y allí se explican detalladamente los pasos seguidos en torno a cada código, instrucciones para instalar el entorno, librerías y versiones necesarias, consejos aportados a raíz de la experiencia obtenida y mis propias conclusiones finales.

Como se puede observar, también se encuentran 3 carpetas disponibles que hacen referencia a cada uno de los capítulos trabajados durante el proyecto:

TFG

En TFG se encuentran todos los código y archivos referentes al entorno web. Movimientos básicos, transmisión de video, control desde teclado y toma de imágenes, mapeo del movimiento, reconocimiento facial y seguimiento, y por último, control del drone mediante las manos.

TFG2

En TFG2 todo lo relacionado con el modo Swarm. Cómo ejecutar el cambio a modo swarm, control del swarm desde teclado y desarrollo de diferentes interacciones entre dos drones (cuadrado, círculo, flips y modo rebote, el cual no funciona correctamente debido a la poca precisión en los movimientos del Tello EDU)

TFG3

Para terminar, en TFG3 se encuentran los archivos necesarios para generar la aplicación móvil, así como el archivo .apk para poder instalarla directamente. Cabe destacar que la aplicación móvil ha sido generada mediante Kivy y Buildozer, es decir, es una aplicación multiplataforma y en un PC también puede ser usada.

A causa de la incompatibilidad entre OpenCv y Android, las imágenes retransmitidas por el Tello EDU no están disponibles en la versión final de la app. Aún así, el código necesario para implementar dichas imágenes junto con la opción de aplicar el seguimiento facial, están disponibles en el código (comentadas) por si se desea trabajar en PC, donde si funciona correctamente.

Para cualquier duda, consulte la memoria donde está todo documentado.

