الـجـمـهـوريـة الـجـزائـريـة الـديـمـقـراطـيـة الـشـعـبـيـة
People's Democratic Republic of Algeria
وزارة الـتـعـلـيـم الـعـالـي والـبـحـث الـعـلـمـي
Ministry of Higher Education and Scientific Research
المـدرسـة الـوطـنـيـة العليا للـتـكـنـولـوجـيـات المـتـقـدمـة
Ecole Nationale Supérieure des Technologies Avancées

génie electrique et informatique industriel

Project Report to Obtain the Diploma of
Master

- Field -
Telecommunication

- Specialty -
Telecommunication Systems and Networking

- Subject -

Using the Tello Edu drone for educational purposes.

Realized by
Khaled Gasmi & Yasser Cherfaoui

Members of the Jury :

Mme Khadidja Zellat	Chair
Mme Djamila Bendouda	Examiner
Mme Nafissa Rezki	Examiner
Mr El Hadi Khoumeri	Supervisor

Algiers, Jun 25th 2024
Academic year 2023-2024

Dedication

Khaled Gasmi

My father, whose unwavering support fueled my determination throughout this journey.
My mother, whose spiritual encouragement provided unwavering strength.
My siblings, for their constant love and understanding.
To all my friends, especially those closest to me, for their unwavering camaraderie and belief in my abilities.
Dr. El Hadi Khoumeri, our supervisor, whose guidance and expertise were instrumental in shaping this project.
To all those dear to me and to everyone who, near or far, offered their support in countless ways.

Yasser Cherfaoui

To my beloved parents, whose unwavering support and encouragement have been my greatest strength throughout this journey. Their love and sacrifices have laid the foundation for all my achievements.
To my esteemed mentor, Mr. El-Hadi Khoumeri, I extend my deepest gratitude. Your guidance, wisdom, and unwavering belief in my abilities have been invaluable.
To my dear friends, Ilyas Brahmi and Mohamed Amine Djaballah, thank you for your constant companionship, support, and encouragement. Your friendship has been a source of great motivation and joy.

Acknowledgment

We would like to express our deepest gratitude to everyone who has supported us throughout the development of this thesis. This work would not have been possible without the guidance, encouragement, and assistance of many individuals.
First and foremost, we would like to thank our parents for their unwavering support and love. Their trust in us has been a constant source of motivation and strength, allowing us to pursue our goals with confidence and determination. We are forever grateful for their sacrifices and belief in our abilities.
We are immensely grateful to our supervisor, Mr El Hadi Khoumeri, for their invaluable guidance, insightful advice, and continuous encouragement throughout this project. Their expertise and dedication have been instrumental in shaping this work, and their constructive feedback has significantly improved the quality of this thesis. Thank you for being a constant source of inspiration and for believing in our vision.
A special thank you goes to our friends, who have stood by us since the beginning of this journey. Your support, encouragement, and understanding have been vital in helping us overcome the challenges we faced. We are deeply thankful for your companionship and for always being there to share both the highs and lows of this endeavor.
This thesis presents the development of a user interface to monitor drone states and notify users of smoke, fire, or human presence in forests. It is the culmination of hard work, perseverance, and the collective support of those mentioned above. We are sincerely grateful to each one of you for your contributions to this project.

Abstract

Conducting a comprehensive review of the use of the DJI Tello drone in educational projects. Exploring the various possibilities offered by this drone for the development of learning in computer programming and artificial intelligence.

Résumé

Faire un état de l'art complet sur l'utilisation du drone DJI Tello dans des projets éducatifs. Explorer les différentes possibilités offertes par ce drone pour le développement de l'apprentissage en programmation informatique et en Intelligence Artificielle.

Contents

List of Figures

List of Tables

Introduction

The advent of drone technology has revolutionized various industries, from entertainment and photography to agriculture and logistics. Among these technological marvels, the Tello EDU drone stands out for its versatility and educational value. Developed by Shenzhen Ryze Technology in collaboration with DJI and Intel, the Tello EDU drone is designed specifically to provide an engaging learning platform for programming, robotics, and STEM (Science, Technology, Engineering, and Mathematics) education.
This thesis explores the capabilities and applications of the Tello EDU drone, focusing on its integration into educational environments and the development of an Android application for real-time drone control. By leveraging the drone's advanced features and user-friendly design, we aim to enhance the learning experience for students and educators alike.
Chapter 1 delves into the technical characteristics of the Tello EDU drone, detailing its components, specifications, and operational capabilities. This chapter also highlights the drone's educational applications, illustrating how it can be used to teach programming and conduct various educational projects.
Chapter 2 transitions to the practical aspect of drone control, guiding the reader through the development of an Android application that transforms an Android device into a remote control for the Tello drone. The chapter covers the entire development process, from setting up the project in Android Studio to implementing the user interface using Jetpack Compose. It also includes the creation of a custom library to manage drone operations and the decoding of video streams for real-time monitoring.
The subsequent sections focus on the integration of the UI with the library using the MVVM (Model-View-ViewModel) design pattern. This approach ensures a clean and maintainable code structure, facilitating future enhancements and scalability. Detailed explanations of ViewModel and UI implementations are provided, showcasing the methods used to control the drone and display its status on the application interface.
This thesis not only highlights the technical and educational potential of the Tello EDU drone but also provides a comprehensive guide for developing a functional and scalable drone control application. By bridging theoretical knowledge and practical application, this work aims to contribute to the fields of drone technology and education, inspiring further innovation and exploration.

Chapter 1

Tello drone

1.1 Introduction

The Tello EDU drone [1], developed by Shenzhen Ryze Technology in collaboration with DJI and Intel, is designed specifically for educational purposes. Its compact design, ease of use, and advanced features make it an ideal tool for teaching programming, robotics, and STEM (Science, Technology, Engineering, and Mathematics) concepts. The Tello EDU drone supports multiple programming environments and languages, allowing educators to tailor their teaching to various skill levels and subjects. The drone's capabilities enable students to engage in hands-on learning, fostering creativity, problem-solving, and technical skills.

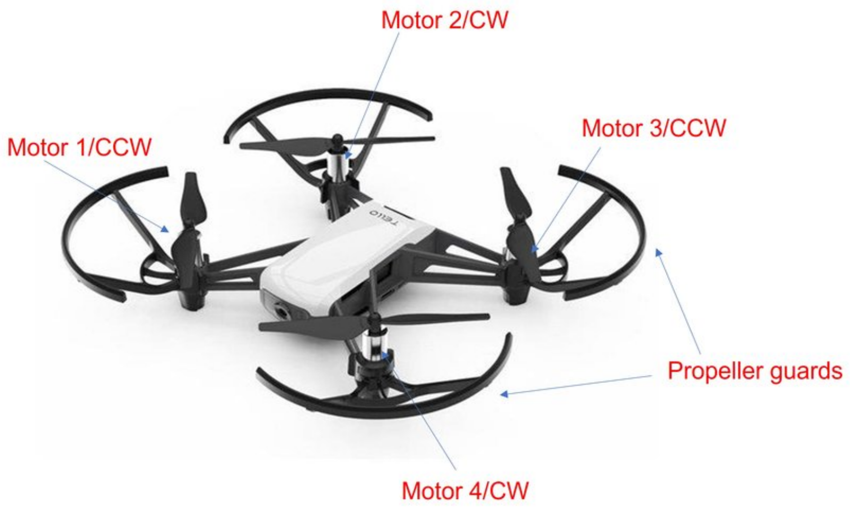


Figure 1.1: tello drone

1.2 Technical Characteristics of Tello EDU Drone

1.2.1 Components and Specifications

The Tello EDU drone is compact and lightweight, with dimensions of 98×92.5×41 mm and a weight of 87 grams. It features an integrated camera capable of 720p video streaming and 5

MP photos, making it suitable for various educational tasks involving image processing and computer vision. The drone is equipped with a telemetric sensor, barometer, LED, vision positioning system, and Wi-Fi connectivity, enhancing its operational capabilities [2].

Component	**Description**
Propellers	3 inches
Engines	Brushless motors
Drone Status Indicator	LED
Camera	720p HD video, 5 MP photos
Power Button	On/Off switch
Antennas	Wi-Fi connectivity
Vision Positioning System	Infrared and camera-based
Battery	1100 mAh, 3.8 V LiPo
Micro USB port	For charging
Propeller protectors	For safe indoor use

Table 1.1: Description of Tello EDU Components

1.2.2 Operational Characteristics

The Tello EDU drone is designed primarily for indoor use. It has a maximum flight distance of 100 meters, a maximum speed of 8 m/s, and a flight time of approximately 13 minutes on a single charge. The maximum flight height is 30 meters, providing ample range for most educational activities.

Characteristic	**Specification**
Maximum Flight Distance	100 meters
Maximum Speed	8 m/s
Maximum Flight Time	13 minutes
Maximum Flight Height	30 meters

Table 1.2: Operational Characteristics

1.3 Educational Applications of Tello EDU Drone

The Tello EDU drone offers a versatile platform for a wide range of educational and technical applications. Its integration into curricula enhances learning experiences in both programming and hands-on project development.

1.3.1 Programming and Coding

Introduction to Drone Programming:

Programming the Tello EDU drone provides an engaging way to learn coding. Learners write code to control the drone's movements and functions, including sending commands, receiving data, and processing feedback to ensure smooth and responsive operation.

Core Programming Skills Development:

Working with the Tello helps to develop essential programming skills. They learn algorithmic thinking and problem-solving, and gain a practical understanding of computational concepts such as loops, conditionals, and functions.

Implementing Control Functions:

We can program the drone for basic controls such as takeoff, landing, and directional movements, as well as advanced maneuvers including rotations, flips, and autonomous missions.

1.3.2 Educational Projects

Obstacle Courses:

Students can design and program the drone to navigate through obstacle courses, applying concepts of pathfinding and sensor integration. This hands-on activity reinforces understanding of spatial awareness and real-time decision-making.

Search and Rescue Simulations:

Simulating search and rescue operations with the Tello EDU drone teaches us about autonomous navigation and the integration of various sensors and technologies. This application underscores the importance of precision and reliability in programming.

Environmental Monitoring:

The drone can be programmed for environmental monitoring tasks, such as surveying areas or collecting data on specific parameters. This project highlights the drone's capability in data collection and environmental science, fostering interdisciplinary learning.

1.4 Individual Drone Applications

The Tello EDU drone provides numerous opportunities to engage in both basic and advanced programming tasks. These activities are designed to enhance the understanding of coding, control systems, and innovative applications.

1.4.1 Basic Missions

Programming Environment:

We can program the Tello EDU drone using a variety of operating systems (Windows, Linux, macOS) and programming languages (Python, Scratch, kotlin, Java, C++). This flexibility allows learners to use tools they are comfortable with or to experiment with new environments.

Simple Missions

In basic missions, we program the drone to perform straightforward tasks. For example, they might write code for the drone to take off, fly in a square pattern, and then land. These simple exercises helps to grasp the fundamentals of drone control and the basics of programming, including writing and debugging code, as well as understanding how different commands translate into drone movements. the following python code do these simple missions:

```python
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
drone_add = ('192.168.10.1', 8889)
sock.bind(('', 9000))
while True:
        try:
                msg = input('')
                if not msg:
                        break
                if 'end' in msg:
                        sock.close()
                        break
                msg = msg.encode()
                sent = sock.sendto(msg, drone_add)
        except Exception as err:
                print(err)
                socket.close()
                break
```

And we can receive its state using this python code:

```python
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('', 8890))
while True:
        try:
                data, server = sock.recvfrom(1024)
                print(data.decode())
        except Exception as err:
                print(err)
                socket.close()
                break
```

1.5 Swarm Applications

1.5.1 Swarm Definition:

A swarm comprises multiple drones collaborating to perform collective tasks, inspired by the natural behaviors of bees or birds. This collaborative approach enhances efficiency in applications such as search and rescue operations or environmental monitoring, allowing for comprehensive area coverage.

1.5.2 Environment Setup:

Setting up a swarm involves configuring multiple drones to connect to a central control system as it is shown in the figure 1.2, often facilitated by a router. Each drone operates in coordination with the others, executing commands synchronously to achieve the desired collective behavior.

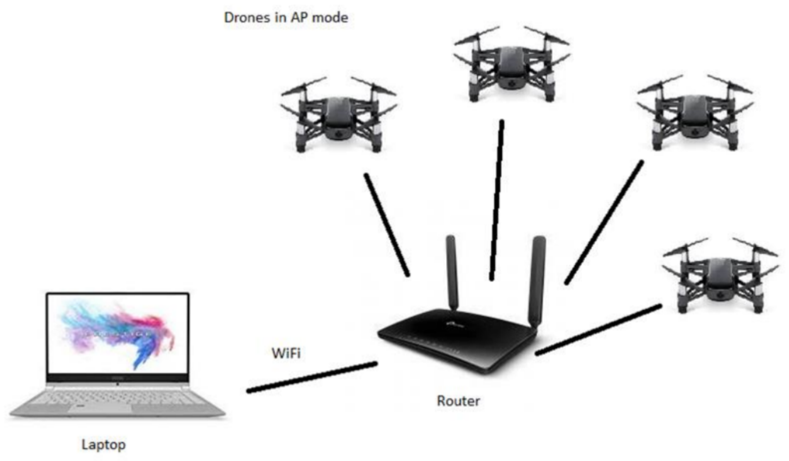


Figure 1.2: swarm environment

1.5.3 Swarm Missions:

Swarm missions extend the principles of individual drone missions to a coordinated group. These missions include synchronized takeoff, precise movement patterns, and coordinated landing, demonstrating the power of collective operation.
Advanced applications of swarm technology involve real-time control through a graphical interface, enabling users to issue commands to the entire swarm or to individual drones. This capability supports complex tasks such as detailed mapping, comprehensive surveillance, and intricate coordinated maneuvers, such as drone "dances."

1.6 AI Applications with Tello EDU Drone

Implementing AI in Tello drone applications is remarkably accessible, thanks to its user-friendly design and extensive support for educational purposes. The Tello drone is compatible with various programming environments allowing both beginners and advanced users to experiment with AI algorithms seamlessly. The availability of comprehensive SDKs facilitates the integration of AI functionalities, enabling developers to easily implement features like autonomous flight, object recognition, and real-time data processing. Additionally, Tello's lightweight and portable design, coupled with its affordability, makes it an excellent tool for schools and universities looking to incorporate AI and robotics into their curriculum. The drone's built-in sensors and camera provide a rich set of data for AI applications, making tasks like computer vision and machine learning projects straightforward. Overall, the combination of intuitive software, robust hardware, and extensive community support makes the implementation of AI in Tello drone applications both straightforward and highly educational.

Chapter 2

Android App to control tello

2.1 Introduction

In previous discussions, we explored the educational potential of the Tello drone and various practical applications it offers. Now, it's time to delve into a hands-on example of programming the Tello drone and controlling it in real-time.
This chapter will guide you through the development of an Android application that transforms our Android phone into a remote control for the Tello drone. This app will manage all the drone's movements and receive live video streams, enabling control even when the drone is out of sight.
We will cover the key concepts of Android development, incorporating the latest technologies such as Jetpack Compose for the user interface instead of traditional XML files. Topics will include design patterns, UI creation with Jetpack Compose, and writing logic code in Kotlin to communicate with the drone and decode the video stream. This project serves as an excellent introduction to both Android development and Tello drone control.

2.2 Android Studio

Android Studio [3] is the official integrated development environment (IDE) for Android app development, endorsed by Google. It offers a comprehensive suite of tools and features specifically designed to streamline the development process. With its powerful code editor, intuitive layout editor, and robust debugging tools, Android Studio provides a seamless experience for developers. It supports the latest Android APIs and libraries, integrates with version control systems, and offers features like intelligent code completion and real-time error checking. These capabilities make Android Studio the best IDE for creating high-quality, efficient Android applications.
We are utilizing the latest stable release of Android Studio, codenamed "Jellyfish," which was launched in 2023. This version brings enhanced performance, new features, and improved stability.

2.3 App Development

To develop our android application [4], we first need to define the functionalities we aim to integrate, ensuring a seamless user experience. Our primary objective is to enable live control of the Tello drone using an Android device. The key functionalities identified are as follows:

- Button for initiating communication
- Buttons for takeoff and landing
- Joystick for forward, backward, left, and right movement
- Joystick for up, down, and left and right rotation around the Z-axis
- Dropdown list for toggling the video stream on or off
- Joystick for executing drone flips
- Indicators for displaying drone status, including battery level, temperature, altitude, connection status, and streaming status

These functionalities were chosen based on their relevance and necessity for effective drone control, as supported by previous research.
After identifying the required functionalities, we used Figma, a widely recognized tool for UI/UX design, to visualize and refine our application layout. Through iterative design processes, we achieved a design that is both intuitive and user-friendly, thereby enhancing the overall user experience 2.1.

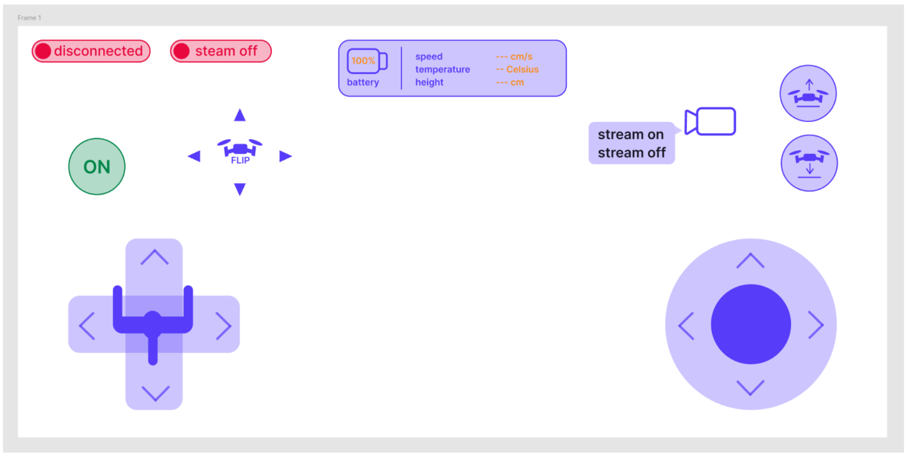


Figure 2.1: Figma design for the application

In addition to the layout, we designed custom icons specifically tailored to our application's needs. These icons include the joystick knob for up and down commands, as well as icons for drone takeoff and landing, ensuring a cohesive and visually appealing interface 2.2.

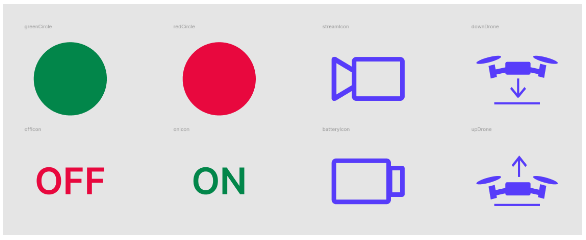


Figure 2.2: Custom Figma icons for the application

The implementation of these elements was guided by best practices in UI/UX design and Android development, ensuring that the application is not only functional but also provides a superior user experience.

2.4 Initiating the Project on Android Studio

In Android Studio Jellyfish, we start by creating a new project and selecting an empty Jetpack Compose project template. Kotlin is chosen as the programming language over Java, with the minimum API level set to 24 and the maximum API level to 34.
Upon project creation, we consider the design pattern and package hierarchy. Although the project is relatively small, containing only one activity, and may not necessitate rigorous adherence to design patterns, it is prudent to adopt a sophisticated and well-organized code structure. This approach ensures maintainability and scalability, particularly if the project is expanded or made public on GitHub. Writing clean, well-structured code is beneficial for other developers who might clone the repository and for future enhancements.
To develop our app, we chose the most popular design pattern among the Android community, which is MVVM (Model-View-ViewModel). The MVVM pattern is distinguished by its components:

- **Model:** Represents the data and business logic of the application. It is responsible for retrieving and storing data, usually from a database or a web service. The Model is independent of the user interface.
- **View:** Represents the user interface of the application. It displays the data provided by the ViewModel and sends user actions (like button clicks) to the ViewModel.
- **ViewModel:** Acts as an intermediary between the Model and the View. It holds the presentation logic, including data binding and commands that the View can call. The ViewModel exposes the data and commands to the View in a way that is easy to bind.

In our application, the Model is the library we created to communicate with the drone, the View is the user interface logic, and the ViewModel consists of the functions that the UI can call to perform actions within the application.

2.5 Coding the UI

In the process of coding the Figma UI design (see Figure 2.1), we chose Jetpack Compose [5]. Jetpack Compose is a modern toolkit for building native Android UIs using a declarative approach. It simplifies UI development by allowing us to describe the UI in Kotlin code, which is then rendered dynamically. This approach reduces boilerplate code, promotes a more intuitive and flexible UI design, and enables real-time UI updates and previews. It offers easier state management, interoperability with existing Android views, and a streamlined API that enhances productivity and code maintainability. Overall, Jetpack Compose accelerates development and improves the quality of our Android applications.
In the UI, the main components is the joysticks. Since there is no predefined joystick component, we had two options: use someone else's code from GitHub or code it ourselves. Given that this project is for educational purposes, we decided to code it ourselves. One way to achieve this is by using the Canvas API of the Jetpack Compose library. We draw the base as a circle, add the directions, and draw the knob. We make it movable and save its position in a mutable state for later use.

```kotlin
var knobPosition by remember { mutableStateOf(Offset.Zero) }
var isDragging by remember { mutableStateOf(false) }
```

Additionally, we save the state of whether the knob is being dragged or not. The code for the knob is as in the code snippet below. Instead of drawing the knob directly, we set it as an image because we are using complex knobs, like a drone icon or an airplane control handle, which would be difficult to draw directly on the canvas.

```
private fun DrawScope.drawJoystickKnob(position: Offset, knobImage:
↪  ImageBitmap, joyStickSize: Int) {
        drawImage(knobImage, Offset((joyStickSize+40)/2f,
↪  (joyStickSize+40)/2f) + position)
}
```

If we leave the knob like this, it will move all around the screen, which is not desirable. Therefore, we set limitations on its movement. In our case, it is useful to have the coordinates in polar coordinates because the radius can represent the strength (e.g., speed of the drone), and the angle represents the direction. We created a function to perform this transformation from Cartesian coordinates.

```
private fun Offset.getDistance(other: Offset): Float {
        return kotlin.math.sqrt((x - other.x).pow(2) + (y - other.y).pow(2))
}

private fun Offset.getAngle(other: Offset): Float {
        return kotlin.math.atan2((y - other.y).toDouble(), (x -
↪  other.x).toDouble()).toFloat()
}
```

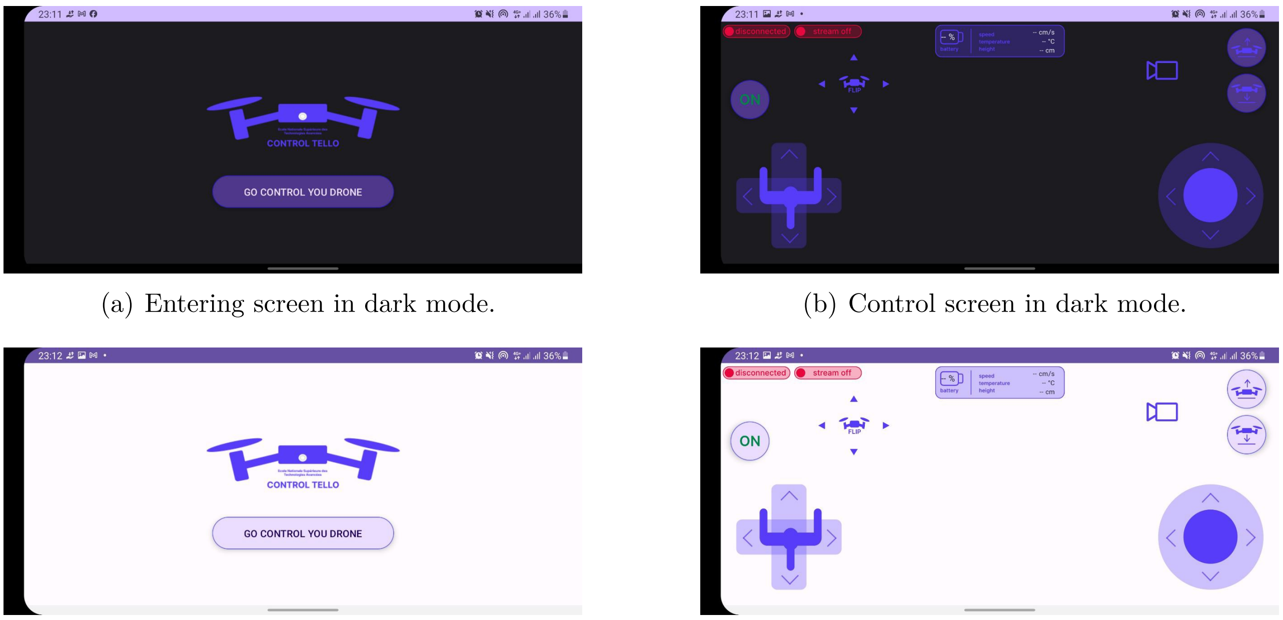
We organized this code in a component folder so we can call it from the main UI file and reuse it for all the joysticks in the application by simply changing the knob and base shape. This is one of the advantages of Jetpack Compose, as opposed to using XML, where we would have to copy and paste the code whenever needed.
For other components, we used simple components already available in the Compose library. Additionally, we used Material UI for beautiful and functional components, such as floating action buttons, images, texts, and drop-down menus. These implementations are relatively straightforward. The final results are shown in Figure 2.3.

2.6 Developing a Library to Control Tello Drone

In our project, we developed a Kotlin library specifically designed to control the Tello drone, aimed at enhancing its utility in educational applications. This library offers a comprehensive suite of functions to manage various drone operations, encompassing nearly all the functionalities provided by the Tello SDK [6]. While our application leverages only a subset of these features to meet our specific needs, the library is versatile and can be utilized by other developers or learners to create a wide range of applications. This makes it an invaluable resource for anyone looking to explore and innovate with the Tello drone in and Android app.

2.6.1 Library Overview

The Tello control library offers a comprehensive suite of functionalities designed to facilitate seamless interaction with the Tello drone. This robust library supports a range of capabilities, including command-based control, state reading, and real-time video streaming [2].



(a) Entering screen in dark mode.
(b) Control screen in dark mode.
(c) Entering screen in light mode.
(d) Control screen in light mode.

Figure 2.3: App UI results in light and dark modes.

This class provides an array of methods to manage various tasks such as establishing a connection with the drone, issuing commands for flight operations, and handling video streams efficiently. By encapsulating these core functionalities within a user-friendly file, the Tello control library simplifies the process of drone control and enhances the overall user experience.

Connection Handling

The library enables seamless connection to the Tello drone via a WiFi network. The `connect` method establishes a connection to the drone's command interface.
The drone's IP address is `192.168.10.1`, and commands can be sent through port 8889. To switch the drone to SDK mode, we send the `"command"` instruction. Once the drone receives this command, it will execute any subsequent commands. The following functions demonstrate how to establish the connection and prepare the Tello drone to execute commands.

```kotlin
@Throws(IOException::class)
fun connect(ip: String = "192.168.10.1", port: Int = 8889) {
        socket = DatagramSocket(port)
        socket.connect(InetAddress.getByName(ip), port)
        sendCommand("command")
}
```

Additionally, the `stateConnect` method connects to the state interface through port 8890 to receive the state of the Tello drone, such as battery level, altitude, speed, and more. This function will receive the data as follows:

```
pitch:%d;roll:%d;yaw:%d;vgx:%d;vgy:%d;vgz:%d;templ:%d;temph:%d;tof:%d;
h:%d;bat:%d;baro:%.2f;time:%d;agx:%.2f;agy:%.2f;agz:%.2f;\r\n
```

Command Execution

The library provides a set of methods to control the drone, with the most important being the sendCommand method, as most other methods in the library rely on it. This method takes a command string, checks if the command is not empty, and the binding state of the command-sending socket, encodes the command, and sends it to the drone. All of this logic is handled within a try-catch block for error handling.

```
@Throws(IOException::class)
fun sendCommand(command: String): String {
        try {
                if (command.isEmpty()) return "Empty command."
                if (!socket.isConnected) return "Socket Disconnected."

                val sendData = command.toByteArray()
                val sendPacket = DatagramPacket(sendData, sendData.size,
 ↪  socket.inetAddress, socket.port)
                socket.send(sendPacket)

                ...

                return response
        } catch (e: Exception) {
                Log.d("exception in sendCommand", "send command exception $e")
                return "$e"
        }
}
```

Real-time Video Streaming

One of the key features of the library is its ability to receive and process the video stream from the drone. The `receiveStream` method handles the video stream and decodes it using Android's `MediaCodec`.
The drone transmits the video stream via the `UDP` protocol on port 11111. The video is sent as a series of binary data, which must be decoded to obtain the desired frames for display. This process will be explained in more detail in later sections. Below is a simplified version of the function:

```
@Throws(IOException::class)
fun receiveStream(port: Int = 11111) {
        // Setup MediaCodec and DatagramSocket for video streaming
        val format =
 ↪  MediaFormat.createVideoFormat(MediaFormat.MIMETYPE_VIDEO_AVC, 960, 720)
        // Setup format and codec
        m_codec =
 ↪  MediaCodec.createDecoderByType(MediaFormat.MIMETYPE_VIDEO_AVC)
        m_codec.configure(format, null, null, 0)
        m_codec.start()

        // Receiving video stream
        videoSocket = DatagramSocket(null)
        videoSocket.reuseAddress = true
        videoSocket.broadcast = true
        videoSocket.bind(InetSocketAddress(port))
        val message = ByteArray(2048)
        while (true) {
                val videoPacket = DatagramPacket(message, message.size)
                videoSocket.receive(videoPacket)
                videoBytes.postValue(videoPacket.data)
                // Further processing and decoding
        }
}
```

2.6.2 Command Set

The library supports a wide range of commands for controlling the drone, including basic flight commands such as takeoff and land, as well as more complex maneuvers like flips and rotations. These commands are executed using the `sendCommand` function, which encapsulates all the necessary logic to communicate with the Tello drone. By defining these commands within the library rather than in the ViewModel, we ensure scalability and flexibility. This design allows for the development of higher-level functions that can, for example, execute a sequence of commands autonomously. Leveraging Kotlin's functional programming capabilities, we can create sophisticated control routines that meet the requirements of various educational and research applications.

```
@Throws(IOException::class)
fun takeOff() = sendCommand("takeoff")

@Throws(IOException::class)
fun land() = sendCommand("land")

@Throws(IOException::class)
fun flip(direction: FlipDirection) = sendCommand("flip
 ↪  \${direction.direction}")
```

2.6.3 Utility Functions

To ensure robust command validation, the library includes several utility functions to check the validity of command arguments. These functions help maintain the integrity of the commands sent to the drone.

```
private fun Int.isValidDistance() = this.toMetric() in distanceRange
private fun Int.isValidSpeed() = this in speedRange
private fun ArrayList<Int>.isValidDistance() = this.all {
 ↪  it.isValidDistance() }
```

2.7 Decoding Video Stream

the Tello drone encodes the camera frames in H.264 format before broadcasting them. Therefore, it is necessary to decode the video stream upon receiving it to retrieve and display the frames. Naturally, the decoding process must use the same H.264 format used for encoding.

2.7.1 H.264

H.264 [7] is a widely used video compression standard known for its high compression efficiency and broad compatibility. It reduces the size of video files while maintaining quality, making it ideal for streaming and storage. Decoding H.264 involves interpreting and decompressing the encoded video data using Sequence Parameter Set (SPS) and Picture Parameter Set (PPS) headers, which provide essential information about the video stream's configuration and individual frames. This process enables playback of high-quality video on various devices and platforms.

2.7.2 Decoding the Frames Received

In this section, we delve into the `receiveStream()` function to understand its components and how it decodes the received video stream. This process requires some external libraries, which we import using the `gradle` build system. We add these dependencies and synchronize gradle to save the changes and import the libraries. The required libraries are `media`, `jcodec`, and `exoplayer`. The following lines of code are added to the `build.gradle` file:

```
implementation("androidx.media:media:1.7.0")
implementation("org.jcodec:jcodec:0.2.3")
implementation("com.google.android.exoplayer:exoplayer-core:2.15.1")
```

First, we define the headers of the H.264 format, specifically the SPS and PPS. We then declare the format using `MediaFormat`, specifying characteristics of our stream such as height, width, frame rate (FPS), and resolution. The following code snippets illustrate this:

```
val headerSps = byteArrayOf(0, 0, 0, 1, 103, 77, 64, 40, -107, -96, 60, 5,
↪  -71)
val headerPps = byteArrayOf(0, 0, 0, 1, 104, -18, 56, -128)
val format = MediaFormat.createVideoFormat(MediaFormat.MIMETYPE_VIDEO_AVC,
↪  960, 720)
```

After setting these video specifications, we initialize the `MediaCodec` and place it within a try-catch block to handle IO exceptions:

```
try {
        m_codec =
↪  MediaCodec.createDecoderByType(MediaFormat.MIMETYPE_VIDEO_AVC)
        m_codec.configure(format, null, null, 0)
        startMs = System.currentTimeMillis()
        m_codec.start()
} catch (e: IOException) {
        e.printStackTrace()
        return
}
```

Next, we create a list of `byteArray` to store the bytes received from the datagram socket, which is used to receive the stream on port 11111. We open a while loop to continually capture the UDP stream. The loop condition is set to true, and we include a breaking condition to exit the loop if the streaming socket is closed.
We also write the logic to save the received `byteArray` until they accumulate to a certain amount necessary for processing. The following code processes the `byteArray` when it contains at least one frame:

```
if (len < 1460) {
        destPos = 0
        val data = output.toByteArray()
        output.reset()
        output.flush()
        output.close()
        val inputIndex = m_codec.dequeueInputBuffer(-1)
        if (inputIndex >= 0) {
                val buffer = m_codec.getInputBuffer(inputIndex)
                if (buffer != null) {
                        buffer.clear()
                        buffer.put(data)
                        val presentationTimeUs = System.currentTimeMillis() -
↪   startMs
                        m_codec.queueInputBuffer(inputIndex, 0, data.size,
↪   presentationTimeUs, 0)
                }
        }
}
```

In the above code:

- When a packet with a length less than 1460 bytes is received, it indicates the end of a frame.
- `destPos` is reset and the complete frame is processed.
- The frame data is copied into the codec's input buffer and queued for decoding.

After this, we extract the decoded frames:

```
val image = m_codec.getOutputImage(outputIndex)
```

Finally, we convert the frame into a bitmap format to display it on the screen:

```
val bm = imgToBM(image)
```

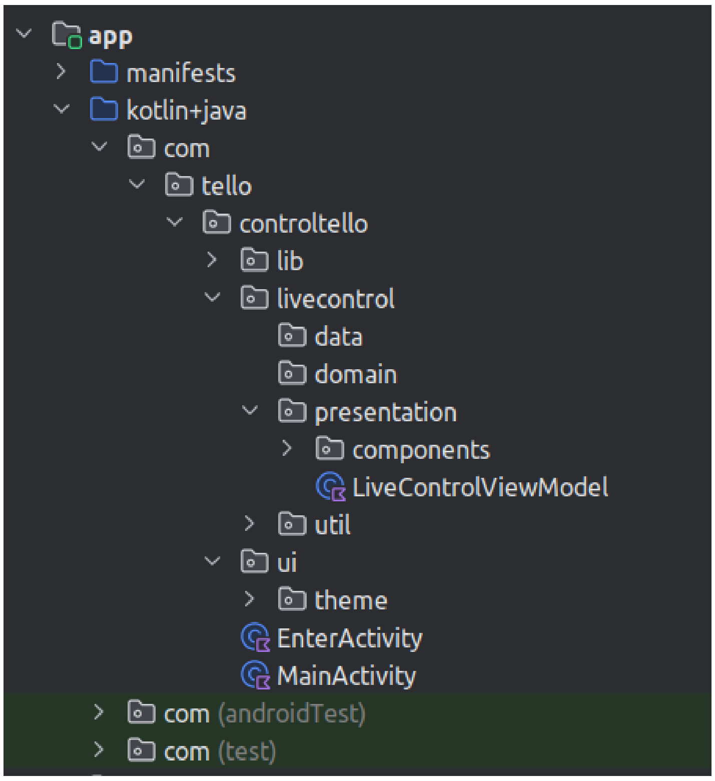
There is no predefined function for this transformation, so we create our own function. The source code for this function is provided on GitHub.
One problem we encounter is that when we receive the UDP stream, we want to save it live without downloading it to the phone's disk. This can be achieved using the `queue` data structure. A queue follows the First-In-First-Out (FIFO) principle, making it suitable for temporarily storing frames for display:

```
val bm = imgToBM(image)
try {
        if (!queue.isEmpty()) {
                queue.clear()
        }
        queue.put(bm)
} catch (e: InterruptedException) {
        e.printStackTrace()
}
```

We declare this queue as an argument to the function, allowing us to call the function as needed and pass the queue without any changes.

2.8 Binding the UI With Library Using View Model

In this section, we will discuss the binding of the UI and the library to make our application functional. As mentioned previously, we are using the MVVM design pattern to construct this application. Consequently, we have structured the packages as illustrated in Figure 2.4.



```
app
  manifests
  kotlin+java
    com
      tello
        controltello
          lib
          livecontrol
            data
            domain
            presentation
              components
              LiveControlViewModel
            util
          ui
            theme
            EnterActivity
            MainActivity
    com (androidTest)
    com (test)
```

Figure 2.4: Folders structure

Package structure is crucial in Android development. The way we organize them can vary based on our project requirements. A well-structured project can be highly scalable in the future, whether it involves numerous features or minimal expansion. In our application, we have two screens: an entry screen, which displays the app logo and a button to enter the control screen, and the control screen itself.

Since our project has potential for future scalability—such as adding auto-patrolling using mission pads or integrating a face recognition model for the drone to follow its user—we have chosen to organize our project according to features. Currently, we have one primary feature: controlling the drone. Within the feature package, we have created sub-packages for data (containing data logic), domain (containing business logic), and presentation (containing everything related to the UI, such as components and view models). Additionally, we included a utils package for other necessary classes related to the control feature. We also created a separate package called lib to store our library.

2.8.1 ViewModel Implementation

We created the ViewModel file and named it `LiveControlViewModel.kt`. This file is located in the presentation package of the control feature. In this file, we write functions necessary to control our drone and retrieve its state, such as checking whether it is connected or not and if it is streaming nor not.
First, since our library is a class, we need to create an instance of this class:

```
val tello = Tello()
```

Now we can use the `tello` object to access all the properties and methods of the class library. Let's examine the connectivity function:

```
fun connect() {
        viewModelScope.launch(Dispatchers.IO) {
                tello.connect()
                delay(1500)
                stateConnect()
                if (tello.isConnected) {
                        isConnected.postValue(true)
                }
        }
}
```

The above function uses the `connect()` method from our library. Although it may seem redundant, it is not. We use A in this function and post the connection state to a mutable A variable to visualize this state on the screen. First, let's clarify why we used coroutines and LiveData variables.
We are sending and receiving data from the drone while interacting with the UI. Since UI components run on the main thread, executing these communication processes on the main thread would cause temporary UI freezes, which is undesirable. Therefore, we use coroutines.
Creating and running a coroutine is straightforward. As shown in the `connect()` function code snippet, we use:

```
viewModelScope.launch(Dispatchers.IO) {}
```

In this declaration, we observe two things which are the A and the dispatcher. In this example, we chose `viewModelScope` and `Dispatchers.IO`. We selected `viewModelScope` to ensure that our function does not run outside the lifecycle of our ViewModel, which itself is related to the app's lifecycle. This prevents data leakage and performance issues due to long-running background processes. We chose `Dispatchers.IO` because it is optimized for IO-bound tasks, such as reading from or writing to files, accessing databases, and making network requests.
The function posts the connection state to a mutable LiveData variable, allowing us to observe the changes and display them on the screen later on when we need to. We also observe other mutable data, such as streaming state and Tello state:

```
val isConnected = MutableLiveData(false)
val telloStates = MutableLiveData<Map<String, String>>()
val isStreaming = MutableLiveData(false)
```

In Kotlin, MutableLiveData can hold any data type. For example, we have MutableLiveData for Boolean and another for a Map of strings.
We obtain the values for this map from the `receiveTelloState()` function, which captures the state sent from Tello as a string and transforms it into a map for easier data retrieval:

```
private fun stateConnect() {
        viewModelScope.launch(Dispatchers.IO) {
                while (true) {
                        val text = tello.stateConnect()
                        val keyValuePairs = text.split(";")
                        val sensorData: MutableMap<String, String> =
 ↪  mutableMapOf()
                        for (pair in keyValuePairs.indices) {
                                if (pair == keyValuePairs.lastIndex) break
                                val (key, value) =
 ↪  keyValuePairs[pair].split(":")
                                sensorData[key] = value
                        }
                        telloStates.postValue(sensorData)
                        delay(200)
                }
        }
}
```

In this function, we use the `stateConnect()` method from the library to receive the state as a string. We then perform some logic to transform the string into a map, as shown in the code snippet.

This overview highlights the work done in the ViewModel and our approach. All other functions follow a similar pattern, creating functions for commands like disconnect, stream on, stop stream, and take off...

2.8.2 UI implementation

In this section, we discuss how our UI responds to user actions, such as clicking a A or moving the joystick knob, and how the text representing the Tello's status updates accordingly.
Floating action buttons are straightforward to handle for click events because they have an onClick listener by default. Here is an example of how we handle the landing command using the floating action button:

```
FloatingActionButton(
onClick = {
        try {
                controlViewModel.land()
        } catch (e: Exception) {
                Log.d("main UI errors", "error found: \$e")
        }
}
)
```

However, things can get a bit more complex with the connect button, where we want the button to change its color and logo based on the connection status. This can be achieved by implementing a conditional statement to check the connectivity and set the appropriate color and logo accordingly.
To display the status, we implement an observer that monitors the LiveData and updates the UI based on the observed data:

```
val isConnected = observeLiveData(controlViewModel.isConnected).value ?: false
val isStreaming = observeLiveData(controlViewModel.isStreaming).value ?: false
```

The `observeLiveData()` function is not predefined; we created it to take LiveData as an argument and return a mutable state:

```
fun <T> observeLiveData(liveData: LiveData<T>): MutableState<T?>
```

When observing the state, we can change the necessary UI components using conditional statements or simple variable assignments.
For the joystick, we receive the knob's position in polar coordinates. To send a command to the drone, indicating how it should move, we use the command:

```
rc left/right forward/backward up/down yaw
```

The angle is provided in radians, and the axes are shown in Figure 2.5:
Based on the figure, we use the following conditional statements to construct the command `rc a b c d`:

```
if (angle <= 45 && angle > -45) rcs[0] = (r * cos(radAngle) * 0.5).toInt()
if (angle <= 135 && angle > 45) rcs[1] = (-r * sin(radAngle) * 0.5).toInt()
if (angle <= -45 && angle > -135) rcs[1] = (-r * sin(radAngle) * 0.5).toInt()
if (angle <= -135 || angle > 135) rcs[0] = (r * cos(radAngle) * 0.5).toInt()
try {
        controlViewModel.sendRc(rcs[0], rcs[1], rcs[2], rcs[3])
} catch (e: Exception) {
        Log.d("rc command error", "error in constructing the rc command")
}
```

Lastly, to display the video stream, we first create a variable for the queue:

```
val videoQueue = remember { ArrayBlockingQueue<Bitmap>(1) }
```

We then call the ViewModel function to display the bitmap images, passing the queue as shown:

```
VideoScreen(controlViewModel, videoQueue = videoQueue)
```

This overview covers the main components used in this project. Although we have not discussed the dropdown menu or the layout structure (such as columns and rows), all UI code can be found in the GitHub repository.

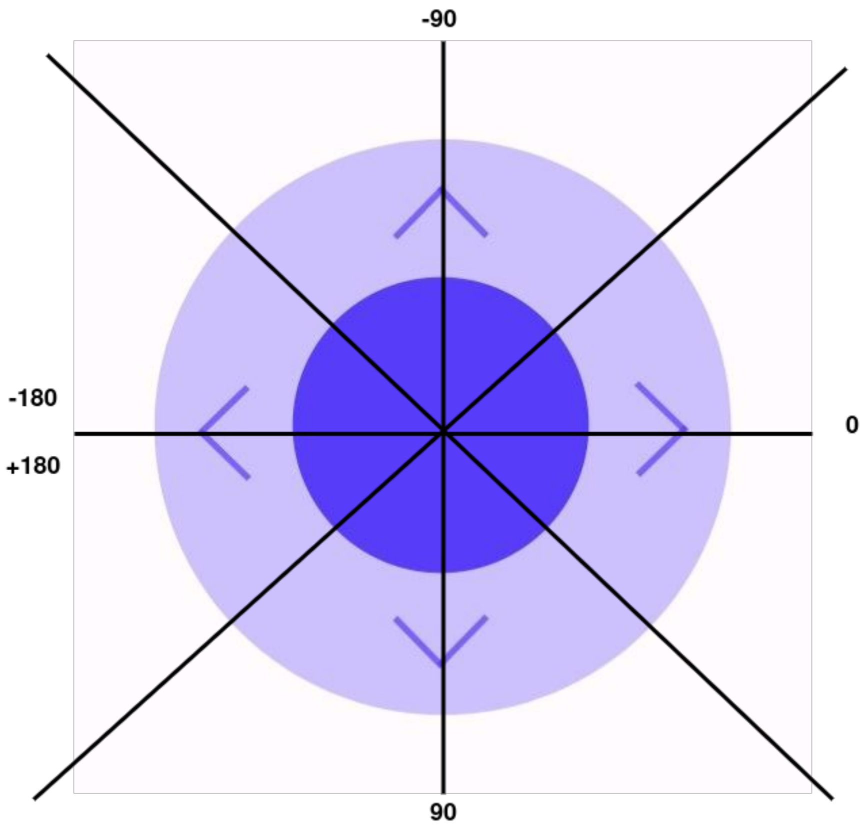


Figure 2.5: Joystick angles

2.9 App Testing and Validation

Upon completing the application development, we installed it on our device and conducted thorough testing to ensure proper functionality. The UI was smooth, and all components operated correctly.
We then connected the application to our Tello drone, which successfully established a connection. We toggled the streaming button, and the video stream functioned as expected. The drone was capable of performing all expected movements, including directional movement and flips. However, we encountered an issue with the forward flip; the drone was unstable and crashed into the floor during attempts to flip forward.
Another issue identified was with the joystick knob, which is placed on the canvas as an image. This design choice made it non-scalable with the device screen, potentially causing problems on devices with different screen resolutions. We plan to address and fix this issue in a future update.
To validate the project's performance, we provide a series of images showcasing the application in its working state (Figures 2.6 - 2.7 - 2.8 - 2.9).

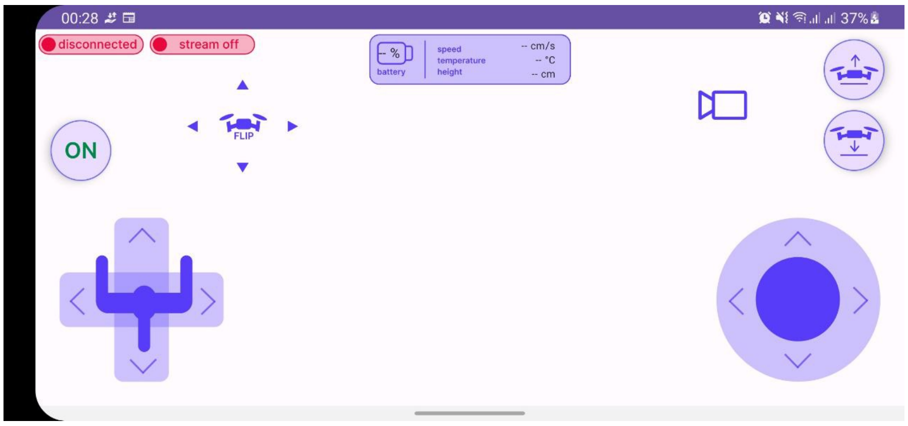


Figure 2.6: Launching the App

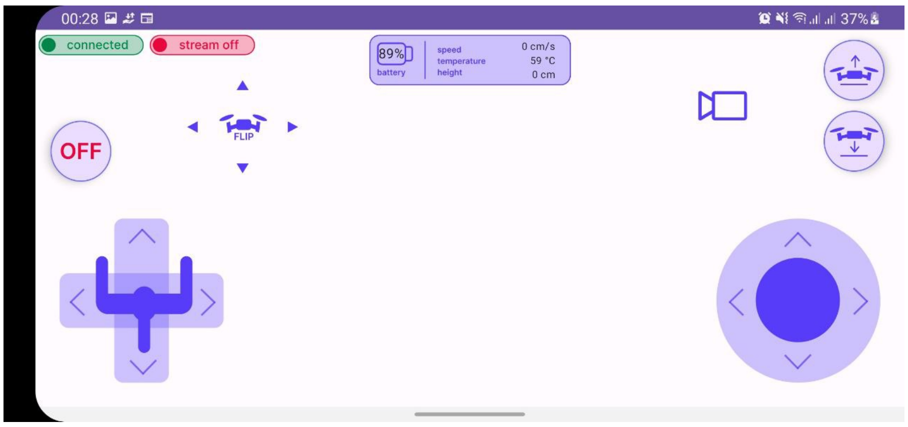


Figure 2.7: Connection the app to drone

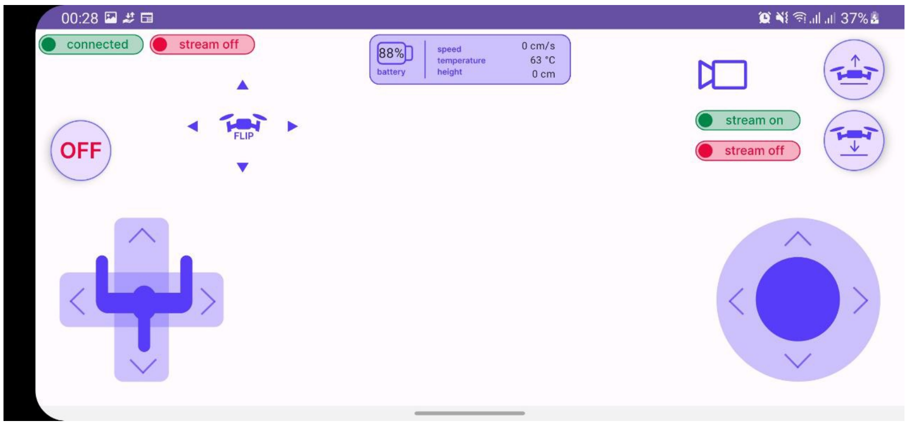


Figure 2.8: Clicking the stream drop down menu

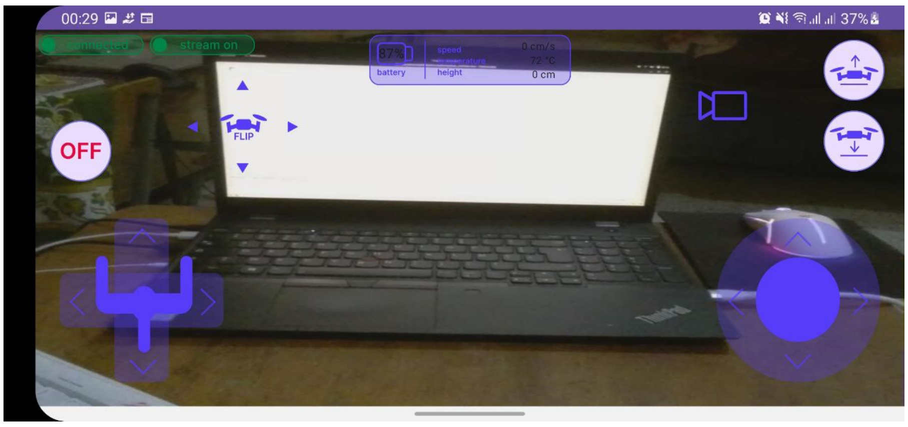


Figure 2.9: Drone streams the video

Conclusion

This thesis explores the multifaceted capabilities of the Tello EDU drone and the development of an Android application for real-time control, showcasing its potential in educational and practical applications. The Tello EDU drone, with its advanced features and ease of use, serves as an excellent tool for teaching programming, and various STEM concepts. Our project demonstrates how this drone can be integrated into an Android application to enhance the learning experience and provide hands-on engagement with cutting-edge technology.
Through the development and testing phases, we established a robust connection between the application and the drone, ensuring smooth operation and responsiveness. The application successfully manages the drone's movements, captures live video streams, and provides an intuitive user interface for real-time control. Despite encountering minor issues, such as instability during forward flips and scalability concerns with the joysticks, the overall functionality and performance of the application validate its effectiveness and potential for future enhancements. This work contributes to the fields of drone technology and education by offering a comprehensive guide for developing scalable drone control application. By bridging theoretical knowledge with practical implementation, we aim to inspire further innovation and exploration.

Appendix A

Android concepts definition

A.1 Coroutines

Coroutines in Android provide a modern and efficient approach to asynchronous programming, offering more readable and maintainable code compared to traditional threads. They allow for sequential code execution while handling asynchronous tasks, simplifying the complexity typically associated with callbacks. Unlike threads, which are heavier and resource-intensive, coroutines are lightweight and can be suspended and resumed without blocking the main thread. This enables coroutines to manage a higher number of concurrent tasks without the performance overhead of creating and managing multiple threads.

A.2 LiveData data type

LiveData variable is a lifecycle-aware, observable data holder that enables UI components to update automatically when the underlying data changes. As part of Android's Architecture Components, LiveData manages UI-related data in a lifecycle-conscious manner. By observing LiveData, activities and fragments can respond to data changes without risking memory leaks or crashes due to stopped activities. LiveData ensures updates are sent only to active observers, making it a safe and efficient choice for updating the UI in response to data changes. This results in a cleaner architecture and more responsive user interfaces.

A.3 Scope

In Android development, a scope specifies the context in which certain operations can be executed and dictates their lifecycle. Scopes are especially crucial when working with Kotlin coroutines, as they help manage coroutine execution in alignment with the lifecycle of components like activities, fragments, and view models.

A.4 Floating action button

A Floating Action Button (FAB) is a prominently displayed button that hovers above the interface of an Android application, providing quick access to a primary action or feature.

A.5 Activity

An Android activity is a distinct, focused operation that a user can engage in. It acts as the primary entry point for user interaction, typically displaying a user interface for a specific task. Each activity is an instance of the Activity class and operates within a defined lifecycle managed by the Android operating system, encompassing states like creation, start, resume, pause, stop, and destruction. Activities are essential components of an Android application, facilitating user interactions and navigation across various parts of the app.

A.6 Manifest file

The Android Manifest file, `AndroidManifest.xml`, is a vital XML document found in the root directory of every Android application. It supplies the Android operating system with critical information about the application, such as its components (activities, services, broadcast receivers, and content providers), required permissions, utilized or needed hardware and software features, and other important metadata. Additionally, the manifest file specifies the application's package name, version details, and minimum and target API levels. It plays a crucial role in defining the structure and behavior of an Android application.

A.7 Socket

A socket serves as one endpoint in a bidirectional communication link between two programs operating on a network. It is associated with a port number, enabling the TCP layer to identify the target application for incoming data.

Appendix B

Layout in jetpack compose

In Android development using Jetpack Compose, layout management and placement of UI elements are key to building responsive and dynamic applications. Jetpack Compose offers various layouts like `Column`, `Row`, and `Box`, along with powerful modifiers to customize the appearance and behavior of UI components.

B.1 Column Layout

A `Column` layout arranges its children vertically. Each child is placed one below the other. This is useful for creating vertical lists of items or stacking elements vertically.

```
Column(
modifier = Modifier.fillMaxSize(),
verticalArrangement = Arrangement.Center,
horizontalAlignment = Alignment.CenterHorizontally
) {
        Text("Item 1")
        Text("Item 2")
        Text("Item 3")
}
```

B.2 Row Layout

A `Row` layout arranges its children horizontally. Each child is placed next to the previous one. This is useful for creating horizontal lists of items or aligning elements side by side.

```
Row(
modifier = Modifier.fillMaxWidth(),
horizontalArrangement = Arrangement.SpaceAround,
verticalAlignment = Alignment.CenterVertically
) {
        Text("Item 1")
        Text("Item 2")
        Text("Item 3")
}
```

B.3 Box Layout

A `Box` layout allows for stacking elements on top of each other. This is useful for creating overlays or combining multiple elements in a single space.

```
Box(
modifier = Modifier.fillMaxSize()
) {
        Image(painterResource(R.drawable.image), contentDescription = null)
        Text("Overlay Text", modifier = Modifier.align(Alignment.Center))
}
```

B.4 Modifiers in Jetpack Compose

Modifiers in Jetpack Compose are used to decorate or augment UI components. They allow you to set properties such as size, padding, alignment, background color, and more.
Common Modifiers:

- `Modifier.fillMaxSize()`: Makes the component fill the maximum size of its parent.
- `Modifier.padding(16.dp)`: Adds padding around the component.
- `Modifier.align(Alignment.Center)`: Aligns the component to the center of its parent.
- `Modifier.background(Color.Gray)`: Sets the background color of the component.

Modifiers are chained together to achieve the desired effect:

```
Text(
"Hello, World!",
modifier = Modifier
.fillMaxWidth()
.padding(16.dp)
.background(Color.LightGray)
.align(Alignment.CenterHorizontally)
)
```

Appendix C

Important function implemented in the App

C.1 Observation of LiveData in a Composable Function

It is often necessary to observe changes in LiveData and reflect those changes in the UI. The following Kotlin code demonstrates a composable function that observes LiveData and updates the UI accordingly.

```
        @Composable
        fun <T> observeLiveData(liveData: LiveData<T>): MutableState<T?> {
                val observedState = remember { mutableStateOf(liveData.value)
 ↪   }

                DisposableEffect(liveData) {
                        val observer = Observer<T> { value ->
                                observedState.value = value
                        }
                        liveData.observeForever(observer)
                        onDispose {
                                liveData.removeObserver(observer)
                        }
                }
                return observedState
        }
```

C.1.1 Function Description

The `observeLiveData` function is a generic composable function that takes a LiveData object of any type `T` and returns a `MutableState` containing the observed value. This function ensures that the UI is updated whenever the LiveData changes, by leveraging the Jetpack Compose framework.

Parameters

- `liveData: LiveData<T>`: The LiveData object to be observed.

Return Value

- `MutableState<T?>`: A mutable state containing the value of the observed LiveData.

C.1.2 Implementation Details

- `val observedState = remember { mutableStateOf(liveData.value) }`: Initializes a mutable state to hold the current value of the LiveData. The `remember` function ensures that this state is retained across recompositions.

- `DisposableEffect(liveData)`: Sets up a side effect to observe the LiveData. The `DisposableEffect` is used to manage the lifecycle of the observer, ensuring that it is properly disposed of when no longer needed.

- `val observer = Observer<T> { value -> observedState.value = value }`: Defines an observer that updates the mutable state whenever the LiveData value changes.

- `liveData.observeForever(observer)`: Starts observing the LiveData object.

- `onDispose { liveData.removeObserver(observer) }`: Ensures that the observer is removed when the composable leaves the composition, preventing memory leaks.

C.2 Conversion of Image to Bitmap

In Android development, converting an image from the YUV format to a Bitmap format is a common requirement for processing and displaying camera frames. The following Kotlin function, `imgToBM`, demonstrates how to perform this conversion efficiently.

```kotlin
fun imgToBM(image: Image): Bitmap {
        val p = image.planes
        val y = p[0].buffer
        val u = p[1].buffer
        val v = p[2].buffer
        val ySz = y.remaining()
        val uSz = u.remaining()
        val vSz = v.remaining()
        val jm8 = ByteArray(ySz + uSz + vSz)
        y.get(jm8, 0, ySz)
        v.get(jm8, ySz, vSz)
        u.get(jm8, ySz + vSz, uSz)
        val yuvImage = YuvImage(jm8, ImageFormat.NV21, image.width,
↪ image.height, null)
        val out = ByteArrayOutputStream()
        yuvImage.compressToJpeg(Rect(0, 0, yuvImage.width,
↪ yuvImage.height), 75, out)
        val imgBytes = out.toByteArray()
        return BitmapFactory.decodeByteArray(imgBytes, 0,
↪ imgBytes.size)
    }
```

C.2.1 Function Description

The `imgToBM` function converts an `Image` object in YUV format to a `Bitmap` object, which can be easily displayed in an Android application. This conversion is essential for handling raw camera frames and rendering them on the screen.

Parameters

- `image: Image`: The input image in YUV format, typically obtained from a camera preview.

Return Value

- `Bitmap`: The resulting bitmap that can be displayed in an ImageView or processed further.

C.2.2 Implementation Details

- `val p = image.planes`: Retrieves the three planes (Y, U, and V) from the input image.
- `val y = p[0].buffer`, `val u = p[1].buffer`, `val v = p[2].buffer`: Extracts the buffers for the Y, U, and V planes.
- `val ySz = y.remaining()`, `val uSz = u.remaining()`, `val vSz = v.remaining()`: Determines the size of each plane buffer.
- `val jm8 = ByteArray(ySz + uSz + vSz)`: Allocates a byte array to hold the combined YUV data.
- `y.get(jm8, 0, ySz)`, `v.get(jm8, ySz, vSz)`, `u.get(jm8, ySz + vSz, uSz)`: Copies the Y, U, and V data into the byte array.
- `val yuvImage = YuvImage(jm8, ImageFormat.NV21, image.width, image.height, null)`: Creates a `YuvImage` object from the YUV data, specifying the NV21 format and the image dimensions.
- `val out = ByteArrayOutputStream()`: Initializes a `ByteArrayOutputStream` to hold the JPEG-compressed image data.
- `yuvImage.compressToJpeg(Rect(0, 0, yuvImage.width, yuvImage.height), 75, out)`: Compresses the YUV image to JPEG format and writes it to the output stream.
- `val imgBytes = out.toByteArray()`: Converts the output stream to a byte array.
- `return BitmapFactory.decodeByteArray(imgBytes, 0, imgBytes.size)`: Decodes the byte array into a `Bitmap` object and returns it.

Bibliography

[1] ryzerobotics. Tello edu drone, 2024-April-08. [online] Available: `https://www.ryzerobotics.com/tello-edu`.

[2] wiedu. Detailed information on the tello edu app, 2024-Mai-05. [online] Available: `https://www.wiedu.com/telloedu/faq_en.html`.

[3] Android Doc. android studio, 2024-April-24. [online] Available: `https://developer.android.com/studio`.

[4] Android Doc. Android doc, 2024-April-20. [online] Available: `https://kotlinlang.org/docs/android-overview.html`.

[5] Android Doc. Jetpack compose documentation, 2024-April-20. [online] Available: `https://developer.android.com/develop/ui/compose/`.

[6] ryzerobotics. Tello sdk, 2024-April-08. [online] Available: `https://dl-cdn.ryzerobotics.com/downloads/Tello/Tello%20SDK%202.0%20User%20Guid`.

[7] github. H.264, 2024-April-24. [online] Available: `https://github.com/topics/h264`.

[8] eduporium. Tips and tricks, the dji tello edu drone, 2024-June-02. [online] Available: `https://developer.android.com/develop`.

[9] itu. H.264, 2024-April-21. [online] Available: `https://www.itu.int/rec/T-REC-H.264`.

Abstract

Conducting a comprehensive review of the use of the DJI Tello drone in educational projects. Exploring the various possibilities offered by this drone for the development of learning in computer programming and artificial intelligence.

Résumé

Faire un état de l'art complet sur l'utilisation du drone DJI Tello dans des projets éducatifs. Explorer les différentes possibilités offertes par ce drone pour le développement de l'apprentissage en programmation informatique et en Intelligence Artificielle.