Drone Performance Tracker Programmers Manual

Eckert, Hayley
hayleyeckert@tamu.edu

Westerfield, Jonathan
jgwesterfield@gmail.com

Elrod, Donald
donald.elrod.96@gmail.com

Rodriguez, Ismael
ir3987@tamu.edu

November 25, 2019

Abstract

Drones have become increasingly commonplace in society. From hobby drones and delivery drones to emergency response drones, drones are the way of the future. As they become more relevant in society, the need to properly train pilots how to fly responsibly is paramount. Even though the price of drones will decrease in the coming years, it is still expensive to replace a drone in the event of a crash. Furthermore, emergency response drone pilots need to be as accurate as possible when flying in crisis situations. This Drone Performance Tracker project enables the user to set up cameras around a bounded space in an open area, record drone flights, and analyze the flight paths of the pilot. This system thus provides the flight instructor with a more detailed look at the pilot's path and also will give the pilot more feedback as to how they are flying. This document contains the details as to how this project was implemented using Java and Python.

Contents

1 Architecture

The code base is split apart into 3 distinct sections. First there is the Android Camera application that records the flight. This portion is written in Java and its specific code document is generated with Doxygen. The second portion of the code base is the OpenCV tools that analyze the footage sent back from the phones. Using OpenCV, both video files are analyzed to extract the coordinates of the drone from the videos in 2 separate Python processes. Lastly, the coordinates are merged together to be displayed on the Flight App user interface. Both the OpenCV process and Flight App are documented using Sphinx.

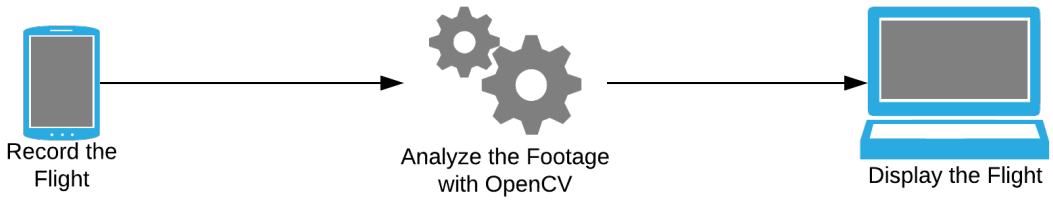

Figure 1: Overall Process Between the Three Architecture Sections.

2 Intercommunication

The Flight App is the component that handles all communication between each section of the architecture. It is the go between for getting footage from the phones and sending it to the OpenCV processes for analysis.

2.1 Flight App to Android Phones

The Flight app controls the phones by communicating to each phone over a TCP network connection. The Flight App will then send network signals to the phones for the phones to either: start recording, stop recording, or transfer their video files to the Flight App. Once the phone starts recording, the Flight App will wait for a signal from both phones that the transfer has completed.

2.2 Flight App To OpenCV

After the footage has been received from the phones, the Flight App will then open 2 OpenCV processes, one for each video, to analyze the drone footage. Once the footage has been analyzed, the Flight App will then take the OpenCV output and display the results in the UI for the user to view.

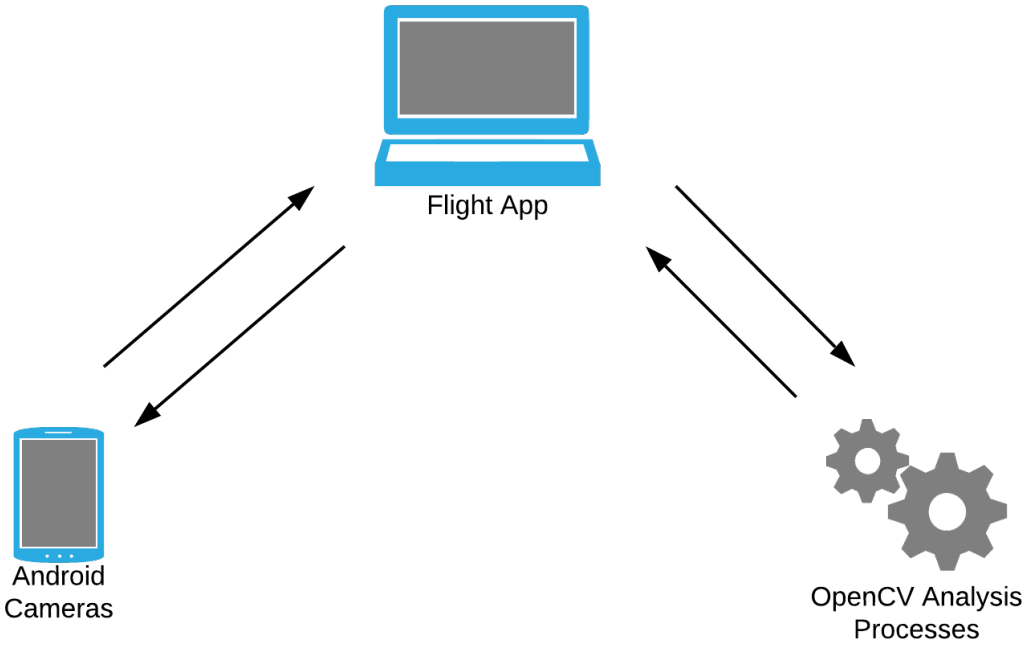

Figure 2: Overall Communication Between Each Section.

3 Android Camera

The phone application is written in Java using the Android SDK. The app was written using:

- Android Studio 3.5.1
- Gradle 4.10.3
- Targeted SDK Version: API 28 Android 9.0 (Pie)
- Minimum SDK Version: API 21 Android 5.0 (Lollipop)
- Java 1.8
- JUnit 4.12
- Jsch 0.1.52

3.1 How it Works

The Camera application is essentially a camera application with a network listener attached to it. When the application starts, the user must input the necessary information to create a TCP and SFTP connection with the laptop. Once the information has been entered, the TCP connection is established. The app then starts a separate thread to listen for signals asynchronously and the app then moves into the camera portion of the

application. The phone can then wait for signals to start recording, stop recording, and start the file transfer from the laptop. When the recording starts, the file name is determined from the phone ID passed in from the laptop and the current system time of the phone. When the recording is stopped, the video is saved to internal memory, the SFTP connection is established and the filename is passed into the connections to start the transfer. When the transfer is complete, the phone will then send a signal to the laptop signaling that the transfer is finished.

An important detail to note is that after testing, it was found that if there were other applications open on the phone (especially any application that relies on the phone's camera) the camera application may either hang or crash. It is important to close all other application on the phone before running the Camera application.

3.2 Reasoning For Using Android Phones

We used the following 2 Android phones:

- Google Pixel 4
- Samsung Galaxy S10e

We chose the use Android phones over a camera and Raspberry Pi combo for several reasons. First, having 2 phones reduces hardware complexity since the phones not only have cameras but also a computer and networking capabilities built in. In addition this saved costs since our team members already owned Android phones. Finally, our developers had the skill to develop in Android which allowed us to simplify hardware setup and user interactions. This means that the user doesn't need to go out and source their own expensive cameras, all they need to is download the app to setup the system.

3.3 Jsch

Jsch is a Java library used for creating Secure File Transfer Protocol (SFTP) connections programmatically. We were originally going to use FTP using the Apache Commons library, but had concerns with security and difficulty getting it to work. The only thing needed to open an SFTP connection to a target address is the username and password for a user account on the target server. In our case, the server was the laptop running the Flight App. After testing, we determined that it takes about 12 minutes to transfer 10 minutes of 4K footage (about 3 GB) from a phone to the laptop.

Unfortunately, we were unable to create a progress bar to indicate transfer progress. Every time we attempted to do so, it would cause the application to crash. However, after extensive research, this library was the only sensible solution to perform file transfers from the phones to the laptop without transferring application control to a third-party app.

3.4 Testing

All tests were run with JUnit 4.12. Most of the testing was focuses on the network connectivity of the application.

3.5 Network Signals

The network signals are represented as Enumerations in Java. The signals that can be sent and received are as follows:

- START
- STOP
- START_ACKNOWLEDGE
- STOP_ACKNOWLEDGE
- START_FTP
- START_FTP_ACKNOWLEDGE
- FTP_COMPLETED
- ILLEGAL
- NULL

The START, STOP, and START_FTP signals are sent from the laptop to signal the phone to start those actions. The START_ACKNOWLEDGE, STOP_ACKNOWLEDGE, START_FTP_ACKNOWLEDGE, FTP_COMPLETED signals are sent from the phone to the laptop in order to verify that the signals have been received. A message can be sent along with each signal provided it is delimited by a semicolon after the signal.

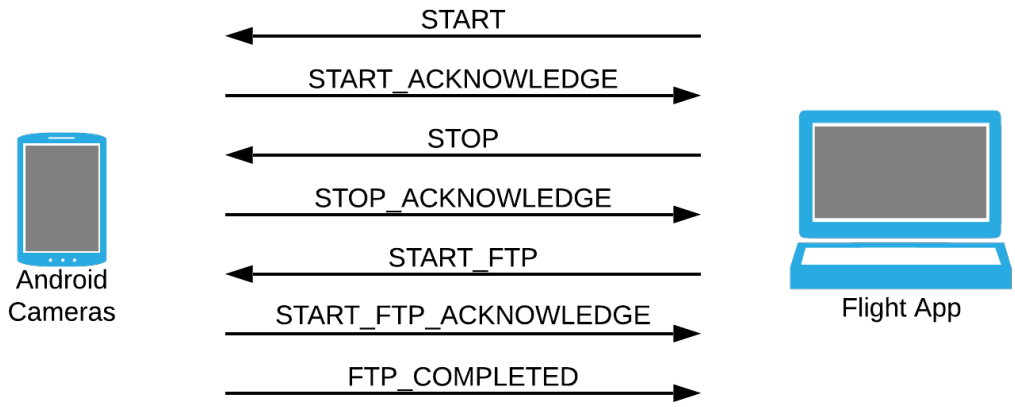

Figure 3: Signals sent between Flight app and phones.

3.6 Future Work

There are numerous things that could be done in order to make the phone application better given time and resources. These are shown below.

3.6.1 General Performance Enhancements

There are a couple issues with the phone application as is. The largest one is if the camera application is in the camera portion of the app and the phone changes orientation from landscape to portrait or vice-versa, the application will lose network connectivity. This is because of the application life-cycle and the fact that state is lost during orientation changes. There are difficulties with fixing this, the largest one being that all network connections (and everything related to them) are stored in a Singleton class instance. This includes the FTP connections and the thread listening for network signals. This must be serialized into bundle, saved, and reloaded after the orientation change. Info on how to do that can be found here: `https://stackoverflow.com/questions/32283853/android-save-state-on-orientation-change`.

The difficulty lies in setting the Singleton instance to a particular state. Even with the appropriate setter function in the Singleton instance there were still issues with keeping the connection to the laptop alive after they were reloaded.

The other issues dealt mostly with app performance. The application loses stability if there are multiple other apps running on the phone at the same time. This is mostly because there aren't adequate methods to handle life-cycle changes within our Camera app. These issues became apparent if another application that used the camera was running or if the phone received a notification during the run-time of the app.

3.6.2 Cross Platform Development

Because our team only had 2 Android phones among us, we were limited to a 2 camera setup due to budget constraints. The best fix would've been to use a cross platform development SDK in order for us to use the application on both Android and Apple devices. There are a couple popular options to choose from, the most prominent being Google's Flutter, and Microsoft's Xamarin. This would allow greater flexibility in using the system by being more accommodating for different phones (assuming the OS of the phone is within the minimum specifications).

3.6.3 Packaging The App

The very last enhancement, regardless of cross-platform availability, is having the app packaged and published on a site with easy access, such as the Google Play Store. This would allow users to easily download and install the app without having to enable developer options on their phone. While this would be more expensive for the Apple app store, this functionality would greatly increase ease of use for end users.

3.6.4 Increase the Recording Frame Rate

When recording basic drone flight paths, the current rate of 30 FPS works perfectly. However, if trying to test the performance of higher speed flights, a higher frame rate on the recording is needed. We would recommend using at least 60 FPS 4K footage, but 120 FPS would be best suited to high speed performance testing. This will definitely impact the performance of the processing of the video, however it will increase the reliability of the tracker and give better results. Using this would require a different method of transferring the footage, however, as the file size would be enormous.

4 OpenCV Analysis

The OpenCV Analysis process was written using the following:

- Visual Studio Code v1.40.1
- Python 3.7
- OpenCV 4.1.1

4.1 Important Files

The only important file in for the OpenCV analysis process is the OpenCVThreaded-Controller.py file in the `Flight-App/src/Controllers/` folder. This file contains the actual DroneTracker class and a main method to run the process. The only other important files are the lock files and the JSON output files that get deposited into the `drone-tracker/opencv-output` folder.

4.2 How it Works

Initially, the OpenCV DroneTracker class was supposed to be imported into the main program and run from there, where the DroneTracker would spawn multiple threads to process both videos at once. This ended up not being possible due to both PyQT5 and OpenCV needing to operate in the main thread. Our final solution to the problem was to structure the entire tracking process as its own program that could be run from the command line, and then spawn a new Python process from the main program to process each video.

When the videos are being processed, the each process creates a lock file at the beginning of the execution in order to signal that the analysis is currently running. The process then analyzes the footage and outputs the coordinates of the drone from the 2D footage. When finished, the extracted pixel coordinates are output into JSON files and the lock files are deleted. The lock files should always be deleted at the end of the program execution, even if an exception is thrown because the Flight app depends on these lock files. After the coordinates from both videos have been extracted, they need to be merged together in order to create 3D coordinates to plot the flight path. This is done is a specific merge function that is not part of the DroneTracker class.

Unfortunately, we were unable to create the correct mathematical method in order to accurately merge the coordinates such that they account for the perspective shift from the cameras. When a flight path is plotted, the path is greatly skewed because of how the drone appears to move from the perspective of both cameras.

4.3 Threading

Performing OpenCV analysis of 4K video footage is incredibly taxing for computers, as 4K videos are enormous in size and each frame has such a large amount of pixels. One of the biggest bottlenecks in analyzing the footage was the delay between reading individual frames from the video files, updating the tracker, and then waiting for the next video

frame to be read into memory. This process was even slower still when multiple videos were being processed at once, as the IO would lag behind the CPU on both threads and tracking would crawl to a stop. In order to fix this, the reading of the video was moved into its own thread, allowing a frame_queue to fill up with the video frames: this sped up the video analysis quite a bit, as the tracker was no longer bottle-necked by the disk read time. This meant the only bottleneck was how much RAM the user had on their computer and the capability of the CPU.

4.4 Tracker

The tracker used in our program is the KCF tracker, which learns fast and is able to recognize when it can no longer track an object (some trackers track random objects when their object goes out of frame or gets lost due to other objects, whereas the KCF tracker is much better about this). Each frame is fed to the tracker via the frame_queue. One thing to note is that, when using 4K footage, the frames need to be scaled down quite a bit in order to actually have it fit on the user's computer screen: sometimes by 50 - 66%. However, the analysis of the footage is done at 4K quality, and only the output shown to the user is scaled down.

Depending on the area being recorded and the size of the drone, we believe the footage could be processed in 1080P instead of 4K, which would also help drastically reduce processing time, however for the drones we are monitoring (the Parrot and Mavic), 4K is necessary. Since the drones are so small, it is imperative that as much data as possible is retrieved to ensure the OpenCV process has enough information to recognize the drone.

4.5 Coordinate System

While the tracker is tracking the drone through the air, every 15 frames, or 0.5 seconds, the x/y coordinates of the center of the bounding box around the drone, as well as the time, are added to a tuple which is in turn pushed to a list. This list of tuples is then written to the output JSON file mentioned above. Because of the orientation of the cameras, each camera is able to extract information about 2 separate 3D planes: one camera can see the x/z plane, while the other can see the y/z plane. Using these values, we are able to roughly, yet still somewhat accurately, reconstruct the flight path of the drone tracked by combining the values at each time value.

For example, you have from the first camera a tuple of the form (time, x_coordinate, z1_coordinate), and a tuple from the second camera (time, y_coordinate, z2_coordinate). The following math is done to convert these two tuples into one representing real world coordinates (keep in mind the coordinates are still in pixel format at this point, so they must be scaled to real world values):

```
for each tuple in list
    // we divide by the total number pixels to get a percent
    // that can be scaled to a real world distance
    x_real = (x_pixel / 3860) * 15
    y_real = (y_pixel / 3860) * 15
    z_real = 10 - ( ( (z1_pixel + z2_pixel) / 2 ) / 2160) * 10
```

This method gives reasonably accurate results when the drone is closest to the cameras, however due to some perspective warping it becomes less accurate the further away the drone is from the cameras.

4.6 Future Work

Since we were unable to find a method to account for the perspective shift of the cameras from 4.2, a new method of coordinate extraction should be used in future development. The merge_data_points function will need to be fixed. The coordinates extracted need to be transformed in such a way that the perspective of the cameras is accounted for. There are several methods we researched to fix this, but we did not have adequate time to implement any of them properly.

4.6.1 Linear Regression

The first, but most inefficient method is to use a calibration flight in which the user flies along the perimeter of the bounded space. From there, we know that the user is flying in straight lines and any issues with the 3D plotting is due to the cameras' perspective. We can then use linear regression on each side of the trapezoidal plot and find the best fit line for each side. From there, we can deduce that depending on where the drone is on one image, we can calculate where it would be based off the linear regression equation, where it *should* be, calculate the difference between these two coordinates and add that difference to the raw coordinate we get from the camera. We would be able to choose which regression formula to use based off of which side of the image the drone was on. This, in theory, would be able to correct for the perspective changes and plot the drone flight path correctly. Unfortunately, this would increase run-time because each point on the graph would have to be recalculated 3 times, one recalculation for each axis.

4.6.2 Proper Simple and Multi-View Stereo Transformations

In computer graphics, this type of problem is approached by using a stereo camera setup. There are two types of this stereo setup that would work well, one being simple stereo and the other being multi-view stereo. In simple stereo, two cameras are recording side by side, and by knowing the distance between the two camera lens' it is possible to calculate distance from an object. This would work alright in an application for this project, however a better way to approach it would be to use multi-view stereo. Multi-view stereo requires a camera setup like we currently have now, but we would also use an MVS pipeline to calculate the area between the two video streams that both cameras can see, and from there the object in the image of each video would be cross-referenced to the other video, and real world coordinates would be output. This approach seems the most promising, however with what we were able to find in the open-source field and with the time remaining, a huge rework would be needed to get this working. The project to check up on would be OpenMVS, readily found on GitHub under that title. The documentation is somewhat sparse, but it appears the implementation must be in C++, which clashed with our entire project structure of Python3.

4.6.3 Hardcore Trigonometry

Lastly, a trigonometric approach could be employed in order to improve the coordinates. The team began exploring this approach with the help of a TAMU mathematics TA named Andrew Winn, and his initial work is attached in the appendix of this report. However, we were not able to finalize the model to improve the accuracy given the time constraints but believe this approach may be possible to make viable moving forward with additional time. Essentially, trigonometry can be used to translate pixel coordinates from the two images into a real world 3D coordinate for the drone at each point in time. In this approach, the pixel coordinates from each of the two respective images are used along with the tangent of the angles at which the cameras are placed, the distance from which the cameras are from the field, and the dimensions of the tracking space, in order to calculate the real world x and y coordinates of the drone at that time moment. Furthermore, using these and additional angle calculation, it would be possible to estimate a real world z component for each of the two images then average these estimations together to produce a more accurate real world z coordinate. This analysis could be run on each set of images in order to more accurately compute the position of the drone in 3D space at each moment in time.

5 Flight App UI

The Flight App was developed using:

- Pycharm Community Edition 2019.1.2
- Python 3.7
- PyQt5
- MatPlotLib 3.1.1
- Numpy 1.17.2

5.1 How to Get It Running

Since Pycharm was the main IDE that was used to write the code, the following instructions are geared for PyCharm.

Steps to ensure that the proper networking functions work:

1. Make sure that port 8000 is unblocked in your networking settings.
2. Make sure that file transfer over SFTP is enabled (this means enabling file sharing and remote login in System Preferences for MacOS).

Make sure that the proper modules are installed from the list above:

1. Python 3.7 (Can use Pyenv or Venv)

2. PyCharm (with the correct version of Python selected for the project)
3. Pip install PyQt5, MatPlotLib, Numpy

Then follow the steps below to get the project running in Pycharm:

1. Mark the `src` and `Views` as sources root so Pycharm knows to look for files in those folders.
2. Create a configuration with `Flight-App/src/Controllers/Program_Controller.py` as the main file.
3. Make sure the configuration has specified Python 3.7 as the configuration interpreter.
4. Make sure all of the modules are installed for that interpreter (you can reference the requirements.txt file in the `Flight-App` folder.
5. That's it! You should be able to start the application from PyCharm by running the configuration.

5.2 Important Files

The main important files that are part of the flight application are the:

- `Program_Controller.py`
- `PhoneController.py`
- `Graph.py`

5.2.1 Program_Controller

The program controller contains the main method needed in order to start the application. Simply run `python3 Program_Controller.py` in order to start the application. This file dictates the program flow and ties together all of the views while also calling the underlying functions needed to communicate with the phones and open the OpenCV processes. This file also includes the calls to transfer the files from the phone to the laptop. All other phone functions are in the other view files.

5.2.2 PhoneController

The file contains the PhoneControl class. This class contains all of the methods needed for sending and receiving signals from the phones. All of the phone network calls are synchronous calls in order to force the application to wait until each network signal has been acknowledged by the phone. In addition, there was a need to have the network calls be blocking so we could freeze the program run-time while the video files transferred from the phone to the laptop.

Once the signal has been sent to the phone to start the file transfer, there is a `waitForFileTransfer` function that will block until it recieves the signal that the file transfer is complete. The network signals that get sent to the phone are the same as those in Section 3.5. Every function in the PhoneControl class sends it signal by opening a thread, one for each phone, sending the signal, and waiting back for the response. The signals are sent as 2 different threads in order to get the signals to send as closely together as possible. This is especially important if one of the phones responds back a little slower than the other one. To send the signal to each phone one at a time may lead to the phone video footage not being synced up closely enough.

5.2.3 Graph.py

The Graph class in Graph.py handles plotting the flight path in a 3D plot and calculating the smoothness metric for the flight. The points are taken from the output of the OpenCV processes after they have been merged together. The smoothness metric function for the flight path is also located in the Graph class.

5.3 The Smoothness Metric of the Flight

The smoothness metric is derived from the Log Dimensionless Jerk formula. This was used because this is largely regarded as the most reliable smoothness metric used in medical studies involving the smoothness of motion in patients with neurodegenerative diseases. The closer to 0 the output of this formula is, the smoother it is. The function takes in array of velocities (ours were derived from the velocities between different points of the flight) and computes the jerk from those velocities. Log Dimensionless Jerk (LDLJ) is based off of Dimensionless Jerk. The formulas for both are shown below.

$$DLJ = -\frac{(t_2 - t_1)}{v_{peak}^2} \int_{t_1}^{t_2} \left| \frac{d^2 v(t)}{dt^2} \right|^2 dt$$

$$LDLJ = -\ln |DLJ|$$

5.4 Connection to the Phones

The Flight App handles all of the connections to the phone. Refer to Figure 3 for the specific signals that pass between the phones and the Flight app. The Flight app controls the phones by starting a Python server and listening for a TCP connection from both phones. There can only be 2 open TCP connections at a time when the phones are synced. Once the connection is established, the `PhoneController` will then send signals to the phones. To send a signal, a thread for each TCP connection will be spawned so the network signals will be sent at the same time (or as close to each other as possible). Each signal sent to the phone must receive an acknowledgement signal sent back from the phone in order to know that the phone is functioning properly. The network socket is set to blocking so the application will not move forward and will block until the acknowledgement signal is received.

There is no timeout set on how long the network connection will block. If the phone connection is dropped and an acknowledgement is never received, the Flight app will hang. The only thing to do in this case is to restart the application.

5.5 Connection to the OpenCV Controller

After the video files have been downloaded into the `drone-tracker/FTP/` folder, the Flight app will then get the file names for the videos and spawn an OpenCV process for each one, passing in the file name as an argument. The Flight app will then look at the `drone-tracker/opencv-output/` folder to see if the `.lock` files for the OpenCV processes still exist. Once the processes finish and both the `.lock` files have been deleted, the Flight app will then take the JSON files containing the OpenCV output, merge the coordinates and display the path with a 3D MatPlotLib graph. Essentially the lock files output by the OpenCV processes function as a simple processes synchronization to make sure the Flight app waits for the full analysis to finish.

5.6 Skipping Wireless File Transfer

While it is not officially supported by the application, there is a way to skip the wireless file transfer of the video footage from the phones. If the wireless file transfer is skipped, the user can connect the phones to the laptop via USB, bypassing the long wait time for the the file transfer. The underpinning of how this is possible is the fact that videos are only deleted off of the phone once the file transfer has been completed. If the file transfer is never initiated, the video footage will never be deleted.

5.6.1 Windows File Transfer

To transfer to the videos from the phone to a Windows computer, simply plug the phone into the computer and go to the `Android/data/com.example.android.camera2video/files/` folder and move those files onto the laptop.

5.6.2 MacOS File Transfer

To transfer to the videos from the phone to a MacOS computer, a third-party application is required. Go to `https://www.android.com/filetransfer/` and download the application. Once downloaded, simply plug the phone into the computer and go to the `Android/data/com.example.android.camera2video/files/` folder through the Android File Transfer app and move those files onto the laptop.

5.6.3 Manipulating the Flight App

Once the videos have been downloaded, the Flight-App needs to be modified in the source code to bypass the file transfer functions. For bypassing the start and stop video functions, go into the `View_TrackingScreen.py` file and comment out the `self.phoneControl.startRecording()` function on line 103 and the `self.phoneControl.stopRecording()` function on line 128. This will stop the app from sending signals to the phone to start and stop recording.

Once that is taken care of, go into the `Program_Controller.py` file and comment out both the `phoneControl.startFileTransfer(self.pathToFTPDir)` and the `phoneControl.waitForFileTransfer()` functions on lines 370 and 371. This will disable sending the signals to the phone to start the file transfer of the video files.

Once these functions are removed (commented out), there is a specific process for doing the video analysis. Since the Flight-App will delete all files in the `drone-tracker/FTP/` and `drone-tracker/opencv-output/` folders on startup, the video files have to be put into the `drone-tracker/FTP/` at a specific time in the application run time.

To put the analyze the phone footage from the Flight-App without performing the wireless file transfer, follow the steps below:

1. Start the Flight-App.
2. Go to the Verify Setup Screen and click the 'Test Full Setup' button. Do NOT hit the 'Phone Sync' button.
3. Go back to the home screen.
4. Click the 'Start Tracking' button.
5. Input the Pilot, Instructor and Flight Instruction information and click 'Confirm' (For exporting the flight).
6. Click the 'Start Tracking' button and then the 'Stop Tracking' button. They won't do anything since we commented out that functionality.
7. MOST IMPORTANT PART. Take the video files from both phones and put them into the `drone-tracker/FTP/` folder. This is done outside of the Flight-App.
8. Go back to the Flight-App and click the Transfer Video Footage for Analysis' button on the Loading Screen.
9. The Flight-App will now take that footage and analyze it!

5.7 Future Work

In the future, there is opportunity to improve the Flight App user interface in several ways. First, the user experience could be improved by making the points on the draggable graph to be clickable, displaying the time value and coordinates when clicked. This would give the instructor even greater insight into the results. Second, the user interface could be improved through decreased processing time of the videos. Given the hardware available with the project budget, the transfer and processing time for videos is currently not insignificant as mentioned earlier, but with increased resources, this problem could be mitigated quickly. Lastly, the user interface could be improved through additional design work, making the interface even more engaging for users in the future.

Another feature to add would be to allow the user to skip the wireless file transfer of the video files in favor of connecting each phone to the computer over a USB cable. This would allow the videos to be transferred far more efficiently and greatly enhance the user

experience. This would be a simple modification to the user interface that would simply give the user the choice of transferring the file wirelessly or over the USB connection, then analyzing the footage once both files are uploaded. Instead of possibly waiting for 30 minutes for both files to transfer to the laptop, the user can cut the transfer time drastically and focus on the tracking portion of the application. This can currently be done but the process is not user friendly as described in section 5.6.

The final feature would be to add the time values to each coordinate as they are graphed onto the plot. This will allow the user to know where they were at what time in order to give them more info about the flight.

6 Project Wide Future Work

With all the details explained above, this product could be taken to the market. With the project, companies would be able to assess the skill of drone pilots. Companies would then be able to identify pilots of novice, intermediate, and expert skill level depending on the smoothness and trace of the individual's drone flight. This would then help to take out the uncertainties involved as the report will have the complete flight of the pilot. However, the project could also be used to further develop pilots. Pilots that use the program would be able to see their flight report from the program, allowing them to identify areas of their flight that could use improvement, which they could better later.

7 Appendix

7.1 Math Related to Section 4.6.3

The next four pages include the trigonometry from the team's work with student Andrew Winn. These formulas could be used as part of Future Work to make the coordinates more accurate.

Formulae for determining location of drone in three dimensions using images from adjacent sides

Andrew Winn—andrewwinn3@tamu.edu

VERSION 2 UPDATES: See new equations (1), (2), (5), and (6).

Assumptions

We assume the following (see Figure 1):

1. a square region of width w in which the drone can fly.
2. a fixed reference frame in which cameras A and B are located on the bottom and right sides of the region, respectively.
3. that cameras A and B are at (approximate) ground level, centered on the flight region, and offset by distance d.
4. that cameras A and B record in the same resolution.
5. an (x, y) coordinate system with x along the bottom axis and y along the vertical axis, with $(0, 0)$ at the bottom left and (w, w) at the top right. (This implies that x and y are measured in physical units relative to the flight area.)
6. consistency of physical units. (Where specified, some measurements will be done in pixels. All angle measurements will be made in radians.)
7. measurement of pixel values from the bottom left of the image.

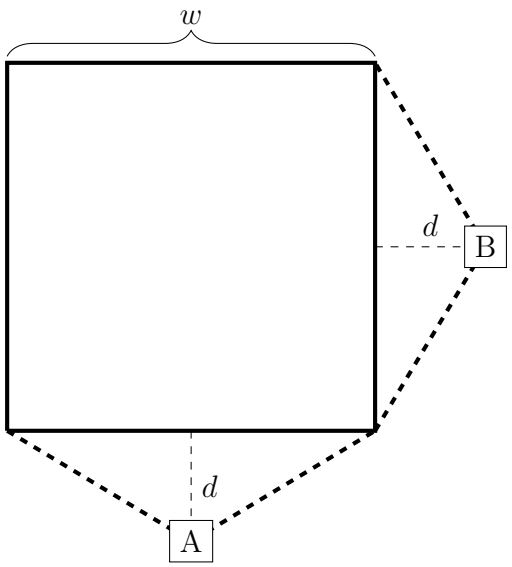

Figure 1: Flight area for drone, viewed from above

Fixed quantities

The following constants need only be determined once, assuming the camera setup does not change:

w, the width of the square flight area

d, the distance of the cameras from the edges of the flight area

θ_c, the angle of (the center of the field of view of) the camera relative to the horizontal (i.e, $\theta_c = 0$ implies a directly-forward view)

I_h, the height of the image (in px)

I_{AL}, the pixel value of the left side of the flight region as seen by camera A

I_{AR}, the pixel value of the right side of the flight region as seen by camera A

I_{AE}, the pixel value of the near edge of the flight region as seen by camera A

I_{BL}, the pixel value of the left side of the flight region as seen by camera B

I_{BR}, the pixel value of the right side of the flight region as seen by camera B

I_{BE}, the pixel value of the near edge of the flight region as seen by camera B

Variable quantities

The following measurements will be dependent on the position of the drone:

x_A, position of the drone measured from left of camera A's image (in px)

z_A, position of the drone measured from bottom of camera A's image (in px)

y_B, position of the drone measured from left of camera B's image (in px)

z_B, position of the drone measured from bottom of camera B's image (in px)

All other variables will be calculated from the values already introduced.

Determination of x and y values

We will find x and y, the lateral coordinates of the drone, in physical units. Define θ_x and θ_y as measured from the centerlines (See Figure 2. Note that both values may be negative.)

Since the fields of view of the cameras extend beyond the flight region, we use the following equations to obtain the values of θ_x and θ_y:

$$\theta_x = \left(\frac{2x_A}{I_A R - I_A L} - 1\right) \arctan\left(\frac{w}{2d}\right) \tag{1}$$

$$\theta_y = \left(\frac{2y_B}{I_B R - I_B L} - 1\right) \arctan\left(\frac{w}{2d}\right) \tag{2}$$

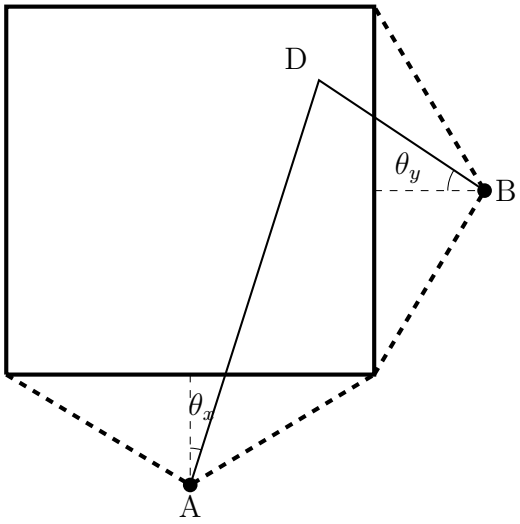

Figure 2: Drone at position D, viewed from above

Observing the intersection of the two segments at D produces the following equations:

$$\tan(\theta_x) = \frac{x - \frac{w}{2}}{d + y} \tag{3}$$

$$\tan(\theta_y) = \frac{y - \frac{w}{2}}{d + w - x}. \tag{4}$$

Combining these results gives us values for x and y in physical units, where $T_x = \tan(\theta_x)$ and $T_y = \tan(\theta_y)$, as obtained from equations (1) and (2), respectively.

$$x = \frac{2\,T_x(T_y + 1)d + (T_x(2\,T_y + 1) + 1)w}{2\,(T_x T_y + 1)}$$

$$y = -\frac{2\,(T_x T_y - T_y)d - (T_y + 1)w}{2\,(T_x T_y + 1)}$$

Determination of z

We now examine the scenario as viewed from the side. In theory, we need only to examine the view of one camera, but we will calculate both in order to mitigate measurement error. Since the camera is angled relative to the ground (at θ_c radians), we must account for that,

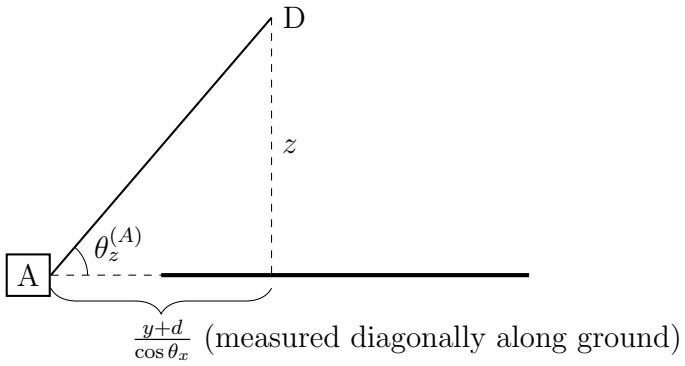

Figure 3: Drone at position D, from the side (heavy line is projection of flight area)

and thus we obtain that

$$\theta_z^{(A)} = \frac{2(z_A - I_{AE})\theta_c}{I_h - 2I_{AE}} \tag{5}$$

$$\theta_z^{(B)} = \frac{2(z_B - I_{BE})\theta_c}{I_h - 2I_{BE}} \tag{6}$$

where (5) is obtained from camera A, and (6) is obtained from camera B (see Figure 3).

Referring to Figure 2 again, we see that the distance AD, measured along the ground, is $\frac{y+d}{\cos(\theta_x)}$ (and likewise, distance BD is $\frac{w-x+d}{\cos(\theta_y)}$). (Note that while we only needed the tangents of θ_x and θ_y until now, we now require the actual angles. Since we already calculated x and y, it is likely easiest to use equations (3) and (4).) Thus, using results (5) and (6), we obtain two values for z, which we may average together for a more accurate estimate.

$$z^{(A)} = \frac{y+d}{\cos(\theta_x)} \tan(\theta_z^{(A)})$$

$$z^{(B)} = \frac{w-x+d}{\cos(\theta_y)} \tan(\theta_z^{(B)})$$

$$z \approx \frac{z^{(A)} + z^{(B)}}{2}$$

We now have values (in physical units) for x, y, and z as desired.

7.2 Flight App Documentation

This section contains all of the code documentation for the Python Flight-App. It was generated with Sphinx.

Drone Tracker

Release 1.0.4

Hayley Eckert, Jonathan Westerfield, Donald Elrod

Nov 25, 2019

CONTENTS:

CHAPTER

ONE

INTRO

Version 1.0.3

Written by: Hayley Eckert, Jonathan Westerfield, Donald Elrod, and Ismael Rodriguez

This application is the controller for the entire project. It interfaces with the cell phones and controls them in order to record footage of a drone flight. It will then open 2 OpenCV processes in order to analyze the footage from the cell phones.

This flight application relies on OpenCV and MatPlotLib.

The main sections of the application are as follows:

- **Program_Controller:** Starts up the entire Drone Tracker program the user interacts with
- **PhoneController:** Sends and receives network signals from the phone over TCP to control the phone.
- **OpenCV:** Takes the file of the video transferred from the phone and extracts the coordinates of the drone from the footage

CHAPTER

TWO

PROGRAM CONTROLLER

class Program_Controller.**Controller**(*phoneControl: PhoneController.PhoneControl*)

Controller class for the application. Changes between application views based on user input. Run this file in order to begin the application.

cleanup() → None

Goes through our file structure and deletes all files (except anything in the Flight folder). This is for when we are done with the program and want to delete the raw footage from the phone, the output points from the opencv processing and any other files that were used during the program execution. Should also be called before any time the user wants to fly.

Returns None

get_all_files(*folderPath: str*) → list

Will get all of the file names in a folder. This will be used after the video footage is transferred from the phone to the laptop. We need to grab the 2 videos filenames in the folder in order to send them to the opencv analysis to have the coordinates extracted. This will also be used when we need to splice together all of the coordinates output by the OpenCVController json files.

Parameters **folderPath** – The directory that we need to get all of the files from.

Returns A list of all of the file names in that directory

get_flight_info(*pilotName: str*, *instructorName: str*, *flightInstructions: str*) → None

Saves the pilot name, instructor name, and flight instructions once confirmed by the user.

Parameters

- **pilotName** – String containing the pilot name
- **instructorName** – String containing the instructor name
- **flightInstructions** – String containing the flight instructions

Returns None

get_in_order(*files: list*) → list

Will return the file names in order. This means that phone-1 will be first in the list and phone-2 will be second. Need this to ensure that a specific phone's coordinates are being used for the Z, X axis and the other is used for Z, Y

Parameters **files** – The list of files we get from the get_all_files() function

Returns The list of file names starting with phone-1 first and phone-2 second

import_flight(*flightPath: str*) → None

Reads in a .flight file and displays the report view for it.

Parameters **flightPath** – String with file path of chosen file to import.

Returns None

setupFileStructure() → None
Sets up the folders that need to exist before we can transfer footage and analyze. The file structure should be: drone-tracker > FTP, opencv-output, Flights. This makes it easier to keep track of the files that we are working with during the application lifecycle.

Returns None

show_home() → None
Loads the home startup screen for the user.

Returns None

show_loading_window() → None
Loads the loading screen for the user.

Returns None

show_report_window(*previousFlight: str*, *usingPreviousFlight: bool*, *flightData: dict*) → None
Loads the report screen for the user.

Parameters

- **previousFlight** – String containing path to flight data. Should be .flight file if usingPreviousFlight is true, or empty if usingPreviousFlight is false.
- **usingPreviousFlight** – Boolean representing if the report view is for an existing .flight file or a new analysis.
- **flightData** – Dictionary containing the flight data. Should be populated with only coordinates if usingPreviousFlight is false, and empty if usingPreviousFlight is true.

Returns None

show_tracking_window() → None
Loads the tracking screen for the user.

Returns None

show_verify_screen() → None
Loads the verify setup screen for the user.

Returns None

start_analysis() → None
Spawns the sub processes that will analyze the footage of the drone footage. We will need to have the files before hand so we can pass them into each OpenCVController process.

Returns None

transfer_complete(*flightData: dict*) → None
Calls the report view using the flight data dictionary.

Parameters **flightData** – Dictionary of flight data

Returns none.

transfer_footage(*phoneControl: PhoneController.PhoneControl*) → None
Transfers footage and calls DroneController to analyze the footage.

Parameters **phoneControl** – Phone Controller object for the active phone connection.

Returns none

updateFlightStatus() → None
Sets the status of the system verification test.

Returns none

wait_for_analysis() → None
Waits for the analysis of the footage to complete. Essentially is just a loop that checks to see if the file locks (*.lock) for our files are still in that directory. If they are, we wait, otherwise, we will exit the loop. From there, we need to get the output from those processes and splice the points together into a single 3D coordinate list.

Returns None

Program_Controller.**close_conn**(*phoneControl: PhoneController.PhoneControl*) → None
Closes the connection from the laptop to the phones.

Parameters **phoneControl** – PhoneControl object containing the active connection.

Returns None.

Program_Controller.**createPhoneConnection**(*portNo*) → PhoneController.PhoneControl
Creates a PhoneControl object used to commmunicate with the phones.

Parameters **portNo** – Port number we are going to be listening for signals from the phone over.

Returns PhoneControl object.

Program_Controller.**main**() → None
Begins the main application.

Returns None

CHAPTER

THREE

OPENCV

class OpenCVThreadedController.**DroneTracker**(*videoFile*)

This class is intended to track sUASs in video recorded from the smartphone app All containing code to track the drone and output the coordinates extracted from the recorded video is contained within this class, and this file is intended to be run as a separate process so that both of the recordings can be processed in parallel (don't try this on Windows though).

is_light_on(*frame*) → bool

Takes in a video frame and returns the frame at which the light turns on.

Parameters **frame** – A single video frame to see if the light is on.

Returns True if the light is on, false otherwise.

read_video() → None

This function is to be threaded, and its purpose is to read in the video file all at once to improve performance.

rescale_frame(*frame*, *percent=50*)

Resizes a frame as a percentage of the original frame size

Parameters

- **frame** – the frame to be resized
- **percent** – the percent value the frame needs to be rescaled to

resize_bbox(*bbox: tuple*, *factor=2*) → tuple

Resizes the bounding box for translating it to the full size video. In order to be able to see enough of the footage on screen to draw the box around the drone, the video frame must be resized, so the drawn bounding box must be translated back into the coordinate system the full size video uses. For example, if the 4k footage is shrunk by 50% (to 1080p), the scale factor here must be 2 so the coordinates chosen in the 1080p frame will match up with the actual drone coordinates in the 4k frame.

Parameters

- **bbox** – bounding box of selected drone, which is (x, y, box_width, box_height)
- **factor** – the factor by which to scale the bounding box

Returns tuple

trackDrone() → list

Function that contains all code to track the drone, and is to be run as a thread. Will run much slower if 2 processes running this method are started and run on different videos at the same time.

Returns List of tuples of the extracted coordinates of the footage, in the format [(time, x_coord, y_coord, z_coord)].

exception OpenCVThreadedController.**VideoCorruptedException**(*message: str*)
This error is raised if the video being read is corrupted, or if the frames cannot be successfully extracted from the video files

exception OpenCVThreadedController.**VideoNotPresentException**(*message: str*)
This error is raised when the video for processing is not there, or if an incorrect path is given

OpenCVThreadedController.**compute_coordinates**(*xA*, *yB*, *zA*, *zB*)
Compute the real world coordinates from pixel coordinates. :param xA: position of the drone measured from left of camera A's image (in px) :param yB: position of the drone measured from left of camera B's image (in px) :param zA: position of the drone measured from bottom of camera A's image (in px) :param zB: position of the drone measured from bottom of camera B's image (in px) :return: x_coord, y_coord, z_coord as the real world coordinates of the drone

OpenCVThreadedController.**get_phone_id**(*filename: str*) → str
Gets the phone Id from the end of the file name so we can keep track of the json and lock files.

Parameters **filename** – The file name of the video file. Should have "phone-#.mp4" file names.

Returns The ID of the phone from the file name

OpenCVThreadedController.**main**(*filename: str*) → None
Will take the filename passed in and analyze the footage. All coordinates of the drone in the footage will be output to a json file.

Returns None

OpenCVThreadedController.**merge_data_points**(*phone1Points: list*, *phone2Points: list*) → dict
Takes the points outputted by the opencv analysis and merges the points together to create the 3D coordinates needed to output the visual flight path.

Parameters

- **phone1Points** – The opencv datapoints created from the main method of this class for the first phone
- **phone2Points** – The opencv datapoints created from the main method of this class for the second phone

Returns List of tuples of coordinates and time values that represent the flight path of the drone in the format [(time, x_coord, y_coord, z_coord)]

CHAPTER

FOUR

EXCEPTIONS

exception Exceptions.**FailedDisconnectException**(*message: str*)
This error is for telling us that something went wrong when we tried to disconnect from the RPI.

exception Exceptions.**FailedRPIFlashException**(*message: str*)
This error is for letting us know that the RPI did not flash the light.

exception Exceptions.**PhonesNotSyncedException**(*message: str*)
This exception is for when the user tries to perform an operation with the phones without actually syncing the phones first.

exception Exceptions.**RPINotConnectedException**(*message: str*)
This class is for letting us know that we had an issue connecting to the raspberry pi.

exception Exceptions.**RecordingNotStartedException**(*message: str*)
This exception class is for alerting the user that the recording has not started and they are trying to access a function that requires the phone cameras to be rolling.

exception Exceptions.**TransferNotStartedException**(*message: str*)
This exception class is for alerting the user that the file transfer process has not been started and any actions that depend on it will fail.

exception Exceptions.**VideoCorruptedException**(*message: str*)
This error is raised if the video being read is corrupted, or if the frames cannot be successfully extracted from the video files

exception Exceptions.**VideoNotPresentException**(*message: str*)
This error is raised when the video for processing is not there, or if an incorrect path is given

CHAPTER

FIVE

PHONECONTROLLER

class PhoneController.**PhoneControl**(*portNum: int*)

This class is for communicating with and controlling the phones out in the field. It uses simple TCP connections with each phone in order to control them.

closeConn() → None

Closes all of the connections and the socket.

Returns None

isTransferring() → bool

A flag for us to access to see if the system is still waiting for the video to finish the file transfer of the videos it recorded.

Returns True if the video has finished transferring, false otherwise.

setupSocket() → None

Creates the socket that we will use to listen for incoming connections.

Returns None

startFileTransfer(*filepath: str*) → None

Will send a signal to the phone that tells it to transfer the video files it recorded over to the laptop by opening an FTP connection.

Parameters **filepath** – The file path on our laptop that the phones will need to send their videos to over FTP.

Returns None

startRecording() → None

Call this in order to send a signal to the phones that they need to start recording.

Returns None

stopRecording() → None

Call this in order to send a signal to the phones that they need to stop tracking. Sends both phones a stop signal and the name of the file path that they will need to send their videos to over FTP.

Returns None

sync() → None

This function will wait until both phones have been connected to this app.

Returns None

synced() → bool

The getter function for seeing if the phones synced or not.

Returns True if they have been synced, false otherwise

threadSendSignal(*conn: socket.socket*, *signal: str*, *sigMessage: str*, *sigAck: str*) → None
This function takes a signal and the expected output so that we don't have to rewrite the same code for every action we have with the phones.

Parameters

- **conn** – A socket connection to a phone that has already been opened.
- **signal** – The Signal we want to send to the phone. Valid options are: START, STOP, and START_FTP
- **sigMessage** – A message that we want to send alongside the signal for the phone to use.
- **sigAck** – The Signal we expect to get back from the phone in response to our signal. Valid options are: START_ACKNOWLEDGE, STOP_ACKNOWLEDGE, START_FTP_ACKNOWLEDGE.

Returns None

threadWaitForFileTransfer(*conn: socket.socket*) → None
This is a thread for waiting for the signal from the phone that the file transfer to the filepath specified in the startFileTransfer() function arguments.

Parameters **conn** – A socket connection to a phone that has already been opened.

Returns None

waitForFileTransfer() → None
Spawns threads that will wait for both phones to send a signal saying that the file transfer of the videos is complete.

Returns None

CHAPTER

SIX

EXPORT FLIGHT

ExportFile.**export_data**(*pilotName*, *instructorName*, *flightDate*, *flightLength*, *flightInstructions*, *xCoordinates*, *yCoordinates*, *zCoordinates*, *velocityValues*, *outPath*) → None

Exports the flight data to a JSON file stored with a ".flight" extension.

Parameters

- **pilotName** – String containing the pilot name
- **instructorName** – String containing the instructor name
- **flightDate** – String containing the flight date
- **flightLength** – String containing the flight length
- **flightInstructions** – String containing the flight instructions
- **xCoordinates** – Array of x coordinates
- **yCoordinates** – Array of y coordinates
- **zCoordinates** – Array of z coordinates
- **velocityValues** – Array of velocity values
- **outPath** – String containing the path to save the file. Should end in ".flight".

Returns none

CHAPTER

SEVEN

IMPORT FLIGHT

ImportFile.**importData**(*inPath*) → dict

Imports the flight data from a JSON file stored with a ".flight" extension.

Parameters **inPath** – String containing the pilot name.

Returns Flight dictionary

CHAPTER

EIGHT

LOADING SCREEN

class View_LoadingScreen.**LoadingWindow**

The view for the loading page that is shown when the user presses the "Stop Tracking" button on the tracking window page.

Variables **__btnHome** – The class property for the 'Return to Home' button.

property BtnHome

The home for the view. Is used to return to home screen.

Returns None

property BtnTestReport

The test report for the view. Is used to switch to the test report screen.

Returns The reference to the test report button.

property LblStatus

Getter property for the timer label. We need to attach a QTimer to it so it can count the time the application has been tracking the drone.

Returns The timer label

property del_BtnHome

The home for the view. Is used to return to home screen.

Returns None

property del_BtnTestReport

The test report for the view. Is used to switch to the test report screen.

Returns The reference to the test report button.

property del_LblStatus

Getter property for the timer label. We need to attach a QTimer to it so it can count the time the application has been tracking the drone.

Returns The timer label

initView() → None

Sets up the view and lays out all of the components.

Returns None

returnHome() → None

Sends a signal to the main controller that the Cancel and Return to Home button was pushed.

Returns none

setSubtitle() → PyQt5.QtWidgets.QLabel

Sets up the subtitle label.

Returns The subtitle label

setTitle() → PyQt5.QtWidgets.QLabel
Sets up the title with the application title on top and the name of the screen just below it.

Returns Layout with the application title and screen title labels

property set_BtnHome
The home for the view. Is used to return to home screen.

Returns None

property set_BtnTestReport
The test report for the view. Is used to switch to the test report screen.

Returns The reference to the test report button.

property set_LblStatus
Getter property for the timer label. We need to attach a QTimer to it so it can count the time the application has been tracking the drone.

Returns The timer label

setupLoadingIcon() → PyQt5.QtWidgets.QLabel
Used for configuring the loading icon on the loading screen. Loading icon is a gif, so QMovie is used to animate the icon.

Returns The icon containing the loading label.

signalTestReport() → None
Sends a signal to the main controller that the Test Report button was pushed. NOTE: ONLY USED FOR TESTING PURPOSES

Returns none

signalTransferFootage() → None
Sends a signal to the main controller that the button to transfer footage was pressed.

Returns none

CHAPTER

NINE

REPORT SCREEN

class View_ReportScreen.**ReportWindow**(*pilotName: str*, *instructorName: str*, *flightInstructions: str*, *previousFlight: str*, *usingPreviousFlight: bool*, *flightData: dict*)

The view for the report page that is shown when the user opens the application.

Variables

- **__btnExport** – The class property for the 'Export Results' button.
- **__btnFlyAgain** – The class property for the 'Fly Again' button.
- **__btnHome** – The class property for the 'Return to Home' button.
- **__btnViewGraphVelocity** – The class property for the 'View Flight Path' button.
- **__btnViewGraphNoVelocity** – The class property for the 'View Flight Path with Velocity Changes' button.
- **__btnViewInstructions** – The class property for the 'View Flight Instructions' button.

property BtnExport

The export button for the view.

Returns None

property BtnFlyAgain

The fly again button for the view.

Returns None

property BtnHome

The home for the view. Is used to return to home screen.

Returns None

property BtnViewGraphNoVelocity

The home for the view graph without velocity button.

Returns None

property BtnViewGraphVelocity

The home for the view graph with velocity button.

Returns None

property LblFlightDate

Getter for the flight date label.

Returns Reference to the flight date label.

property LblFlightInstructions
Getter for the flight instructions label.

Returns Reference to the flight length label.

property LblFlightLength
Getter for the flight length label.

Returns Reference to the flight length label.

property LblInstructor
Getter for the instructor label.

Returns Reference to the instructor label.

property LblPilot
Getter for the Pilot label so we can set who the pilot is for the flight in child class.

Returns Reference to the pilot label.

analyzeFlight (*flightDict: dict*) → dict
Analyzes the flight data to extract coordinates, velocity values, and statistics.

Parameters **flightDict** – Dictionary of flight data, with only coordinates populated.

Returns Updated dictionary, with legal points and flight statistics included.

createStatisticsTable () → PyQt5.QtWidgets.QTableWidget
Creates a table containing flight statistics.

Returns QTableWidget containing flight statistics.

property del_BtnExport
The export button for the view.

Returns None

property del_BtnFlyAgain
The fly again button for the view.

Returns None

property del_BtnHome
The home for the view. Is used to return to home screen.

Returns None

property del_BtnViewGraphNoVelocity
The home for the view graph without velocity button.

Returns None

property del_BtnViewGraphVelocity
The home for the view graph with velocity button.

Returns None

property del_LblFlightDate
Getter for the flight date label.

Returns Reference to the flight date label.

property del_LblFlightInstructions
Getter for the flight instructions label.

Returns Reference to the flight length label.

property del_LblFlightLength
Getter for the flight length label.

Returns Reference to the flight length label.

property del_LblInstructor
Getter for the instructor label.

Returns Reference to the instructor label.

property del_LblPilot
Getter for the Pilot label so we can set who the pilot is for the flight in child class.

Returns Reference to the pilot label.

handleEndSliderValueChange(*value*) → None
Listener that will updated the stop time label when the user moves the stop time slider.

Parameters **value** – The value that the stop slider has been moved to horizontally.

Returns None

handleStartSliderValueChange(*value*) → None
Listener that will updated the start time label when the user moves the start time slider.

Parameters **value** – The value that the start slider has been moved to horizontally.

Returns None

initView(*pilotName: str*, *instructorName: str*, *flightInstructions: str*, *previousFlight: str*, *usingPreviousFlight: bool*, *flightData: dict*) → None
Sets up the view and lays out all of the components.

Parameters

- **pilotName** – String containing pilot name
- **instructorName** – String containing instructor name
- **flightInstructions** – String containing flight instructions.
- **previousFlight** – String containing path to flight data. Should be .flight file if usingPreviousFlight is true, or blank if usingPreviousFlight is false.
- **usingPreviousFlight** – Boolean denoting if the report should be populated from the same file or a different one.
- **flightDict** – Dictionary containing flight data. Should be empty if usingPreviousFlight is true.

Returns None

setButtonLayout() → PyQt5.QtWidgets.QHBoxLayout
Lays out the 'Export Results', 'Fly Again' and 'Import Previous Flight' buttons into a horizontal layout to be put on screen.

Returns The horizontal layout containing the 3 buttons

setSubTitle(*text*) → PyQt5.QtWidgets.QLabel
Sets up a subtitle label for the window

Parameters **text** – String as name for label

Returns Subtitle label

property set_BtnExport
The export button for the view.

Returns None

property set_BtnFlyAgain
The fly again button for the view.

Returns None

property set_BtnHome
The home for the view. Is used to return to home screen.

Returns None

property set_BtnViewGraphNoVelocity
The home for the view graph without velocity button.

Returns None

property set_BtnViewGraphVelocity
The home for the view graph with velocity button.

Returns None

property set_LblFlightDate
Getter for the flight date label.

Returns Reference to the flight date label.

property set_LblFlightInstructions
Getter for the flight instructions label.

Returns Reference to the flight length label.

property set_LblFlightLength
Getter for the flight length label.

Returns Reference to the flight length label.

property set_LblInstructor
Getter for the instructor label.

Returns Reference to the instructor label.

property set_LblPilot
Getter for the Pilot label so we can set who the pilot is for the flight in child class.

Returns Reference to the pilot label.

setupFlightInfo() → PyQt5.QtWidgets.QGridLayout
Sets up the flight info (pilot, instructor, date, length, and smoothness score) in a grid.

Returns Grid layout of the flight information

setupGraph(*flightData: dict*, *displayVelocity: bool*) → None
Sets up the 3d plot for viewing upon click of button. displayVelocity is a boolean denoting if the graph should display colored segments for velocity.

Parameters

- **flightData** – Dictionary of flight data
- **displayVelocity** – Bool denoting if velocity should be plotted or not.

Returns None

setupSlider() → PyQt5.QtWidgets.QVBoxLayout
Setups the slider that will be used to adjust the times of the flight path that will be displayed on the graph. This lets us control the beginning and ending bounds of the time of the flight that we want to view. We

have to use 2 sliders. One for the start time and one for the end. We originally wanted to make this such that both sliders were on top of each other to make it more intuitive, but we ran into a bug that prevented us from doing that.

Returns A horizontal layout containing both sliders and the labels displaying the time that they represent.

setupTitle() → PyQt5.QtWidgets.QVBoxLayout
Sets up the title with the application title on top and the name of the screen just below it.

Returns Layout with the application title and screen title labels

showWindow() → None
Takes all of the elements from the view and displays the window.

Returns None

signalExportResults() → None
Sends a signal to the main controller that the Export Results button was pushed.

Returns none

signalReturnHome() → None
Sends a signal to the main controller that the Return Home button was pushed.

Returns none

signalStartTracking() → None
Sends a signal to the main controller that the Fly Again button was pushed.

Returns none

CHAPTER

TEN

STARTUP SCREEN

class View_StartupScreen.**StartupWindow**(*flightModeEnabled: bool*)

The view for the home Startup page that is shown when the user opens the application.

Variables

- **__btnVerifySetup** – The class property for the 'Verify Setup' button.
- **__btnStart** – The class property for the 'Start Tracking' button.
- **__btnImport** – The class property for the 'Import Previous Flight' button.

property BtnImport

Getter for the Import Previous Flight button. Is used to import past flight files. Use to attach functionality.

Returns None

property BtnStart

Getter for the startTracking button. Use to attach functionality.

Returns None

property BtnVerifySetup

Getter for the verifySetup button. Use to attach functionality.

Returns The reference to the verifySetup button

property del_BtnImport

Getter for the Import Previous Flight button. Is used to import past flight files. Use to attach functionality.

Returns None

property del_BtnStart

Getter for the startTracking button. Use to attach functionality.

Returns None

property del_BtnVerifySetup

Getter for the verifySetup button. Use to attach functionality.

Returns The reference to the verifySetup button

initView() → None

Sets up the view and lays out all of the components.

Returns None

openFileNameDialog() → None

Allows user to select a .flight file from a file dialog window.

Returns Path to selected file as a string.

setButtonLayout() → PyQt5.QtWidgets.QHBoxLayout
Lays out the 'Test Config', 'Start' and 'Import' buttons into a horizontal layout to be put on screen.

Returns The horizontal layout containing the 3 buttons

setTeamMembers() → PyQt5.QtWidgets.QVBoxLayout
Sets up the team members label for the window

Returns Team members label of the application

setTitle() → PyQt5.QtWidgets.QVBoxLayout
Sets up the title with the application title on top and the name of the screen just below it.

Returns Layout with the application title and screen title labels

property set_BtnImport
Getter for the Import Previous Flight button. Is used to import past flight files. Use to attach functionality.

Returns None

property set_BtnStart
Getter for the startTracking button. Use to attach functionality.

Returns None

property set_BtnVerifySetup
Getter for the verifySetup button. Use to attach functionality.

Returns The reference to the verifySetup button

setupAMLogo() → None
Used for configuring the display for the A&M logo on the startup screen.

Returns None

setupPicture() → None
Used for configuring the display for the logo on the startup screen.

Returns None

signalImportFlight() → None
Calls function to allow user to select a file for import. Sends a signal to the main controller that the Import Previous Flight button was pushed.

Returns None.

signalStartTracking() → None
Sends a signal to the main controller that the Start Tracking button was pushed.

Returns none

signalVerifySetup() → None
Sends a signal to the main controller that the Verify Setup button was pushed.

Returns none

CHAPTER

ELEVEN

TRACKING SCREEN

class View_TrackingScreen.**TrackingWindow**(*phoneControl: PhoneController.PhoneControl*)

The view for the tracking view page that is shown when the user presses the "Start Tracking" button on the home page. Allows the user to enter in flight information and begin tracking the drone.

Variables

- **__btnConfirm** – The class property for the 'Confirm' button.
- **__btnClear** – The class property for the 'Clear' button.
- **__btnStart** – The class property for the 'Start Tracking' button.
- **__btnStop** – The class property for the 'Stop Tracking' button.

property BtnClear

Getter for the Clear button so we can attach functionality to it later.

Returns Reference to the clear button

property BtnConfirm

Getter for the Confirm button so we can attach functionality to it later.

Returns Reference to the confirm button

property BtnStart

Getter for the Start button

Returns Reference to the start button

property BtnStop

Getter for the Stop button

Returns Reference to the stop button

property LblInstructor

Getter for the Instructor label so we can attach functionality to it later.

Returns The instructor label

property LblPilot

Getter for the Pilot label so we can attach functionality to it

Returns The pilot label

property LblTimer

Getter property for the timer label. We need to attach a QTimer to it so it can count the time the application has been tracking the drone.

Returns The timer label

property TBInstructor
Getter for the Instructor textbox so we can attach functionality to it later.

Returns The instructor textbox

property TBPilot
Getter for the Pilot Textbox so we can attach functionality to it

Returns The pilot textbox

property TEInstructions
Getter for the instructions text edit box

Returns Reference to the instructions text edit box

clearValues() → None
Clears the values in the text boxes.

Returns None

confirmValues() → None
Confirms the values in the textboxes by displaying a pop up message of the values.

Returns None

property del_BtnClear
Getter for the Clear button so we can attach functionality to it later.

Returns Reference to the clear button

property del_BtnConfirm
Getter for the Confirm button so we can attach functionality to it later.

Returns Reference to the confirm button

property del_BtnStart
Getter for the Start button

Returns Reference to the start button

property del_BtnStop
Getter for the Stop button

Returns Reference to the stop button

property del_LblInstructor
Getter for the Instructor label so we can attach functionality to it later.

Returns The instructor label

property del_LblPilot
Getter for the Pilot label so we can attach functionality to it

Returns The pilot label

property del_LblTimer
Getter property for the timer label. We need to attach a QTimer to it so it can count the time the application has been tracking the drone.

Returns The timer label

property del_TBInstructor
Getter for the Instructor textbox so we can attach functionality to it later.

Returns The instructor textbox

property del_TBPilot
Getter for the Pilot Textbox so we can attach functionality to it

Returns The pilot textbox

property del_TEInstructions
Getter for the instructions text edit box

Returns Reference to the instructions text edit box

initView() → None
Initializes and lays out all of the controls and elements on the view.

Returns None

returnHome() → None
Sends a signal to the main controller that the Return Home button was pushed.

Returns none

setClrConfirmBtns() → PyQt5.QtWidgets.QHBoxLayout
Sets the buttons for clearing and confirming the pilot, instructor, and flight instruction information.

Returns The confirmation button

setFlightInstructions() → PyQt5.QtWidgets.QVBoxLayout
Sets the textbox that will allow the instructor to type in the flight instructions for the pilot to try to match.

Returns A vertical layout with the Instructions label on top of the text box

setInstructor() → PyQt5.QtWidgets.QVBoxLayout
Sets up the instructor label and the textbox that will be used to set the instructor flying during this session.

Returns Returns a vertical layout with the instructor label over the instructor textbox

setPilot() → PyQt5.QtWidgets.QVBoxLayout
Sets up the Pilot label and the textbox that will be used to set the pilot flying during this session.

Returns Returns a vertical layout with the pilot label over the pilot textbox

setStartAndStopBtns() → PyQt5.QtWidgets.QHBoxLayout
Sets up the start and stop buttons for tracking the drones.

Returns

setStatusLabel(*text*) → PyQt5.QtWidgets.QLabel
Sets up a status label for the window

Returns Label of the application taken from the "text" parameter

setSubTitle(*text*) → PyQt5.QtWidgets.QLabel
Sets up a subtitle label for the window

Returns Subtitle of the application taken from the "text" parameter

setTimerLabel() → PyQt5.QtWidgets.QVBoxLayout
Sets the label that will continuously update and display the time that the application has been actively tracking. Need to attach a QTimer() to it.

Returns The label that will contain the timer.

setTitle() → PyQt5.QtWidgets.QVBoxLayout
Sets up the title with the application title on top and the name of the screen just below it.

Returns Layout with the application title and screen title labels

property set_BtnClear
Getter for the Clear button so we can attach functionality to it later.

Returns Reference to the clear button

property set_BtnConfirm
Getter for the Confirm button so we can attach functionality to it later.

Returns Reference to the confirm button

property set_BtnStart
Getter for the Start button

Returns Reference to the start button

property set_BtnStop
Getter for the Stop button

Returns Reference to the stop button

property set_LblInstructor
Getter for the Instructor label so we can attach functionality to it later.

Returns The instructor label

property set_LblPilot
Getter for the Pilot label so we can attach functionality to it

Returns The pilot label

property set_LblTimer
Getter property for the timer label. We need to attach a QTimer to it so it can count the time the application has been tracking the drone.

Returns The timer label

property set_TBInstructor
Getter for the Instructor textbox so we can attach functionality to it later.

Returns The instructor textbox

property set_TBPilot
Getter for the Pilot Textbox so we can attach functionality to it

Returns The pilot textbox

property set_TEInstructions
Getter for the instructions text edit box

Returns Reference to the instructions text edit box

startTracking() → None
Sends a signal to the main controller that the Start Tracking button was pushed.

Returns none

stopTracking() → None
Sends a signal to the main controller that the Stop Tracking button was pushed.

Returns none

CHAPTER

TWELVE

VERIFY SETUP SCREEN

class View_VerifySetupScreen.**VerifySetupWindow**(*phoneControl: PhoneController.PhoneControl*)

The view for the verify setup page that is shown when the user presses the "Verify Setup" button on the home page.

Variables

- **__btnPhoneSync** – The class property for the 'Phone Sync' button.
- **__btnTestLight** – The class property for the 'Test Light' button.
- **__btnTestFull** – The class property for the 'Test Full Setup' button.
- **__btnCheck** – The class property for the 'Check Status' button.
- **__btnHome** – The class property for the 'Return to Home' button.

property BtnCheck

The check status button for the view so we can attach functionality to it later on. Is used to check the status of the set up procedures.

Returns None

property BtnHome

The home for the view. Is used to return to home screen.

Returns None

property BtnPhoneSync

The phone sync button so we can attach functionality to it later on.

Returns The reference to the phoneSync button

property BtnTestFull

The test full setup button for the view so we can attach functionality to it later on. Is used to import past flight files.

Returns None

property BtnTestLight

The test light button so we can attach functionality to it later on.

Returns None

checkStatus() → None

Shows the status of the system setup.

Returns None

property del_BtnCheck
The check status button for the view so we can attach functionality to it later on. Is used to check the status of the set up procedures.

Returns None

property del_BtnHome
The home for the view. Is used to return to home screen.

Returns None

property del_BtnPhoneSync
The phone sync button so we can attach functionality to it later on.

Returns The reference to the phoneSync button

property del_BtnTestFull
The test full setup button for the view so we can attach functionality to it later on. Is used to import past flight files.

Returns None

property del_BtnTestLight
The test light button so we can attach functionality to it later on.

Returns None

initView() → None
Sets up the view and lays out all of the components.

Returns None

returnHome() → None
Sends a signal to the main controller that the Return Home button was pushed.

Returns none

setButtonLayout() → PyQt5.QtWidgets.QHBoxLayout
Lays out the 'Phone Sync', 'Test Light' and 'Test Full Setup' buttons into a horizontal layout to be put on screen.

Returns The horizontal layout containing the 3 buttons

setTitle() → PyQt5.QtWidgets.QVBoxLayout
Sets up the title with the application title on top and the name of the screen just below it.

Returns Layout with the application title and screen title labels

property set_BtnCheck
The check status button for the view so we can attach functionality to it later on. Is used to check the status of the set up procedures.

Returns None

property set_BtnHome
The home for the view. Is used to return to home screen.

Returns None

property set_BtnPhoneSync
The phone sync button so we can attach functionality to it later on.

Returns The reference to the phoneSync button

property set_BtnTestFull
The test full setup button for the view so we can attach functionality to it later on. Is used to import past flight files.

Returns None

property set_BtnTestLight
The test light button so we can attach functionality to it later on.

Returns None

setupPicture() → PyQt5.QtWidgets.QLabel
Used for configuring the display for the logo on the screen.

Returns Returns the label that contains our logo so it can be put on the main panel.

syncPhone() → None
Runs phone sync test.

Returns None

testFull() → None
Runs full system test.

Returns None

testLight() → None
Runs light test.

Returns None

CHAPTER

THIRTEEN

MATPLOT GRAPH

Graph.**checkCoordinates**(*flightDict: dict*)
Reads the input dictionary of flight data from a .flight file.

Parameters **flightDict** – Dictionary containing flight data.

Returns Dictionary of flight data.

Graph.**checkLegalInput**(*x*, *y*, *z*)
Checks if the inputted 3D coordinate is within legal bounds. Legal bounds are: x, y, z > 0 and x < 15 and y < 15 and z < 10.

Parameters

- **x** – x value to check.
- **y** – y value to check.
- **z** – z value to check.

Returns A boolean denoting if legal or not (true if legal, false if outside bounds).

Graph.**computeVelocity**(*x1*, *y1*, *z1*, *x2*, *y2*, *z2*, *t1*, *t2*)
Computes the velocity of the drone between two points (x1,y1,z1) and (x2,y2,z2) at respective times t1 and t2.

Returns A float value representing the velocity of the drone.

Graph.**computeVelocityStatistics**(*flightDict: dict*)
Computes statistics on the velocity points.

Parameters **flightDict** – Dictionary of flight data

Returns Updated dictionary.

Graph.**dimensionless_jerk**(*movement: list*, *fs: int*) → float
Calculates the dimensionless jerk for a 1 dimensional array of points.

Parameters

- **movement** – The numpy array of points to calculate the jerk for. The array containing the movement speed profile. Doesn't need to be numpy array but it MUST at least be a 1 dimensional list.
- **fs** – The sampling frequency of the data points.

Returns The dimensionless jerk estimate of the given movement's smoothness.

Graph.**generateGraph**(*flightData: dict*, *displayVelocity: bool*, *t1: float*, *t2: float*)
Driver function for generating the 3d graph of drone coordinates.

Parameters

- **flightData** – Dictionary containing flight data.
- **displayVelocity** – Boolean saying if velocity changes should be displayed on the graph.
- **t1** – Minimum time bound for plotting coordinates.
- **t2** – Maximum time bound for plotting coordinates.

Returns The figure to display as the 3d graph.

Graph.**log_dimensionless_jerk**(*movement: list*, *fs: int*) → float

Calculates the smoothness metric for the given speed profile using the log dimensionless jerk metric.

Parameters

- **movement** – The numpy array of points to calculate the jerk for. The array containing the movement speed profile. Doesn't need to be numpy array but it MUST at least be a 1 dimensional list.
- **fs** – The sampling frequency of the data points.

Returns The dimensionless jerk estimate of the given movement's smoothness.

Graph.**velocityColors**(*flightDict: dict*)

Determines the color of the line segment between two points based on velocity values. The line color is determined by the change in velocity of the drone between two points. The color of the line should be green if the velocity point is greater than the previous velocity point, yellow if within 0.5 m/s, and red if less.

Parameters **flightDict** – Dictionary of flight data.

Returns An array of color values to use when plotting line segments between points.

Graph.**velocityPoints**(*flightData: dict*)

Calculates the velocity of the drone between consecutive points for the entire flight.

Parameters **flightData** – Dictionary of flight Data.

Returns Modified flightData dictionary.

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

E

F

G

H

I

L

M

O

P

R

S

T

U

V

W

7.3 Android Camera Documentation

This next section contains all of the code documentation for the Android Camera app. This was generated using DOxygen.

Drone Tracker Camera2

Generated by Doxygen 1.8.16

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

Class Documentation

3.1 com.example.android.camera2video.AutoFitTextureView Class Reference

Inheritance diagram for com.example.android.camera2video.AutoFitTextureView:

3.2 com.example.android.camera2video.Camera2VideoFragment Class Reference

Inheritance diagram for com.example.android.camera2video.Camera2VideoFragment:

Collaboration diagram for com.example.android.camera2video.Camera2VideoFragment:

Classes

- class **CompareSizesByArea**
- class **ConfirmationDialog**
- class **ErrorDialog**

Public Member Functions

- View **onCreateView** (LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
- void **onViewCreated** (final View view, Bundle savedInstanceState)
- void **onResume** ()
- void **onPause** ()
- void **onClick** (View view)
- void **onRequestPermissionsResult** (int requestCode, @NonNull String[ ] permissions, @NonNull int[ ] grantResults)
- void **startRecordingVideo** ()
- void **stopRecordingVideo** ()

Static Public Member Functions

- static Camera2VideoFragment **newInstance** ()

3.2.1 Detailed Description

I pulled this straight from Google's examples and made modifications of my own in order to tailor it for our app. It receives controls from the CameraActivity so we know when to start and stop recording video. Also sets video footage to 4K in order to have as much data as we can to analyze the drone footage and recognize the drone.

Author

Jonathan Westerfield

Version

1.0.3

The documentation for this class was generated from the following file:

- main/java/com/example/android/camera2video/Camera2VideoFragment.java

3.3 com.example.android.camera2video.CameraActivity Class Reference

Inheritance diagram for com.example.android.camera2video.CameraActivity:

Collaboration diagram for com.example.android.camera2video.CameraActivity:

Public Member Functions

- void update (Signal signal, String message)

Protected Member Functions

- void **onCreate** (Bundle savedInstanceState)

3.3.1 Detailed Description

I pulled this code from the Google examples. This activity uses the Camera2VideoFragment in order to take video from the camera. This implements the observer pattern which listens to the signals from the Readloop class so we can control the camera and tell it to start and stop recording. This class also calls the controls needed to transfer the video file to the laptop once the recording has finished.

Author

Jonathan Westerfield

Version

1.0.3

3.3.2 Member Function Documentation

3.3.2.1 update()

```
void com.example.android.camera2video.CameraActivity.update (
            Signal signal,
            String message )
```

```
    This is for when the phone gets rotated. When the phone rotates or changes orientation,
    application state is lost. This means that the network connectivity in the NetConn Singleton
    also gets lost. This function should save that NetConn instance and hopefully keep the
    connection open. This gets called when the application state is deleted.
    @param savedInstanceState The bundle that contains the application state.
   &zwj;/
```

@Override protected void onSaveInstanceState(Bundle savedInstanceState) { super.onSaveInstanceState(saved↩ InstanceState); savedInstanceState.putSerializable("netconn", NetConn.getInstance()); }

/∗∗ This is for when the phone changes orientation. When the phone changes orientation, the application state is deleted. This leads to the NetConn class getting deleted. This function is called when the application is looking to reload the application state.

Parameters

savedInstanceState	The bundle that contains the application state. / @Override public void onRestoreInstanceState(Bundle savedInstanceState) { super.onRestoreInstanceState(savedInstanceState); // Restore UI state from the savedInstanceState. // This bundle has also been passed to onCreate. // TODO: figure out how to do a deep copy on this and set the Singleton to what was saved // NetConn.getInstance() = (NetConn) savedInstanceState.getSerializable("netconn"); }
	
	`/**`
	`The update function for this application. We need to listen for network signals so we can start/stop recording and file transferring.`
	`@param signal The signal type that was received from the laptop.`
	`@param message The optional message that accompanied the signal.`

Implements com.example.android.camera2video.Observer.

The documentation for this class was generated from the following file:

- main/java/com/example/android/camera2video/CameraActivity.java

3.4 com.example.android.camera2video.ConnectActivity Class Reference

Inheritance diagram for com.example.android.camera2video.ConnectActivity:

Collaboration diagram for com.example.android.camera2video.ConnectActivity:

Public Member Functions

- void init ()
- void btnConnClk (View view)
- void switchToCameraActivity ()
- void showFailedConnectionAlert (View view, String title, String message)

Protected Member Functions

- void **onCreate** (Bundle savedInstanceState)

3.4.1 Detailed Description

This class takes in the connection information such as the IP address, port number, username and password for the laptop so that it can create a connection to the laptop and transfer to the camera portion of this application. If the connection to the laptop fails, the exceptions that caused the failure are displayed on screen.

Author

Jonathan Westerfield

Version

1.0.3

3.4.2 Member Function Documentation

3.4.2.1 btnConnClk()

```
void com.example.android.camera2video.ConnectActivity.btnConnClk (
            View view )
```

Handles the click event for the connect button. Once clicked, the phone should take the IP address in the IP address textbox and store the connection info for the laptop. Then the phone should open a TCP connection to the laptop to receive control signals. Here is the call graph for this function:

3.4.2.2 init()

```
void com.example.android.camera2video.ConnectActivity.init ( )
```

Initializes the view and associates all of the view controls with class members in this class.

3.4.2.3 showFailedConnectionAlert()

```
void com.example.android.camera2video.ConnectActivity.showFailedConnectionAlert (
            View view,
            String title,
            String message )
```

An alert view that displays any exceptions that may have been thrown when trying to connect to the laptop.

Parameters

view	The context that this alert will show on.
title	The title of the alert.
message	The message of for the cause of the alert.

Here is the caller graph for this function:

3.4.2.4 switchToCameraActivity()

```
void com.example.android.camera2video.ConnectActivity.switchToCameraActivity ( )
```

Switches to the camera activity. Here is the caller graph for this function:

The documentation for this class was generated from the following file:

- main/java/com/example/android/camera2video/ConnectActivity.java

3.5 com.example.android.camera2video.FTPConn Class Reference

Inheritance diagram for com.example.android.camera2video.FTPConn:

Collaboration diagram for com.example.android.camera2video.FTPConn:

Public Member Functions

- FTPConn ()
- FTPConn (String host)
- FTPConn (String host, String username, String password, String destFilePath)
- FTPConn setHost (String host)
- FTPConn setUsername (String username)
- FTPConn setPassword (String password)
- FTPConn setSrcFilePath (String filepath)
- FTPConn setDestFilePath (String filePath)
- String getHost ()
- String getUsername ()
- String getPassword ()
- String getDestFilePath ()
- boolean isLive ()
- void setupConn () throws JSchException
- void closeConn ()
- boolean upload ()
- void update (Signal signal, String message)

3.5.1 Detailed Description

This class provides all of the functions necessary for the application to transfer the recorded video file back to the laptop over an SFTP connection. All that is needed for the connection to succeed is for the user's laptop to have openSSH installed and the proper permissions to open an SFTP connection. While this class implements the Observer pattern, the Observer pattern functions were cast aside and no longer work.

Author

Jonathan Westerfield

Version

1.0.3

Since

1.0.0

See also

NetConn

FTPUpload

3.5.2 Constructor & Destructor Documentation

3.5.2.1 FTPConn() [1/3]

```
com.example.android.camera2video.FTPConn.FTPConn ( )
```

Empty Constructor

3.5.2.2 FTPConn() [2/3]

```
com.example.android.camera2video.FTPConn.FTPConn (
            String host )
```

Class constructor.

Parameters

host	The ip address of the laptop we want to connect to.

3.5.2.3 FTPConn() [3/3]

```
com.example.android.camera2video.FTPConn.FTPConn (
            String host,
            String username,
            String password,
            String destFilePath )
```

Class constructor

Parameters

host	The ip address of the laptop we want to connect to.
username	The username for the laptop.
password	The password for the laptop.

3.5.3 Member Function Documentation

3.5.3.1 closeConn()

```
void com.example.android.camera2video.FTPConn.closeConn ( )
```

Allows us to close the sftp connection cleanly Here is the caller graph for this function:

3.5.3.2 getDestFilePath()

```
String com.example.android.camera2video.FTPConn.getDestFilePath ( )
```

Getter for the destination file path we want to send the file to.

Returns

The currently set destination file path.

3.5.3.3 getHost()

```
String com.example.android.camera2video.FTPConn.getHost ( )
```

Getter for the host.

Returns

The currently set host.

3.5.3.4 getPassword()

```
String com.example.android.camera2video.FTPConn.getPassword ( )
```

Getter for the password.

Returns

The currently set password.

3.5.3.5 getUsername()

```
String com.example.android.camera2video.FTPConn.getUsername ( )
```

Getter for the username.

Returns

The currently set username.

3.5.3.6 isLive()

```
boolean com.example.android.camera2video.FTPConn.isLive ( )
```

A way to check if our connection to the host machine is alive.

Returns

True if it is alive, false otherwise.

3.5.3.7 setDestFilePath()

```
FTPConn com.example.android.camera2video.FTPConn.setDestFilePath (
            String filePath )
```

Setter for the destination file path we need to send the video file to.

Parameters

filePath	The destination filepath

Returns

The current instance of this class according to the builder design pattern

3.5.3.8 setHost()

```
FTPConn com.example.android.camera2video.FTPConn.setHost (
            String host )
```

Setter for the host we need to connect to.

Parameters

host	The ip address of the laptop we want to connect to.

Returns

The current instance of this class according to the builder design pattern

3.5.3.9 setPassword()

```
FTPConn com.example.android.camera2video.FTPConn.setPassword (
            String password )
```

Setter for the password so we can login to the laptop.

Parameters

password	The password of the user needed to login to the laptop.

Returns

The current instance of this class according to the builder design pattern

3.5.3.10 setSrcFilePath()

```
FTPConn com.example.android.camera2video.FTPConn.setSrcFilePath (
            String filepath )
```

Setter for the destination file path we need to get the video file from.

Parameters

filepath	The source file path

Returns

The current instance of this class according to the builder design pattern

3.5.3.11 setupConn()

```
void com.example.android.camera2video.FTPConn.setupConn ( ) throws JSchException
```

Creates the sftp connection to the host (the laptop).

Exceptions

JSchException	In case some connection issue happens or ssh isn't setup on the host.

3.5.3.12 setUsername()

```
FTPConn com.example.android.camera2video.FTPConn.setUsername (
            String username )
```

Setter for the username so we can login to the laptop.

Parameters

username	The username of the user for the laptop.

Returns

The current instance of this class according to the builder design pattern

3.5.3.13 update()

```
void com.example.android.camera2video.FTPConn.update (
            Signal signal,
            String message )
```

Gets called when the file transfer is complete so we can send a signal to the laptop to let it know that the file transfer is done and it can start analyzing the footage.

Parameters

signal	The signal type that was received from the laptop.
message	The optional message that accompanied the signal.

Implements com.example.android.camera2video.Observer.

3.5.3.14 upload()

```
boolean com.example.android.camera2video.FTPConn.upload ( )
```

Sends the file to the destination folder on the host (the laptop)

Returns

True if it succeeded, false otherwise.

Exceptions

SftpException	

Here is the call graph for this function:

The documentation for this class was generated from the following file:

- main/java/com/example/android/camera2video/FTPConn.java

3.6 com.example.android.camera2video.FTPUpload Class Reference

Inheritance diagram for com.example.android.camera2video.FTPUpload:

Collaboration diagram for com.example.android.camera2video.FTPUpload:

Public Member Functions

- FTPUpload (String srcFilePath, String destFilePath, ChannelSftp sftp, Session session)
- void run ()
- void deleteVideo () throws Exception

3.6.1 Detailed Description

This class is a class to house the function for transferring the video file to the laptop. This function was originally carried out on the main thread. However, Donald's phone (the Google Pixel 4) is a lot more finicky than Ismael's phone (the Samsung S10e). The Pixel will throw an error message saying that frames were skipped on the main UI and the work we are trying to do is too taxing for the main thread. To get around this, we are putting the file transfer function into a separate thread so the main thread can still run while the file is being uploaded. This thread will be called from the FTPConn class.

Author

Jonathan Westerfield

Version

1.0.0

See also

FTPConn

3.6.2 Constructor & Destructor Documentation

3.6.2.1 FTPUpload()

```
com.example.android.camera2video.FTPUpload.FTPUpload (
            String srcFilePath,
            String destFilePath,
            ChannelSftp sftp,
            Session session )
```

Class constructor. Takes in the necessary file information to transfer the file as well as an open SFTP connection to the laptop in order to actually transfer the file. Must call the `run()` function to actually upload the file.

Parameters

srcFilePath	The file we need to transfer from the phone to the laptop.
destFilePath	The folder on the target laptop where the file will be transferred.
sftp	An open sftp connection to the laptop.
session	The sessions representing the connection to the laptop. (I don't know if this is needed tbh).

3.6.3 Member Function Documentation

3.6.3.1 deleteVideo()

```
void com.example.android.camera2video.FTPUpload.deleteVideo ( ) throws Exception
```

Deletes the file after it has been transferred to the phone so we can still have storage on the phone after multiple recordings. Here is the caller graph for this function:

3.6.3.2 run()

```
void com.example.android.camera2video.FTPUpload.run ( )
```

This function implements the run function for the Runnable interface. This function will take the file stored at the specified file location, transfer it over SFTP to the laptop and then delete the file once the transfer has finished. Here is the call graph for this function:

The documentation for this class was generated from the following file:

- main/java/com/example/android/camera2video/FTPUpload.java

3.7 com.example.android.camera2video.MyProgressMonitor Class Reference

Inheritance diagram for com.example.android.camera2video.MyProgressMonitor:

Collaboration diagram for com.example.android.camera2video.MyProgressMonitor:

Public Member Functions

- void addObserver (Observer observer)
- void removeObserver (Observer observer)
- void notifyObservers (Signal signal, String message)
- void init (int op, String src, String dest, long max)
- void end ()
- boolean **count** (long i)

3.7.1 Detailed Description

This class was created so that we could easily monitor when the SFTP file transfer started and finished. This also implements the Observer in the Observer Pattern to let any observer know when the file transfer was completed.

This has been deprecated as its functions are no longer needed.

Author

Jonathan Westerfield

Version

1.0.3

3.7.2 Member Function Documentation

3.7.2.1 addObserver()

```
void com.example.android.camera2video.MyProgressMonitor.addObserver (
            Observer observer )
```

Add an observer to tell them that the file transfer has finished.

Parameters

observer	The concrete observer to add.

Implements com.example.android.camera2video.Observable.

3.7.2.2 end()

```
void com.example.android.camera2video.MyProgressMonitor.end ( )
```

Called at the end of the file transfer Here is the call graph for this function:

3.7.2.3 init()

```
void com.example.android.camera2video.MyProgressMonitor.init (
            int op,
            String src,
            String dest,
            long max )
```

Called at the beginning.

Parameters

op	No Idea
src	Source file path.
dest	Destination file path on the server.
max	No Idea

3.7.2.4 notifyObservers()

```
void com.example.android.camera2video.MyProgressMonitor.notifyObservers (
            Signal signal,
            String message )
```

Notify the observers that the file transfer has completed.

Parameters

signal	The signal received from the server.
message	The optional message that accompanied the signal.

Implements com.example.android.camera2video.Observable.

Here is the caller graph for this function:

3.7.2.5 removeObserver()

```
void com.example.android.camera2video.MyProgressMonitor.removeObserver (
            Observer observer )
```

Remove observers to satisfy the interface.

Parameters

observer	The concrete observer we want to remove.

Implements com.example.android.camera2video.Observable.

The documentation for this class was generated from the following file:

- main/java/com/example/android/camera2video/MyProgressMonitor.java

3.8 com.example.android.camera2video.Observable Interface Reference

Inheritance diagram for com.example.android.camera2video.Observable:

Public Member Functions

- void addObserver (Observer observer)
- void removeObserver (Observer observer)
- void notifyObservers (Signal signal, String message)

3.8.1 Detailed Description

The interface that is required for any class that want to use this to send network signals to the classes that need it. Uses the Observer design pattern.

Author

Jonathan Westerfield

Version

1.0.0

3.8.2 Member Function Documentation

3.8.2.1 addObserver()

```
void com.example.android.camera2video.Observable.addObserver (
            Observer observer )
```

Adds a new observer to notify when there is a network signal received.

Parameters

observer	The concrete observer to add.

Implemented in com.example.android.camera2video.ReadLoop, and com.example.android.camera2video.MyProgressMonitor.

3.8.2.2 notifyObservers()

```
void com.example.android.camera2video.Observable.notifyObservers (
            Signal signal,
            String message )
```

The function that will call the observers update() method and send the signal and message for that observer to work with it.

Parameters

signal	The signal received from the server.
message	The optional message that accompanied the signal.

Implemented in com.example.android.camera2video.ReadLoop, and com.example.android.camera2video.MyProgressMonitor.

3.8.2.3 removeObserver()

```
void com.example.android.camera2video.Observable.removeObserver (
            Observer observer )
```

Removes the specified observer from the subscriber list.

Parameters

observer	The concrete observer we want to remove.

Implemented in com.example.android.camera2video.ReadLoop, and com.example.android.camera2video.MyProgressMonitor.

The documentation for this interface was generated from the following file:

- main/java/com/example/android/camera2video/Observable.java

3.9 com.example.android.camera2video.Observer Interface Reference

Inheritance diagram for com.example.android.camera2video.Observer:

Public Member Functions

- void update (Signal signal, String message)

3.9.1 Detailed Description

Is paired with the Observable class. Follows the Observer design pattern. Any class that wants to subscribe to the received network signals so it can react to them accordingly.

Author

Jonathan Westerfield

Version

1.0.0

3.9.2 Member Function Documentation

3.9.2.1 update()

```
void com.example.android.camera2video.Observer.update (
            Signal signal,
            String message )
```

The update function in the observer pattern.

Parameters

signal	The signal type that was received from the laptop.
message	The optional message that accompanied the signal.

Implemented in com.example.android.camera2video.FTPConn, and com.example.android.camera2video.CameraActivity.

The documentation for this interface was generated from the following file:

- main/java/com/example/android/camera2video/Observer.java

3.10 com.example.android.camera2video.ReadLoop Class Reference

Inheritance diagram for com.example.android.camera2video.ReadLoop:

Collaboration diagram for com.example.android.camera2video.ReadLoop:

Public Member Functions

- void addObserver (Observer sub)
- void removeObserver (Observer sub)
- void removeAllObservers ()
- void notifyObservers (Signal signal, String message)
- ReadLoop (BufferedReader reader)
- int getNumObservers ()
- Signal parseSignal (String message)
- String parseMessage (String message)
- void run ()

3.10.1 Detailed Description

This class is a listener loop that is spawned as a separate thread from the NetConn class. This allows the camera application to continuously listen for network signals and controls from the laptop even while other camera functions are occurring.

Author

Jonathan Westerfield

Version

1.0.3

3.10.2 Constructor & Destructor Documentation

3.10.2.1 ReadLoop()

```
com.example.android.camera2video.ReadLoop.ReadLoop (
            BufferedReader reader )
```

Class constructor. Takes in the BufferedReader instance from the parent NetConn class in order to fully setup and start listening to the laptop.

Parameters

reader	The BufferedReader instance with an open connection to the laptop.

3.10.3 Member Function Documentation

3.10.3.1 addObserver()

```
void com.example.android.camera2video.ReadLoop.addObserver (
            Observer sub )
```

Implements the addObserver function in the Observable interface. Adds objects that need to be updated of the status from this loop.

Parameters

sub	The object that needs to subscribe to the messages on this loop.

Implements com.example.android.camera2video.Observable.

3.10.3.2 getNumObservers()

```
int com.example.android.camera2video.ReadLoop.getNumObservers ( )
```

Allows us to get the number of subscribers current subscribed to this observable instance.

Returns

3.10.3.3 notifyObservers()

```
void com.example.android.camera2video.ReadLoop.notifyObservers (
            Signal signal,
            String message )
```

Goes through all observers in our subscriber list and calls their update functions to notify them of a network signal that came through. It is up to the observer to do with that information what they will.

Parameters

signal	The signal received from the server.
message	The optional message that accompanied the signal.

Implements com.example.android.camera2video.Observable.

Here is the caller graph for this function:

3.10.3.4 parseMessage()

```
String com.example.android.camera2video.ReadLoop.parseMessage (
            String message )
```

Parses the message recieved by the server and pulls out the message sent after the signal

Parameters

message	The raw message recieved from the server.

Returns

The message attached to the server signal.

Here is the caller graph for this function:

3.10.3.5 parseSignal()

```
Signal com.example.android.camera2video.ReadLoop.parseSignal (
            String message )
```

```
     Gets this instances thread object so that it can interrupt it
     @return The thread of this class
    &zwj;/
```

public Thread getThread() { return this.t; }

```
/**
   Parses the message received by the server and pulls out the Signal type
   that the server sent.
   @param message The full message that we need to parse.
   @return The Signal type to send to the rest of the app.
```

Here is the caller graph for this function:

3.10.3.6 removeAllObservers()

```
void com.example.android.camera2video.ReadLoop.removeAllObservers ( )
```

Is used to remove all of the observers when this thread needs to be closed. Here is the caller graph for this function:

3.10.3.7 removeObserver()

```
void com.example.android.camera2video.ReadLoop.removeObserver (
            Observer sub )
```

Allows for observers to remove themselves from the event feed. This isn't really needed for this application but I'm going to stick to the Observer Design Pattern dammit.

Parameters

sub	The observer (subscriber) that wants to be removed from updates.

Implements com.example.android.camera2video.Observable.

3.10.3.8 run()

```
void com.example.android.camera2video.ReadLoop.run ( )
```

Starts the thread Here is the call graph for this function:

The documentation for this class was generated from the following file:

- main/java/com/example/android/camera2video/ReadLoop.java

3.11 com.example.android.camera2video.Signal Enum Reference

Public Attributes

- START
- STOP
- START_ACKNOWLEDGE
- STOP_ACKNOWLEDGE
- START_FTP
- START_FTP_ACKNOWLEDGE
- FTP_COMPLETED
- ILLEGAL
- NULL

3.11.1 Detailed Description

Enumeration for different signals that can be handled by this app.

Author

Jonathan Westerfield

Version

1.0.0

3.11.2 Member Data Documentation

3.11.2.1 FTP_COMPLETED

`com.example.android.camera2video.Signal.FTP_COMPLETED`

Signal to tell the laptop that the file transfer of moving the video file to the laptop finished and was successful.

3.11.2.2 ILLEGAL

`com.example.android.camera2video.Signal.ILLEGAL`

A signal for this application to recognize that a signal that came from the laptop was not a recognized signal and is therefore illegal.

3.11.2.3 NULL

`com.example.android.camera2video.Signal.NULL`

This is a signal in case there was no signal sent from the laptop. SHOULD NEVER BE USED.

3.11.2.4 START

`com.example.android.camera2video.Signal.START`

Signal to tell the camera to start recording video.

3.11.2.5 START_ACKNOWLEDGE

`com.example.android.camera2video.Signal.START_ACKNOWLEDGE`

Signal to tell the laptop that this phone acknowledges the signal to start recording video.

3.11.2.6 START_FTP

`com.example.android.camera2video.Signal.START_FTP`

Signal to tell the phone to start transferring the recorded video file to the laptop for analysis.

3.11.2.7 START_FTP_ACKNOWLEDGE

`com.example.android.camera2video.Signal.START_FTP_ACKNOWLEDGE`

Signal to tell the laptop that this phone acknowledges the signal to start transferring the recorded video file to the laptop.

3.11.2.8 STOP

`com.example.android.camera2video.Signal.STOP`

Signal to tell the camera to stop recording video.

3.11.2.9 STOP_ACKNOWLEDGE

`com.example.android.camera2video.Signal.STOP_ACKNOWLEDGE`

Signal to tell the laptop that this phone acknowledges the signal to stop recording video.

The documentation for this enum was generated from the following file:

- main/java/com/example/android/camera2video/Signal.java

Index