Drone Air Traffic Control System

Software to Coordinate Automated Drones, Safely

INSTITUTE of TECHNOLOGY CARLOW

Institiúid Teicneolaíochta Cheatharlach

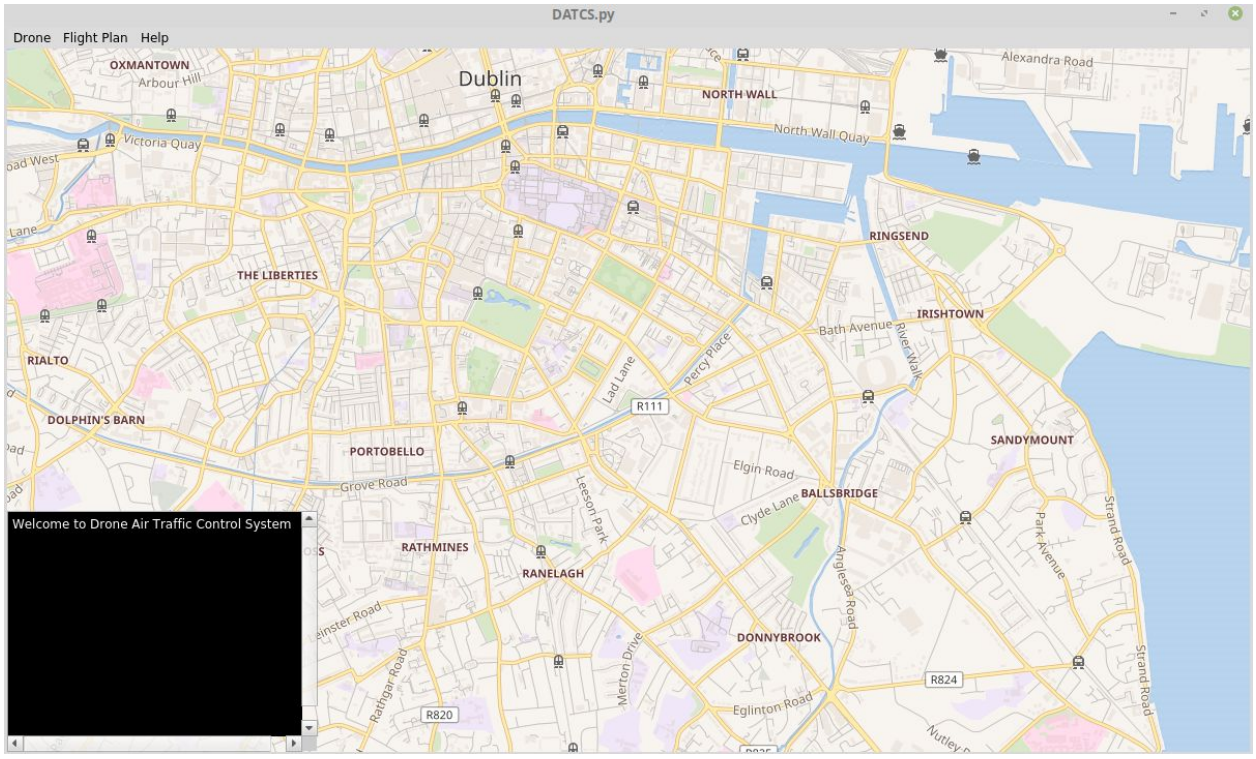
Ronan Donohue - c00208501

Introduction

The main goal for this project was to create an air traffic control algorithm for automated drones. Whilst drones are airborne there is always a risk of drones colliding with one another, so some sort of collision avoidance/prediction mechanism was also required. In the absence of access to a multitude of drones for testing, simulated drones were created to help mimic drone behaviour within the application. The application was also tested with an actual drone, a Parrot Bebop 2. The technologies used in development were Python 3.7, Qt5.9 (specifically Qt Quick/QML), Mysql and pymysql, and the pyparrot library developed by Dr. Amy McGovern.

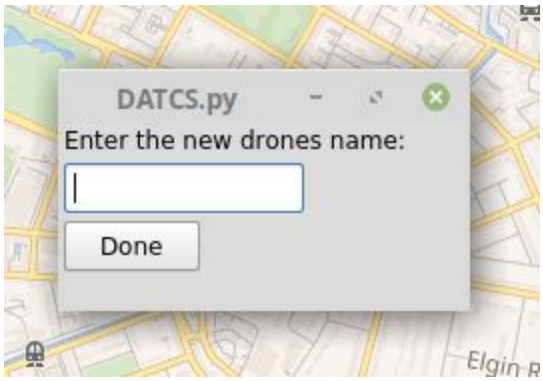
Description

Home Screen:



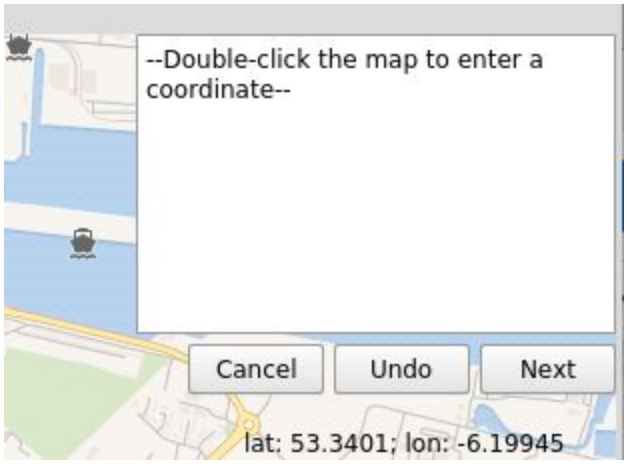
This is the main home screen for Drone Air Traffic Control System. The user is presented with a map, based off of their current location. The location data is retrieved from their ip address based off of geocoder. Using this location data, the Map is given a center point and is drawn. A terminal window in the bottom left displays a greeting to the user. As drones begin flying their flight plans and receive instructions from the AirTrafficController module, the terminal window Will populate with information to be displayed to the user. The menu at the top allows the user to add simulated drones, add a flight plan, display information about how to use the application and quit the application. The map can be moved around by using the mouse, and the user can zoom in and out.

Add Sim Drone:

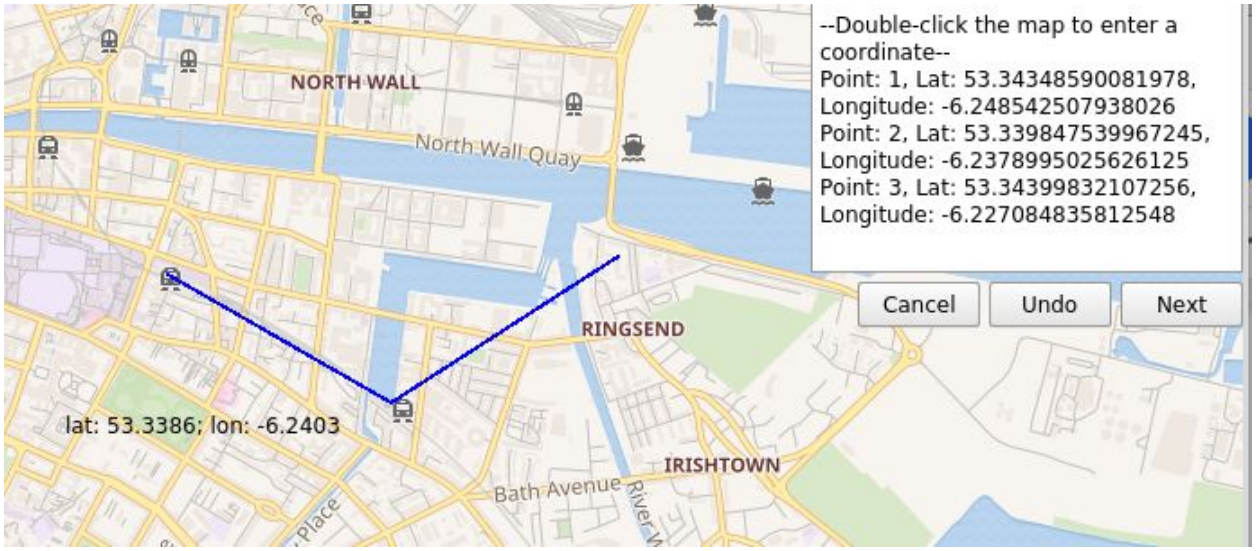


When the user wishes to add a simulated drone, they will need to click on the *Drone* menu, and select *New Sim Drone*. The user will be prompted to enter a drone name. Once the drone has been created, these names are unique and the user will be warned that the duplicate name is already in use. After entering a valid name, the user can choose where to place the drone using the mouse. A label will appear beside the mouse cursor instructing the user to double click to place the drone. Once placed, a red circle will appear on the map. This red circle represents the simulated drone. At this point, the user can right-click this red circle and a context menu will be displayed to them. Certain options will be greyed out and the user cannot select them. Some are options like Speed Up and Slow Down that are not applicable to a stationary drone, others are options that only a real drone can perform. To assign a flight plan to the drone, the user must select “Assign a Flight Plan” and select a flight plan from the displayed list of flight plans. Once assigned to a drone, a flight plan cannot be reassigned until the drone is no longer active. To begin running a flight plan once one has been assigned, the user just has to double click the red circle representing the drone.

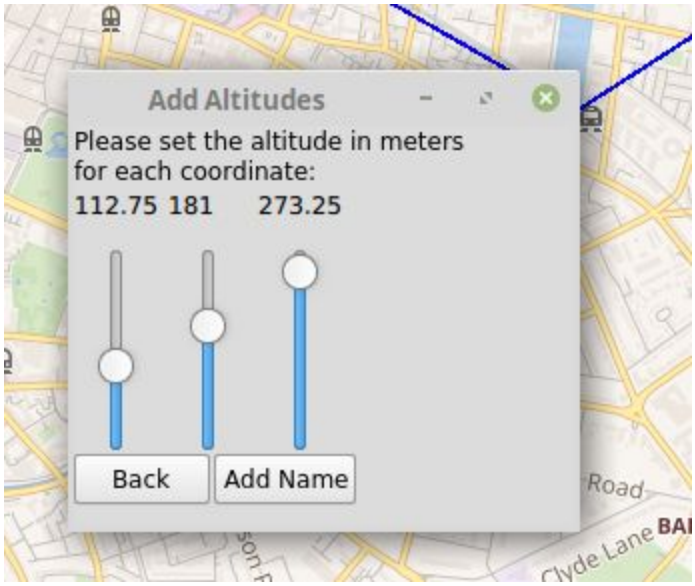
Add a Flight Plan:



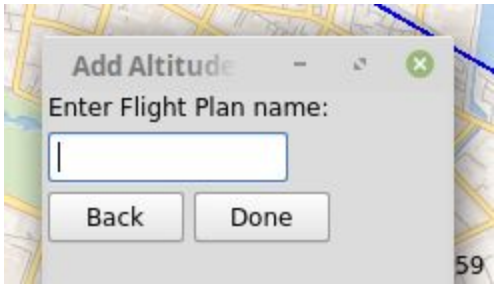
To create a flight plan, the user needs to select *Flight Plan -> New Flight Plan*. A Textbox (non-editable) will be displayed in the top right corner instructing the user to double click the map to enter a coordinate. A label beside the mouse cursor will display the latitude and longitude of the location the mouse cursor is pointing to.



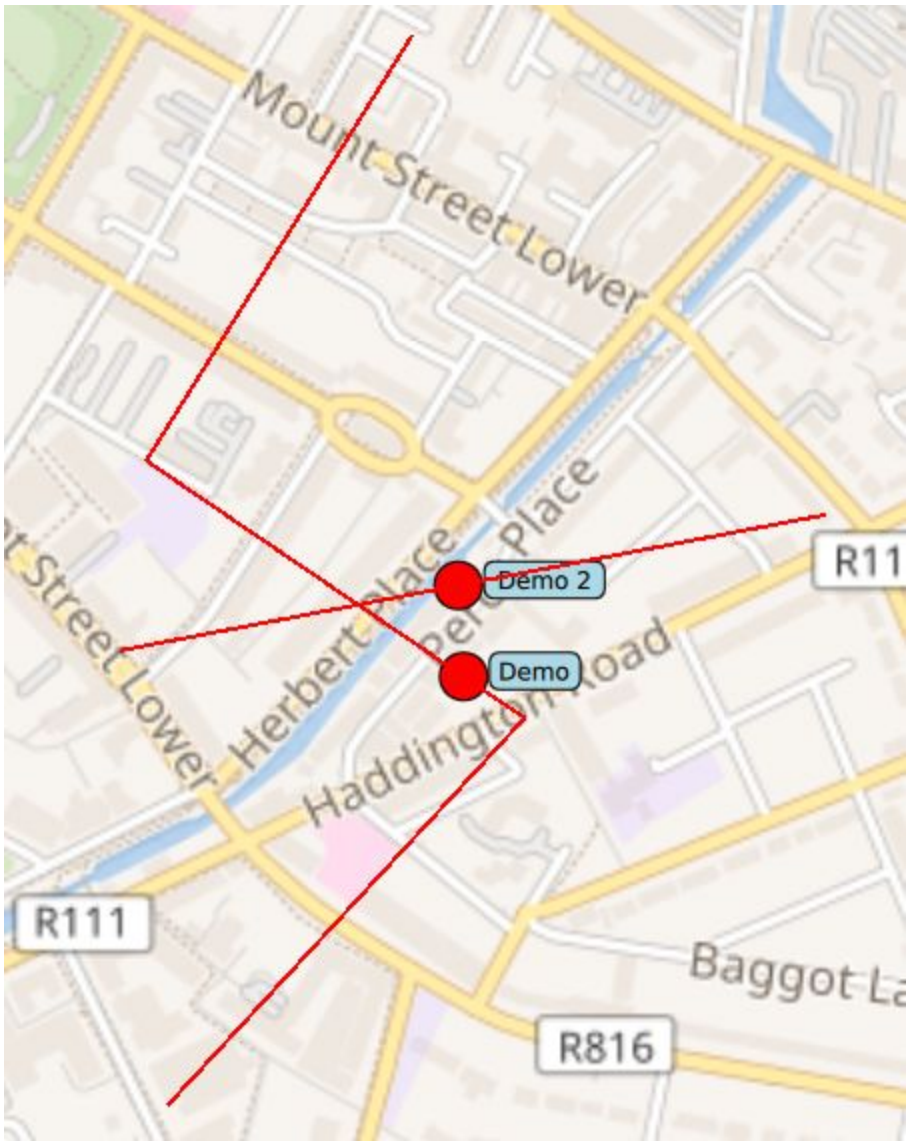
Once the user double clicks a point on the map, the corresponding latitude and longitude are captured and stored temporarily. As the user adds more points, a blue line will be drawn to display the connection between all points. If the user makes a mistake, they can press the *Undo* button to remove the last point they added. If the user decides they want to do something else instead, they can select *Cancel*. If the user is satisfied with their flight plan, they can select *Next* and move forward to the next screen.



The user will need to add the altitudes to each point selected in their flight plan. The altitude of any point is not stored in the map and the user must input an altitude for each desired point on the flight plan. The user will not be allowed to progress past this screen if any point has a value of 0 for safety reasons. If the user feels they have made an error, they can return to the previous screen. If they are happy with the altitudes that are set, they can then give a name to their flight plan and once that's done, it is ready to be assigned to a drone.



Drones in flight:



To have a drone begin its assigned flight plan, the user only needs to double click the drone once. It will begin moving towards the first point in the flight plan immediately. The proposed flight route will be shown to the user in red. Should multiple routes intersect, the user should be aware that if the drones have a similar altitude as well, a collision is imminent. While there is a CollisionPredictor module built into the app, prevention should be the first port of call in flight plan setup. While running a flight plan, should the user wish to recall a drone, they can right click on the drone in question and select *Return Home*. Similarly, if the user needs to speed up or slow down their drone, they can select *Speed Up* or *Slow Down* by right clicking the drone and selecting either option from the context menu.

New Real Drone:

To create a new real drone instance in the application, the user must first have their computer connected to the drone wifi. When the user selects *Drone -> New Bebop Drone*, the application will pick up the drones emitted GPS locations (provided the drone is outside) and display a green circle on the map representing the drones location. Similarly to a simulated drone, the user can right click on the green circle and be presented with the same context menu that is associated with simulated drones. The main difference is that this context menu will have more options to choose from that a simulated drone.

Conformance to Specification:

This was the description for the project initially:
"The ultimate goal here would be an air traffic control system for drones. Assuming fixed travel routes for drones, the system would coordinate the drones in that airspace in a safe manner. Drones would communicate their telemetry data (GPS locations, altitude, speed, battery power, etc) to the control system, which would then issue commands to the drone depending on other drones in the airspace."

There were a number of points listed as 'targets' or 'goals' that would be good indicators of progress during the course of the project. They were as follows:

- Application runs on a desktop or phone
- Travel routes defined
- Drone data being received by the control system
- Controller implements a basic air traffic control algorithm
- Drone responds to commands issued by the controller
- Visualisation of the drone's route in real time
- Access drones on board camera and display on screen
- User interface for setting up flight plans
- Simulated with multiple drones
- Works with multiple real drones

The submitted implementation matches a lot of the above specifications as much as possible. The application can be run on a desktop (tested mostly on Linux Mint 19). The user can create a flight plan for a drone using the application, and once a drone begins flying a flight plan, the route is defined and visualised to the user. The AirTrafficController object responsible for most of the coordination of drones can send information to and receive information from simulated drones and the Parrot Bebop 2. The AirTrafficController object assigns a FlightPlan to each drone and plots the points the drone must traverse; the drone only moves when passed data by the AirTrafficController. If the AirTrafficController decides that a collision is imminent, the drones in question will be stopped before any further action is taken. When the user is connected to a real drone, such as the Parrot Bebop 2, the drone's onboard camera can be activated and displayed to the user via VLC.
Parts that proved more difficult were controlling the Parrot Bebop 2 drone and connecting to multiple drones; the second task proving impossible given the hardware constraints (a 5 year old Acer laptop with similar aged network card). While it was possible to connect to the Parrot Bebop 2, run a simple test where the drone would takeoff, hover for ten seconds and land, it proved more troublesome to move the drone to a specified location. When instructed to move the drone would take off, but remain in a holding position around where it took off, not moving towards the target location as was supposed to happen. At this point in the project, little time remained to debug this behaviour.

Technical & Personal Achievements:

Technical:

Over the course of the project, various technical obstacles were overcome. How to best capture the users desired flight plan info? A QML Map provided a good view to the user, but how to convert the mouse click to a point with latitude and longitude? What is best way to have this data sent by interprocess communication from QML to Python? Do you use Signals and Slots, or the more modern approach of Properties, datum that exists in both the Python model and the QML view? As this is a real time application for the most part, how quickly can your various functions run? Does the CollisionPrediction object complete its calculations in under a second before the AirTrafficController polls the currently active drones and invokes its decision-making functionalities again? How do we know that two or n drones are set for a collision? Is it better to have just-in-time collision avoidance or map all flight routes and determine when a collision may occur?

Knowledge of Python and QML was drastically improved over the course of this project; to gain better understanding of QML, there is little recourse available online other than to consult the documentation. Within Python, preferring dictionaries over lists for speedy lookups and developing and testing functions within Jupyter Notebook were *de rigueur* as the project wore on. Unit tests were written in PyTest in the beginning, but as development progressed, changing functionality in modules and then updating test cases was eating into development time. The notebooks assuaged this by providing a sandbox within which code could be tested until correct.
QML allowed for the use of inline Javascript to handle the necessary imperative aspects of a GUI in an otherwise declarative setting. QML works using the Model-View-Controller pattern by design; to pass information between the model and the view requires signalling and slots, or properties. pyQT/QML also sets up an event loop by default. Both these aspects proved invaluable in structuring the submitted implementation. Within the event loop, threads did not perform as expected and alternative methods for seemingly asynchronous behaviour was needed. QTimer events, such as the AirTrafficControllers repeated requests for drone information, proved a far more economic solution to threading in this instance.

There is always a learning curve associated with learning a new language or skill; the learning curve for QML was steep but once overcome, the language proved itself to be incredibly useful and featureful. It may be possible to build the entire project specification in only QML.

Global Positioning was another area to be learned when it came to coordinating drones. Prior to utilising the QGeoCoordinate module, various mathematical formulas were implemented to convert the latitude and longitude of a coordinate into Earth-Centered, Earth-Fixed XY

coordinates for easier line plotting in cartesian space. Distances between GPS coordinates were calculated using the Haversine formula, and each point along a straight line between points needed to be plotted so they could be passed to a drone. While the QGeoCoordinate class ultimately dealt with a lot of these issues, implementing these solutions manually gave an important grounding in GPS terminology and conventions.

Prior to discovering pyparrot, as a necessity to control a real drone the Parrot Bebop sdk was built and a working connection established to the drone via a laptop. Without pyparrot, the drone SDK, written in C, would have to be made available in Python. One approach considered was to use Cython to bridge the gap between languages.

Personal Achievements:

To be candid, I learned a tremendous deal from this project. Among other things, I learned that presentations are an important way to convey information to a group of your peers and like all things in life, they must be practised until perfect. While some people can make it look easy, I found this to be a challenge and it's one I will need to work on in the future.

"By failing to prepare, you are preparing to fail" - Benjamin Franklin. I never understood the importance of software design and the effect it can have on the quality of the code being produced until this project. After a false start to the project, it became apparent to me that smart, strategic design pays dividends in the long run and can help provide a safety net for a tired mind when things get hairy.

I really learned the value of good quality documentation. Qt has maintained the high quality of their documentation through every version they roll out and it has been nothing short of a blessing during this project. All QML knowledge I acquired during this project came straight from the Qt documentation.

"Courage is grace under pressure" - Ernest Hemingway. There is a lot of pressure on students to perform, get good grades, decide on a career, compete with their peers, write well, and communicate effectively. It is important to an individual that they learn how well they can handle pressure. This lesson is something highly personal to each person and what works for one individual will invariably fail for another. My experience this past year has given me some valuable insights into the effect pressure and stress (real or imagined) can have on the performance of a student. Going forward, I know now that when tasks start mounting, I should analyze how effectively I can handle them and to anticipate new challenges.

Review of Project:

What went right?:

A lot of specified functionality was implemented. Simulated Drones can be placed by the user, they can undertake flight plans that the user creates. In terms of real drones, a connection was m established and the drone video feed could be displayed. A rudimentary collision avoidance algorithm was implemented and all this information could be displayed to the user in the application.

What went wrong?:

Towards the end of the project, the real drone proved difficult to coordinate. While a successful connection was established and it could receive basic takeoff and land commands, it could not execute a flight plan the same as a simulated drone could. While there wasn't enough time remaining to assess what was really going on, it would seem that by having the AirTrafficController send the drone a moveTo each second, each consecutive moveTo was overwriting the previous one and the drone remained motionless, not knowing what to do. This is only speculative and further investigation is required.

What is outstanding/left to do?:

Additional work remains to be done with the real drone. The user should be able to edit flight plans once they've created them. Once a drone has been assigned a flight plan, the flight plan cannot be assigned to another drone. This is something that could be addressed in future work. Also, the GUI layout is very barebones, it would benefit from a tidying up.

What would be done differently?:

If time wasn't a factor, building the Parrot SDK in C and bringing it into Python using Cython would have been an interesting approach. Otherwise, the benefits of QML's inbuilt event loop and MVC pattern proved incredibly beneficial and were probably the right choice for the job at hand. Python provided a lot of complex functionality out-of-the-box that would have eaten up development time in any other langauge. QML and Qt are compatible with C++ so that is also another language option that could have been utilised going forward.

The implications of the technologies chosen:

By committing to QML and Python, a lot of material needed to be learned to get the application into a decent state to work with. This cost time. However, once learned, a lot of functionality became quite easy to implement, such as QGeoCoordinates taking over for a lot of GPS data and functions.

Module Descriptions:

AirTrafficController.py

This module is responsible for creating simulated drones and flight plans, passing flight plan and coordinate information to a drone, simulated or real, and ensuring that drones stop if a collision is imminent. It contains a number of pyqt Signals to emit information to the QML view, pyqt Slots to receive data from the QML view and a select number of pyqt Properties that exist in both model and view.

DroneController.py

This module acts as a 'pilot' . It is responsible for moving the drone to a coordinate specified by the AirTrafficController module. This module also relays drone telemetry data back to the AirTrafficController. Commands issued to the DroneController module from the AirTrafficController module are relayed to either the SimulatedDrone module or the DATCS_Bebop module.

SimulatedDrone.py

This module is meant to act as a 'dummy' drone the DroneController module pilots *in lieu* of a real drone. Common drone states such as "landing" and "flying" are included, along with drone speed.

Loader.py

This module handles all requests to the database. The FlightPlan and the AirTrafficController modules both have a Loader instance to make data acquisition easier.

FlightPlan.py

This module acts as an Abstract Data Type for flight plan information. Each instance of a FlightPlan object with have an index, which is used to traverse a list of dictionaries containing the flight plan information. Extraneous functionality is included for editing FlightPlan data, but is not implemented in the GUI.

DATCS_Bebop.py

This module uses functionality from the pyparrot library and some additional code to establish a connection to a Parrot Bebop 2 drone and control it using Python code.

CollisionPredictor.py

This module determines whether any of the airborne drones is going to collide with each other. If drones share a similar altitude, if their paths intersect and if the distance between the drones in question is below a certain threshold, the CollisionPredictor module will decide which drones need to be stopped.

datcs_main.qml

This file contains most of the code used to display the GUI to the user, including the menus, Map, MapPolylines, mouse functionality and various windows and dialogs that are displayed to the user.

Drone.qml

This file is a custom MapQuickItem that is used to draw the various drone blips on the screen and add custom functionality to each blip, such as different mouse functionality and inline Javascript.

Acknowledgements:

I would like to thank Dr. Oisin Cawley, whos support, guidance and assistance has been invaluable throughout the course of this project.