Informatics Large Practical
Coursework 2 Report

Chris Perceval-Maxwell
s1839592

December 4, 2020

Contents

1 Software Architecture Description

1.1 Outline

This section will provide an extensive description of the software architecture, capturing the significant design liberties and conveying their importance.

1.2 Requirements

The goal of this project was to develop an application to control a drone, within a confinement area on a map, that will take the readings of 33 sensors on a given day and also avoid a set of buildings marked as no-fly zones. If the drone was able to collect readings from all the sensors, it should attempt to fly back to where it started.

1.3 Classes

My implementation requires 10 classes in total. They consist of the main class to run the application: App.java, and seven classes to divide up the tasks of connecting to the server, moving the drone and taking sensor's readings:

- Drone - This is required so we can have a singular object to control. The drone has many tasks to complete so it is neater to keep all its functionality in its own class. It also means in the future more than one drone could be controlled at the same time within the app,
- HttpConnection - This is required to create a connection to the server of our choice. We could connect to more than one server if we needed to in the future,
- JsonParser - This is used to parse specific files from the server into their respective classes,
- Location - This is required to store the location of any required objects, right now it used to store the location of the drone as an object and the individual sensors as others.
- Map - This is required to specify what the drone will be navigating. Right now it creates the confinement area, no-fly zones and sensors,
- PollutionLookUp - This is used to find the GeoJSON properties to apply to a marker by taking in the information read from the sensor by the drone,
- Sensor - This is required as the drone will need to interact with a sensor to take its readings;

and 2 classes to parse the air-quality and What3Words [1] location JSONs into so other classes within the application can access their information: AirQualityData and Location-Details.

1.4 Static Model

Figure 1 is a UML class model that denotes the variables and methods of each class.

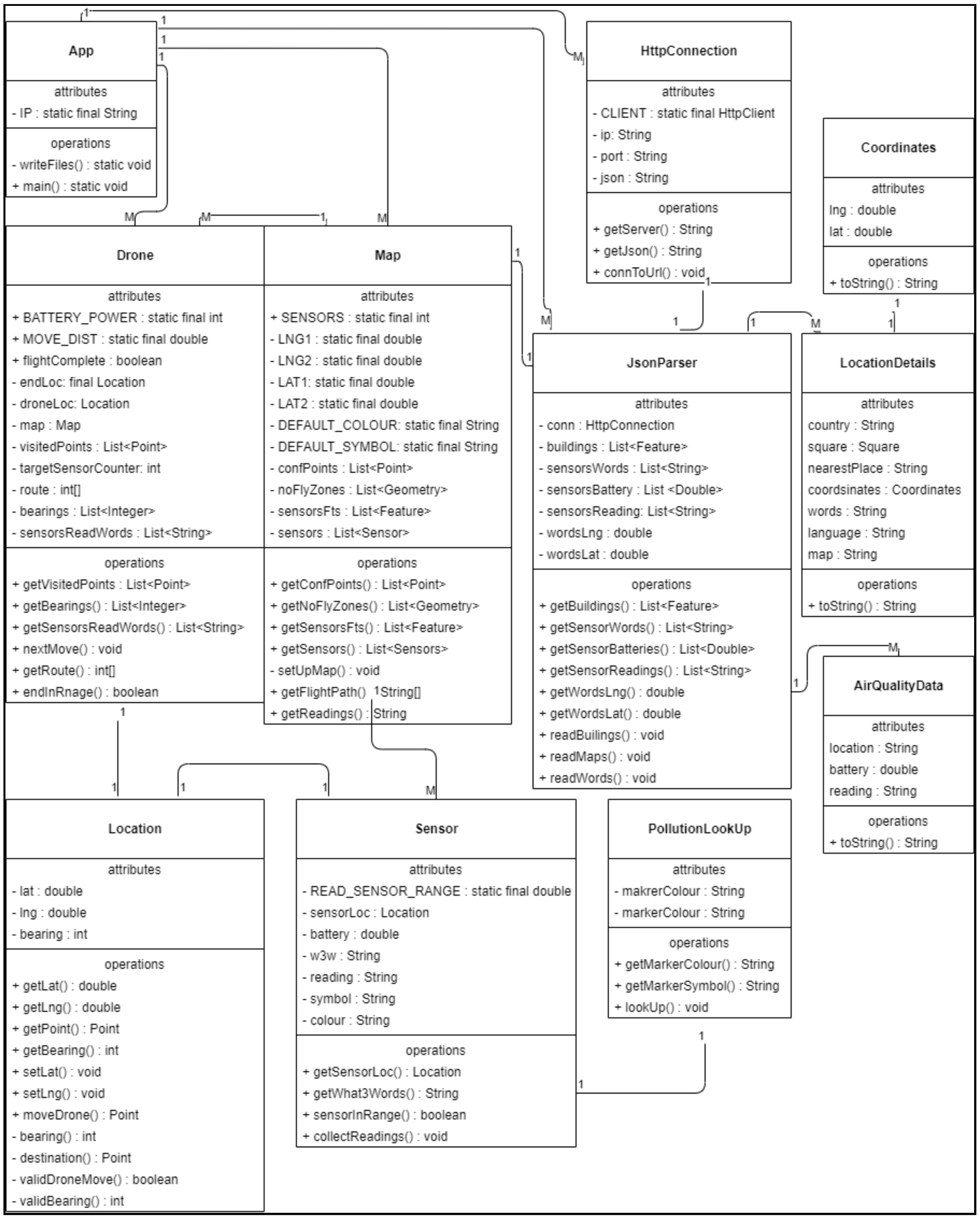

Figure 1: UML Class Model

1.5 User Interaction

Figure 2 is a UML sequence diagram that describes how all the classes interact with each other for the core functionality of application.

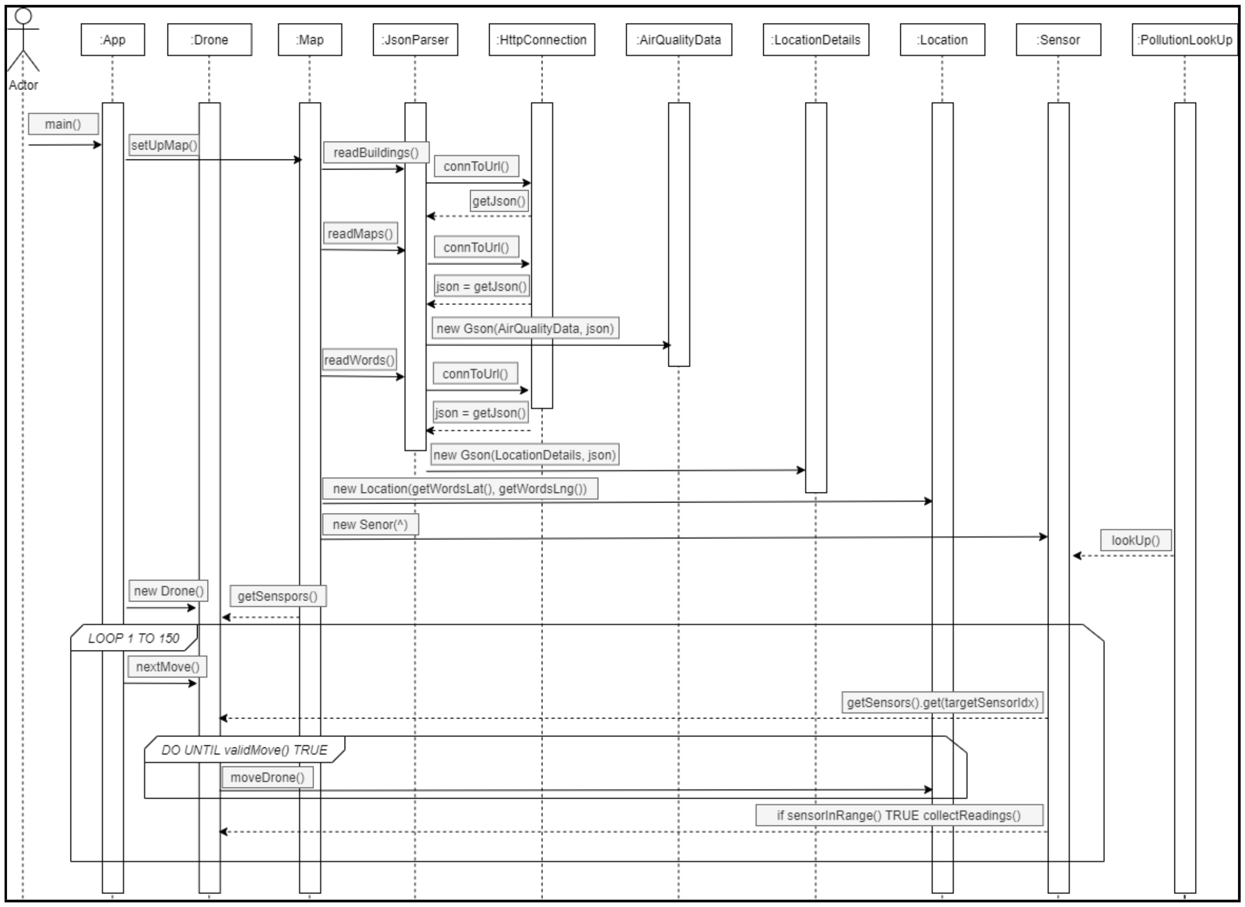

Figure 2: UML Sequence Diagram

1.6 Third-Party Software and Libraries

This project utilised three third-party dependencies:

- Mapbox Java SDK GeoJSON [2] - This is is used handle and create GeoJSON features.
- Mapbox Java SDK Turf [3] - This has many useful methods and constants for navigation calculations.
- Gson API [4] - This is used to parse the JSON files to Java classes.

2 Class Documentation

Here I will give a comprehensive description of each class. I have excluded rudimentary methods, namely getters and setters.

2.1 AirQualityData

This class is used to parse the `air-quality-data.JSON` for a specific date (the file can be found on the server via `/maps/YYYY/MM/DD/` for the required date). This file holds the air-quality data of each sensor on the map for that date. The JSON file is composed of an array of objects of the same schema of the locations, battery charge and readings.
The `toString()` method allows us to pull the information from the object.

2.2 App

This class is the one run by the user. It sets up everything on the map, flies the drone and writes the outputs to the disk. It has 2 methods:

2.2.1 main(String args[])

This method sets up everything required for the drone's flight. There are 7 arguments parsed in for the flight's details, `args[0]` - the day, `args[1]` - the month, `args[2]` - the year, `args[3]` - the starting latitude, `args[4]` - the starting longitude, `args[5]` - the seed and `args[6]` - the port for the server. My implementation does not use the seed so it is not assigned to variable, but it could easily be added.
Firstly, it creates a connection to the server and a parser that uses it. Next it attempts to set up the map using the parser and the date the user has entered - if it is successful it will create a drone for the map at the starting point specified by the user.
Secondly, it begins the flight of the drone. It moves the drone a maximum of 150 times, denoted by the `BATTERY_POWER` constant declared in the `Drone` class.
Lastly it calls to write the flight path and readings to their respected files, see Section 2.2.2.
This method does not return anything.

2.2.2 writeFiles(String[] flightpath, String readings, String yyyy, String mm, String dd)

This method writes the 2 files required to the disk. It parses in 6 variables: `flightpath` - an array of strings for all the moves, `readings` - a GeoJSON string of the sensors read and the flight path, `yyyy` - the year of the flight, `mm` - the month of the flight and `dd` - the day of the flight.
It then attempts to write `flightpath` to a text file, line by line, and `readings` to a GeoJSON file.
This method does not return anything.

2.3 LocationDetails

This class is used to parse the `details.JSON` for a specific What3Words location (the file can be found on the server via `/words/WORD1/WORD2/WORD3/` for the required location). The file holds the location details for a selection of What3Words string. The JSON file is composed of a single object with a nested schema of the country, coordinates and areas of the What3Words square, etc. This implementation is only interested in the *coordinates* and *words*, but can be easily expanded to use all the information as the class is ready to spit out all the information.
The `toString()` method allows us to pull the information from the object.

2.4 Drone

This class contains methods and calls to other classes that aid in navigating the drone around the map. The constructor creates an object with the map the drone will be navigating and the starting Point. It also adds the starting position to the list of visited locations and finds the route in which to visit the sensors, see Section 2.4.2. There are 3 methods:

2.4.1 nextMove()

This method is called in the `App` class at most 150 times. It deals with moving the drone to its next, valid, position, on its journey to read all the sensors on the map.
Firstly, it checks whether it still needs to visit any sensors. If not, its target location is the starting location. However if it does, find the next sensor in the route it is yet to visit and set it as its target, then find its location.
Next it will move the drone to a new valid position towards the target sensor, see Section 2.7.1 and update the drone's location.
After it has moved, attempt to collect the readings of the target sensor - it does so by checking if it is range of the sensor. If it is, collect the readings - see Sections 2.10.1 and 2.10.2 respectfully. If it is not in range, add `null` to the list used to track the location when a move reads a sensor.
Lastly, update the the list of visited points with the current position and the list of bearing with the bearing it took to get there.
This method does not return anything.

2.4.2 getRoute()

This method finds an efficient route in which to visit all the sensors on the map, using a Greedy Best-First algorithm, see Algorithm 1. This algorithm finds the indices of the sensors to visit. When it has determined the route, it will return them as any integer array.

2.4.3 endInRange()

This method simply checks whether the drone is close enough to the starting position that it can end its flight. It does so by finding the Euclidean distance between the drone's location and the starting location. It is called when the drone knows it has visited all the sensors.
The method returns the truth value of whether it is in range or not.

2.5 HttpConnection

This class is used to connect to the server of choice. The constructor creates an object with a given IP address and port. There is 1 method:

2.5.1 connToUrl(String urlString)

This method is used to connect to the server. It has 1 argument: `urlString` - the complete URL of the file we wish to request.
It then attempts to request said file, and if it does so successfully, i.e. a response code of 200, it will assign the contents of the page to a variable which can be accessed with a getter. If it is unsuccessful, it will exit the application.
This method does not return anything.

2.6 JsonParser

This class deals with parsing the JSON and GeoJSON files on the server. The constructor creates an object with the connection to the server, see Section 2.5.1. There are 3 methods:

2.6.1 readBuildings()

This method requests the GeoJSON file `no-fly-zones.GEOJSON` from the server and takes the features, i.e. the individual no-fly zones, and adds them to a list of type `Features`.
This method does not return anything.

2.6.2 readMaps(String yyyy, String mm, String dd)

This methods has 3 arguments that make up the date of the flight the user has requested: `yyyy` - the year, `mm` - the month and `dd` - the day.
It then requests the `air-quality-data.JSON` from the server, and parses it into the class `AirQualityData`, see Section 2.1.
It then retrieves the What3Words location, the battery charge and the reading of each sensor in the list of sensors for the given date.
This method does not return anything.

2.6.3 readWords(String w1, String w2, String w3)

This method has 3 arguments that make up the location (of a sensor): `w1` - the first word, `w2` - the second word and `w3` - the third word.
It then requests the `details.JSON` from the server, and parses it into the class `LocationDetails`, see Section 2.3.
It then retrieves the longitude and latitude of the What3Words location.
This method does not return anything.

2.7 Location

This class is used to keep track of, and control, objects' locations. In my implementation it is used for the drone and the sensors. The constructor parses in the object's longitude, latitude and an initial bearing of 000 degrees. There are 5 methods:

2.7.1 moveDrone(Map map, Location targetPos)

This method is used to move the drone to a valid position on its journey to the target sensor. It has 2 arguments: `map` - the map the drone is navigating and `targetPos` - the location of the target sensor.
It does so by finding the bearing to the target location (see Section 2.7.2) and attempting to move the drone at that bearing (see Section 2.7.3). It then checks if the proposed move was valid, i.e. doesn't go out-of-bounds or through a no-fly zone (see Section 2.7.4). If it's valid, state that the drone is allowed to move. If it isn't, recalculate the bearing and check the validity of the move again, see Algorithm 3.
This method returns the GeoJSON Point of the next when it is sated that the proposed move was valid.

2.7.2 bearing(Location prevPos, Location targetPos)

This method calculates the bearing between 2 points. It has 2 arguments: `prevPos` - where the drone was and `targetPos` - where the drone wants to go.
First it must find the difference in latitude and longitude of the 2 locations. It then uses the arc-tangent of all 4 quadrants of these 2 values to find the bearing. We must then adjust our values so that it follows the convention of 000 degrees representing East, 090 degrees North, etc. Then we round to the nearest 10 and ensure it is still within 000-350 degrees, see Section 2.7.5.
This method returns the integer that is the bearing.

2.7.3 destination(Location prevPos, int bearingToTarget)

This method find the GeoJSON Point of the move using the calculated bearing. It has 2 arguments: `prevPos` - where the drone was and `bearing` - the bearing to the next location. We use rudimentary trigonometry to calculate new longitude and latitude at the given

bearing.
This method returns a GeoJSON Point of the destination.

2.7.4 validDroneMove(Map map, Point nextPos)

This method checks whether the proposed move cross any boundaries of the confinement area or no-fly zones. It has 2 arguments: `map` - the map the drone is navigating and `nextPos` - the proposed destination.
We check whether the line between the the last point and new point of the drone intersects any of the lines we should not cross. See Algorithm 2 for how we check this. We apply this to the confinement areas and every no-fly zone.
This method returns the truth value of the validity of the move.

2.7.5 validBearing(int bearing)

This method simple ensure the bearing remains with 000-350 degrees. It has 1 argument: `bearing` - the bearing we wish to validate.
If the value of the bearing is < 0, we add 360 degrees, whereas if it is $>= 360$ degrees, we subtract 360 degrees.
This method returns the integer value of a valid bearing.

2.8 Map

This class sets up the map for the drone to navigate and the flight the drone has taken. The constructor creates an object with the JSON parser for the desired server, and the date of the flight requested by the user. It also calls the `setUpMap` method, see Section 2.8.1. There are 3 methods:

2.8.1 setUpMap(JsonParser parser, String yyyy, String mm, String dd)

This method sets up the flying confinement area, the no-fly zones and the sensors to visit. It has 3 arguments: `parser` - this is a JsonParser for the chosen server, `yyyy` - the year of the flight, `mm` - the month of the flight and `dd` - the day of the flight.
To set up the confinement area, we have 4 constants: the longitude and latitude of the Northwest corner and Southeast corner of the area. From this we create an array list of the 4 vertices as GeoJSON Points.
Secondly, we use `parser` to request the `no-fly-zones.JSON`, then get the list of GeoJSON Features and add each the geometry of each no-fly zone to a list.
Thirdly, for each sensor, we parse their What3Words location, their battery charge and air-quality reading. We create a sensor object and add it to a list of sensors, then convert it to a feature and it to another list with the default GeoJSON colour and symbol marker properties (this list will be used for outputting our readings as a GeoJSON).
This method does not return anything.

2.8.2 getFlightPath(List<Point> dronePoints, List<Integer> bearings, List<String> words)

This method sets up the moves of the drone to be written to the disk. It has 2 arguments: `dronePoints` - a list of GeoJSON Points the drone visited, `bearings` - a list of bearings the drone took on each move and `words` - a list of the locations for each move: the What3Words location when the drone read a sensor's data, null if it didn't.
It then stores each move to an index of an array of strings. One entry contains the following information: the move number, the longitude and latitude of the drone's current position, the bearing it decided for the next move, the next longitude and latitude of the drone and finally a What3Words location if applicable.
This method returns the array of moves, ready to be written to individual lines of a text file.

2.8.3 getReadings(List<Point> dronePoints)

This method sets up the GeoJSON features of the senors that have been read and a graphical representation of the path. It has 1 argument: `dronePoints` - a list of GeoJSON Points the drone visited.
It creates a GeoJSON LineString from all the Points in `dronePoints` and converts it to a feature. Then it makes a GeoJSON FeatureCollection of the flight path and the sensors, represented as Points.
This method returns the JSON of the senors that have been read and a graphical representation of the path.

2.9 PollutionLookUp

This class is used to look up the graphical representation of a sensor's readings. The constructor does not take any arguments. It has 1 method:

2.9.1 lookUp(double battery, String reading)

This method assigns a colour and symbol to a sensor's readings. It has 2 arguments: `battery` - this is the battery charge of the sensor and `reading` - this is the reading of air-quality of the sensor.
It first checks if the sensor wasn't able to take a reading, i.e. the reading was "null" or "NaN". If so, assign the appropriate colour and symbol. If it did take a reading, but the battery charge was under 10%, assign the same properties. Lastly, if the reading was valid, assign the appropriate colour and symbol depending on the value of the reading.
This method does not return anything.

2.10 Sensor

This class keeps track of all the sensors and their properties. The constructor creates an object with the location of the sensor, its What3Words location, the battery charge and the air-quality reading. There are 2 methods:

2.10.1 sensorInRange(Location droneLoc)

This method is used to check whether the drone is in range of it's target sensor. It has 1 argument: `droneLoc` - the location of the drone.
It takes said sensor and checks if the Euclidean distance between the 2 locations. If the drone is close enough to collect the sensor's readings, state it is in range of the sensor.
This method returns the truth value of whether the drone is in range of the sensor.

2.10.2 collectReadings(Map map, int targetIdx)

This method collects the readings of a sensor if the drone was in range. It has 2 arguments: `map` - the map the drone is navigating and `targetIdx` - the index value of the sensor the drone is attempting to read.
To collect the readings, it will update the GeoJSON FeatureCollection stored by the map with the appropriate marker properties from the sensor object - this is found by creating a PollutionLookUp for the sensor, see Section 2.9.1.

3 Drone Control Algorithm

Here I will discuss the algorithm used to control the order in which the drone visits the sensors and other logical decisions the drone must make on its flight, along with 2 example flights with analyses.

3.1 The Algorithm

3.1.1 Determining the Route of Sensors

I implemented a standard Greedy Best-First algorithm to find a decently efficient route of sensors for the drone to follow. Greedy simply finds the closest node, removes it from the list of nodes to visits, then finds the closest to that one. Algorithm 1 is the pseudocode of the Greedy algorithm implemented - it will return the indices of the sensors to visit.

Algorithm 1 Greedy Best-First Algorithm

```
procedure GREEDY()
 1: $list\_of\_sensors\_points\_copy \leftarrow list\_of\_sensors\_points$
 2: $route \leftarrow \{\}$
 3: $next\_sensor \leftarrow min\_dist(starting\_location, list\_of\_sensors\_copy)$
 4: $route \leftarrow route \cup index\_of(next\_sensor, list\_of\_sensors\_points)$
 5: for each $sensor$ in $list\_of\_sensors\_points$ do
 6:     $next\_sensor \leftarrow min\_dist(next\_sensor, list\_of\_sensors\_copy)$
 7:     $route \leftarrow route \cup index\_of(next\_sensor, list\_of\_sensors\_points)$
 8:     $list\_of\_sensors\_copy \leftarrow list\_of\_sensors\_copy - next\_sensor$
 9: end for
10: return $route$
end procedure
```

This method proved effective enough in general and was pretty efficient for the dates that we were required to submit. It obviously has its downfalls as it doesn't take into consideration the rerouting to avoid no-fly zones before the drone begins its flight, i.e. the closet physical distance isn't always the lowest cost to travel to. More comprehensive search algorithms, such as A*, where a heuristic is used and an overall cost to the goal is taken in consideration, would likely prove much more efficient than Greedy.

3.1.2 Avoiding No-Fly Zones and Leaving the Confinement Area

To prevent the drone from flying into, or over, the areas deemed as no-fly zones or flying out-of-bounds, I checked that the proposed journey of the drone from the current position of to the next did not intersect any of the boundaries. If it did, it must recalculate the bearing at which to travel in and check that proposed move again. Algorithm 2 is how I checked if there were any intersections for a given no-fly zone.

Algorithm 2 Determine Validity of Move

```
procedure VALID-DRONE-MOVE()
 1: $proposed\_move \leftarrow create\_line(coords(current\_pos), coords(next\_pos))$
 2: $corss\_barrier \leftarrow false$
 3: for each $vertex$ in $list\_of\_area\_vertices$ do
 4:     $area\_barrier \leftarrow create\_line(coords(vertex), coords(next\_vertex))$
 5:     if $intersect(proposed\_move, area\_barrier)$ then $cross\_barrier \leftarrow true$
 6:     end if
 7: end for
 8: return $cross\_barrier$
end procedure
```

Algorithm 3 is how the drone recalculates its bearing. It does so by incrementing alternations of +10 and -10 degrees from the bearing of the last valid move, until the move is valid. If it attempts to back to it's previous point (i.e. a difference of 180 degrees) it will go back to the bearing it just tried but the direction of change will be the opposite.

Algorithm 3 Recalculate Bearing

```
procedure NEW-BEARING()
 1: $increment\_bearing \leftarrow false$
 2: $counter \leftarrow 1$
 3: while not a VALID-DRONE-MOVE() do
 4:     if $increment\_bearing$ then
 5:         $bearing \leftarrow bearing + 10 * counter$
 6:         if $difference(bearing, last\_valid\_bearing) = 180$ then $bearing \leftarrow bearing - 10$
 7:         end if
 8:         $increment\_bearing \leftarrow false$
 9:     else
10:         $bearing \leftarrow bearing - 10 * counter$
11:         if $difference(bearing, last\_valid\_bearing) = 180$ then $bearing \leftarrow bearing + 10$
12:         end if
13:         $increment\_bearing \leftarrow true$
14:     end if
15:     $counter + +$
16: end while
end procedure
```

3.1.3 Returning to the Start

When the drone has visited all the sensors, if it does so within 150 moves, it will set its target as the starting location and terminate the flight if reaches within range of the starting location before 150 moves. The drone will always attempt to visit all the sensors before returning to the start. There is the possibility of the drone visiting every sensor but having too few remaining moves to reach the starting location.

3.2 Example Flight Paths

I have chosen two flights that I found interesting. The first one being 12/12/2020, Figure 3. This is a very good demonstration of the done avoiding every no-fly zone on its journey, and returning to its starting location, in 119 moves.

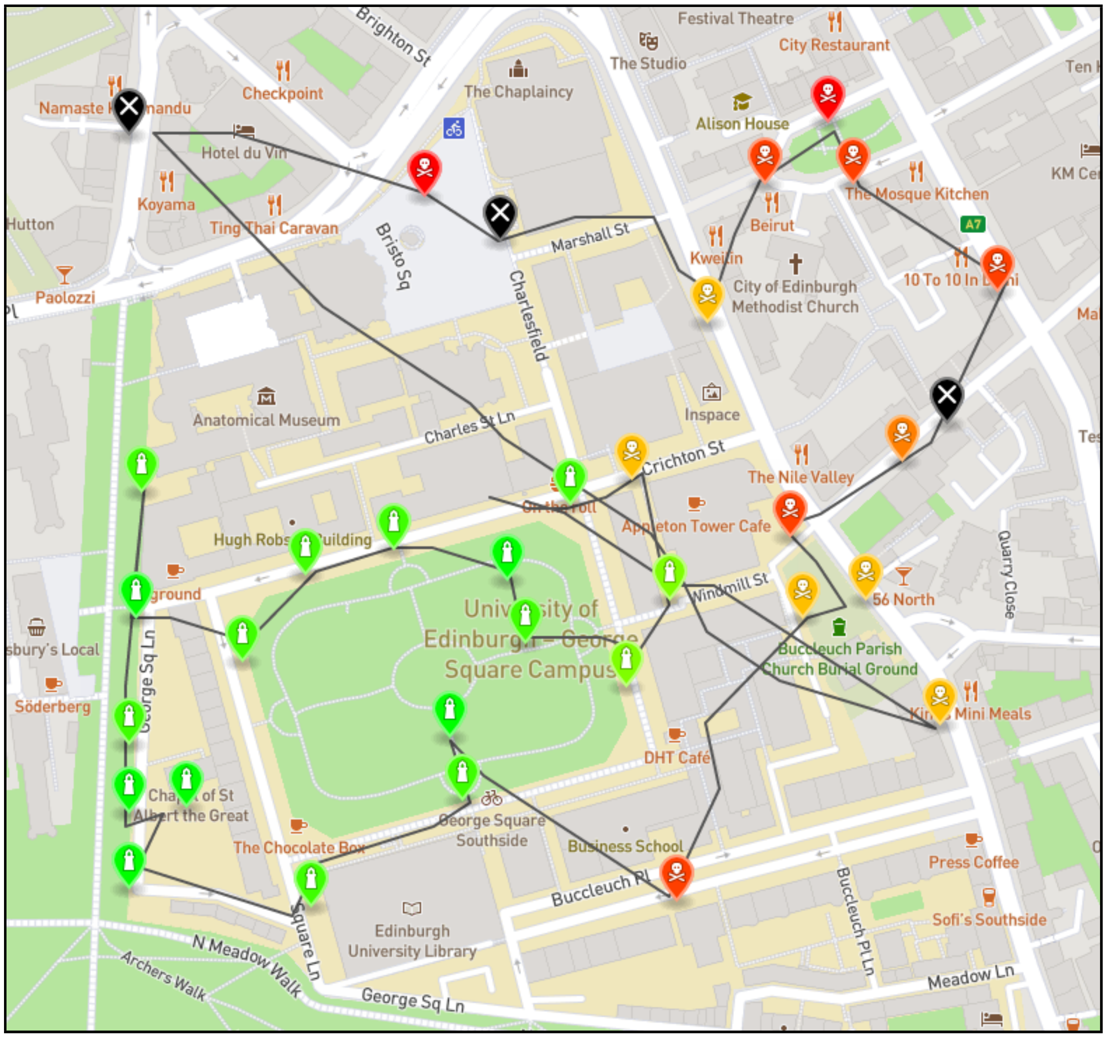

Figure 3: Drone Flight 12/12/2020

The second flight I have shown, Figure 4, demonstrates the weaknesses in my method of recalculating the bearing. Outside the Informatics Forum, beside Uniview, since it isn't logically choosing which direction to recalculate to, the first valid one unfortunately leads it on a much longer path around the no-fly zone. It also finishes very close the maximum number of moves, clocking in at 147.

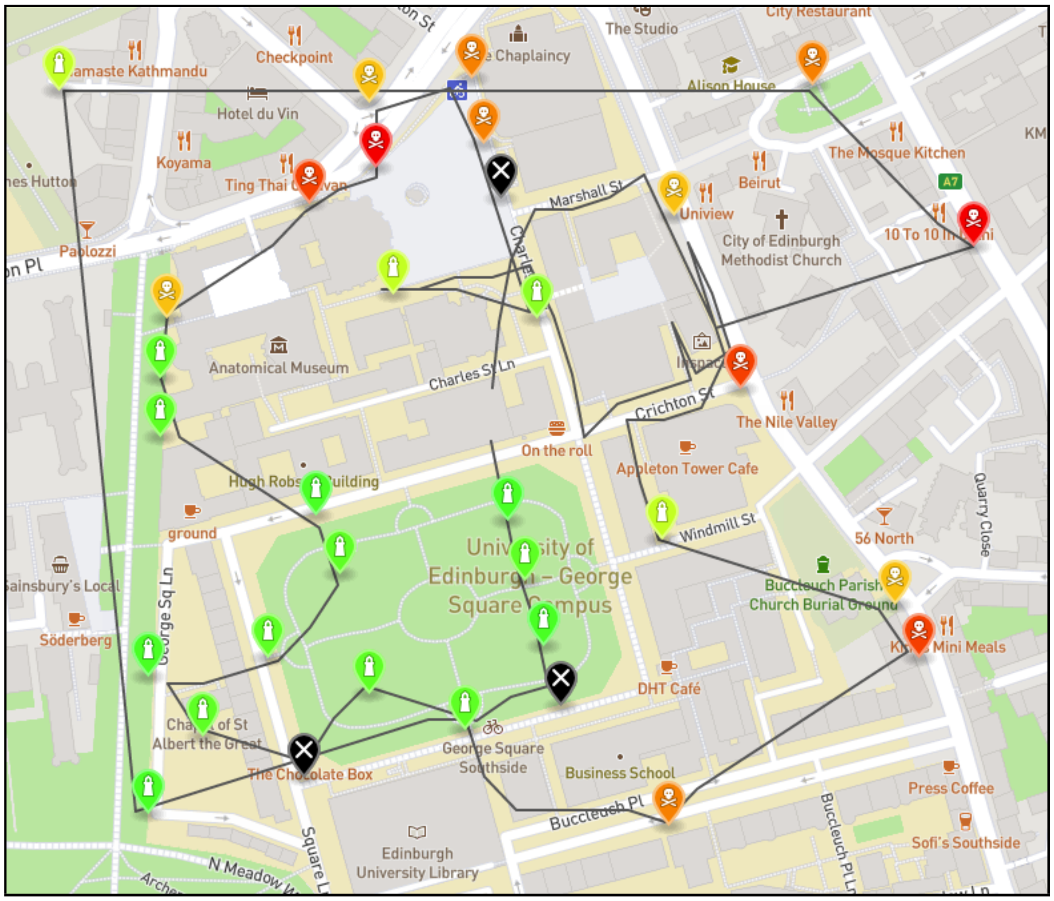

Figure 4: Drone Flight 17/09/2020

Bibliography

[1] `https://what3words.com/`, Accessed 01/12/2020.

[2] `https://docs.mapbox.com/android/java/overview/`, Accessed 01/12/2020.

[3] `https://docs.mapbox.com/android/java/overview/turf/`, Accessed 01/12/2020.

[4] `https://javadoc.io/doc/com.google.code.gson/gson/latest/com.google.gson/module-summary.html`, Accessed 01/12/2020.