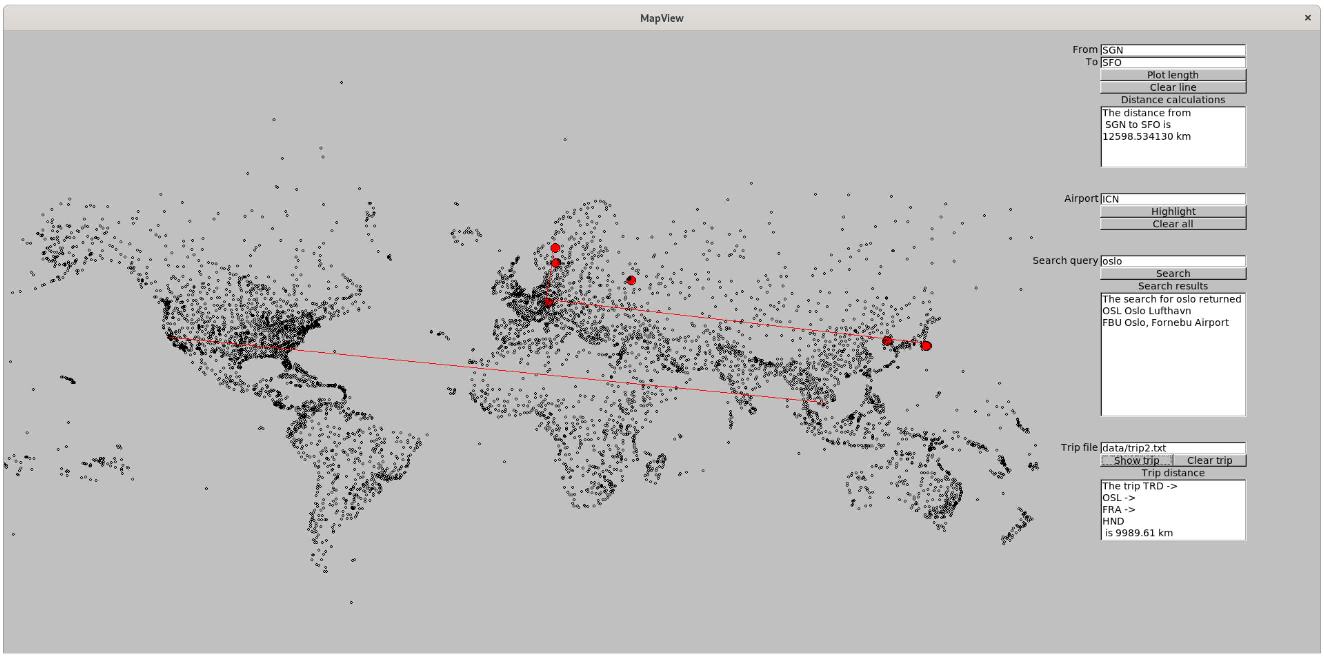

Figure 1: Den komplette applikasjonen

Del 3: Airports of the world

This part 3 of the exam asks you to complete a small graphical application for interactively exploring airports around the world and the routing between them. Part 3 consists of 12 sub-questions.

The assignment centers around the dataset provided by openflights.org[1] containing all of the worlds airports. The dataset, in a slightly simplified form, is included with this assignment (found in `data/airports.csv`). The code for parsing the file is already implemented for you, so you do not need to implement this yourself.

Since the location of the airports in the dataset are provided as geographic coordinates on the spherical earth, we need to project them onto a plane using a map projection method. An example of such a method is the ubiquitous Mercator projection[2]. In this assignment, we use a variation of the Mercator projection called Web Mercator [3], which is commonly used by online mapping services.

When complete, the application provides the following features. The assignments which need to be completed in order to implement a feature are listed in the parentheses.

- Plotting of airport locations on a map (Assignment U1, A1)
- Highlighting a specific airport on the map (Assignments U1, A1, A2, E1, E3)
- Searching for airports based on their name (Assignments A3, E1, E4)
- Calculating the distance between two airports (Assignments U1, U2, A1, E1, E5)
- Plotting and calculating the total length of trips (Assignments U1, U2, A1, A2, E1, E6)
- Validation of airport codes (Assignment E2)

A screenshot of the complete application can be seen in Figure 1.

[1] `https://openflights.org/data.html#airport`
[2] `https://en.wikipedia.org/wiki/Mercator_projection`
[3] `https://en.wikipedia.org/wiki/Web_Mercator_projection`

Part U: Model implementation

In this U-part of the exam (`util.cpp`), we implement the essential parts of the mathematical model driving the main application: The web mercator equation which projects spherical coordinates to a 2D plane and the Havesine equation which determines the great-circle distance between two points on a sphere.

Note that you can solve subsequent assignments without solving the assignments in this U-part so if you get stuck, we advise you to move on.

U1: The Web Mercator projection

Implement the Web Mercator projection as given below and return the resulting x and y coordinates as a `Point`.

The Web Mercator projection is given as two formulas which maps longtitude and latitude to discrete x and y coordinates respectively.

$$
\begin{aligned}
x &= \left\lfloor \frac{w}{2\pi}(\lambda + \pi) \right\rfloor \text{ pixels} \\
y &= \left\lfloor \frac{h}{2\pi} 2^{0.2} \left(\pi - \ln \left[\tan \left(\frac{\pi}{4} + \frac{\varphi}{2} \right) \right] \right) \right\rfloor \text{ pixels}
\end{aligned} \tag{1}
$$

where w and h are the width and height of the area the map should be projected onto (in pixels) and φ and λ are the latitude and longitude, respectively, of the point to project in radians. x and y are the pixel coordinates of the points to be plotted.

For your convenience, we provide the `deg_to_rad` function for converting degrees to radians. This is needed since the formulas expect coordinates in radians rather than degrees. Use the constant `M_PI` to get the value of π. Additionally, all of the mathematical functions used in the formula are available from the C++ standard library as `floor` ($\lfloor x \rfloor$), `log` ($\ln x$), `pow` (b^n) and `tan`.

U2: Distance calculation

The Havesine formula is given as [4]

$$
\begin{aligned}
a &= \sin^2 \left(\frac{\Delta\varphi}{2} \right) + \cos \varphi_1 \cdot \cos \varphi_2 \cdot \sin^2 \left(\frac{\Delta\lambda}{2} \right) \\
c &= 2 \cdot \text{atan2} \left(\sqrt{a}, \sqrt{1-a} \right) \\
d &= R \cdot c
\end{aligned} \tag{2}
$$

where φ and λ are latitude and longtitude in radians and R is the mean radius of earth (6371 km). d is the distance between two coordinates in km.

In the `distance` function of `util.cpp`, implement the havesine formula for calculating the great-circle distance between two geographic coordinates.

For your convenience, we provide the function `deg_to_rad` which converts degrees to radians. This is needed since the formulas expect coordinates in radians rather than degrees. Additionally, all of the mathematical functions used in the formula are available from the C++ standard library as `sin`, `cos`, `sqrt` and `atan2`.

Note that

$\sin^2 x = \sin x \cdot \sin x$

Part A: Data representation

This group of assignments extends the airport data manipulation and representation parts of the program. All of the following assignments are to be implemented in the file `airports.cpp`

[4] `http://www.movable-type.co.uk/scripts/latlong.html`

A1: Coordinate mapping

Implement the function `get_map_coord` such that it uses the static `to_map_coord` function declared in the `Util` class to return a `Point` representing the projected coordinates for the current airport instance. Remember that the relevant parameters can be found as the `map_w`, `map_h`, `latitude` and `longtitude` class member variables.

A2: Airport highlighting

Implement the function `highlight` which highlights this airport on the map by

- Setting its fill color to visible red
- Setting its radius to 7

Recall that this class (`Airport`) extends the `Circle` class from `Graph_lib` so you have access to all the functions from that class. Look at the corresponding `restore` function, implemented below, for inspiration.

A3: Airport search

Implement the function `search` which returns a vector of pointers to `Airport` instances (vector<shared_ptr<Airport») whose name contains the string passed as the `needle` parameter. Remember to make the search case insensitive. For your convenience, we provide the function `string_to_lower` for converting a string to lower case.

Return an empty vector if no matching airport is found. Remember that pointers to all the `Airport` instances are held in the class-member vector `airport_list`.

For example, if the `needle` parameter is "Tron" or "tron" the only matching airport is Trondheim Værnes. The returned vector should therefore only contain a single Airport object instance pointer. You can use the existing declaration and return of the `result` vector.

Part E: The airport explorer interface

The following E assignments implements the functionality of the graphical user interface. All of the assignments are implemented in the file `explorer.cpp`.

E1: Airport lookup

Implement the function `lookup_airport` which looks up an airport by its three-letter code passed through the `code` parameter. Use the map accessed using `airports->airport_map_by_code` to do this. This is a normal C++ `map` which can be accessed as you are used to, except that key lookups are case insensitive. If the airport isn't found you should

- call the function `alert` to show a meaningful error message
- throw the exception `AirportNotFoundException` defined in `explorer.h`

E2: Airport code validation

Implement the function `validate_code` which checks if the airport code passed as a string through the `code` parameter is valid. A valid airport code must

- be exactly three characters long
- contain only letters between a and z (inclusive, case insensitive)

For example, BRU and icn are valid codes, while br, br0 or ENGM are invalid.

If the airport code is found to be invalid

- use the function `alert` to display a meaningful error message
- throw the `InvalidAirportCode` exception.

E3: Airport highlighting

Implement the handler for the airport highlighting action of the graphical user interface. This action highlights the airport whose code is entered into the text field named `in_airport` in the this class. The function should

- get the airport code entered in the `in_airport` text field
- Call the function `validate_code` to ensure that the code is valid
- look up the name of the airport to get a pointer to the corresponding `Airport` object
- call the `highlight` function of the returned `Airport` object pointer to highlight the airport
- Add the `Airport` pointer to the class member vector `highlighted` so that it can be restored to normal later

E4: Airport searching

Using the `search` function from the `Airport` class (Assignment A3), implement the handler code for the search action of the graphical user interface. The result of the search should be displayed in the output box `search_results`.

For example, a search for the string "oslo" yields two results, OSL (Gardemoen) and FBU (Fornebu)[5]. An example of a suitable output to put in the `search_results` box is

```
The search for oslo returned
OSL Oslo Lufthavn
FBU Oslo, Fornebu Airport
```

To generate the printout above, remember that you can access the properties (such as code and name) of each `Airport` object pointer returned by the search by directly accessing its public class member variables. In this case, assuming that `a` is a pointer to an `Airport` object instance, `a->code` and `a->name` will return the code and name of the airport respectively. Refer to the declaration of the `Airport` class in `airports.h` for more details.

E5: Distance calculation

Implement the handling code for the distance calculation action which calculates the distance between two airports and prints the result in the Distance calculation text box (declared as `distance_results`). The text fields `in_from_airport` and `in_dest_airport` contains the codes of the two airports.

Your function should use the distance function from the `Util` class (assignment U2). Get the latitude and longtitude parameters from the Airport object pointers returned from the `lookup_airport` function.

For example, querying the distance between OSL (Gardemoen) and SFO (San Francisco) could print the following in the distance calculation box:

```
The distance from
OSL to SFO is
8344.854 km
```

[5]Which, for some reason is, still in the database

E6: Trip files

In this assignment you will implement a part of a feature that plots and calculates the aggregated length of a multi-leg journey between several airports.

A trip is defined in a file containing a single line with a sequence of airport codes separated by spaces. For example:

```
TRD OSL EWR
```

specifies a trip from Trondheim (TRD) to Newark (EWR) via Oslo (OSL).

Specifically, we ask you to implement the code for opening and parsing such files into a vector of strings containing the airport codes. The distance calculation and plotting itself is already implemented in the function `calculate_trip` which takes a vector of airport codes as its parameter. The path of the trip file is entered into the `in_trip_file` text box. You can use the preexisting declaration of the `codes`-vector and the call to `calculate_trip` in your implementation.

Several sample trip files are provided in the `data` folder, and they are named `tripX.txt`, where X is a number.

The function should call the `alert` function to display a suitable error message and throw the `FileReadError` exception if something goes wrong. In particular this should happen when

- Opening the file fails
- The file contains less that two airport codes