



**Politecnico  
di Torino**

Politecnico di Torino  
Corso di Laurea Magistrale in Ingegneria Aerospaziale

# **Development of a graphical interface for aeronautical application**

**Relatori:**  
Angelo Lerro  
Piero Gili

**Candidato:**  
Matteo Mangoni

Sessione di Laurea Luglio 2024  
Anno accademico 2023/2024

Development of a graphical interface for aeronautical application .....	1
1. INTRODUCTION .....	4
2. REFERENCES .....	5
3. SYSTEM OVERVIEW .....	6
3.1 Hardware .....	6
3.1.1 Odroid N2.....	6
3.1.2 Xenarc 10' Display - 1029 GNH.....	6
3.2 Software .....	7
3.2.1 Development Environment .....	7
3.2.2 Target Environment.....	7
4. AVIONIC DISPLAY DESIGN AND FUNCTIONALITIES.....	8
4.1 PRIMARY FLIGHT DISPLAY (PFD).....	8
4.1.1 Attitude Indicator .....	9
4.1.1.1 Waterline .....	9
4.1.1.2 Background (Sky, Ground and Horizon) .....	10
4.1.1.3 Pitch Ladder .....	10
4.1.1.4 Roll Indicator .....	11
4.1.1.5 Skid/Slip Indicator.....	12
4.1.2 Compass.....	13
4.1.3 Airspeed + AoA Indicator.....	14
4.1.4 Altitude Indicator .....	16
4.1.4.1 Altitude tape .....	17
4.1.4.2 Vertical Speed Indicator (VSI).....	18
4.1.5 Digital Map .....	19
4.1.6 Other Readings .....	20
4.2 MAP.....	21
4.3 DATA.....	22
4.4 USER CUSTOMIZATION – settings.ini.....	23
5. CODE IMPLEMENTATION.....	24
5.1 openFrameworks - OVERVIEW AND STRUCTURE.....	24
5.1.1 Folder Structure .....	25
5.1.2 Code structure and main functions .....	26
5.1.3 Graphic functions / classes.....	28
5.1.4 Add-ons.....	31
5.2 Scenes and Indicator Classes.....	33
5.2.1 PfdScene .....	34
5.2.1.1 Attitude Indicator.....	35
5.2.1.2 Compass .....	39
5.2.1.3 Airspeed Indicator.....	41
5.2.1.4 Altitude Indicator.....	43
5.2.1.5 Map .....	45
5.2.1.6 Other Readings .....	46
5.2.2 MapScene .....	47
5.2.3 DataScene .....	48

5.3 INPUT DATA INTERFACE.....	49
5.4 GLOBAL VARIABLES.....	51
5.5 MAIN LOOP: ofApp and other functionalities .....	52
5.5 PORTING TO LINUX (ODROID, UBUNTU MATE) .....	53
6.CONCLUSION.....	54

# 1. INTRODUCTION

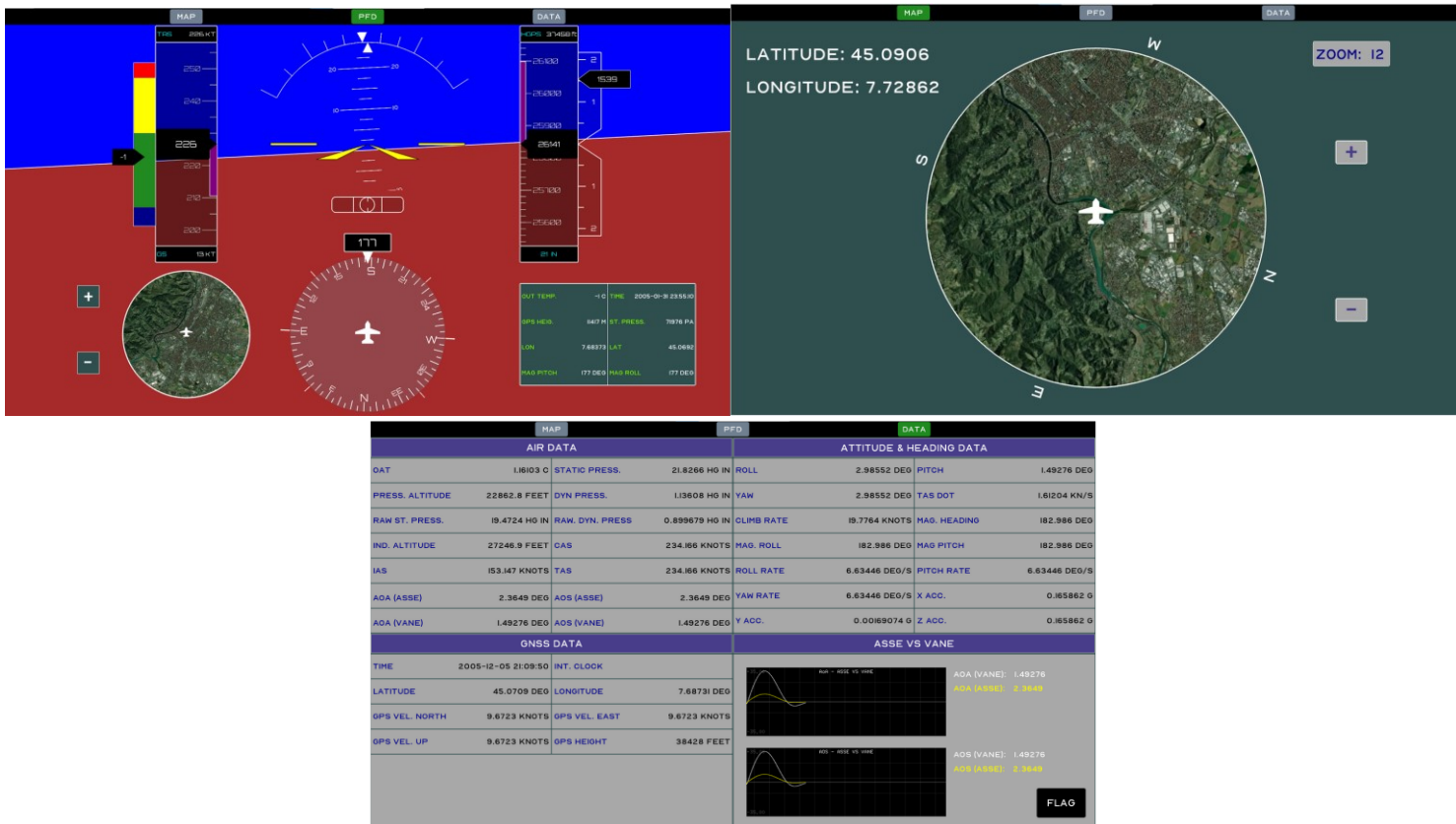


Figure 1.1 SAIFE Avionic Display tabs (PFD, MAP, DATA)

The objective of this project was the development of the software for an avionic display to be used as the Human-Machine Interface for the SAIFE demonstrator. From RD 1:

*“[...] the project SAIFE [28]—Synthetic Air Data and Inertial Reference System—where a demonstrator of the ASSE technology is designed and manufactured to verify the Technology Readiness Level (TRL) 5. The technological demonstrator is based on “all-in-one” air data and inertial systems (commonly known as ADAHRS) able to provide multiple information to pilots or to automatic control systems, partially based on synthetic sensors that are used for flow angle estimation. The proposed approach for flow angle estimation does not require dedicated physical sensors but at the same time guarantees, under recognizable circumstances, the same reliability of flow angle vanes (or probes) in order to optimize the efficiency of on board avionics for both modern and future aircraft.”*

The software was developed in C++/OpenGL, using the openFrameworks set of libraries to take advantage of simplified and ready-to-use functions and add-ons to streamline the process while maintaining the efficiency and optimization of C++ and OpenGL. More details on the framework and its implementation in the context of this project are given in the following sections.

The target hardware is composed of a Odroid N2 as the main processing unit, with a 10 inch touch screen display (Xenarc 1029 GNH) as the visual output device. The touchscreen functionality of the display is utilized by the software to give a certain level of intractability to the user (e.g. switching between different tabs, adjusting the map levels...).

## 2. REFERENCES

RD 1	Verification in Relevant Environment of a Physics-Based Synthetic Sensor for Flow Angle Estimation
RD 2	SAIFE_IO ( SW ICD)

## 3. SYSTEM OVERVIEW

### 3.1 Hardware

Hardware wise, the application is developed to be implemented and executed on a Odroid N2, while the graphical output is provided through the Xenarc 1029 GNH 10' touch screen display.

#### 3.1.1 Odroid N2

The Odroid N2 is a new generation single board computer. The main CPU is based on big.Little architecture which integrates a quad-core ARM Cortex-A73 CPU cluster and a dual core Cortex-A53 cluster with a new generation Mali-G52 GPU. Thanks to the modern 12nm silicon technology, the A73 cores run at 2.2Ghz without thermal throttling using the stock metal-housing heatsink allowing a robust and quiet computer.



Figure 3.1 Odroid N2

#### 3.1.2 Xenarc 10' Display - 1029 GNH

The 1029 GNH offers all of the innovation and inputs as our 1020 series models along with a **capacitive touchscreen panel**. Capacitive sensing provides a more responsive touch interface than resistive touch. It achieves this by using the capacitance of human skin to detect the proximity or position of the input. Proximity sensing allows the touch interaction to work even if protective glass or coating is used over the **10.1" industrial display**. The touch panel is optically bonded to the LCD panel, eliminating the air gap between the two, increasing clarity, contrast, and durability. Further, because the unit is water tight and air tight, ambient humidity will not affect the clarity or viewability because of the optical bonding.



Figure 3.2 Xenarc 10' Display

## 3.2 Software

The chosen programming language is C++, making use of the OpenGL library for graphical applications. C++ and OpenGL provide a good compromise between performance and availability of high level tools and frameworks for development.

*openFrameworks* is the chosen framework for developing the avionic display, as it's open source and provides high level wrapper library and functions around pure OpenGL, simplifying the development, structure and readability of the code without sacrificing performance. Several community add-ons for a variety of functionalities are also available (e.g. ready-to-use stencil masks).

A more detailed overview of how OF is structured and utilized in the context of this project can be found in section 5.

### 3.2.1 Development Environment

The code was developed and tested mainly in Windows 10, using the Visual Studio IDE. *openFrameworks* provides a framework folder structure that's fully compatible with Visual Studio, making it possible to take advantage of functionalities such as code completion, resources profiling and source compiling + debugging directly from the IDE.

### 3.2.2 Target Environment

The target environment for running the application is the Linux distribution *Ubuntu Mate* installed on the Odroid N2.

Note that the application compiled in the development environment is not directly compatible with the Linux Operating System, but it had to be compiled directly from the source files. The code is compiled through the use of *makefile(s)*, which are included in the Linux ARM x86 *openFrameworks* distribution. More details on the "porting" of the application from Windows 10 to *Ubuntu Mate* can be found in section 5.5.

## 4. AVIONIC DISPLAY DESIGN AND FUNCTIONALITIES

This section will give an overview of the design and functionalities of the display. There are 3 selectable tabs: PFD, MAP and DATA, described in detail in the following subsections.

The type of indicators and displayed data are based on the data available as input from the mission computer (see RD 2, SW ICD), which is processed as needed before it's rendered and shown on screen through the different indicators or through raw text. More details on the input interfaces can be found in section 5.3 (Data Interfaces).

On the top of the screen and at all time, 3 toggle buttons are present, one for each selectable tab. The currently selected tab button is highlighted in green.

A set of parameters and options can be customized via a .ini file, located in the bin/data directory. See section 4.4 for more details.

### 4.1 PRIMARY FLIGHT DISPLAY (PFD)

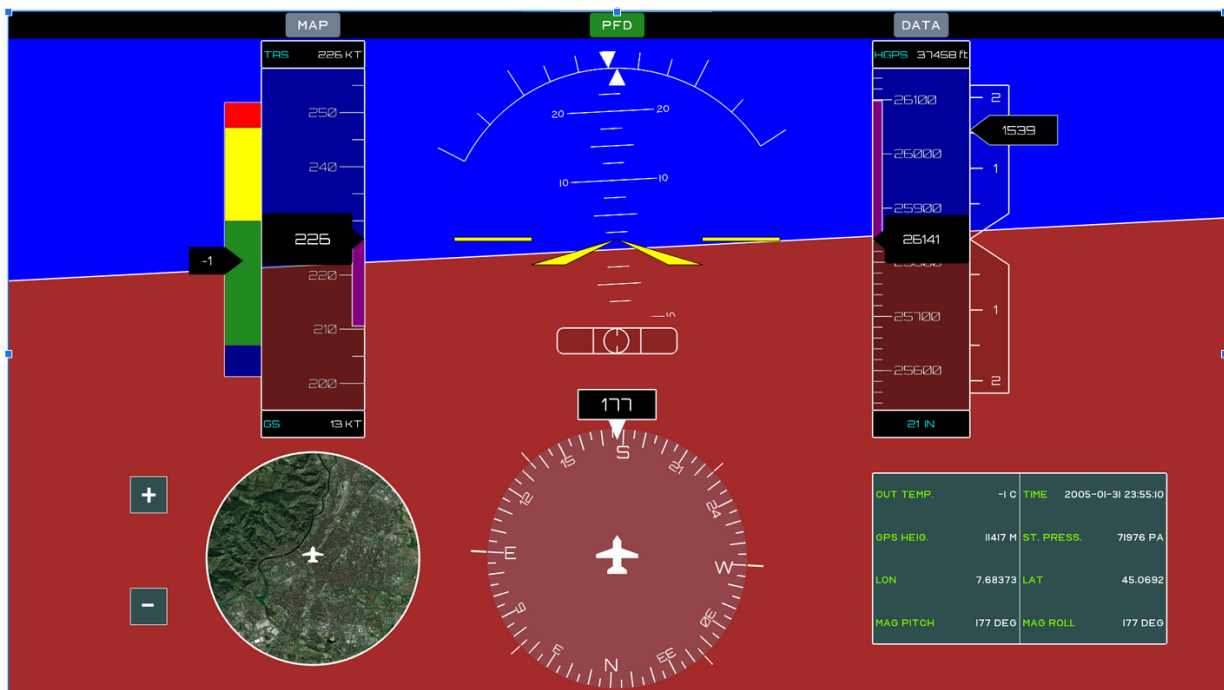


Figure 4.1 PFD

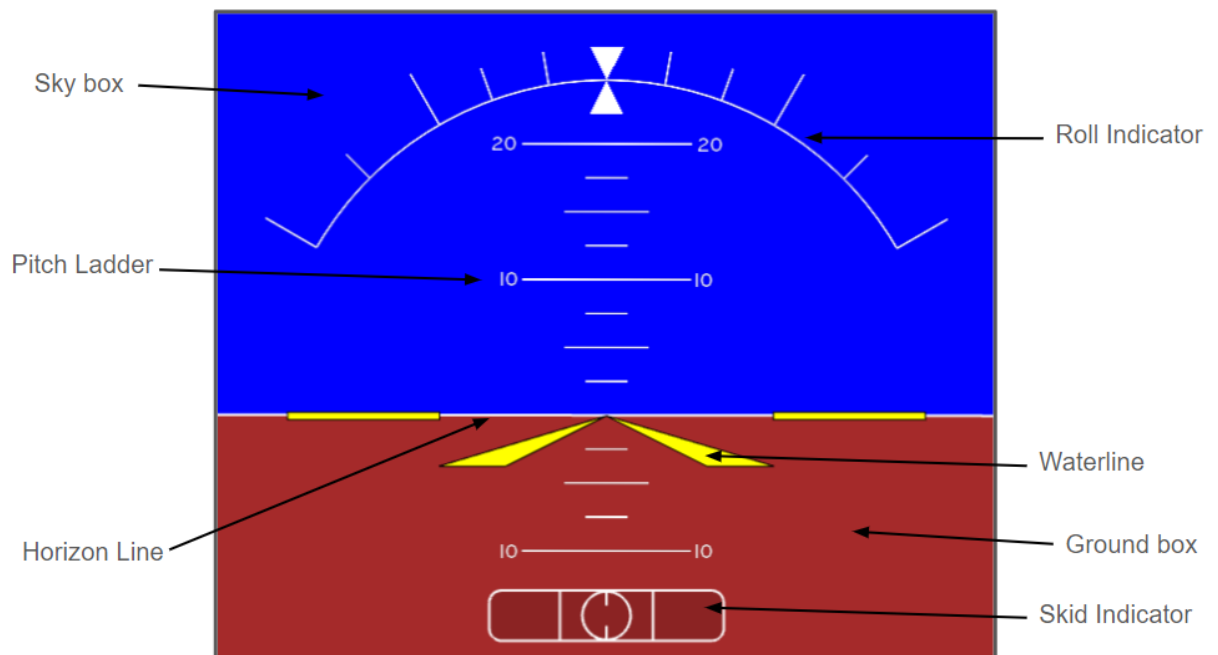
Figure 4.1 shows the full display with its main indicators and their layout, which are in line with a typical Primary Flight Display used in general aviation.

One of the objectives of the PFD is to implement an intuitive interface that should be easy to read and provide awareness of the aircraft general attitude and flight configuration "at a glance". This is achieved via the choice of position, color and size of each graphical element favoring adequate size and contrast between each distinct indicator.



The following subchapters will focus on each group of indicators, their readings and functionalities.

#### 4.1.1 Attitude Indicator



**Figure 4.2: Attitude Indicator**

The Attitude Indicator provides visual information on the attitude of the aircraft with respect to the horizon, by displaying the rotations around the Y axis (pitch angle) and the X Axis (roll angle), as well as skid/slip situation.

##### 4.1.1.1 Waterline

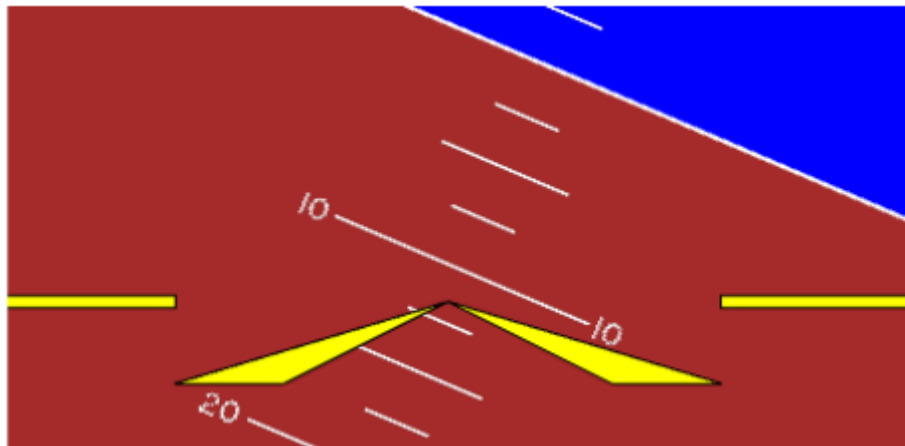


**Figure 4.3 Waterline**

The Waterline is composed of a couple of yellow triangles pointing in the direction of the nose and two yellow rectangles on each side representing the wings.

The Waterline symbol represents the orientation of the nose of the aircraft with respect to the horizon. During flight, it remains fixed on the screen, while the background moves depending on pitch and roll angles as detailed in section 4.1.2.

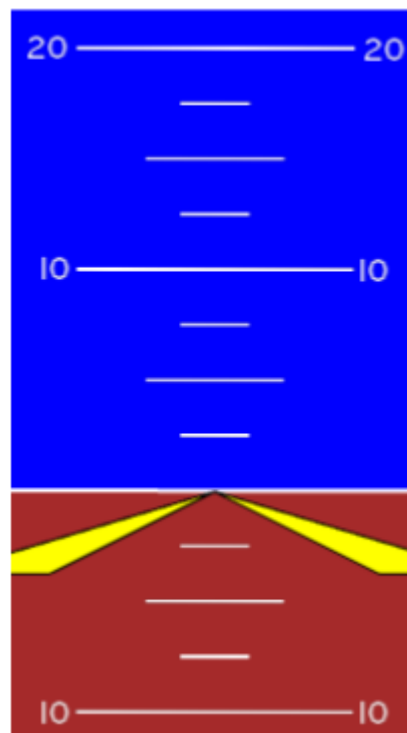
#### 4.1.1.2 Background (Sky, Ground and Horizon)



**Figure 4.4 Sky, Ground and Horizon**

The Ground and Sky boxes are monochromatic sections of the displays separated by the white horizon line in the center, and they fill the entire display at all times. All three of these objects move in the background both sliding vertically (Pitch angle) and rotating (Roll angle) around the fixed position of the Waterline.

#### 4.1.1.3 Pitch Ladder



**Figure 4.5 Pitch Ladder**

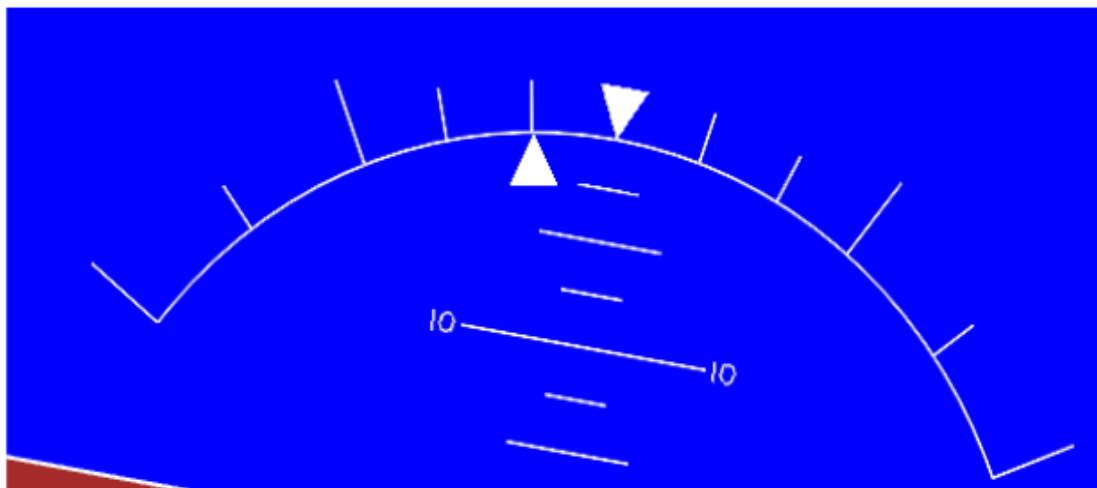
The Pitch ladder is a representation of the pitch angle of the nose of the aircraft. It moves alongside the background (section 4.1.2).

The scale ranges from -90 to 90 degrees, and it only shows in the portion of the screen between the Skid indicator and the Roll indicator. The sections of the ladder that fall outside this “window” are masked to avoid visual clutter.

The scale lines are of three types:

- **10s:** Widest lines, representing intervals of 10s (degrees). They are the only lines where the angle is indicated with a numerical text on both sides of the line.
- **5s:** Middle width, representing values of 5s at an interval of 10 degrees between one another (e.g. 5, 15, 25...).
- **2.5s:** Shorter lines, representing increments of 2.5s at an interval of 5 degrees between one another (e.g. 2.5, 7.5, 12.5...).

#### 4.1.1.4 Roll Indicator



**Figure 4.6 Roll Indicator**

The Roll Indicator provides a scale for the Roll angle of the aircraft. The Roll ticks (white lines perpendicular to the white arc) provide a range from -60 to + 60 at varying intervals as follows:

- 0 +/- 30 degrees: one short line every 10 degrees, a longer line at +/- 30.
- +/- 45 degrees: a short line.
- +/-60 degrees: a longer line.

The actual indicator symbol is the white triangle on the lower side of the arc: it remains fixed on the screen (same approach as the Waterline), and the rest of the Roll indicator rotates around the Waterline point.

If the Roll exceeds -60 or +60 degrees, the arc is expanded up to the position of the lower triangle.

The upper white triangle is fixed on the arc and represents the 0 degree roll orientation. This approach makes it intuitive for the pilot to “fly to” the triangle by aligning them to reach a 0 roll degree flight attitude.

Figure 4.6 shows an attitude of a -10 degrees roll angle.

#### 4.1.1.5 Skid/Slip Indicator



**Figure 4.7 Skid/Slip indicator**

The Skid/Slip indicator is located on the lower side of the Attitude indicator group, and it represents the slip or skid situation of the aircraft.

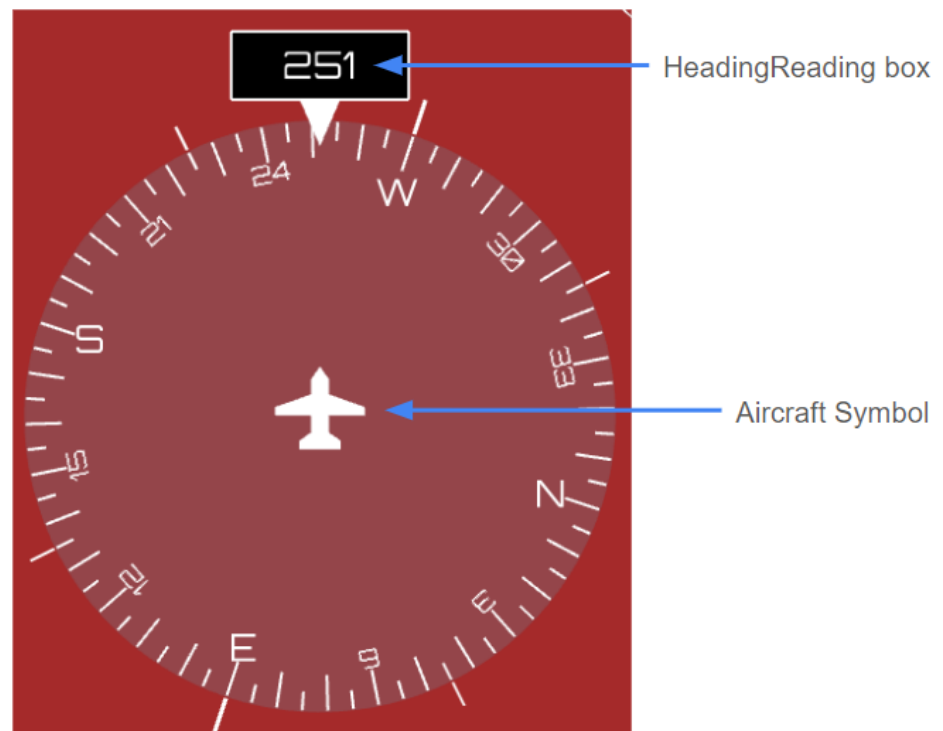
The actual indicator is the hollow white circle which moves inside the box depending on the lateral acceleration (y-axis) of the aircraft. The position of the ball can range from -9.8 to +9.8 m/s<sup>2</sup>, or -1g to 1g.

In general, a non-zero position of the ball means that the aircraft nose of the aircraft is not pointing into the flight path, and it's particularly crucial to help the pilot correct for any deviations in a turn. If the ball is centered on the indicator, it means that the turn is coordinated.

Depending on the direction of the turn, the lateral position of the ball represents either skid or slipping:

- **Left turn:** if the ball is moving right, the aircraft is skidding. If it's moving left, it's slipping.
- **Right turn:** if the ball is moving right, the aircraft is slipping. If it's moving left, it's skidding.

### 4.1.2 Compass



**Figure 4.8 Compass**

The compass is situated on the lower portion of the display, and provides information on the heading of the aircraft.

The white aircraft symbol in the center of the compass remains fixed while the compass rotates depending on the heading angle.

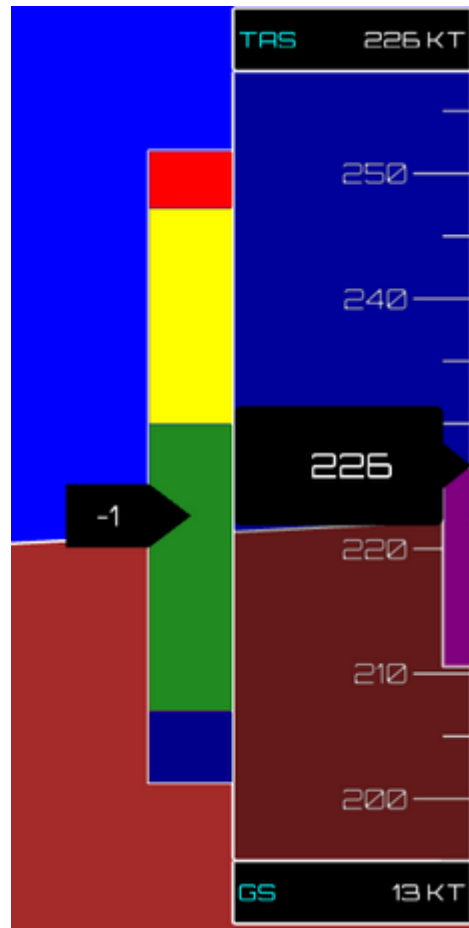
Above the compass, a reading box provides the heading angle with respect to the North.

The ticks inside the compass represent different intervals of degrees, as follows:

- **N,E,S,W:** Positioned at 0,90,180 and 270 degrees on the compass, they represent the North, East, South and West. The white ticks are long, and the text (N,E,S,W) always stays upright during the rotation.
- **10s:** Every 10 degree interval is marked with a white long ticks (same as N,E,S,W), and every 30 degrees a numeric text represents the angle in 10s. This means that the 30 degrees is 3, 60 degrees is 6 and so forth.
- **5s:** Starting from 5 degrees, at an interval of 10 degrees from each other, shorter white lines represent rotation of 5s degrees (5, 15, 25....).

The ticks outside the compass are positioned at 45 degree intervals: 0,45,90,135 and so on. Thicker lines correspond to N, E, S, W.

### 4.1.3 Airspeed + AoA Indicator



**Figure 4.9 Airspeed + AoA Indicator**

The airspeed and AoA indicator is positioned on the left side of the display, and it mainly provides information on the aircraft speed through the air and its trend over time, as well as the current angle of attack (AoA).

The IAS reading box is positioned at the center of the tape, while the scale translates vertically behind it depending on the current Indicated Air Speed. It provides an exact reading in knots.

At the top of the scale, a TAS reading box provides the raw value as text of the True Air Speed.

*Note: the default measurement unit is Knots, however this can be changed via the .ini file as described in section 5.*

The scale is composed of different ticks, as follows:

- **10s:** Wider white lines represent the IAS values at intervals of 10s, starting from 0. At the end of these ticks, a text representing its value is shown.
- **5s:** Shorter white lines mark the IAS values at intervals of 10s, starting from 5. (e.g. 5,15,25...)

The range of the displayable Indicated Airspeed on both the IAS reading box and the scale is 0 to 460 knots.

The TAS reading box is positioned at the top of the scale. It provides a reading of the True Air Speed, in a range from 0 to 460 knots.

The Ground Speed reading box is positioned at the bottom of the scale, and it provides a reading of the ground speed of the aircraft. It is calculated based on North + East GPS velocities.

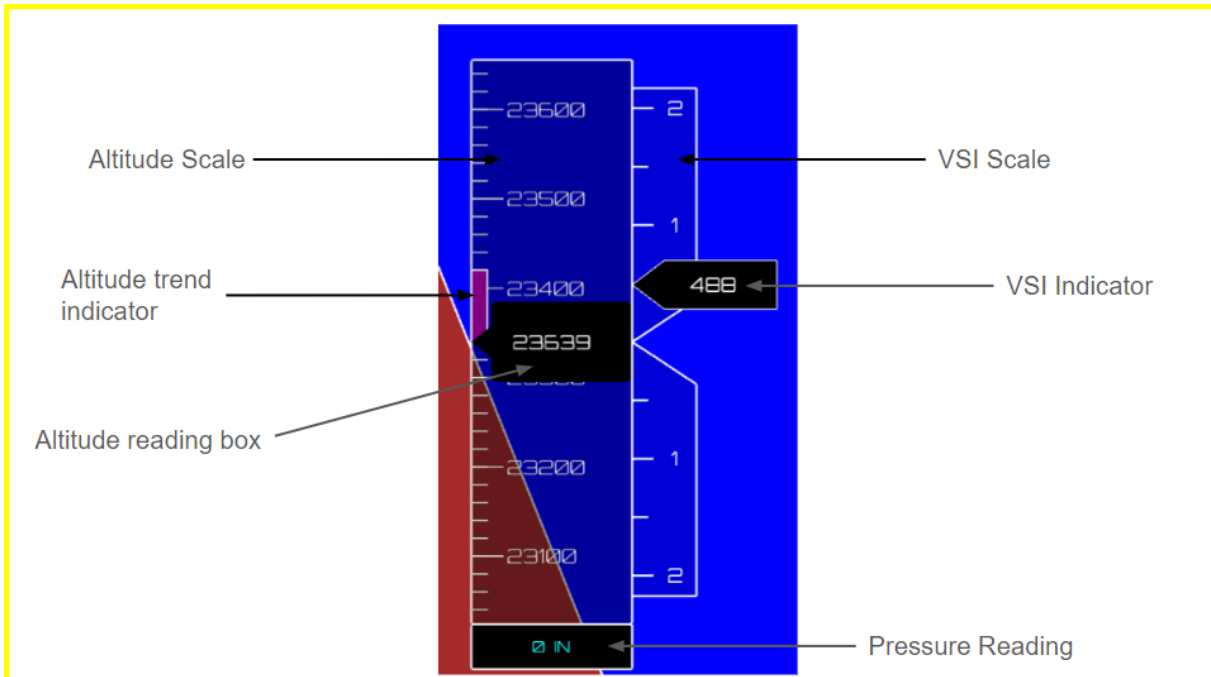
The Airspeed trend indicator is represented as a purple tape on the right edge of the scale. It expands either upwards or downwards depending on the sign of the TAS time derivative, reaching the calculated airspeed 10 seconds in the future on the tape.

The AoA indicator is composed of regions with different colors depending on the degree range, as follows:

- -20 to -15: blue
- -15 to +5: green
- +5 to +17: yellow
- +17 to +24: red

A AoA indicator (represented as a black arrow) provides the actual value of the AoA as it moves vertically along the scale.

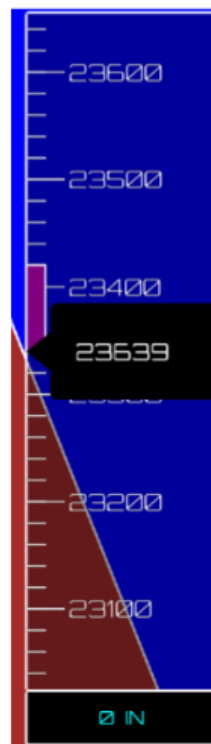
#### 4.1.4 Altitude Indicator



**Figure 4.10 Altitude Indicator**

The Altitude Indicator is positioned on the right side of the display, and provides information on the altitude and vertical velocity of the aircraft, as well as outside pressure.

##### 4.1.4.1 Altitude tape



**Figure 4.11 Altitude Tape**



The Altitude Tape is composed of the Altitude Scale, Altitude reading box, Altitude Trend Indicator and Outside Pressure Reading.

The Altitude Tape ranges from -980 to 46000 feet.

The scale ticks represent the values in increments of 20 feet (short lines) and 100 feet with text (wider lines).

At the center of the tape is positioned the Altitude Reading box, which is fixed and provides an exact reading of the current Indicated Pressure Altitude.

The Altitude Trend Indicator is represented as a purple box on the left edge of the scale. It expands either upwards or downwards depending on the sign of the Vertical Speed (climb rate), reaching the calculated altitude 10 seconds in the future on the tape.

The Outside Pressure Reading box is positioned at the bottom of the tape, and it provides the exact value of the indicated outside pressure in inches of mercury, ranging from about 6.5 to 32.

*Note: the default measurement system is feet for altitude and inches of mercury for outside pressure. The measurement system can be changed via a .ini file, as described in section 5.*

#### 4.1.4.2 Vertical Speed Indicator (VSI)

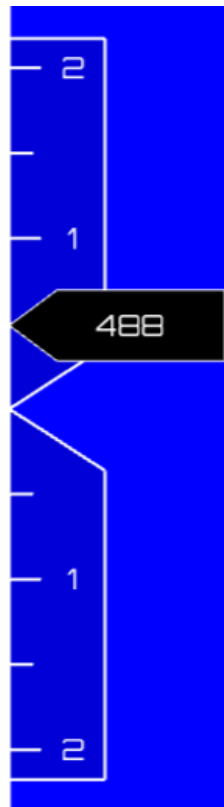


Figure 4.12 Vertical Speed Indicator (VSI)

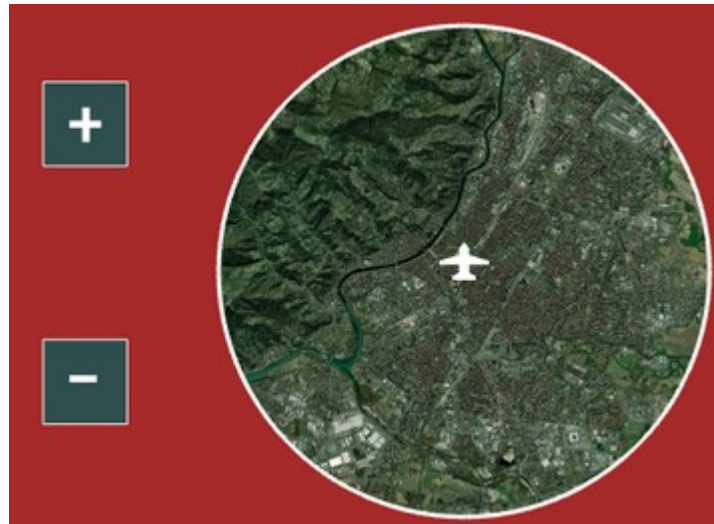
The Vertical Speed Indicator is composed of the Vertical Speed Scale and the Reading box / indicator, and it is positioned right of the Altitude Tape.

The scale ranges from -2000 to 2000 feet per minute, with ticks spaced at 500 increments. The text is shown on the side of the +/- 1000 and 2000 value, indicated as thousands.

The Reading box moves along the scale, which is fixed, indicating the current climb rate. It provides the exact value inside as text. While the scale only ranges up to +/- 2000 feet per minute, the reading inside the box can reach +/- 4000 feet per minute.

*Note: The measurement system can be changed via a .ini file, as described in section 5.*

#### 4.1.5 Digital Map



**Figure 4.13 Digital Map**

In the bottom left of the PFD, a small digital map shows a satellite view of the terrain at the current position of the aircraft. The Map is circular, and it rotates depending on the current magnetic heading.

The aircraft symbol is centered on the map and represents the current aircraft position. It is fixed, similar to the Compass implementation.

On the left side of the map are positioned a “+” button and a “-“ button, which can be touched on the screen to increase or decrease the zoom level.

This zoom level is shared with the MAP tab (see section 4.2), and the boundaries of the zoom levels are defined via the .ini file. The default min and max values are 11 and 17. The map is implemented with raster tiles, which are queried to a map tile provider (MapBox) the first time they are needed and are then saved locally for subsequent usage. For more details see section 5.2.2.

#### 4.1.6 Other Readings



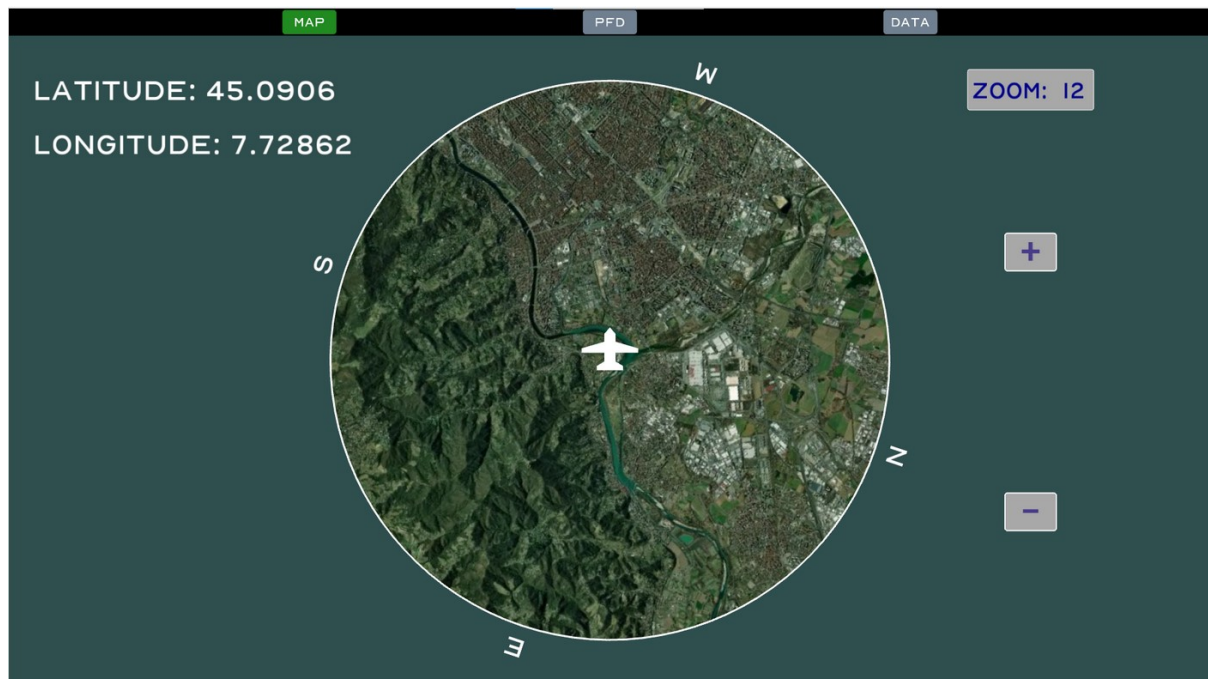
OUT TEMP.	-1 C	TIME	2005-01-31 23:55:10
GPS HEIG.	11417 M	ST. PRESS.	71976 PA
LON	7.68373	LAT	45.0692
MAG PITCH	177 DEG	MAG ROLL	177 DEG

**Figure 4.13 Other readings**

In the bottom right of the PFD, a set of useful parameters and values are shown as raw text to give the user a quick reference of the following data:

- Time (yyyy-mm-dd hh:mm:ss)
- Outside Temperature
- Latitude
- Longitude
- Static outside pressure
- GPS Height
- Magnetic Pitch
- Magnetic Roll

## 4.2 MAP



**Figure 4.14 MAP tab**

The MAP tab is fairly simple in its implementation: at the center of the screen the same circular map as present in the PFD is shown at full scale. In addition, the N, E, S and W symbols are drawn to give a general heading indication.

On the right of the screen the “+” and “-” symbols can be interacted with to increase or decrease the zoom level, which is shared with the map shown in the PFD. The zoom level is bounded between a minimum and maximum value which can be changed via .ini file.

The zoom level is also displayed as text on the top right.

On the top left of the screen the Latitude and Longitude current values are displayed as raw text.

## 4.3 DATA



Figure 4.15 DATA tab

In the DATA tab, the full set of input data is displayed as raw text, divided into 3 sections depending on the set of sensors from which they are derived: Air Data, AHRS, GPS.

A fourth quadrant on the bottom right is reserved for displaying 2 plots:

- Angle of Attack (vane) , Angle of Attack (ASSE) vs time
- Angle of Sideslip (vane), Angle of Sideslip (ASSE) vs time

These graphs are a useful representation of the difference between the data obtained through the instruments (vane) and the data obtain via the ASSE method (see RD 1).

On top of the graphical representation, on the right side of each graph the raw values of the angles are displayed with matching colors with respect to the plot.

These values, as well as the delta (as %) between them, are saved at a fixed interval inside an output text file (in the bin/data directory). The filename is a string representing the date (ASSE\_VS\_VANE\_YYYY\_MM\_DD\_HH\_MM\_SS) to avoid overwriting of the output file between different runs. The rate at which these values are saved can be customized via the .ini file.

On the bottom right of the quadrant, a FLAG button can be toggled. When the FLAG is toggled, the button is highlighted in red and a “flag” value in the output file is switched to a “1” until the FLAG button is toggled off. When the FLAG is toggled off, the “flag” value in the output file is “0”.

This functionality is useful for the user to easily keep track and highlight the set of data in a specific time interval, or during a particular maneuver and limit cases.

## 4.4 USER CUSTOMIZATION – settings.ini

```

*settings - Notepad
File Edit Format View Help
[general]
starting_tab = 1
frame_rate = 60
circle_res = 100

[attitude indicator]
center_x = 0.5
center_y = 0.333
roll_indicator_radius = 0.25

[nav]
center_x = 0.5
center_y = 0.8
compass_radius = 0.19

[airspeed indicator]
measurement_system = 1
center_x = 0.25
center_y = 0.333
width = 0.085
height = 0.5

[altitude indicator]
measurement_system = 1
center_x = 0.75
center_y = 0.333
width = 0.08
height = 0.5

[map]
preload_tiles = true
map_box token = .....|
min_zoom = 11
max_zoom = 17
starting_zoom = 12
min_lat = 45.0184
max_lat = 45.1217
min_lon = 7.57704
max_lon = 7.79399
center_x = 0.25
center_y = 0.8

[other_readings]
measurement_system = 1

[data]
measurement_system = 1
output_frequency = 10

```

In the bin/data folder, a .ini file “settings.ini” can be modified to customize a set of parameters and options without the need to modify the source code.

The .ini file is divided into subsections, and each parameters is explained via comments inside the .ini file. If the program can’t find a “settings.ini” file in the correct directory, a new one is created with default values. Another .ini file, “settings\_default.ini” can be found in the same directory: this file is not used by the program at runtime, serving just as a reference to the user of the default parameters with which a new “settings.ini” file would be created.

## 5. CODE IMPLEMENTATION

This section will provide an overview of code structure, starting from the general framework and going through each main class, which represent the different scenes and Indicator groups (as seen in section 4).

It will also provide details on the differences between the development environment (W10) and the target environment (Linux, Ubuntu), and instructions on how to “port” the application between the two.

Additional details on the code can be found as comments in the source code files.

### 5.1 openFrameworks - OVERVIEW AND STRUCTURE

From openFrameworks home page: <https://openframeworks.cc/about/>

*openFrameworks is designed to work as a general purpose glue, and wraps together several commonly used libraries, including:*

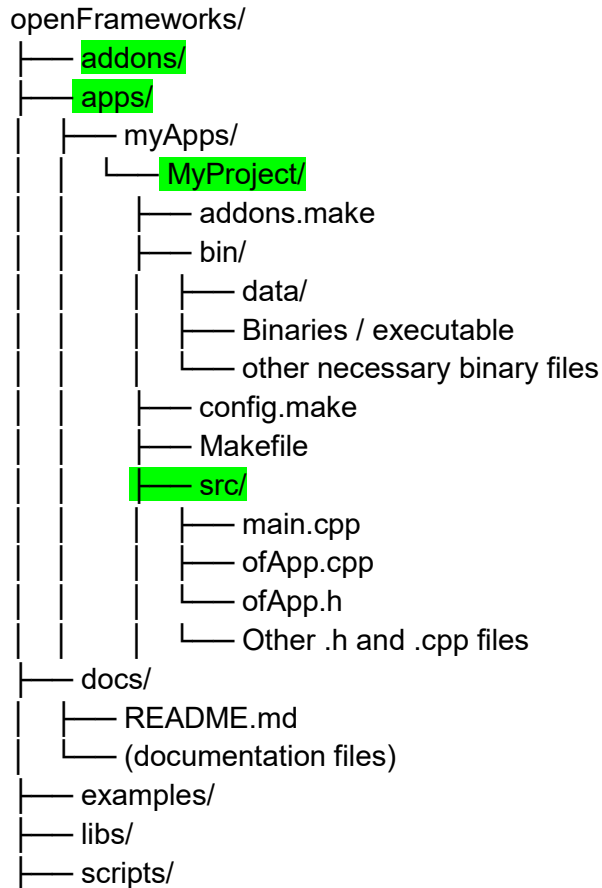
*OpenGL, GLEW, GLUT, libtess2 and cairo for graphics*  
*rtAudio, PortAudio, OpenAL and Kiss FFT or FMOD for audio input, output and analysis*  
*FreeType for fonts*  
*FreeImage for image saving and loading*  
*Quicktime, GStreamer and videoInput for video playback and grabbing*  
*Poco for a variety of utilities*  
*OpenCV for computer vision*  
*Assimp for 3D model loading*

*The code is written to be massively cross-compatible. Right now we support five operating systems (Windows, OSX, Linux, iOS, Android) and four IDEs (XCode, Code::Blocks, and Visual Studio and Eclipse). The API is designed to be minimal and easy to grasp.*

*openFrameworks is distributed under the MIT License. This gives everyone the freedom to use openFrameworks in any context: commercial or non-commercial, public or private, open or closed source. While many openFrameworks users give their work back to the community in a similarly free way, there is no obligation to contribute.*



### 5.1.1 Folder Structure

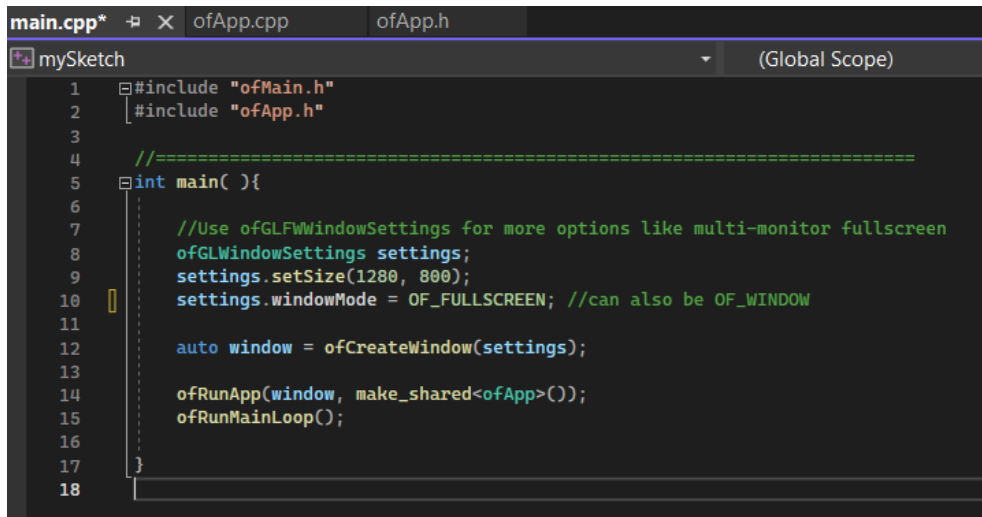


The folder structure above is the general structure with which openFrameworks is distributed. Highlighted in green are the main folders that are used in developing an openFrameworks app:

- **add-ons:** Any addon downloaded from the openFrameworks website will need to be moved in this folder. In order for the addon to be included and compiled with the project, the addons.make file will need to be updated by adding a line containing the name of the addon.
- **apps/MyProjects:** The source files (.h and .cpp) will need to be included in the /src folder, and the binary/executable file will be found in the /bin folder. Inside the /bin/data folder, supporting files will be included such as .ini, TrueTypeFont (TTF) for text and so on.

Maintaining this structure is important as the Makefile is implemented to reflect it.

## 5.1.2 Code structure and main functions



```

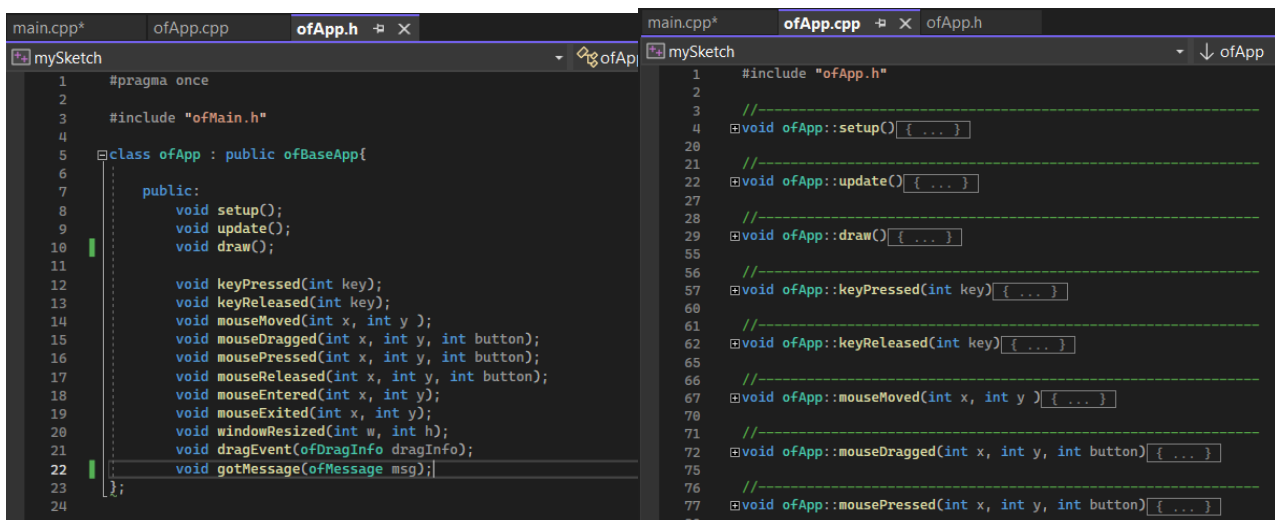
main.cpp*  ofApp.cpp  ofApp.h
mySketch  (Global Scope)
1  #include "ofMain.h"
2  #include "ofApp.h"
3
4  //=====
5  int main() {
6
7      //Use ofGLFWWindowSettings for more options like multi-monitor fullscreen
8      ofGLWindowSettings settings;
9      settings.setSize(1280, 800);
10     settings.windowMode = OF_FULLSCREEN; //can also be OF_WINDOW
11
12     auto window = ofCreateWindow(settings);
13
14     ofRunApp(window, make_shared<ofApp>());
15     ofRunMainLoop();
16
17 }
18

```

Figure 5.1 General main.cpp

The openFrameworks main loop is composed of a set of functions that are called at different times: at the start of the program (setup), once every cycle (update and draw) and when certain events trigger them (e.g. mousePressed, keyPressed...).

The main.cpp sets up the parameters and general settings of the program (figure 5.1), and then calls the main loop through the “ofApp” class (figure 5.2).



```

main.cpp*  ofApp.cpp  ofApp.h  mySketch
1  #pragma once
2
3  #include "ofMain.h"
4
5  class ofApp : public ofBaseApp{
6
7      public:
8          void setup();
9          void update();
10         void draw();
11
12         void keyPressed(int key);
13         void keyReleased(int key);
14         void mouseMoved(int x, int y );
15         void mouseDragged(int x, int y, int button);
16         void mousePressed(int x, int y, int button);
17         void mouseReleased(int x, int y, int button);
18         void mouseEntered(int x, int y);
19         void mouseExited(int x, int y);
20         void windowResized(int w, int h);
21         void dragEvent(ofDragInfo dragInfo);
22         void gotMessage(ofMessage msg);
23
24 };

```

```

main.cpp*  ofApp.cpp  ofApp.h  mySketch
1  #include "ofApp.h"
2
3  //=====
4  void ofApp::setup() { ... }
5
20 //=====
21
22 void ofApp::update() { ... }
23
27 //=====
28
29 void ofApp::draw() { ... }
30
55 //=====
56
57 void ofApp::keyPressed(int key) { ... }
58
60 //=====
61
62 void ofApp::keyReleased(int key) { ... }
63
65 //=====
66
67 void ofApp::mouseMoved(int x, int y ) { ... }
68
70 //=====
71
72 void ofApp::mouseDragged(int x, int y, int button) { ... }
73
75 //=====
76
77 void ofApp::mousePressed(int x, int y, int button) { ... }
78

```

Figure 5.2 General ofApp.h and .cpp

The ofApp class is the main “canvas” inside which the actual code is developed and implemented. The main functions are the following:

- **setup()**: The code inside the setup function will be executed only once at the start of the program. This is the recommended section in which to initialize most of the objects and define the main parameters. For example, a few graphical objects

(created through the `ofPath` function) are computationally intensive and therefore defining them once in the setup function is ideal.

- **update():** The update function of the `ofApp` class will be executed during each loop of the program runtime, just before the draw function. This section is recommended for the updating of inputs and other parameters which may change overtime.
- **draw():** Like update, the draw function is executed during each loop after the update function. The main distinction between the two is conceptual, in order to separate the update code from the drawing functions that actually render the graphical objects on screen.
- **inputs listeners:** Under the draw functions, several input listeners functions are defined depending on the type of input to be read and processed. These functions provide a direct way to execute certain blocks of codes for example when the mouse is pressed, dragged or released or when a certain key is pressed.

### 5.1.3 Graphic functions / classes

This section will give a brief overview of the main graphic functions utilized in the scope of this project.

For the full documentation, consult the following link:

<https://openframeworks.cc/documentation/graphics/ofGraphics/>

- **ofDraw functions:** most of the graphical shapes are defined and rendered through the ofDraw functions. All draw functions take as input the rendering position in the current transformation matrix, and the main parameters to define the shape. Example: `ofDrawRectangle(0,0,100,100)` will draw a rectangle 100 pixels wide, 100 pixels high, with the top left corner on the top left of the screen.

Note: the default reference system has its origin on the top left pixel of the screen, with the z-axis going through the screen as shown in Figure 5.3

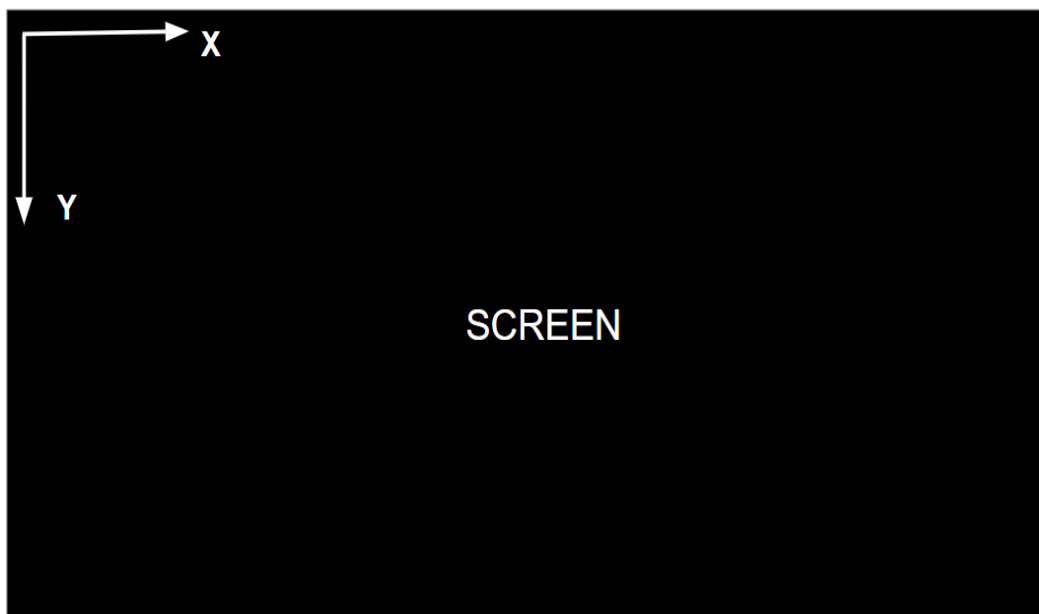


Figure 5.3 Screen reference system

- **ofTranslate and ofRotate:** These functions provide a straightforward way for moving the reference system by translating it and rotating it around vectors, making it possible to dynamically change the position on the screen of certain graphical objects based on certain parameters such as input data.

*Example:*

```
ofTranslate(200,200,0);
ofDrawRectangle (0,0,100,100);
```

The ofTranslate function translates the reference system by 200 pixels right and 200 pixels down, and draws a rectangle in the origin (0,0) of the current transformation matrix.

This means that the rectangle will be drawn with its upper left corner at the screen coordinate (200,200).

*Note: this is equivalent to calling ofDrawRectangle(200,200,100,100).*

- **ofPush and ofPop:** These functions are called to isolate certain transformations. The most relevant are ofPushMatrix and ofPushStyle and their ofPop counterparts. ofPushMatrix() saves the current transformation matrix, and the first call of ofPopMatrix() restores it. ofPushStyle() does the same thing, but for style-related parameters (such as color).

*Example:*

```
ofPushMatrix();
ofTranslate(200,200,0);

ofDrawRectangle (0,0,100,100);

ofPopMatrix();
```

*[other drawing code]*

The code example above saves the current transformation matrix (for example, the default one as per Figure 5.3). After that, the reference system is moved 200 pixels right and down and a rectangle is drawn with its upper left corner in the origin of the current transformation matrix (0,0).

The ofPopMatrix() function then restores the previously saved transformation matrix, making the drawing code which comes after unaffected by the previous transformations.

This approach is very useful in making sure that a certain set of graphical objects “move” together, isolating them from other elements of the scene.

- **ofColor class:** The ofColor class is a convenient tool to manage colors in the scope of this project. It can be used to define and store a specific color by using RGB and Alpha values, as well as hex values. It also provides a ready-to-use list of colors via keywords.

*Example:*

```
ofSetColor(ofColor::yellow);  
ofDrawCircle(400,400,100);
```

The code above sets the current color via the ofSetColor() function, to which the ofColor::yellow object is passed as argument. Then, a yellow circle of radius of 100 pixels is rendered with its center at (400,400).

- **ofGetWidth() and ofGetHeight():** These functions return the pixel width and height size of the current window. They are particularly useful to make the application able to dynamically adapt and fit to different screen and window sizes.

*Example:*

```
ofDrawCircle(ofGetWidth() / 2, ofGetHeight() / 2, 100);
```

This line of code renders a circle centered at the exact center of the window, regardless of its dimension or position on the screen.

### 5.1.4 Add-ons

On top of the included libraries in the base distribution of openFrameworks, the following addons are installed to take advantage of specific implementations and functionalities useful in the scope of this project.

- ofxLayerMask:** This addon provides direct functions and methods to implement masking effects of certain graphical objects. It makes use of FBOs (Frame Buffer Objects) and shaders to define a shape or region in the screen (mask) inside which to “clip” the graphical elements (layers).  
 To use the addon, a `ofxLayerMask` object has to be initialized (in the setup function), and then the `.beginMask()`, `endMask()`, `beginLayer()` and `endLayer()` functions are called to wrap around the drawing code of the mask shape and graphical objects to be masked.

*Example:*

```

ofApp::setup() {

  ofxLayerMask mask;
  mask.setup(ofGetWidth(), ofGetHeight());
  mask.newLayer();

}

ofApp::draw() {

  mask.beginMask();
  ofDrawCircle(400,400, 200);
  mask.endMask();

  mask.beginLayer();
  ofDrawRectangle(400,400,300,300);
  mask.endLayer();

  mask.draw(0,0);
}

```

The `ofLayerMask` object is initialized in the setup function, and then in the draw function the mask and layer are defined separately. The mask in this case is a circle of radius 200 pixels centered at (400,400), while the object to be rendered on the screen is a square of 300 pixels per side, drawn at (400,400).

The actual rendering happens through the `mask.draw(0,0)` function, which also takes a position (0,0) as argument. By changing the position of the `mask.draw()` function, the rendering can be moved as a whole on the screen.

- **ofxIniSettings:** This addon provides functions tailored to the openFrameworks environment useful in working with .ini files. In the context of this project, a set of parameters (positions, size, measurement system...) are read and initialized from a .ini file that can be customized by the user. The main aspect ratio of the indicators is maintained, but the position and size can be modified as well as the measurement system of each indicator.

Added details on the implementation of the .ini file reading and writing for this project can be found in the comments inside the source code.

- **ofxHistoryPlot:** This addon is used specifically to implement plots in the DATA tab of the display which shows the evolution over time of the Angle of Attack / Angle of sideslip from the sensor (vane) and the same angle calculated using ASSE.



## 5.2 Scenes and Indicator Classes

For each scene, a dedicated class is implemented in order to encapsulate the methods and variables specific to that scene. The PFD indicators also have dedicated classes. Instances of these classes (objects) are then initialized in the ofApp class.

Most of these classes follow a generic structure, similar to the main ofApp class structure. For example, each class will have a setup(), update() and draw() method, which in turn will be called in the appropriate functions of the scene objects and the main ofApp loop (more details in section 5.4).

The scene classes (PFD, MAP and DATA) are inherited from a baseScene class, which acts as a class template in order to allow the handling of the different scenes using a vector. The baseScene template has a setup, update and draw method , which are overridden by each scene class.

Every class and global variable is declared and implemented inside the ofApp.h file (see image...), while the ofApp class itself is implemented inside the ofApp.cpp file.

Each class will be described in the following sections, with notes on the most significant code implementations. For specific details on each function and method, see the source files which contain comments to the main parts of the code.

### 5.2.1 PfdScene

For each indicator group, the PfdScene class implements the respective indicator subclasses which are defined separately (more details on each PFD subclass in the following sections).

- **setup():** each subclass setup method is called, and the “+” and “-” buttons for the zoom of the minimap are defined and initialized.
- **loadIni():** each subclass loadIni() method is called, initializing the main parameters depending on the .ini file.
- **update():** each subclass update method is called.
- **draw():** each subclass draw method is called. In particular, the map is scaled down and drawn in the bottom left portion of the screen, with respect to the map subclass implementation which is more general as it's also used by the MAP scene. The “+” and “-” buttons are also drawn.
- **mousePressed():** this is a listener method which checks if the mouse (or touch in case of a touchscreen) is pressed inside either the “+” or “-” boxes, in which case it updates the mapZoom value (shared with the MAP scene).

### 5.2.1.1 Attitude Indicator

```

143 // Attitude Indicator class
144 class AttitudeIndicator
145 {
146
147     public:
148
149     variables declaration
150
195
196     AttitudeIndicator() { ... }
226
227     void loadIni() { ... }
238
239     void setupFont() { ... }
249
250     void setup() { ... }
262
263     void update(INPUTS INPUT_DATA) { ... }
279
280     void drawWaterLine() { ... }
310
311     void drawPitchScale() { ... }
347
348     void drawPitchScaleMask() { ... }
407
408     void drawMaskedPitchScale() { ... }
445
446     void drawRollIndicator() { ... }
527
528     void drawMaskedRollIndicator() { ... }
560
561     void drawSlipIndicator() { ... }
601
602     void drawAll() { ... }
646
647 };

```

Figure 5.4 Attitude Indicator Class

The Attitude Indicator class encapsulates all the variables and methods relevant to the Attitude Indicator group as described in section 4.1.1.

The approach in defining separate methods or not for a particular set of code depends on the complexity and length, with the objective to optimize readability and traceability of the transformations.

An overview of each class method can be found below:

- **variables declaration:** this is a block of code inside which every variable and object used inside the class is defined. More details can be found in the actual code.
- **AttitudeIndicator():** This is the constructor of the class, with the initializations to default values of the class variables. It is executed once when the object is created in the main ofApp class.

- **loadIni():** This method reads from the .ini file to save the values into the appropriate variables. It's called inside the main ofApp loadIni() function at the start of the program.
- **setupFont():** This method loads the TTF (True Type Font) files, defining the sizes of the small, medium and large font depending on the ratio between the default rollIndicatorRadius and the actual one (if it is changed via the .ini file). It's called inside the more general setup() function of this class.
- **setup():** This method initializes the mask objects (for the Roll indicator and the Pitch Ladder) and it calls the setupFont() method.
- **update():** This method updates the variables used in translating and rotating the relevant objects depending on the INPUT DATA. The input values are clamped to be inside the min and max values defined in the ICD. It also updates the reference dimensions of the Attitude Indicator based on the .ini parameters and the current window/screen size.
- **drawWaterline():** This method draws the Waterline (aircraft symbol) at the center of the current transformation matrix. It is called in the general drawAll() method.
- **drawPitchScale():** This method draws the unmasked Pitch Scale at the center of the current transformation matrix. It is called in the drawMaskedPitchScale() method;
- **drawPitchScaleMask():** This method draws the Pitch Scale Mask, at the center of the current transformation matrix. It is called in the drawMaskedPitchScale() method. The Mask is shaped as a portion of a circle with a radius slightly smaller than the Roll Indicator radius, and it's then cut off at the base just above the Slip indicator.
  
- **drawMaskedPitchScale():** This method draws the masked Pitch Scale, making use of the ofxLayerMask addon. It is called inside the general drawAll() method.

```

void drawMaskedPitchScale()
{
    maskPitchLadder.beginLayer();
    ofClear(0, 0, 0, 0);

    ofPushMatrix();
    ofTranslate(centerX, centerY);
    ofRotateZDeg(rotation);
    ofTranslate(0, -ladderTranslation);

    drawPitchScale();

    ofPopMatrix();

    maskPitchLadder.endLayer();

    maskPitchLadder.beginMask();
    ofClear(0, 0, 0, 0);

    ofPushMatrix();
    ofTranslate(centerX, centerY);

    drawPitchScaleMask();

    ofPopMatrix();

    maskPitchLadder.endMask();

    maskPitchLadder.draw(0, 0);
}

```

The pitch Ladder is drawn in the Layer section; it's translated to be positioned in the center of the Attitude Indicator group, and further rotated and translated depending on the INPUT DATA.

The Mask is drawn in the mask section, positioned in the center of the Attitude Indicator group.

The full result is then drawn via the .draw() method at the end, with (0,0) as position as this draws the full FBO which is initialized to be the size of the window.

**Figure 5.5 drawMaskedPitchScale()**

- **drawRollIndicator():** This method draws the Roll Indicator centered in the origin of the current transformation matrix. It is called in the drawMaskedRollIndicator() method.
- **drawMaskedRollIndicator():** This method draws the Roll Indicator masked to be only visible inside a box defined in the mask section of this method. It is the same approach used for the masked Pitch Scale, and the method is called in the general drawAll() method.
- **drawSlipIndicator():** This method draws the Slip Indicator positioned to be just below the Pitch Scale Mask lower edge. It is called in the general drawAll() method.
- **drawAll():** This method combines all the previous drawing methods by positioning them on the screen depending on the center of the Attitude Indicator group and input data (pitch and roll). It is called in the main draw() method inside the pfdScene class.

*Note: The order of the drawing methods is important, as it determines the "layer" in which the graphical object is rendered. The last draw method to be called will be rendered on top of everything else.*

```

void drawAll() {

    ofPushMatrix();

    //set reference system to the center of the AI (1/3 down from the top of the display)
    ofTranslate(centerX, centerY);

    ofPushMatrix();

    //apply rotation and translation depending on inputs
    ofRotateZDeg(rotation);
    ofTranslate(0, - ladderTranslation);

    //draw sky and ground box
    ofSetColor(_SKY_COLOR);
    ofDrawRectangle(-rectWidth / 2, -rectHeight, rectWidth, rectHeight);
    ofSetColor(_GROUND_COLOR);
    ofDrawRectangle(- rectWidth/2, 0, rectWidth, rectHeight);

    //draw horizon line (white)
    ofSetColor(255);
    ofSetLineWidth(2);
    ofDrawLine(-HorizonlineLenght / 2, 0, HorizonlineLenght / 2, 0);

    ofPopMatrix(); //restores the matrix before the translation and rotation

    drawSlipIndicator();

    ofPopMatrix(); //restores the original matrix (0,0) screen coordinates

    drawMaskedPitchScale();
    drawMaskedRollIndicator();

    ofPushMatrix();

    ofTranslate(centerX, centerY);
    drawWaterLine();

    ofPopMatrix();
}

```

Figure 5.6 AttitudeIndicator.drawAll()

## 5.2.1.2 Compass

```

642 // Compass class
643 class Compass {
644
645     public:
646
647     variables declarations
675
676     Compass() { ... }
698
699     void setupFont() { ... }
708
709     void loadIni() { ... }
719
720     void setup() { ... }
770
771     void update(INPUTS INPUT_DATA) { ... }
785
786     void drawCompass() { ... }
963
964     void drawPlaneIcon() { ... }
971
972     void drawAll() { ... }
988
989 };

```

Figure 5.7 Compass Class

The Compass class encapsulates all the variables and methods relevant to the Compass group as described in section 4.1.2.

An overview of each class method can be found below:

- **Compass():** This is the constructor of the class, with the initializations to default values of the class variables. It is executed once when the object is created in the main ofApp class.
- **setup():** This method initializes and defines the ofPath object used to draw the Plane icon at the center of the compass. It also calls the setupFont() method.
- **update():** This method updates the variables used in translating and rotating the relevant objects depending on the INPUT DATA. The input values are clamped to be inside the min and max values defined in the ICD. It also updates the reference dimensions of the Compass based on the .ini parameters and the current window/screen size.
- **drawCompass():** This method draws the compass at the origin of the current transformation matrix. It includes the compass ticks and text, as well as the heading

text box above the compass. It also takes care of rotating the compass depending on the INPUT DATA (heading).

It is called in the general drawAll() method.

- **drawPanelcon():** This method draws the panelcon (ofPath object defined in the setup()). It is called in the general drawAll() method.
- **drawAll():** This method combines all the previous drawing methods by positioning them on the screen depending on the center of the Compass group and input data (Heading). It is called in the main draw() method inside the pfdScene class.

```

973 void drawAll()
974 {
975     ofPushMatrix();
976     //set reference system to the center of the Compass group
977     ofTranslate(centerX, centerY);
978
979     drawCompass();
980     ofPushMatrix();
981
982     ofScale((radius/defaultCompassRadius)*1.2f, (radius / defaultCompassRadius) * 1.7, 1);
983     drawPlaneIcon();
984
985     ofPopMatrix();
986     ofPopMatrix();
987 }
988

```

Figure 5.8 Compass.drawAll()

### 5.2.1.3 Airspeed Indicator

```

// Airspeed Indicator class
class AirspeedIndicator
{
public:
+ variables delcaration
+ AirspeedIndicator() { ... }
+ void setupFont() { ... }
+ void loadIni() { ... }
+ void setup() { ... }
+ void update(const INPUTS& INPUT_DATA) { ... }
+ void drawBox() { ... }
+ void drawIndicatorMask() { ... }
+ void drawScale() { ... }
+ void drawMaskedScale() { ... }
+ void drawAoAIndicator() { ... }
+ void drawAll() { ... }
};

```



### Figure 5.9 Airspeed Indicator class

- **AirspeedIndicator():** This is the constructor of the class, with the initializations to default values of the class variables. It is executed once when the object is created in the main ofApp class.
- **setup():** This method initializes the mask objects and calls the setupFont() method.
- **update():** This method updates the variables used in translating and rotating the relevant objects depending on the INPUT DATA. The input values are clamped to be inside the min and max values defined in the ICD.
- **drawMaskedScale():** This method draws the masked Airspeed Indicator scale inside the box. It is called by the drawAll() method.
- **drawAoAIndicator():** This method draws the Angle of Attack Indicator to the left of the Airspeed Indicator.
- **drawAll():** This method combines all the previous drawing methods by positioning them on the screen depending on the center of the Airspeed Indicator group and input data.  
It is called in the main draw() method inside the pfdScene class.

```
void drawAll()
{
    ofPushMatrix();

    ofTranslate(centerX, centerY);

    drawBox();

    drawAoAIndicator();

    ofPushMatrix();
    ofTranslate(lineWidth*2, 0);
    drawMaskedScale();
    ofPopMatrix();

    drawIndicatorMask();

    ofPopMatrix();
}
```

Figure 5.10 AirspeedIndicator.drawAll()

#### 5.2.1.4 Altitude Indicator

```

// Altitude Indicator class
class AltitudeIndicator
{
public:
variable declration

AltitudeIndicator() { ... }

void loadIni() { ... }

void setupFont() { ... }

void setup() { ... }

void update(const INPUTS& INPUT_DATA) { ... }

void drawBox() { ... }

void drawScale() { ... }

void drawIndicator() { ... }

void drawMaskedScale() { ... }

void drawVertBox() { ... }

void drawAll() { ... }

```

**Figure 5.11 Altitude Indicator class**

- **AltitudeIndicator():** This is the constructor of the class, with the initializations to default values of the class variables. It is executed once when the object is created in the main of App class.
- **setup():** This method initializes the VSI indicator box (as ofPath objects), sets up the mask objects for the altitude tape and calls the setupFont() method.
- **update():** This method updates the variables used in translating and rotating the relevant objects depending on the INPUT DATA. The input values are clamped to be inside the min and max values defined in the ICD.
- **drawMaskedScale():** This method draws the masked Altitude Indicator scale inside the box. It is called by the drawAll() method.
- **drawVertBox():** This method draws the VSI box and indicator.
- **drawAll():** This method combines all the previous drawing methods by positioning them on the screen depending on the center of the AltitudeIndicator group and input data.  
It is called in the main draw() method inside the pfdScene class.

```
void drawAll()
{
    ofPushMatrix();

    ofTranslate(centerX, centerY);

    drawBox();
    drawMaskedScale();
    drawIndicator();
    drawVertBox();

    ofPopMatrix();
}
```

Figure 5.12 AltitudeIndicator.drawAll()

## 5.2.1.5 Map

```

class Map {
public:
variables declaration

void setup(bool isMapScene) { ... }

void update(INPUTS INPUT_DATA) { ... }

void drawMaskedMap() { ... }

void drawMaskedMapRect() { ... }

void drawAll() { ... }

void draw() { ... }

std::string getTileKey(int x, int y, int zoom) { ... }

bool isTileLoaded(int x, int y, int zoom) { ... }

void loadTile(int x, int y, int zoom) { ... }

void loadExistingMapTiles(int zoom) { ... }

glm::vec2 latLonToTile(float lat, float lon, int zoom) { ... }
};

```

Figure 5.13 Map class

This class is utilized by both the PfdScene and MapScene, and it's implemented to generally display a circular and rotating map depending on the current heading input value.

- **setup():** This method initializes the panelcon of Path object, sets up the mask objects for the map and initializes the font objects.
- **update():** This method updates the variables used in translating and rotating the relevant objects depending on the INPUT DATA. The input values are clamped to be inside the min and max values defined in the ICD.
- **latLonToTile():** This method takes as input lat,lon and zoom values, and transforms them to the x and y tile coordinates that contain the position passed as input.
- **loadTile():** This method constructs the url depending on the x,y and zoom level passed as input, and checks whether the tile is present in memory or if it needs to be loaded through a query to the map tile provider (MapBox). In either case, the tile is then stored in the unordered\_map TileCache.
- **isTileLoaded():** This method checks whether a tile is already loaded in the TileCache.

- **draw():** This method draws the map as a 4x4 grid of tiles that are either loaded from memory or queried to the tile provider (MapBox), depending on checks performed on the availability of the tile in the TileCache, local memory or neither.
- **drawMaskedMap():** This method draws the masked map. The mask is a circle of radius = tileSize.

### 5.2.1.6 Other Readings

```

// OtherReadings class
class OtherReadings
{
public:
variable declaration

OtherReadings() { ... }

void setup() { ... }

void setupFont() { ... }

void update(const INPUTS& INPUT_DATA) { ... }

void drawAll() { ... }
};

```

Figure 5.14 Other Readings class

- **setup():** This method sets up the font for this class
- **update():** This method updates the variables used in translating and rotating the relevant objects depending on the INPUT DATA. The input values are clamped to be inside the min and max values defined in the ICD. It also converts the GPS time to the format YYYY-MM-DD HH:MM:SS using the **convertGPSTimeToString()** function defined in the global scope (see code implementation for more details).
- **drawAll():** This method draws the actual box and text in the bottom right of the screen.

## 5.2.2 MapScene

The mapScene class “wraps” the Map class described above, displaying additional supporting elements and implementing the “+” and “-” buttons to provide interactivity to the user on the zoom level.

```
class MapScene : public BaseScene {
public:

    ofImage image;
    ofURLFileLoader loader;

    Map map;

    ofRectangle zoomPlusButton;
    ofRectangle zoomMinusButton;

    ofTrueTypeFont fontLarge;
    ofTrueTypeFont fontMedium;
    ofTrueTypeFont fontButton;

    void setup() override { ... }

    void update(const INPUTS& INPUT_DATA) override { ... }

    void draw() override { ... }

    void keyPressed(int key) override { ... }

    void mouseMoved(int x, int y) override { ... }

    void mouseDragged(int x, int y, int button) override { ... }

    void mousePressed(int x, int y, int button) override { ... }

    void mouseReleased(int x, int y, int button) override { ... }

};
```

Figure 5.15 MapScene class

- **setup():** The setup method calls the map.setup() method (from the Map class), and also initializes the font objects and the “+” and “-” buttons.
- **update():** This method calls the map.update() method (from the Map class) passing as input the INPUT\_DATA structure.
- **draw():** This method draws the MAP tab scene, by calling the map.drawMaskedMap() method at full scale in the center of the screen, as well as the “+” and “-” buttons and the LAT, LON and Zoom Level strings.
- **mousePressed():** This method is an input listener that checks whether the mouse was clicked inside either the “+” or “-” buttons and updates the zoomLevel singleton accordingly.

### 5.2.3 DataScene

```

class DataScene : public BaseScene {
public:
variables declaration
#pragma region main loop

void setup() override { ... }

void update(const INPUTS& INPUT_DATA) override { ... }

void draw() override { ... }

void keyPressed(int key) override { ... }

void mouseMoved(int x, int y) override { ... }

void mouseDragged(int x, int y, int button) override { ... }

void mousePressed(int x, int y, int button) override { ... }

void mouseReleased(int x, int y, int button) override { ... }

#pragma endregion

#pragma region custom methods

void drawADCPanel() { ... }

void drawGPSPanel() { ... }

void drawAHRSPanel() { ... }

void drawVaneASSEPanel() { ... }

void drawHeader(float x, float y, string label) { ... }

void drawTextBox(float x, float y, float width, float height, string label, string value) { ... }

#pragma endregion
};

```

Figure 5.16 DataScene class

This class handles the DATA tab, and on top of the usual setup, update and draw methods it makes use of other custom methods. The scene is divided into 4 panels, and each panel has a dedicated method that makes use of the general drawHeader and drawTextBox methods to draw each text field.

- **setup():** This method sets up the parameters from the .ini file, initializes the font objects and sets up the plot objects (for the ASSE vs VANE panel).
- **update():** This method updates all input data values which are stored in separate maps (label, value). These maps are then used in the drawing functions for each panel to loop through the list of inputs in drawing each entry in the panels.
- **draw():** This method calls the other draw methods for each panel positioning them accordingly on the screen. The FLAG button is also drawn in the bottom right of the ASSE vs VANE panel.

For more details on other methods for this class, refer to the source code and the comments inside.



## 5.3 INPUT DATA INTERFACE

The inputs to the avionic display in this project are managed through a structure that reflects the ICD table as per RD 2 and figure below:

### 1.2 Output

La Tabella 2 Tabella 1 descrive il set minimo di input da ricavare dalla suite di sensori del dimostratore SAIFE.

#	Description	Unità	SW acronimo	Min	Max
1	Outside Air Temperature	°C	ADC_OUT_OAT	-70	70
2	Static Pressure	Pa	ADC_OUT_STATICPRESS	22600	108000
3	Pressure Altitude	m	ADC_OUT_PRESSALT	-300	14000
4	Pressione dinamica	Pa	ADC_OUT_DYNPRESS	0	6400
5	Indicated Airspeed	m/s	ADC_OUT_IAS	0	237
6	Calibrated Airspeed	m/s	ADC_OUT_CAS	0	155
7	Indicated Altitude	m	ADC_OUT_INDPRESSALT	-300	14000
8	True Airspeed	m/s	ADC_OUT_TAS	0	237
9	Angle of Attack from ASSE	deg	ADC_OUT_AOA	-20	24
10	Angle of Sideslip from ASSE	deg	ADC_OUT_AOS	-20	20
11	Angolo d'incidenza di riferimento	deg	ADC_OUT_AOA_VANE	-90	90
12	Angolo di derapata di riferimento	deg	ADC_OUT_AOS_VANE	-90	90
13	Roll (angolo di sbandamento)	deg	ADC_OUT_ROLL	-180	180
14	Pitch (angolo di elevazione)	deg	ADC_OUT_PITCH	-90	90
15	Yaw (angolo di azimuth)	deg	ADC_OUT_YAW	-180	180
16	Rotazione asse X	deg	ADC_OUT_INCLIN_X	-90	+90
17	Rotazione asse Y	deg	ADC_OUT_INCLIN_Y	-90	+90
18	Rotazione asse Z	deg	ADC_OUT_INCLIN_Z	-90	+90
19	Derivata temporale della TAS	m/s <sup>2</sup>	ADC_OUT_TASDOT	-50	50
20	Vertical speed (variometro)	m/s	ADC_OUT_VERTICAL_SPEED	-50	50
21	Tempo GPS	s	ADC_OUT_GPS_TIME	TBD	TBD
22	Clock interno	ms	ADC_OUT_INTERNAL_CLOCK	TBD	TBD
23	Magnetic Heading	deg	ADC_OUT_MAGHEADING	0	360
24	Magnetic Pitch	deg	ADC_OUT_MAGPITCH	0	360
25	Magnetic Roll	deg	ADC_OUT_MAGROLL	0	360
25	Angular Velocity X	deg/s	GIROSC.	-400	400
26	Angular Velocity X	deg/s	ADC_OUT_ROLLRATE	-400	400
27	Angular Velocity Y	deg/s	ADC_OUT_PITCHRATE	-400	400
28	Angular Velocity Z	deg/s	ADC_OUT_YAWRATE	-400	400
29	Body Acceleration X	m/s <sup>2</sup>	ADC_OUT_NX	-98.1	98.1
30	Body Acceleration Y	m/s <sup>2</sup>	ADC_OUT_NY	-98.1	98.1
31	Body Acceleration Z	m/s <sup>2</sup>	ADC_OUT_NZ	-98.1	98.1
32	Latitude	deg	ADC_OUT_LAT	-90	90
33	Longitude	deg	ADC_OUT_LON	-180	180
34	GPS Height	m	ADC_OUT_HGPS	-1000	22000
35	GPS velocity north	m/s	ADC_OUT_VN	-300	+300
36	GPS velocity east	m/s	ADC_OUT_VE	-300	+300
37	GPS velocity up	m/s	ADC_OUT_VU	-300	+300
38	Raw static pressure	Pa	ADC_OUT_STATICPRESS_LOC	22600	108000
39	Raw dynamic pressure	Pa	ADC_OUT_DYNPRESS_LOC	0	6000

Tabella 2 Output

Figure 5.17 SAIFE\_IO table

In the code, for testing purposes the structure includes methods for updating the values using Perlin noise functions to simulate a continuous change inside the minimum and maximum values for each variable. The next section will give an overview of the code implementation, more details are available in the comments inside the source files.

### 5.3.1 INPUT DATA

In order to define an overall structure that holds all input data as per the ICD, a first structure template is implemented (inputData). This is a general structure that holds a name, value, min and max as well as member methods to update the values over time (see Figure 5.X below)

```

14 // inputData is a generic struct that holds the input name, the value (read from the OBC outputs), min and max values as well as the method for updating the value
15 struct inputData {
16
17     // Member variables
18     std::string name;
19     float value;
20     float min;
21     float max;
22
23     // Constructor
24     inputData(const std::string& n, float val, float min, float max) : name(n), value(val), min(min), max(max) {}
25
26     //Member function: used to display the values of each input data (for debugging)
27     void display() const { ... }
28
29     // Member function: fake noise (used for testing/simulating inputs updated with a certain pattern)
30     float updateFakeNoise(float elapsedTime, float duration)
31     {
32         float noiseValue = ofNoise(fmod(elapsedTime*0.1,duration*0.1));
33
34         value = ofMap(noiseValue, 0, 1, min, max);
35
36         return this->value;
37     }
38 };
39
40
41
42
43

```

Figure 5.X inputData structure

A second structure (INPUTS) is then implemented, holding instances of the first structure initialized to reflect the ICD. See figure 5.X below.

```

46 struct INPUTS
47 {
48     // members declaration
49
50     // Initializer (Name, value, min, max). Values are initialized all at 0 and are updated from the first cycle onward via readings from OBC outputs.
51     INPUTS() { ... }
52
53     // The update method calls all the member update methods. Note that some randomize parameters for certain members are scaled down to soften the update speed
54     void update(float elapsedTime, float duration=5) { ... }
55 };
56

```

Figure 5.X INPUTS structure

The INPUT structure is then initialized in the main ofApp class, which calls its update method inside the ofApp::update, as well as all other classes update methods by passing the INPUT structure as input.

## 5.4 GLOBAL VARIABLES

A few global variables and data structures are defined to be shared between different classes.

- **inputData + INPUTS** (C++ structures, see section 5.3 above)
- **mapTiles** (C++ unordered\_map, singleton): This is a singleton, a C++ class that is designed to provide one and only one instance and a global access to this instance. This implementation ensures that the vectors inside the map are correctly handled and freed when the program is exited. This map serves as a cache to load and store the map tiles as the program runs. It is pre-loaded at the start of the program with the tiles downloaded locally from previous runs and inside the min and max lat and lon values as defined in the .ini file. This implementation makes it so new map tiles are queried and downloaded only the first time they are needed, across different runs, to avoid “freezing” of the program every time the tile grid needs to be updated.
- **zoomLevel** (int, singleton): The zoom level is also implemented as a singleton to ensure correct initialization and destruction when the program is exited. The value of the zoomLevel is shared across all instances of the Map class, so that it stays consistent between the PFD and MAP scenes.

## 5.5 MAIN LOOP: ofApp and other functionalities

The main loop of the application is run in the ofApp object, implemented in the ofApp.cpp file. It is structured in 3 main functions, and several listeners functions as detailed below.

- **setup():** The setup method enables and sets up a few parameters for the whole program (such as antialiasing, vertical sync..). It then creates a vector of scenes (PFD, MAP, DATA) and calls the setup method from each of them. If enabled via .ini file, the map tiles are also preloaded inside the zoom range defined in the .ini file. Other variables are initialized, mainly used in the fakenoise functions to update input data.
- **update():** the INPUT\_DATA.update method is called, as well as update methods from each scene class. The AoA and AoS values (vane vs ASSE) are also written to a file if the time interval is greater than the one defined via .ini file.
- **draw():** This method calls the current scene draw method. The DATA draw method is also always drawn before the rest to keep updating the plot of ASSE vs Vane even when the DATA tab is not the selected one. In this method, the 3 toggle buttons for selecting the scenes are also drawn to the screen.
- **mousePressed():** This function "listens" to mouse interactions with the scene and passes the screen coordinates to the currently active scene class for further processing. It also checks whether the mouse is activated inside one of the three scene toggle buttons at the top of the screen, changing the currentScene enum (PFD, MAP, DATA) that is used to call the appropriate update and draw functions from the different scene classes.

*Note that in the current implementation and target HW (Odroid + touchscreen display), the mouse touch event is equivalent to the touch on the screen.*

## **5.5 PORTING TO LINUX (ODROID, UBUNTU MATE)**

Refer to the readme file provided with the source code for detailed indications on how to compile and run this software on the target HW (ODROID, with Ubuntu Mate distribution).

In general, a specific Ubuntu image has to be installed, as it includes the correct drivers for the Mali GPU of the Odroid N2, and the OpenGL version to target within the main.cpp file is OpenGLES (Embedded Systems) version 2.0.

## 6.CONCLUSION

While this project involved the development of an Avionic display with a specific target implementation (SAIFE demonstrator), it can also serve as a modular and scalable base for a more generic implementation of an avionic display, taking advantage of a high level and open source framework which is a far more accessible and ready-to-use tool to develop this type of interactive Human-Machine Interfaces programs.

For the specific use case, the following open points/enhancement opportunities are identified:

- Linking the Avionic Display SW with the actual input data incoming and processed from the sensor suite.
- Optimizing the SW via the use of dynamic cache and multi-threading, in particular in regards to the map tiles.
- Expanding the .ini file with more parameters and options to give additional control to the user on the customization of the program, depending on the use cases and needs.