TÉCNICO LISBOA

Integrated Avionic Systems

Professor Bertinho Costa

Professor Agostinho Fonseca

Simulation of an Aircraft, Generation of the Navigation Solution and Emulation of a TCAS

Aerospace Engineering

23rd of January of 2019

António Negrinho, **81828**

Pedro Trindade, **81612**

Pedro Valdeira, **81227**

Contents

1 Introduction

In this report, we will address the topics regarding the experimental project of the Integrated Avionic Systems course.

The aim of this project was to, in a first stage, present the solution of the navigation equation of a simulated aircraft, using the *X-Plane* flying simulator. Then, a communications protocol would have to be established with other two student groups, in order to transmit data from *X-Plane* to their computers, for them to use in their projects.

Finally, the group would have to, once again, using data from *X-Plane*, emulate a TCAS (Traffic Collision Avoidance System) and establish a communications protocol with other the other groups that also had to emulate a TCAS, in order to simulate a situation of several aircraft crossing the same airspace, to test the response of the created system.

This project proved very helpful for the group to gain a better understanding on, on the one hand, computer networks a communications and, on the other hand, gain a thorough insight on the functioning of a TCAS and all the aspects that need to be taken into account when developing one.

2 Data retrieval, navigation solution and data transmission

In this section, the data retrieval from the *X-Plane* simulator will be described, as well as the navigation solution, and the way in which data is transmitted to other groups.

2.1 Simulator data retrieval and Navigation solution

The *X-Plane* simulator features a lot of functionalities regarding the presentation and exporting of data. An image depicting the available data from the simulator can be seen below.

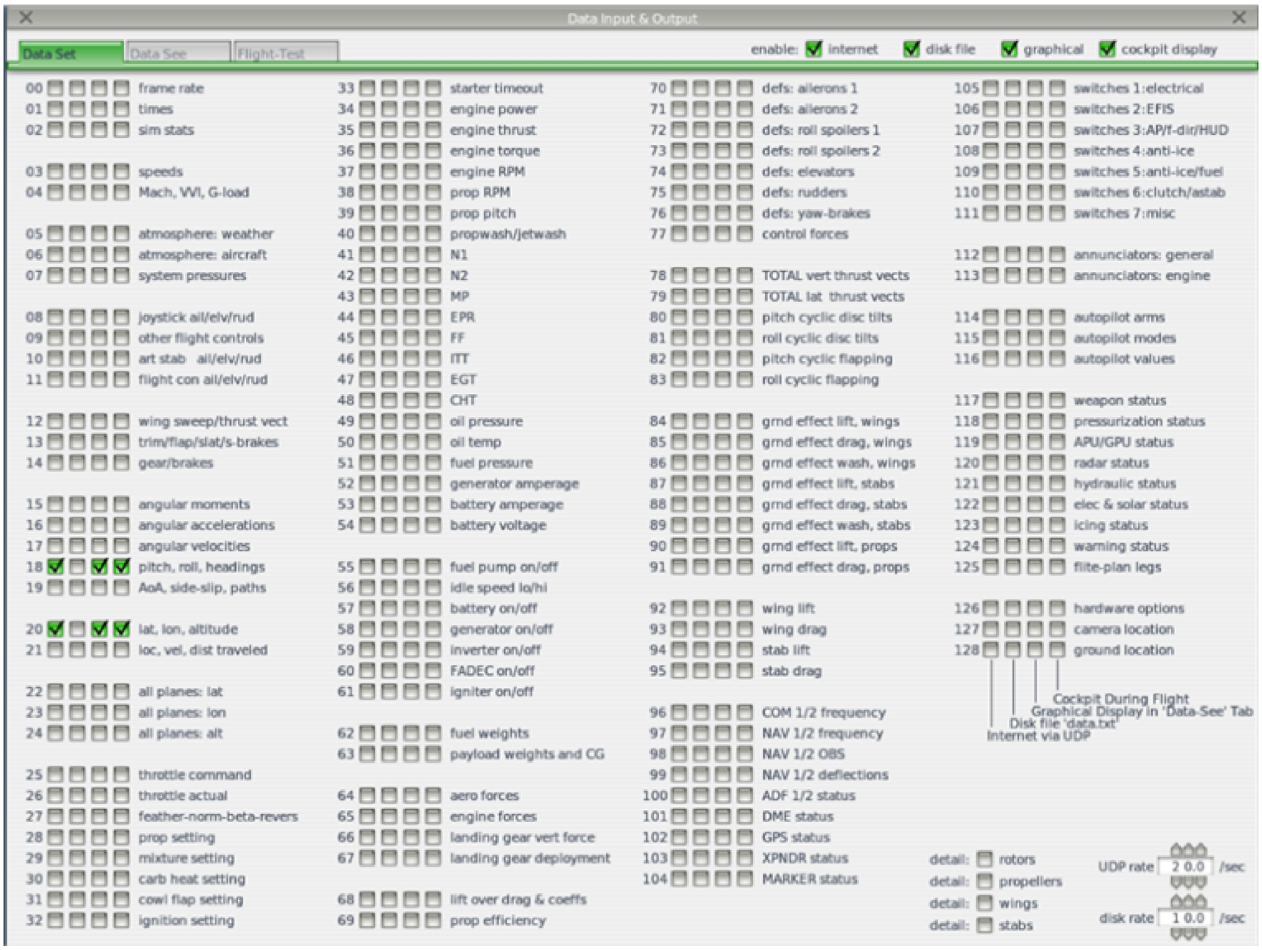


Figure 1: Available data from *X-Plane* for exporting and presenting.

From the available data, illustrated in figure 1, it was chosen the ones the group is interested in, for this application. The data is sent by the simulator in packets, with an header describing the several types of data selected. *X-Plane* uses an UDP (User Datagram Protocol) type of connection to send data to a given endpoint, described by its IP address and port. This was set to send the data to the host's machine IP address (loopback address), at port 9999. This way, the data could then be received and decoded in a program developed by the group, and then resent to other groups, after decoded, in an agreed upon format.

With this in mind, the presenting of the navigation equation was fairly easy, as the needed data were directly retrieved from the simulator and presented in a cockpit view.

2.2 Data Transmission to assigned groups

Once the data is being retrieved from the *X-Plane* simulator, and with a better understanding on how the data is obtained from the simulator, and properly decoded, the group then needed to transmit data to two other groups, in charge of emulating a VOR/DME system and an ILS system (Group E4 - Emulation of output signals generated by VOR and DME receivers, presentation of VOR and DME signals on indicators similar to those of an aircraft; Group E5 - Emulation of output signals generated by an ILS receiver, presentation of ILS signals on indicators similar to those of an aircraft), and in order to do so, it was necessary to set up a communications protocol between three computers, where one would be sending the data (our own) and the other two would be receiving.

For the development of this communications protocol, the group decided to use the *Python* programming language, as it is an object-oriented programming language, whereas C is not, and, taking into account that an emulation of a TCAS display would also be needed (and to re-use some code for that second part of the project) it allows for a much easier usage of graphic libraries than C++ for example (which is also object oriented). Other reasons to choose *Python* are its easier readability and de-bugging capability, and the fact that it is a programming language that is becoming ever more popular, with forecasts showing it may surpass the usage of C/C++ for embedded systems programming.

An UDP (User Datagram Protocol) type of connection was used for transmitting data to the groups in need. This type of internet connection was chosen because of its fastness in data packet transmission, and by request on the part of the receiving groups. In a UDP protocol, packets are sent to the receiver without need for an acknowledgement from the said receiver. This speeds up the transmission but renders the connection more vulnerable to transmission errors. However, since the data transmitted is continuously being broadcast, missing some packets (or receiving them incomplete) will not be critical.

The developed program assumed that the VOR/DME and ILS groups would connect to an IP address and choose a specific port to receive the data, while our group would be continuously sending data. The IP address and port were, obviously, agreed upon between the groups prior to the connection, so to guarantee that the code from all the groups would be coherent. Another aspect that was agreed upon was the data packet format, in order to allow for the other group to decode the sent message.

Firstly, our group decided that the sent data was to be organized in the following way: 12 float (each float size is 4 bytes) values ordered in a specific way. This way, the message would be composed of a 48 bytes packet. Since representation of floating numbers may vary from machine to machine, a standard floating representation was to be used, which was the IEEE 754 32-bit (single precision) floating point representation, described in [4]. This format is illustrated in figure 2.

There is a *Python* library (`struct`[1]), that allows to convert the float data in machine representation to this standard format, and then re-convert when received.

However, after requests by one of the groups to represent the data in a way that would be more easily received by them, the data was actually sent in a "string" format. This way, each float was represented by a sequence of characters, and separated from others using a comma. This however

[1] see `https://docs.python.org/3/library/struct.html` for documentation

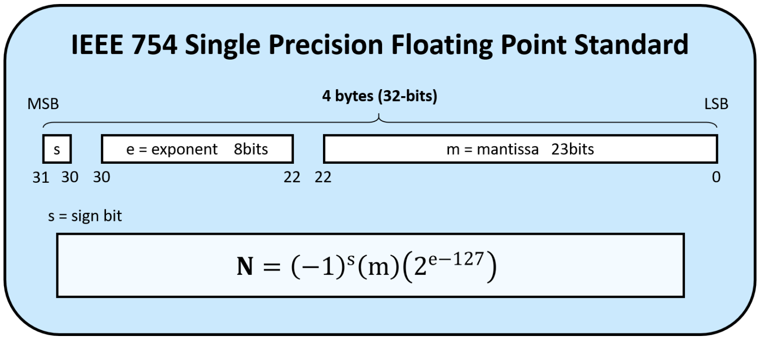


Figure 2: IEEE 754 single precision floating point representation.

will imply a bigger message sent, since now four bytes wont be enough to represent the data. Also, the amount of bytes that will be necessary to represent the data will depend on the value, unlike the float format. For example, it will be necessary 10 bytes to represent the value 10091.1302 (one per each digit, plus the point), but only 5 to represent the value 10.02. Also note that there may be a decrease in precision, depending on the number of digits used in the string representation. The message format is illustrated in figure 3, when passing the floating point value of 12.227, which will require 6 bytes to be represented.

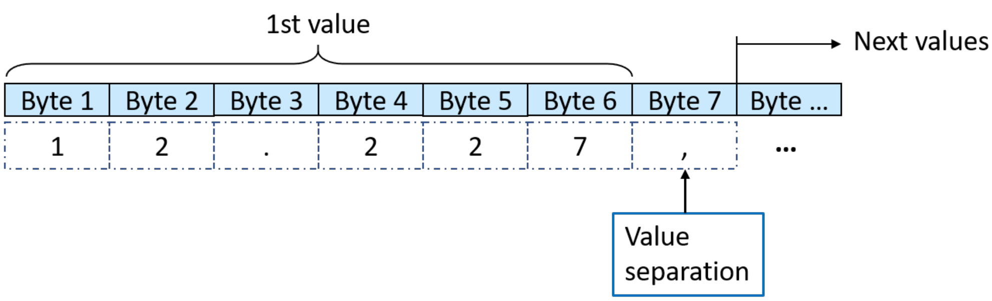


Figure 3: Format of a value in a sent message.

The data sent is ordered as follows, separated by commas (there is no string terminator in the message sent):

1. Latitude (decimal degree);
2. Longitude (decimal degree);
3. Altitude (feet);
4. Climb angle (decimal degree);
5. Heading angle (decimal degree);
6. Ground speed (knots);
7. Indicated Air Speed (knots);
8. Equivalent Air Speed (knots);
9. True Air Speed (knots);
10. ILS Outer Marker (1 or 0);
11. ILS Middle Marker (1 or 0);
12. ILS Inner Marker (1 or 0);

Note some of the data might be unused, but it was prepared in this way, because there was some uncertainty from the part of the receiving groups on what data they were going to need.

As a side note, the data could have been passed directly to the other groups from the simulator. However, we felt that this might not be wise, since XPlane sends the data in float format, using the representation of the host machine (in this case ours). This way, the implementation would depend on the machine, and such is not desirable.

Since the data from the simulator is being read and decoded by our program, this can be easily resent to the groups developing the VOR/DME and ILS emulations. A diagram illustrating the connection to these groups is presented in figure 4.

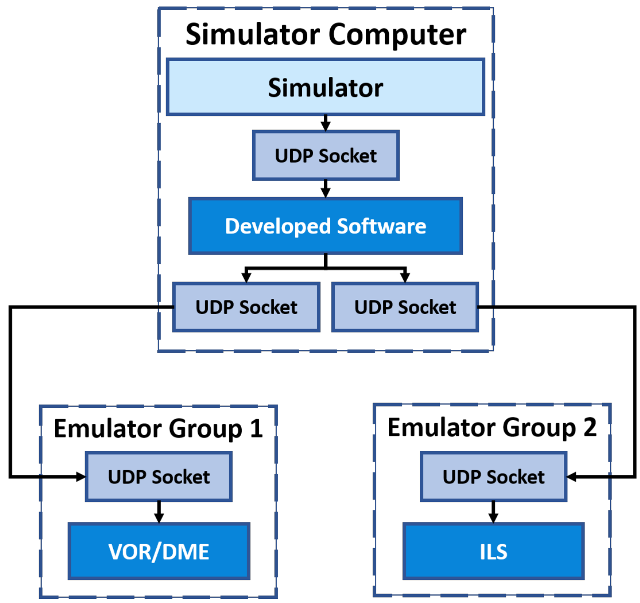


Figure 4: Diagram representing the connection to other groups.

These connections are implemented using *Python* objects to represent them, as will be further described a posterior section of this report. Also, the code can be visualized in appendix.

3 Analysis of a TCAS

A Traffic Collision Avoidance System (TCAS) is a system designed in order to avoid mid-air aircraft collision, as the name implies. It functions by monitoring the airspace surrounding an aircraft, identifying other transponder-equipped aircraft in the vicinity and warning the pilot of those that could pose a threat of mid-air collision. This system is independent of air traffic control (ATC) services and pilots should follow TCAS orders instead of ATC orders in a situation where both are issued.

The TCAS is mandatory, by ICAO, to be fitted in all aircraft with that carry more than 19 passengers or possess a maximum take-off weight (MTOW) of more than 5700 Kg.

The equipment that composes a TCAS is: Computer Unit, Antennas and a Cockpit Display. The computer unit is responsible for all the computations regarding advisories, trajectories, building the 3-D representation of the surrounding spaces and tracking intruder aircraft. The antennas are responsible for sending and receiving the radio-frequency signals. There are usually 4 antennas, two for the TCAS signals (one mounted on the top and the other on the bottom of the aircraft), and two to enable the transponder mode S, which is necessary for the TCAS well-functioning (once again, one on the top and one on the bottom of the aircraft). Finally, the cockpit display is responsible for presenting to the pilot and co-pilot all the information relative to the TCAS, such as surrounding traffic and advisories.

A TCAS functions through the usage of a transponder that actively and continuously interrogates other transponder-equipped aircraft, within a certain range (usually 20 nautical miles), about their position, and responds with its own position to inquiries from other aircraft. The radio bands used for these communications are the 1.03GHz, for inquiries, and 1.05GHz, for responses.

With the received data from other aircraft, the TCAS is able to construct a 3-D map of its surrounding aircraft and warn the pilot of possible situations of mid-air collision, through estimations of future positions of the said surrounding aircraft (using the position information, and performing its derivative over time, to obtain velocity estimates). A sort of "safety bubble" is, thus, created surrounding the aircraft. This "safety bubble" is divided into zones, depending on the distance to the aircraft, resulting in different advisories being issued, for different zones.

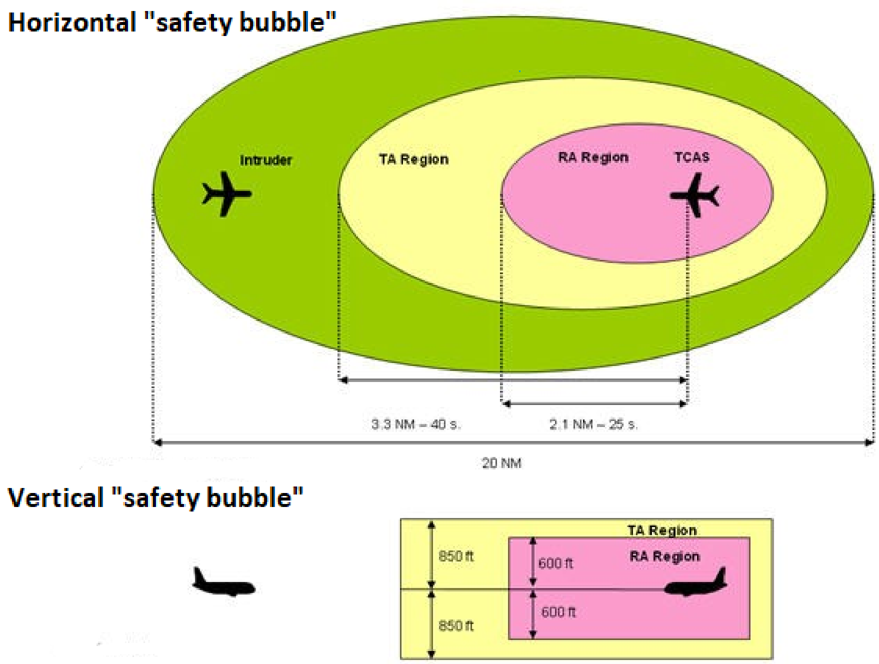


Figure 5: TCAS generated "safety bubble"

The TCAS of the several aircraft communicate with one another, in order to agree upon a maneuver that will avoid a collision between the aircraft. The information regarding these maneuvers is then displayed in the cockpit or reported through pre-recorded audio warnings. Currently, the TCAS is only capable of issuing orders for altitude change or climb/descent rate change. The TCAS resolves conflicts by the pair, meaning it cannot resolve a 3-way conflict, instead solving 2-way one and then the other.

3.1 TCAS protocol, alert description and display

3.1.1 Protocol: transmission and reception

Transponders in aviation communicate according to the following protocols:

- Military modes: 1-5
- Civilian modes: A, C, S

Mode A: provides the identification of the aircraft through a 4-digit octal code as assigned by ATC (Air Traffic Control). The selection of this code is controlled in the cockpit. Often combined with Mode C. This mode alone does not have capability for communication with the TCAS of other aircraft.

Mode C: provides pressure altitude of the aircraft. Usually, it comes combined with Mode A. This mode has capability for communication with the TCAS of other aircraft.

Mode S: Provides multiple information formats to a selective interrogation. Each aircraft is assigned a fixed 24-bit address. Upon interrogation, Mode S transponders transmit information about the aircraft to the SSR system, to TCAS receivers on board aircraft and to the ADS-B SSR system. This mode has capability for communication with the TCAS of other aircraft.

3.1.2 Types of alerts transmitted by a TCAS

A TCAS can provide three types of alerts to the pilot and co-pilot, by visual (cockpit display) and audio (pre-recorded audio files) means.

- **Traffic Advisory (TA)** - When this alert is issued, the pilot should begin visual screening of its surroundings, from the cockpit, in order to establish visual contact with proximate aircraft, that may pose a threat to its own.
- **Resolution Advisory (RA)** - When this alert is issued, there is danger of imminent mid-air collision. Pilots **must** immediately respond to the issued command, unless it would further endanger the safety of the flight. Since ATC is not aware of the RA, until it is communicated by the crew, pilots may ignore or circumvent ATC orders, when resolving the issued RA. Possible RA orders include: Climbing, Descending, Increasing/Reducing rate of climb/descent or maintaining those rates.
- **Clear of Conflict (CC)** - When this alert is issued, the intruding aircraft is no longer considered a threat and the pilot should return to their assigned flight level and follow all ATC orders. Regular flight is resumed.

3.1.3 Typical TCAS display

A depiction of a typical TCAS display, along with an explanation for its symbology can be seen below.

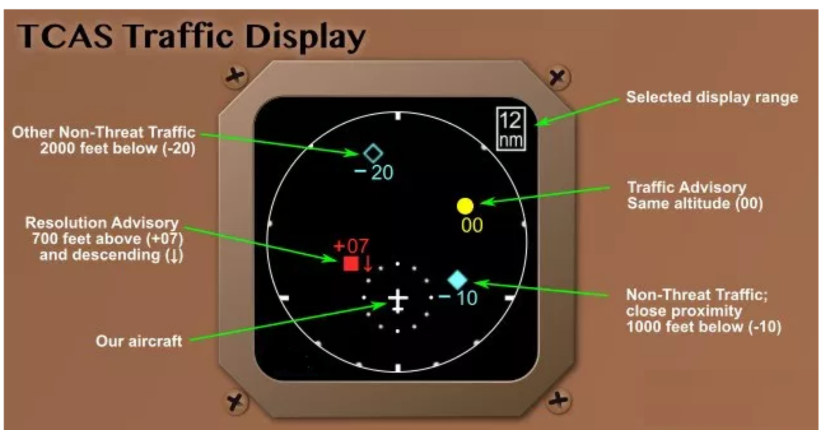


Figure 6: Typical TCAS display.[3]

In the TCAS display, aside from our own airplane's symbol, we can see four different symbols. A hollowed diamond represents regular non-threat air traffic (more than 20Nm in distance); a filled diamond represents non-threat traffic but closer in distance (between 20NM and 3.3NM); a yellow circle represents traffic for which a TA has been issued (between 3.3Nm and 2.1Nm); a red square represents traffic for which a RA has been issued (less than 2.1Nm). There can also be represented, alongside the symbols, numbers and arrows. An upward pointing arrow indicates the airplane is increasing its altitude, whereas a downward pointing one, represents the opposite. The numbers indicate an intruding aircraft's relative altitude to our own, with a positive value indicating it is above and a negative one indicating it's below. The value for the relative altitude is an integer divided by 100, meaning that, for example, 06 will represent 600ft and 14 would represent 1400ft.

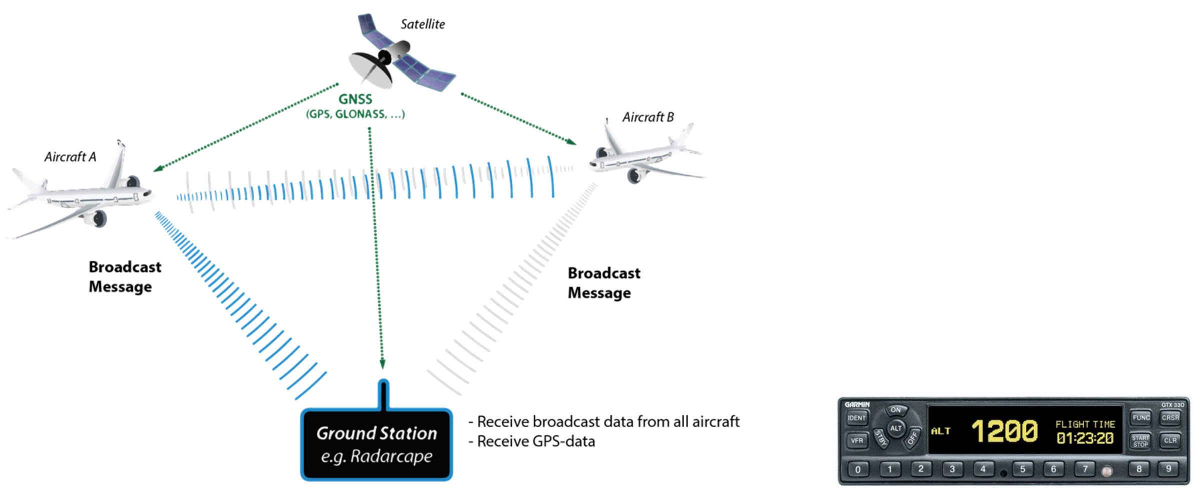
3.2 Coordinates conversion

In order to achieve a functioning TCAS one must convert the available position related data, the information received, from geographic coordinates into a coordinate system more suitable to the application in hands, the TCAS system.

The original information is received via network cable in the designed system, although this merely simulates the real life transmission through ADS-B (Automatic Dependent Surveillance – Broadcast) or the SSR transponder's Mode S. This information has its origin in the other group's systems, which simulate other aircraft, being composed of:

- Heading;
- Speed;
- Climb angle;
- Aircraft's coordinates in LLH (Geographic Coordinate).

It is worth noting that speed and climb angle are used in the calculation of the vertical speed, which although not directly transmitted is the truly valuable information carried by these variables.



(a) ADS-B. (b) SSR Transponder.

Figure 7: ADS-B e ssr Transponder

LLH (Latitude, Longitude, Altitude) or **Geodetic** coordinates, consists of a Geographic Coordinate System, composed of latitude, longitude and elevation, as represented in the spherical representation of Earth in Figure 8, where the elevation corresponds to the distance from the point corresponding to the aircraft's location projected in the sphere's surface (black dot) to the airplane's actual location

This is the starting point from which the TCAS coordinates must be derived.

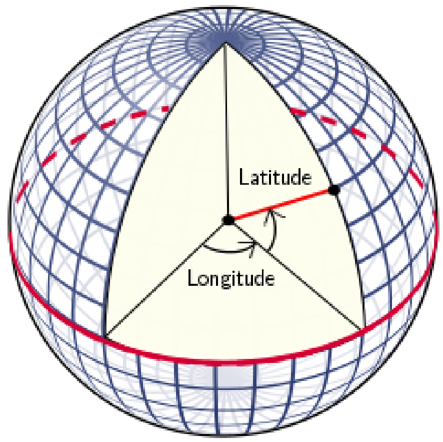


Figure 8: Geographic coordinate system.

3.2.1 LLH to ECEF

The first of these transformations is the conversion from LLH to ECEF coordinates (Earth-Centered, Earth-Fixed). ECEF is a Cartesian Geographic coordinate system, with its origin in the Earth's center of mass (i.e., it is a geocentric coordinate system).

The system's positions are represented by (X,Y,Z) coordinates where, as can be viewed in Figure 9: the x-axis corresponds to the intersection of 0^o latitude (Equator) and 0^o longitude (Greenwich meridian), leading to ECEF rotating with the earth, hence being earth-fixed and allowing for a point fixed on the surface of the earth to not change location; the z-axis passes through the true north, which differs from the instantaneous earth rotational axis, leading to a relative polar movement; the y-axis is defined as such that (X,Y,Z) is orthogonal.

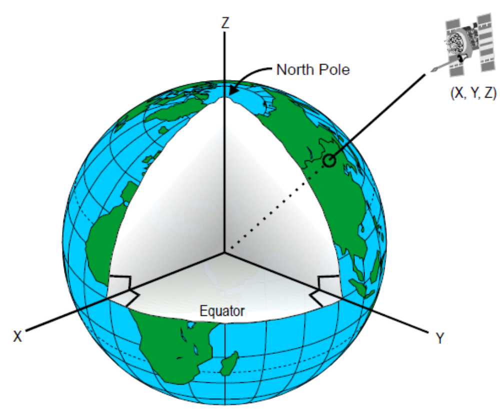


Figure 9: ECEF point.

Regarding the transformation from geodetic coordinates (latitude ϕ, longitude λ, height h) to ECEF coordinates, the following equations can be used:

$$\begin{cases} X = (N(\phi)+h)\cos\phi\cos\lambda \\ Y = (N(\phi)+h)\cos\phi\sin\lambda \\ Z = \left(\frac{b^2}{a^2}N(\phi)+h\right)\sin\phi \end{cases} , \quad N(\phi) = \frac{a^2}{\sqrt{a^2\cos^2\phi + b^2\sin^2\phi}} = \frac{a}{\sqrt{1-e^2\sin^2\phi}}$$

Where a and b correspond to the equatorial and polar radius respectively, i.e., the semi-major axis and the semi-minor axis; and $e = \sqrt{1-\frac{b^2}{a^2}}$ corresponds to the eccentricity of the ellipsoid. $N(\phi)$ is called *prime vertical radius*.

3.2.2 ECEF to ENU

After computing the formerly exlpainned transformation, we now have the aircraft's data in terms of ECEF coordinates. The next step is to convert these into ENU (East, North, Up) coordinates. The relationship between these two coordinate systems is illustrated in the following pictures (Figure 10), where left one aims to show an overall view of the coordinate systems and a more in depth explanation of the relation between ECEF and ENU coordinate systems.

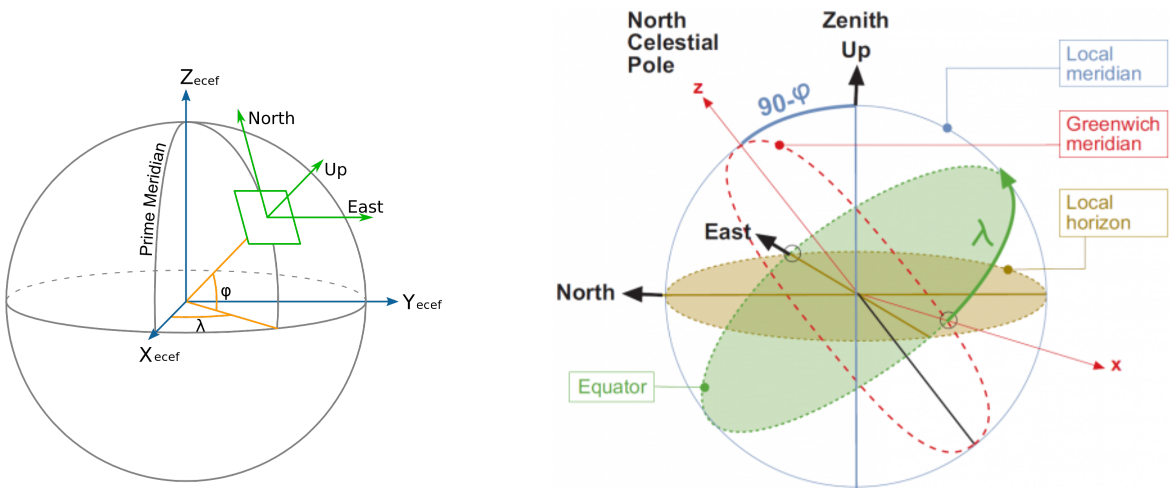


Figure 10: ENU coordinate system in relation to ECEF.

As can be seen in the illustration on the right, using the exact same notation, the transformation can be decomposed into two different rotations:

- An anti-clockwise rotation over east-axis of $90 - \varphi$ degrees;
- A clockwise rotation over z-axis of $90 + \lambda$ degrees.

These two rotations along different axis can be expressed, resorting to some linear algebra, by the aplication of the following linear transformation (rotation matrix) to the $[x, y, z]^T$ ECEF relative position, obtaining $[E, N, U]^T$, the coordinates of the same point represented in the ENU coordinate system:

$$\begin{bmatrix} E \\ N \\ U \end{bmatrix} = \begin{bmatrix} -\sin\lambda & \cos\lambda & 0 \\ -\cos\lambda\sin\varphi & -\sin\lambda\varphi & \cos\varphi \\ \cos\lambda\sin\varphi & \sin\lambda\cos\varphi & \sin\varphi \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Note that the ECEF relative position is $[x, y, z]^T = [X_t - X_0, Y_t - Y_0, Z_t - Z_0]^T$, where $[X_t, Y_t, Z_t]^T$ are the target aircraft's coordinates in ECEF and $[X_0, Y_0, Z_0]^T$ are the TCAS's aircraft coordinates in ECEF.

So, at this point in the coordinate systems conversion process we have already transformed the Geodetic coordinates into ECEF ones and in turn, transform these into ENU ones. The only transformation missing is from the ENU coordinate system into the relative positions to be used in the TCAS display.

3.2.3 ENU to TCAS display

TCAS uses ENU coordinates, however, in order to display this data, some details have to be taken into consideration.

Altitude is not graphically represented, only an altitude difference is indicated next to each aircraft. So, in terms of graphical representation it is as if a 2D projection of the airspace is made on a horizontal plane, with the 3D space being squashed along the altitude component of the position.

On top of this, the heading's representation is also changed to simplify the interpretation of the TCAS, more specifically, the orientation is set according to the TCAS's aircraft's heading as a reference angle, as if it were 0^o (straight ahead), with other angles being read in relation to that. Once again, relative position is the one that truly matters, although other absolute values can also be computed from this information.

Lastly, it is also required to scale the positions obtained from the two previous computations so that the final values to be represented fit the TCAS's display window.

The resulting graphical display is presented in Figure 11, where a grid composed of two parts is used in order to facilitate the reading of other aircraft's relative position: part of this grid consists of circumferences of different radii centered around the TCAS's location, in this case, 20 (outter circumference, *i.e.*, range), 15, 10 and $5NM$ (innermost circle); the rest of the grid is formed by a set of line segments along points with the same orientation regarding the TCAS's aircraft (with a 45^o spacing), allowing for an easier identification of the angle at which a certain aircraft is.

The value shown in green in the top center of the display (0^o in the case presented), corresponds to TCAS's aircraft heading (used as a reference), while the outermost circumference of the grid (which in this case happens to have North aligned with the heading value, since it is 0^o) varies turns in clockwise or anticlockwise directions according to the changes in heading.

Different aircraft are represented accordingly to the symbology presented in section 3.1.3, as to maintain technical accuracy to the highest level. An explanation for the meaning of those symbols can be seen in that same section.

Note that the area of the circle around its center does not include the grid that is present in the remainder of the TCAS, as to not overload this section with graphical information, since the continuation of the radius axes would be too close together, making it difficult to read information about nearby aircraft, compromising the TCAS's functioning and endangering the flight. The screen shot presented in Figure 11 presents a situation in which this characteristic proves necessary.

On top of this graphical representation of positional data, some additional information is provided in the top left and right corners of the TCAS display, more specifically, *Ground Speed* and *True Air Speed* of the aircraft on the left (both in *knots*), and the *Range* covered by the TCAS (radius of the circle partially represented) on the right (in NM).

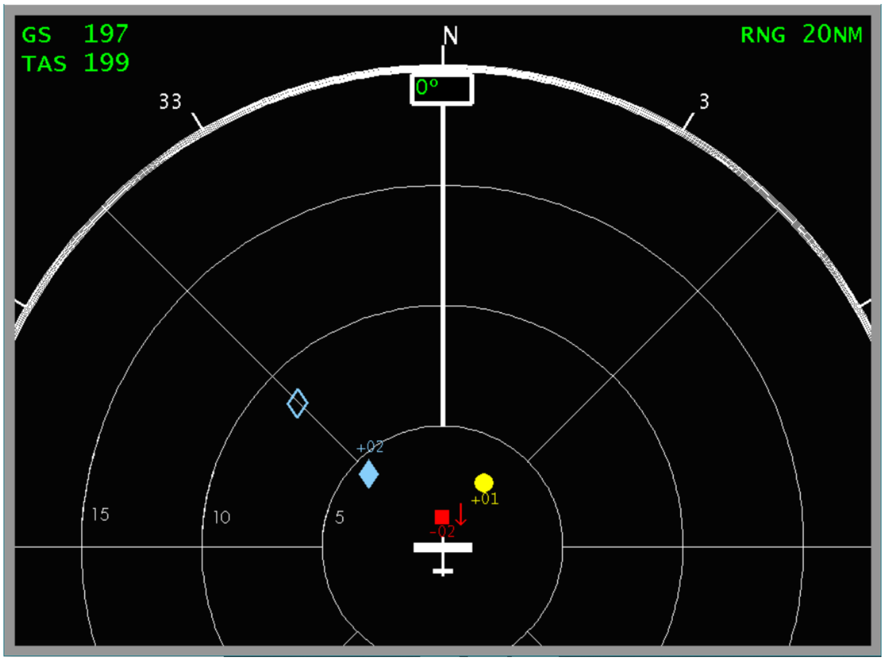


Figure 11: TCAS display

4 Implementation

In this section we will address the implementation of the TCAS and the respective communication protocol with the other groups that also developed a similar system. All the code created was also developed in *Python*, for the same reasons presented in section 2. Also, since it was necessary to install the simulator, and due to hard drive constraints, this had to be done in the Windows OS, *Python* was simpler to use in Windows than for example C++, which proved difficult to use (for example, socket and threading libraries are slightly different from Linux, and command line usage quite different).

For the communication between TCAS, the groups responsible for those systems agreed upon using the TCP (Transmission Control Protocol) protocol, which seemed fitter for the purpose, since it provides high reliability due to the connection "handshake"[2]. This choice was motivated by the fact that now 5 groups would be connected at the same time, thus requiring better monitorization and robustness in communications. It was critical that all the connected groups did not lose data packets and that the data reception was organized and coherent. Since the developed project was a TCAS, it was even more important to have a robust data transmission/reception, to ensure the system would operate effectively.

With this in mind, the envisioned communication protocol should be robust to failures, and communication should not depend on a "main server". For this reason, it was chosen a fully connected network topology (structure of the connections made between the several subsystems), in order to increase integrity. Whenever a system fails, all others can continue communication and detect that one has failed. This network topology is illustrated by figure 12.

[2]see `https://www.diffen.com/difference/TCP_vs_UDP` for a comparison between UDP and TCP

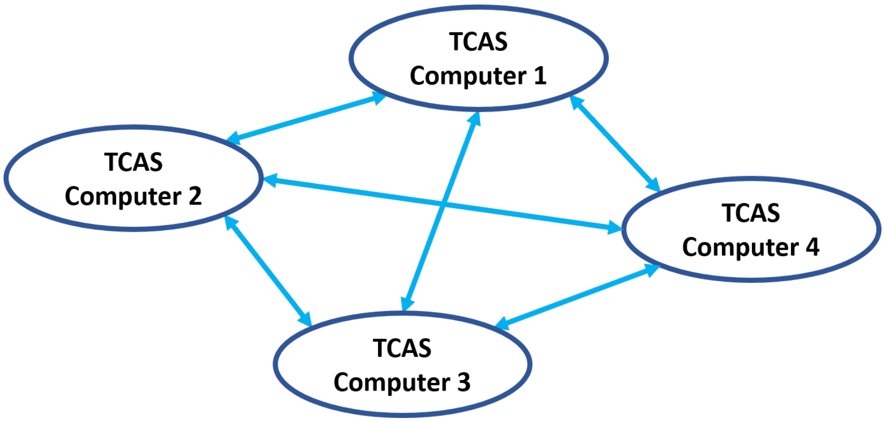


Figure 12: Fully connected network (Example with 4 nodes).

A fluxogram representing the operation of the developed program for connection between computers is also presented, in figure 13.

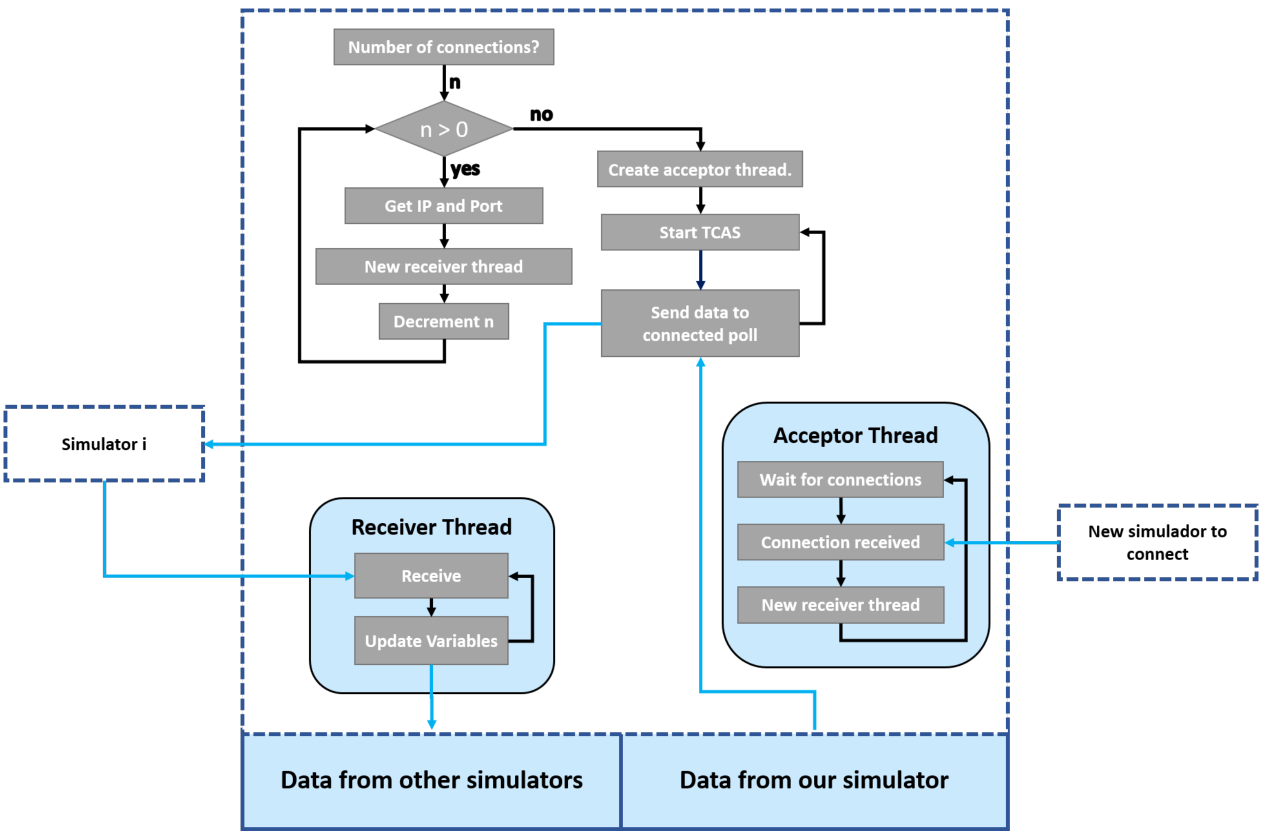


Figure 13: Fluxogram of the program's functioning.

In order to provide a modular implementation, for ease of use and integration of the several systems and parts of the software, objects were used to implement the several components. The relevant object classes developed are the following:

- **XPlaneConnection()** - This class is used to represent the connection to the simulator. It holds a socket and a thread for data reception from the simulator, and holds the lastly received simulator data. This data can be accessed by using an appropriate class method. Also, the class implements mutual exclusion internally, so one can simply call the method to retrieve most recent data. Also, the data is decoded in the receiving thread from the class, upon reception, using an internal class method.

- **TcasConnection()** - An object form this class represents a connection another simulator group. Each object from this class contains a socket for the connection, and a receiving thread, similarly to the **XPlaneConnection()** class. This object holds the lastly received data from another group, and has a method to decode the received data, and to access it, also implementing mutual exclusion inside the method. Further, there is a method to send new data through this connection.

- **EmulatorConnection()** - An object of this class represents a connection to one of the groups emulating other systems. This is a slightly different kind of connection, thus having another class. Each object hold a socket for the connection, and has a method to send data. Note there is no need for a thread here, since we are using UDP protocol, so we wont get blocked sending data.

- **Acceptor()** - An instance of this class holds an acceptor socket, which was binded to an endpoint, and it is listening for new connections. When a new request for a connection is made, it creates a new socket, associated with that connection. Then, it creates a new object of class **TcasConnection()** (which receives the new socket) to internally handle the new connection.

- **TCAS()** - This class is used to perform the TCAS computations, store data and prepare the graphical interface. It contains a method to update the data from all the airplanes (done at a constant rate), which is acquired from all the **Connection()** class objects and the **XPlaneConnection()** class object, using the appropriate methods. Then it makes the appropriate computations and update the graphical window. It also contains methods to add and remove airplanes.

In this way, it was simple to develop the code separately for each module, and this way simplified the integration of all parts.

In order to implement the graphical interface, the *Python*'s *pygame* library was used, which provides simple and useful functions to perform our graphics, and a very intuitive interface.

To keep up with different number of groups connected, a list is kept, with all the current connections. Every time new connection is received, by the acceptor thread, this is added to the list. Every time that another group disconnected, this connection was removed from the list. It is important to notice that the implementation is not dependent on the number of airplanes, and so the program is scalable, i.e., we could connect any number of groups, since the list is dynamically allocated.

5 Results

In order to visualize the functioning of the developed TCAS in conjunction with the TCAS developed by the other groups responsible for such systems, a workshop-type session was arranged by the professors of this course, where 5 computers would be connected through a desktop switch, using Ethernet cables, running the simulator and the TCAS program at the same time.

Prior to this, however, the groups responsible for developing TCAS tested their software together to ensure the well-functioning of it as both a stand-alone solution and in network environment.

Bellow, in Figure 14, part of the demo presented in class can be seen. More specifically, the part respecting the exhibiting of the TCAS's functioning while the *X-Plane* simulator of different computers are connected.

Figure 14: Simulator window along with TCAS display (demo).

6 Conclusion

With this experimental project, the group managed to acquire a deep insight on the functioning of, firstly, a computer network for transmission and receiving of data and the TCP and UDP internet transmission protocols. Secondly, the group managed to obtain first-hand experience with the TCAS, through the developing of such system, having to take into account its nuances. It was also of high interest and value for the group the testing of this system with others of the same kind, developed by other groups, and visualize its response in a simulated scenario of a possible air collision.

With the conclusion of this project, the group acknowledges it has fulfilled the following objectives:

- Familiarization with the *X-Plane* flying simulator and its data exporting interface
- Presentation of the solution of the navigation equation in the cockpit view
- Transmission of the relevant data for the two groups in charge of emulating a VOR/DME system and an ILS

- Developing of a functional TCAS, utilizing data retrieved from the *X-Plane* flying simulator
- Transmission of TCAS relevant data to the other four groups responsible for, also, developing a TCAS

Bearing all this in mind, the group feels it has completed the established goals for this project and is fully satisfied with the results it has obtained.

Repository
https://github.com/ptrindade96/SAInt-Proj

References

[1] B. Hofmann-Wellenhof; H. Lichtenegger; J. Collins. GPS - theory and practice.

[2] ESA - Transformations between ECEF and ENU coordinates.
Available at:
https://gssc.esa.int/navipedia/index.php/Transformations_between_ECEF_and_ENU_coordinates

[3] TCAS display image.
Available at: https://aerosavvy.com/tcas/

[4] 754-2008 - IEEE Standard for Floating-Point Arithmetic
Available at:
https://ieeexplore.ieee.org/document/4610935

Appendices

A Developed code

A.1 main.py

```python
import sys
import time
import numpy
import threading
import Definitions as Prompt
from Definitions import Global
from Connections import TcasConnection
from Connections import EmulatorConnection
from Connections import Acceptor
from XPlane import XPlaneConnection
import TCAS


##############################################################################
##                              Main function                              ##
##############################################################################
if __name__ == '__main__':
    print("\n------------------------------------------------------------")
    print("------------------ Starting Program ------------------")
    print("------------------------------------------------------------")

    # Begin connection with XPlane
    Simul = XPlaneConnection()
    XPlane_port = Prompt.port("XPlane")
    while not Simul.connect(XPlane_port):
        print("XPlane connection failed.")
        answer = input("Try again?[y/n]  ")
        if answer == 'n':
            sys.exit(5)
        elif answer == 'y':
            XPlane_port = Prompt.port("XPlane")

    EMUL_ON = Prompt.is_on("emulator")
    if EMUL_ON:
        n = Prompt.how_many("emulator")
        emulator_connections = []
        i = 1
        while n > 0:
            ip = Prompt.ip("emulator " + repr(i))
            port = Prompt.port("emulator " + repr(i))
            C = EmulatorConnection(ip,port)
            emulator_connections.append(C)
            n = n - 1
            i = i + 1

    TCAS_ON = Prompt.is_on("TCAS")
    if TCAS_ON:
        n = Prompt.how_many("TCAS")
        tcas_connections = []
        i = 1
        while n > 0:
```

```
            ip = Prompt.ip("TCAS number "+repr(i))
            port = Prompt.port("TCAS number "+repr(i))
            C = TcasConnection(ip, port)
            if C.connect():
                C.start_rcv()
                tcas_connections.append(C)
                n = n - 1
                i = i + 1
            else:
                print("Unable to connect to endpoint. Try inserting again.")
        Tcas = TCAS.TCAS()
        Global.MUTEX_CONN.acquire()
        for connection in tcas_connections:
            Tcas.new_airplane(connection.id,connection.get_values())
        Global.MUTEX_CONN.release()
        A = Acceptor(tcas_connections, Global.MY_PORT, Tcas)
        A.start_accept()
        print("Started accepting connections on port " + repr(Global.MY_PORT))

    if not TCAS_ON and not EMUL_ON:
        print("So... you wish to do nothing...")
        print("Then I will exit...")
        Simul.stop_simulation()
        sys.exit(0)

    while Simul.is_on():
        my_values = Simul.get_values()
        others_values = []
        if TCAS_ON:
            Global.MUTEX_CONN.acquire()
            for connection in tcas_connections:
                connection_values = connection.get_values()
                connection_values['id'] = connection.id
                others_values.append(connection_values)
                if not connection.send(my_values): # connection died..
                    tcas_connections.remove(connection)
            Tcas.update(my_values, others_values)
            Global.MUTEX_CONN.release()
            Tcas.clock.tick(TCAS.SAMPLING_TIME)
            if Tcas.exited():
                Tcas.quit()
                TCAS_ON = False
        if EMUL_ON:
            for connection in emulator_connections:
                connection.send(my_values)
        time.sleep(Global.SAMPLING_TIME)


    print("Not receiving from XPlane")
    Global.FINISH = True
    if TCAS_ON:
        Tcas.quit()
        A.stop_accept()
        tcas_connections = []
    print("Exiting.")
    sys.exit(0)
```

A.2 Definitions.py

```python
from threading import Lock as Mutex


class Global:
    SAMPLING_FREQ = 15
    SAMPLING_TIME = 1/SAMPLING_FREQ
    MY_PORT = "8000"
    MUTEX_CONN = Mutex()
    FINISH = False


def is_on(str):
    while True:
        answer = input("You wish to connect to " + str + " systems?[y/n]    ")
        if answer == "y":
            return True
        elif answer == "n":
            return False
        else:
            print("Unvalid answer. Please, answer 'y' or 'n'.")

def how_many(str):
    while True:
        try:
            answer = input("How many " + str + " systems?    ")
            N = int(answer)
            if str == "TCAS" and N>=0:
                return N
            elif str == "emulator" and N>0:
                return N
            else:
                print("Not a valid number... Try again.")
        except:
            print("Not a valid number... Try again.")

def ip(whos):
    while True:
        try:
            ip = input("Insert " + whos + " ip:   ")
            return ip
        except:
            print("Invalid input. Please try again.")

def port(whos):
    while True:
        try:
            port = int(input("Insert " + whos + " port:    "))
            return port
        except:
            print("Invalid port. Please try again.")
```

A.3 Connections.py

```python
import socket
import errno
import struct
from threading import Thread
from threading import Lock as Mutex
from Definitions import Global
```

```

# self.valid does not require mutex (atomic variable -> atomic operations)

class Acceptor:

    def __init__(self, connections, port_,Tcas_):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.threadAcceptor = Thread(target=self.__acceptor__)
        self.connections = connections
        self.port = int(port_)
        self.Tcas = Tcas_
        return

    def __del__(self):
        self.sock.close()
        return

    def __acceptor__(self):
        self.sock.bind(('0.0.0.0', self.port))
        self.sock.listen(1)
        while not Global.FINISH:
            try:
                sock, addr = self.sock.accept()
                conn = TcasConnection('0.0.0.0', self.port, sock)
                conn.start_rcv()
                Global.MUTEX_CONN.acquire()
                self.Tcas.new_airplane(conn.id,conn.get_values())
                self.connections.append(conn)
                Global.MUTEX_CONN.release()
            except:
                if not Global.FINISH:
                    print("Accept error.")
                    Global.FINISH = True
                else:
                    try:
                        self.sock.close()
                    except:
                        pass
                    return

    def start_accept(self):
        self.threadAcceptor.start()
        return

    def stop_accept(self):
        try:
            self.sock.close()
        except:
            pass
        return

class TcasConnection:

    count = 100

    def __init__(self, ip_, port_, sock_=None):
        self.ip = ip_
        self.port = port_
        self.thread_rcv = Thread(target=self.__receive__)
```

```
        self.mutex_values = Mutex()
        if sock_== None:
            self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock_
        TcasConnection.count += 1
        self.id = TcasConnection.count
        self.values = self.__init_values__()
        self.valid = True
        return

    def __del__(self):
        self.sock.close()
        try:
            self.thread_rcv.join()
        except:
            pass
        return

    def connect(self):
        try:
            self.sock.connect((self.ip, self.port))
            return True
        except socket.error as e:
            print(e)
            self.valid = False
            return False

    def start_rcv(self):
        self.thread_rcv.start()
        return

    def get_values(self):
        self.mutex_values.acquire()
        return_values = self.values.copy()
        self.mutex_values.release()
        return return_values

    def __update_data__(self, data):
        list_ = struct.unpack("!5f", data)
        self.values["lat"] = list_[0]
        self.values["lon"] = list_[1]
        self.values["alt"] = list_[2]
        self.values["vpath"] = list_[3]
        self.values["GS_knots"] = list_[4]
        return

    def __receive__(self):
        while not Global.FINISH:
            if self.valid:
                try:
                    data = self.sock.recv(20)
                    self.mutex_values.acquire()
                    self.__update_data__(data)
                    self.mutex_values.release()
                except socket.error as e:
                    if e.errno == errno.ECONNRESET:
                        print("Client disconnected")
                        self.valid = False
```

```
                        return
                except:
                    pass
            else:
                return

    def send(self, data):
        if self.valid:
            try:
                a = data["lat"]
                b = data["lon"]
                c = data["alt"]
                d = data["vpath"]
                e = data["GS_knots"]
                buff = struct.pack("!5f", a, b, c, d, e)
                self.sock.send(buff)
            except socket.error as e:
                if e.errno == errno.ECONNRESET:
                    print("Client disconnected")
                    self.valid = False
        return self.valid

    @staticmethod
    def __init_values__():
        values = dict()
        values["lat"] = 0
        values["lon"] = 0
        values["alt"] = 0
        values["vpath"] = 0
        values["GS_knots"] = 0
        return values


class EmulatorConnection:

    def __init__(self,ip_,port_):
        self.ip = ip_
        self.port = port_
        self.emulator_addr = (self.ip,self.port)
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        return

    def connect(self):
        try:
            self.sock.bind((self.IP, self.PORT))
            return True
        except socket.error as e:
            print(e)
            return False

    def send(self, values):
        try:
            string = str(values["lat"]) + ','
            string += str(values["lon"]) + ','
            string += str(values["alt"]) + ','
            string += str(values["vpath"]) + ','
            string += str(values["hpath"]) + ','
            string += str(values["GS_knots"]) + ','
            string += str(values["IAS_knots"]) + ','
```

```
            string += str(values["EAS_knots"]) + ','
            string += str(values["TAS_knots"]) + ','
            string += str(values["ILS_OM"]) + ','
            string += str(values["ILS_MM"]) + ','
            string += str(values["ILS_IM"])
            self.sock.sendto(string.encode('ascii'), self.emulator_addr)
            return True
        except socket.error as e:
            print(e)
            return False
```

A.4 XPlane.py

```
import socket
import struct
from threading import Thread
from threading import Lock as Mutex
from Definitions import Global


class XPlaneConnection:

    IP = '127.0.0.1'
    TIMEOUT = None

    def __init__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.thread_recv = Thread(target=self.__rcv_thread__)
        self.mutex_values = Mutex()
        self.values = self.__init_values__()
        self.OK = False
        return

    def connect(self, port_):
        try:
            self.port = port_
            self.sock.bind((XPlaneConnection.IP, self.port))
            self.OK = True
            self.thread_recv.start()
            return True
        except:
            return False

    def is_on(self):
        return self.OK

    def get_values(self):
        self.mutex_values.acquire()
        return_values = self.values.copy()
        self.mutex_values.release()
        return return_values

    def stop_simulation(self):
        self.OK = False
        return

    def __rcv_thread__(self):
        self.sock.settimeout(XPlaneConnection.TIMEOUT)
        while self.OK and not Global.FINISH:
```

```
            try:
                data = self.sock.recv(1024)
                self.mutex_values.acquire()
                self.__decode_data__(data)
                self.mutex_values.release()
            except:
                self.OK = False
                self.sock.close()
                return
        self.OK = False

    def __decode_data__(self, data):
        i = 0
        while 5 + (i+1)*36 <= len(data):
            d = struct.unpack('i8f', data[5 + i * 36:5 + (i + 1) * 36])
            if d[0] == 20:
                self.values['lat'] = d[1]
                self.values['lon'] = d[2]
                self.values['alt'] = d[3]
            elif d[0] == 19:
                self.values['hpath'] = d[3]
                self.values['vpath'] = d[4]
            elif d[0] == 3:
                self.values['IAS_knots'] = d[1]
                self.values['EAS_knots'] = d[2]
                self.values['TAS_knots'] = d[3]
                self.values['GS_knots'] = d[4]
            elif d[0] == 104:
                self.values['ILS_OM'] = d[1]
                self.values['ILS_MM'] = d[2]
                self.values['ILS_IM'] = d[3]
            i += 1
        return

    @staticmethod
    def __init_values__():
        values = dict()
        values["lat"] = 38.79
        values["lon"] = -9.14
        values["alt"] = 0
        values["vpath"] = 0
        values["hpath"] = 0
        values["GS_knots"] = 0
        values["IAS_knots"] = 0
        values["EAS_knots"] = 0
        values["TAS_knots"] = 0
        values["ILS_OM"] = 0
        values["ILS_MM"] = 0
        values["ILS_IM"] = 0
        return values
```

A.5 TCAS.py

```
import pygame as pg
from pygame import font
from pygame import draw as pgD
from math import pi, sqrt, sin, cos, atan2


```

```
## Definition of some constant values
WIDTH = 910
HEIGHT = 750
SAMPLING_TIME = 8
BACKGROUNG_COLOR = [4, 4, 4]
WHITE = [255, 255, 255]
BLACK = [0, 0, 0]
GREEN = [0, 255, 0]
BLUE = [135, 206, 250]
ORANGE = [255, 165, 0]
YELLOW = [255, 255, 0]
RED = [255,0,0]
GREY = [150, 150, 150]
NM2m = 1852
ft2m = 0.3048
AP_POS = 5 / 6 * HEIGHT * 0.9
D1 = int(1 / 6 * HEIGHT * 2)
D2 = int(2 / 6 * HEIGHT * 2)
D3 = int(3 / 6 * HEIGHT * 2)
D4 = int(4 / 6 * HEIGHT * 2)
MY_ID = 100
RANGE = 20


class Airplane:
    """docstring for Airplane - This class defines an airplane object, based
    on its position, id, TCAS status and vertical velocity"""
    ## tcas_status:
    # 0-> nothing
    # 1-> other aircraft
    # 2-> proximate aircraft
    # 3-> TA
    # 4-> RA

    coord_geo = {}
    coord_ecef = {}
    coord_enu = {}
    coord_disp = {}
    coord_real = {}
    status = 0
    tcas_status = 0
    dist = 0
    relative_alt = 0
    v_z = 0
    checkmove = 0
    id = 0

    def __init__(self, coord_geo):
        self.coord_geo = coord_geo
        self.update_coord_ecef(coord_geo)

    def update_coord_geo(self, new_coord_geo):
        lat = new_coord_geo['Latitude']
        long = new_coord_geo['Longitude']
        alt = new_coord_geo['Altitude']
        self.coord_geo = {'Latitude': lat, 'Longitude': long, 'Altitude': alt}

    def update_coord_ecef(self, coord_g):
        #according to WGS84
```

```
        a = 6378137.0  # meters
        f = 1/298.257223563
        aux = 1-f*(2-f)*sin(coord_g['Latitude'])*sin(coord_g['Latitude'])
        N = a/(sqrt(aux))
        x = (N + coord_g['Altitude'])
        x = x*cos(coord_g['Latitude'])*cos(coord_g['Longitude'])
        y = (N + coord_g['Altitude'])
        y = y*cos(coord_g['Latitude'])*sin(coord_g['Longitude'])
        z = ((1-f)*(1-f)*N + coord_g['Altitude']) * sin(coord_g['Latitude'])
        self.coord_ecef = {'x': x, 'y': y, 'z': z}

    def update_coord_enu(self, coord_geo, coord_ecef):
        c = self.coord_ecef
        lat = 'Latitude'
        lon = 'Longitude'
        x = -sin(coord_geo[lon])*(c['x']-coord_ecef['x'])
        x = x+cos(coord_geo[lon])*(c['y']-coord_ecef['y'])
        y = -sin(coord_geo[lat])*cos(coord_geo[lon])
        y = y*(c['x']-coord_ecef['x'])
        y = y-sin(coord_geo[lat])*sin(coord_geo[lon])*(c['y']-coord_ecef['y'])
        y = y+cos(coord_geo[lat])*(c['z']-coord_ecef['z'])
        z = cos(coord_geo[lat])*cos(coord_geo[lon])
        z = z*(c['x']-coord_ecef['x'])
        z = z+cos(coord_geo[lat])*sin(coord_geo[lon])*(c['y']-coord_ecef['y'])
        z = z+sin(coord_geo[lat])*(c['z']-coord_ecef['z'])
        self.coord_enu = {'x': x, 'y': y, 'z': z}

    def update_display_coord(self, ref_head):
        x_ = cos(ref_head)*self.coord_enu['x']-sin(ref_head)*self.coord_enu['y']
        y_ = sin(ref_head)*self.coord_enu['x']+cos(ref_head)*self.coord_enu['y']
        self.coord_real = {'x': x_, 'y': y_, 'z': self.coord_enu['z']}
        self.dist = sqrt(x_ * x_ + y_ * y_)
        self.relative_alt = self.coord_enu['z']
        theta = atan2(y_, x_)
        r = D4*sqrt(x_*x_+y_*y_)/(20*NM2m)/2
        x_disp = r*cos(theta)
        y_disp = -r*sin(theta)
        dict = {}
        dict['x'] = x_disp + WIDTH/2
        dict['y'] = y_disp + AP_POS
        dict['z'] = self.coord_enu['z']
        self.coord_disp = dict

    def update_coord(self, new_coord_geo, ref_heading, coord_geo_radar={},
    coord_ecef_radar={}, own=False):
        self.update_coord_geo(new_coord_geo)
        self.update_coord_ecef(self.coord_geo.copy())
        if own is False:
            self.update_coord_enu(coord_geo_radar, coord_ecef_radar)
            self.update_display_coord(ref_heading)
        else:
            self.update_coord_enu(self.coord_geo.copy(), self.coord_ecef.copy())
            self.update_display_coord(ref_heading)
        return


class TCAS:

    def __init__(self):
```

```python
        self.Our_Airplane = self.__make_airplane__(0, 0, 0, 0, MY_ID, own=True)
        self.Other_Airplanes = []
        pg.init()
        font.init()
        self.gD = pg.display.set_mode((WIDTH, int(HEIGHT*0.9)))
        pg.display.set_caption('Simulation')
        self.clock = pg.time.Clock()
        self.gD.fill(BACKGROUNG_COLOR)
        pg.display.update()
        self.clock.tick(SAMPLING_TIME)
        self.gameExit = False
        return

    def quit(self):
        pg.quit()

    def new_airplane(self, id, initial_values):
        lat = initial_values['lat']
        long = initial_values['lon']
        alt = initial_values['alt']
        head = 0
        airplane = self.__make_airplane__(lat,long,alt,head,id,
                self.Our_Airplane.coord_geo,self.Our_Airplane.coord_ecef, False)
        self.Other_Airplanes.append(airplane)

    def remove_airplane(self,id):
        for airplane in self.Other_Airplanes:
            if airplane.id == id:
                self.Other_Airplanes.remove(airplane)

    def exited(self):
        for event in pg.event.get():
            if event.type == pg.QUIT:
                self.gameExit = True
                pg.quit()
        return self.gameExit

    def update(self, ref, others):
        new_ref_coord = {}
        my_head = ref['hpath']*pi/180
        new_ref_coord['Latitude'] =  ref['lat'] * pi / 180
        new_ref_coord['Longitude'] = ref['lon'] * pi / 180
        new_ref_coord['Altitude'] = ref['alt'] * 0.3048
        self.Our_Airplane.update_coord(new_ref_coord, my_head, own=True)
        for airplane in self.Other_Airplanes:
            for other in others:
                if airplane.id == other['id']:
                    new_coords = {}
                    new_coords['Latitude'] = other['lat']*pi/180
                    new_coords['Longitude'] = other['lon']*pi/180
                    new_coords['Altitude'] = other['alt']*0.3048
                    airplane.update_coord(new_coords, my_head,
                                          self.Our_Airplane.coord_geo.copy(),
                                          self.Our_Airplane.coord_ecef.copy())
                    airplane.v_z = other['GS_knots']*other['vpath']
        self.gD.fill(BACKGROUNG_COLOR)
        self.__draw_background__()
        self.__draw_ticks__(my_head)
        self.__set_tcas_status__()
```

```
        self.__draw_airplanes__()
        self.__draw_our_airplane__()
        self.__represent_data__(ref['GS_knots'], ref['TAS_knots'], my_head)
        ##  Draw envolving
        pgD.rect(self.gD, GREY, [0, 0, WIDTH, HEIGHT-75], 20)
        pg.display.update()
        self.clock.tick(SAMPLING_TIME)

    def __represent_data__(self,GS,TAS,head):
        ##  Add heading to display
        HDG_str = str(int(head*180/pi)) + '  '
        font_size = int(WIDTH/40)
        f = font.SysFont('Lucida Console', font_size)
        HDG_text = f.render(HDG_str, False, GREEN)

        ##  Add groundspeed, TAS and range on display
        GS_str = 'GS  ' + str(int(GS))
        TAS_str = 'TAS ' + str(int(TAS))
        RNG_str = 'RNG ' + str(int(RANGE)) + 'NM'
        font_size = int(WIDTH/35)
        f = font.SysFont('Lucida Console', font_size)
        GS_text = f.render(GS_str, False, GREEN)
        TAS_text = f.render(TAS_str, False, GREEN)
        RNG_text = f.render(RNG_str, False, GREEN)

        rect = [WIDTH/2-WIDTH*0.07/2, AP_POS-D4/2+10, WIDTH*0.07, font_size+5]
        pgD.rect(self.gD, BLACK, rect, 0)
        pgD.rect(self.gD, WHITE, rect, 5)
        self.gD.blit(GS_text, [0.02*WIDTH, 0.02*WIDTH])
        self.gD.blit(TAS_text, [WIDTH*0.02, 0.02*WIDTH + 1.15*font_size])
        self.gD.blit(RNG_text, [WIDTH - WIDTH*0.16, 0.02*WIDTH])
        self.gD.blit(HDG_text, [WIDTH/2 - WIDTH*0.06/2, AP_POS - D4/2 + 13])

    def __draw_our_airplane__(self):
        a = [WIDTH/2, AP_POS+30]
        b = [WIDTH/2, AP_POS-10]
        pgD.line(self.gD, WHITE, a, b, 3)
        a = [WIDTH/2-10, AP_POS+25]
        b = [WIDTH/2+10, AP_POS+25]
        pgD.line(self.gD, WHITE, a, b, 5)
        a = [WIDTH/2-30, AP_POS]
        b = [WIDTH/2+30, AP_POS]
        pgD.line(self.gD, WHITE, a, b, 10)
        return

    def __draw_background__(self):
        ##  Draw equidistance lines
        self.__draw_dist_circle__(D1/2, 1)
        self.__draw_dist_circle__(D2/2, 1)
        self.__draw_dist_circle__(D3/2, 1)
        self.__draw_dist_circle__(D4/2, 8)
        ## Draw equidistance lines' distance in NM
        self.__draw_dist_numbers__(-70, 5, D1/2)
        self.__draw_dist_numbers__(-80, 10, D2/2)
        self.__draw_dist_numbers__(-83, 15, D3/2)
        self.__draw_dist_numbers__(-84.5, 20, D4/2)
        ## Draw lines of constant heading
        self.__draw_heading_lines__(0, 5)
        for head in [-135, -90, -45, 45, 90, 135]:
```

```
            self.__draw_heading_lines__(head, 1)
        return

    def __draw_heading_lines__(self, angle, line_width):
        angle = angle*pi/180
        point_1 = [WIDTH/2 + D1/2*sin(angle), AP_POS-D1/2*cos(angle)]
        point_2 = [WIDTH/2 + D4/2*sin(angle), AP_POS-D4/2*cos(angle)]
        pgD.line(self.gD, WHITE, point_1, point_2,line_width)
        return

    def __draw_dist_numbers__(self, angle, r, r_display):
        f = font.SysFont('Lucida Console', int(WIDTH/55))
        text_angle = angle*pi/180
        margin = 0.007*WIDTH
        text = f.render(str(r), False, WHITE)
        a = int(WIDTH/2 + r_display*sin(text_angle)+margin)
        b = int(AP_POS - r_display*cos(text_angle))
        self.gD.blit(text, [a, b])
        return

    def __draw_dist_circle__(self, r, line_width):
        loc_x = int(WIDTH/2) - int(r)
        loc_y = AP_POS - r
        arc = (loc_x,loc_y, 2*r, 2*r)
        pgD.arc(self.gD, WHITE, arc, -1.1*pi, pi, line_width)
        return

    def __draw_ticks__(self, head):
        inc = 30*pi/180
        for inc_head in range(-int(pi/inc), int(pi/inc)):
            ## Draw ticks
            a = int(WIDTH/2 + D4/2 *sin(-head+inc_head*inc))
            b = int(AP_POS - D4/2*cos(-head+inc_head*inc))
            c = int(WIDTH/2 + (D4/2 + 20) *sin(-head+inc_head*inc))
            d = int(AP_POS - (D4/2 + 20)*cos(-head+inc_head*inc))
            pgD.line(self.gD, WHITE, [a, b], [c, d], 3)

            display_angle = (inc_head)*30 - 180
            if display_angle > 360:
                display_angle = display_angle - 360
            if display_angle < 0:
                display_angle = display_angle + 360
            ## Display tick number
            disp_n = int(display_angle/10)
            if disp_n == 0 or disp_n == 18 or disp_n == 9 or disp_n == 27:
                f = font.SysFont('Lucida Console', int(WIDTH/30))
                if disp_n == 0:
                    tick_text = f.render('N', False, WHITE)
                elif disp_n == 18:
                    tick_text = f.render('S', False, WHITE)
                elif disp_n == 9:
                    tick_text = f.render('E', False, WHITE)
                else:
                    tick_text = f.render('W', False, WHITE)
            else:
                f = font.SysFont('Lucida Console', int(WIDTH/40))
                tick_text = f.render(str(disp_n), False, WHITE)

            a = int(WIDTH/2 + (D4/2 + 30)*sin(-head+(inc_head-int(pi/inc))*inc))
```

```
            b = int(AP_POS - (D4/2 + 45)*cos(-head+(inc_head-int(pi/inc))*inc))
            vertix = [a, b]
            if vertix[0] < WIDTH/2:
                font_size = WIDTH/45
                a = (D4/2 + 30)*sin(-head+(inc_head-int(pi/inc))*inc)
                a = a + WIDTH/2 - font_size*1.5
                b = AP_POS - (D4/2+45)*cos(-head+(inc_head-int(pi/inc))*inc)
                vertix = [int(a), int(b)]
            self.gD.blit(tick_text, vertix)
        return

    def __draw_airplanes__(self):
        for airplane in self.Other_Airplanes:
            x = int(airplane.coord_disp['x'])
            y = int(airplane.coord_disp['y'])
            rel_alt = int(airplane.relative_alt/0.3048)
            climb_status = self.__set_status__(airplane.v_z)
            if airplane.tcas_status == 0:
                pass
            elif airplane.tcas_status == 1:
                self.__otherAircraft__(x, y)
            elif airplane.tcas_status == 2:
                self.__proximateAircraft__(x, y, climb_status, rel_alt)
            elif airplane.tcas_status == 3:
                self.__drawTA__(x, y, climb_status, rel_alt)
            elif airplane.tcas_status == 4:
                self.__drawRA__(x, y, climb_status, rel_alt)

    @staticmethod
    def __set_status__(v_z):
        if v_z > 0:
            return 2
        elif v_z < 0:
            return 1
        else:
            return 0

    @staticmethod
    def __set_move__(own_id, relative_alt, other_id, status):
        new_checkmove = 'None'
        if status == 4:
            if relative_alt > 0 or (relative_alt == 0 and own_id > other_id):
                new_checkmove = 1
            elif relative_alt < 0 or (relative_alt == 0 and own_id < other_id):
                new_checkmove = 2
        if status == 3:
            new_checkmove = 3
        return new_checkmove

    def __otherAircraft__(self, x, y):
        pgD.polygon(self.gD, BLUE, [[x,y-15], [x-10,y],[x,y+15], [x+10,y]],3)
        return

    def __proximateAircraft__(self, x, y, status, relative_alt):
        pgD.polygon(self.gD, BLUE, [[x,y-15], [x-10,y],[x,y+15], [x+10,y]],0)
        if status==1:
            pgD.line(self.gD, BLUE, [x + 20, y - 5], [x + 20, y + 10], 2)
            pgD.line(self.gD, BLUE, [x + 20, y + 10], [x + 15, y + 5], 2)
            pgD.line(self.gD, BLUE, [x + 20, y + 10], [x + 25, y + 5], 2)
```

```
        elif status==2:
            pgD.line(self.gD, BLUE, [x + 20, y - 5], [x + 20, y + 10], 2)
            pgD.line(self.gD, BLUE, [x + 20, y - 5], [x + 15, y + 0], 2)
            pgD.line(self.gD, BLUE, [x + 20, y - 5], [x + 25, y + 0], 2)
        relative_alt = int(relative_alt/100)
        if abs(relative_alt) > 99:
            feet = str(99)
        else:
            feet = str(abs(relative_alt))

        f = font.SysFont('Lucida Console', 16)
        if abs(relative_alt*100) < 1000:
            feet = '0' + feet
        if relative_alt > 0:
            textsurface = f.render('+' + feet, False, BLUE)
        elif relative_alt < 0:
            textsurface = f.render('-' + feet, False, BLUE)
        else:
            textsurface = f.render(' '+ feet, False, BLUE)
        self.gD.blit(textsurface, (x - 12, y - 35))


    def __set_tcas_status__(self):
        for airplane in self.Other_Airplanes:
            old_status = airplane.tcas_status
            rel_alt = airplane.relative_alt
            Altitude = self.Our_Airplane.coord_geo['Altitude']
            if airplane.dist < 20*NM2m:# and Altitude > 1500*ft2m:
                if airplane.dist < 6*NM2m:
                    if airplane.dist < 3.3*NM2m:
                        if airplane.dist < 2.1*NM2m:
                            if rel_alt >= -600*ft2m and rel_alt <= 600*ft2m:
                                airplane.tcas_status = 4
                            elif rel_alt >= -850*ft2m and rel_alt <= 850*ft2m:
                                airplane.tcas_status = 3
                            else:
                                airplane.tcas_status = 2
                        else:
                            if rel_alt >= -850*ft2m and rel_alt <= 850*ft2m:
                                airplane.tcas_status = 3
                            else:
                                airplane.tcas_status = 2
                    else:
                        airplane.tcas_status = 2
                else:
                    airplane.tcas_status = 1
            else:
                airplane.tcas_status = 0
            status = airplane.tcas_status
            if (status==4 and old_status==3) or (status==3 and old_status==4):
                our_id = self.Our_Airplane.id
                move = self.__set_move__(our_id, rel_alt, airplane.id, status)
                self.__alert__(move)
            if airplane.tcas_status == 3 and old_status == 2:
                self.__alert__(0)
        return

    def __drawRA__(self,x,y,status,relative_alt): #Resolution Advisory
        side = 15
```

```
        vert_x = x - side/2
        vert_y = y + side/2
        pgD.rect(self.gD, RED, pg.Rect(vert_x,  vert_y, side, side))
        if status == 1:
            pgD.line(self.gD, RED, [x + 18, y + 0], [x + 18, y + 22], 2)
            pgD.line(self.gD, RED, [x + 18, y + 22], [x + 13, y + 17], 2)
            pgD.line(self.gD, RED, [x + 18, y + 22], [x + 23, y + 17], 2)
        elif status == 2:
            pgD.line(self.gD, RED, [x + 18, y + 7], [x + 18, y + 22], 2)
            pgD.line(self.gD, RED, [x + 18, y], [x + 13, y + 12], 2)
            pgD.line(self.gD, RED, [x + 18, y], [x + 23, y + 12], 2)
        relative_alt = int(relative_alt/100)
        feet = str(abs(relative_alt))
        f = font.SysFont('Lucida Console', 16)
        if relative_alt < 1000:
            feet = '0' + feet
        if relative_alt > 0:
            textsurface = f.render('+' + feet, False, RED)
        elif relative_alt < 0:
            textsurface = f.render('-' + feet, False, RED)
        else:
            textsurface = f.render(' '+ feet, False, RED)
        self.gD.blit(textsurface, (x - 15, y + 22))


    def __drawTA__(self, x, y, status, relative_alt):
        # Traffic Advisory
        # status=0->levelled
        # status=1->descending
        # status=2->ascending
        pgD.circle(self.gD, YELLOW, [x, y], 10)
        if status==1:
            pgD.line(self.gD, YELLOW, [x + 20, y - 8], [x +20, y + 10], 2)
            pgD.line(self.gD, YELLOW, [x + 20, y + 10], [x +15, y + 5], 2)
            pgD.line(self.gD, YELLOW, [x + 20, y + 10], [x +25, y + 5], 2)
        elif status==2:
            pgD.line(self.gD, YELLOW, [x + 20, y - 8], [x +20, y + 10], 2)
            pgD.line(self.gD, YELLOW, [x + 20, y - 8], [x + 15, y + 0], 2)
            pgD.line(self.gD, YELLOW, [x + 20, y - 8], [x + 25, y + 0], 2)
        relative_alt = int(relative_alt / 100)
        feet = str(abs(relative_alt))
        f = font.SysFont('Lucida Console', 16)
        if relative_alt <1000:
            feet = '0' + feet
        if relative_alt > 0:
            textsurface = f.render('+' + feet, False, YELLOW)
        elif relative_alt < 0:
            textsurface = f.render('-' + feet, False, YELLOW)
        else:
            textsurface = f.render(' ' + feet, False, YELLOW)
        self.gD.blit(textsurface, (x - 13, y + 9))

    @staticmethod
    def __alert__(move):
        list_sounds = []
        list_sounds.append('./../AlertsTCAS/Traffic.mp3')
        list_sounds.append('./../AlertsTCAS/DescendDescend.mp3')
        list_sounds.append('./../AlertsTCAS/ClimbClimb.mp3')
        list_sounds.append('./../AlertsTCAS/ClearOfConflict.mp3')
```

```
        if move != 'None':
            pg.mixer.init(26000)
            pg.mixer.music.load(list_sounds[move])
            pg.mixer.music.play(0)
        return

    @staticmethod
    def __make_airplane__(coord_lat, coord_long, coord_alt, ref_head,
    airplane_id, coord_geo={}, coord_ecef={}, own=False):
        dict = {}
        dict['Latitude'] = coord_lat*pi / 180
        dict['Longitude'] = coord_long*pi / 180
        dict['Altitude'] = coord_alt
        airplane = Airplane(dict)
        airplane.id = airplane_id
        if own is False:
            airplane.update_coord(dict, ref_head, coord_geo, coord_ecef, own)
        else:
            airplane.update_coord(dict, ref_head, own=True)
        return airplane

    def __del__(self):
        pg.quit()
```