Concrete Architecture of FlightGear
CISC 326/322 Assignment 2 Report
March 23rd, 2024

Authors:
Max Lindsay - 19mjl5@queensu.ca
Vincent Proulx - 20vp24@queensu.ca
Mandi Wilby - 19mlw1@queensu.ca
Tyler Swan - 23lpg1@queensu.ca
Isaac Wood - 20ipw@queensu.ca
Vansh Panwar - 19vp8@queensu.ca

Table of Contents

Abstract

This report dissects the core functionality of the open source flying simulator FlightGear at a concrete level, and provides an additional detailed analysis of the artificial intelligence subsystem within FlightGear, unveiling it's architectural style and dependencies.

By using the Understand program to be able to see the various dependencies between components within FlightGear and the AIModel, and having already analysed the software at a conceptual level, we were able to draw much deeper conclusions as to what architectural styles the program implements, and highlight the functionalities of the various components within the AIModel itself. Upon this analysis, we were able to improve our high-level architecture of the system as a whole by discovering the importance of the ATC, Multiplayer, Navaids, Autopilot and Cockpit subsystems, and create an architectural design for the AIModel subsystem based on the 20 key components we found within the model. Having revisualized our architectural designs, we have decided to focus on an object-oriented combined with repository architectural style for both the high-level and AIModel graphs.

Introduction

FlightGear is an open source flight simulation. Due to being freely available to access and modify, it has many contributors all collaborating to enhance the simulation, thus greatly impacting the architecture as the system is constantly changing and evolving.

The first section of this report delves into the concrete architecture of FlightGear, based upon the conceptual architecture report and our findings from the source code. This report covers how the original conceptual architecture was changed from the previous report to its new updated architecture.

Furthermore, this report delves deeper into one of the pivotal aspects of FlightGear previously covered in the conceptual report, specifically the AIModel. The in-depth investigation reveals the fundamental components of the main chosen component, meticulously examining their respective roles with a focus on their interactions and interdependencies. This was accomplished through the use of Understand, a tool used for breaking down and analysing code. It provides many features to aid in code comprehension, however in this case it is used to better understand the interactions and dependencies of AIModel as well as FlightGear overall.

Once there was a good grasp of the concrete architecture of FlightGear, a reflexion analysis was done on the conceptual architecture and the conceptual architecture to examine the convergences and divergences in their structure and behaviour. In a perfect world, the concrete architecture would mirror the conceptual architecture, however as the real world, FlightGear included, is ever-changing and mutable, unsurprisingly this means that there are discrepancies abound.

Finally, we will also cover some use cases. The first use case generates and processes a flight plan where a subcomponent takes input and creates a proper flight plan from it. The second use case covers dynamic flight planning, in which an aircraft reacts to an error in its flight.

Top-level Concrete Architecture

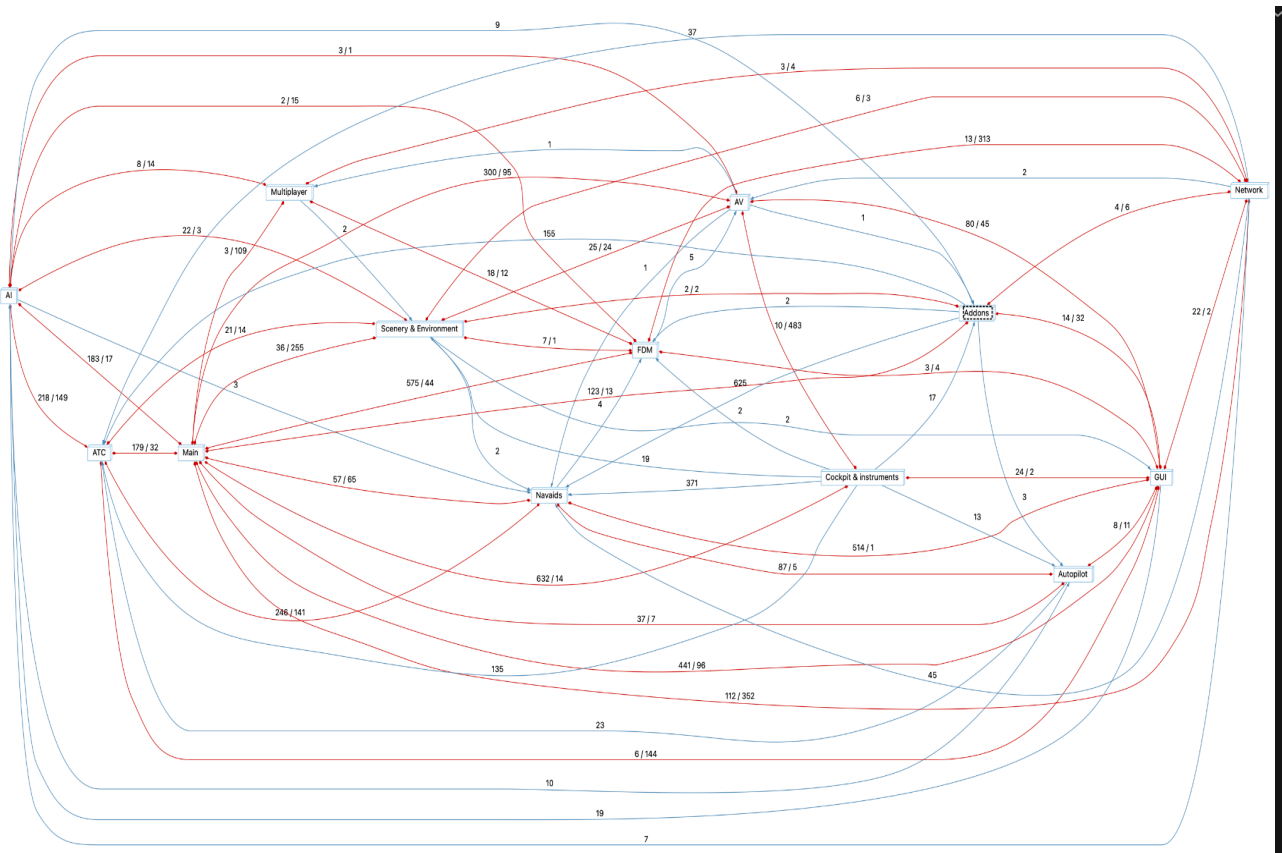
Architecture Style

We believe the concrete Architecture of FG is very similar to what we mentioned in the Conceptual Report and can be split into three main parts. Firstly, in our initial report we discussed pub-sub due to the property tree and its use in establishing global state. However, after looking at the implementation and finding that the property tree isn't as universal in terms of determining the state of the program as initially expected, and doesn't follow certain pub-sub properties like explicit pushing to subscribers (e.g. fdm and networking directly communicate), we came to the conclusion that the property tree better suits a repository style architecture, being a central data source for the subsystems

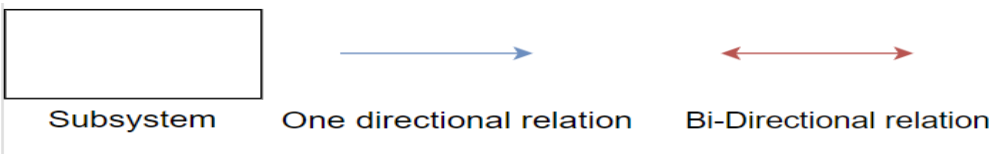
to interact with each other. Second is the Client-Server style which we discussed within the conceptual report in terms of its use within networking and multiplayer usage, what we didn't mention however was its usage within downloading data from servers for singleplayer usage, such as on the fly scenery via TerraSync. This functionality does allow users to not need to download large amounts of data in the initial installation but also means that players must be always online when flying through un-downloaded areas if they wish to have full details. Lastly, it is object oriented which we did not discuss in the previous report, FG makes liberal use of object orientation throughout its subsystems such as AI which we will discuss in depth later. Overall, the main architectural style that the software follows can be noted as a repository style with object oriented subsystems and hints of client-server where needed.

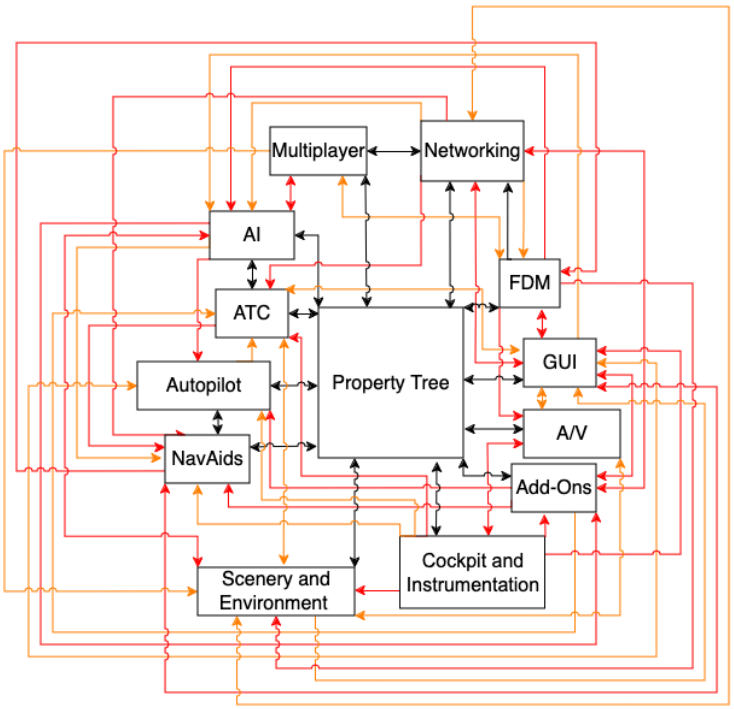
Derivation Process

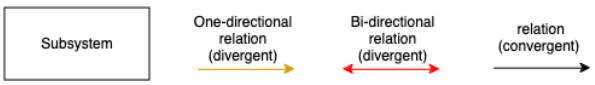
The process we followed in deriving the concrete architecture for flight gear, commonly known as the High-Level Architecture style, involved a few steps. Firstly, we took a look at our conceptual style and noted differences we saw within our concept and those shown in the several resources provided such as the feedback and other reports. These discrepancies and their updates are talked about in further detail within the discrepancies section of this report. After updating our understanding of the concept, we updated our box and arrow diagram. From this point onwards, we needed to find the concrete style. To do so, we first referenced the same resources we used to update our concept and took note of the commonly mentioned styles, those being; Repository, Object-Oriented, and Client-Server. We then used the 'Understand' software to get a better understanding of the source code. After looking at several dependency diagrams for various subsystems, and the source code itself, we were able to come to the conclusion that the system uses all three styles mentioned simultaneously. How each style is used is discussed in the section above. After figuring out the concrete style, we used the 'design architecture' feature within the 'Understand' software to group the several subsystems into the major subsystems we identified earlier when comparing conceptual styles and combined that with the knowledge we gained while we derived the concrete style and the READMEs found in the code base. Below you can find a diagram showing the general dependencies within the major subsystems and a list of the major subsystems and the smaller subsystems within. The dependency diagram is refined in the section below in the form of a box-and-arrow diagram.



- FG arch
  - AI
    - Ai
    - traffic
  - ATC
    - Atc
    - airports
    - radio
  - AV
    - model
    - sound
    - viewer
  - Addons
    - Add-ons
    - Scripting
  - Autopilot
    - autopilot
  - Cockpit & instruments
    - cockpit
    - input
    - instrumentation
  - FDM
    - aircraft
    - fdm
    - systems
  - GUI
    - canvas
    - gui
  - Main
    - main
  - Multiplayer
    - MultiPlayer
  - Navaids
    - Navaids
  - Network
    - network
  - Scenery & Environment
    - environment
    - scenery
    - time







Subsystems and Interactions

Figures 1 & 2 (above): screenshots from Understand, showing the generated dependencies chart and the file folders we used to make subsystems for FG.
Figure 3 (left): the hand made Concrete arch with discrepancies in orange and red, and the original conceptual connections in black.

Below you will find the thirteen main subsystems of FG, each with a brief description and their interactions listed.

- Property Tree / Main - The main subsystem is what serves as the repository. It contains the main routine for FlightGear, together with various utility classes and functions that do not fit in anywhere such as the factories that initially create subsystems. This repository has a bidirectional relation with every subsystem since it is the repository. Other subsystems use this subsystem to access data from each other and write to global variables.
- AIModel - The AIModels subsystem is in charge of simulating other aeroplanes and ground vehicles within the flight simulation environment. In terms of interactions, GUI needs AI positions and source/destination in order to draw the map. Networking depends on AIManager for aircraft types to add to the swift aircraft manager. ATC depends on AI to retrieve information like flight plans and performance data for the ground controller, AI depends on it to receive instructions for the AI aircrafts. AI relies on Autopilot to receive data such as offsets and pitch to model a submodel. FDM depends on AI for its AI aircrafts to calculate their physics and uses performance data as well. Multiplayer calls on AI for AI aircraft data to manage it as well for multiplayer purposes such as callsigns and paths, AI depends on it for external motion info to factor it in for its own usage. AI uses Addons for its Nasal scripts to check the possible scripts and can load new modded AI scenarios via Addons. It also uses Navaids for its records on runways. Environment and Scenery depends on AI to receive information for the nearest AICarrier location, AI Depends on it for things such as gravity and environment manager to create flight plans and other AI models.
- ATC - Simulated air traffic control instructions to the pilot during the flight. Autopilot depends on ATC for route management. Cockpit and Instrumentation depends on ATC for navigation display tools for things such as airports, runways, and GPS. Networking relies on ATC for airport information and to create assets such as runway polygons and airport geometry. GUI relies on ATC for a lot of airport functions in order to find details of it for map, ATC depends on GUI to make dialogue windows for ATC functions. NavAids relies on ATC for components to create flight plans and scripts, ATC relies on NavAids for its positional data and navigation records for its communication station. Environment and Scenery depends on ATC for airport data, ATC relies on it for environment variables such as weather for runway management. Add-ons depend on ATC for its flight planning via Nasal similarly to Navaids.
- AV - The audio subsystem is responsible for managing all sound effects heard during flight simulation. Things like engine noise, environmental sounds like wind and warnings. The viewer subsystem is responsible for images. It takes data from the simulated world, including aircraft location and environment, and converts it into 3D graphics that can be seen on the screen. FDM relies on AV for event handling and replays. Cockpit and Instrumentation relies on AV for rendering and operating functions such as radio and notifications, AV relies on cockpit and instrumentation for camera movements and updating/rendering based on inputs. GUI relies on AV for things such as message boxes and cursor positions and to let it know there's a snapshot ready to use, AV relies on GUI for handling certain macros and updating the GUI. Environment and Scenery works very closely with AI, using each other to update the environment around the player and making sure to display and output the correct visuals and sounds.
- Add-ons - The add-ons subsystem allows users to install additional aircraft, scenery, and features, as well as handling the Nasal scripting language of which all add-ons are written in. Add-ons relies on autopilot for the nasal flight planner, GUI for access to the pc clipboard as well as nasal access to canvas, navaids for the nasal flight planner similar to autopilot, Networks to allow http access to nasal. It is depended on by cockpit and instrumentation for taking in inputs

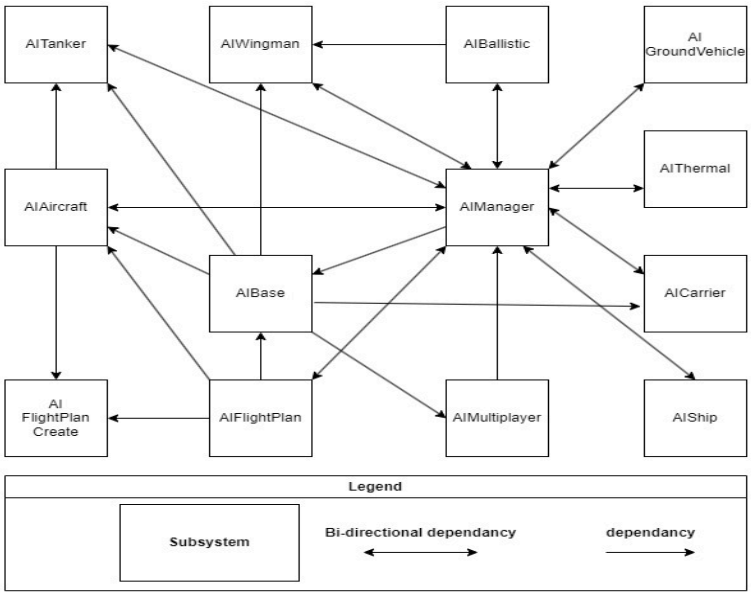
via nasal, GUI for allowing new add-on elements to the GUI, and networking for an intermediary layer between networking and ATC via nasal.

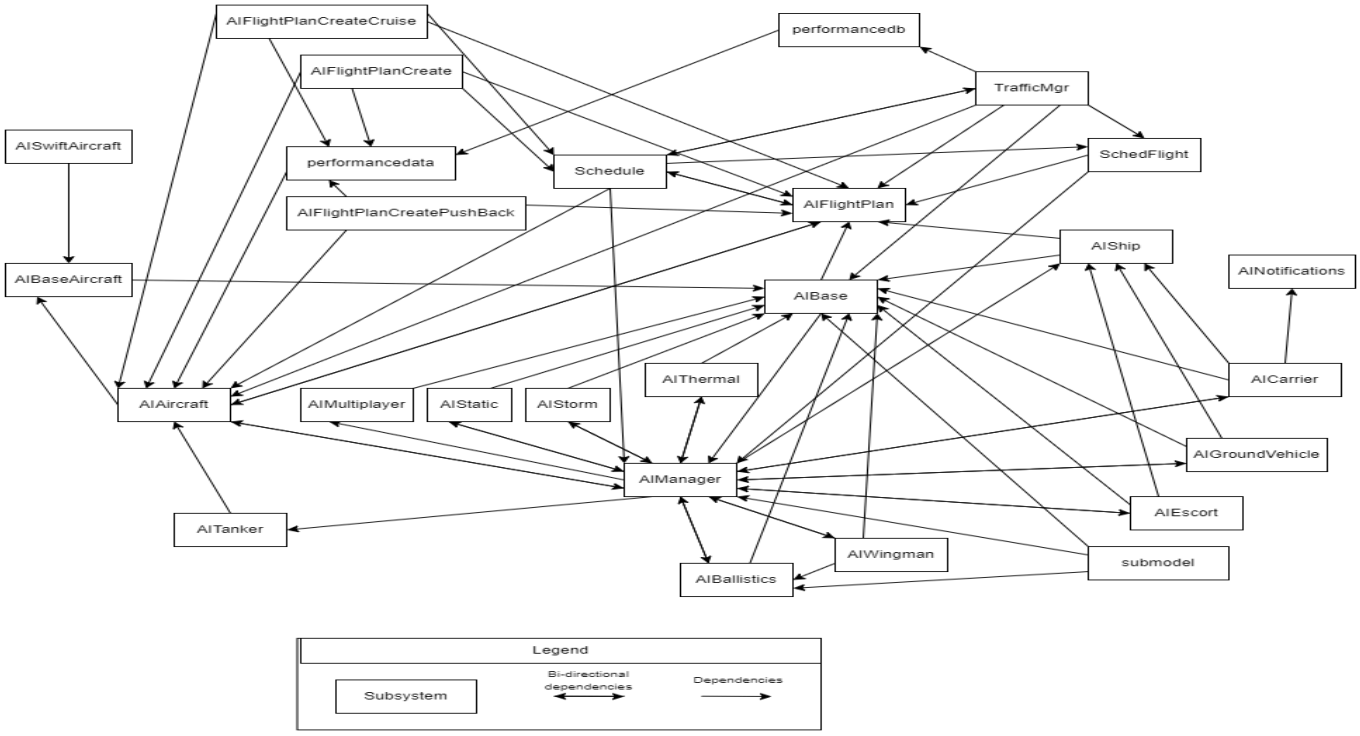
- Autopilot - The autopilot subsystem is responsible for automatically controlling the aircraft's heading, altitude, and airspeed based on your input. Autopilot depends on GUI to give confirmation windows to the player, and Navaids to provide the many parts of the flight plan such as heading and setting speed. It is depended on by Cockpit and Instrumentation to process flight plans for the gps and displaying navigation, GUI to populate the map with the flight plan, and navaids to put entire flight plans together.
- Cockpit and Instrumentation - The cockpit and instrumentation subsystem replicates the pilot's environment. It displays essential instruments like altitude gauges and airspeed indicators. GUI handles mouse movements via instrumentation. Cockpit and Instrumentation relies on NavAids for a lot of its instrumentation updating such as flight plans, waypoints, routes etc. Cockpit and Instrumentation relies on Environment and Scenery for certain things such as mach calculation, atmosphere and weather to update the instrumentation.
- FDM - Flightgear FDM subsystem is the core component of aircraft motion calculations. It simulates how the aircraft reacts to air forces, controls and external factors. The FDM depends on GUI to provide an error box to the player if an FDM error occurs, Multiplayer to record other players for the flight recorder when necessary, Networking for support in using external systems for the FDM, and Environment and scenery for calculations with regards to the external environment. It is depended on by GUI to include flight history on the map, Multiplayer to make calculations before sending data to the mp server, Navaids to find what constraints there are while making flight plans such as the max speed of the current plane, and networking, again for support in using external systems for the FDM.
- GUI - The GUI subsystem utilises the Nasal scripting language and the Canvas subsystem to allow for the creation of on-screen elements like menus and panels. It depends on Navaids in order to get waypoints and flight plans in order to draw the map, show lists of waypoints etc, and networking in order to gain http access to update things such as the launcher and find updates. It is depended on by navaids to write a message box to the player in case of error, networking for screenshot support via http, and Scenery and environment for access to the mouse.
- Multiplayer - The multiplayer subsystem allows you to connect to a server and fly with other players. It depends on networking for resolving the dns of multiplayer servers, and scenery and environment for access to time. It is depended on by networking so it can change multiplayer attributes when new packets arrive.
- NavAids - The Navaids subsystem works as a virtual navigation system. It simulates ground navigation equipment, providing guidance and information to help track position and follow flight path during flight. It is depended on by networking to provide it with information to the navigation database.
- Networking - The networking subsystem facilitates communication between multiple instances of the software. It enables multiplayer functionality, allows connections to external tools and data sources, and extends simulation functionality. It relies on scenery and environment for gravity information to support network attached simulation hardware and other information for a network attached fdm. It is also dependent on scenery and environment for http access in order to gain real time weather information.
- Scenery and Environment - The environment and scenery subsystem manages the loading and rendering of landscapes, buildings, and weather effects.

AIModel

Inner Architecture Style

To derive the inner architecture style of AIModel, we first made a conceptual architecture using a list of the files included and reading up on their uses and purposes found via the wiki, the source code and the READMEs. To the right is the conceptual architecture we derived for the AIModel subsystem. We then combed through the source code using the 'Understand' software previously mentioned. Using the dependency feature, we sorted all dependencies and filtered things such as false includes and useless calls to derive the concrete structure. The diagram for the concrete structure can be seen below.





Inner Architecture Interactions

AIAircraft

This file defines a class, FGAIAircraft, which inherits from FGAIBaseAircraft; it's responsible for creating and controlling the AI aircraft within FG. The class functions include things like initialising the aircraft, setting its performance characteristics, handling its flight plan, updating its state (such as position, speed, altitude, etc), and responding to ATC instructions. It mainly accesses AIFlightPlan, and AIManager.

AIBallistics

AIBallistics deals with AI projectiles within FlightGear. The main course of action is to create a ballistic object. The idea is that projectiles, like tank ammunition, can be abstracted into a ballistic object. The class methods for this object include things like handling the impact of the object, calculating the

effects of force that the object has, and anything relating to how an object interacts with the environment. AIBallistics accesses the gravity part of the environment file and most of the main files.

AIBase

Base AI class for flight gear that deals with managing AI objects and functionalities throughout the entire AI subsystem. It governs the various AI elements that are made by the AI systems. Some of the key functionalities include governing positioning and properties for the several objects, creating parent-child relationships for the objects, managing flight plans and adjusting model detail based on distance from object. A part of the file is also responsible for calculating flight dynamics, specifically the Mach number which is the ratio of the object's speed to the speed of sound. In terms of interactions within the subsystem, since it is the base file, it interacts with almost every single file within the subsystem.

AICarrier

This file inherits from AIShip since its main use is creating aircraft carrier ships objects with the AICarrier class. In addition to all the same attributes and functionalities as the AIShip objects, AICarrier objects have deck position, deck height and wind attributes and methods that turn to accommodate planes landing and taking off of the object.

AIEscort

This file inherits from AIShip to create escort ships objects that follow a parent ship, called the escort's station, during an AI scenario. The ship is generated and bound to a model with the use of functions belonging to AIShip. It then uses the setParent() function to bind itself to a parent AIShip model so that it can move at the same speed and share bearings with its parent ship.

AIFlightPlan

In FlightGear, a flight plan is a predefined route that an aircraft intends to fly. It outlines the journey from departure to destination, including waypoints, airways, altitudes, along with other details necessary for the flight.

The core of this document is filled by the FGAIFlightPlan class, which is responsible for creating, loading and executing AI aircraft flight plans in the FlightGear virtual environment. It provides a way to make detailed and dynamic flight plans that could include such things as waypoints; airways; speeds and altitudes that should be followed by AI aircraft from takeoff through landing. It also supports reading of flight plans predefined in files and also dynamically creates flight plans based on different parameters like aircraft performance, airports of origin and destination, and routes meant not be travelled.

AIFlightPlanCreate

The implementation interacts with various parts of the flight simulator, like airports (FGAirport), runways (FGRunway), and aircraft (FGAIAircraft), to gather necessary data for flight planning. It includes functions for generating taxi, takeoff, climb, cruise, descent, approach, landing, and taxi to parking phases. Each of these phases involves calculating waypoints, speeds, and altitudes appropriate for the aircraft's performance and the route's specifics. It handles dynamic creation of flight plans, meaning it calculates the flight path based on the given parameters, and environmental conditions.It accesses AIAircraft, Schedule, and the airport and environment components like dynamics, runways and airport. It accesses AIFlightPlan and AIAircraft, along with other global variables through other files.

AIFlightPlanCreateCruise

The AIFlightPlanCreateCruise part seems to mostly focus on one function which is the CreateCruise function. Without executing dynamic route computation, the FlightGear createCruise function statically sets a cruising waypoint at a given latitude, longitude, altitude, and speed to begin the cruise phase of an AI aircraft's flight plan. The active runway for arrival is then determined by the kind of flight and the direction of the aircraft, and waypoints are set up to initiate the descent phase at the runway's threshold as well as 10 kilometres beforehand. To provide a seamless transition from cruise to descent into the airport, these waypoints are included to the flight plan. The function's design emphasises a simple approach to FlightGear's flight planning, focusing on the establishment of crucial flight phases without requiring real-time adjustments. This guarantees that the AI aircraft proceeds logically from cruise to landing preparation. It accesses AIFlightPlan, AiAircraft, and scheduling for integrating the cruise phase within the broader AI traffic schedule.

AIGroundVehicle

Makes the FGAIGroundVehicle class which is responsible for AI controlled ground vehicle simulation such as cars, trucks, tows etc. It includes methods for initialising the vehicle, updating its state over time, and interacting with the simulation environment. The RunGroundVehicle method is the main function responsible for controlling the behaviour of the ground vehicle based on several factors such as its surroundings or a parent object such as a tow truck. In terms of interactions; it is called by AIManager when initialising a new instance of the object, it calls the base file for obvious reasons, references AIFlightPlan and AIShip to get several data values made by those files such as altitude and acceleration.

AIManager

The AIManager file defines an AIManager class that is in charge of managing everything to do with AI scenarios. To accomplish this, it interacts with every file that defines a class for use in an AI scenario. This file has multiple functions for registering, loading and unloading scenarios that are set to or have finished playing out. Some of these functions take a scenario name and find a file while others have files as parameters. Once a scenario is loaded, the AI objects that are necessary for the scenario must be created, the scenario must run, and then the objects must be discarded. This is also done by an AIManager object. The creation of objects for AI scenarios, the binding of these objects to models, the updating of the objects' states in real time, and the deletion of the objects are all accomplished with one or more dedicated methods.

AIMultiplayer

AIMultiplayer is a file that adjusts the speed at which the AI subsystem generates its calculations based on the amount of lag the user is facing when playing multiplayer. The subsystem compares the timestamps of the most recent network packet the user has received with the previous packet they have received to update the flight position, orientation and velocity accordingly with the network transmission rate. AIMultiplayer receives data from the multiplaymgr file outside of the AIModel subsystem and passes the information it creates to the AIBase file.

AIShip

This file defines a class named FGAIShip which represents a ship used in an AI scenario based on the FGAIBase class from the AIBase files. It inherits the readFromScenario method from the FGAIBase class in order to initialise the ship's attributes according to the scenario's requirements and to

attach a model to the FGAIShip object. The object's attributes are then bonded to the model by another method. This file also has methods to set these objects on the course plotted by the scenario. The run method for running the model through the scripted scenario is also in this file along with the method that processes the flight plan which, once more, utilises FGAIBase to do so. This file is also a base file from which the AICarrier and AIEscort files inherit the ship class.

Submodel

The submodel file creates usable submodels for the AIBallistic, AIManager, and AIBase classes encapsulating all elements they need from the Model subsystem combined with AI specific data sets in object-oriented form. Its main purpose is to streamline inheritance through the AI subsystem. Submodel receives information from the subsystemFactory file outside of the AIModel, and passes the objects it creates to AIBallistic, AIBase and AIManager.

AITanker

Defines a class called FGAITanker which represents an AI-controlled tanker aircraft. This class inherits from FGAIAircraft sharing common functionalities with other AI-controlled aircraft. The purpose of the tanker aircraft is refuelling operations. The class includes methods for reading data from a scenario, binding variables, setting the TACAN channel ID, and running the tanker's behaviour over time. The main ‘Run’ and ‘Update’ methods are used to perform calculations related to the tanker's behaviour, update its state and then perform transformations. These methods ensure that the tanker behaves according to its defined logic within the simulation. It is called by AIManager when initialising a new instance of the object, it calls the base file for obvious reasons, and references the AIAircraft file since it inherits most of its properties from it due to it being an aircraft as well.

AIThermal

The AIThermal component is responsible for creating all thermal related calculations within the AI subsystem to use in its flight trajectory calculations. Thermal calculations in aerodynamics seek to measure how the changes in temperature impact a plane’s flight lift, as hotter air rises above cooler air. Factors such as wind speed and altitude are what determine the significance of the thermal variance in a flight, and the AIThermal subsystem uses these variables and many more to provide these calculations to the rest of the AI model. AIThermal receives and passes information back and forth with the AIManager and AIBase subsystems.

AISwiftAircraft

AISwiftAircraft is responsible for allowing compatibility with the third-party flight simulation network Swift in the AI subsystem. Swift is a network that allows connection to FSD (Flight Simulation Daemon) servers, which is used for privatised air traffic control communication across multiple flight simulator platforms such as Microsoft Flight Simulator and X-Plane. AISwiftAircraft provides all information Swift would need to incorporate the user's flight into their network, such as flight position and speed. AISwiftAircraft does not call information from any other classes, but it passes the objects it creates to AIBase and AIBaseAircraft.

AIWingman

A wingman is a plane, or more than one plane, that flies in formation with the original plane. The wingman can be in formation to the player’s aircraft, meaning it can maintain a specific relative position and orientation to that aircraft. This includes maintaining formation during various manoeuvres, breaking

away from the formation, or rejoining after a separation. There are initial states that the wingman is set to. The class also updates the wingman's state, including position, heading, pitch, roll, and speed, based on its current orders (formate, break, or join) and the state of the aircraft it is forming to; this requires calculations to ensure that the wingman adjusts its flight path and speed.

SchedFlight

This file defines a class called ScheduledFlight which is used by FlightGear's Traffic Manager's schedule objects to store arrival and departure information, as well as some additional info such as how long it takes for the scheduled flight to 'cool down' and be allowed to be used again by the traffic manager. There are multiple of these objects within schedule, which are then within the traffic manager for the traffic manager to check which flight is to occur for each ai plane, and decide on which of these schedules to use if none are currently being used. It relies on airports in order to get the airport data for arrival/departure, and the property tree for getting the current time.

Schedule

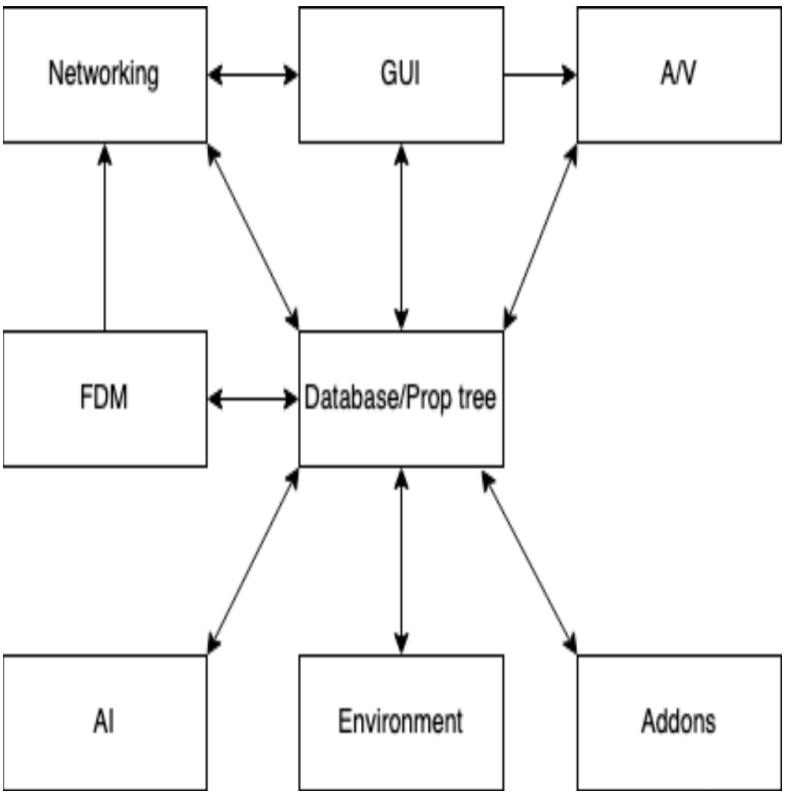
Defines a class called schedule, which contains multiple scheduled flights for the traffic manager to use. Each ai aircraft has a schedule with multiple flight plans contained within it. The point of the traffic part of AI is to be able to check where ai aircrafts are without having to model them within the AIaircraft class when they are too far away for the player to be able to see them, and schedule contains a function to replace the traffic object with an equivalent AIAircraft in its place once the player is close enough. The main two things relied upon are obviously AIAircraft to create the objects, as well as SchedFlight in order to populate the schedules.

TrafficMgr

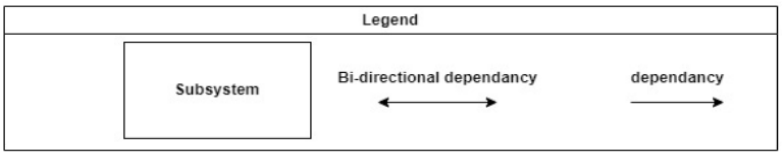
Three classes are defined in this, the parse thread, traffic manager, and heuristics.
The parse thread class spins off a new thread to read schedule data from an external text file, and populate a database of schedules with scheduled flights within them. This data is then used by the traffic manager class which is the main part of this file. This class primarily deals with choosing what flight plans the scheduled traffic uses, and using the heuristics class to determine the approximate position of each AI aircraft in its database. It relies primarily on the schedule classes as that's what the database is populated with, and through them the AIAircraft.

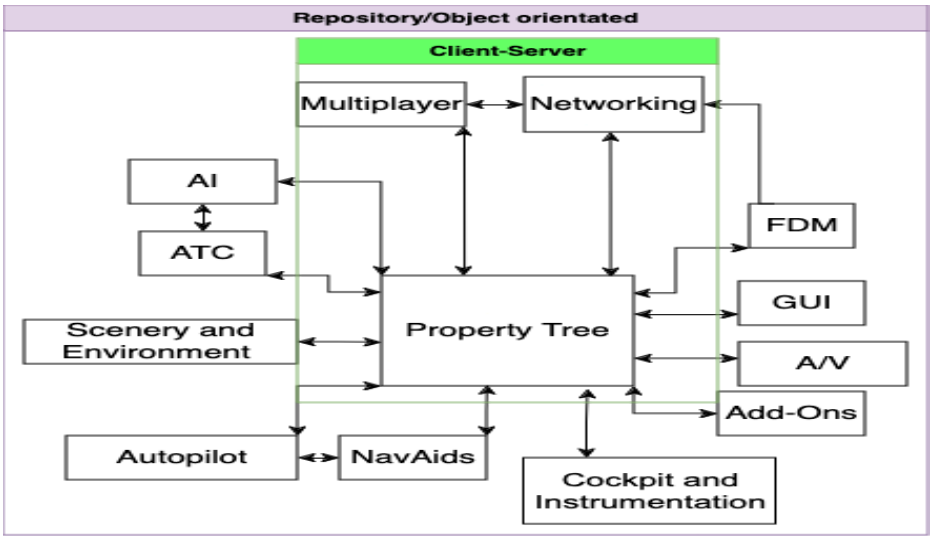
Discrepancies in Architecture

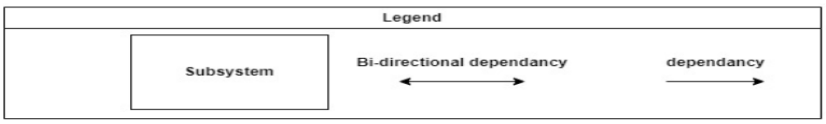
Top-level Architecture



The first diagram on the right was our original conceptual architecture; it has many errors and oversights. After looking at the feedback, several reports from our peers and the given structure analysis for A2, we made our new conceptual architecture below. Having an accurate conceptual architecture is important for many reasons, mainly to ensure that discrepancies in the concrete architecture can properly be addressed. Some changes include the following. Firstly, changing the initial conceptual architecture from pub-sub, while it should be more of a repository style, due to the Property Tree being a central data source rather than exhibiting pub-sub properties. It serves as a common data source for subsystems to interact with each other, enabling communication and coordination within the system. We originally grouped multiplayer and networking together instead of making them distinct from one another, with dependencies. This was an oversimplification. While multiplayer and networking work intertwined, their structure is very much different. Lastly, in our originally conceived architecture, AI encompassed NavAids and Autopilot. NavAids and Autopilot, while conceptually similar, accomplishes different things than AIModel. Additionally, our old model also included ATC within the AI since we thought since it had an “AI” element to it, that it was under AI. ATC is its own component, since it’s such a large part of FlightGear.







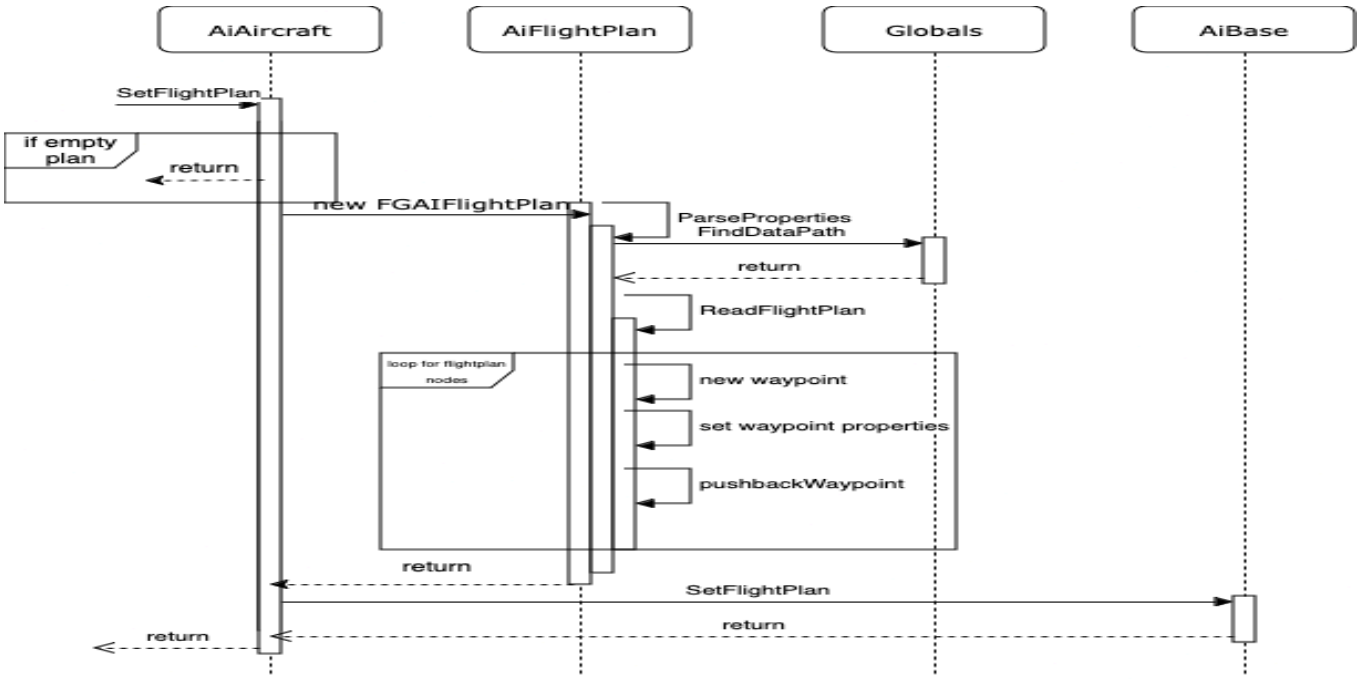
Now comparing this conceptual structure to the concrete structure displayed in the box-arrow diagram in the ‘subsystems and interactions’ section, there are several discrepancies that are seen. The main one is the addition of the Cockpit and Instrumentation subsystem. This subsystem was overlooked during our conceptual architecture planning due to the reason that we thought it is a much smaller part of the system. However, after looking at the code base we realised it is a big component of FG and needs its own subsystem. The rest of the divergences seen are dependencies that emerged during the implementation of the code. There are several divergences and it would take too much space to explain each and every single one of them, however, they all share the exact same rationale. Within a repository style system, all subsystems are supposed to be decoupled. There should not be any communication within systems directly, only through the repository. However, due to the nature of an open source project, developers are not restricted to certain coding ethics. Many of the developers did not follow the common pattern of using the main system as a repository, and instead would directly call other subsystems, causing dependencies to emerge. That is why our concrete diagram has several overlaps and dependencies where they shouldn't exist.

AIModel Inner Architecture

The top and bottom diagrams at the start of the AIModel section are the conceptual and concrete architectures respectively of the AIModel module. As both diagrams are quite different, it is clear that either the conceptual architecture was wrong, or the concrete architecture made deviations from the original plan for this module, or a bit of both. Here, we argue that the third option is the correct one in the case of AIModel. The conceptual architecture was missing a few components that are listed in the above section that goes into depth about each component of the module. The following components were the listed ones missed: AIEscort, AIFlightPlanCreateCruise, submodel, AISwiftAircraft, Schedule, SchedFlight, TrafficMgr, as well as a few more that are not listed above: AIFlightPlanCreatePushBack, AIStorm, AIStatic, AIBaseAircraft, AINotifications, performancedata, performancedb. These last few files are not very big nor do they interact with many other files (so we did not deem it necessary to have sections dedicated to them), but they are present nonetheless. That said, we could not find their interdependencies or relevance from the documentation, so they were not included in the conceptual architecture. Even though the concrete diagram indicates that this module has many Object Oriented and repository style interactions between components, which leads us to believe that the overall architecture of this module is a mix of those two styles, the huge collaborative open source nature of the project has led to significant architectural erosion, making it tough to grasp the style at first glance. Many components have dependencies that do not seem necessary or that could seemingly be fulfilled by other components, such as the AIBaseAircraft component whose sole purpose is to create a base class for Swift aeroplanes. This could have easily been accomplished using the AIAircraft component. Also, since the project is collaborative and open source, the documentation of any changes made to the source code is far and few between. This makes it tough to tell which deviations from the documentation were done for valid reasons and which were not.

Use Cases

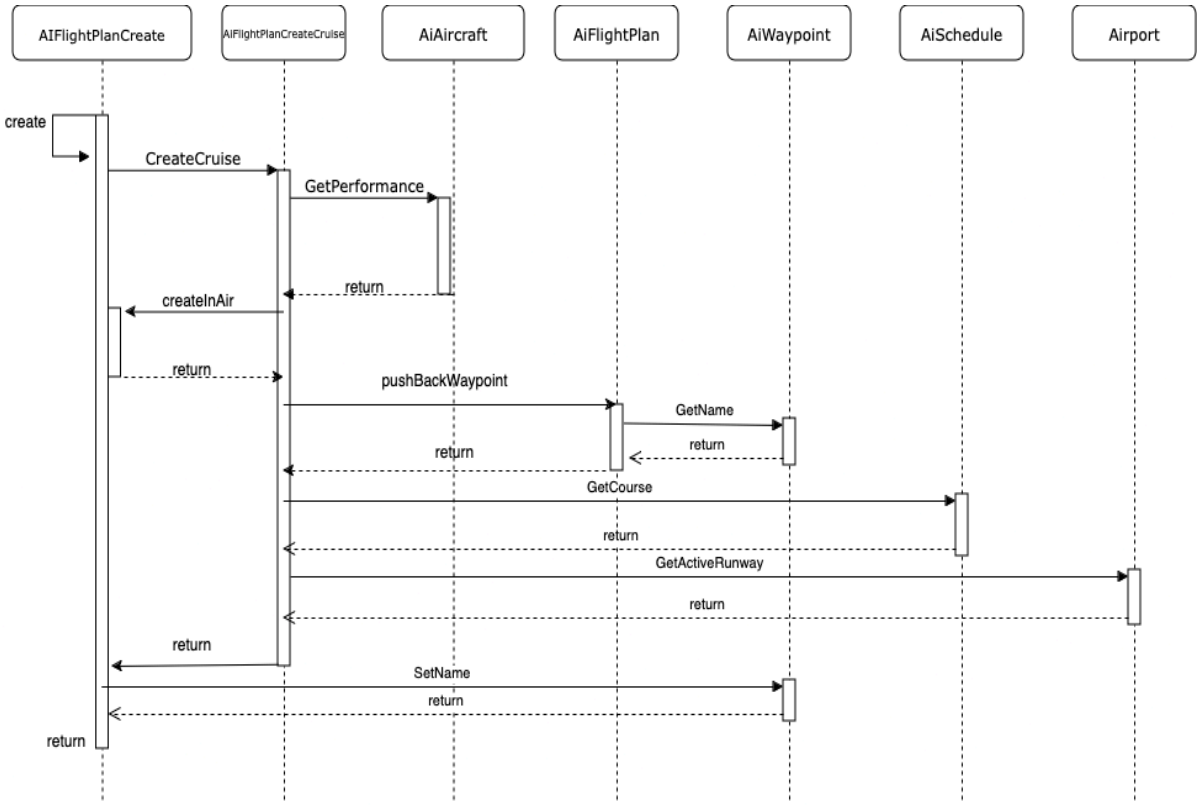
Case 1 - AI Flight Plan Generation and Processing



This sequence diagram explains the process steps for initiating and executing a flight plan in an aircraft automation system. During initialization, the aircraft initialises a new FGAIFlightPlan object using the entered flight plan information. The AIFlightPlan module then analyses and interprets the received data, extracts relevant features and organises it into a flight plan. After this analysis phase, a flight plan is prepared to be executed and finally it is set in the base AI module. This method allows the aircraft to perform planned and interceptive manoeuvres according to a specified plan.

Case 2 - Dynamic Flight Planning

In the following sequence diagram, an AI aircraft comes across an error in the cruising leg of its flight. It thus has to dynamically create one according to the performance aspects of the plane, and where its destination lies. Then sets its current waypoint to the calculated cruising leg



Data Dictionary and Naming Conventions

ATC: Air-Traffic Control
FG: FlightGear
FDM: Flight Dynamics Model
A/V: Audio/Visual
TACAN: TACtical Air Navigation
GUI: Graphical User Interface
AI: Artificial Intelligence
NavAids: Navigation Aids

Conclusions

In conclusion, this detailed examination of FlightGear's concrete architecture has provided valuable insights into its complex structure, which highlights the interactions between its subsystems and architectural styles. Our analysis of the source code revealed that its concrete architecture is similar to what we had in our report for the conceptual architecture, with a few changes due to our newfound understanding. FlightGear is a combination of multiple architecture styles in order to fit the demands of the flight simulation. As we thought, client-server is still an aspect of the architecture, along with object oriented, however with a repository style instead of pub-sub. The in-depth exploration of the AIModel subsystem revealed a richly interconnected design, facilitating dynamic and realistic AI behaviour within the simulation. It incorporates the many components of AIModel to make decisions, communicate and adapt to its parameters for flexibility and to better reflect the natural world. Our reflection analysis shows that FlightGear is open source and that there are many divergences between the conceptual and concrete architecture, none that can easily be changed.

Lessons Learned

There are many valuable lessons to be learned from this assignment, both with regards to better understanding how to conceptualise architectural styles as well as how our team needs to improve to work more cohesively. Firstly, our group was able to see how effective a dependency graph software such as Understand can be for understanding how a system is designed, and we were all able to gain quality experience using this software we can carry forward. We also came to the realisation during this assignment that it is far more ideal to fully understand the architectural style of a system before creating architecture designs for it, opposed to creating the architectural designs first and then try to justify an architecture that best fits your creation. With regards to lessons learned about team functionality, our team still has a lot left to be desired when it comes to scheduling completion of tasks and communication amongst team members. Our meetings, while very important for discussing grey areas in the assignment, would have been more useful as work periods, where we collectively worked on various portions of the assignment opposed to discussing our updates of our individual work since the last meeting. Further, our team needs to get much better at communication when faced with external conflicts, as dependencies from other team members' work was left at a stand still for far too long without any explanation for the delay, which could have resulted in someone picking up the depended on work, moving everyone else forward. Overall, this assignment was very educational for both technical and interpersonal development.

References

Conceptual Architecture in FlightGear