FlightGear's Concrete Architecture

Group 4, Chicken AI

CISC 322/326

Instructor: Amir Ebrahimi

March 25, 2024

Abbey Cameron - 19aec6@queensu.ca
Akash Singh - 20as166@queensu.ca
Derek Youngman - 20day1@queensu.ca
Marion Anglin - 20mma2@queensu.ca
Shrinidhi Thatahngudi Sampath Krishnan - 21stsk@queensu.ca
Ximing Yu - 20xy@queensu.ca

Abstract

This paper investigates the software of FlightGear - a flight simulator - in order to derive a proposed concrete architecture. FlightGear is a unique flight simulator as it is open source, encouraging evolvability and modifiability, and also contains a multiplayer feature.

This report will delve into the various components and subsystems of FlightGear's architecture, expressing the reasoning behind the choices for the concrete architecture found. Comparisons between the concrete architecture and the previously proposed conceptual architecture by the same team will also be featured to showcase the similarities and differences between the two. The concrete architecture was derived from investigating and breaking up FlightGear's source code, building off the previously proposed conceptual architecture to form a concrete architecture. The lessons learned during the time spent on this project are also discussed.

Finally, after the derivation of the concrete architecture from investigating the source code, the architecture styles proposed in this report are as follows: repository style, client-server style, and object-oriented.

Introduction & Overview

FlightGear is a flight simulator that is used for pilot training and entertainment purposes. This simulator provides a practical flying experience through various features like authentic flight controls, accurate world view, and weather, as well as realistic flight aerodynamics. Not only this, but FlightGear also supports a multiplayer feature, making it appealing to those who use FlightGear as a video game for entertainment [1]. Another feature that makes FlightGear unique is its open-source software. This encourages the contribution of participants to evolve and adapt FlightGear as a software [2].

This paper will investigate FlightGear's concrete architecture. Previously, this team derived and proposed a conceptual architecture for FlightGear based on various reports and documents for flight simulators. This conceptual architecture has been used, with slight modification, as a reference when deriving the concrete architecture of FlightGear. Not only will the conceptual architecture help the team investigate and conclude a concrete architecture, but it will also allow the comparison of both kinds of architecture. This report will display the similarities and differences found between the conceptual and concrete architecture of FlightGear.

To derive such a concrete architecture, the team used FlightGear's code and ran it through a software called Understand. This allowed the team to analyze each aspect of the code and combine elements to create the proposed concrete architecture expressed in this report.

This paper will express the changes from the conceptual to the concrete architecture, the various subsystems and components present in this architecture, and different use cases to explain the usage of the concrete architecture. Finally, this report will discuss how the team derived the concrete architecture described, as well as various lessons learned during the process.

Derivation Process

To derive the proposed concrete architecture of FlightGear, the team used a software tool called Understand. Understand allows users to import software projects and investigate and analyze the source code. The team was able to import FlightGear's source code into Understand and begin to get an overarching idea of what the software was made of, and how it was divided into components.

From here, the team used the proposed conceptual architecture found in the previous report to compare with what was being seen in Understand. The components in FlightGear's source code were then broken down and split up into various subsystems, some of which fit with the proposed conceptual architecture, and some which contrasted with it. This was done by looking at the grouping of subsystems in the conceptual architecture, and slowly building similar groupings in the concrete architecture. This allowed the team to rethink some components and where they should be placed in the system. In addition, Understand allowed the team to see what components had not yet been placed in the concrete architecture, so the team was able to analyze these components and find their proper place. Once each component was identified, the Understand tool provided information about the dependencies between each component. Figure 1 presents the raw concrete architecture derived from Understand. The architecture is presented later in a more readable format.

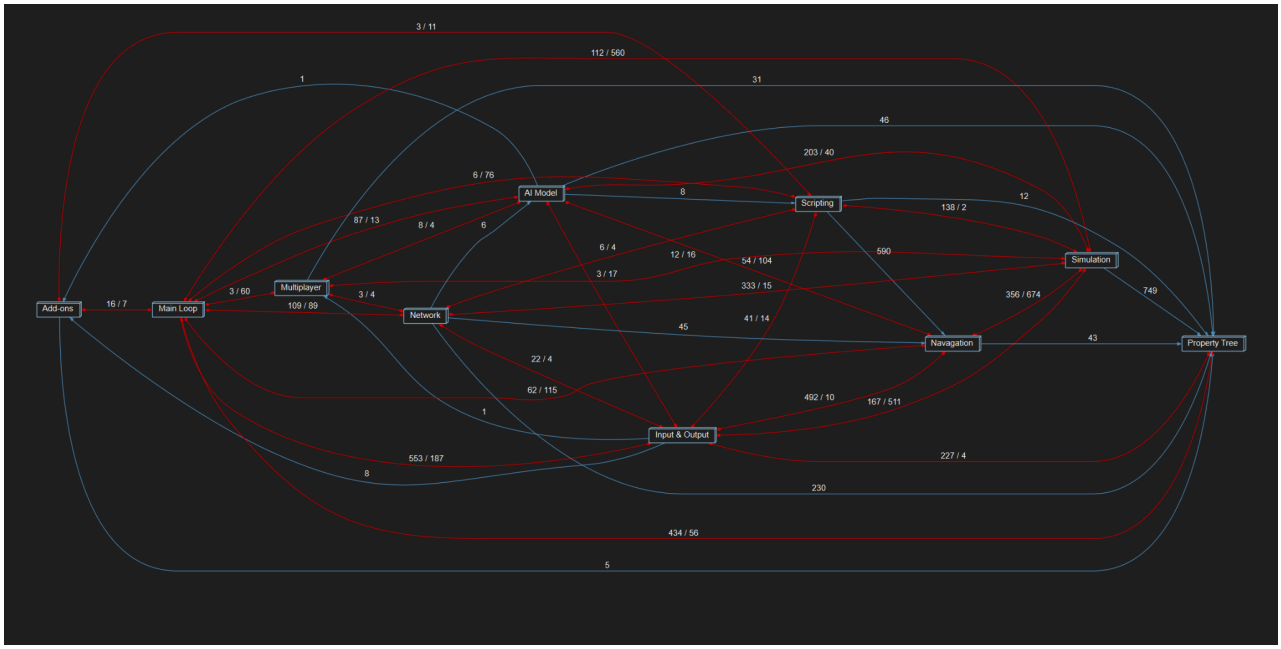

Figure 1: Raw Concrete Architecture from Understand

Previous Conceptual Architecture

The team's previous report derived FlightGear's conceptual architecture. One change to this conceptual architecture is presented here, the addition of a main loop component. The main loop initializes FlightGear and manages the control flow. It's a single-threaded process that periodically passes control to other components as needed [4]. In the updated conceptual architecture, every single component depends on and is depended on by the main loop.

The main loop is reasonably well documented [4], its exclusion from the initial conceptual architecture was simply an oversight. The original conceptual architecture never specified how FlightGear would be initialized and how each of the subcomponents would be initially called upon. Something of this nature would be an essential part of practically any program. The oversight was largely because the property tree was labeled early on in the derivation of the conceptual architecture as the primary component of FlightGear, so it didn't seem that an additional central component was missing, even though the property tree really only acts as a database.

The single-threaded nature of the main loop also contradicts research cited in the previous report which specifies the need for a multi-threaded system in flight simulation. FlightGear could benefit from a more multi-threaded approach, but it seems that the architects did not originally plan this.

Without examining the code, it seems that the main loop depends on and is depended on by every single component since it controls the entire system. This update is indicated in Figure 2.

While the main loop is not the only component present in this report's concrete architecture that was not present in the original conceptual architecture, other new components come as an extension of already existing components, not as a whole new thing, and were only discovered after examining the code. Therefore, the main loop is the only addition to the conceptual architecture.

In this report's concrete architecture, some of the components of the source code are merged into larger components made up of sub-components. This is done to get a more high-level overview, as FlightGear's code base is made up of many small components. The updated conceptual architecture, as shown in Figure 2 is previously derived conceptual architecture, with the addition of the main loop. However, Figure 3 provides the conceptual architecture in the same framework as the later derived concrete architecture, where smaller components are put together into larger components. Figure 3 is provided only for the sake of being consistent in the reflexion analysis, this is not a change in the conceptual architecture. It would not make sense to group the smaller number of components in the conceptual architecture into larger components in a presentation of the conceptual architecture.

As shown in Figure 2 the conceptual architecture revolves around the property tree and the main loop. The team's previous report had deemed the property tree to facilitate a publish-subscribe style. Around the property tree and main loop, there is the Flight Dynamics Model (FDM) which provides flight physics alongside the aircraft subsystem which would model the physical aircraft. Further, the AI traffic control subsystem is included for air traffic control with some AI techniques. The environment system provides scenery and weather simulation for an enriched experience. The rendering, UI, I/O, and sound & audio subsystems provide the user experience. Finally, the multiplayer component is included, which was deemed to operate under a client-server architectural style in the previous report.

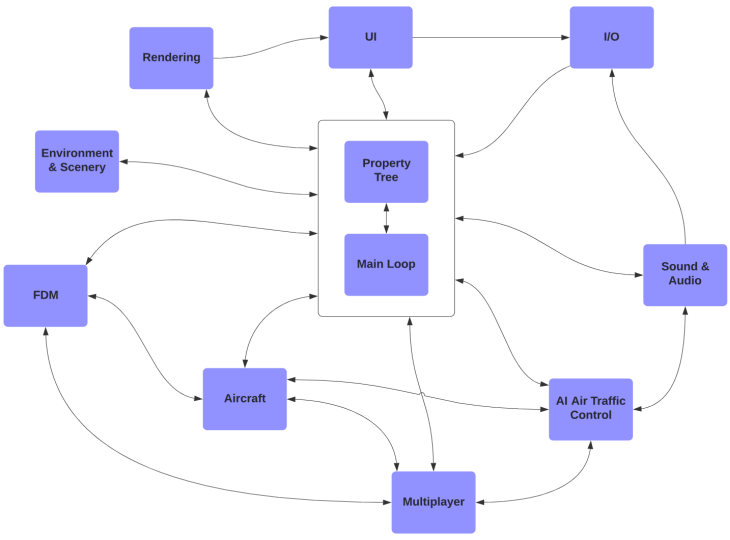

Figure 2: Updated Conceptual Architecture of FlightGear

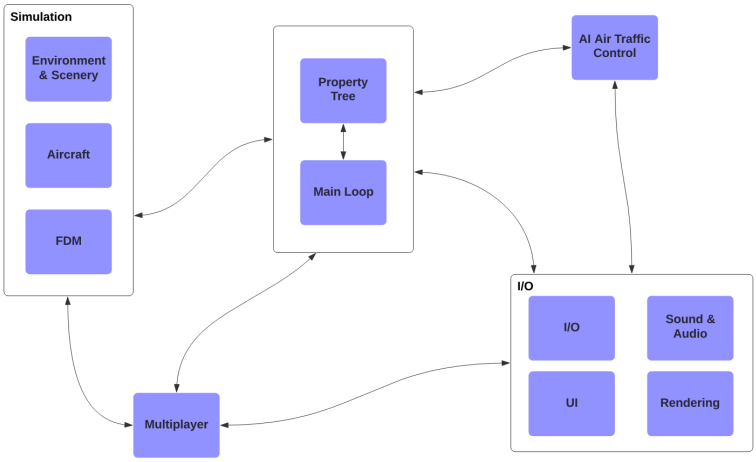

Figure 3: Simplified Conceptual Architecture for the Sake of the Reflexion Analysis

Concrete Architecture - High-Level Overview and Reflexion Analysis

The derived concrete architecture is presented in two parts. Figure 4 shows a very high-level overview, showing only the dependencies of the main loop and property tree. Figure 5 shows a high-level overview of the outer components, without the main loop and property tree. As FlightGear's architecture is very interconnected, these two separate diagrams are provided to show a clearer high-level overview of the system.

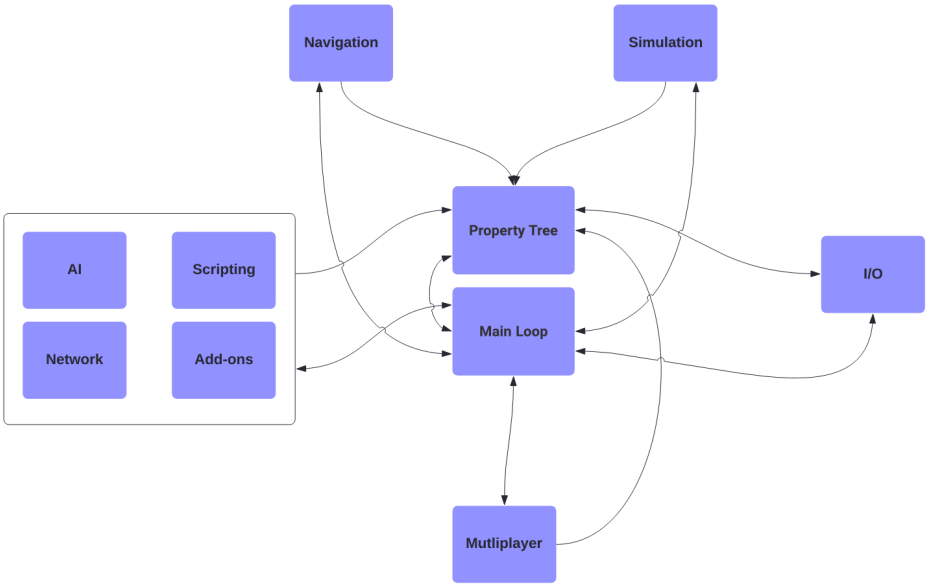

Figure 4: Concrete Architecture of the Property Tree and Main Loop

In Figure 5, two components that are both connected to the central black box share a bi-directional dependency unless a single-directional dependency is specified otherwise.

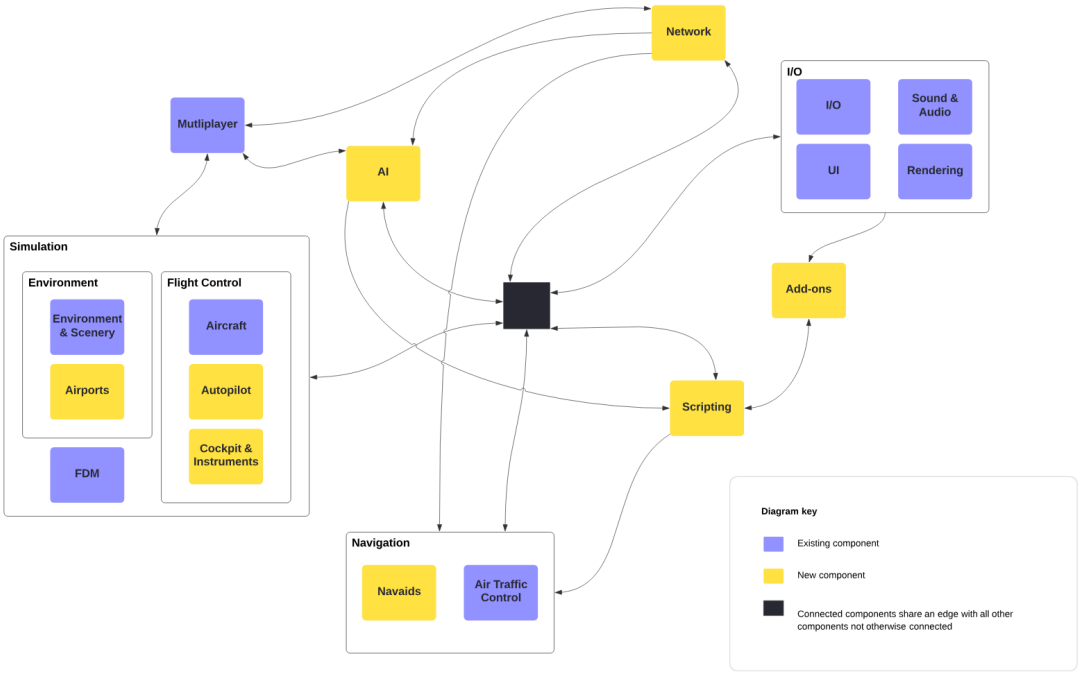

Figure 5: Concrete Architecture of FlightGear's Outer Components

Figures 4 and 5 do not indicate the divergences in dependencies for readability. The divergences are explained in the following sections.

Architecture Style - Description and Reflexion Analysis

In FlightGear's implementation, the team finds that 3 primary architectural styles are incorporated: repository, client-server, and object-oriented. The repository style is facilitated by the property tree, which acts as a database, storing most of the flight simulation data. This is highlighted in Figure 4. As shown in both the conceptual and concrete architecture, every single component of FlightGear depends on the property tree. These dependencies largely occur by reading data from the property tree. This is done through methods such as fgGetNode() and addChangeListener(). Unlike the conceptual architecture, most components are not depended on by the property tree, they only read/write data to the property tree. This is an oversight of the conceptual architecture, it wouldn't make sense for the property tree to depend on many other components. The team's conceptual architecture also identifies the property tree as a message bus that facilitates a publish-subscribe style, but FlightGear's implementation does not employ a publish-subscribe style. Components "listen" for changes, but they are not at all decoupled from other components, as they would be in a publish-subscribe style.

The second style employed is a client-server style, which was also identified in the previous conceptual architecture. The client-server is primarily present in FlightGear's multiplayer environment, but additionally, a client can connect to FlightGear's property tree to receive flight simulation data, as in src/network/props.cxx.

The final style is object-oriented. FlightGear's codebase makes extensive use of C++ classes which are used in addition to the property tree for sharing data between components. Figure 5 highlights FlightGear's object-oriented style, where all of the components are very interconnected with each other.

While this concrete architecture is presented in layers, this is only done to give a more abstract high-level overview, FlightGear does not follow a layered style. There are many dependencies between each component or “layer”, as opposed to a hierarchical structure.

The previous report also presented the process-control style as a style used to provide autopilot features. While this may still be present deep in FlightGear’s codebase, it is not a dominant style, so it is omitted from the concrete architecture.

Existing Components - Description and Reflexion Analysis

Simulation

The simulation component encapsulates FlightGear’s primary functionality, flight simulation. Included in this component are the FDM, environment, and aircraft subcomponents, which were present in the conceptual architecture. The FDM handles the physics of the plane, the environment subcomponent provides weather and scenery simulation, and the aircraft subcomponent provides some basic modeling of an aircraft. Additionally, the team finds that airports form their own large subcomponent of the environment component, providing runways and parking. As well, the team finds that the aircraft component also includes autopilot and cockpit & instrument subcomponents, allowing the pilot robust control of the aircraft. These new subcomponents were mentioned in the previous report but were not thought of as big enough to constitute their own component.

The only changed dependency between the simulation subsystem and components present in the conceptual architecture is the addition of the two-way dependency between simulation and I/O. One example of the IO depending on simulation is the use of the method getRunwaysWithoutReciprocals(), which is used as part of a process to draw airports for the GUI. On the other hand, the simulation component depends on the I/O component for example to provide radio communication in src/instrumentation/commradio.cxx. Such dependencies make sense and should have been included in the conceptual architecture.

Multiplayer

The multiplayer component remains in the concrete architecture. However, it carries some divergences in its dependencies. The previous report described FlightGear’s network capabilities as a part of the multiplayer component. Now that the network is a distinct component, the multiplayer component naturally shares dependencies with the network component. The multiplayer also shares a dependency with the new distinct AI model component, which provides AI multiplayer functionality in the file src/AIModel/AIMultiplayer.cxx. Most notably, the multiplayer component now lacks a dependency on IO and ATC. The team had anticipated a dependency between multiplayer and IO to provide voice communication between players. It turns out this can be entirely implemented without the use of the sound & audio component in src/Network/fgcom.cxx. The team had also anticipated a dependency between multiplayer and ATC for players to keep track of other players' locations, but it turns out this is entirely done through the property tree, for example in src/MultiPlayer/mutiplayermgr.cxx.

Navigation

The conceptual architecture includes an ATC component. In the concrete architecture, ATC is put together with the new Navaids subcomponent under the Navigation component. ATC provides typical air traffic control navigation by providing flight plans. Navaids is provided as a separate component with two

additional methods of navigation: visual navigation and instrument navigation. Visual navigation in its easiest way, often called pilotage, is just looking out of the window, comparing the outside with a map or with what is known about the area. Instrument navigation completely relies on instruments and radio beacons, view outside is not necessary at all. This might be done using radio navigation, route manager, and GPS navigation. Instead of relying on visual landmarks (such as roads, rivers, churches, and towns) for navigation under VFR (visual flight rules) - radio navigation depends on radio beacons and it is a critical part of instrument flight rules. Radio beacons are placed near airports, can be aligned with runways, and can be placed in remote areas to mark a route over them.

The only divergence present in the dependencies of the navigation is between the multiplayer and network components. This divergence is justified in the multiplayer description.

Property Tree

The property tree is the central database within FlightGear. As previously discussed with the architectural style, the property tree carries some divergences in its dependencies relative to the conceptual architecture.

Input & Output

The Input & Output component manages all of the input taken by the user and output given to the user. This includes subcomponents I/O, GUI, sounds & audio, and rendering.

The previous conceptual architecture report states that the I/O subsystem manages data flow into and out of the property tree subsystem. In addition, the report states that I/O directs data to the appropriate subsystem from the property tree. This is not the case in FlightGear's implementation. While the now larger I/O component still shares a bidirectional dependency with the property tree, this is only because the property tree makes use of some message boxes from the GUI component. The property tree does not need help directing data to the right subsystem, other subsystems can access the data they need themselves. In the concrete architecture, the I/O subsystem also loses a dependency with the multiplayer subsystem, which is explained in the multiplayer section.

I/O also picks up dependencies with the simulation component and all the new components of the concrete architecture. The dependency with the simulation component is explained previously in the simulation section. Other components incorporate the I/O subsystem for basic I/O functionality such as through the method guiErrorMessage(). The I/O subsystem depends on other components to output information. For example, the GUI component calls a method drawAirport(), which requires airport information from the simulation component.

Main Loop

The main loop as described in the updated conceptual architecture is present in the concrete architecture. The main loop initializes and controls the system. The method fgMainLoop() is the method responsible for invoking each process. This is done with the aid of src/Main/globals.cxx, which helps to keep track of the global subsystems of FlightGear.

New Components - Description and Reflexion Analysis

AI Model

The team's conceptual architecture included an AI air traffic control component. Originally, this was a component of FlightGear. Over time, FlightGear has implemented a small number of AI techniques that can help with more than just air traffic control [5]. Many components use the AI model to incorporate AI techniques. These AI methods are grouped into a separate AI model component which still includes ATC functionality (AIFlightPlan.cxx). Other parts of the AI model include modeling for ground vehicles (AIGroundVehicle.cxx) and advanced weather simulation such as storm creation (AIStorm.cxx). The different AI models are jointly managed within the AI Model component (AIManager.cxx), where the correct AI methods are invoked as needed. The new separate ATC control component and several other components work with the AI model, as shown in Figure 5.

Network

FlightGear has extensive network capabilities. In the conceptual architecture, the team described FlightGear's network capabilities as something only used for multiplayer functionality. In FlightGear's implementation, it is a standalone component that serves a role bigger than providing multiplayer functionality. In addition to the multiplayer component, the network component shares dependencies with components such as the simulation component and AI model component. The environment subcomponent, for example, makes an external request to an online aviation weather service through the method call: http->makeRequest(), a method of the network component. Other components interact with the network analogously. The conceptual architecture failed to consider how components would make requests to external services, hence the divergence.

Scripting

FlightGear has a massive network of inbuilt scripting components. This "scripting" component is not mentioned in the conceptual architecture and is new to the concrete architecture report. In FlightGear's implementation, the scripting component, titled "Nasal," supports reading and writing of internal FlightGear properties, creating GUI dialogs, and more. It has bidirectional dependencies with the other major components such as simulation, AI, and network, and with add-ons. It also depends on the Navigation 'navaids' component in methods like NasalFlightPlan->wayptFromArg() to return a NavaidWaypoint. The team missed this component in the conceptual architecture as the focus was not particularly on how internal properties interacted within the system and how internal data was accessed and presented.

Add-ons

In the concrete architecture, the team discovered FlightGear's add-on component, used to implement and import additional .xml libraries to FlightGear. This component was missed in the initial conceptual architecture as the team didn't consider looking at ways users could expand on FlightGear by adding their own modules or dialogue. The add-ons component contains dependencies with the scripting component. The add-ons depend on the scripting component in the method found in AddOn->getAddonPropsNode(), which utilizes a class in Scripting/FGNasalSys. The reason for this dependency is unclear from the code, it is possibly not a justified dependency. The scripting component provides scripts in NasalAddons.cxx to help manage add-ons, so the scripting component depends on the add-ons component.

The I/O component depends on the add-ons component, where it provides a GUI to control add-ons through the AddOnsController() method.

Environment Subsystem - Detailed Overview and Reflexion Analysis

A detailed, more low-level overview and reflexion analysis is provided here for the environment and scenery subsystem, a subcomponent of the simulation component

Conceptual Architecture

The team's previous report described the environment subsystem but did not provide a full conceptual architecture. The following conceptual architecture is derived from this description and is not based on any implementation details.

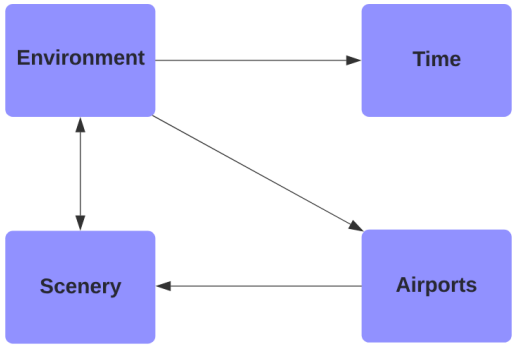

Figure 6: Conceptual Architecture of the Environment Subsystem

As shown, in Figure 6, the conceptual architecture of FlightGear described 4 primary components: environment, scenery, time, and airports. The environment component, which provides a climate and atmosphere, would work closely with the scenery component, which provides terrain modeling, to put together the entire global picture. The airport component would depend on the scenery component to model airports relative to the scenery. The environment component would depend on the airport component to incorporate real-world airports into the picture. Finally, the environment component would incorporate time-of-day modeling (i.e. provide lighting) with the help of the time component.

Implementation

Environment and scenery represent the landscape in FlightGear. The data is released as World Scenery collections. TerraGear supports FlightGear by creating the files used by FlightGear to represent the elevation and texture of the earth, including airports, cities, fields, forests, rivers, and roads. It contains data files containing ground elevation, airport locations, layouts, and geographical land-cover data. Environment class has multiple subcomponents such as atmosphere, climate, gravity, precipitation, etc. These are the different features and surroundings that involve simulating FightGear. Some functions include pressure and temperature within layers, season, wind conditions, temperature, and altitude.
Environment sub-components include atmosphere, climate, environment manager, fgclouds, precipitation manager, terrain, and tile manager.

The environment manager includes structures and classes for modeling atmospheric layers, interpolating environment values at different altitudes, and controlling environment interpolation. The 'LayerTable' class represents a column of the atmosphere by stacking environments vertically, with methods for reading configurations, interpolating values, and managing property bindings. The 'LayerInterpolateControllerImplementation' also handles initialization, updating, and synchronization of

environment interpolation, utilizing tied properties and node bindings for consistency. The environment manager does not provide any of the implementations of environment features but rather communicates with the rest of the FlightGear system and maintains the structure of the environment component.

The atmosphere subcomponent contains functions related to atmospheric modeling and altimetry calculations. It includes definitions for standard atmosphere layers, functions to calculate pressure and temperature within a layer as a function of height, and methods to determine pressure and temperature profiles based on altitude. There are also functions to handle altimeter readings, conversions between pressure units, and altitude calculations based on pressure readings. Fgclouds.cxx, is a file that is responsible for constructing cloud layers based on meteorological data. It includes functionalities to generate and manipulate 3D clouds within the simulator environment. It includes parameters cloud coverage, altitude, and type based on meteorological information like temperature, dew point, and pressure. It defines methods to add, remove, and reposition individual clouds within a layer, forming dynamic clouds.

The climate subcomponent contains several functions within the ‘fgclimate’ class that are defined to manage environmental parameters and user interaction within the flight simulation environment. The ‘set’ function is responsible for setting various environmental variables such as temperature, humidity, precipitation, and wind speed based on different climate conditions. These functions utilize interpolation techniques such as linear, sinusoidal, and triangular to calculate these parameters dynamically. The ‘test’ function is implemented to evaluate the behavior of these interpolation methods over different months, ensuring accurate simulation of seasonal variations.

The precipitation manager manages the intensity of precipitation based on altitude, wind direction, and velocity, and subsequently updates the rendering of precipitation within the simulator environment. It includes functionalities to manage rain and snow intensities, determine the maximum altitude for precipitation, and adjust precipitation type based on temperature conditions. Additionally, the code integrates with the simulator's scene graph to handle the rendering of precipitation effects, ensuring a realistic representation of weather conditions during virtual flights.

Within the scenery component, the terrain subcomponent contains classes and methods responsible for pre-sampling terrain roughness, including elevation data analysis and histogram generation. Additionally, it handles the configuration of sampling parameters such as radius, computation time, and maximum samples.

The tile manager includes routines for initializing the tile manager subsystem, scheduling tile loading based on the viewer's position and visibility range, and updating tile queues. The code handles various aspects with the respective functions such as caching, level of detail (LOD), and interaction with the TerraSync subsystem for synchronizing tile directories. Additionally, there are functions for checking the status of scenery loading and determining whether tile directories are being synchronized.

The airport component provides the physical outdoor layout of airports. This is done in AirportBuilder.cxx, which includes methods such as createRunway(), createPavement(), and createBoundary(). The airport component includes a ground network that facilitates the navigation of aircraft on the ground. An interesting method of groundnetwork.cxx is findShorestRoute(), which employs Dijkstra's algorithm [6] to find the shortest route for a taxi to move from point A to point B through the ground network. Additionally, the airport component provides an extensive runway implementation to model the runways and let aircrafts land and take off.

The time component provides time-of-day modeling. This is done primarily in TimeManager.cxx where real local times are imported, and world time is updated relative to frame rates and according to user specifications. The update() method takes into account all relevant factors to elapse the correct amount of

time. Methods in the implementation of update() include computeFrameRate() and computeTimeDeltas(). With the calculated times of day, the time subcomponent provides lighting calculations for display in the environment. Part of the calculation includes calculating the position of the sun or moon relative to the current viewpoint.

Concrete Architecture

The previously described small components of the environment system are merged into bigger components to provide a picture of the concrete architecture. The Understand tool is used to map each part of the code into the original 4 components and examine the dependencies.

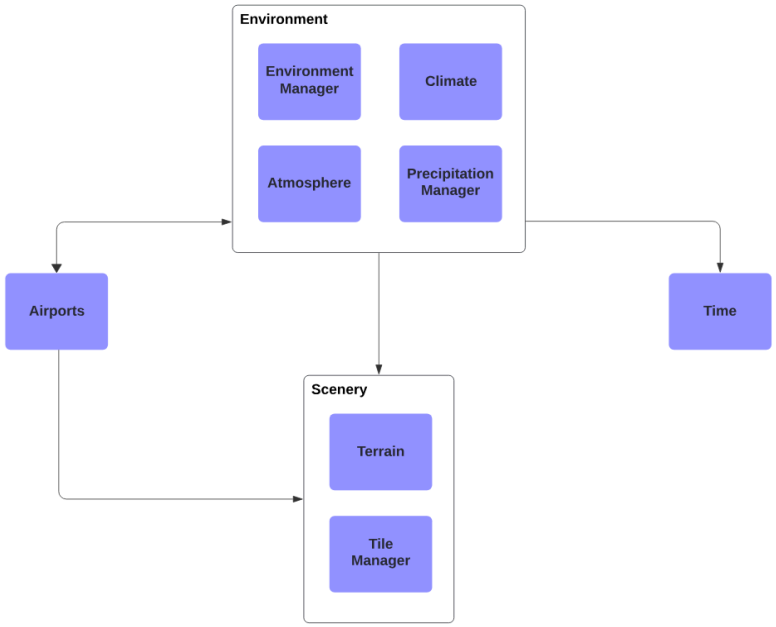

Figure 7: Environment Subsystem Concrete Architecture

Reflexion Analysis

Figure 7 carries two discrepancies relative to Figure 6. The first difference is that the airport component now depends on the environment component. One example of this dependency is that the airport component calls stationWeather.get_wind_speed_kt() of the environment component. This is done to choose the best runway for landing. The divergence is present because the original conceptual architecture only thought of the airports as physical buildings for scenery, not carrying the functionality of actually landing, for example. The second difference is that scenery no longer depends on the environment. This divergence is because the scope of the scenery component is relatively limited compared to what the team's conceptual architecture imagined. All that the scenery component is tasked with doing is modeling the basic terrain, irrespective of the environment. It is not tasked with, for example, reflecting environmental conditions such as precipitation in the terrain.

The inclusion of finer subcomponents is to illustrate some of the implementation details of this component but does not reflect a divergence from the higher-level conceptual architecture.

Use Cases

Use Case #1: Pilot takes off the aircraft

The first use case is the pilot taking off the aircraft. In the conceptual architecture, the team thought the UI subsystem would send the inputs received to the I/O subsystem to process. However, after looking into the source code, the team figured that the I/O subsystem has listeners subscribing to the inputs from the UI, such as mouse input, keyboard input, and joystick input.

When the user is ready for takeoff, they send appropriate mouse or keyboard inputs through the UI. The I/O subsystem listens to the inputs and publishes them to the property tree. During this process, the FDM subsystem runs the equations of motion to update the states related to the aircraft, such as geodetic positions, climb rate, and Euler angles. The environment subsystem listens to any scenery changes and updates the parameters like wind speed and elevation. The sound subsystem reads the updates from the property tree subsystem and makes the appropriate sound. The rendering subsystem reads rendering-related parameters, builds the visuals, and delivers the new frame to the UI. The user continuously provides inputs via the UI subsystem, and other subscriber components keep responding to input changes. This process continues until the aircraft is taken to the sky and takeoff is successful. *Figure 8* presents the sequence diagram for the described use case.

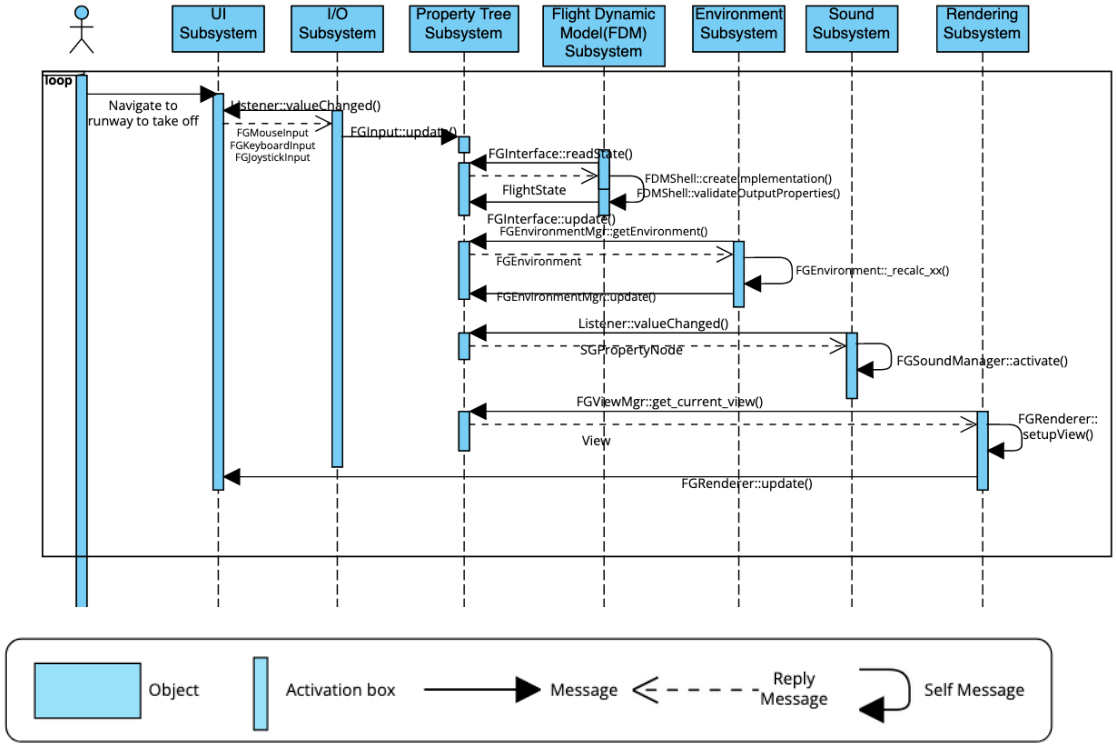

Figure 8: Use Case #1 Sequence Diagram

Use Case #2: Weather Configuration and Simulation

The user can select the desired weather condition to simulate in the manual configuration dialog. The user can configure cloud layers, precipitations and pressure, wind direction and speed, visibility, and

turbulence via the UI subsystem. The I/O subsystem collects all the inputs and updates the settings stored in the property tree subsystem.

After the simulation starts, as the aircraft's altitude changes, the FDM subsystem publishes the flight property-related parameter changes, such as altitude and climb rate. The environment subsystem first gets the configured weather information, then gets the altitude information from the property tree subsystem, and finally updates the environment values for a given altitude according to the rules. The rendering subsystem listens to the environmental parameter changes in the property tree subsystem and renders the updated weather visuals, which are subsequently delivered to the UI. As the aircraft's position changes during the simulation, the weather is continuously updated based on the user's configuration. *Figure 9* presents the sequence diagram for the described use case.

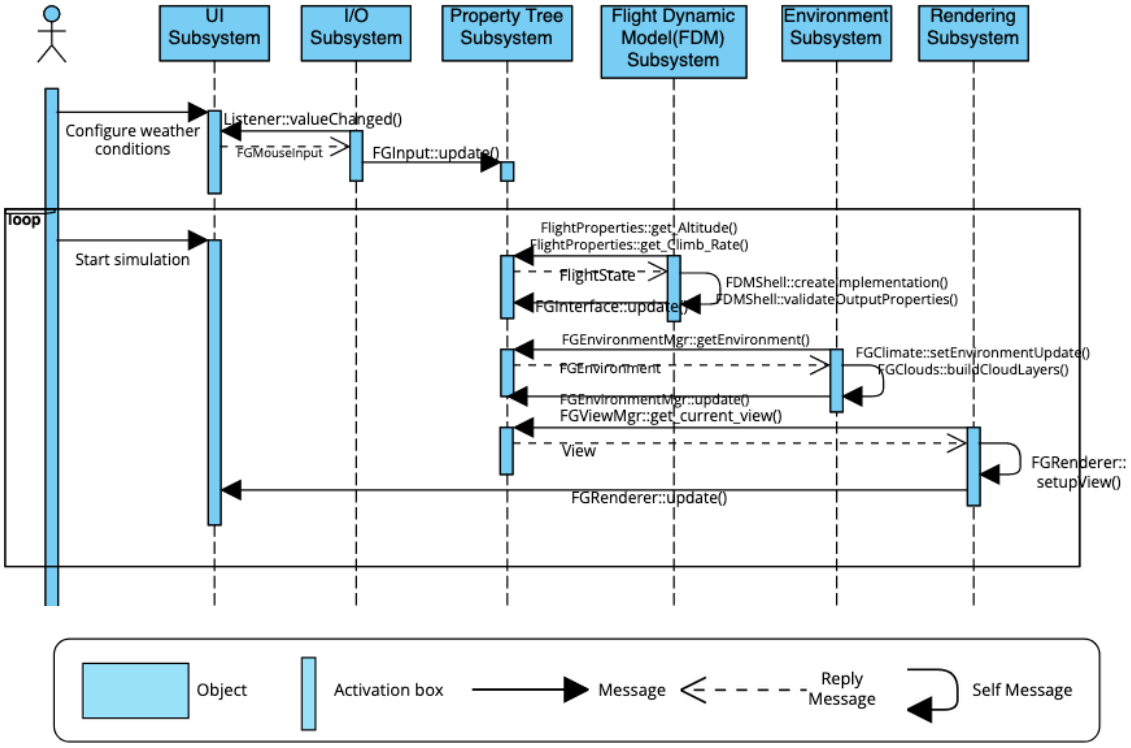

Figure 9: Use Case #2 Sequence Diagram

Data Dictionary

- TerraGear - a collection of open-source tools and rendering libraries [7]

Naming Conventions

- AI - Artificial Intelligence
- ATC - Air Traffic Control
- FDM - Flight Dynamics Model
- I/O - Input/Output
- UI - User Interface
- GUI - Graphical User Interface

Conclusions

After examining the code, the conceptual architecture was shown to be mostly incorrect. A further updated conceptual architecture would look very similar to the proposed concrete architecture. The concrete architectures for FlightGear feature a repository style, an object-oriented style, and a client-server style primarily for the multiplayer component. From the team's previous conceptual architecture, the 11 components were grouped and summed together to form five components that are in the team's concrete architecture. Upon analysis of FlightGear's source code, five additional components were identified and added to the team's concrete architecture. The main component of these ten is the property tree which acts as a central database. All other components depend on the property tree. An important newly added component is the main loop component that initializes and controls the system allowing FlightGear's processes to run. Additionally, the environment and scenery component was analyzed in great detail showing the 4 subcomponents of this component to be environment, scenery, airports, and time. Overall, FlightGear has many components that are very interconnected and depend on each other in different ways.

Lessons Learned

Throughout this assignment, the team learned a lot about the intricacies of maintaining software architecture. In putting together a meaningful architecture from a large code base, the team encountered many difficulties due to the high level of interconnectedness of the many components. Even though FlightGear's code base is relatively well organized, it was still a struggle to discover the dependencies between components and determine how they interact. Having a highly interconnected code base is likely true for most systems and so the team's experience with FlightGear's code base highlighted general problems in maintaining software architecture and increased the team members' understanding of how this task is done in the real world.

In addition to the difficulties the team experienced when constructing the concrete architecture, the team learned how to perform reflexion analysis and how conceptual and concrete architectures compare in practice.

Moreover, while using Understand to look into FlightGear's many files, it was noticed that a large majority of them had very few comments explaining their code and functionality. At first, it was surprising that FlightGear, which is also used in a professional context, did not seem to emphasize clear and explanatory coding practices, especially since FlightGear is open-sourced and allows anyone to view and add new code. However, it could be precisely because of FlightGear's open-sourced nature that it is difficult to maintain a common coding practice and ensure code is commented.

Analyzing FlightGear's code base and the dependencies of the different files and components allowed the team to learn about how software architecture maintenance is done in practice, and why commenting and keeping a record of functionality is one of the most important things to being a software developer.

References

[1] FlightGear - Wikipedia: https://en.wikipedia.org/wiki/FlightGear
[2] FlightGear - About: https://www.flightgear.org/about/
[3] FlightGear - Source Code: https://github.com/FlightGear/flightgear
[4] The FlightGear Main Loop: https://wiki.flightgear.org/The_FlightGear_Main_Loop
[5] AI Systems - FlightGear wiki: https://wiki.flightgear.org/AI_Systems
[6] Dijkstra's Algorithm - Wikipedia: http://en.wikipedia.org/wiki/Dijkstra's_algorithm
[7] TerraGear - FlightGear wiki: https://wiki.flightgear.org/TerraGear