CMPE322 Assignment 1 - Conceptual Architecture

FlightGear

Submitted By: Group 20
Jasper Lim, Laurie Yuzichuk, Jordan Herzstein, Campbell Love, Christopher Seguin Bianchi, Cameron Bennett

Contents

Abstract

This paper aims to explore the conceptual architecture of FlightGear, an open-source flight simulator. As an open-source development project, FlightGear has a long history, with many contributors, and rapidly evolves. Its contributors are divided into 'normal' and 'core' developers in which core developers have commit rights to the master branch repository and normal developers do not but can still contribute code. It had to undergo major restructuring to adapt its architecture to parallel processing-oriented computing. Now, FlightGear utilizes High-Level Architecture, which divides the simulation into different components. FlightGear focuses on modularity so as to be customizable to any particular user's preferences, saving size and resources on the base version of the program.

In terms of system functionality, FlightGear has an FDM Server, which handles the bulk of the flight dynamics as well as atmospheric and environmental simulation, in addition to multiplayer sessions. The FDM Client presents users with the input/output interface, and components like rendering. Both the FDM Client(s) and Server connect to a network via UDP ports, and relay data to each other. The FDM Server through the network manager will call upon its components which handle the various aspects of the flight simulation, which are organized to be modular. Thus, we can say that FlightGear partly utilizes a modified Client-Server model of architecture. FlightGear relies on its restructured concurrency to operate with speed and responsiveness, especially concerning the modularized flight simulation components.

Introduction

FlightGear is a free, open-source flight simulator in development since 1997, supported on multiple OS's including Windows, MacOS, Linux, IRIX, and FreeBSD [1]. FlightGear has been used in academic research, education, training, and for recreational purposes [1].

FlightGear was originally proposed as an open-source project by developers David Murr, Curt Olson, Michael Basler, and Eric Korpela in 1996, who were dissatisfied with proprietary flight simulators, and released to the public their own simulator with open-source volunteers and resources [1]. FlightGear had its first full release in 2007 and has been in constant development since then [1]. The base files amount to less than 2GB, which compared to other competing flight simulators like X-Plane or Microsoft Flight Simulator, which contain 60GB and 120GB of base files respectively, make FlightGear a much more streamlined application [1] [2] [3].

FlightGear features an atmospheric and environmental physics simulation, including real time weather patterns. Players can also choose between JSBSim and YASim for flight dynamics, and engage in multiplayer functionality [1]. The features and their sophistication make FlightGear favored by the Federal Aviation Administration (FAA), and NASA amongst many other aeronautics and space industry organizations [1]. Amongst these organizations it is considered the standard for incorporating into training, modelling, simulation, and other software [1].

Per this report, there are many resources used including general books and published papers on flight simulators, documentation of FlightGear, and the FlightGear wiki.

Architecture

In this section, FlightGear's system functionality will be discussed. First part of this section will go over system functionality including the FDM server and client alongside networking capabilities, followed by system evolution, data flow and control, concurrency, and finally division of responsibilities and how that effects workflow.

System Functionality

To address the issues posed by the 'main-loop' architecture used in previous versions of the flight simulator, FlightGear redesigned their architecture to optimize computing power usage [4]. By creating an environment capable of threading resource intensive tasks, it could much more easily foster improvements and additional features. This architecture is centered on dividing the simulator into two primary components: the Flight Dynamics Model (FDM) server and the client, utilizing a Model-View-Controller (MVC) architecture for increased flexibility and efficiency [5].

FDM Server

The FDM server performs the main calculations of the flight models and is responsible for the core simulation tasks, including the computation of flight dynamics and the management of environmental variables [6]. It now supports parallel processing, allowing for the simulation of multiple aircraft simultaneously, multiplayer, and shared AI traffic data across sessions [7]. This server-centric approach enables a scalable and robust simulation environment that can accommodate complex flight scenarios and interactions within that multiplayer setting.

FDM Client

The FDM client serves as the interface between the user and the simulation by handling input/output operations as well as rendering the simulation's visual and audio components [6]. As with the previous architecture, these rendering tasks formed the bottleneck for system performance, so in the new architecture these I/O tasks are performed in a separate thread from the rendering tasks [4]. The client's structure allows for a responsive and immersive experience, with the capability of displaying detailed and dynamic environments. It handles communications with the FDM server to reflect real-time changes in the simulation state, ensuring that user inputs directly influence the flight dynamics [5].

Networking and Multiplayer Capabilities:

The new architecture also supports multiplayer functionalities, allowing users to interact within a shared simulation environment [4]. This is possible using efficient network communication protocols to ensure consistency and synchronization across different clients. The system's design allows for seamless integration of new features, such as multi-pilot capabilities and AI traffic, increasing the realism of the simulation experience [7].

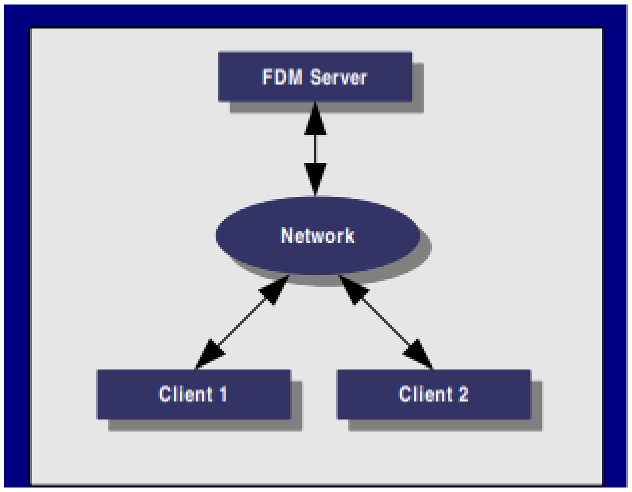

Figure 1: An example of a setup where multiple users can inhabit a single aircraft and operate its controls. [4]

The new architecture emphasizes the system's robustness and capability of handling multiple users and complex simulations without compromising performance [4]. This architecture not only addresses the limitations of the previous designs but also sets a foundation for future developments, leveraging parallel processing and networked components to offer a complete and immersive flight simulation experience [4].

System Evolution

FlightGear is a program that evolves quickly due to the nature of how development occurs. FlightGear is an open-source flight simulator that grows using an open collaboration model, allowing any developers to contribute to the project. There are two forms of developers, core developers and open or 'normal' developers [5]. The difference between these two are explained in the *Division of Responsibilities* section of this report. Where this is applicable to system evolution is that when a core developer reviews the normal developer's work, they can commit the changes to the official version and send out a new patch (i.e. version) [8]. This approach enforces innovation, diversity, and rapid iteration of code, 3D models, documentation, and other assets, while assuring that the product stays in top shape and no errors are committed to the software.

Flight Gear's system architecture also is built upon a modular design to enhance the modifiability of the system. Various modules include the aircraft models, scenery databases, user interface components, etc [9]. Each of the modules are designed to be independent, which allows any developer to work on specific portions of the code without conflicting with other components of the project [8]. This modular design in tandem with the open collaboration creates an efficient dynamic that allows rapid modifications, iterations, and testability.

FlightGear’s comprehensive and accessible documentation helps it succeed with the mass number of collaborators installing, updating, and troubleshooting the products. The documentation of the product updates with the software and reflects changes in features, functionality, and best practices [8]. Other documentation such as support forums, mailing lists, and chat channels offer additional help to users of the product [8]. These forums allow users to also post about bugs and participate in providing feedback and testing, improving ongoing development efforts [8].

Data Flow and Control

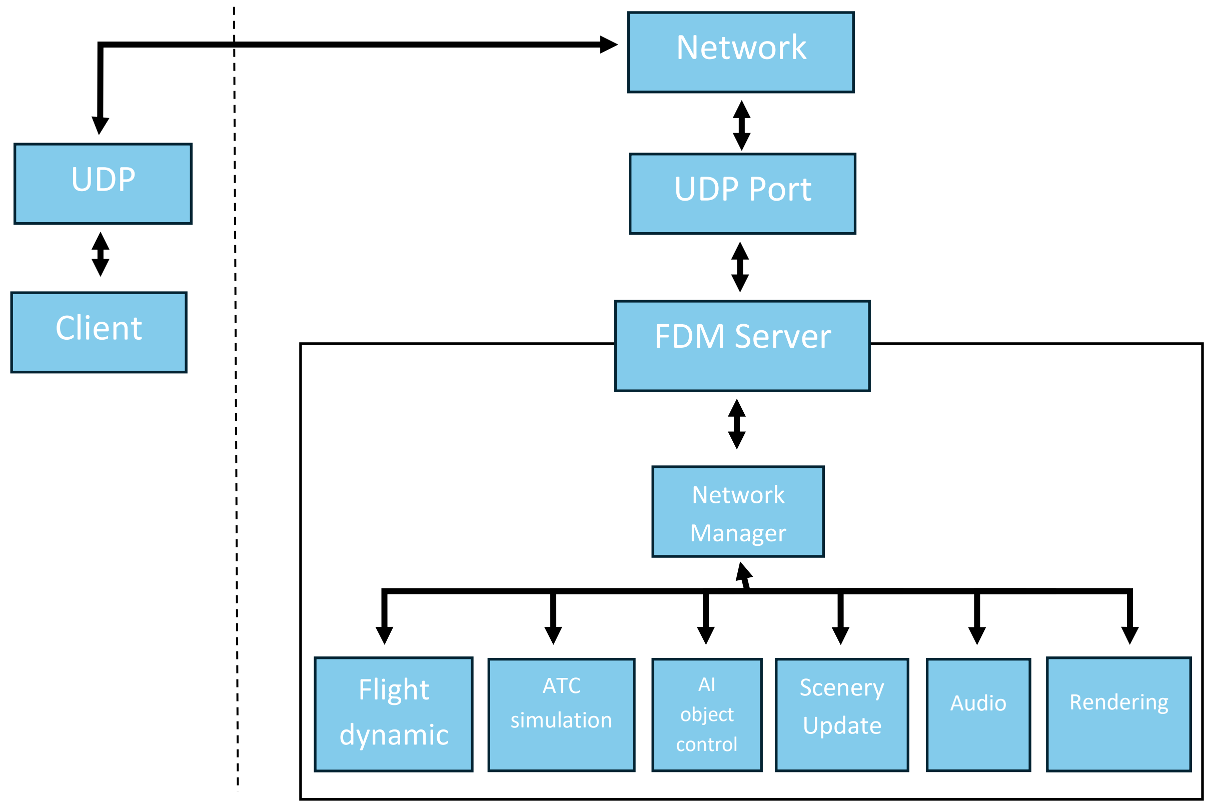

Figure 2: Diagram of FlightGear's Data Flow and Control Flow [4]

As shown in Figure 2, through the UDP Ports, the FDM server and client connect to a network. In the network manager, changes can be uploaded to and from one another and allow for either side to react accordingly.

The input the user gives using the UI, such as steering the aircraft and pushing various control buttons on the dashboard, is communicated through the network to the FDM server. Upon receiving the data, the network manager will call relevant components to make the necessary updates. Primarily, calculations based on the given input, equations of motion and aerodynamics will be made [4]. However, audio and visual rendering as well as updating the scenery outside the user’s aircraft allow for the user to receive confirmation that their inputs have been registered by FlightGear's system. These updates will be sent back to the network manager and from there, the FDM server will update the network and thus the client [4].

Additionally, the system controls various AI systems that act separately to the user's inputs [7]. One of these is AI traffic, where the scenery is populated with AI controlled vehicle models with their own traffic schedules that dictate how that AI object behaves [7]. This includes AI aircraft, both on the ground in airports and in the air, and control of other ground traffic such as cars, trains, and ships [7]. Other instances of AI systems are the ATC, which simulates communications between the airport tower and the pilots, and various AI scenarios, such as weather/turbulence [7]. All these systems add realism to FlightGear and provide training opportunities to the user so that they can practice safely piloting the aircraft regardless of the circumstances and what is in their environment.

Having these components act independently from one another instead of one after another in a large loop allows for partitioning and ensures that any errors or corruption in one component will not cause problems to the others [4].

Below are some instances of use cases for the system represented through sequence diagrams:

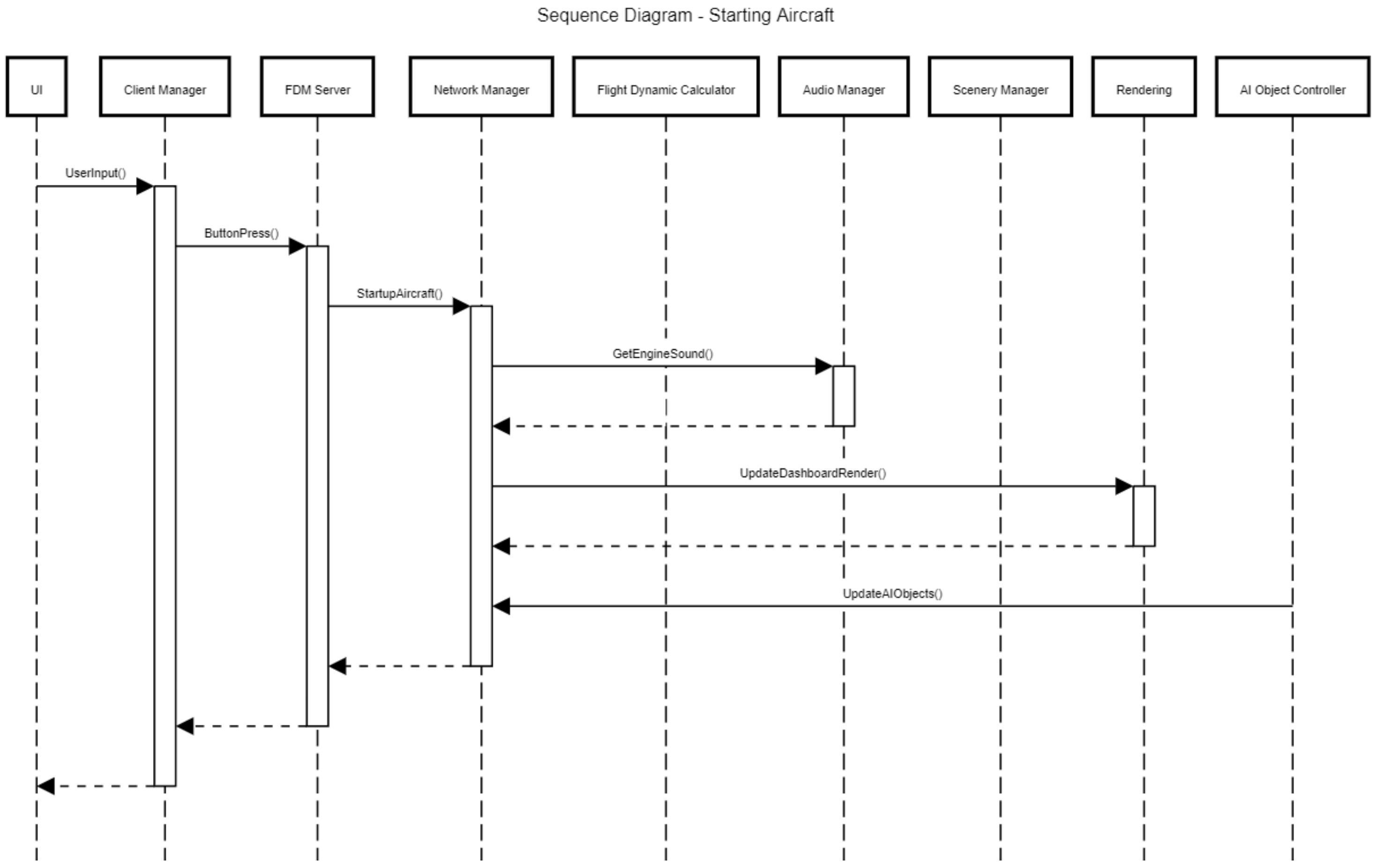

Figure 3: Sequence Diagram for Starting Aircraft

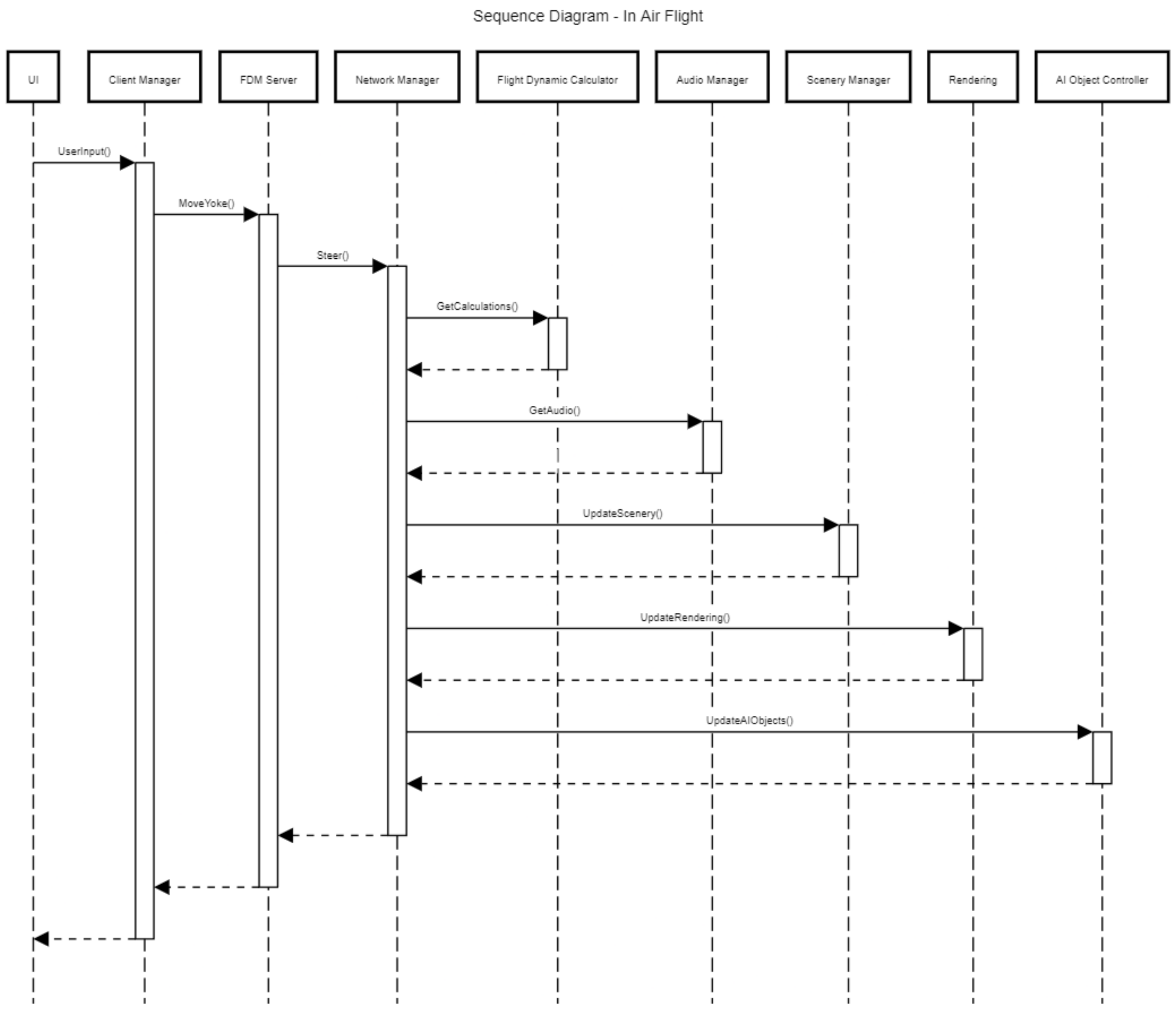

Figure 4: Sequence Diagram of When Flying Aircraft

Concurrency

In the context of flight simulation software such as FlightGear, concurrency plays a pivotal role in achieving real-time performance and realism essential for an immersive flight simulation experience. The evolution of processor architecture, notably the reduction in cycle times and the introduction of multi-core processors, has significantly improved the ability to perform parallel processing of tasks across different cores, enhancing the speed and efficiency of simulations [10]. Operating systems further optimize this process by managing data transfers through mechanisms like Direct Memory Access (DMA), enabling concurrent data handling alongside ongoing processor operations crucial for maintaining the simulation's real-time responsiveness [10].

The operating system's scheduler is used for managing multiple processes, focusing on maximizing throughput and ensuring real-time inputs are prioritized accordingly [10]. Flight simulation software typically operates under a fixed frame rate, necessitating that all processing activities, from input acquisition and flight dynamics computations to visual updates, are completed within each frame [10]. This emphasizes the importance of efficient concurrency management to avoid processing overruns and ensure timely execution. Distributed computing also enhances flight simulation by partitioning applications across multiple computers, potentially improving processing speed proportionally to the number of computers used. However, challenges arise in managing interconnections and minimizing latency [10].

Within individual computers, the software design embraces multitasking, allowing parallel execution of tasks using processes and threads managed by the operating system [10]. Threads, which operate independently but share the same code and data, can be executed on separate processor cores, enabling parallel execution while ensuring efficient synchronization and data sharing among them [10]. Synchronization between threads is achieved using semaphores, ensuring mutual exclusion for shared resources, and controlling the sequencing of threads' execution to prevent conflicts [10]. Moreover, asynchronous input handling employs threads to manage input and output operations efficiently, allowing for real-time data acquisition and immediate response to control inputs, crucial for maintaining the simulation's real-time performance [10].

Overall, the implementation of concurrency in flight simulation software through improved processor capabilities, operating system management, distributed computing, and efficient use of threads and synchronization mechanisms ensures that FlightGear and other simulators can process complex simulations in real-time. This highlights the importance of concurrency in a large software system.

Division of Responsibilities

The FlightGear development process and role of volunteers is detailed in the development portal of the official FlightGear wiki. Participating programmers are separated into two categories: normal and core developers [5]. Core developers are those who have commit rights to the master branch of the FlightGear Git repository and make direct contributions to the source code (i.e. they are the ones who develop FlightGear "core") [5]. Normal developers can make improvements to the source code, but they need to be reviewed by core developers to be committed to the main project [5]. These sets of checks and balances make sure that competent modifications are being made to the project while maintaining quality assurance to the end application.

The development contribution process is separated into several steps for normal developers. First, developers are encouraged to fork and clone the repositories from Source Forge that they would want to contribute to so that they can work on those repositories without changing the main project before the modifications are accepted [8]. The original repositories are established as an upstream remote that can be pulled into the developer's local clone [8]. Once that developer is ready for their work to be accepted, they can submit a merge request, in which a core developer will check that contribution to fix any issues before merging with the main repository [8]. After their work is accepted, the developer needs to be responsible for monitoring the build server, mailing lists, and the forum to check for any issues related to that developer's addition [8]. Finally, the developer must pull their changes from the upstream remote and remove their local branch to push to origin [8].

It is important to note that FlightGear employs High-Level Architecture (HLA), a general-purpose architecture for distributed computer simulation systems [11]. HLA splits the simulation into different components called Federates, which interact via a Run-Time Infrastructure (RTI) [11]. One of the major advantages HLA has over monolithic simulation is that it provides a framework to allow anyone to create components in FlightGear that are language flexible (not just C/C++), attracting many developers who may have other languages of expertise [11]. Since FlightGear mainly relies on volunteers, this is very beneficial for the project as it attracts a larger quantity and wider variety of programmers to contribute.

All repositories have certain policies and restrictions applied to them, such as software licenses [12]. All the FlightGear software repositories either use the GNU Public License (GPL) or require a GPL compatible license [12]. The GNU Public License is a copyleft, free software license [13]. To summarize, Table 1 contains all the relevant repositories and their required software licenses, taken from the FlightGear policy document and information on the wiki about the git repositories [12] [9].

Repository	Comments	License
FlightGear	FlightGear Itself	GPLV2+
simgear	Simulation engine FlightGear uses	LGPLV2+
fgdata	All Data used by FlightGear	GPL Compatible
fgaddon	SVN containing all unofficial flightcraft	GPL Compatible

Table 1: FlightGear Repositories and their compatible Licenses [11]

Some of these repositories do not necessarily contain code contributions since anyone can contribute to the open-source project, which can also include assets such as unofficial flightcraft in the fgaddon repository, FlightGear data in the fgdata repository, and also contributions to documentation on the FlightGear Wiki itself [12] [14].

Conclusion

To summarize the conceptual structure of FlightGear, the purpose of the system is to accurately simulate aircraft flight in a way that is open, accessible, and easily modifiable for developers and non-developers alike. Its own infrastructure, which utilizes a combination of a modified client-server architecture, HLA, and MVC architecture, reflects this goal [6] [11]. It is in part able to accomplish this by creating an environment for threading resource intensive tasks. For example, using a modified Client-Server model of architecture, it utilizes the FDM server for computing intensive core simulation tasks to accommodate complex flight scenarios even in a multiplayer setting through the architecture's networking capabilities. Between client/server interaction, I/O and rendering tasks are updated in real time so the simulation is as immersive and responsive as possible. The components from the client-side that interact with the network manager are separated so that individual components are partitioned and do not interfere with each other making modular development and usage easier.

The way in which FlightGear sets up its architecture and policies encourage a large quantity of developers, a collaborative development environment, and a modifiable project. FlightGear's Higher-Level Architecture allows development and contributions to be mostly programming language agnostic as to attract a large quantity and variety of developers [11]. The development process itself encourages collaboration between dedicated core developers and those who only want to contribute to a few components while also assuring quality of code through its practices. Additionally, the policies of the repositories attached to the project assure all components are free and open source through GPL compatible licenses, keeping development open for all [12]. Even for non-programmers, volunteers can contribute and add their own components and assets such as aircraft to the project, reflecting the goal of a realistic flight simulator that is open and collaborative in its creation [14].

Lessons Learned

Most of our research was sourced from the FlightGear wiki which is reviewed and maintained by FlightGear contributors, and this helped us get accurate information on the project's architecture and approach to development. One limitation of this approach was that we were not able to find much information on the FlightGear repository for concurrency specifically, so for our concurrency section we also consulted books with information on how flight simulators are developed generally such as D. Allerton's book "Flight Simulation Software" [10]. Sourcing books about general flight simulators helped us understand the theory behind flight simulators, though there could be some missing information on how it uniquely applies to some aspects of FlightGear. Another limitation of our methodology is that we did not consult information that would compare in depth different flight simulators such as Microsoft Flight Simulator with

FlightGear. However, it is important to note that some comparisons would be difficult to find where there is not as much information on the software architecture of more closed projects like Microsoft Flight Simulator. In the process of learning about FlightGear's conceptual architecture, it is apparent that a modular approach to the software architecture and policies of open-source projects can benefit the product in terms of performance and continued development. It is suggested that any software projects that want to emulate the open nature of FlightGear adopt some similar practices that are in the project such as its HLA that allows it to be code agnostic and allow non-programmers to also contribute assets that programmers would not have the time to add [11] [14].

Terminology and Naming Conventions

Core Developers: Developers that have commit rights to the official FlightGear Git repo.

Normal Developers: Developers without commit access to the official FlightGear but contribute code to the project.

Semaphores: a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system.

AI: Artificial Intelligence

ATC: Air Traffic Control

DMA: Direct Memory Access

FDM: Frequency-Division Multiplexing

GPL: GNU Public License

- *GPLV2(+):* GNU Public License Version 2 (and above)
- *LGPLV2(+):* Library GNU Public License Version 2 (and above)

HLA: Higher Level Architecture

MVC: Model-View-Controller

RTI: Run Time Infrastructure

UDP: User Datagram Protocol

Bibliography

[1] "FlightGear About," FlightGear, [Online]. Available: https://www.flightgear.org/about/. [Accessed 19 02 2024].

[2] "Microsoft Flight Simulator," Microsoft, [Online]. Available: https://www.xbox.com/en-US/games/microsoft-flight-simulator. [Accessed 20 February 2024].

[3] "X-Plane," X-Plane, [Online]. Available: https://www.x-plane.com/. [Accessed 20 February 2024].

[4] A. MacLeod, A. K. Hardraade, M. Koehne and S. Knoblock, "New FG Architecture," [Online]. Available: https://wiki.flightgear.org/w/images/1/1e/New_FG_architecture.pdf. [Accessed 20 February 2024].

[5] "Understanding the FlightGear development process," [Online]. Available: https://wiki.flightgear.org/Howto:Understand_the_FlightGear_development_process. [Accessed 18 February 2024].

[6] "FDM Engine Feature Standardization," [Online]. Available: https://wiki.flightgear.org/FDM_engine_feature_standardization. [Accessed 20 February 2024].

[7] "Artificial Intelligence," [Online]. Available: https://wiki.flightgear.org/Artificial_intelligence. [Accessed 14 February 2024].

[8] "Developer Workflow," [Online]. Available: https://wiki.flightgear.org/Development_workflow. [Accessed 18 February 2024].

[9] "FlightGear Git," FlightGear, [Online]. Available: https://wiki.flightgear.org/FlightGear_Git. [Accessed 18 February 2024].

[10] D. Allerton, Flight Simulation Software, Hoboken: Wiley, 2023.

[11] "High-Level Architecture," [Online]. Available: https://wiki.flightgear.org/High-Level_Architecture. [Accessed 18 February 2024].

[12] "FlightGear Policy Document," FlightGear, [Online]. Available: https://www.flightgear.org/flightgear-policy-document/. [Accessed 18 February 2024].

[13] "Various Licenses and Comments about Them," [Online]. Available: www.gnu.org/licenses/license-list.en.html..

[14] "Volunteer," [Online]. Available: https://wiki.flightgear.org/Volunteer. [Accessed 20 February 2024].