FlightGear Conceptual Architecture

Prof. Amir Ebrahimi
Queen's University
CISC 322
20 February, 2024

Al-Barr Ajiboye (albarr.ajiboye@queensu.ca)
Kristina Arabov (20kah10@queensu.ca)
Jack Caldarone (21jsc5@queensu.ca)
Safowan Mostaque (20sbm4@queensu.ca)
Grace Odunuga (20goo@queensu.ca)
Jenna Wang (20jw99@queensu.ca)

Table of Contents

Abstract

FlightGear is an open-source flight simulation software purposed for education and research, recreational use, and virtual flight practice. In this program, the main functionalities are allowing users to control different types of aircraft models (i.e. airliners, helicopters, military planes, etc.), adjust weather settings, navigate between numerous locations, and potentially interact with others on the same simulation. The capabilities of FlightGear are intricate and extensive, allowing the user free-reign in customizing numerous aspects of the simulation and maneuvering through it. FlightGear operates on multiple OS platforms, using modest processing and graphical requirements. Thanks to its open-source code, users and developers can freely contribute new aircraft models and frameworks to the software for others to use.

In this report, an analysis of FlightGear's conceptual architecture is performed. This required our group to avoid looking at the concrete architecture, and instead compare the requirements of general flight simulation components, interactions, and patterns to those of FlightGear. This report finds that FlightGear utilizes almost all common components of general flight simulation software, and allows for more customization due to its open-source code and active user base. Through this report's analysis and understanding of different architectural styles, our group believes that one of the most significant aspects of FlightGear is its scalability as a program. Therefore, because of FlightGear's similar functions and procedures of flight simulation programs, as well as its modifiability, this report concludes that the best-fitting conceptual architectural style for FlightGear is publisher-subscriber.

Introduction and Overview

The evolution of flight simulation technologies from the early mechanical simulators to today's advanced digital platforms marks a significant stride in aviation and software engineering. The historical development, as chronicled by David Allerton in "Principles of Flight Simulation," delineates a technological journey from analogue computing post-World War II to the current era dominated by microelectronics. Within this continuum, FlightGear, an open-source flight simulator, emerges as a beacon of innovation and collaborative development.

Initiated in 1997, FlightGear is distinguished by its commitment to open-source principles, facilitated under the GNU General Public License. This foundational approach has enabled a diverse global community of developers and enthusiasts to contribute to its development, making FlightGear a versatile tool for research, education, pilot training, and recreational purposes. The simulator's architecture, embracing the publisher-subscriber (pub-sub) or implicit invocation style, represents a paradigm shift towards modularity, scalability, and adaptability in software design. This architectural choice supports a loosely coupled system where components, such as the simulation of environmental conditions, aircraft dynamics, and user interfaces, operate independently yet cohesively, reacting dynamically to events and data updates.

FlightGear's software architecture encapsulates various subsystems critical to flight simulation, including but not limited to, the equations of motion, aerodynamic models, engine simulations, and weather patterns. These components are intricately designed to mirror the complexities of real-world aviation, offering an immersive simulation experience. The pub-sub model plays a pivotal role in integrating these components, facilitating a seamless interaction that enhances the simulation's realism and fidelity.

The open-source model of FlightGear not only democratizes flight simulation technology but also fosters a culture of innovation and shared learning. This approach aligns with the educational ethos of third-year university studies, emphasizing collaboration, critical thinking, and practical application of software engineering principles. The conceptual software architecture of FlightGear, particularly its implementation of the publisher-subscriber model, serves as an exemplary case study in understanding the dynamics of modern software systems in complex, real-time simulation environments.

As FlightGear continues to evolve, propelled by the contributions of its vibrant community and the inherent flexibility of its architectural design, it stands as a testament to the potential of open-source collaboration in advancing technology. This report aims to delve into the architectural nuances of FlightGear, exploring how the publisher-subscriber model underpins its operation and facilitates a rich, interactive simulation platform that pushes the boundaries of virtual aviation.

Architecture

Based on the provided readings of both general and specific flight simulators, as well as the functionality and goals of FlightGear, our group hypothesizes that FlightGear's conceptual architecture is in the publisher-subscriber/implicit invocation style. As stated by the course material, the implicit invocation architectural style consists of a loose collection of components that do not communicate with each other individually, but rather choose to take part in the system through a connecting procedure they can subscribe to (Ebrahimi, 2024, p. 107). This is one of the key reasons why we think FlightGear utilizes this style since the simulation must consider a wide variety of constantly changing variables and user requests often related to the operation of the aircraft, the tools available to the user, and any calculations which affect the behaviour of the simulation. In addition, one of the unique features of this architectural style is that components can be freely introduced to the system without affecting the overall software architecture (Garlan & Shaw, 1993, p. 10). This fits with the fact that FlightGear is open-source and developers can contribute new additions and components freely. We understand that a significant drawback of the implicit invocation style is that variables and components are not considered by the system in any specific order when they are subscribed, however, we conclude that the real-time functionality of FlightGear accounts for this by only consistently updating specific behaviours of the simulation based on certain variables, instead of updating all parts of the simulation at once.

Functionality of the System and its Parts:

To start, the goal of FlightGear is to simulate realistic flights of various aircraft models and their controls between different locations, while also taking into consideration changing weather patterns, air traffic, and aircraft behaviour. FlightGear has the functionality to either simulate the aircraft's flight in real, fast, or slow time. Users also have the ability to control multiple parts of the software, and may interact with other users within the same simulation.

Before analyzing the specific components of FlightGear, it is important to understand the basic expectations of general flight simulation software. As noted by various research papers, flight simulation applications often include navigation functionalities, aircraft displays, accurate visuals and sounds, pilot controls, data acquisition and utilization, weather patterns, aerodynamics and various physics calculations (Allerton, 2022, p. 3). FlightGear applies these component expectations while also factoring in its purposes of being used for research, pilot practice, and individual experimentation. Along with the consistently updating flight simulation as the broadcasting system, FlightGear's implicit invocation architecture is broken into these main components: the aircraft and its characteristics, weather conditions, physics and aerodynamic calculations, user controls and requests, navigational systems, image and sound generation, and time functionality. One significant point to consider is that the components may be passive or updated by the user, where user-based components rely on user requests and input (e.g. controlling the aircraft), and passive components act independently of the user (e.g. changing weather characteristics). The processing and accuracy of the simulation is entirely dependent on these components.

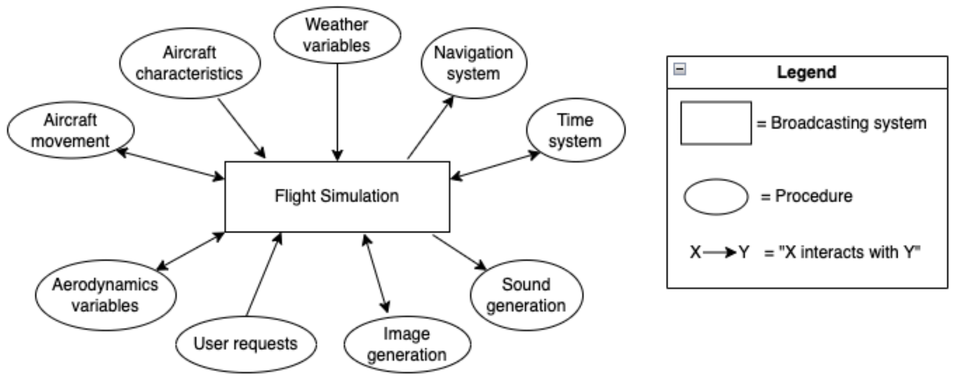

Figure 1: Conceptual architecture of FlightGear using the publisher-subscriber style.

Aircraft Type – Users have the ability to import various aircraft models or develop their own models, with each having unique attributes such as size, wingspan, and type ("Aircraft," 2023). The aircraft model subsystem can determine the capabilities of the user's control, how it reacts to certain weather conditions and forces, and the behaviour of the flight simulation system. This subsystem is static since it is independent of other subsystems, similar to the user component.

Weather Conditions – The weather subsystem affects the flight, environment, and behaviour of the plane in the simulation through diverse weather patterns (e.g. sunny, snowing, raining, etc.). This is done through various calculations of the atmosphere, such as temperature, pressure, humidity, wind, and visibility ("Weather," 2023). The user can customize weather conditions during the simulation, which aligns with the software's goal of allowing users to experiment and visualize the behaviour of their aircraft in certain scenarios. This subsystem is crucial to the simulation because any updates and changes in variables to this component will almost always affect the performance of variables in other subsystems.

Physics and Aerodynamics – This subsystem updates variables related to the forces present during flight. It utilizes two different frameworks for this: JSBSim and YASim. Both of these models allow the subsystem to reconfigure forces the aircraft may experience, such as dynamic pressure, drag, lift, and wind, in relation to its mass, size, wind, or angles ("JSBSim Aerodynamics," 2019, "YASim", 2023). Aspects like flight duration and navigation of the aircraft are consistently reconditioned, and in turn, continue to affect physics and aerodynamics.

User controls – The user subsystem largely determines the behaviour of the flight simulation since the user can initialize almost all aspects of the simulation through the software or a command line interface. More specifically, the user can customize the type of aircraft they control, the weather/wind patterns, their starting location, and the movement of the aircraft ("Chapter 8," 2023). This subsystem is not affected by the simulation. Rather, the user publishes any requests or changes and the simulation must act accordingly.

Navigational systems – The navigation subsystem relies on the changing flight simulation elements to accurately update the user's position, map, and flight path. This is done through the utilization of different air laws and radio frequencies, as well as changing weather patterns ("Chapter 5," 2023). The navigational systems are subscribed to the overall flight simulation broadcast, thus giving the user the ability to understand their surroundings and act accordingly, similar to navigational systems in real life.

Image and Sound Generation – The image subsystem updates the visuals of the simulation based on the user's environment and potential weather conditions, and the sound subsystem outputs relevant sound effects dependent on what is happening in the simulation. The image system is managed by a rendering pipeline named Compositor which is responsible for various shaders and effects in the simulation ("Compositor," 2024). The subsystem may either be subscribed to the flight simulation (if any other components that are rendered experience any changes) or publish to it to output updated visuals to the simulation. The sound system would mostly be reliant on the flight simulation broadcast as it would only play sounds based on what is currently happening.

Time – The time subsystem determines the passage of time and the speed of the simulation. The user has the ability to choose the values of these components. Time may either passively update to the flight simulation in real-time, or can be adjusted to be sped up or slowed down ("Time," 2023). Either way, this subsystem may rely on changing aspects of the flight simulation based on user requests, or can publish updated time values to the broadcasting system.

Interaction Between Parts:

Since FlightGear's conceptual architecture may be in the publisher-subscriber style, there is potentially very little interaction between the individual components. The components will only interact with the overall simulation when there is an update to a specific variable at any point in time. This form of interaction between components where all variable updates and subscriptions are discrete events strongly aligns with those in general flight simulation systems. More specifically, flight simulations should behave in a way where the entities "only change state when an event occurs" (Allerton, 2022, p. 54). Therefore, FlightGear's components are anticipated to be independent of one another and are only accounted for by the simulation itself to ensure a seamless portrayal of aircraft flight in certain models, atmospheric, and weather conditions.

Control and Data Flow:

The overall simulation is controlled and affected by user requests and selections. Due to this proposed architectural style, the user essentially controls most of the performance of the flight simulation, which in turn controls the performance of other variables present. Additionally, the data is hypothesized to flow asynchronously from one another and it is up to the flight simulation system to account for these changes. Component changes may not be directly dependent on each other but may be related based on probability (e.g. the weather conditions may affect physics and aerodynamics).

Evolution of the System:

The evolution of FlightGear as a system is representative of its commitment to scalability, modifiability, maintainability, and flexibility, attributes that are amplified by its adoption of the publisher-subscriber (pub/sub) architectural pattern. This pattern has fundamentally shaped FlightGear's development, enabling it to manage asynchronous communication between disparate components with remarkable efficiency.

Scalability is at the forefront of FlightGear's architectural advantages. The pub/sub model supports the simulator's growth, accommodating an expanding array of features and user demands without necessitating significant alterations to the core system. This aspect is crucial for a project like FlightGear, which prides itself on a comprehensive and evolving simulation environment that spans a vast range of aircraft models and dynamic weather conditions. The pattern's ability to facilitate the addition of new publishers and subscribers seamlessly aligns with the simulator's goal of a continuously expanding and inclusive aviation universe.

The modifiability and flexibility afforded by the pub/sub pattern are pivotal for FlightGear's development ethos. By decoupling system components, developers can introduce enhancements or integrate new functionalities with minimal disruption to the existing system. This modularity ensures that FlightGear remains at the cutting edge, incorporating the latest advancements in simulation technology and responding adeptly to the community's feedback and contributions. The flexibility to adopt various external systems and data sources further enriches the simulation experience, offering users an immersive and accurate representation of flight dynamics and environmental interactions.

Maintainability, a critical consideration for the longevity of open-source projects, benefits significantly from the pub/sub architecture. The clear delineation of system components and their interactions simplifies the debugging and enhancement process. This streamlined approach to system architecture not only facilitates easier integration of community contributions but also ensures that FlightGear can evolve in response to new challenges and opportunities in the realm of flight simulation.

By integrating the pub/sub model, FlightGear has effectively harnessed the benefits of scalability, modifiability, maintainability, and flexibility, ensuring its position as a premier open-source flight simulator. This architectural choice underscores FlightGear's capability to adapt and grow, reflecting a commitment to providing a rich, dynamic, and accessible simulation platform. The ongoing evolution of FlightGear, powered by the collaborative efforts of its global community and the inherent advantages of the pub/sub architecture, promises to propel the simulator to new heights of realism and technological sophistication.

Concurrency:

Concurrency is the ability of a system to handle different tasks or requests at the same time. Following the publisher-subscriber conceptual architecture, concurrency is a major concern for the FlightGear system, so it must utilize multiple different techniques to solve concurrency-related issues. Due to the large number of publishers that need to constantly update the rest of the system with new information, the software must have some ways of handling these concurrent processes. The primary way that FlightGear does this is through Nasal, a scripting language that was made for programming functionalities in FlightGear (“Multi threaded programming in Nasal,” 2015). Nasal has built-in support for multithreading, which is an extremely useful function for handling concurrency-related issues.

Multithreading is the ability of a processor to handle multiple different tasks, called threads, at a time. Multithreading is essential for a publisher-subscriber architecture due to the fact that the publishers will all need to run concurrent processes, such as calculating trajectory or velocity, to gather information to update the subscriber with. Rather than each process occurring one at a time, multithreading can be used to synchronize them, causing the subscriber to get useful and up-to-date information all at once.

As previously mentioned, this thread synchronization can be created through the use of Nasal, which provides functionalities for mutex locks and semaphores, two of the main ways that multithreading is achieved. By using mutex locks and semaphores, programmers can pause a task until it has access to a resource that another task is currently using, allowing for seamless sharing of resources among threads.

Division of Responsibilities:

Responsibilities among participating developers are divided in a well-organized yet flexible manner, involving skilled volunteers from all over the globe (“Introduction,” n.d.). This global network of contributors brings together a diverse array of talents and expertise, ensuring that each aspect of the simulator—from the core system architecture and software development to hardware integration, testing, and quality assurance—is handled by individuals best suited to the task. By working with developers from different parts of the world, this project benefits from a rich pool of innovative ideas, and specialized skills, allowing the project to grow and improve by combining the best of what everyone has to offer.

At the heart of FlightGear's development process are the 10 core developers (“Developer Portal,” 2023), who possess commit rights to the master FlightGear Git repository. These individuals, including well-known figures like James Turner and Mathias Fröhlich, play a pivotal role in shaping the simulator's core, reviewing and committing patches submitted by the wider developer community (“Howto: Understand the FlightGear development process,” 2014). This core team ensures that modifications to the architecture are seamlessly integrated, maintaining the simulator's integrity and functionality.

Beyond the core development team, FlightGear's development ecosystem is inclusive, embracing a wide range of contributions:

Non-Programming Developers – This group includes contributors focused on the base package development. This diverse group includes artists, sound engineers, and scenario designers, who work on creating realistic aircraft models, immersive environments, engaging soundscapes, and intricate animations. Their contributions are vital in enhancing the simulator's realism and ensuring a comprehensive and authentic flight experience.

Modding Community – Modders play a unique role by customizing FlightGear, allowing enthusiasts and hobbyists to tailor the simulator to their preferences. This aspect of development encourages innovation and personalization, enabling users to modify aircraft, scenery, and even the fundamental behaviour of the simulator through the addition of new features or adjustments to existing ones.

Documentation Contributors – An often underappreciated yet critical component of FlightGear's development is the effort dedicated to documentation and tutorial creation. Contributors who focus on documenting the simulator's features, writing tutorials, and maintaining the project's wiki play a crucial role in supporting both new and experienced users. Their work helps demystify complex aspects of the simulator, making FlightGear more accessible and enjoyable for a wider audience (“Howto: Understand the FlightGear development process,” 2014).

The process of making architectural changes involves community input, detailed planning, and careful implementation. Developers utilize forums, mailing lists, and the bug tracker to discuss potential changes, ensuring that modifications align with the project's goals and community needs. FlightGear's development portal offers a wide array of ongoing projects, plans for upcoming releases, and a list of areas where help is needed, such as maintaining specific flight dynamics models or contributing to the development of the World Scenery 3.0 roadmap. The portal serves as a central point for developers to find areas where they can contribute their skills and expertise (“Developer Portal,” 2023).

External Interfaces

FlightGear utilizes multiple external interfaces for functionalities that cannot be handled by the system alone. In order to generate graphics, the FlightGear system uses OpenGL, a cross-platform application programming interface (API) that assists in rendering graphics and is used by many programs other than FlightGear (“Chapter 2,” 2023). FlightGear uses this API in order to write instructions detailing the graphical environments, objects, and user interfaces that need to be rendered. This information is sent to the graphics drive of the computer, where the driver will communicate instructions to the graphics processing unit, in order to display the necessary graphics to the screen. FlightGear only works with OpenGL, and is not compatible with Direct3D or DirectX, due to those APIs only working on Windows.

Another external interface used by FlightGear is OpenAL, which is an API with very similar functionality to OpenGL, however it assists in creating auditory effects, rather than graphics ("FlightGear," 2024). The process of how FlightGear uses OpenAL is the same as how it uses OpenGL, with the only difference being that the information is sent to the audio driver and the sound card, rather than the graphics driver and the graphical processing unit.

Use Cases

Case 1: Increasing velocity during flight simulation

Assume a user is already running a flight simulation, and they choose to increase the velocity at which their plane is flying. They will make this change through the user interface, which will send information to the aircraft movement subsystem. The aircraft movement subsystem will update the main flight simulator with the new information. After this, multiple different other subsystems will need to update the flight simulation, starting with aerodynamics variables and the navigation system. Both navigation and aerodynamics (the physics subsystem) would be heavily impacted by the aircraft changing speeds, so it makes sense these would require updating the flight simulation. After this, image and sound generation would also need to be updated, due to the aircraft potentially making different sounds at a higher velocity, and because a higher velocity means more environmental objects will need to be rendered. These changes made in the flight simulator will also be reflected in the user interface.

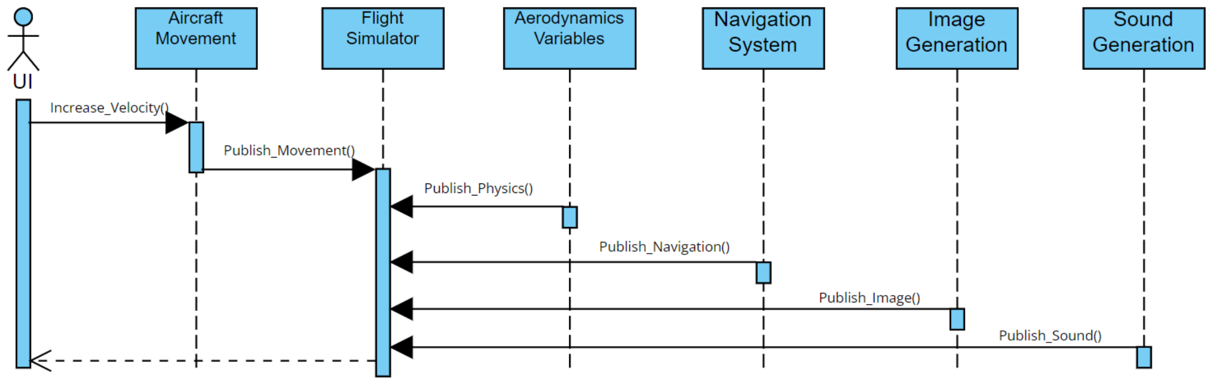

Figure 2: Sequence diagram for the first use case.

Case 2: The user sets the weather to rainy

Assume a user is already running another flight simulation, and they go into the settings and choose to set the weather to rainy, as opposed to sunny. Similar to the previous scenario, this will start by the user interacting with another subsystem through the user interface. By changing the weather setting, the user will directly affect the weather variables subsystem, which will publish new information to the flight simulator. After this, because of the structure of a publisher-subscriber architecture, the other relevant subsystems will also publish updated information to the flight simulator. Aerodynamic variables will be updated to calculate how rain will affect the physics of the flight, aircraft characteristics will need to update the flight simulator on how this particular aircraft reacts to harsh weather conditions, and finally, image and sound generation will publish rendered rain and the sound of precipitation to the flight simulator. After the flight simulator has been entirely updated, similarly to the previous case these changes will be reflected in the user interface.

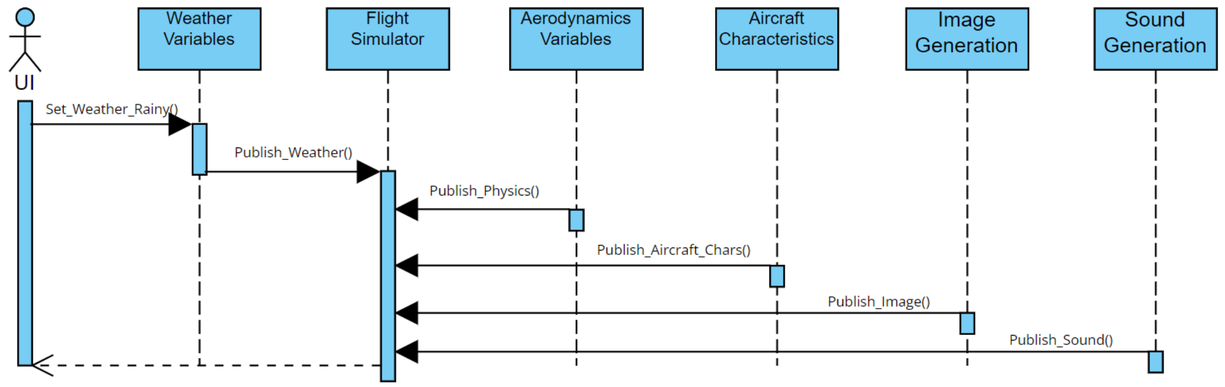

Figure 3: Sequence diagram for the second use case.

Data Dictionary

Adaptability. The capability of adjusting to new conditions.

Aerodynamics. The properties of an object regarding the manner in which air flows around it.

Broadcasting system. The system components can publish or subscribe to this system in a publisher-subscriber/implicit invocation architecture style.

Component. A software object that encapsulates a certain functionality or a set of functionalities, capable of interacting with other software objects.

Concurrency. The ability to execute more than one program or task simultaneously.

Flexibility. The capability of changing to integrate new technology as it comes along.

FlightGear. An open-source flight simulation software made for education and research, recreational use, and virtual flight practice.

Maintainability. The ease of which a system can remain well-supported as time goes on.

Modifiability. The ease of which something can be changed in the overall system.

Modularity. A logical partitioning of components that makes otherwise complex software manageable in terms of implementation and maintenance.

Nasal. A scripting language made for programming functionalities in FlightGear.

Open-source. Denoting software for which the original source code is made freely available and may be redistributed and modified.

Publisher. The origin of broadcasted messages in a publisher-subscriber/implicit invocation architecture style.

Publisher-subscriber. Also known as the "implicit invocation" architecture style, it consists of a loose collection of components that do not communicate with each other individually, but rather take part in the system through a connecting procedure they can subscribe to.

Scalability. The capability to increase or decrease system performance and cost in response to changes in demand.

Subscriber. The recipient of broadcasted messages in a publisher-subscriber/implicit invocation architecture style.

Subsystem. A computer system that is part of a larger computer system.

Naming Conventions

API. API stands for "application programming interface."

GNU. GNU stands for "GNU's Not Unix."

OS. OS stands for "operating system."

OpenAL. OpenAL stands for "Open Audio Library."

OpenGL. OpenGL stands for "Open Graphics Library."

Lessons Learned

Working on the Conceptual Architecture report for FlightGear not only deepened our understanding of flight simulation software but also provided a unique learning experience, combining insights into the complexities of the program with practical lessons in teamwork and research methodology. Our initial approach was somewhat simplistic, underestimating the intricacy of FlightGear's architecture. FlightGear, with its many capabilities ranging from simulating various aircraft models to rendering realistic environmental effects, demonstrated how a flight simulation software's realism hinges on the seamless integration of countless subsystems. This realization was pivotal, teaching us the importance of delving deep into each component's functionality and interdependencies.

One of the more significant challenges we faced was the dynamics of working in a group where members had conflicting schedules, particularly with half the group attending different tutorial times. This situation underscored the importance of flexibility, effective communication, and the use of digital tools to facilitate collaboration. Despite these obstacles, we learned valuable lessons in time management and the crucial role of clear, continuous communication in keeping the project on track, particularly through regular communication over Discord group chat. We divided the work early on, which allowed members to work asynchronously despite the scheduling conflicts. This approach helped us to adapt and plan proactively in our group project, ensuring that everyone remained aligned with the project's goals and progress.

Although we were provided with a solid foundation of resources by our professor, we quickly realized the importance of external research in fully understanding FlightGear's flight simulation architecture. Seeking out additional information from developer forums, and technical documentation to list a few, expanded our knowledge of the simulator's inner workings and its community-driven development model. Going beyond the provided materials, encouraged us to seek out and combine diverse sources of information to fully understand the complexity of FlightGear's architecture.

Conclusion

In conclusion, FlightGear exemplifies a successful integration of the publisher-subscriber architectural style in flight simulation software. Its open-source nature not only fosters an environment of collaboration and development but also significantly contributes to the software's adaptability, scalability, and modularity. This has allowed FlightGear to be more than a mere simulation tool instead it is a versatile platform for education, research, and recreation, continuously evolving to include a wide array of aircraft models, weather conditions, and geographical landscapes.

Through its publisher-subscriber model, FlightGear allows a high degree of interaction between different simulation components without sacrificing the quality or performance of the overall system. This architecture not only facilitates easy integration of new features and models by developers around the globe but also ensures that the system remains strong and responsive to the ever-changing demands of its user base.

Looking forward, the insights gained from this report highlight several pathways for further development and research within the FlightGear community and the broader field of flight simulation. The evolution of FlightGear will likely continue to be shaped by emerging technologies and the ongoing contributions of its diverse developer and user communities. As such, FlightGear not only serves as a model for the successful implementation of the publisher-subscriber architectural style but also as a beacon for future innovations in flight simulation technology.

Works Cited

Aircraft. (2023, Aug. 24). FlightGear Wiki. Retrieved Feb. 18, 2024, from wiki.flightgear.org/Aircraft

Allerton, D. (2022). *Flight Simulation Software : Design, Development and Testing*. John Wiley & Sons Inc.

Allerton, D. (2009). *Principles of Flight Simulation*. John Wiley & Sons Inc.

Chapter 2. (2023, Dec. 14). FlightGear Manual. Retrieved Feb. 17, 2024, from flightgear.sourceforge.net/manual/2020.3/en/getstart-ench2.html#system-requirements

Chapter 5. (2023, Dec. 14). FlightGear Manual. Retrieved Feb. 17, 2024, from flightgear.sourceforge.net/manual/2020.3/en/getstart-ench5.html#inflight-all-about-instruments-keystrokes-and-menus

Chapter 8. (2023, Dec. 14). FlightGear Manual. Retrieved Feb. 17, 2024, from flightgear.sourceforge.net/manual/next/en/ getstart-ench8.html#a-basic-flight-simulator-tutorial

Compositor. (2024, Feb. 7). FlightGear Wiki. Retrieved Feb. 18, 2024, from wiki.flightgear.org/Compositor

Developer Portal. (2023, Aug. 10). FlightGear Wiki. Retrieved Feb. 17, 2024, from wiki.flightgear.org/Portal:Developer

Ebrahimi, A. (2024, Jan.). *Week 3's lecture slides* [Lecture notes]. Queen's University. onq.queensu.ca/d2l/le/content/861602/viewContent/5155034/View.

FlightGear. (2024, Jan. 30). FlightGear Wiki. Retrieved Feb. 17, 2024, from wiki.flightgear.org/FlightGear

Garlan, D., & Shaw, M. (1993). An Introduction to Software Architecture. *Advances in Software Engineering and Knowledge Engineering*, *vol. 1*, pp. 1-39. onq.queensu.ca/d2l/le/content/861602/viewContent/5155035/View.

Howto: Understand the FlightGear development process. (2014, Jun. 13). FlightGear Wiki. Retrieved Feb. 17, 2024, from wiki.flightgear.org/Howto:Understand_the_FlightGear_development_process

Introduction. FlightGear. Retrieved Feb. 17, 2024, from www.flightgear.org/about/

JSBSim Aerodynamics. (2019, Aug. 22). FlightGear Wiki. Retrieved Feb. 18, 2024, from wiki.flightgear.org/JSBSim_Aerodynamics

Multi threaded programming in Nasal. (2015, Apr. 6). FlightGear Wiki. Retrieved Feb. 17, 2024, from wiki.flightgear.org/Multi_threaded_programming_in_Nasal

Time. (2023, Jul. 12). FlightGear Wiki. Retrieved Feb. 17, 2024, from wiki.flightgear.org/Time

Weather. (2023, Jan. 16). FlightGear Wiki. Retrieved Feb. 18, 2024, from wiki.flightgear.org/Weather

YASim. (2023, Jul. 21). FlightGear Wiki. Retrieved Feb. 18, 2024, from wiki.flightgear.org/YASim