Enhancement Proposal for FlightGear

CISC 326/322 Assignment 3 Report
April 12th, 2024

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

Table of Contents

Abstract

This report discusses a potential feature that would greatly improve the user experience of the FlightGear program, the AICopilot model. The idea of the AICopilot is to incorporate a flight assistant in the cockpit with the user, that can provide helpful tips when facing adverse flight conditions, create small talk and teach the user about the environment they are flying in. The AICopilot would support full customizability settings to fit the users needs, from low involvement where the copilot would only engage in gameplay for very urgent circumstances, to very high involvement that can walk the user through every step of a successful flight.

When playing FlightGear for the first time, it can be quite an overwhelming experience just to get the plane to take off, it is a very difficult game. There is a massive window of opportunity to improve the instructional format throughout the game, and while there are currently tip pop-ups that do occur, the AICopilot would make the learning experience far more engaging and realistic as to how a person would learn to fly in various conditions.

After countless brainstorming sessions, we were able to analyze the logistics of adding this feature by performing a SAAM analysis on two different implementation options, bringing potential risks to light, and highlighting the architectural changes that would be required for a seamless integration.

Introduction

FlightGear's co-pilot features are limited to FGCom which is a communication module, a simple nasal announcement script, and dual control feature for multiplayer purposes. These features suffice in providing a co-pilot experience, however, there is no solid external co-pilot in place. The goal of this report is to propose an artificial co-pilot, show how it would be implemented and why it would be useful. The first part of this report involves our proposed enhancement which is an AICopilot that allows for an AI guided flight experience. The proposal and motivation behind it are discussed first, where we explain why we think this would be a good addition to FlightGear. We also provide two possible ways the feature can be realized into the system. As with any piece of software, adding new functionality usually affects other previously existing software. Thus, the next part of the report dives into the effects of adding an AICopilot module in respect of maintainability, testability, performance, and evolvability. The impacts on other parts of FlightGear are discussed as well, how existing features are impacted in terms of interactions and alterations. After considering the interactions and impacts, we give a relative look at the system with respect to the enhancement, explaining what the system looks like with the enhancement in place. Next, we consider the alternatives for realizing the proposed enhancement. There are multiple ways to implement the feature. We take a look at them and their respective architecture styles, comparing them as we introduce other possibilities. Now we have a strong understanding of the architectural style at hand, and how the feature is being implemented. Using this information, the next few sections explain the impact on the HLA and the LLA, and how files and directories are impacted directly. This provides a strong foundation for understanding how the feature is going to impact the system at large. Having a complete understanding of the proposal at this point, at both a high-level and a low-level, plans for testing the module are provided as a crucial part of any software's evolution is testing the new features and making sure they do not impact or collide with other existing features. Finally, two use cases are given to show how a practical use of the module would look. We conclude the report by taking a look back at everything that was discussed and tying the components of the report together into a final conclusion. The conclusion that is drawn is that AICopilot would be a beneficial feature enhancement for FlightGear, but would come with its upsides and downsides.

Motivation for Enhancement

Within flight simulation, the integration of advanced technology is very important in order to enhance user experience and user learning. FlightGear, as a leading open-source flight simulator, has fallen behind in consistently incorporating innovative features, as it seems to be more focused on its legacy features. FlightGear already includes the beginnings of an AI copilot for players to use, however the majority of these copilots are incredibly simplistic, for instance only certain aircrafts implement nasal scripts for copilot announcements and they are limited to announcing when certain speeds or altitudes are hit. Our proposal mainly hinges on providing an airplane agnostic copilot for players to use, with a more expansive list of things it may assist with, and most importantly, be able to provide solutions to these problems in a suitable language for pilots of all skill levels.

The addition of our copilot lowers the barrier to entry for users who don't know the ins and outs of a plane. It will provide an accessible platform for learning and practicing common flight maneuvers to be practiced without the fear of not understanding the complex nature of flight. It's important that the user has real-time feedback for them to learn from. With this in mind, the copilot will provide feedback if it predicts that the user has made a short-sighted error that will impact the flight later on. And with this new technology, FlightGear can be more immersive, interactive, and engaging for new users.

Enhancement proposal

Our proposed enhancement is multifaceted in order to properly support the copilot into the future of flightgear.

Firstly we propose a dedicated information gathering system in order to take in the instrumentation data available to the player normally, possibly including checks to not take in any information from instruments with current failures as can occur in flightgear to enhance realism, as well as access the player's current flight plans. Then the next part entails calculations to try and gain information from the data that a skilled pilot may find from instrumentation (eg. most planes in flightgear don't show how much fuel you lose per hour, but a skilled pilot can intuit or calculate this fairly accurately with regards to whether they can make it to the next airport), these calculations should hopefully primarily come from preexisting functions in the fdm to encourage code reusability, but may also involve some tailor made functions. These will then be compared against the expected values and depending on what settings the player previously set as to what they want the copilot to keep an eye on and the severity of the discrepancy they will either be dropped, or placed in a priority queue to be sent as a single coherent message for the next major component.

The most major part of this proposed enhancement in terms of impact and scale is that of adding a language model (LM) interface for copilot features. The rise of large language models such as Chat-GPT has created a new glut of research around how to use AI in order to communicate complex concepts with people, and whilst implementing something like Chat-GPT 3.5 would be impossible from a performance standpoint, the rise of tiny specialized LMs such as TinyStories[1], and Atlas[2] show that it's very possible to be able to create a small, targeted LM for a specific domain (such as flying a plane) while training on relatively little hardware such as a single cuda compatible gpu, and queried on even less hardware. This allows us to train a LM for FlightGear using sources such as flight manuals and books on general principles for flying, depending on the size of the models themselves we could also specialize them even further for common training planes for beginners such as the Cessna-172, the default plane in flight gear and what most beginners will learn on. As the player wouldn't be talking to the copilot directly and instead they would be just commenting on the semi structured data being passed to them, it may be possible to finetune its performance even further than the models shown, however discussion on optimisation is outside the purview of software architecture. But by adding a LM as an intermediary rather than just telling the player that variables are outside the expected range, the language model can tell the player specifically how to correct these issues as well. This is especially beneficial to beginners as they may not understand what to do when they need to cut down on fuel usage, but being told to lean the fuel mixture by 5% and reduce throttle by 10%, and *why* that helps is much more advantageous. This doesn't just apply to beginners either, as experts can also be assisted for things like more advanced strategies and maneuvers when coming across major issues, for example if in a dogfighting scenario the opponent is detected as right behind the player on radar, the copilot can then advise how to proceed to try and get them off of the players tail.

After the response from the language model is generated it can then be post processed, for instance by adding in controls relating to certain phrases (something like 'pull up' would be additionally tagged to add press 9), and passed along to output for the player. This output may be something such as a simple text box on the player's hud passing through the LM's output with these tags, or a more interesting option may be text to speech. Flightgear already has text to speech through mods such as Red

Griffen's ATC[3] but this is very primitive in it's implementation as can been seen in the following clip, and thus our recommendation would be a new implementation of tts if used, as AI has also caused this to progress significantly in recent years, such as with Piper[4], a Neural network based TTS system which is made for the raspberry pi, and thus unlikely to affect performance.

AICopilot - Risks and Effects

There's many parts of the AICopilot to consider in terms of maintainability, firstly the internal structure should be made as modular as possible in order to make it easier to modify individual parts of the new system without affecting the others, as we're using an object oriented architecture for this, we can achieve this fairly easily via abstract data types and inheritance, such as splitting off the part of Copilot that pulls in the new information from the instrumentation (or perhaps the fdm but we shall discuss that more in performance). While using a modular style like this does mitigate the risk of implementation negatively affecting the entire system, it is still unavoidable that this will have an impact on the maintainability of the entire system as a whole. If any resources that the copilot gets its data from changes format (ie if instead of knots, the instrumentation now does speed in km/hr, or any number of minor changes to how the routing is done in the flight plans). This does make it more annoying to refactor the system, but would likely be the same as changes in most other components and relatively minor so definitely workable.

Many of the parts of copilot would be capable of being integrated into flight gears current testing framework, being able to test basic functionality such as checking that it can pick up details from the property and check they are within specified ranges, and of high enough priority to send a message to the user. However a significant risk of this addition in terms of testability is the fact it requires intensive playtesting to dial in frequency and level of assistance with certain settings to ensure that players get the essential information without being overburdened with it. Playtesting does add a lot of stress on a developer base as opposed to the automated testing systems, but given advanced enough settings for preferences much of this playtesting should hopefully be unnecessary. An additional way to be able to reduce this strain is by also supplying adequate testing environments in shell scripts to launch the game while the plane is already at several different boundary lines for autopilot (eg. 100 feet away from being too high, only 10 knots above stall speed, on the boundary of being outside of the flight plan etc). In terms of the LM as well this is even more severe as much of the testing required for that is impossible via the current automated testing framework of flightgear and automation in terms of training would likely require external tools and specialists to infer the results of testing, while significant frameworks already exist to help with this such as numerous python libraries, which also can provide interoperable models after training, this may cause additional risks in terms of complexity of the system as it involves working outside of the c++/nasal split Flightgear already has.

The property tree in flightgear is already seen as an issue for performance due to the additional layers of indirection it causes, while we wouldn't be using as many property tree calls in comparison with something like canvas which requires thousands of calls per element, it is still something to keep in mind as the property tree is not scalable and overuse can lead to major frame delays if pushed towards capacity, so this will be our main bottleneck in terms of performance, however we can still aim for getting new data to get calculations on every ~5 seconds by spacing the calls by category, this allows us to still maintain a constant flow of calculations to add new information to the speech queue while not overloading the property tree. However beyond all of this is the proposed addition of using a LM could be a large performance drain for many players, particularly on lesser hardware as it may rely on gpu performance that may not be available. An argument could be made that this is not a problem however as it's as an optional feature much in the same way as many graphical fidelity improvements such as anti-aliasing, However as this feature could be a major boon to many would be players who may not have the hardware required a mitigation could be to cache some vitally important things that are high priority, like if the instrumentation indicates the plane is in a tailspin or stalling to improve general performance where it is needed quickly, as well as some of the more beginner oriented common interactions so that those without the necessary hardware can still access the basic copilot tips.

The main risk in terms of evolvability is the fact that this does tie flightgear further into the property tree, and as previously mentioned there are already performance issues surrounding this, restricting other features that may also wish to access the property tree in future, which is most of them. Additionally by tying it further into the property tree this can block much progress that could be made in multithreading as it is one of the major roadblocks with current efforts, but it is unlikely the property tree will be removed any time soon and any overhauls would be unlikely to cause mass rewriting of all features that use the property tree individually so this risk may be moot. In terms of the LM, current progress in the field is extremely rapid and this may be a double edged sword for the

Interactions with Existing Features

From our understanding of FlightGear, features tend to interact with most other feature in the program in one way or another, and the AICopilot is no exception. AICopilot would interact with a multitude of subsystems, specifically: AIModel, Traffic, Instrumentation, FDM, Scenery, Cockpit, GUI, and A/V. The AICopilot will be able to gather information from currently running AI scripted scenarios through the AIManager. Since the AIManager manages each AI scenario, the AICopilot's LM can generate small talk based on the scenarios running in close proximity to the user. For example, if a Boeing model was flying near the user in a scripted scenario, the AICopilot could mention that the pilot scheduled to fly that aircraft on that in game day was their buddy and talk about that pilot, or they could make currently relevant small talk about the controversies surrounding Boeing. The information gathered through the Traffic subsystem would also be used to generate small talk. The information gathered through the Instrumentation subsystem would be used to make both surface level observations about the aircraft's moment to moment position, velocity, fuel level, etc. and inferences that a more skilled pilot could make from the surface level observations. The calculations necessary for these inferences would be accomplished easily by calling functions from the FDM. The connection to the Scenery and FDM would also be used to check at regular intervals of time that the user's aircraft is not on track to crashing into something or going past its maximum altitude for safe flying. The Cockpit, GUI and A/V subsystems will be connected to in order to output the AICopilot's instructions to the user.

Current State of AIModel

Since the core of AICopilot is a black box LM, it will be a rather disconnected enhancement for AIModel. The rest of this section will discuss the components of AIModel with which the AICopilot does connect and will end with a discussion on how the LM part of AICopilot can be expanded with respect to the current state of AIModel. Since an AIManager object is in charge of running each scripted scenario, some functions might be necessary to add to the AIManager to extract the information necessary for the AICopilot. Furthermore, since the Traffic subsystem was grouped in with the AIModel subsystem in our concrete architecture, the TrafficMgr and Schedule components of Traffic would provide more information about surrounding aircrafts and the user aircraft's scheduled flight (such as departure time, departure airport, arrival airport, cruising altitude, call sign, and flight rules for the chosen flight path) respectively.

As discussed in the proposal section, the biggest addition to the AIModel that this enhancement would provide is the LM. Since the current AIModel does not have any infrastructure for an LM, that will need to be the main focus for developers at the start of the implementation of the enhancement. Once the proposed enhancement is implemented, the LM could potentially be integrated further into AIModel by training it to create radio chatter from AI aircrafts in scenarios near the user. Currently, the AICopilot would not have any need to create vehicle objects of any kind. So it won't have any dependencies on the following components of AIModel: AIAircraft, AIBallistics, AIWingman, AIBase, AIShip, AIEscort, AIGroundVehicle, AICarrier, AISwiftAircraft, AIBaseAircraft, AITanker

Alternatives Considered

With the proposed AICopilot enhancement, as it is meant to monitor events and deviations in real-time flight, the utilization of a Pub-Sub architecture was considered as a potential style. Under this framework, the relevant data would be acquired, such as flight plans and instrumentation readings from their respective components which would then be published with both event/deviation detection and text-to-speech synthesis being subscribed to it.

The use of a Pub-Sub architecture could streamline communication, resulting in enhanced efficiency, between the copilot and the user, potentially avoiding having to rely on the Property Tree. This style would offer greater responsiveness, flexibility and scalability, appearing in many ways to be more efficient.

However, having the enhancement use a different architecture from the repository style of FlightGear could have some negative consequences. Due to its strengths, Pub-Sub is inherently more complex and this in turn means that integrating this architecture into the current system could be challenging for the developers.

Even if a Pub-Sub architecture style is implemented, FlightGear already has the Property Tree, a vital part of the flight simulator. This could be disrupted and developers may need to substantially change established code to ensure seamless integration with the rest of the system, leading to an increased workload and a greater project complexity for developers.

Furthermore, Pub-Sub could also introduce additional overhead and latency issues. Real-time flight simulations require optimal system performance, maintainability and minimal latency for a true immersive experience so an introduction of Pub-Sub must be carefully managed to reduce any potential disruptions. FlightGear is already a complex and unclear system so while Pub-Sub offers certain advantages, developers must prioritize ensuring the best possible experience for users.

SAAM Analysis

NFRs	Implementation 1 - Use a LM or a database of AI scripted scenarios to output instructions in the form of speech in order of the most pressing issue to the least.	Implementation 2 - Use a database of AI scripted scenarios to create text boxes containing the instructions that are in a pinned list on the screen in order of importance.	Comparison
Maintainability	Pros: If a LM is used to generate the instructions, then only a light amount of consistent maintenance will be required to ensure the instructions are still valid. Otherwise, the logic for choosing which instructions to play at a given moment should stay consistent. So, adding new logic should not be too complex. The infrastructure for playing the sounds would not need much maintenance once it is set up. Cons: If a LM is not used and changes are made anywhere in the game’s physics or scenery, each possible instruction given by the AICopilot that makes reference to the changed aspect will have to be listened to in order to see if any changes need to be made.	Pros: The notes could easily be modifiable since they can simply be implemented as text files. Maintaining the layout of the messages on screen would not be difficult either, since the flexible canvas system can be employed to create these graphics. If done by someone who has previously worked with the canvas system for FlightGear, it should not be much trouble. Cons: Since the notes are posted in the cockpit or through the GUI, maintenance of this implementation would extend to the files in charge of those aspects as well which means a heavier workload for the developers.	Given the pros and cons for both options, Implementation 2 would most likely lead to a longer maintenance process where each step is relatively straight forward, while Implementation 1 would most likely lead to a shorter maintenance process if the developers keep an up-to-date table of what each sound file says.

	Listening to each of these instruction files could significantly increase the developers' work load, especially with the disorderly state of FlightGear's documentation. Stakeholder Impact: *Developers*: Would need to consistently maintain the database of instructions and update it when changes are made. *Users*: New users benefit the most from the AICopilot feature's maintenance.	Stakeholder Impact: *Developers*: They would need to consistently maintain the database of instructions and update it when changes are made. Would potentially also have to maintain the graphical interface used to post the notes. *Users*: New users benefit the most from the AICopilot feature's maintenance.	
Evolvability	Pros: The first version of the system could simply be used to help prevent disasters from occurring during flight, and from there the system could evolve to increasingly introduce flight terminology to increase player immersion. Cons: If an LM is used, the developers will need to consistently check to make sure the AICopilot is not acquiring any strange or unwanted biases; hence evolving into an AICopilot that isn't very helpful. Stakeholder Impact: *Developers*: Will need to keep checking in on the LM and/or updating the database regularly. *Users*: Will see that the feature is getting consistent tweaks which could increase their interest in FlightGear.	Pros: The first version of the system could simply be used to help prevent disasters from occurring during flight, and from there the system could evolve to increasingly introduce flight terminology to increase player immersion. Cons: There isn't room for much non-stylistic evolvability of the enhancement (other than the above) if we take this route for the implementation. Stylistic evolution isn't bad, but it isn't very substantial. Stakeholder Impact: *Developers*: Will be in charge of updating the database regularly. *Users*: Will benefit from further immersion as the system evolves.	A similar path of evolution could be taken for both implementations but the downsides to implementation 1 essentially mean the developers will have to keep a close eye on the LM to ensure it isn't hallucinating or saying strange things during the development period, which outweighs the scope related downside of implementation 2. However, if things go well, the first implementation could offer much more.
Testability	Pros: Automated testing could be done to test that each sound file says what it is supposed to, this might take some extra external code but is definitely do-able.	Pros: Automated testing could be done to test that each text file says what it is supposed to and appears correctly on the user's screen, this might take some	Testing could be a frustrating endeavor for both developers and users in the case of

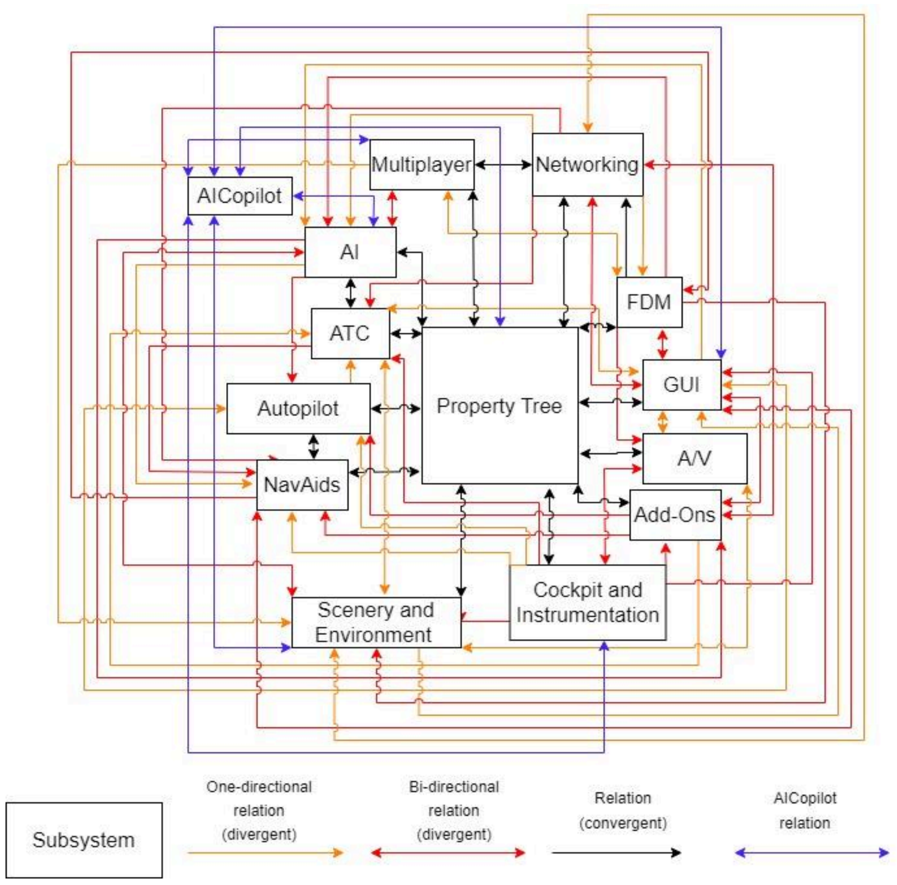
	<u>Cons</u>: Testing each scenario in game will take a long time. Since FlightGear is open source, it won't be possible to simply spend money on hiring testers. However, since there are so many different scenarios to test, there could be a version released such that users who want to help test the new feature and report bugs can do so. This could help lighten the load. That being said, different users might feel differently about the same sound clip, with some thinking it is clear enough and other thinking it isn't clear enough. <u>Stakeholder Impact</u>: *Developers*: Will have to spend lots of time doing the bulk of the testing. *Users*: Could help playtest the new feature.	extra external code but is definitely do-able. <u>Cons</u>: Testing each scenario in game will take a long time. Since FlightGear is open source, it won't be possible to simply spend money on hiring testers. However, since there are so many different scenarios to test, there could be a version released such that users who want to help test the new feature and report bugs can do so. This could help lighten the load. <u>Stakeholder Impact</u>: *Developers*: Will have to spend lots of time doing the bulk of the testing. *Users*: Could help playtest the new feature.	implementation 1. Since testing when sound is involved is a little trickier than simply testing when text is involved, Implementation 2 would probably be favorable testability-wise.
Reliability	<u>Pros</u>: Instructions are placed in a priority queue so that the most pressing issues are dealt with first. <u>Cons</u>: If the AICopilot's instructions are given via sound, the player could misunderstand or be confused about the instructions. An LM could also pose issues if hallucinations occur. <u>Stakeholder Impact</u>: *Developers*: Must ensure the reliability of the information given to the user by the AICopilot. *Users*: Will benefit from this feature if reliable, and will most likely avoid it if not.	<u>Pros</u>: Instructions are placed in a priority queue so that the most pressing issues are dealt with first. This would mean the instructions with the highest priority get placed at the top of the pinned list. <u>Cons</u>: If too many things are going wrong, the user could experience information overload which could worsen the user's situation and turn them away from the feature. <u>Stakeholder Impact</u>: *Developers*: Must ensure the visuals do not get too crowded and the user can reliably receive and understand the instructions. *Users*: Will benefit from this	Both implementations would have a similar backend if and LM is not used for the first implementation, so any reliability issues would be dealt with in similar ways for both implementations. The possibility of hallucinations makes implementation 1 slightly less appealing than implementation 2.

		feature if reliable, and will most likely avoid it if not.	
Accessibility	Pros: In situations where the user must act quickly to prevent a crash, this option would allow them to receive the information while still looking at the scenery leading to a faster reaction. Cons: This option would not be accessible to players who are experiencing hearing loss or who cannot hear at all. A possible solution to this would be to have subtitles, but that would negate the pro above. Stakeholder Impact: *Developers*: Will have to find a way to make this option accessible to users while not overloading them with information. *Users*: Will have a better time using this feature if developers find a solution to the problems above. Otherwise, some might not be able to enjoy it.	Pros: This option is accessible to players who are experiencing hearing loss or who cannot hear at all. Cons: It might be harder for users to look away from the scenery and read the notes if their aircraft is in a dire situation. Stakeholder Impact: *Developers*: Will have to find a way to make this option accessible to users while not overloading them with information. *Users*: Will be able to use this feature if they are experiencing hearing loss or cannot hear at all.	Accessibility-wise, implementation 1 would be beneficial over implementation 2 to users who experience loss of vision and implementation 2 would be more beneficial than implementation 1 for users who experience hearing loss. Haptic feedback could also be implemented as a solution to these issues, but it is outside of the scope of this project.

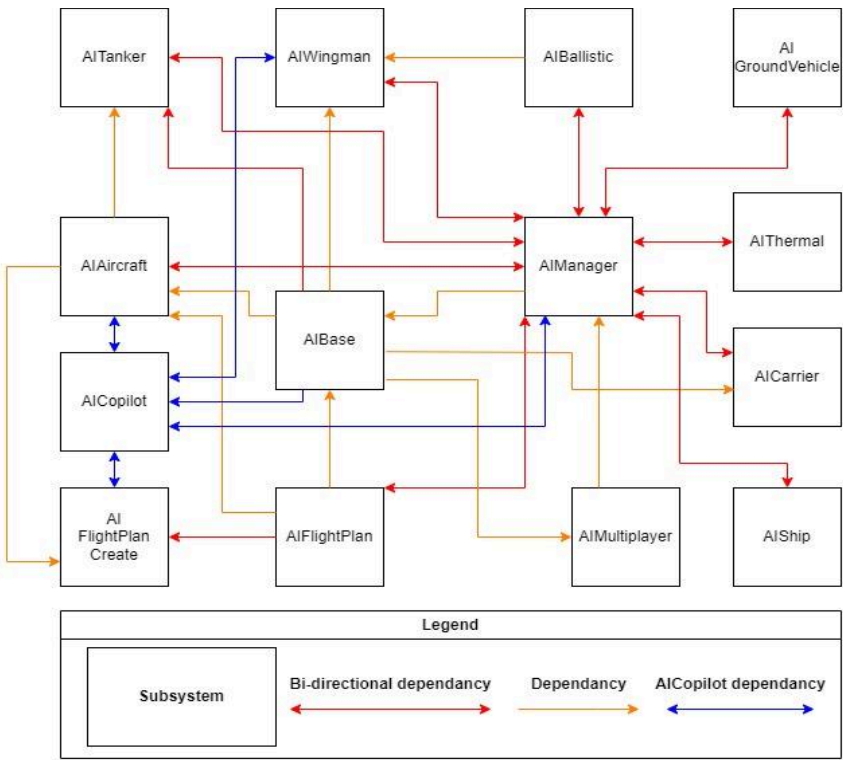
Conclusion to SAAM Analysis:

Given the information in the table above, we've decided that the best option to choose to implement is a mix of the two implementations. Since we want to make use of a LM, we propose that the LM would create the speech and text for each instruction and that both be output to the player in the fashions described at the top of the table. That way, the feature will be as accessible as possible within the scope of the current project. This will take lots of work to deal with hallucinations, but they should be apparent through the text when they happen. Testability-wise, this solution will also take lots of work, but the project is not working within a tight schedule, so this should be feasible, especially with users play-testing. This solution offers maximal avenues for evolvability, and eases the maintaining of the enhancement with proper documentation.

Impact on High And Low Level Architectures



High-level:
As discussed in the Interactions With Existing Features section, the impact AICopilot would have on the high-level conceptual architecture is relatively minimal. Bi-directional dependencies to AIModel, Scenery, Cockpit, Multiplayer and GUI would need to be implemented for successful functionality throughout the rest of FlightGear.



Low-level - AIModel:
Throughout the low-level conceptual architecture of AIModel, bi-directional dependencies to AIAircraft, AIFlightPlanCreate, AIManager and AIWingman would need to be configured, as well as a one-way dependency from AIBase to AICopilot.

Impact on Directories

The implementation of the new autopilot interface will require changes within FlightGear's directories. The primary directory impacted is AIModel; within this folder, there may need to be a new subfolder created called AICopilot, which might house scripts designed for the LM to interface with. The current implementation of copilot within FlightGear is only within specific aircrafts, and it will not help with the new implementation of AICopilot. These new scripts will control the interaction between the LM and the simulation environment; this includes the new text-to-speech interface and the API calls to the LM.

A directory that will most likely be impacted is the Aircraft directory. The aircraft directory has ways of benchmarking performance within the AircraftPerformance file; this will be important information for the Copilot to know when informing the user's decisions. The autopilot will also be able to access files related to the aircraft's position through the FDM directory, so it can access things like altitude, speed etc. The Environment directory will also be helpful to Copilot, as it will be able to inform the user's decisions based on things like current climate and wind.

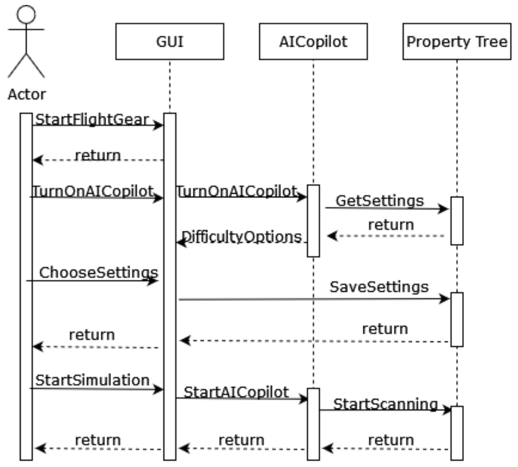
Plans For Testing Against Other Existing Features

To test the correct functionality of AICopilot against the existing features within FlightGear, there will need to be test cases written that verify that all attributes of an object being passed to or from AICopilot are being inherited correctly. For example, there would need to be a test within the AICopilot test cases that checks if the objects within AICopilot are allocating attributes needed from Scenery, as well as the opposite, that Scenery test cases are updated to verify it's receiving AICopilot object attributes if needed. Writing test cases that check that all inheritances between objects are working correctly allows the developers to notice any gaps in the software where they may have missed a connection between subsystems. It would also be critical to run performance tests of the software before and after adding AICopilot to ensure the dialogue or any other backend calculations it generates is not slowing down interactions between subsystems it is not related to. This would involve both time-based test classes, and extensive gameplay testing by the developers as it would be possible that performance decreases may occur that may not be directly visible by just test methods.

Use Cases

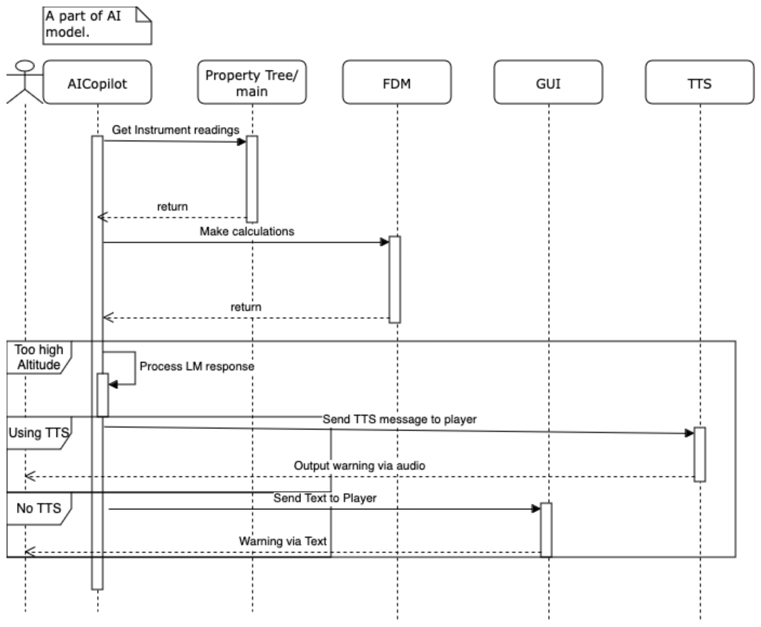
Use case 1 - Setting up AICopilot

The actor may or may not choose to have their AICopilot active, but if they do wish for aid, the AICopilot must first be set up specifically to the user's taste. For example, the actor must choose their level of ability ranging from the AICopilot providing minimal help to providing maximum help.



Use case 2 - AICopilot detects Error and Outputs Tips

The actor in this case does not need to do anything for the system to function, instead the system constantly works in the back and talks to the actor as needed. For example, imagine the user is flying the plane too close to the max altitude limit, the AICopilot will warn the user. Here is the sequence diagram:



Data Dictionary and Naming Conventions

- HLA - High-level architecture
- LLA- Low-level architecture
- LM - Language Model

Conclusion

In conclusion, the proposed AICopilot enhancement would be a significant improvement to FlightGear's current copiloting abilities. This feature provides users a more interactive AI-guided flight experience, improving the immersiveness of the simulation. Throughout this report, we have outlined the motivation for our proposal of the AICopilot, explored various implementations and analyzed its potential impacts on the FlightGear system.

While having the AICopilot could introduce many benefits such as improved user understanding, it also presents some challenges with maintainability, testability and performance. The integration of the AICopilot must be carefully considered as it interacts with multiple existing components and FlightGear's established architecture and minimal disruption is a must to ensure that the users' experiences are satisfactory.

Despite these obstacles, the potential advantages of AICopilot would far outweigh the negatives, if done correctly. With proper implementation and testing, the enhancement would greatly benefit FlightGear. By providing users with greater understanding and aid for their flights, the AICopilot would further solidify FlightGear as the leading flight simulation platform.

Lessons Learned

Similar to the previous two assignments, this one has taught us several valuable lessons in both respects of teamwork/group work and the process of enhancing old software. In regards to technical lessons learned, the first thing we took aways is the importance of providing a strong motivation for improving software. Adding new functionality to almost any large scale software is a demanding task, thus, it must properly be motivated and have strong reasons to be implemented due to the large amount of work it takes. Things like architecture styles have to be refined, stakeholders and their demands have to be considered, impacts on the system must be studied and testing needs to be thorough. In terms of our group work ethic and planning, we had the same issues as the previous assignment. Our final take away for future projects with large groups would be to enforce strict deadlines, something similar to sprints. This would ensure that people are not doing all their work last minute and no person with dependencies on another person's work is stuck waiting for days on end. Lastly, we would also benefit from scheduling work sessions instead of general meetings to hold each other accountable to work on the project.

References

1. Eldan, R., & Li, Y. (2023). TinyStories: How Small Can Language Models Be and Still Speak Coherent English? *ArXiv, abs/2305.07759*.
2. Izacard, G., Lewis, P., Lomeli, M., Hosseini, L., Petroni, F., Schick, T., Yu, J.A., Joulin, A., Riedel, S., & Grave, E. (2022). Few-shot Learning with Retrieval Augmented Language Models. *ArXiv, abs/2208.03299*.
3. https://sourceforge.net/projects/red-griffin-atc/
4. https://github.com/rhasspy/piper?tab=readme-ov-file
5. Conceptual Architecture in FlightGear