A NEW ARCHITECTURE FOR FLIGHTGEAR FLIGHT SIMULATOR

AJ MacLeod, Ampere K. Hardraade, Michael Koehne, Steve Knoblock

MVC architecture,
FDM Server,
FDM Instance,
Client

To continue improving existing features and add new ones, FlightGear must make better use of computing power. Preparing for the widespread adoption of multicore CPU architectures is an important step in FlightGear's development. Today, CPU clock rate has reached its peak. The old idea, that features can be added without regard to their effect on performance because computers will become ever faster, has ceased to hold. In addition, as more features are added, developers are inceasingly bumping up against existing limitations in the current FlightGear architecture. Now would be a good time to begin the process of restructuring FlightGear to address the above issues. This proposal decribes a new architecture for FlightGear, one which would greatly improve FlightGear's efficiency and flexibility by making extensive use of parallel processing. It is also hope that this new architecture will improve the quality of multiuser sessions, as well as providing a true support for the simulations of time-critical systems.

ABSTRACT

To continue improving existing features and add new ones, FlightGear must make better use of computing power. Preparing for the widespread adoption of multicore CPU architectures is an important step in FlightGear's development. Today, CPU clock rate has reached its peak. The old idea, that features can be added without regard to their effect on performance because computers will become ever faster, has ceased to hold. In addition, as more features are added, developers are inceasingly bumping up against existing limitations in the current FlightGear architecture. Now would be a good time to begin the process of restructuring FlightGear to address the above issues. This proposal decribes a new architecture for FlightGear, one which would greatly improve FlightGear's efficiency and flexibility by making extensive use of parallel processing. It is also hope that this new architecture will improve the quality of multiuser sessions, as well as providing a true support for the simulations of time-critical systems.

INTRODUCTION

Since its inception, Flightgear has proved nothing short of a revolution in flight simulation. For perhaps the first time, a serious flight simulator has been available which allows a vast (and theoretically virtually unlimited) degree of flexibility and extensibility. For instance, several FDMs are supported and the output data, both aeronautical and graphical, can be transmitted and displayed in a remarkable number of ways. Such flexibility and extensibility allow FlightGear to have a following in the academic and aero-engineering world.

FlightGear has also found popularity among hobbyists. As such, the multiuser implementation and the AI traffic system have begun to progress into a very usable and important enhancement to the simulator. In fact, the multiuser feature proves to be so popular that it gave an unprecedent boost in FlightGear's user base.

These developments and the simultaneous developments in the abilities of commonly available computing hardware have however begun to highlight some fundamental problems in the current structure of the Flightgear simulator. As computing hardware places more and more emphasis on multiple cores and the threading necessary to take advantage of these, the current "main loop" of FlightGear begins to pose severe limitations on the growth of FlightGear. This can only be addressed by a restructuring FlightGear, which, although certainly far from trivial, will have massive benefits in many different areas.

This report discusses both the current problem areas and a proposed solution, and highlights some of the benefits which will follow on from such a move.

BACKGROUND

FlightGear is an open source flight simulator originally developed as a single user application capable of running on multiple platforms. The architecture of FlightGear is based on an infinite loop called the “mainloop”. Unlike the mainloop of typical event-driven programs, the mainloop in FlightGear is not a loop which runs in idle polling for the user's input. In FlightGear, many components work in the background regardless of inputs. Thus, the mainloop in FlightGear is responsible for periodically telling these components to update themselves. The mainloop was established at the beginning of the FlightGear project; however, recent work such as the expansion of multiuser functionality and sophisticated NASAL scripting have raised some questions regarding the current architecture of FlightGear.

Currently, the following tasks are done sequentially in the mainloop (see Figure 1):

- ATC simulation

- control of AI objects
- update of other aircrafts in a multiuser environment
- flight dynamics calculations
- scenery update
- audio scheduling
- rendering

As the loop is run, the control of the program is passed to different sections of the simulator for handling different tasks. This control is not returned to the mainloop until the task is finished. To elaborate, the FDM cannot be run while the renderer is updating the scene; NASAL scripts cannot be run before other components finish running; the entire sim is frozen while a script is running. Consequently, the performance of each component greatly affects all others.

The most resource intensive process in FlightGear is rendering. A rendering call is made for every iteration of the mainloop. Since control is passed to the renderer and is not returned until the renderer finishes with the scene update, the frequency at which the mainloop can run is slaved to the framerate. It would therefore be more correct to say that the loop is run once for every rendering update. This in turn affects the frequency at which other components can run.

Since framerate is a function of scene complexity and not a constant value, all other components in the simulator are updated at irregular intervals. The autopilot is one of the systems that is most sensitive to the fluctuations caused by the framerate (see Figure 2). When the framerate is low, the autopilot will overcompensate for any attitude change. This overcompensation leads to a rapid attitude change, which is in turn overcompensated by the autopilot. The result is a positive feedback ending with the autopilot plunging the aircraft into the ground. Should the autopilot be replaced with codes that drive a full flight simulator, low framerate situations would result in such unpredictable behaviors that not only could cause injury to the occupants, it might also shake the simulator apart. As a result, FlightGear is undesirable for use as a serious flight simulator, and defeats one of the primary goals of the project.

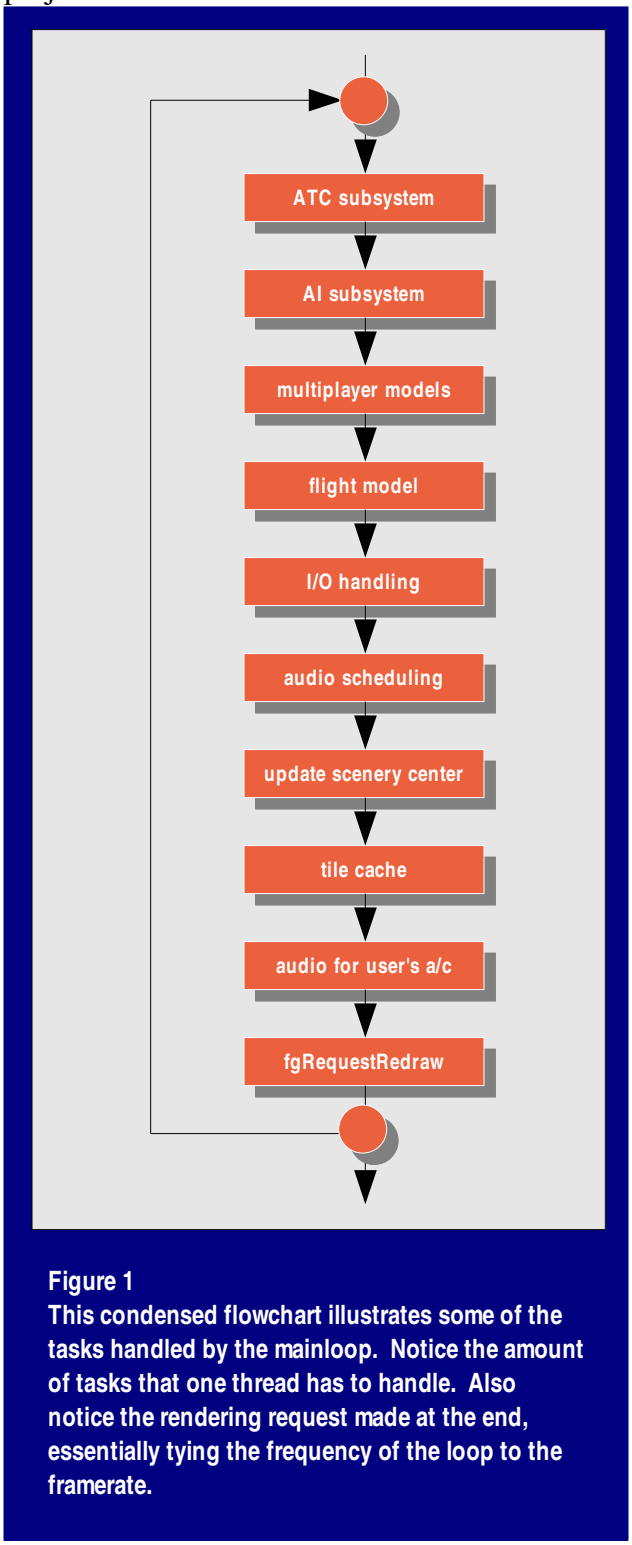

Figure 1
This condensed flowchart illustrates some of the tasks handled by the mainloop. Notice the amount of tasks that one thread has to handle. Also notice the rendering request made at the end, essentially tying the frequency of the loop to the framerate.

Scripts within FlightGear are not immune to the irregular update interval either. Theoretically, a NASAL script should be able to run at a frequency of 100Hz easily. However, due

to the fact that every component in the simulator is tied to the framerate, the practical frequency at which a script can run is at the frequency of scene updates.

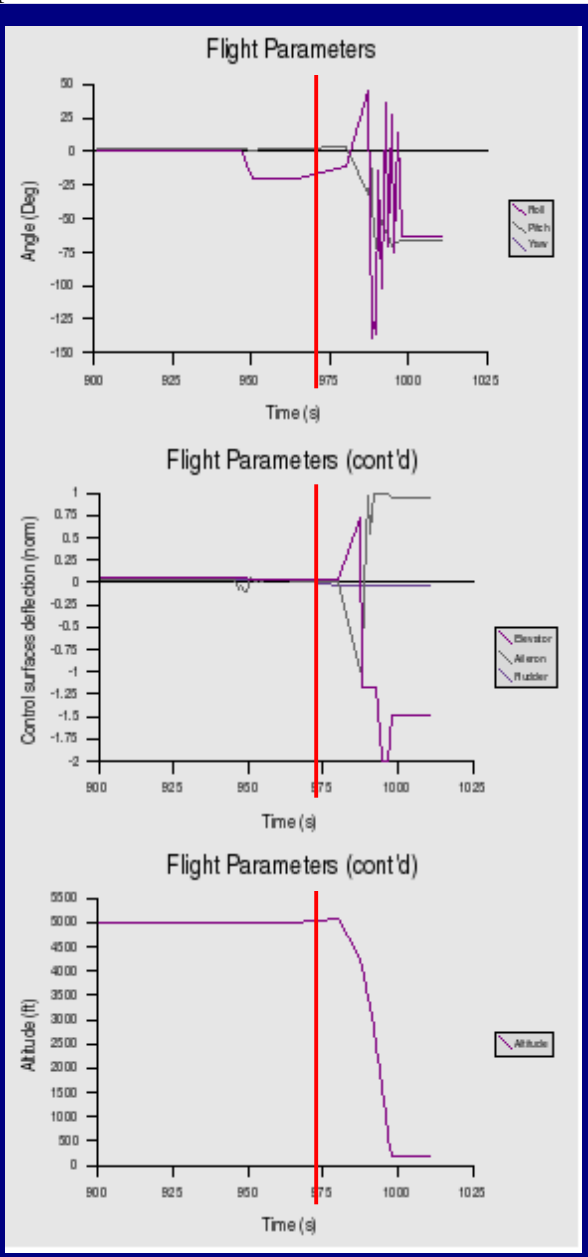

Figure 2
Three graphs showing the effects of framerate hit on the autopilot. The red line indicates the time of framerate drop.

With all components controlled from within one loop, FlightGear is essentially a single thread application. The present configuration makes it impossible to control the performance of individual components. For example, the framerate cannot be throttled, because doing so would affect the performance of the FDM engine as well. Moreover, in order to make everything run smoothly, the loop must run as quickly as possible, leading to a high loading on a single processor.

Another consequence of being effectively single threaded is all but one CPU sits idle on multi-processor platforms, and available resources that can improve FlightGear's performance are not utilized. This problem would only become more prominent in the future, as CPU manufacturers have shifted their priority from increase CPU clockcycle to developing multicore processors[1]. As more features are added and the mainloop becoming ever larger, the current architecture of FlightGear would only cause the performance of future versions to decrease rather than increase.

It should also be pointed out that FlightGear is not currently oriented toward multiuser capabilities, and it can be seen from the way moveable objects in the simulator are handled. In FlightGear, the user's aircraft, AI models, and aircrafts of other users are all considered as separated classes. Feature implementation would require a lot of redundant coding, as each feature would need to be duplicated and implemented differently for different objects types. The current multiuser system also exhibits many limitations. For instance, the inner workings of AI models do not allow AI models to be updated at more than 3Hz[2], thus aircrafts in the multiuser environment suffer noticeable jitters. Limitations also come in the form of constraints on what can be implemented for multiuser. Sophisticated features such as multi-pilot capability are extremely difficult, if not impractical to add because of jitters and latency involved with the current system.

METHODOLOGY

In designing this solution, the foremost concern is shielding the simulator from the effects of fluctuating framerates. For FlightGear

to be a serious flight simulator in the truest sense, the core components must be run in a time-critical manner. Therefore, one of the objectives of this project is to remove the rendering process from the mainloop and place it in a separate thread. Focus was also placed on restructuring FlightGear from a single-user application to a fully network-aware program, so as to benefit the development of multiuser and other complex simulator arrangements.

Emphasis was also placed on ensuring packets to be transferred at well regulated intervals and unifying how objects are handled within FlightGear. The design should incorporate support for desirable features (some reserved for a future time), such as the ability for users to switch aircraft while the simulator is running, showing the same instance of an AI object across the entire network or the implementation of multi-pilot capabilities.

PROPOSED SOLUTION

Overview

This document proposes solutions to the various obstacles standing in the way of further FlightGear development. Specifically, the existing code base of FlightGear should be refactored using MVC (Model-View-Controller) architecture[3], which is a software architecture in which the data model, user interface, and control logic are separated into distinct components. Under this scheme, FlightGear would be refactored into two primary independent components, one being an FDM server hosting the data model, the other a client handling the viewing and controlling components. After the refactoring process, the FlightGear's architecture would bear similarities to that of the X-window system[4].

Without going into too much detail, the following is a brief description introducing the FDM server and client. The purpose of the FDM server is to provide services to simulate the flight dynamics and systems of multiple aircrafts; in essence, a stripped down version of FlightGear, without the graphics-related components. The client is a controller/viewer without any simulation components, which would have two responsibilities – to process inputs from the user and forward them to the server, and to listen to the servers and display objects in the scene accordingly.

The FDM server and client would communicate property changes through UDP ports. Typically, the server and client would be hosted on the same machine (see Figure 3), although they could also be hosted on different machines (see Figure 4). Inputs from the user are first processed by the clients, and then sent to the server. The server would then make adjustments to the FDM based on the information it has received from the client.

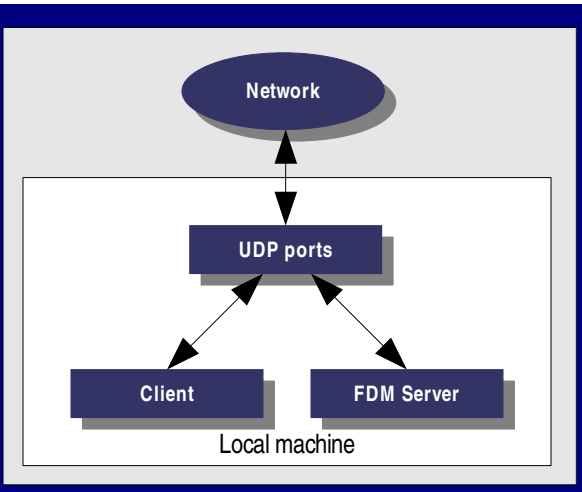

Figure 3
FDM Server and Client being hosted on the same machine.

To reduce the chance of pilot-induced-oscillation, the frequency at which the client uploads the user's inputs must be higher than the typical human reaction time of 0.25s. A suggested uplink frequency would be above 20Hz for the setup where the server and the client are hosted on different computers, but this frequency could be higher when the server and the clients are being hosted on the same computer. The frequency of downlink – the frequency at which the server broadcasts updates would be dependent

on the aircraft being flown, although for most aircrafts, this value would be 10Hz.

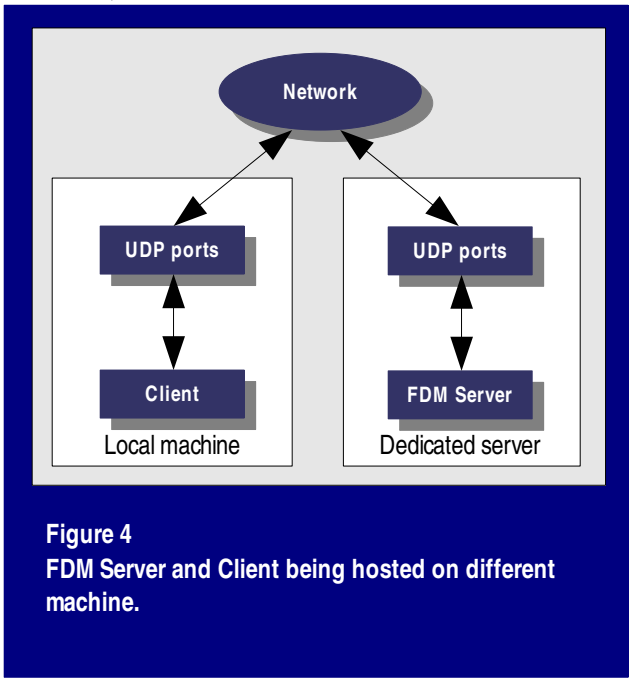

Figure 4
FDM Server and Client being hosted on different machine.

FDM Server

The FDM server is a dedicated component mainly for providing flight dynamics simulation services to the clients. Its responsibility would include handling communication between the clients and FDM, as well as communicating the position of various aircrafts to a MP server. As a server, it would be capable of running multiple FDMs simultaneously, and if necessary, merge or average the outputs of these FDMs.

The actual FDM would be encapsulated in a component called the FDM instance within the server (see Figure 5). This FDM instance (see Figure 6) is the "model" portion in the MVC architecture, and each would have an independent property tree hosting inputs, outputs, environmental variables, information about the aircraft being flown, as well as information about the clients that are related to the FDM being run. Multiple instances could be run on the server so that services could be provided to multiple clients. Since the instances are completely independent from one another, partitioning could also be achieved. For example, a corrupted property tree from one instance would not pose problems to other instances.

Each FDM instance would carry a unique ID, which would serve two purposes: to associate the correct AI model in the viewer with the FDM, and to identify the aircraft being simulated in a multiuser environment. The ID of two instances could be swapped so as to allow the flight dynamics of an aircraft flown by the client to change. This feature could be used to allow the user to change aircraft while the simulator is running, or to change the handling characteristics of an aircraft for simulating the effects from structural failures.

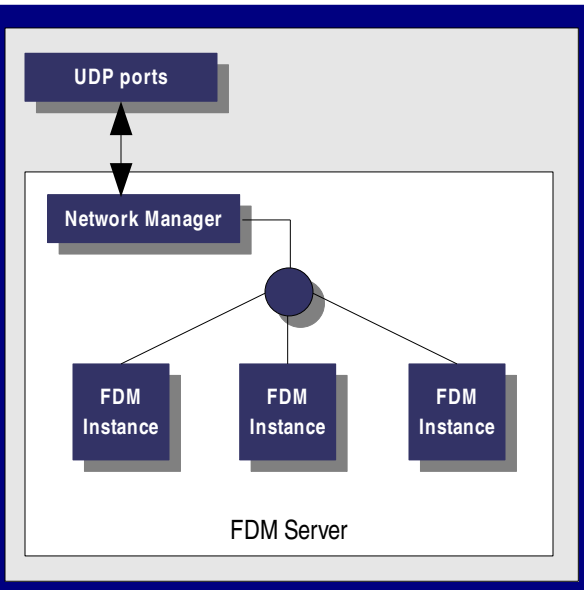

Figure 5
The FDM Server would be capable of running multiple FDM Instances. The circle and lines represent communication channels between various threads.

The FDM server could be dedicated to hosting AI objects such as the carrier, which would allow all users to see the same instance of the AI objects in a multiuser environment. In addition, the FDM server could provide a very unique feature: planes that are not being used could be represented by null FDM's, with an external "AI server" updating their positions. An airliner flown by a user could be turn-around and send back to its home airport automatically. Finally, hosting FDM's on a dedicated server could allow damages of aircrafts to persist across

multiple sessions. To elaborate, suppose that a pilot landed a plane too hard, the landing gear of this plane could collapse during its next landing in a different session.

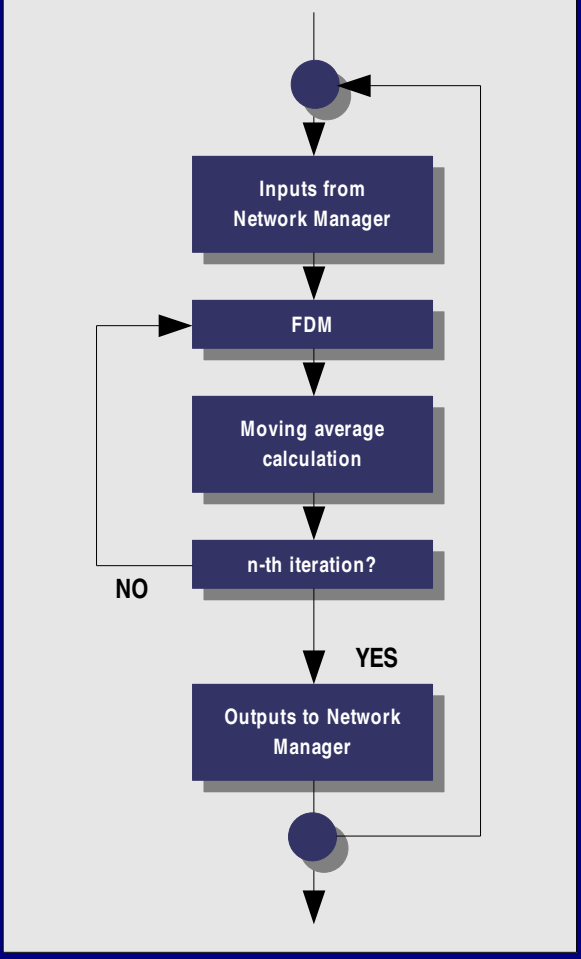

Figure 6
A conceptual work flow of the FDM Instance. The purpose of the "moving average calculation" is to filter out fluctuations from the FDM, so as to reduce the amount of jitter observed in multiuser mode.

Robustness of FDM Server

Since the server would be providing services to many users simultaneously, the server must be very robust and capable of surviving as well as recovering from multiple failures.

Due to the amount of uncertainty involved, one cannot trust the robustness of the code encapsulated by an FDM instance. In other words, an assumption has to be made that the encapsulated code would eventually pose problems. One method of mitigating such problems is to avoid them.

Let's assume that a memory leak occurs in the code being run by one instance. Repeatedly creating and destroying this instance would result in pointers being created and not subsequently freed. Eventually memory would run out, and the FDM server would likely be killed by the operating system. This would not only affect the client hosted by the particular instance, it would also affect all clients whose FDMs are being hosted on the server. Therefore, to contain the effects of memory leaks, only a fixed number of instances would be initialized for any session of the server. In other words, no FDM instance would be created or destroyed while the server is running. In such a scheme, the mistake of creating a pointer and not freeing it would only be made once instead of multiple times. The unused FDM instances could lie dormant and be skipped by the server.

Of course, errors are impossible to avoid completely. When an unhandled exception occurs, the FDM server must be able to recover from it: specifically, methods would be needed to contain the errors and prevent them from migrating outward from the encapsulated instance. One example of a mitigation technique would be to restart an errornous component from its last error-free state. When multiple errors have occurred, the entire FDM instance could be restarted. Likewise, if an FDM instance has an error, the FDM server could restart the instance and let it continue from the last error-free state. After multiple failures, the problematic FDM instance could be shutdown permanently.

The FDM server would also need the capability to transfer the FDMs it is running to another server, for many reasons. A client might want to switch to another server because of a better connection, or the server may have encountered too many errors, and is no longer

suitable for providing services. Whatever the reason, the changeover should not be noticeable at the client end.

Client

The client (also known as viewer/controller in MVC terminology) is the portion with which the user would interact. The client would provide visual and audio cues to the user and manage communications between the user and the FDM server.

The client would require two threaded loops, one handling I/O and another rendering. Leaving aside the I/O handler for the moment, one can simply view the client as an "observer" that displays aircrafts and objects currently existing in the multiuser "world". The point of this design is to unified into a single object framework, with the user's aircraft, the AI aircrafts and other user's aircrafts in the multiuser environment represented as a single class of objects. This framework would create the potential for sub-classing of any specialized moveable object types, which will inherit shared characteristics from the parent class.

The I/O loop handles the communication between client and server. In addition, the I/O loop would also manage property changes, as well as performing audio scheduling. The rendering loop would be dedicated solely to scene updates. Having two loops would avoid interference to data transfer posed by fluctuating framerates. With the renderer within its own thread, framerate throttling could also be implemented. A hard limit could be set (using the sleep() function), which would maintain the framerate below a certain peak value. As an alternative, a "soft limit" could be provided, which would vary the framerate by adjusting the complexity of the rendered scene.

Separating and encapsulating all visual rendering into a "viewer" allows for multiple cameras. Cameras could be assigned different views into the FlightGear environment or assigned to follow a number of particular aircrafts, thus opening up the possibility of enabling multiple users to inhabit a single aircraft and operate its controls (see Figure 7). This latter feature is critical to realistic simulation of certain aircraft operations, such as the B-29 or Clipper. Moreover, the capability to have multiple viewers interacting in a multiuser environment would allow FlightGear to be used as an online teaching tool. An instructor may fly along with the student pilot, by flying within the same aircraft (copilot's seat view).

Although it may be something best left to the future, the new architecture would make it possible for each client to utilize a different graphic engine.

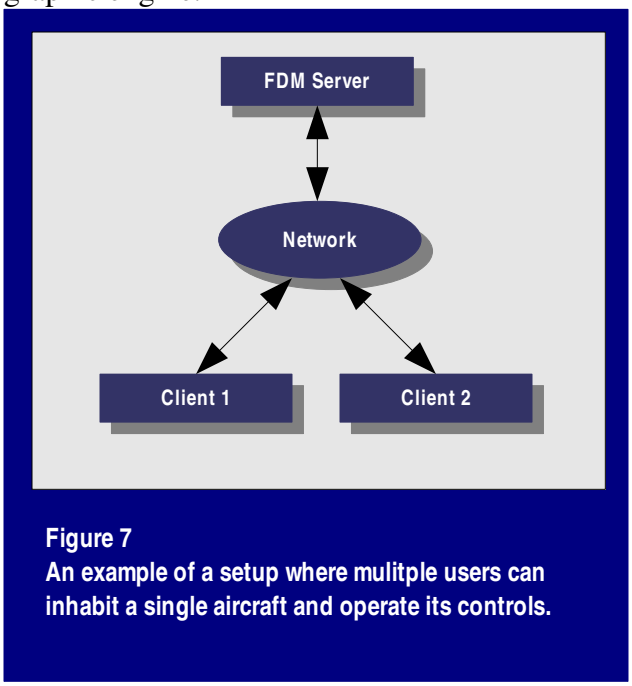

Figure 7
An example of a setup where mulitple users can inhabit a single aircraft and operate its controls.

SUMMARY

FlightGear is experiencing a rapid proliferation of sophisticated features. However, the current architecture of FlightGear prevents the simulator from utilizing available resources on multi-processors platforms, as well as on future multicore CPUs. In addition, the framerate has influence on every aspect of FlightGear, causing the outputs of the simulator to be unpredictable and making simulations of time-critical systems an impossibility.

The proposed new architecutre would address these problems through extensive use of parallel processing. By moving to a Model-View-Controller architecture, code division within FlightGear would become more clear and better organized. Flexibility would also increase, since through networking, many new features could be "added" without having to physically be placed inside of FlightGear's source code. Fewer changes to the FlightGear's source code would also mean that the simulator would be less error prone.

The multiuser experience would also be enhanced, as very little effort would be needed to utilize the new architecture to allow multiple users to control a single aircraft. AI simulation could be hosted from a central location on the network, ensuring the same AI objects are displayed in all clients that are connected. In the future, ATC and weather systems could be hosted in the same manner, ensuring all users fly in a consistent weather and air traffic environment.

FlightGear has clearly demonstrated how well an Open Source Flight Simulator can work. The past decade of development has allowed us to gain much knowledge and experience in the area of flight simulation. Post-1.0 release would be an ideal time to make use of the knowledge gained to provide a step change in the flexibility and opportunities offered by FlightGear.

REFERENCES

1. Herb Sutter (2005). The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Retreived March 19, 2006.
   Website:
   http://www.gotw.ca/publications/concurrency-ddj.htm
2. pigeond (2006). A list of videos captured from FlightGear's multiuser sessions. Retreived March 19, 2006.
   Website:
   http://www.pigeond.net/photos/FlightGear/videos/
3. Wikipedia (2006). Model View Controller. Retreived March 19, 2006.
   Website:
   http://en.wikipedia.org/wiki/Model_view_controller
4. Wikipedia (2006). X Window System protocols and architecture. Retreived March 19, 2006.
   Website:
   http://en.wikipedia.org/wiki/X_Window_System_protocols_and_architecture