The FlightGear Flight Simulator

Alexander R. Perry
PAMurray, San Diego, CA
alex.perry@flightgear.org
http://www.flightgear.org/

Abstract

The open source flight simulator *FlightGear* is developed from contributions by many talented people around the world. The main focus is a desire to 'do things right' and to minimize short cuts. FlightGear has become more configurable and flexible in recent years making for a huge improvement in the user's overall experience. This overview discusses the project, recent advances, some of the new opportunities and newer applications.

Introduction

The open source flight simulator *FlightGear* has come a long way since first being showcased at LinuxWorld in San Jose.

In April 1996, David Murr proposed a new flight simulator developed by volunteers over the Internet. This flight simulator was to be distributed free of charge via the Internet and similar networks. Curt Olson made a multi-platform release of FlightGear[1] in July 1997.

Since then, it has expanded beyond flight aerodynamics by improving graphics, adding a shaded sky with sun, moon and stars correctly drawn, automatically generated worldwide scenery, clouds and fog, head up display and instrument panel, electronic navigation systems, airports and runways, network play, and much more.

Recent changes to the simulator have simplified the customization of those features by the user, as discussed in more detail below. Instead of being in the source code, the configuration data is now specified on the command line and/or accessible using menu items and/or loaded from simple files.

Simulator Portability

FlightGear aims to be portable across many different processors and operating systems, as well as scale upwards from commodity computers. The source has to be clean with respect to address width and endianness, two issues which in-

convenience most open source projects, in order to to run on Intel x86, AMD64, PowerPC and Sparc processors.

In addition to running the simulation of the aircraft in real time, the application must also use whatever peripherals are available to deliver an immersive cockpit environment to the aircraft pilot. Those peripherals, such as sound through speakers, are accessed through operating system services whose implementation may be equivalent, yet very different, under the various operating systems. FlightGear currently supports Windows, FreeBSD, Linux, Solaris, MacOS, Irix and OS-X.

For those services which are common across most video games, the independent project *PLIB* offers a simple API that acts as a **P**ortable **Lib**rary[2]. Compared to Windows, MacOS and the Unix's, the various distributions and releases of Linux-based operating systems are very similar. There are important differences, most of which cause problems when trying to build and test *PLIB*, so these rarely impact FlightGear directly. Once the facilities required by video games are available through the Portable Library, the remainder of the code acts as a conventional application.

Any Linux user can download the source, compile it and safely expect it to run. FlightGear, and other applications with extensive 3D visual effects as in figure 1, may use different libraries under the same Linux distribution so that the binary may not be portable between computers. Some hardware only has accelerated 3D under XFree86 version 3, other hardware requires version 4, and their GLX APIs differ. This is being addressed by the *Linux OpenGL Application Binary Interface* (ABI)[4] but continues to be a source of frustration for new would-be users.

Once the issues associated with running the simulation program on a specific computer are resolved, everything else is portable. The configuration information can be freely transferred across processors and operating systems. Each installation can benefit from the broad range of scenery, aircraft models, scenarios and other adaptations that have been made available by any user.

Figure 1: Pilot's view from a Cessna 172 making a poor approach for landing at San Francisco International Airport. The "Time of Day" pop up menu is partially visible, with seven quick selects and an additional button for typing in the time. Seven consecutive screen dumps, from a wide screen display, have been combined to demonstrate the simulator's lighting model

Simulator Structure

Unlike proprietary commercial PC flight simulators, the Open Source nature of the FlightGear project permits modification and enhancement. Since the project wishes to encourage its user community to embrace, extend and improve the simulation, the FlightGear source offers a flexible framework.

The FlightGear source tree is only one level deep, except that all the flight data models are each in their own subdirectory located under the FDM directory. Each directory contains a few header files that expose its object definitions. Other source files refer to the headers directly, without the frustrations of path globbing or multiple include paths for the compiler. The directory names are mostly pretty self-explanatory:

AIModel, Aircraft, Airports, ATC, Autopilot, Cockpit, Controls (in the aircraft cockpit), Environment, FDM (only one constituent is in use), GUI (menus and the like), Include (for compiler configuration), Input (joysticks etc), Instrumentation, Main (initialization and command line parsing), Model (3D aircraft), MultiPlayer, Navaids, Network (data sharing), Objects (dynamically loaded and changing scenery), Replay, Scenery (static), Scripting (remote script control), Server, Sound, Systems and Time (in the simulated world).

FlightGear exposes the internal state of the simulation

through the property database, part of the generic simulation infrastructure offered by *SimGear*[6]. This dynamically maps a name (such as /position/latitude) into an object with getter and setter methods. If the property will be accessed frequently, the pointer to the object can be stored so that the string lookup of the name only occurs one time. The single pointer indirection is still somewhat slower than hard linkage, but enhances the modularity of the code base. For example, the user interface definitions in XML files (panels, instruments, 3D animation, sound, etc) are able to refer to any property offered by any subsystem.

The simulator state is also accessible on the network. Adding the command line option --telnet=5555 allows another computer (such as the instructor console) to interact with the simulator computer using a command such as telnet simulator 5555. This network interface allows any property in the database to be viewed or modified and includes remote enumeration of all the property names. This has been used to implement a complete external operator GUI and for automating FAA certification tests.

Many tasks within the simulator need to only run periodically. These are typically tasks that calculate values that don't change significantly in 1/60th of a second, but instead change noticeably on the order of seconds, minutes, or hours. Running these tasks every iteration would needless degrade per-

Figure 2: Panoramic scenery, configured by Curt Olson

formance. Instead, we would like to spread these out over time to minimize the impact they might have on frame rates, and minimize the chance of pauses and hesitations.

We do this using the *Event Manager and Scheduler*, which consists of two parts. The first part is simply a heap of registered events along with any management information associated with that event. The second part is a run queue. When events are triggered, they are placed in the run queue. The system executes only one pending event per iteration in order to balance the load. The manager also acquires statistics about event execution such as the total time spent running this event, the quickest run, the slowest run, and the total number of times run. We can output the list of events along with these statistics in order to determine if any of them are consuming an excessive amount of time, or if there is any chance that a particular event could run slow enough to be responsible for a perceived hesitation or pause in the flow of the simulation.

FlightGear itself supports threads for parallel execution, which distributes tasks such as scenery management over many iterations, but some of the library dependencies are not themselves thread clean. Thus all accesses to non-thread-clean libraries (such as loading a 3D model) need to be made by a single thread, so that other activities wishing to use the same library must be delayed. These small delays can lead to minor user-visible hesitations.

Simulator Execution

FlightGear is packaged by all major distributions and most others too, so that installation of pre-built binaries can usually be completed in a few minutes. Almost all customization, including adding new aircraft and scenery, occurs through XML and through a structured directory tree. Most users can now simply use the prepackaged distributed binaries and no longer need to recompile the simulator to add new features.

However, this is a rapidly changing project and new functionality is added to the source code on an continuing basis. If a user wishes to take advantage of a new capability or function, when customizing the simulation for their individual needs, that source version does of course need to be compiled.

Installing and running FlightGear is relatively easy under Linux, especially compared to other operating systems with weak tool automation.

1. Install Linux normally and test Internet access.

2. Add video card support, using a maximum of 25% of memory for the 2D display, as 3D uses the remainder.

3. Enable hardware accelerated OpenGL support and test for speed, using *glTron*[5] for example.

4. Install[2] *PLIB* 1.8 or above, which is already in many distributions, and test with all the supplied examples to ensure all the API features are working.

5. Verify that headers for zlib and similar are present.

6. Download[6], compile and install *SimGear*.

7. While that compiles, download the *FlightGear* source.

8. With *SimGear* installed, compile and install *FlightGear*

9. While compiling, download FlightGear's base package. This contains data files that are required at runtime.

10. Type `runfgfs` and enjoy.

Starting from a blank hard drive with no operating system, FlightGear can be running in less than an hour.

Simulating the Pilot's view

The new FlightGear pilot will probably not want to remain within the San Francisco bay area, which is the small scenery patch included in the Base package. The scenery server allows the selection and retrieval of any region of the world. Joining other users in the sky is another possibility.

Due to limited monitor size, the view that is available on a normal computer is a poor substitute for the wraparound windows of general aviation aircraft. This is especially true when the simulated aircraft has an open cockpit and an unrestricted view in almost all directions.

FlightGear can make use of multiple monitors to provide a nicer external view, possibly even wrap around, without special cabling. The additional computers and monitors need not be dedicated to this purpose. Once the command lines and fields of view (relative to the pilot) for each of the additional computers have been established, the main computer will make the necessary data available irrespective of whether those other computers are actually running FlightGear. In consequence, each of the additional computers can change from a 'cockpit window' to a office software workstation (for someone else) and, when available again, rejoin the FlightGear simulation session.
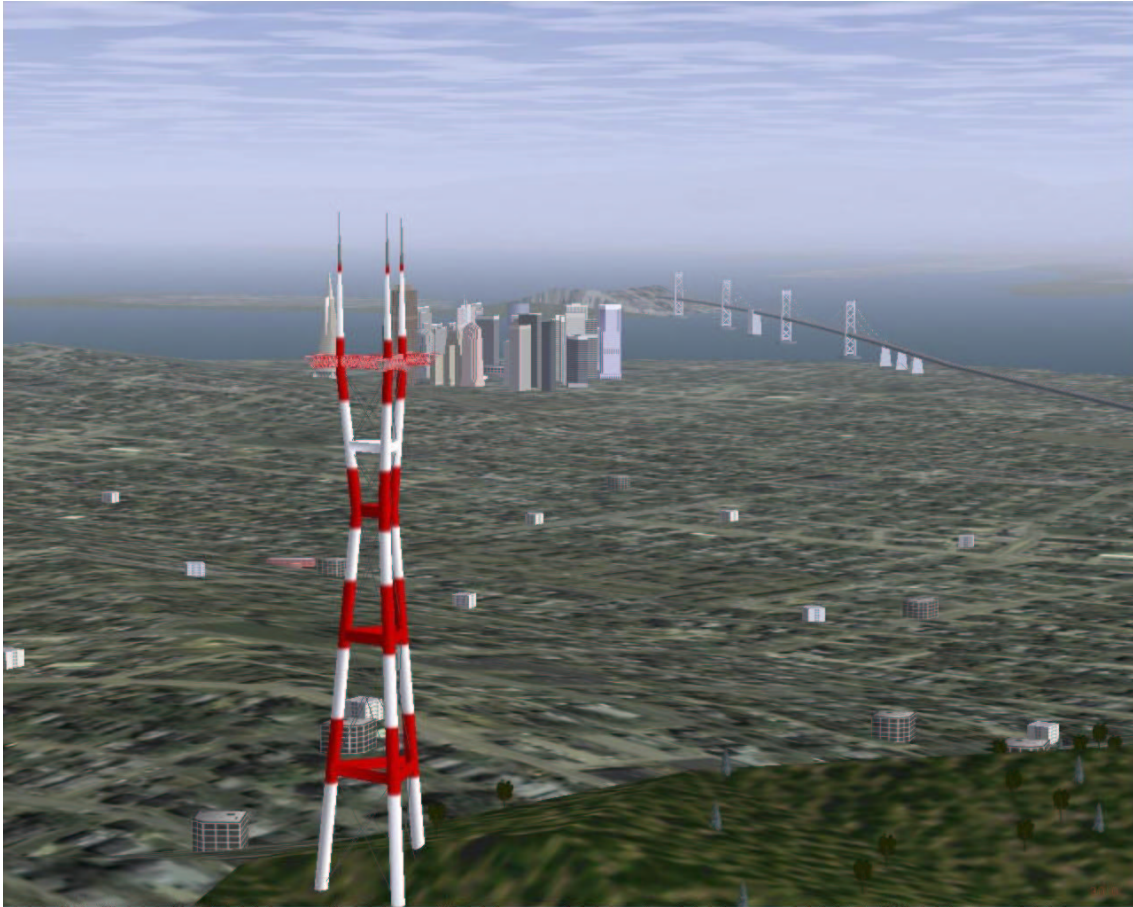
Figure 3: Looking east from Mount Sutro, just south of Golden Gate Park. The scenery shows the shoreline, variations in land use from scrubland on the hill and urban beyond, randomly placed trees in the scrubland and buildings in the urban area, custom created additional objects for the tower, downtown buildings and the Bay Bridge, and an interstate freeway. The sky shows the limited visibility due to haze and some scattered clouds.

FlightGear has built in support for network socket communication and the display synchronizing is built on top of this support. FlightGear also supports a null or do-nothing flight model which expects the flight model parameters to be updated somewhere else in the code. Combining these two features allows you to synchronize displays.

Here is how Curt Olson set up the example in figure 2:

1. Configure three near identical computers and monitors.

2. Pick one of the computers (i.e. the center channel) to be the master. The left and right will be slaves *s1* and *s2.*

3. When you start `runfgfs` on the master, use the command line options
   `--native=socket,out,60,s1,5500,udp`
   `--native=socket,out,60,s2,5500,udp`
   respectively to specify that we are sending the "native" protocol out of a udp socket channel at 60 Hz, to a slave machine on port 5500.

4. On each slave computer, the command line option
   `--native=socket,in,60,,5500,udp` shows that we expect to receive the native protocol via a udp socket on port 5500. The option `--fdm=external` tells the slave not to run it's own flight model math, but instead receive the values from an "external" source.

5. You need to ensure that the field of view on the scenery matches the apparent size of the monitor to the pilot. `--fov=xx.x` allows you to specify the field of view in degrees on each computer display individually.

6. `--view-offset=xx.x` allows you to specify the view offset direction in degrees. For instance,
   `--view-offset=0` for the center channel,
   `--view-offset=-50` for slave 1, and
   `--view-offset=50` for slave 2.

There is no built in limit to the number of slaves you may have. It wouldn't be too hard to implement a full 360° wrap around display using 6 computers and 6 projectors, each covering 60° field of view on a cylindrical projection screen. Ideally, the master computer should be chosen to be whichever visual channel has the lightest graphical workload. This might be the dedicated instrument panel, for example. If the master computer has a heavy graphical workload, the other channels will usually lag one frame behind. Select the graphics realism parameters to ensure that all the visual channels consistently achieve a solid and consistent frame rate ($30\,Hz$ for example) and, if your video card supports it, lock the buffer swaps to the vertical refresh signal.

For optimal results, make sure the FOV each display subtends, matches the actual FOV that display covers from the pilot's perspective. From the top view, draw a line from the pilot's eye to each edge of the display. Measure the angle and use that for your visual channel configuration. Draw a line from the eye to the center of the display. Measure the angle from that line to the dead center straight ahead line. Use that angle for the view offset. This ensures that all objects in your simulator will be exactly life size.

Simulating the Aircraft

The aerodynamic simulation may be only one constituent of the whole environment being simulated for the user, but its performance is critical to the quality of the user's simulation experience. Errors in this *Flight Dynamics Model* (FDM) are distracting to the pilot. Other simulator components, such as the autopilot, are designed to expect a realistic aircraft, may respond incorrectly as a result of FDM errors and provide additional pilot distractions. These factors can ruin the immersive experience that the user is seeking.

As a result of this concern, FlightGear abstracts all of the code that implements an FDM behind an object oriented interface. As future applications find that existing FDM choices do not meet their requirements, additional FDM code can be added to the project without impacting the consistent performance of existing applications.

The original FlightGear FDM was LaRCsim, originally modeling only a Navion, which currently models a Cessna 172 using dedicated C source that has the necessary coefficients hard coded. It is sufficient for most flight situations that a passenger would choose to experience in a real aircraft. Unusual maneuvers that are often intentionally performed for training purposes are poorly modeled, including deep stalls, incipient and developed spins and steep turns. The code also supports a Navion and a Cherokee, to a similar quality.

A research group at the University of Illinois created a derivative of LaRCsim, with simplified the models such that they are only really useful for cruise flight regimes. They enhanced the code with a parametric capability, such that a configuration file could be selected at simulation start to determine how the aircraft will fly. Their use for this modification was to investigate the effect on aircraft handling of progressive accumulations of ice.

Another group is developing a completely parametric FDM code base, where all the information is retrieved from XML format files. Their JSBSim project[7] can run independently of a full environmental simulation, to examine aerodynamic handling and other behavior. An abstraction layer links the object environment of FlightGear to the object collection of JSBSim to provide an integrated system. Among many others, this FDM supports the Cessna 172 and the X-15 (a experimental hypersonic rocket propelled research vehicle), providing the contrast between an aircraft used for teaching student pilots and an aircraft that could only be flown by trained test pilots.

An additional FDM code base, *YASim*, generates reasonable and flyable models for aircraft from very limited infor-

mation. This is especially valuable when a new aircraft is being added into FlightGear. Initially, there is often insufficient public data for a fully parametric model. This FDM allows the aircraft to be made available to the user community, thereby encouraging its users to find sources of additional data that will improve the model quality.

The rest of FlightGear's configuration files are now also XML, such as the engine models, the instrument panel layouts and instrument design, the HUD layout, the user preferences and saved state. The real benefit of using XML here is that people with no software development experience can easily and effectively contribute. Pilots, instructors, maintenance technicians and researchers each have in-depth technical knowledge of how a specific subsystem of an aircraft, and hence the simulator, should behave. It is critical that we allow them direct access to the internals that define their respective subsystem, without burdening them with having to dig through other subsystem data to get there. We have made huge progress on achieving this in the last two years.

Simulating the Cockpit

In order to simulate the cockpit environment around the pilot, additional information is included from subsidiary XML files. These include a 3D model of the cockpit interior, associations of keyboard keys to panel switches, the instrument panel layout and position, visual representations of the individual instruments, mappings between joystick channels and flight controls, parametric descriptions of the head up display elements, a 3D model of the aircraft exterior, animation of moving aircraft surfaces and other reference data.

In the same way that aircraft manufacturers reuse much of their designs and instruments across product lines, many XML files are included by multiple aircraft models. Once loaded, this information is integrated into the simulation.

For example, one can look into the cockpit from outside, as shown in figure 4, and see the live instrument panel indications. This is the same exact instrument panel being shown to the pilot when inside the cockpit, demonstrating that FlightGear has a fully working 3D animated cockpit that is visible from inside or out.

Some aircraft types have 'glass cockpit' displays, as shown in figure 7. The independent project *OpenGC*[8] is a multi-platform, multi-simulator, open-source tool for developing and implementing high quality glass cockpit displays for simulated flightdecks. Due to the resolution and size limitations of computer monitors, the OpenGC images are best on a separate monitor from FlightGear.

The head up display of a real aircraft uses computer generated graphics, so the software can generally detect, and correct for, the flaws and inaccuracies in the sensors that are feeding it data. As a result, the information presented to the pilot is generally accurate. Simulating that is relatively easy,
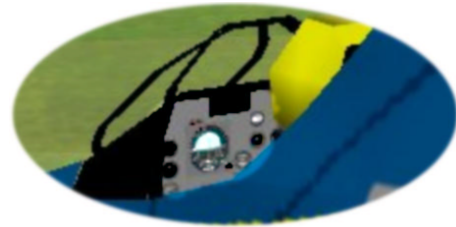

Figure 4: Closeup of an A4 with the animated cockpit interior

since the actual state of the aircraft (in the properties) can be retrieved and directly displayed.

An important aspect of learning to fly an aircraft (without computer assistance) is understanding what the limitations and errors of the various instruments are, and when their indications can be trusted as useful flight data. Unfortunately, the information from panel instruments has errors, which in general only read a single sensor value with negligible correction for the limitations of the sensors being used. When the FlightGear panel advanced from no errors to having only two of the limitations implemented (VSI lag and compass turning errors), the non-pilot developers went from trivially flying instrument approaches to frequent ground impacts. Many more limitations have been realistically implemented since.

Considerable effort is needed to write this code. Gyroscopes can slow down and wobble, their rotation axis can drift, they can hit gimbal stops and tumble and their power source can be weak or fail. Air-based instruments are wrong in certain weather conditions, tend not to respond immediately, can be blocked by rainwater in the lines, or become unusable when iced over. Radio navigation is subject to line-of-sight, signals bounce off hills and bend near lake shores or where another aircraft is in the way and distant stations can interfere. Still more errors are associated with the magnetic compass, and other instruments that seem 'trivial'.

Currently, the communication radios are not implemented, so that pilots cannot use their microphone inputs to interact.

Radio usage is a large part of the complexity in operating at large and busy airports. Unfortunately, this often encourages pilots to fly the microphone and forget about the airplane, occasionally with disastrous results. We hope to implement this feature soon, to provide another source of challenging distractions to the pilot.

Although voice communication between pilots is not yet supported, there is an artificial intelligence (AI) subsystem that seeks to make the airspace feel less empty. This subsystem moves other aircraft around the sky as a source of distraction, issues ATC style instructions and responses, and ATIS messages. Somewhat surprisingly, everywhere in the world, the ATIS always has a British accent.

Simulating the World

The purpose of the TerraGear project[9] is to develop opensource tools and rendering libraries and collect free data for building 3D representations (or maps) of the earth for use in real time rendering projects. There is much freely available *Geographic Information System* (GIS) data on the Internet. Because the core data for FlightGear has to be unrestricted, the default use of the project only uses source data that doesn't impose restrictions on derivative works. Three categories of data are used.

Digital Elevation Model (DEM) data is typically a set of elevation points on a regular grid. Currently, 30 arcsecond (about $1\,km = 0.6\,mi$) data for the whole world, and 3 arcsecond $(90\,m)$ data from the Shuttle Radar Topography Mission (SRTM) for the United States and Eurasia, is available from the U.S. Geological Survey (USGS). Although the SRTM data was originally recorded in February 2000, the signal processing by the Jet Propulsion Laboratory (JPL) is being continuously improved. The recent releases with even finer grids, including 1 arcsecond, would offer much better resolution of landscape features but still suffer from artifacts around large buildings. Future improvements in these data sources are hoped for.

Irrespective of which data source is selected for a given area, an optimizing algorithm seeks to find the smallest number of flat triangles that provide a fairly smooth and realistic terrain contour. This algorithm reduces the number of triangles need to render an area while preserving all the detail within some specified error tolerance.

Other more specialized data such as airport beacon, lighthouse locations, radio transmission towers and the like are available in listings from various government agencies. These generally provide a short text description of the item and its geographic coordinates. The challenge is to convert each entry into a realistic visual object which can be inserted into the scenery database.

Polygonal data such as landmass outlines, lakes, islands, ponds, urban areas, glaciers, land use and vegetation are available from the USGS and other sources. Unfortunately,

the land use data is many years old and thus may not be current with a pilot's local real world knowledge and this is not expected to change in the near future. The GSHHS database provides a highly detailed and accurate global land mass data so we can model precise coast lines for the entire world. Based on the source of the data and factoring in the land use data, we can select an appropriate texture which will be painted onto the individual triangles. Where necessary, triangles are subdivided to get the effect correct. Runways and taxiways are generated by converting the list of runway segments into polygons, painted with appropriate surface texture and markings, and then integrated into the scenery in the same way.

Clearly, someone can gain access to data sources that are under more restrictive licenses, use the TerraGear project tools to generate enhanced scenery and then distribute those files as they choose. Both the FlightGear and TerraGear projects encourage this kind of enhancement, because the basic open source packages cannot do this.

There is a trade-off between the quality of the scenery and the speed at which it can be rendered by the graphics card. As cards get faster, it becomes feasible to place more detail into the scenery while maintaining a useful and smooth visual effect. There are many techniques for adjusting the level of detail according to the altitude and attitude of the aircraft, to optimize the visual quality, but none of them are currently implemented as they cause visual artifacts.

The scenery system adds trees, buildings, lights (at night) and other objects. The choice of object, as well as the coverage density, is determined by the land use data. These objects are relatively slow to display in large quantities, so the user must trade off the reduction in display responsiveness against the improved cues for height, speed and underlying terrain. Like other settings, this property may be adjusted in real time. Figure 3 shows randomly placed buildings and trees, with a maximum range of about half way to downtown San Francisco, together with the manually placed downtown area and bay bridge.

Airports have runways and taxiways that are automatically created from the best available information to ensure that their locations, dimensions and markings correspond to real life. This is not trivial, since there are dozens of different levels of painting complexity in use. This information is also used to determine the pattern of lighting, if any - since some airports have no lights, that is shown at night and during twilight. Some lights are directional, multicolored, flashing or are defined to have a specific relative brightness. Figure 1 shows the lights and markings for KSFO.

Currently, the visual effect is clearly synthetic, as can be seen in figures 3 and 4, but it has sufficient information to readily permit navigation by pilotage (i.e. comparing the view out of the window to a chart). The compressed data requires about one kilobyte per square kilometer. All the information inside the scenery database is arranged in a four-
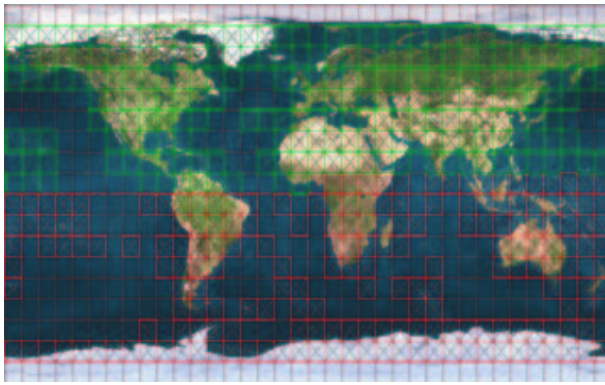
Figure 5: World scenery

level hierarchy, each level changing scale by a factor between 10 and 100:

1. One planet, currently only the Earth

2. $10^o \times 10^o$ rectangle as shown in figure 5,

3. $1^o \times 1^o \approx 70\,mi \times 50\,mi = 100\,km \times 60\,km$,

4. $50\,mi^2 = 100\,km^2$ approximately.

One of the difficulties facing the TerraGear developers is that most information sources are only generated at a national level. It is easy to justify writing special code to read and process data files for the largest ten countries, since they cover most of the land surface of the planet, but this approach rapidly reaches the point of diminishing returns.

There are already many organizations that painstakingly collect and transform the data into standardized formats, precisely for these kinds of applications. However, the huge amount of effort involved requires them to keep the prices extremely high in order to fund the conversions. Therefore, in the medium term, it is possible that these organizations (or one of their licensees) may start selling TerraGear compatible scenery files that is derived from their data archive. You can expect a high price tag for such reliable data though.

Downloading the World

The scenery for the entire world currently requires 3 DVD-ROMs, which is a significant download for users with broadband access and an prohibitive barrier for dial up access users.

It was hoped that someone would get around to writing a utility for on-demand streaming of scenery to the user, but this hasn't happened. A significant factor is that this streaming real time bandwidth is much more expensive to host than the existing bulk retrieval. Money aside, it isn't a difficult problem.

Suppose we consider the pilot's viewpoint. Most general aviation aircraft cruise below 200 knots and flight visibility is (in real life) usually below 20 miles at their cruise altitudes. The database uses about one megabyte for 600 square miles so the peak streaming rate would be 12 megabytes/hour, less for areas previously visited. A 56K modem is easily capable of 12 megabytes/hour.

The utility for streaming scenery download does not need to be integrated into the core FlightGear source code. The latitude and longitude of the aircraft are already exported for use by independent programs, so the center of interest is trivially available. Since the scenery is stored in $100\,km^2$ pieces, an independent program need only generate a list of the closest elements that have not been fetched yet, and issue a `wget` to ensure that they will be available before the aircraft gets close enough for the pilot to see them.

Simulating the Charts

Laptop/PDA applications for use in flight are becoming increasingly popular with light aircraft pilots, since they assist in situational awareness and in managing flight plan logs, navigation data and route planning. While *FlightMaster*, *CoPilot* and other applications are valuable tools, it is dangerous for pilots to use them in an aircraft without first becoming familiar with the user interface and gaining some practice.

FlightGear offers several specialist interfaces, one of which emits a stream of NMEA compliant position reports (the format used by GPS units) to serial port or UDP socket. This can be fed directly into one of those applications, which doesn't notice that this isn't coming from a real GPS, enabling the user to practice realistic tasks in the context of the simulated aircraft and all the realistic workload of piloting.

Data that is released into the public domain is generally of reduced quality, or out of date, or does not give widespread area coverage. The *TerraGear* scenery from such data is actually wrong, compared to the real world, but generally only in ways that are visually unobtrusive to the casual user.

These errors are much more visible in electronic navigation, such as needed for instrument flight, since the route tolerances are extremely tight. Navigating the simulated aircraft around imperfect scenery according to current Jeppesen (or NOS, etc) charts (or electronic databases) can be extremely frustrating and occasionally impossible when a piece of scenery is in the way.

To avoid the frustration, the *Atlas* project[10] has developed software which automatically synthesizes aviation style charts from the actual scenery files and databases being used by FlightGear. These charts, while inaccurate to the real world and therefore useless for flight in an aircraft, are extremely accurate for the simulated world in which the FlightGear aircraft operate. Thus, it is often easier to make printouts from the `Map` program of the *Atlas* project.
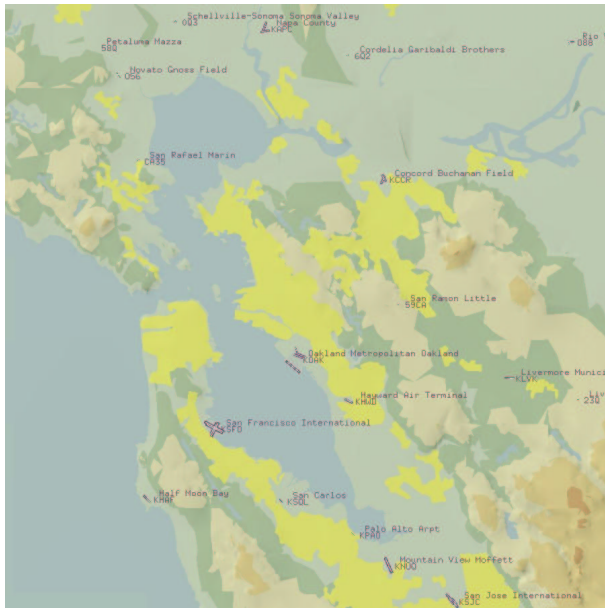
Figure 6: *Atlas* chart of San Francisco, California

The project also includes the namesake *Atlas* application. This can be used for browsing those maps and can also accept the GPS position reporting from FlightGear in order to display aircraft current location on a moving map display. This capability must be used selectively by the simulator pilot, since most small aircraft do not have built in map displays.

The *Atlas* moving map need not run on the same computer as the simulator, of course. It is especially valuable running on the instructor's console, where the pilot cannot see the picture, for gauging the student performance at assigned tasks.

Simulating the Weather

Weather consists of many factors. Some items, such as air temperature and pressure, are invisible but have a strong effect on aircraft performance. Other items, such as smog layering, have no effect on the aircraft or piloting duties but contribute to realism (in Los Angeles, for example). In between these limits are many other items, such as wind and cloud, that must be simulated in order to reproduce the challenges facing the aircraft and pilot. Some complex items, such as turbulence, affect the simulation in many ways and are capable of making the aircraft realistically unflyable.

While FlightGear supports all these items, each of which can vary by location, altitude and time, leading to the difficulty of enabling the user to explain the desired configuration without too much effort. Three modes are currently available.

First, a single set of conditions can be specified on the command line which will be applied to the entire planet and do not change over time. This is very convenient for short duration

and task specific uses, such as flying a single instrument approach from the IAF to the airport, where the same task will recur for each successive student session.

Second, all the weather configuration is accessible through the property database and so can be tweaked by the instructor (for example). This is useful for training on weather decision making, such as choosing between VFR, SVFR, IFR during deteriorating conditions.

Third, a background thread can monitor current weather conditions from `http://weather.noaa.gov` for the closest station. This is useful when conditions may be too dangerous to fly into intentionally, yet the pilot seeks experience with them. Such training, often an opportunity when a training flight is canceled, addresses the situation where the pilot had taken off before the weather deteriorated. Unfortunately, the transitions in weather conditions are necessarily harsh because the official weather reports may be issued as infrequently as once per hour. In any case, when flying between airports, the thread must at some point switch from old airport's report to the one ahead.

None of those is the 'correct' approach. All of them are especially suitable for specific situations. Other weather management approaches can be quickly created, if needed, since all the weather configuration parameters are properties and thus can be managed and modified across the network from a small specially-created utility.

The FlightGear environmental lighting model seeks to offer the best image that can be achieved with the limited dynamic range of computer monitors. For dusk and night flights, as shown in the left side of figure 1, it is best to use a darkened room in order that the subtle differences between the dark grays and blacks can be seen.

Applications for the Simulator

We have a wide range of people interested and participating in this project. This is truly a global effort with contributors from just about every continent. Interests range from building a realistic home simulator out old airplane parts, to university research and instructional use, to simply having a viable alternative to commercial PC simulators.

The Aberystwyth Lighter Than Air Intelligent Robot (ALTAIR)

The Intelligent Robotics Group at the University of Wales, Aberystwyth, UK is using FlightGear as part of their aerobot research[11] to design aerial vehicles that can operate in the atmosphere of other planets.

For those planets and moons that support an atmosphere (e.g. Mars, Venus, Titan and Jupiter), flying robots, or aerobots, are likely to provide a practical solution to the problem of extended planetary surface coverage for terrain mapping and surface/subsurface composition surveying. Not only

could such devices be used for suborbital mapping of terrain regions, but they could be used to transport and deploy science packages or even microrovers at different geographically separate landing sites.

The technological challenges posed by planetary aerobots are significant. To investigate this problem the group is building a virtual environment to simulate autonomous aerobot flight.

The NaSt3DGP computational fluid dynamics (CFD) software package generates meteorological conditions, which are 'loaded' into the FlightGear simulator to create realistic wind effects acting upon an aerobot when flying over a given terrain. The terrain model used by both FlightGear and NaSt3DGP is obtained from the MGS Mars Orbiter Laser Altimeter (MOLA) instrument, and the Mars Climate Database (MCD) is used to initialize the CFD simulation.

University of Tennessee at Chattanooga

UTC has been using Flightgear as the basis of a research project started in August, 2001, with the goal of providing the Challenger Center at the university (and hopefully other centers in the future) a low cost virtual reality computer simulation.

The project is using flightgear and JSBSim, specifically the shuttle module, to develop a shuttle landing simulator. They are aiming to contribute instructions, on how to interface their virtual reality hardware with Flightgear, back to the OS community. The project is funded by the Wolf Aviation Foundation[12]. Dr. Andy Novobiliski is heading the research project.

ARINC

Todd Moyer of ARINC used FlightGear as part of an effort to test and evaluate Flight Management Computer avionics and the corresponding ground systems. Certain capabilities of the Flight Management Computer are only available when airborne, which is determined by the FMC according to data it receives from GPS and INS sensors.

They wrote additional software that translates the NMEA output of FlightGear (including latitude, longitude, and altitude) into the ARINC 429 data words used by GPS and INS sensors. These data words are fed to the Flight Management Computer. the position information from FlightGear is realistic enough to convince the FMC that it is actually airborne, and allows ARINC to test entire 'flights' with the avionics.

MSimulation

Marcus Bauer and others worked on a simulator cockpit environment using FlightGear as the software engine to drive a real cockpit, including three cockpit computers.

Space Island

Space Island[13] Space Island are using FlightGear as the software for a moving cockpit entertainment simulator that supports both flight and space environments.

Other applications

Many applications started using FlightGear years ago:

1. University of Illinois at Urbana Champaign. FlightGear is providing a platform for icing research for the Smart Icing Systems Project[14].

2. Simon Fraser University, British Columbia Canada. Portions of FlightGear were used in simulation to develop the needed control algorithms for an autonomous aerial vehicle.

3. Iowa State University. A senior project intended to retrofit some older sim hardware with FlightGear based software.

4. University of Minnesota - Human Factors Research Lab. FlightGear brings new life to an old Agwagon single seat, single engine simulator.

5. Aeronautical Development Agency, Bangalore India. FlightGear is used as as the image generator for a flight simulation facility for piloted evaluation of ski-jump launch and arrested recovery of a fighter aircraft from an aircraft carrier.

6. Veridian Engineering Division, Buffalo, NY. FlightGear is used for the scenery and out-the-window view for the Genesis 3000 flight simulator.

Configuration User Interfaces

Scripting languages such as Python and Perl use a wrapper API for the network interface, hiding the protocol and operating system. This approach enables instructor interfaces to be developed in accordance with regulatory requirements and quickly customized to meet specific local needs.

Simulating Flight Training

FlightGear could also be helpful when learning to fly aircraft. Flight training is carefully regulated by the government, to ensure that aircraft generally stay in the sky until their pilot intends for them to come down safely. There are thus some real concerns which need to be addressed before authorities can approve a system.

1. Do the controls feel, and operate, sufficiently like the ones in the aircraft that a pilot can use them without confusion? Are they easier to use and/or do they obscure dangerous real-life effects?

2. Does the software provide a forward view that is representative for the desired training environment?

3. Are the instruments drawn such that a pilot can easily read and interpret them as usual? Do they have the systematic errors that often cause accidents?

4. Are the cockpit switches and knobs intuitive to operate?

5. Operating within the limited envelope of flight configurations that is applied to the training activity, does it match the manufacturer's data for aircraft performance?

6. Are weather settings accessible to the instructor and sufficiently intuitive that they can change them quickly?

7. Are there easy mechanisms for causing the accurate simulation of system failures and broken instruments?

8. Can the pilot conduct normal interactions with air traffic control? Can the instructor easily determine whether the pilot is complying with the control instructions and record errors for subsequent review?

9. Is the pilot's manual for the simulator similar in content and arrangement to that of the aircraft being represented, such that it can readily be used in flight by the pilot?

10. Can all maneuvers be performed in the same way?

In that (partial) list of concerns, the quality of the actual flight simulation (which is really what FlightGear is offering) is a minor topic and and acceptable performance is easily achieved. In contrast, a large package of documentation must be added to the software to explain and teach people how to use it correctly. This has led to a number of separate projects whose goals are to meet or exceed the standards created by the United States Federal Aviation Administration (FAA).

It is easy to suggest that the FAA is being unrealistic in requiring this documentation, but they are responding to important traits in human nature that won't go away just because they're inconvenient.

For example, the things learnt first leave an almost unshakeable impression and, at times of severe stress, will over-rule later training. Thus, any false impressions that are learned by a beginning student through using a simulator will tend to remain hidden until a dangerous and potentially lethal situation is encountered, at which time the pilot may react wrongly and die. Pilots who use a simulator on an ongoing basis to hone their skills will get an excessively optimistic opinion of their skills, if the simulator is too easy to fly or

Figure 7: Example display from the OpenGC[8] project

does not exhibit common flaws. As a result, they will willingly fly into situations that are in practice beyond their skill proficiency and be at risk.

Clearly, a flight simulator (such as FlightGear) can only safely be used for training when under the supervision of a qualified instructor, who can judge whether the learning experience is beneficial. The documentation materials are essential to supporting that role.

What's in the future?

In many areas of the project, the source code is stable and any ongoing programming rarely affects the interfaces used by XML files. The majority of the current developer effort centers around the crafting of nice looking 3D aircraft, animating their control surfaces, synthesizing appropriate sound effects, implementing an interactive cockpit, and adding detail to the aerodynamics parameters. The aerodynamic models are not (yet) accurate enough for use in all flight situations, so they don't reflect the challenges and excitement of acrobatic maneuvering.

Surround projectors, head mounted displays, directional sound and cockpit motion are rapidly converging into consumer technologies. Maybe we can immerse the users so well that they fly conservatively because they forget that they're not in real danger.

Aircraft wake is invisible, can last five minutes, descends slowly or spreads across the ground, is blown around by the wind and is extremely dangerous to following aircraft. A future extension to `fgd` could keep track of the hundreds of miles of wake trails in a given area and notify individual aircraft when they are encountering invisible severe turbulence.

Replication and scalability is only starting to take hold in the desktop environment. A room of several hundred computers acting as X terminals for word processing can reboot and, within a couple of minutes, all be running FlightGear identically. They're ready for the next class of student pilots.

Conclusions

On the surface, FlightGear is a simple Open Source project that builds on many existing projects in the community traditions. Due to the subject it addresses, many issues and concerns are raised that rarely inconvenience most other project teams. These elements are providing the exciting challenges and variety of associated activities that the developer team is enjoying.

About the Author

Alexander Perry holds M.A. and Ph.D. degrees in engineering from Cambridge University in England. A member of the IEEE Consultants Network in San Diego, he is one of the *FlightGear* developers, a commercial and instrument rated pilot, ground instructor and an aviation safety counselor in San Diego and Imperial counties of California.

References

[1] http://www.flightgear.org/

[2] http://plib.sourceforge.net/

[3] http://www.linuxbase.org/

[4] http://oss.sgi.com/projects/ogl-sample/ABI/

[5] http://www.gltron.org/

[6] http://www.simgear.org/

[7] http://jsbsim.sourceforge.net/

[8] http://opengc.sourceforge.net/

[9] http://www.terragear.org/

[10] http://atlas.sourceforge.net/

[11] http://users.aber.ac.uk/dpb/aerobots.html

[12] http://www.wolf-aviation.org/

[13] http://www.spaceisland.de/

[14] http://www.aae.uiuc.edu/sis/mainpapers.html

[15] http://fgatd.sourceforge.net/