FLIGHT GEAR MULTIPLAYER ENGINE

The development of a flight simulator multiplayer engine for AI purposes.

Boogaard, Jeroen
Otte, Leon

Technical Report DKS-02-05 / ICE 02
November, 2002
Mediamatics / Data and Knowledge Systems group
Department of Information Technology and Systems
Delft University of Technology, The Netherlands

ROTTERDAM

TU Delft

Boogaard, Jeroen (boogaardjeroen@hotmail.com)
Otte, (liono@zeelandnet.nl)
"FlightGear Multiplayer Engine : The development of a flightsimulator multiplayer engine for AI purposes"

Technical Report DKS-02-05 / ICE 02
November, 2002

Mediamatics / Data and Knowledge Systems group
Department of Information Technology and Systems
Delft University of Technology, The Netherlands

http://www.kbs.twi.tudelft.nl/Research/Projects/ICE/

Keywords: ICE project, multiplayer, artificial intelligence, flightsimulator, flightgear, simgear

Preface

This thesis describes our research activities on the development of our Flight Simulator Multiplayer Engine (MPE, for short) at the Knowledge Based Systems (KBS) group of the Delft University of Technology. We developed the ‘Flight Gear Multiplayer Engine’ for one of the projects of the KBS group, which is called the Intelligent Cockpit Environment (ICE) project.

Project

The ICE project focuses on the possibilities of Artificial Intelligence (AI) in the field of pilot awareness using the flight simulation software ‘Flight Gear’. Participating in the ICE team we wanted to increase our knowledge of AI. Our first idea was to develop artificial pilots (bots) to be implemented in Flight Gear. For this purpose we needed the Multiplayer engine as more than one airplane has to be displayed simultaneously. Unfortunately the existing Multiplayer engine in Flight Gear, with which multiple players can employ the same simulation area at the same time, was incomplete and not portable to other platforms than Linux. So we decided to develop a new one. This Multiplayer Engine would deliver more research possibilities in the field of flight avionics, pilot behavior and would enable Flight Gear to implement bots.

Report overview

This report is separated into three parts. The first part contains theory about air navigation, Flight Gear and Client Server techniques. The second part contains the design by description and schematic representation. Finally in the third part, the results are described.

Acknowledgements

At first we want to thank Mohammed Abdelghany for his help in finding an organization for our traineeship and all of his ideas and advices for our project. We also want to thank Leon Rothkrantz for his guidance and advice during our project. Furthermore we want to thank Jean Paul van Waveren, Patrick Ehlert and Boi Sletterink for their help. Finally we want to thank all the people on the Flight Gear mailing list for answering our questions.

Abstract

PC Flight Simulators are appropriate for pilot training and research. One important feature of a PC Flight Simulator is the possibility to run the simulation with multiple players simultaneously. There are professional flight simulators such as Microsoft Flight Simulator 2002 that is quite realistic and is not limited to one player. But the problem with commercial flight simulators is that they do not offer features to enable users to make modifications and/or add new functionality. Other disadvantages are the price and the platform limit.

Flight Gear is a sophisticated multi-platform open source flight simulator framework for research, development pursuit of other flight simulation ideas. As Flight Gear is developed under the GNU Public Licence, it is freely available and everyone may modify, extend and recompile the source code. Flight Gear does not have a complete multiplayer engine yet.

In this report the development of our Flight Gear multiplayer engine (MPE) will be described. A prototype of our modular MPE is implemented in Flight Gear and the results are described. The source code as well as the specification will be freely available on the Internet.

Table of contents

CHAPTER 1: INTRODUCTION

This study contains the possibilities and development of a multiplayer engine for an open source flight simulator. The first section describes the problem setting. Next a brief description of Flight Gear will be given (section 1.2). The third section introduces the problems involved with multiplayer engines for flight simulators. The ICE-project will be discussed in section 1.4. The goals of the Flight Gear MPE can be found in section 1.5. The final section (1.6) describes contains an overview of this report.

1.1 Problem setting

Flight simulators are useful for both pilot training and aviation research. A flight simulator can be a serious training tool and fun at the same time. Simulation gives pilots a chance to push limits and explore boundaries without real life consequences. The disadvantages of large moving platform flight simulators are that they are expensive and place dependent.
PC flight simulators are a good alternative as they provide a safe, low-cost, place independent practice environment.

There are many different flight simulators from simple games to realistic research simulators. To compare flight simulators, one can consider:

- user control (cockpit instruments, joystick support);
- analogy to real airplanes;
- the amount of different airplanes, and flight models;
- the amount of different airports;
- the presence of different weather conditions;
- the details of the scenery;
- real time aspects;
- real life aspects;
- possibilities to log data;
- possibilities for user configuration and modification;
- the amount of users that can play simultaneously (multiplayer ability);
- the presence of artificial traffic.

An ideal flight simulator for research would be one:

- with an advanced user control;
- has many different sophisticated airplanes and flight models;
- has many different airports with Air Traffic Control (ATC);
- provides different weather conditions;
- has a very detailed scenery;
- supports datalogs;
- can be configured and modified easily by users;
- supports many simultaneous players;
- can generate artificial traffic;
- has real time aspects;
- has real life aspects.

A realistic flight simulator ‘feels’ like controlling a real aircraft instead of playing a game. Of course this is also very dependent on the hardware, that is supposed by the flight simulator. Some flight simulators support a rudder for control and multiple monitors so one can ‘look around’ in the simulator environment. Others only support one monitor and a keyboard for control; this will ‘feel’ more like a game than a (professional) flight simulator.

Probably one of the most famous flight simulators is Microsoft Flight Simulator. Microsoft received the “Bravo Zulu” Award for Best Commercial GA (General Aviation) Simulator for outstanding achievement with Flight Simulator 2002. This flight simulator seem to be an appropriate simulator since it is an advanced simulator containing a number of different airplanes, flight models and airports and also includes an Air Traffic Control, a multiplayer option and an AI system generating air traffic. But what about the ability for users to make modifications? Microsoft provides patches with new aircrafts and scenery but significant changes cannot be made. Microsoft Flight Simulator is commercial software and the source code is unavailable. Commercial programs have a magnificent drawback. They are built by a small group of developers defining their properties – often quite inert and sometimes not listening too much to the customers [BAS98].

1.2 Flight Gear

A very advanced open source flight simulator is Flight Gear. There is a need for many people involved in education and research to use a flight simulator as a framework for their own projects. Unfortunately it is not possible to use commercial flight simulators for this purpose since commercial software is unable for modification and enhancement. What is needed is an open source flight simulator that can be expanded and recompiled by the users.
In 1996 the idea to build an open source flight simulator was discussed on the Internet. The name of the simulator would be Flight Gear and the targets were:

- it should be freely available according to the GNU Public License;
- implemented in different modules, modules can be added and changed apart from other
- modules (except the main module);
- scenery of high quality but without the requirement of extraordinarily hardware (but it needs
- a 3D graphics card with OpenGL support);
- Flight Gear must be a civilian flight simulator;
- Flight Gear should be able to build and played in a platform independent way.

Flight Gear became a multi-platform general aviation flight simulator for which anyone with network access and a C++ compiler can contribute patches or new features. It is very advanced including , different aeroplanes and flight models, a very detailed scenery based on real maps, different time zones, wheater conditions and even a representation of the sun, moon and stars that are very close to the reality.

Although Flight Gear already provides many research posibilities, there are still components that still need to be implemented to make it more complete inluding:
multi-player option for local networks, modem connections, and over the Internet;
an interactive, intelligent Air Traffic Control (ATC) [FGD98].

The research posibilities could be increased by the development and implementation of a complete multiplayer engine which supports the basic functionality on both Linux and Windows.

1.3 Multiplayer Engine

The term multiplayer engine (MPE) is commonly used in the world of games. There are also different ways to implement such an engine. In this section multiplayer engines are briefly discussed.

A multiplayer engine is software that enables multiple users to enjoy the same game at the same time.

1.3.1 Separate multiplayer engine

Usually developers have decided already at the first design to add multiplayer functionality to their game. This will cause less latency, compatibility and synchronistation problems, etc. than when the MPE is added to an already developped game. Flight Gear, the flight simulator used for this project, is already evolved into an advance single player flight simulation. Despite one can think "the more players, the more fun", the flight simulator has to be able to run in single mode even if it is compiled with multiplayer functionality. Flight Gear is a flight simulator, not a 'first person shoot-em up game' like Quake Arena (without bots) at which one is dependent of other (network) players to participate in the game [JPW01]. Therefore the MPE can best be developed as 'loadable modules' i.e. software modules that provide a multiplayer option and can be add to the flight simulator. The main purpose of the loadable modules is that they avoid modifications to the simulator engine.

1.3.2 Split screen

For a better understanding of what a flight simulator MPE is supposed to do, one can consider a single player simulator. Figure 1-1 shows a very simple configuration of a single user flight simulator. The player controls the aircraft by use of input devices such as a joystick, keyboard or rudder. The visual environment and textures of the simulator can be displayed on a screen via a 3D supporting graphical card. Sounds can be send to the player using a sound card and speakers.

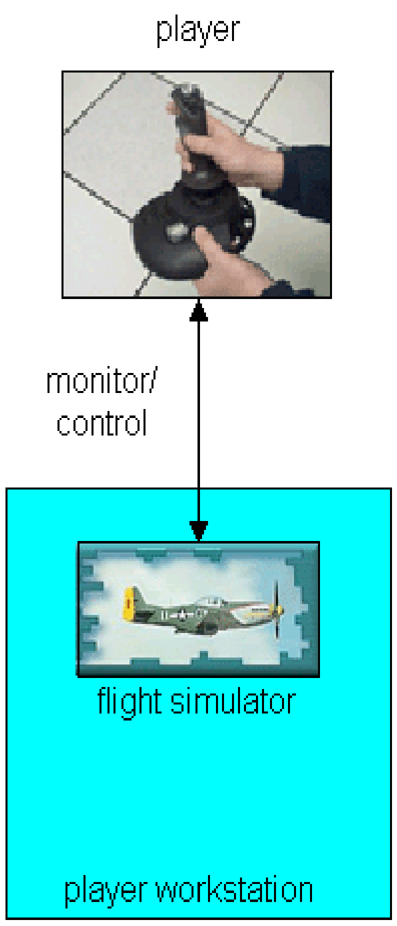


Figure 1-1 Single player flight simulator

All a single player flight simulator needs is a stand alone PC with input and output control for one user. When multiple players will enjoy the same flight simulator on the same PC simultaneously, both the input and output capacities have to be shared. This can be done by splitting the screen and installing multiple input devices on the same PC. Using the same audio card, the sound will be mixed too. This form of multiplayer requires that the players are able to be at the same place. It is also restricted to two players.

1.3.3 Head-to-head

Using a serial link or modem, players will not have to use the same workstation (PC) so splitting the screen and deviding the input devices will not be necessary. This form of multi player is known as 'head-to-head.' With head-to-head, one of the workstations has to act as server. This server has to organize the data transport and manage the different input. At first the server needs to know what aircraft type is used by the other player which in his turn needs an unique identification. The continuous information stream will contain position and orientation data to be able to draw the 'other' aircraft. For the real time position data, a position logger is needed for continuous receiving position and orientation information from the flight simulator. This form is still restricted to two players.

1.3.4 Network mutliplayer engine

More than two players can enjoy the flight simulator using a LAN (Local Area Network) or the Internet. This can be done connecting each workstation with each other (peer-to-peer). The problem of this form is that there is no 'game-master' which has the overview and can take 'fair' decisions to avoid that some players are privileged and others have a relative handicap.

Another way to enable multiple network players is using a central system where the player and aircraft data of all involved players are registered (client/server). This architecture is shown in figure 1.2.

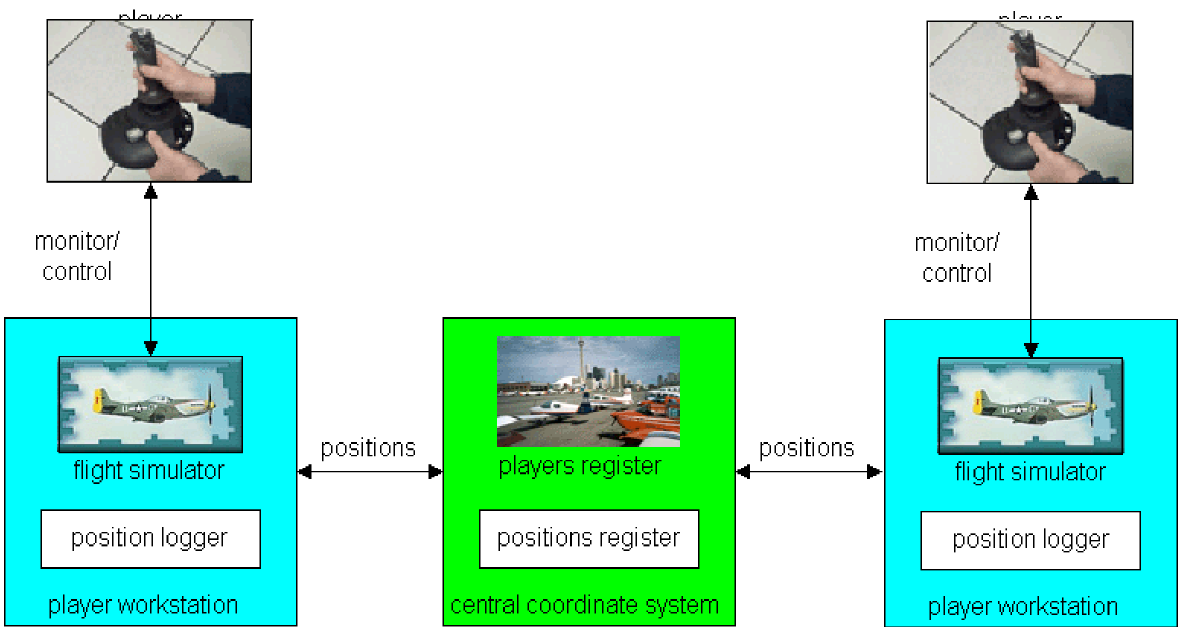


Figure 1-2 multiplayer flight simulator using a central coordinate system

In this architecture there is not a continuous data stream so extrapolation is needed to keep fluent movements. There should also be decided how much time each involved PC will be given to send and receive data. Another aspect will be the amount of position data that will be sent at a time.

1.3.5 Extrapolation

Using a network MPE, the data stream will not be continuous since the frequency with which a workstation will receive position data will be lower than its screen update frequency. Therefore extrapolation is needed. The positions and orientation of an 'other' aircraft can be calculated using dynamics formulas. Every time the screen must be updated and there is no new data packet from the corresponding workstation received, the new positions will be derived using the previous data packet and calculation. This solution is not ideal as during special maneuvers an aircraft's new orientation and position is less predictable. The reliability of this method will depend on the amount of available information of the 'other' aircraft as well as the frequency with which this data can be send. At this point the 'right' size of data, that will be transmit, must be chosen. Much information will increase the exactness of the extrapolation but will decrease the transmission speed. Less information will decrease the extrapolation exactness but will also decrease the need for extrapolation since more packets can be send. Another advantage of small data packets is that the transmitted information will be more up-to-date.

1.3.6 Extrapolation based MPE

In the previous section data transmission is the base to acquire the position data of ‘other’ aircrafts and extrapolation is a tool to compensate the ‘empty intervals’ i.e. time periods in which no up-to-date data is received. Another approach can be to use the extrapolation as data source and compensate it is aberrations with data transmission. For this, both a player workstation and the central coordinate system have to keep track of extrapolation. The player workstation will continuously compare it is extrapolation data with the data available from the position logger, if the differences are above a certain level the ‘real’ data will be send to the central coordinate system which will than replace it is derived data with the ‘real’ data to be send to ‘other’ player workstations.

1.3.7 Circle of sensing

The previous two sections described MPE’s that use a central coordinate system that keeps track of the data exchange between the involved player workstations as no data will be send directly from one workstation to another as with the head-to-head approach (see section 1.3.3). As aircrafts can be to far to notice each other, one can consider cases in which it is irrelevant to receive position data from a certain ‘other’ aircraft, as it is to far away to notice. For this, a more selective (and therefore efficient) procedure can be used that will decrease the amount of data copies to send to ‘others.’ To realize such a selective data transmission the central coordinate system has to keep track of which aircraft is in the ‘neighborhood’ i.e. within a noticeable circle. Despite the fact that in the simulator one can look only in forward direction, it is best to use a circle in case one want to add a radar system to the simulator cockpit. The ‘circles of sensing’ of aircrafts can be implemented as using a list with player id’s (since a player can only control one aircraft simultaneously) and their ‘radius of sensing.’ If someone ever adds a fast (military) aircraft such as an F16 to the flight simulator, probably the ‘circles of sensing’ method has to be extended since military aircrafts will be for ‘normal’ aircrafts and visible for a very short time.

1.4 Project goals

The first sections describe the need for a multiplayer engine for an open source flight simulator. The goal of this project is to study the incomplete Flight Gear multiplayer engine and develop our own prototype that can be used to add ‘bots.’ The prototype can be split up in two major parts:

1. A network module to exchange data between the players;
2. An interpretation module that will use the data for the visualization of multiple players.

The development of a perfect multiplayer engine is very difficult especially since it is an addition to a very advanced flight simulator that provides different airplanes, flight models, weather circumstances, etc. It is also difficult to create the possibility to play via the Internet since involved workstations can have big differences in network speed. So our prototype will only provide a basic functionality at which further adjustments can be added.

The goals of our project will be:

1. to study the multiplayer possibilities in Flight Gear;
2. to make a design of a multiplayer engine;
3. to develop a prototype and implement it in Flight Gear;
4. to test the prototype and describe its possibilities and limits.

Once I read about software development:
"After you finish the first 90 percent of a project, you have to finish the other 90 percent."

Our goal is to reach at least the first 90 percent and at that point there will be lots of functionality which can be add to make it better but for we needed twice the time we had to spend.

1.5 Intelligent Cockpit Environment

The Flight Gear Multiplayer Engine Project is a subproject the 'Intelligent Cockpit Environment', which will be briefly introduced in this section.

1.5.1 Introduction

The goal of the Intelligent Cockpit Environment (ICE) project is the development of new techniques for intelligent interfaces for military aircrafts. Pilots of military aircraft have to deal with a lot of information while controlling the airplane. The situation can arise that the pilot will be overloaded with information. In such a situation there will be the risk that a pilot will miss important information. Also in critical situations such as air combats the pilot do not want a big flow of information. To increase the help to the pilot, the system should control the progression of the flight mission. Figure 1-2 shows the block diagram of the ICE system.

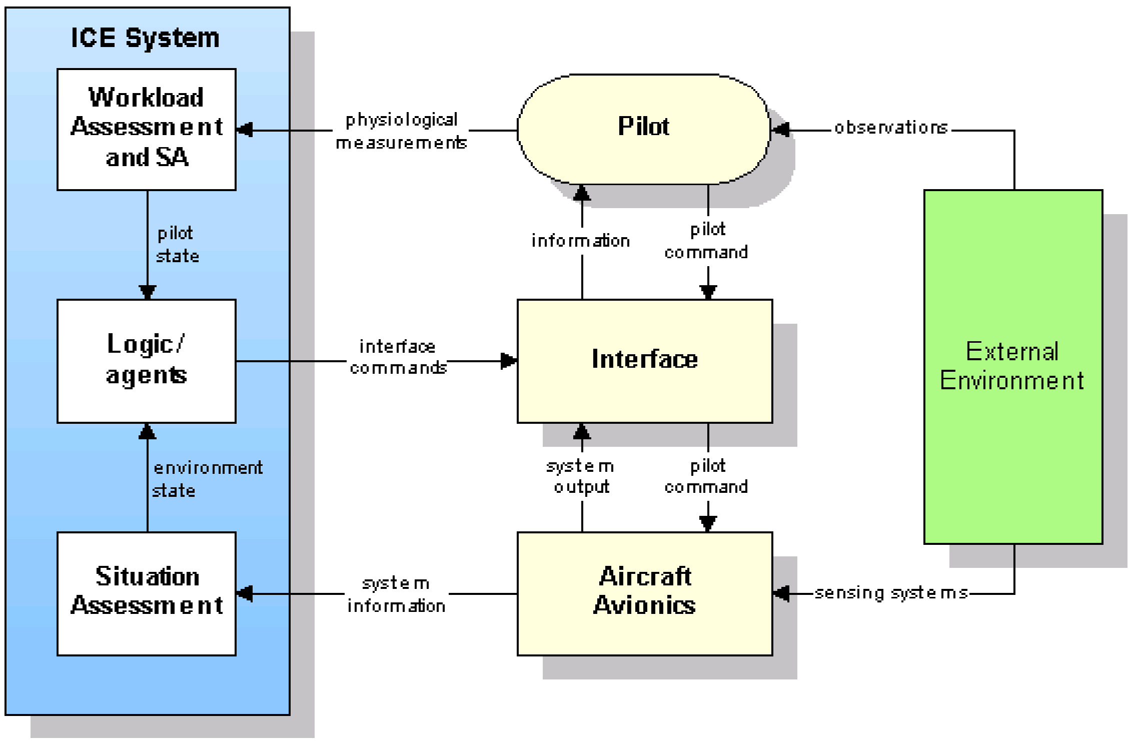


Figure 1-3 ICE System

To control the information flow in a more pilot friendly manner, one needs to know: the *situation assessment*, the *workload assessment* and the external environment. The workload assessment defines 'how busy' the pilot is i.e. the actions the pilot takes as a response to the external environment. The workload assessment contains also the pilot's amount of stress. The actions of the pilot will contain controlling the aircraft and can therefore be logged. A gaze tracker can be used as indication of the amount of stress of the pilot. To determine the progression of the pilot's mission, the system must 'know' the flight mission purpose.

As one can see in the diagram, the external environment is being observed by both the pilot and sensing system. The pilot and environment state are delivered to an interface control module that communicates with a knowledge base and commands the interface as needed.

1.5.2 External environment

An important component of the external environment contains the 'Approaching Airtraffic' i.e. other aircrafts flying in environment or landing on or taking off from a close airport. The goal of the Flight Gear Multiplayer Engine Project as a sub-project of ICE is to enable approaching air traffic in the ICE simulation environment.

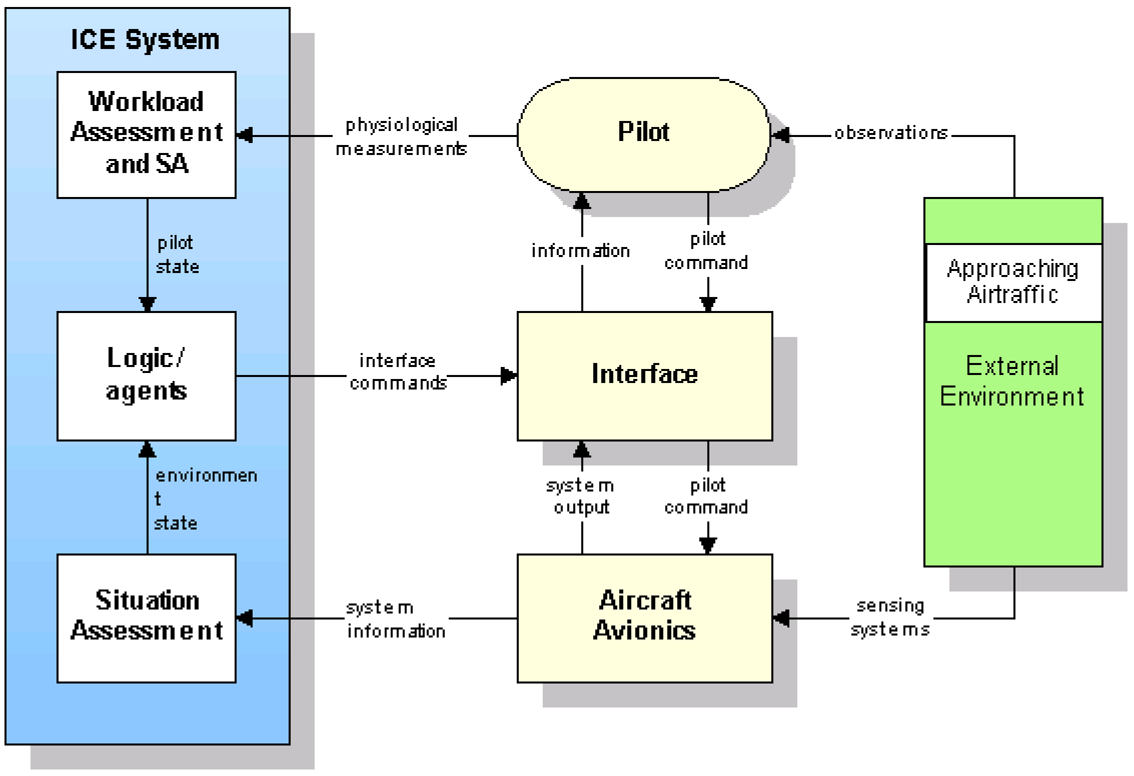


Figure 1-3 Approaching Airtraffic

1.6 Report structure

The second chapter gives an overview of the current modules of Flight Gear, the MPE modules of our prototype and the relation between them. Chapter 3 contains a brief introduction to air navigation that will be used to determine the system's necessary data. Client/Server strategies will be discussed in chapter 4, this includes a comparison between multi-threading and multiplexing-IO. Chapter 5 contains some system analysis displayed by Data Flow Diagrams, Entity Relationship Diagrams, State Transition Diagrams, Event Lists, etc. Chapter 6 describes the main differences between UDP and TCP. It also contains an overview of the structures of the different layers of the MPE protocol. Subjects like data compression and data containers are used for the MPE protocol and are also described in chapter 6. In chapter 7, techniques in the field of Internet applications and portability are described as well as how they are used by the MPE server. The MPE client is discussed in chapter 8.

CHAPTER 2: DESCRIPTION OF THE ORIGINAL SYSTEM

2.1 General

From section 1.2 it is clear that Flight Gear is an appropriate open source flight simulator.
The simulator functions as a systems integrator by bringing together the various flight simulator components composed of a reconfigurable aircraft model, flight mechanics, aerodynamics and propulsion [SDS02].

2.1.1 Framework for development

As said in the introduction (chapter 1), Flight Gear is a framework for development by being configurable and extensible. It is an open source flight simulator that adheres to the GNU General Public License (GPL). So it permits everybody to modify and redistribute the code (without changing the copyright notice). The classes contain fairly 'clean' C++ code so it is possible to understand them and to make modifications. For help and questions, the Flight Gear development group delivers support via the Flight Gear mailing list. (An example of questions to the mailing list as well as the answers to them can be found in Appendix C).

2.1.2 Framework for research and pilot training

New 3D Flight Gear models (aircrafts) are being developed but also models that were originally intended for Microsoft Flight Simulator can be (freely) downloaded to use in Flight Gear.

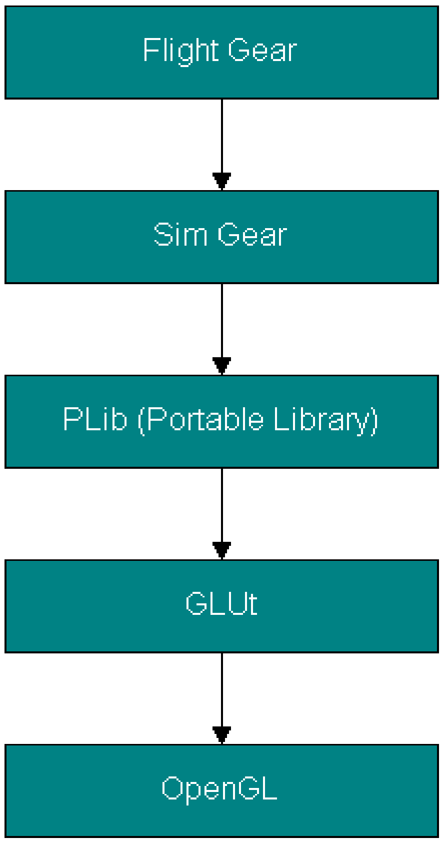


Figure 2-1 Layers Flight Gear

Flight Gear is a real time 3D flight simulator. All the polygonal drawing is done using OpenGL (Open Graphics Library) in an indirect manner i.e. the flight simulator does not allocate the OpenGL functions itself but through other libraries in between. The technical structure consists of five hierarchical layers as shown in figure 2-1. As one can see, OpenGL is the lowest level in this library hierarchy. In the following sections the components of figure 2-1 will be described.

OpenGL

OpenGL is a portable graphics library for graphical operations and film-effects.
OpenGL provides a broad set of rendering, texture mapping, special effects and other powerful visualization functions. It is commonly used for the development of 3D games. Unfortunately it is not (yet) supported by all graphic cards.

Glut

One layer above OpenGL one finds GLUt (GL Utilities) is a utility library for OpenGL programming. With Glut the complex OpenGL functions are easier to program. A collection of Glut libraries are provided by the 'Simple Scene Graph Library', the graphical component of PLIB described in the next section.

PLIB

Portable Library (PLIB) is a suite of portable game libraries. PLIB includes sound effects, music, a complete 3D engine, font rendering, a GUI, networking, 3D math library and utility functions, all portable across nearly all modern computing platforms.

PLIB components:

- Picoscopic User Interface Library (PUI)
- Sound Library (SL)
- Standard Geometry Library (SG)
- Simple Scene Graph Library (SSG)
- SSG Auxiliary Library (SSGA)
- Joystick wrappers (JS)
- Fonts and Text Library (FNT)
- Utility Library (UL)
- Pegasus Network Library (NET)

Sim Gear

The simulator engine for Flight Gear is Sim Gear, so this layer can be seen as a 'simulation layer' providing general 3D simulation functionality. This is a set of libraries that can be used as building blocks for quickly assembling 3D simulations, games and visualization applications. Sim Gear contains libraries like math, screen, route, math, timing, sky and io. Flight Gear is using these libraries for its general simulator functions. Positions are stored in 'points' and 'polars', objects that are provided by the Sim Gear math (mathematics) library. As described in the next chapter, Flight Gear uses different coordinate systems. The Sim Gear math library also provides conversion functions between these different systems. Points and polars contain double values, to keep the data packets small it will be better to use floats as they have a length of 4 bytes instead of 8. More about floats will be described in section 5.2.1.

Flight Gear

As said in the previous section, Flight Gear uses Sim Gear for general simulator functionality.

Flight Gear itself contains flight simulator specific modules like ATC, aircraft, airports, autopilot, cockpit and FDM (Flight Dynamics Model). In the next section an overview of Flight Gear modules is presented.

2. 2 Used modules

Flight Gear consists of various modules that can be compiled independently. Figure 2-2 shows an overview of the Flight Gear modules.

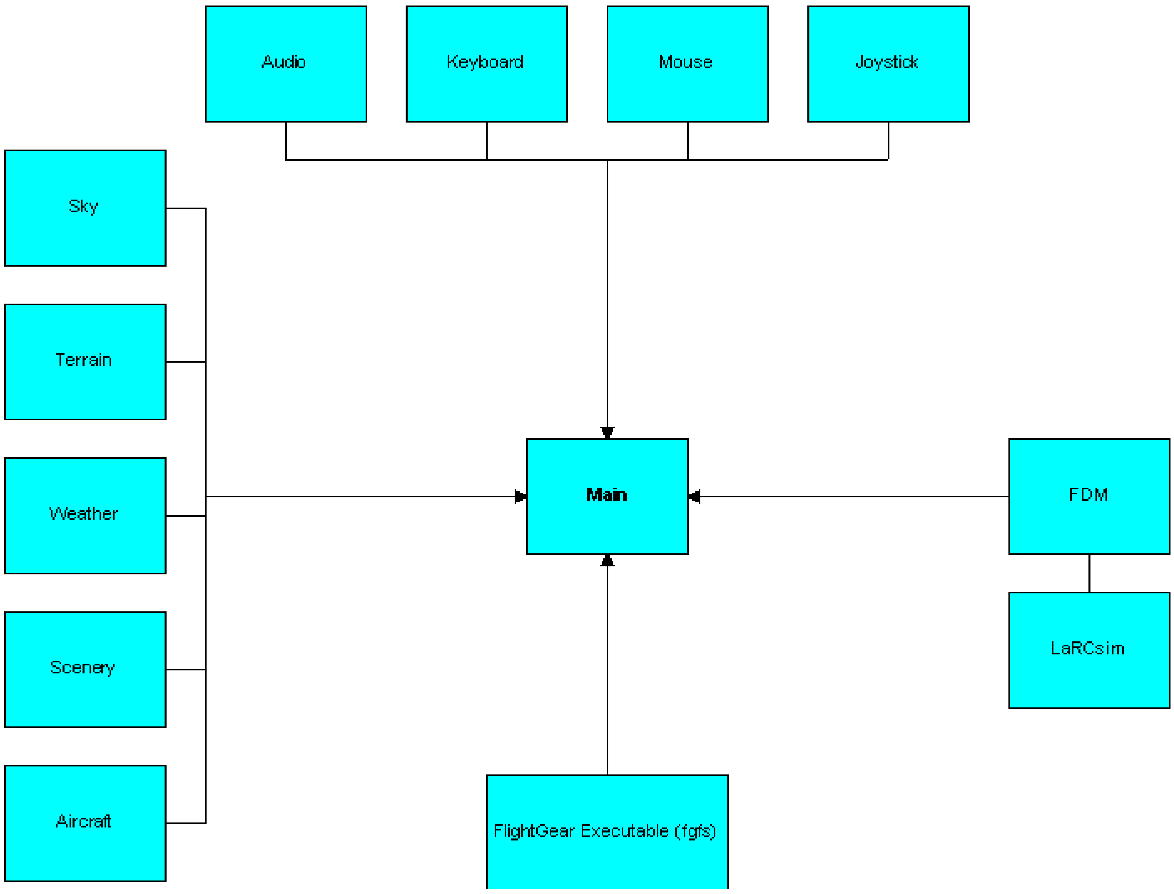


Figure 2-2 Flight Gear modules

note: fgfs stands for Flight Gear Flight Simulator.

The components at the top are responsible for the user input. This layer contains the hardware drivers that enable user control. The components at the left hand side include all graphic components to be displayed during simulation. At the right hand side the module 'FDM (Flight Dynamics Model)' contains different flight dynamics, 'LaRCsim by default.' The center module is Main, this is the heart of the flight simulator from where the other modules are allocated. To run the flight simulator, the execute module 'fgfs (Flight Gear Flight Simulator)' is coupled to the main module. Fgfs supports parameters to enable options to choose the aircraft, airport, etc.

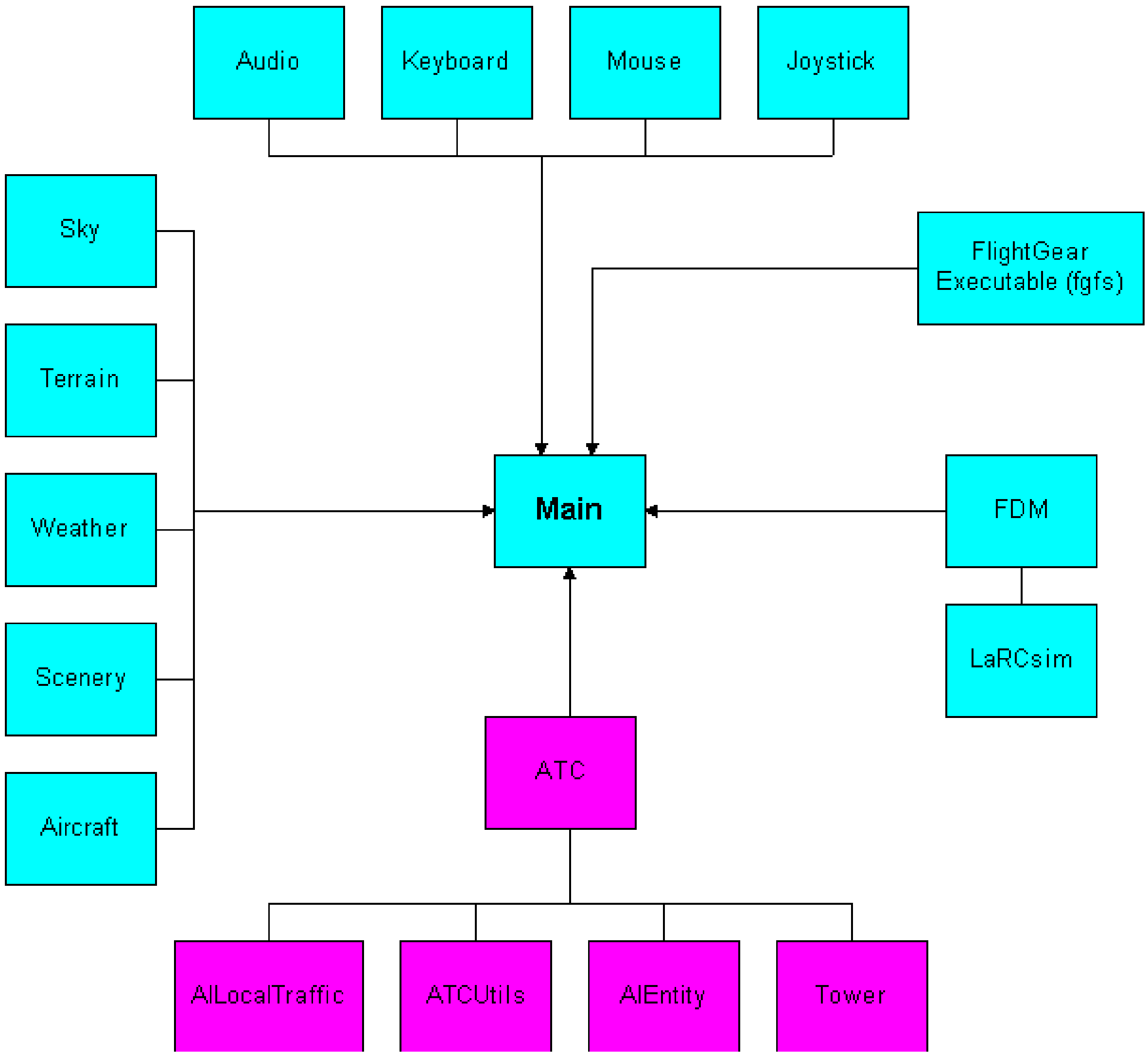


Figure 2-3 Flight Gear modules ATC

2.2.1 Air Traffic Control

One of the available modules is Air Traffic Control (ATC), showed in figure 2-3. The purpose of these modules is presence and control of local traffic. ATC also includes incomplete sources for AIEntities. Despite the ATC tower can be implemented successfully, the AI traffic is not yet usable. But at least parts of the sources AIEntity and AILocalTraffic can be used to display 'other airplanes' i.e. other airplanes than the one controlled by the user itself (see Appendix C, question 2). ATCUtils contains formulas to determine a new position given a current position and distance, which can be usable for interpolations.

2.2.2 Network and NetworkOLK

Curtis Olson developed these modules for his multipilot project. As said before, this project has never been completed. The network module delivers a socket for data communication, using TCP. The code still contains a lot of bugs and is written for Linux only.

CHAPTER 3: AIRCRAFT POSITIONING

Using position data acquired from Flight Gear, different coordinate systems must be considered as well as the conversion between them. The chapter starts with a brief introduction in section 3.1. The second section describes annular positioning considering a flat circle. In the third section a description of the earth is given from a geocentric point of view. In the next three sections latitude, longitude and altitude are discussed respectively. The different coordinate systems are described in section 3.7. Finally section 3.8 contains formulas to calculate the North and East distance.

3.1 Introduction

Because the earth is a not a flat object, positioning cannot be done just using simple x, y coordinates. Instead aircraft positioning has been done relative to fictional axes of the earth. There are several positioning systems each with an own point of view. Three of them that are used in Flight Gear will be discussed in this chapter.

3.2 Position on earth

Direction can be defined as the annular position of one point to another without reference to the distance between them. The simplest way to describe a direction is to consider a flat circle divided into 360 degrees, clockwise positive.

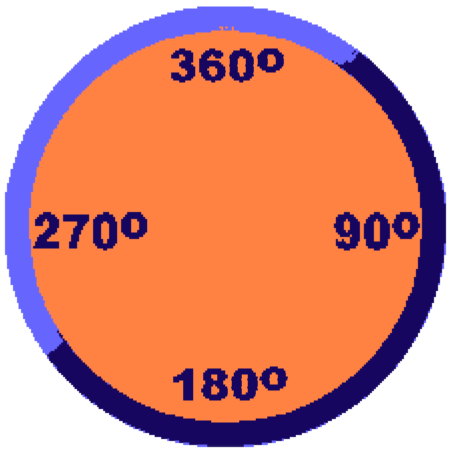


Figure 3-1 A flat circle divided into 360 degrees

3.3 The geocentric earth

The geocentric system considers the earth being a perfect sphere. The earth has a Northern and Southern Hemisphere, separated by the equator and an Eastern and Western Hemisphere, separated by the Prime Meridian [THO90].

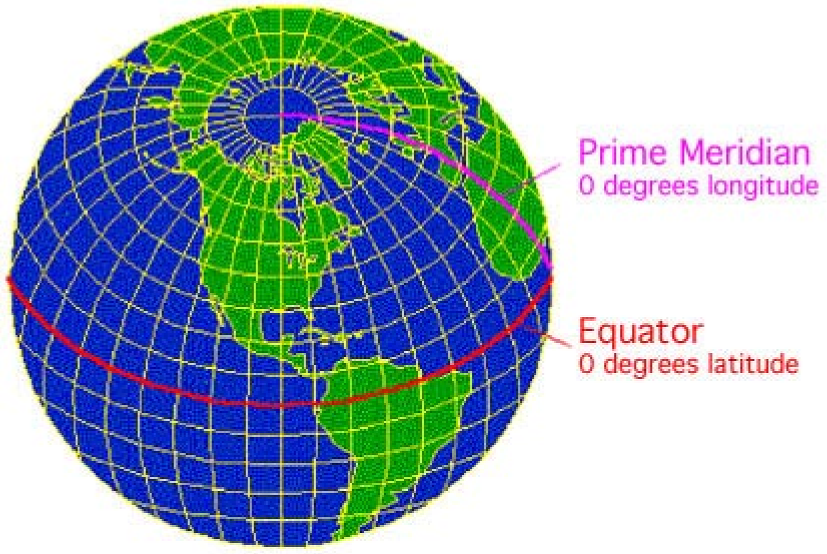


Figure 3-2 Prime Meridian and Equator

3.4 Latitude

One can define a place North or South from the equator, this is known as the latitude.

3.4.1 Great Circles

A great circle is the largest circle that can be drawn on the surface of the earth or on any sphere. The equator is a great circle whose plane is perpendicular to the polar axis.

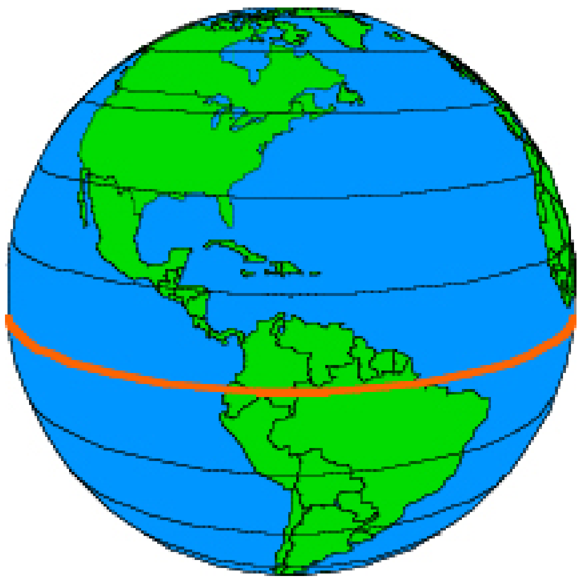


Figure 3-3 Equator

3.4.2 Parallels of latitude

Circles parallel to the equatorial plane, grow smaller near the poles and are therefore small circles i.e. circles on the surface of the sphere that are not at the center of the world. These small circles are knows as parallels of latitude.

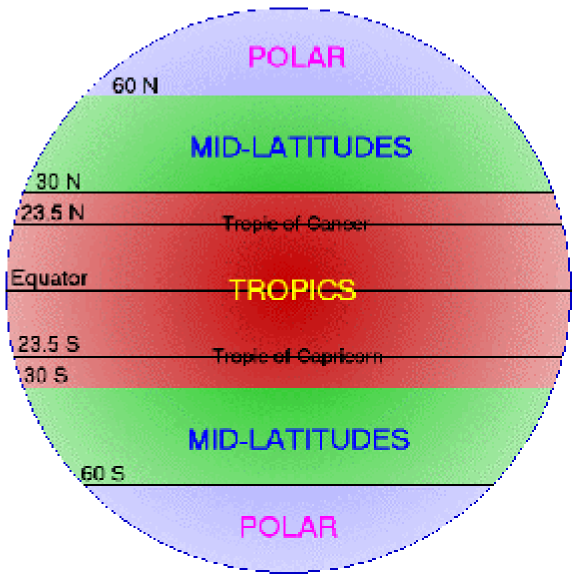


Figure 3-4 Parallels of latitude

3.4.3 Angle of latitude

The latitude is defined, in degrees, by the angle between its parallel of latitude and a point on the equator. The equator itself has latitude 0. All the places on the same parallel of latitude have the same latitude.

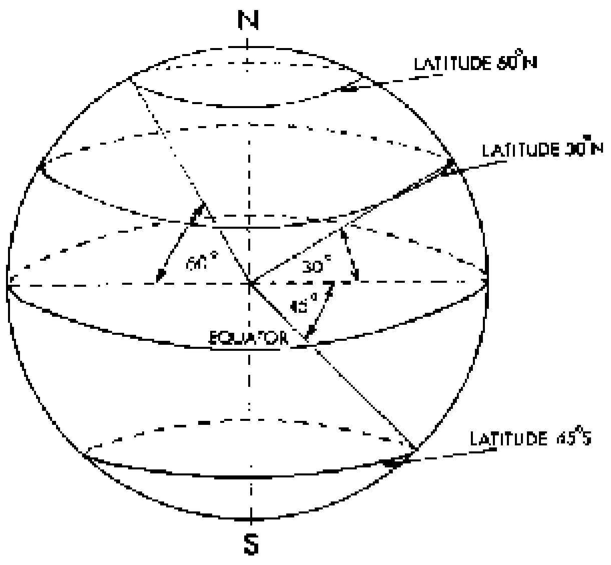


Figure 3-5 Angle of latitude

3.5 Longitude

One can define a place East or West from the equator, this is known as the longitude.

3.5.1 Polar Axis

The polar axis runs from the North Geographic Pole (True North) through the center of the earth to the South Geographic Pole (True South). The polar axis is the axis on which the earth itself rotates (revolution) causing day and night.

3.5.2 Prime Meridian

The prime meridian (longitude 0) runs through the Royal Observatory in Greenwich, England. It is the half of the great circle that connects the two ends of the Polar Axis. The other half, the ante meridian, passes down the Western side of the Pacific Ocean (longitude 180). So the Parallel to this great circle are great circles called meridians of longitude.

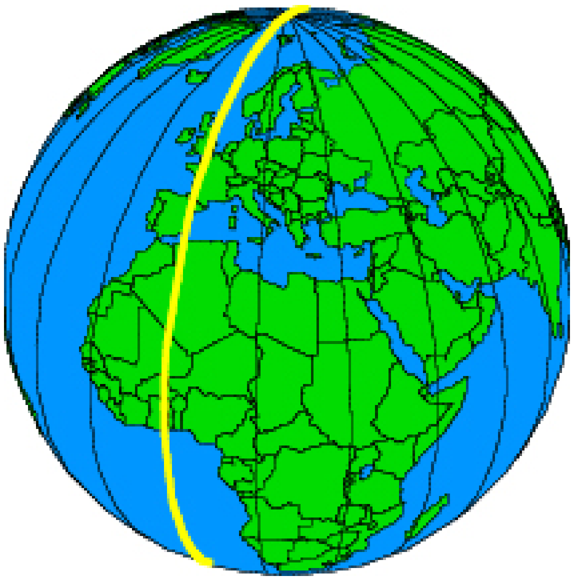


Figure 3-6 Prime meridian

3.5.3 Meridians of longitude

Meridians of longitude are separated by 15 degrees. There are meridians of longitude both on the Eastern and Western Hemisphere.

3.5.4 Angle of longitude

The longitude is defined as the angle between its meridian of longitude and the prime meridian.

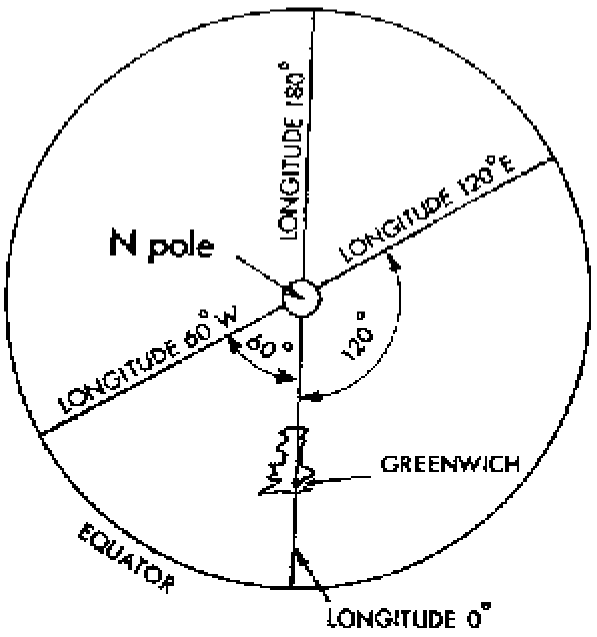


Figure 3-7 Angle of longitude

3.6 Altitude

Altitude is the vertical distance of a level, point, or object, measured from Mean Sea Level (MSL). In Flight Gear also the Average Ground Level (AGL) to compensate big differences in ground level.

3.7 Coordinate Systems

Internal, all FG scenery is defined using a cartesian coordinate system centered at the center of the earth.

3.7.1 Geocentric Coordinate System

Geocentric coordinates are the polar coordinates centered at the center of the earth. Points are defined by the longitude, latitude and from the center of the earth. Geocentric coordinate systems are conventionally taken to be defined with the x-axis through the insertion of the Greenwich meridian and equator. The geocentric coordinate system is based on the consideration of the earth as a perfect spere. Because the earth is not a perfect sphere, a more realistic coordinate sytem is the Geodetic Coordinate System. The LaRCsim flight model uses the geocentric coordinate system.

3.7.2 World Geodetic System 1984 (WGS 84)

Because the earth is not a perfect sphere, the mass center of the earth is not exactly the geographic center of the earth. The earth's physical surface is a tangible one encompassing the mountains, valleys, rivers and surface of the sea. It is highly irregular and not suitable as a computational surface. A more smoothed representation of the earth is the Geoid.

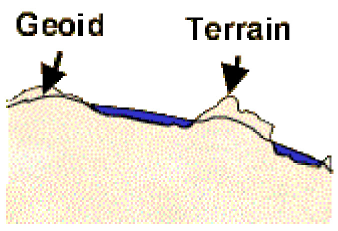


Figure 3-6 Geoid

The ellipsoid is a smooth mathematical surface that best fits the shape of the geoid and is the next level of approximation of the actual shape of the earth [WON02].

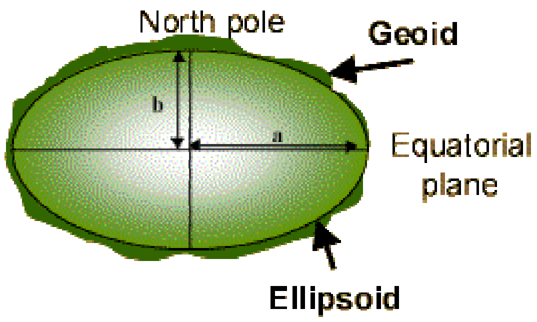


Figure 3-7 ellipsoid

Geodetic coordinates are represented by longitude, latitude, and elevation above sea level. These are the coordinates on maps. GPS (Global Positioning System) also works with Geodetic Coordinates since the satelites are dynamically influenced by the mass center of the earth. Also maps are based on the Geodetic system. In Flight Gear this coordinate system is used to present typically map data.

3.7.3 Geocentric vs. Geodetic coordinates

The difference between the Geocentric and the Geodetic Coordinate System is that the consideration of the earth as a sphere or as an ellipsoid respectively. For this reason the coordinates of both systems differ as they are referenced from the center of the earth.

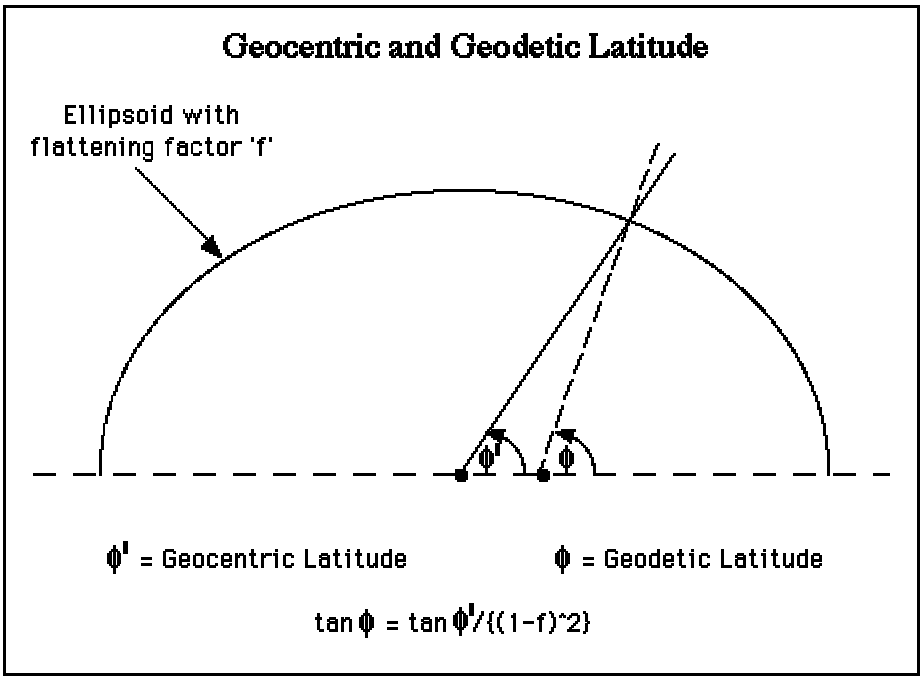


Figure 3-8 Geocentric and Geodetic Latitude

Because Flight Gear uses both the Geocentric and Geodetic coordinate system, it needs to convert geocentric coordinates to geodetic ones. For this it makes use of converters available in the Mathematical libraries of Sim Gear (see section 2.1).

3.7.4 Cartesian coordinates

The Cartesian coordinate system is a system for specifying the locations of a point in a plane (or space) by means of its distances from a fixed origin along two (or three) fixed, mutually perpendicular axes [CLU98]. Representing the earth in a Cartesian coordinate system will cause little deviations the earth represented by a Cartesian coordinate system will now be described. The origin of this system is the center of the earth. The earth's equator is represented by the x-axis. For the z-axis, the prime meridian is chosen. Finally the y-axis, runs through somewhere in the Indian Ocean [CLO99].
For internal representation of the scenery, Flight Gear needs the Cartesian coordinate system.

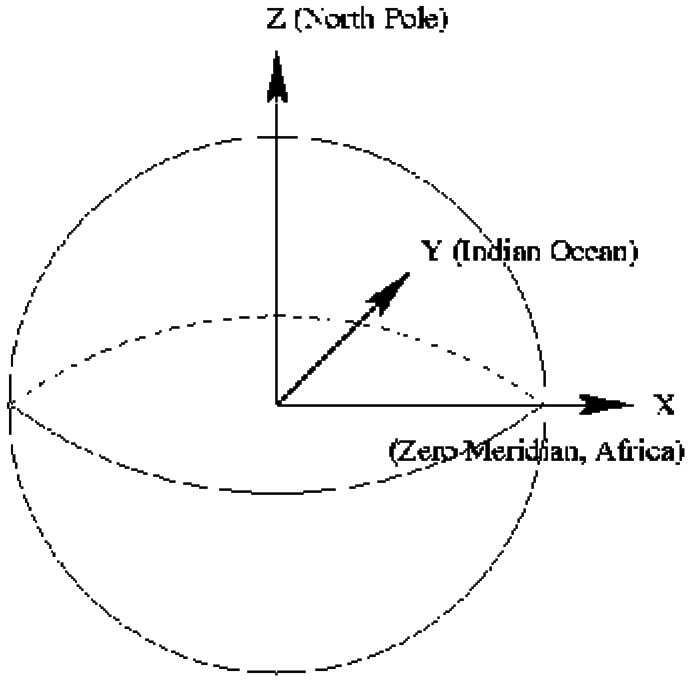


Figure 3-9 Cartesian Coordinate system

3.8 Formulas Geocentric coordinates

Depending on its heading, an aircraft is flying in North or South direction, increasing and decreasing its latitude respectively. Also it can be in- and decreasing its longitude, depending on its East direction. To calculate the latitude, the North distance of the aircraft i.e. the distance to the equator is needed.

3.8.1 North distance

From (sub)section 3.3.2 is the angle of latitude is defined as the angle between its parallel of latitude and a point on the equator. Considering the earth as a perfect sphere, the length of the equator and prime meridian are the same: the radius of the earth. So at the top of the prime meridian (latitude 90), the distance to the equator will be the half of the equatorial radius. At the prime meridian itself the North distance will be zero. This implies the North distance can be defined as:

$$s_{north} = \sin(\gamma) * r_{equatorial} \qquad (3.\_1)$$

note: sinus in radians

Where s_{north} = North distance, γ = latitude and $r_{equatorial}$ = equatorial radius.

3.8.2 East distance

The longitude is defined as the angle between its meridian of longitude and the prime meridian (section 3.4.5). Assuming that the equator has the same length as the radius of the earth (Geocentric), the East distance (s_{east}) can be calculated by:

$s_{east} = \sin(\eta) * \cos(\gamma) * r_{equatorial}$ (3_2)

note: cosinus in radians

From the trigonometry is known that: $\sin(\alpha) * \cos(\beta) = ½[\sin(\alpha-\beta) + \sin(\alpha+\beta)]$
-> $s_{east} = ½[\sin(\eta-\gamma) + \sin(\eta+\gamma)] * r_{equatorial}$

CHAPTER 4: DATA STORAGE

Flight Gear is developed in the C++ language. This chapter discusses containers provided by the C++ Standard Template Library (STL), which can be used to store MPE data. The chapter starts with an introduction in the first section. Section 4.2 gives a brief introduction to the STL, describes what a container is and compares different containers provided by the STL.

4.1 Introduction

The MPE System will enable the exchange of position data. For this reason, containers are needed to store airplane-id's and their positions. Containers can also be used to (temporally) store data packets. Several containers are that available at the C++ Standard Template Library as well as the STL itself are described in this chapter. The chapter will end with conclusion describing which container accomplishes the requirements.

4.2 The Standard Template Library

A template is a generic function i.e. a general algorithm applicable to various types of data. The Standard Template Library (STL) provides the framework for building generic, highly reusable algorithms and data structures. It is a C++ library of container classes, algorithms, and iterators. Each of these templates can contain any kind of object.

4.2.1 Containers

A container is an object that contains other objects. Containers provide methods to access the elements. To make generic algorithms i.e. algorithms that operate on different kind of containers, the methods make use of iterators.

4.2.2 Iterators

Iterators are generalized pointers i.e. objects that point to other objects independent of the kind of container. They can be incremented and decremented to access the next or previous element respectively.

There are six different iterators:

- Input Iterator: provide read access and permit single pass algorithms;
- Output Iterator: provide write access and permit single pass algorithms;
- Forward Iterator: can be mutable or immutable and provide multi pass algorithms but can only step forwards;
- Bidirectional Iterator a Forward Iterator that can step also backwards;
- Random Acces Iterator: allow arbitrary offsets;
- Trivial Iterator: can be dereferenced to refer to some other object.

There is a collection of different containers using the different iterator properties.

4.2.3 Sequences

A sequence is a container that stores objects in a strict linear order. Sequences have a variable length and elements can be inserted and removed.

Sequences can be distinguished in:

- Vector:
  A vector is a random access sequence. It provides constant time insertion and removal at the end and linear time insertion at the beginning or in the middle. Because the memory management is done automatic, the number of elements may vary dynamically;
- Dequeue:
  The name deque stands for "doubly-ended" queue. Deques are vectors that provide constant time insertion and removal also at the beginning of the sequence;
- List:
  A list supports both forward and backward traversal. Because lists are doubly linked lists, they will not have invalidate iterators after insertion or removal;
- Slist:
  The S stands for single, slists are single linked lists and therefore only provide forward traversal;.
- Associative Container:
  An Associative Container doesn't support insertion at a specific position. Each element in an Associate Container has a key. The value types of an Associative Container are not assignable.

4.2.4 Associate Containers

There are three refinements of the Associative Container:

- Sorted Associative Container:
  This container compares the keys of his elements. It considers two keys to be the same if neither one is less than the other;
- Simple Associative Container:
  The elements of a Simple Associative Container, have the same type of keys;
- Unique Associative Container:
  All the elements of a Unique Associative Container have a unique key;
- Multiple Associate Container:
  An Associate Container without the restrictions of a Unique Associate Container so it can contain elements with the same key;
- Pair Associate Container:
  This Associative Container associates a key with some other object. Both the object and corresponding key are stored together as a pair.

4.2.5 Combinations of different Associate Containers

Combining the different Associative Containers lead to new containers:

- Set:
  A set is a combination of a Sorted Associative Container, a Simple Associative Container and a Unique Associative Container. Sets can be used to store keys that have unique keys;
- Multiset:
  To store multiple keys which have the same value, multisets can be used. A multiset is a Sorted Associate Container, a Simple Associate Container and, unlike a set, a Multiple Associate Container Single Associate Container;
- Map:
  A map is a combination of a Sorted Associate Container, a Pair Associate Container, and a unique Associate Container what means that it stores paires with unique elements;
- Multimap:
  In contrast to a map, a multimap has no limit on the number of elements with the same key. A multimap combines a Sorted Associate Container Multiple Associate Container and a Pair Associate Container.

CHAPTER 5: PROBLEM DEFINITION

In this chapter several expected problems involved in the development of a client/server MPE are being discussed. The chapter starts with an overview of appropriate client/server architectures and the expected problems related to them. In the next section (5.2), Endian handling is introduced defining the difference between Little an Big Endian as well as its relation to MPE development.

5.1 Client/Server strategies

As described in the problem setting, the client/server architecture provides the best possibilities for the intended MPE. Developing a client/server system, one has to consider overhead, IO-bottlenecks, data corruption and deadlocks. These problems are discussed in the following sections as well as some possible client/server architectures.

5.1.1 Overhead

Every time a new process is started, the operating system has to allocate memory to it, initialize libraries, schedule the process, etc. This is what is meant in this chapter by overhead.

5.1.2 IO-bottleneck

One of the problems that can arise with multi-processing (described in a later section) is that of the IO-bottleneck. This problem can be explained using a metaphor of a supermarket. Consider a supermarket with cash register for every customer. For every new client session, a checkout clerk must be created. If the checkout clerks are fast i.e. they can handle more than hundred clients per second, for most of the time they have nothing to do than waiting for a customer. If the clarks are slow an IO-bottleneck will emerge [TAN97].

5.1.3 Data corruption

Data corruption can be caused by a wrong memory allocation as a result of failed processes. When using a multi threading architecture (described in a later section) the chance of data corruption will be bigger. This is because within a thread multiple processes are able to access the same memory.

5.1.4 Resources

During program execution, different resources will be used like: a database, a network connection, memory and peripherical devices.

Resources can be divided into:

- preemptive resources:
  resources that can be taken away from a process
  example: memory;
- nonpreemptive resources:
  resources that can not be taken away from a process
  example: printer.

5.1.5 Deadlocks

When a process wants to use a resource that is in use (by another process), it will have to wait is therefore blocked. This can be dangerous as processes that our blocked cannot free their resources.

Example

Process A is using resource A in the same time when process B is using resource B. If process A wants to use resource B, it will have to wait until process B will free it. So process A will block. Next, if process B wants to use resource A, it will also block because process A can not free resource A. This situation is known as 'dead lock.' A deadlock arises when each process has to wait for an event that must be caused by another process. Deadlocks will only arise when the resources are non-preemptive.

Because the MPE system will use network connections (non-preemptive) the danger of deadlocks exist.

5.1.6 Multi-processing

A process is a running program including a program counter, registers and variables.
In a multi-process architecture, an individual process is dedicated to each simultaneous connection. A process in a client/server environment contains the initialization and the data exchange.

5.1.7 Single-processing

To avoid deadlocks and data corruption, one single process can be used to handle the client connections. This will also cause less overhead. The risk of this single processing is that if one connection fails the entire process including all client connections can fail.

5.1.8 Multi-threading

A system that is multi-thread capable allows programs to split tasks between multiple execution threads. Mathematical computations on large amounts of scientific data can be quite intensive and are ideal candidates for threading on systems with multiple CPU's [RES01].

In a multi-threaded architecture, multiple independent threads of control are employed within a single shared address space. Each thread performs all of a transaction’s initialization steps and services a connection completely before moving on to service a new connection [SOU00].

Multi-threaded systems will not have an IO-bottleneck since the (sub)processes can be divided over multiple threads. But there is still overhead because every thread needs its own stack and there is also a scheduler needed. Also, the development of multi-threaded applications is very complex to avoid deadlocks and data corruption.

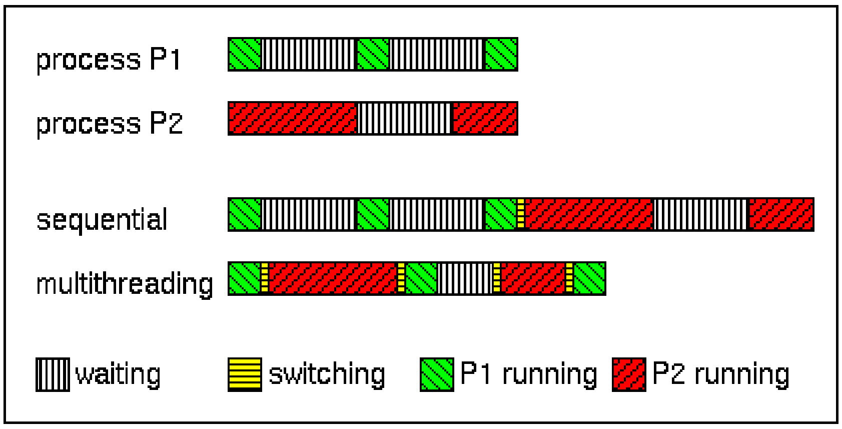


Figure 5-1 Multi-threading

5.1.9 Multiplexing IO

Multiple clients must exchange data with the server. When a single-processing architecture is used, the clients will have to wait for each other in a queue. Using multiplexing IO the server will switch between the client connections so fast that it seems the connections are handled simultaneously (pseudo parallelism).

5.2 Endian handling

One of the requirements of the MPE is that it must be appropriate to different processors (portable). This means that the system must be independent from the order in which multi-byte data types are stored.

5.2.1 Floating-point

The exchange of positions between clients (via the server) needs data (latitude, longitude, altitude, etc) that has to be stored in multiple data types. Flight Gear uses doubles to store this data. A double (double-precision floating-point) needs 64 bits of memory. Because we want to keep the data packets small, we choose floats instead of doubles. A float is a 4 byte data type.

Byte 0	Byte 1	Byte 2	Byte 3
MSB			LSB
32 bits			

Figure 5-2 Floating-point data type

As one can see in the picture, byte 0 and byte 3 are the Most Significant Byte (MSB) and Least Significant Byte (LSB) respectively. This order refers to the mathematical positions of the bits and not to their actual physical locations [RFC1014].

According to the order of MSB and LSB, there are two ways to store a multi-byte data type: ‘Little Endian’ and ‘Big Endian’ [EEO96]. Intel processors use Little Endian byte order whereas Motorola processors use Big Endian byte order.

5.2.2 Little Endian

Little Endian means that the low-order byte of the data type is stored at the lowest memory address, and the high-order byte at the highest address (little end first).
Figure 5-3 shows how a float is stored in Little Endian order.

Base	Address + 0	Byte 0
Base	Address + 1	Byte 1
Base	Address + 2	Byte 2
Base	Address + 3	Byte 3

Figure 5-3 Storing a float in Little Endian order

5.2.3 Big Endian

As the opposite of Little Endian, Big Endian means storing the high-order byte (MSB) in memory at he lowest address, and the low-order byte (LSB) at the highest address (big end first). Figure 5-4 shows how a float is stored in Big Endian order.

Base	Address + 0	Byte 3
Base	Address + 1	Byte 2
Base	Address + 2	Byte 1
Base	Address + 3	Byte 0

Figure 5-4 Storing a float in Big Endian order

CHAPTER 6: GENERAL DESIGN

This chapter contains the analysis of the MPE System. In section 6.1 the statement of purpose is given. The next section (6.2) the functionality is described and showed by a context diagram, a processor environment model, entity relationship diagram and a data dictionary respectively.

6.1 Statement of purpose

For more research and AI possibilities there is a need for a multiplayer flight simulator possibility. Players should be able to exchange their positions via a network so they can fly in the same area. Also a player has to be able to configure the multiplayer mode. An operator able to configure the entire system and switch it on and off should control the system. The multiplayer should work with different flight simulator versions, different airplanes and different flight models.

6.2 Analysis

For the analysis presented in this section the book "A practical guide to Real Time Systems Development" is used [SGS93]. The border of the system is showed using a context diagram in section 6.2.1.

6.2.1 Context Diagram

A player enables the MPE system by enabling the multiplayer mode in the flight simulator, which is not a part of this system so the diagram shows only a control flow from the player to the system. Players, more than one is necessary for a multiplayer mode, are also able to configure the system represented by the data flow 'configuration'. Every player must have a player name but it is not necessary to use a unique name. Operator is the second human terminator who communicates with the system. The operator, there is no need for more than one, can also control the system via the flows 'configuration' and 'E/D'. Finally a terminator represents the flight simulator. This is because the flight simulator is a separate system. Of course the MPE system cannot run without the flight simulator but the flight simulator will still be able to run in single player mode. Now the systems environment is known, one can zoom in to get a better understanding of the system 'ansich'.

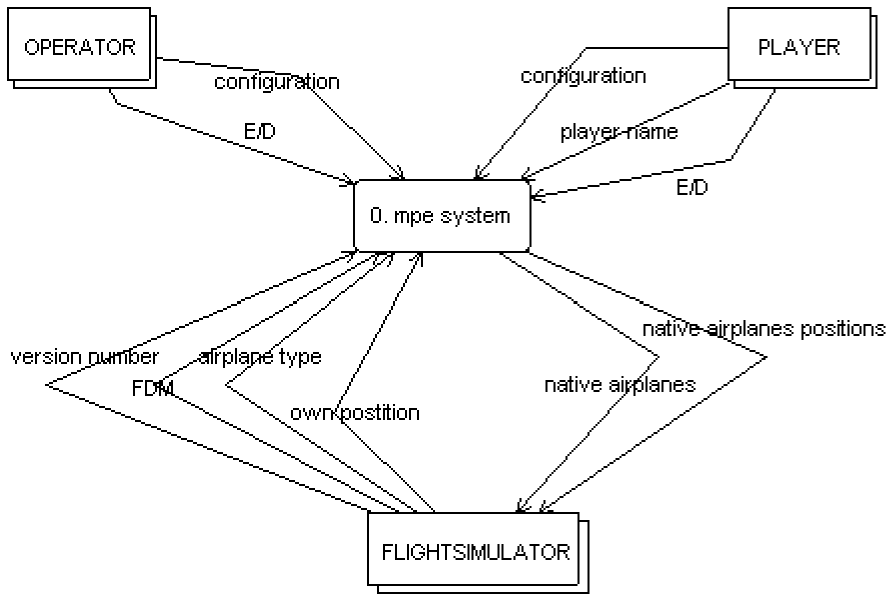


Figure 6-1 Context Diagram MPE

6.2.2 Processor Environment Model

The data exchange must take place via a network, which means that the system can be thought of as a multiprocessor system. Therefore the system is split up in a server system and a client system, implemented on separated computers (read processors). The operator will control the server system. Players can only control their own client system. For example they can configure the frequency with which they will receive updates i.e. other client positions with the data flow 'max downstream'. Different clients will be able to exchange position by communication with the server. Therefore each client will send its position to the server and will receive the positions of other clients. To keep track of the different clients logged i.e. players flying at the same time in the same area, the server needs the data flow 'identification' from all of his clients. As a response to a clients 'version number', the server will send a 'required version number' back. All the data flows used in the Processor Environment Model, will be described in the Data Dictionary, later. The relationship between the server and clients can be illustrated using an Entity Relationship Diagram.

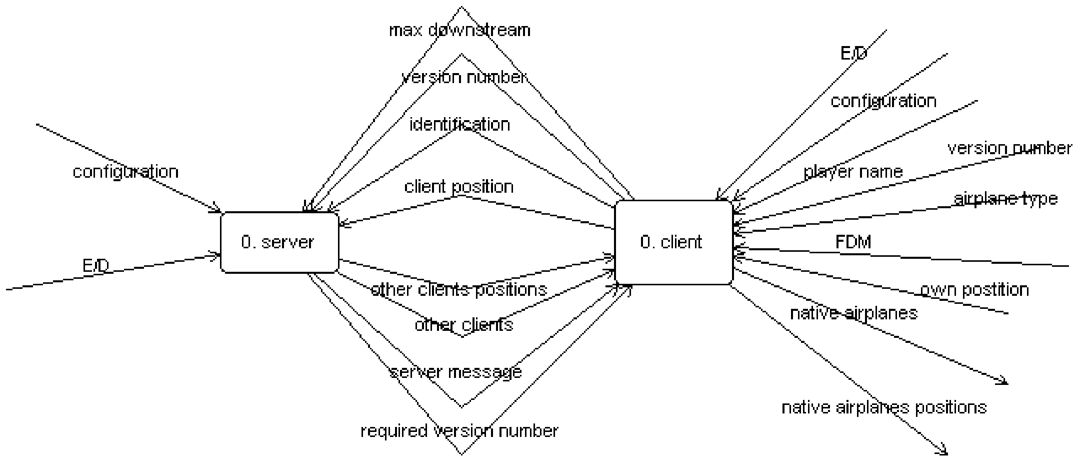


Figure 6-2 Processor Environment Model MPE

6.2.3 Entity Relationship Diagram

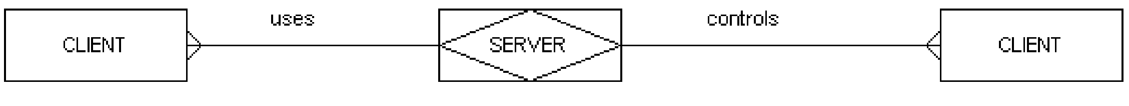


Figure 6-3 Entity Relationship Diagram

As the diagram shows there are *n* clients but there is only one server. The server controls the clients and the clients use the server to communicate with each other.

6.2.4 Data Dictionary

The names used in the previous diagrams are defined in the data dictionary below.

configuration	submitted settings to the client system made by a player
max downstream	the maximum speed with which the client is able to receive data from the server
version number	the version number of Flight Gear used by the player
player name	the non-unique name chosen by a player
airplane type	FlightGear consists of 9 different airplane types
FDM	Flight Dynamics Model, a model for flight controls
identification	player name + airplane type + FDM
other clients	a list of identifications of other players
perceivable airplanes	airplanes that are in the vicinity
required version number	the version number that is needed to participate the multiplayer environment
other clients postions	the positions of the airplanes controlled by other players
own position	the position of the airplane a player itself
client position	the position of the airplane of a player
native airplanes positions	the positions of the airplanes controlled by other players within a certain range

Figure 6-4 Data Dictionary MPE System

From the processor environment model one can zoom in on either the client or server system, this will be done in chapter 8 and 9 respectively.

CHAPTER 7: NETWORK DESIGN

This chapter contains some theory about network protocols used within a client/server architecture, to be used by the MPE to transport position data between (network) players. In the first section, the role of TCP based protocols in the development of the MPE is given. The second section contains a comparison between TCP and UDP. Data compression techniques are discussed in the third section. To increase the network reliability, one can use confirmation handling. An appropriate confirmation handling method, which can be used for the MPE, is discussed in section 7.4. Finally the design of the MPE Network protocols is presented in section 7.5.

7.1 Introduction

Some games contain a multiplayer mode that supports multiple players on the same PC. Depending on the type of game it is sometimes possible to display both player characters on the same screen, others have to split the screen or sometimes it is possible to use multiple screens. This type of multiplayer support is limited because players need to use the same PC.

There are also two-player games with network support to allow both players to use their own PC and play 'head-to-head.' This is usually implemented as one PC acting as a server and the other acting as client using a TCP/IP based protocol. Using an IP address this multiplayer mode can be played on a LAN as well as on a WAN or internet.

A more sophisticated multiplayer engine would allow multiple players to play via a client/server system. Gameplay is handled having each user's game client communicate with the server. The server is responsible for passing information on to the other users. The server also has to schedule all of his clients i.e. allowing clients to send and receive information sequentially. For this a TCP/IP based network protocol is needed. The next section contains a brief discussion about TCP and UDP.

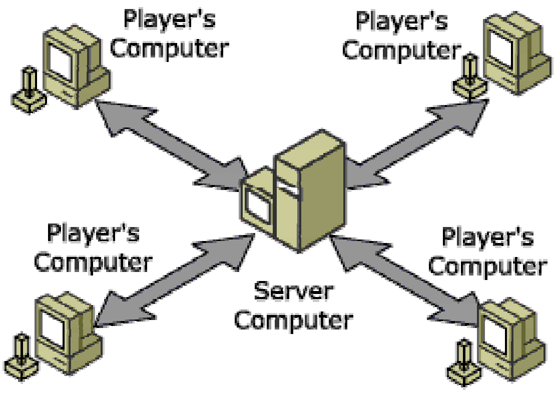


Figure 7-1 Multiplayer via client/server architecture

7.2 UDP vs TCP

User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) are transport layer protocols for IP networks. They differ mainly in reliability. UDP is called a connectionless protocol because it does not perform retransmission of data. UDP is also called a 'fire-and-forget' protocol.

TCP is a connection-oriented protocol that enables an established connection between two hosts for a reliable exchangement of data. TCP guarantees that data packets will be delivered and the packets will arrive in the right order. Because TCP enables retransmission of lost or corrupt data packets it is more suitable for real time data communication, this is why UDP will be the best to use for the MPE.

7.3 Data compression

To keep the data packets small, a free data compression library can be used. A second benefit of using a compressing algorithm is that it can check whether the data (before compressing) is corrupt or not. Of course the situation can arise that the compression itself is corrupt but this will be discovered during decompression. The decision is made to provide the data compression as an option via a compression identifier (see figure 7-3).

In this section two data compression libraries will be discussed.

7.3.1 LZO

The data compression library LZO (LeMPEl-Ziv-Oberhumer) provides a number of compression levels each with a different compression rate. Low compression rates are less fast than compression levels with relatively high compression rates.
LZO features:

- very fast compression and decompression;
- no memory needed for decompression;
- requires 64 kB of memory for compression and has a compression level which only needs 8 kB;
- has compression levels at which the compression ratio can be increased by generating pre-compressed data;
- is portable across platforms.

LZO is a block compression algorithm i.e. it compresses and decompresses a block of data.

7.3.2 Zlib

A data compression library with a very high compression is Zlib. Zlib is not as fast as LZO but is more focused on the compression itself rather than compression speed.

Features of Zlib:

- platform independence;
- can use a number of different compression methods;
- has a very high compression rate;
- the memory footprint is independent of the input data and can be reduced.

7.3.3 Conclusion

Both LZO and Zlib can be used to implement in the MPE. When a fast compression speed is needed, for example if there is a lot of data traffic, LZO is the most appropriate to use.

7.4 Confirmation handling

As said in section 7.2, UDP is a fire-and-forget protocol because it does not perform retransmission of data. To increase the reliability of the data transmission, some kind of confirmation is needed. This can be by adding request id's of received packets to confirmation slots (see figure 7-4) of the data packet that will be send back (since all the data packets will be send from or to the server). There is decided to implement confirmation handling in the form of a confirmation handler. This confirmation handler will check the confirmation flag, if this flag is set it will add a request id with which the receiver can confirm that the packet has been received.

Both the sender and receiver need a list with packets that are sent and packets that are received. So two lists are needed:

1. a list of packets that are sent but not yet confirmed;
2. a list of packets that are received.

The first list exists of data packets that are sent by the workstation itself and needs to be confirmed. This list can be implemented as a map containing pairs of data packets as objects and their corresponding request id's as keys (see section 4.2.5) since maps:

- are portable;
- are fast;
- provide random acces;
- use iterators;
- have a variable length.

The second list exists of data packets that are received. Of these data packets only the request id needs to be stored. This maximum size of this list will be the equal to the maximum of confirmation slots. Although this list will only contain request id's, it will also be implemented using a map.

The confirmation handling as described in this section is illustrated in by the description of different (exception) scenarios in Appendix B.

The decision is made to implement optional confirmation handling. One can choose to use confirmation handling by setting the 'confirmation flag' (see figure 7-5). If the confirmation flag is not set, the confirmation slots will be empty. This empty space cannot be used for data, but if one chooses to use compression (as described in the previous section) the empty data will not lead to much redundancy.

7.5 MPE Network protocols

In section 7.2 is are the TCP and UDP protocols where discussed. The UDP protocol is used for the MPE network modules. This section describes the design of the protocol levels at the application and transport layer using the UDP protocol.

7.5.1 Data Encapsulation

Before data is sent to or from a server, it will be 'encapsulated' to a UDP data packet i.e. on every next layer blocks of data (headers) are added until the data packet is ready to send. Data encapsulation works like a stack: the sending system adds headers from the application layer to the network layer and the receiving system removes the headers from the network header to the actual data at the application layer. So the data that is add at last will be removed at first (LIFO).

The application layer contains the functional information at MPE level 0. In most cases this will include the position information but it can also contain identification data. The data at this level is stored in a (static) buffer called MPE data buffer.

Before the UDP header is added to the MPE data buffer, 3 MPE protocol levels are passed:

- protocol level 2: server control control / type of service (TOS);
- protocol level 1: compression;
- protocol level 0: raw data.

Application layer	|MPE DATA BUFFER |
Transport layer	|MPE DATA BUFFER |UDP HEADER |
Network layer	|MPE DATA BUFFER |UDP HEADER | IP HEADER

Figure 7-2 Data encapsulation UDP protocol

Application layer	Protocol level 3	Functional information
Transport layer	Protocol level 2	Server Control / TOS
	Protocol level 1	Compression
	Protocol level 0	Raw data

Table 7-1 Data encapsulation protocol layers

7.5.2 Protocol level 0: raw data protocol

At this level only the implementation of the data-buffer itself exists. In the protocol this is the raw part that goes into the UDP-datagram datafield i.e. the raw data buffer: 'MPE data buffer'.
The net library Plib provides a dynamic netBuffer, but we choose to develop a new static buffer instead for two reasons:

- using dynamic buffers, memory leaks can occur;
- static buffers enables error handling.

7.5.3 Protocol level 1: compression layer protocol

At this layer the compression will take place. One of the data compression techniques Zlib or LZO, described in section 7.2, can be used. Because it is difficult to detect whether Zlib or LZO compression is used, the (redundant) data field 'compression id's is needed. The data compression needs 16 bits in total.

There are 3 compression identifiers implemented:

- raw (none): no compression technique will be used, the data will keep its original format;
- zlib (compress): the data will be compressed using zlib: high compression rate;

- lzo (lzo1x): compression of more than 53%, compression speed of more than 4.5 Mb/s.

Only one identifier can be active for any single data packet. The server should respond using the algorithm used by the client to initialize the connection.
Although it would be better to use 'tree' algorithms, 'general purpose' compression algorithms are implemented because the latter are less difficult to implement.
The layout of the 'compression layer protocol':

10b length message	6b Compression identifier	
(Compressed) data, allowed maximum +/- 560 bytes		

Figure 7-3 Protocol level 1: compression layer protocol

7.5.4 Protocol level 2: Server control / Type-of-Service layer

In unpredictable situations the MPE server may need some special information. This information is contained in the header of this layer. It is primary use is to force the server to handle the packet in a special way, such as:

- Send to server / Field Of View (FOV) only;
- Loop back packet;
- Do not parse the packet but forward it to all / in FOV;
- Set Type Of Service;
- Sequence protection.

0/0	1/15	2/16 3/31
4b size header	12b FLAGS	16b TOS (Type of Service)
16b Sequence no. (from client/TOS)		16b Sequence no. (from server/TOS)
16b Confirmation Replies slot 1		16b Confirmation Replies slot 2
16b Confirmation Replies slot 3		16b Confirmation Replies slot 4
Datafield (maximum 512 bytes)		

Figure 7-4 Protocol level 2: Server control / Type-of-Service layer

1	Confirm msg	Message must be confirmed by server (either by empty msg, with the confirm code (sequence no. client) in it or a appropriate response.
2	Do not Parse	Server should not try to parse the message. Instead it should only forward it to the given targets (in fov or to all)
3	To Server	Message is meant for the server and should never be forwarded.
4	To ALL	Message should be forwarded to all clients on the server.
5	To FOV	Message should be forwarded to all clients within fov of sending client.
6	Loopback	Message should be sent back to the client (for connection testing)

7		
8		
9		
10		
11		
12		

Figure 7-5 Flags at protocol level 3

7.5.5 Protocol level 3: Functional Information layer

This layer includes the function information about position and orientation:

- latitude;
- longitude;
- altitude;
- pitch: rotation around the y-axis;
- roll: rotation around the x-axis;
- yaw: rotation around the z-axis (related to the true North);
- vcas: Calibrated Airspeed.

The figures 7-5 and 7-6 show the two different MPE data packet at protocol level 3, the gray data fields are repeated within this packet for usually ten times.

Latitude, longitude and altitude are described in chapter 2.

0 15	16 31
24b Reserved	8b Flags
32b Position latitude	
32b Position longitude	
32b Position altitude	
32b theta (pitch)	
32b phi (roll)	
32b psi (yaw)	
32b vcas (velocoity)	

Figure 7-5 MPE data packet at protocol level 3 containing functional information.

0/0	1/15	2/16 3/31	
8b flags(1)	7b min ver. sender	8b major ver. sender	8b minor ver. Sender
8b updates/sec	8b FOV (x1000)	16b Max traffic downstream (kB/s)	
8b length fields	8b Flags	16b Player ID	
8b pos nick	8b pos FDM	8b pos planename	8b pos plane id

Player name+’\r’+FDM+’\r’+plane name+’\r’+plane id (total length max 50 bytes) (nickname usually around 10 chars, fdm is around 10 too, plane names and id are 7 or so ?)	

Figure 7-6 MPE data packet at protocol level 3 containing initialization information.

CHAPTER 8: SERVER DESIGN

In this chapter the design of the MPE Server is discussed. The first section continuous the analysis from chapter 6 by presenting the analysis of the MPE Server. This analysis will be given using an extern event list, a data flow diagram and a state transition diagram respectively. In the next section (8.2), internet applications are introduced. Finally, scheduling algorithms are discussed in the last section (8.3).

8.1 Analysis

For the analysis presented in this section the book "A practical guide to Real Time Systems Development" is used [SGS93]. This analysis starts with an extern event list with which the border with the total MPE system is determined.

8.1.1 Extern Event List

To find out what processes are needed, one can consider events at which the server has to react. For this, an extern event list can be used as showed in figure 8-1. For every event, a certain response is defined. The event list is called 'extern' because it contains events from outside the MPE server system. The responses to the events indicate data flows that will be used for the data flow diagram that will be presented in the next section. As processes can be data processes, control processes of a combination of the two, the responses are classified with a D (data processes), C (control process) or C/D (control/data process).

Event	Response	Classification
1. Operator enables server	Parse configuration parameters	D
2. Server configured	Receive version number	D
3. Version number received	Retrieve required version	D
4. Version number retrieved	Exchange client info	D
5. New client identified	Start exchanging positions	C/D
6. Operator disables server	Stop exchanging positions	C

Figure 8-1 Extern Event List MPE Client

8.1.2 Data Flow Diagram server

Like the client system, all the functions a controlled by a control process using control flows. The server system consists of three stores: clients, client positions and exchange parameters. These are necessary to keep track of the clients and the information they need. A control flow can be a trigger. Triggers continuously control a data processes. Other control processes are indicated by E/D (Enable/Disable) and present a binary control flow for switching a data process on and off.

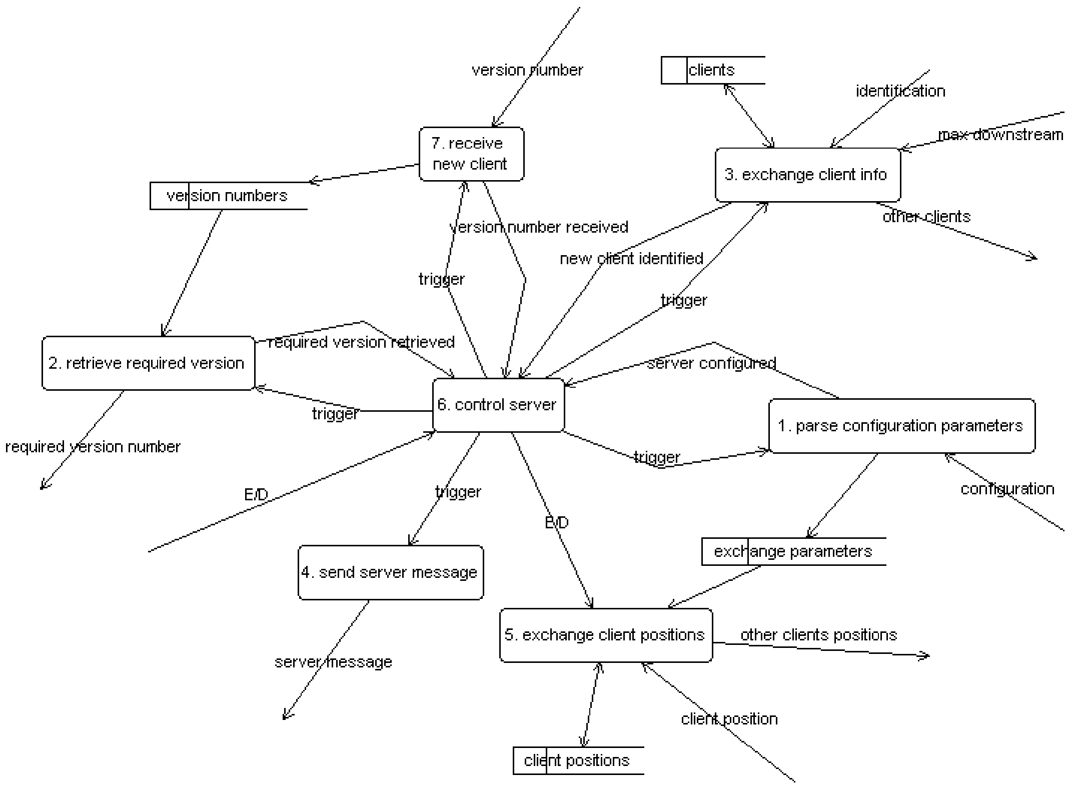


Figure 8-2 Data Flow Diagram MPE Server

8.1.3 State Transition Diagram

Because this system is a *real time* system, control is needed to guarantee that the actions/processes will be done in the right order. State Transitions Diagrams are appropriate diagrams to show the how the control has to take place. The data processes from the data flow diagram of figure 8-2 are represented by states in the state transition diagram of figure 8-3. In this state transition diagram, the events presented in the event list of figure 8-1 are represented as conditions.

After the first client logged on, the server should be able to accept more. Therefore after the server is configured, the states: receiving version number, synchronizing, exchanging client info and exchanging client positions can be interrupted and repeated.

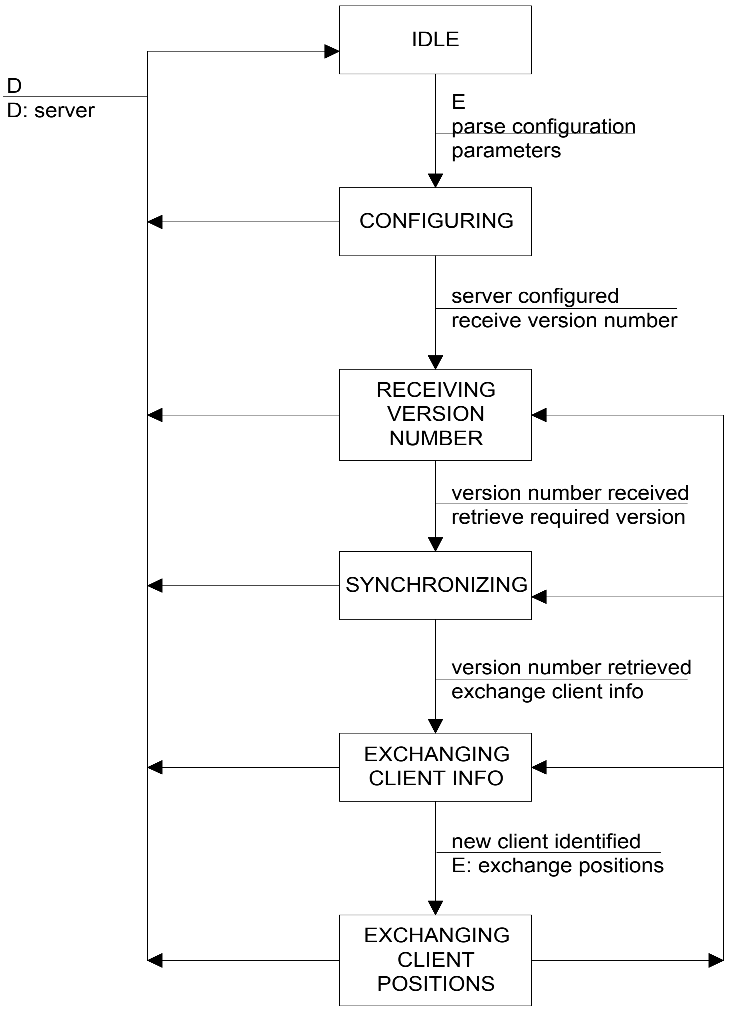


Figure 8-3 State Transition Diagram MPE Server

8.2 Internet Applications

For a good performance of the total system, it is useful to know influences on the performance of the MPE server. Since the MPE server accepts connections from clients (even connections to different clients simultaneously), it can be considered as an Internet Application. The performance of an Internet Application is constrained by the CPU speed rather than the available network bandwidth [SOU00].

8.2.1 Scalabitity

Scalability is a measurement for the performance of an application to sustain its performance when some external condition changes [SOU00]. There are two different types of scalability:
System scalability: hardware (number of CPU's, memory size, etc);
Load scalability: the number of simultaneous connections.

Because the MPE server will be installed on a single processor, there is no need to consider the performance when using more than one CPU. The hardware will not consist of more than one network card either. The number of CPU's, network cards, disks, etc is defined as the *real concurrency* of the server. Good system scalability means that it is possible to run an Internet Application small system.

Other than the real concurrency, the *virtual concurrency* defines the number of supported simultaneous connections i.e. connected clients. An Internet Application that can sustain its throughput over a wide range of loads is said to have good load scalability. So the load scalability of the MPE server can be measured by its virtual concurrency.

8.3 Scheduling Algorithms

8.3.1 Introduction

Real-time systems are control oriented. The heart of a real-time system is a scheduler that orders the actions of the system scheduling.

Clients will have to send their position frequently so other clients will receive new positions and can *update* their list of perceivable airplanes i.e. airplanes that are close enough to be relevant. The server will exchange updates, i.e. new position data with each client. Therefore the server will need an algorithm so that every client will be able to send and receive updates.

There are two possible ways to organize this scheduling:

- Fair: assuming that every client can handle updates with the same speed;
- With priorities: separate 'fast' clients from slow ones using priorities.

8.3.2 Fair

In this case a simple algorithm can be used that enables every client to send and receive data after one another. This is fair because no difference is made so every client will have the same chances. Assuming that every client has the same speed is usable when all the clients are using the same local area network but it is non-realistic for data communication via the Internet.

8.3.3 With priorities

Clients with a fast connection can handle updates more frequently than clients with a slow connection. For this reason an priority algorithm can decide to make a separate fast clients from slow ones.

CHAPTER 9: CLIENT DESIGN

This chapter contains the design of the MPE Client. In the section 9.1 the analysis of the MPE Client is presented as a continuation of the analysis of chapter 6. Like the MPE Server, the analysis of the MPE Client will be described using an extern event list, a data flow diagram and a state transition diagram.
In section 9.2 the conceptual design will be given.

9.1 Analysis

Like the analysis presented in chapter 6 and 8, for the analysis of the MPE client the book "A practical guide to Real Time Systems Development" is used [SGS93]. For the MPE Client the same diagrams are used as for the MPE Server. In the previous chapter, one can find descriptions about the components of these diagrams (section 8.2).

9.1.2 Extern Event List

As with the analysis of the MPE client, the first step is the creation of the extern event list to find the data processes as a response to actions from outside the server system.

Event	Response	Classification
1. Player enables client-system by choosing multiplayer mode	Parse configuration	D
2. Client configured	Synchronize version	D
3. Version synchronized	Send identification	D
4. Acceptation received	Receive other clients	C
5. Other clients received	Start exchanging positions	C
6. Player disables client-system by leaving multiplayer mode	Stop exchanging positions	C

The event list is useful to find out what actions must take place in reaction to the occurrence of particular events i.e. the way the system should *behave*. From an extern event list one can derive the data transformations needed inside the client system.

9.1.2 Data Flow Diagram

As the data flow diagram shows, every function of the client system is controlled by control process 'control client'. 'Control client' controls the order in which the transformations will take place. 'Exchange positions' is a repeating function which can be switched on and off via the control flow 'E/D'. All the other transformations are non-repeating and therefore *triggered*. The functions return event flows after succeeding their job. The control process is shown in a lower level State Transition Diagram.

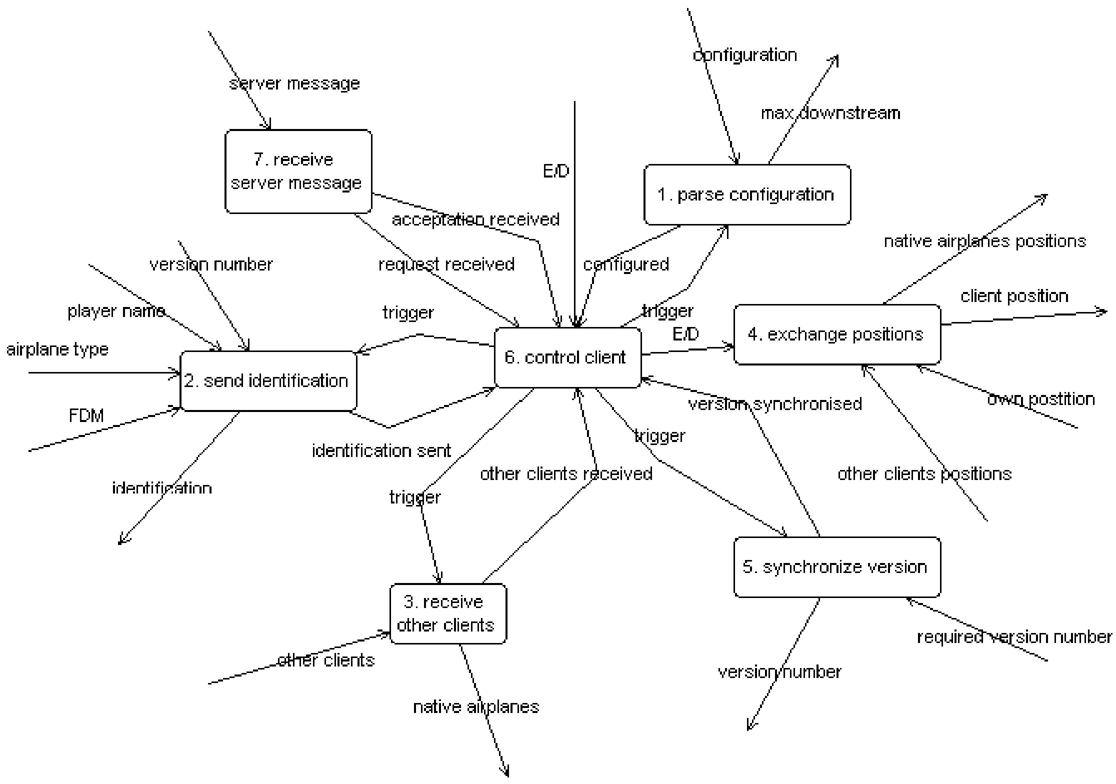
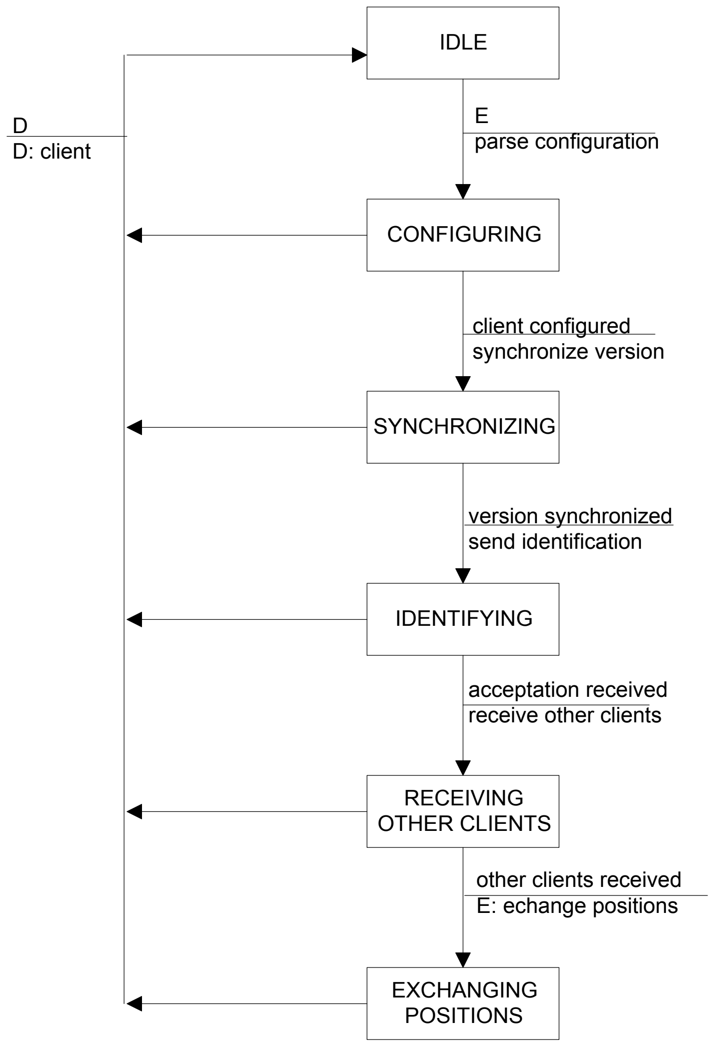


Figure 9-1 Data Flow Diagram MPE Client

9.1.3 State Transition Diagram

The STD shows the order in which the transformations will take place. Once a client is successfully configured, synchronized and identified it will receive information about other players participating the multiplayer area. Finally the client is able to exchange positions.



9.2 Conceptual design

9.2.1 Implementation in Flight Gear

The MPE client can be integrated in Flight Gear as modules:

- MPEHandlerClient: child of FG subsystem;
- list of clients: all the other clients that participate in the simulation simultaneously;
- fgClient: the module that handles the data;
- MPE Socket: channel to and from the MPE server;
- MPESocketWrapper: quick fix for correct integration of MPEHandlerClient.

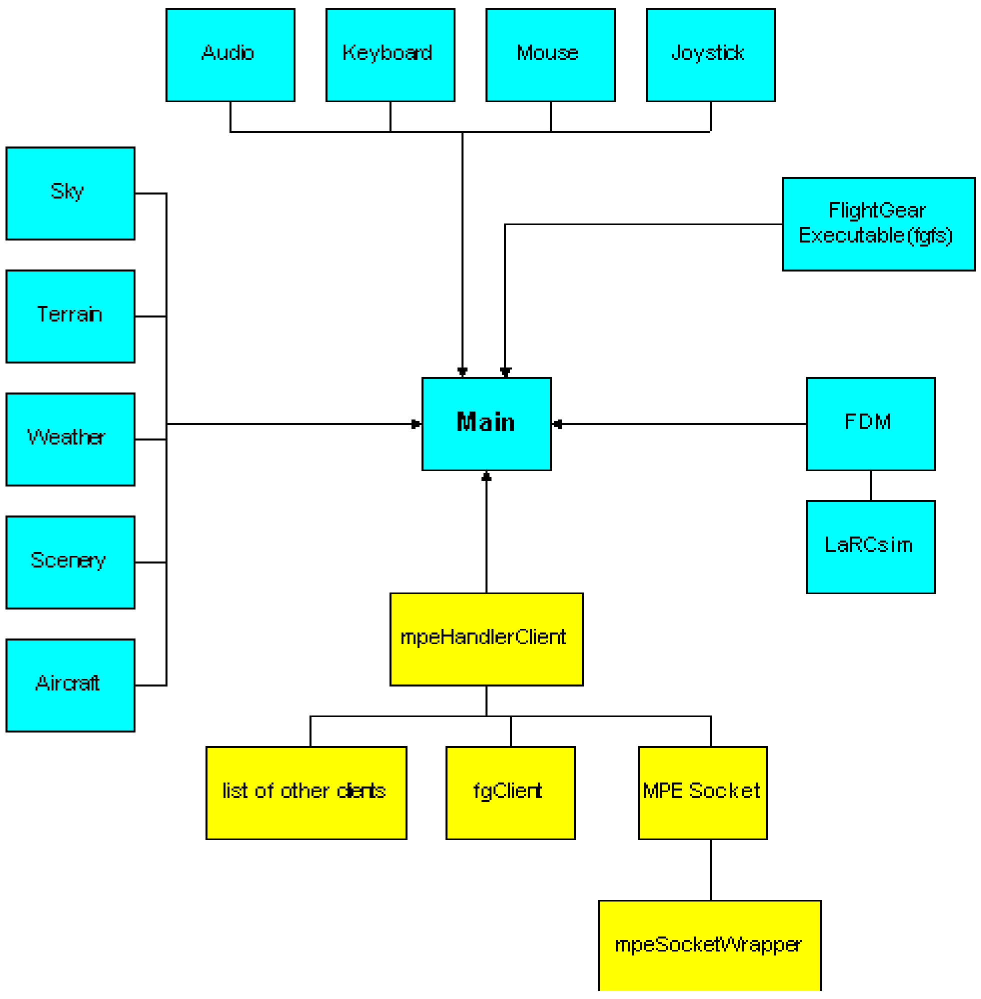


Figure 9-1 MPE client integrated in Flight Gear

9.2.2 Timestamps

Before a client sends a packet, it timestamps the packet with a value between 1 and 65535 milliseconds. When the server receives this packet, it validates the packet. If the packet is valid i.e. the time between this packet and the previous one took no longer than 10 seconds, the packet will be queued otherwise it will be thrown away. For the timestamps plibul, the utility library of PLIB is used (see section 2.1.4).

9.2.3 Prediction

During simulation coordinate information (latitude, longitude and altitude) as well as orientation information (yaw, pitch and roll) are dynamically generated. But if this data stream cannot be obtained constantly i.e. with discrete time intervals, one need to calculate them. Assuming that, in the worst case, an airplane will fly in the same direction as 10 seconds ago, one can use formulas from the dynamics with acceptable deviations.
Time stamps as described in the previous section will determine the time intervals.

The following formula is used to calculate the distance after a certain time interval:

$$s_{(t)} = v_{(t)} * \Delta t + a_{(t)} * \Delta t^2$$

Because the data packets do not contain accelerations (see chapter 7), the formula becomes:

$$s_{(t)} = v_{cas} * \Delta t$$

Where v_{cas} is the Calibrated Airspeed.

Formula to calculate the new orientation after a rotation:

$$\varphi_{(t)} = \omega_{(t)} * \Delta t \qquad (9\_1)$$

The horizontal distance(s_x) is at its maximum if the pitch(α) is zero, so:

$$s_{x(t)} = v_{cas} * \cos(\alpha) * \Delta t \qquad (9\_2)$$

note: cosinus in radians

The vertical distance(s_y) is maximal if the pitch is 90 degrees ($\pi/2$):

$$s_{y(t)} = v_{cas} * \sin(\alpha) * \Delta t \qquad (9\_3)$$

To calculate the latitude, the North distance is needed (see chapter 3). The North distance is calculated by the vertical distance and the yaw. The North distance is maximal if the yaw is zero:

$$s_{north} = s_y * \cos(\beta)$$

Substitution with (9_3) gives:

$$s_{north} = v_{cas} * \sin(\alpha) * \cos(\beta) * \Delta t^2 \qquad (9\_4)$$

note: sinus in radians

After a certain time interval, the rotation could be different. Therefore it is better to use the average rotation calculated by the old yaw and the *predicted rotation*, calculated by (9_1):

$\varphi_{(t)} = [\ \varphi_0 + (\omega_{(t)} * \Delta t)\ ] / 2$ (9_5)

Substitution of (9_5) for the pitch in (9_4) gives:

$s_{north} = v_{cas} * \sin(\alpha) * \cos \{ [\ \beta + (\omega_{(t)} * \Delta t)\ ] / 2\ ] \} * \Delta t^2$

From the trigonometry is known that: $\sin(\alpha) * \cos(\beta) = ½[\sin(\alpha-\beta) + \sin(\alpha+\beta)]$

-> $s_{north} = v_{cas} * ½ [\ \sin (\alpha - \{ [\ \beta + (\omega_{(t)} * \Delta t)\ ] / 2 \}) + \sin (\alpha + \{ [\ \beta + (\omega_{(t)} * \Delta t)\ ] / 2 \})\ ] * \Delta t^2$ (9_6)

Using (3_1) for the latitude(γ):

$\gamma = \operatorname{asin} (s_{north} / r_{equatorial})$ (9_7)

Finally, substitution of (9_6) in (3_1):

$\gamma = \operatorname{asin} \{ v_{cas} * ½ [\ \sin (\alpha - \{ [\ \beta + (\omega_{(t)} * \Delta t)\ ] / 2 \}) + \sin (\alpha + \{ [\ \beta + (\omega_{(t)} * \Delta t)\ ] / 2 \})\ ] * \Delta t^2 \}$

Longitude
To calculate the longitude, the East distance (s_{east}) is needed (see chapter 3).
The East distance will be minimal when the yaw (β) is zero:

$$s_{east} = s_x * \sin(\beta) \qquad (9.2\_8)$$

Substitution of (9.2_2) in (9.2_8):

$$s_{east} = v_{cas} * \sin(\beta) * \cos(\alpha) * \Delta t^2$$

Using (9.2_5) for an average yaw:

$$s_{east} = v_{cas} * \sin \{ \beta + [\beta + (\omega_{(t)} * \Delta t)] /2 \} * \cos(\alpha) * \Delta t^2$$

From the trigonometry is known that: $\sin(\beta) * \cos(\alpha) = ½ [\sin(\beta-\alpha)-\sin(\beta+\alpha)]$

$$\rightarrow s_{east} = v_{cas} * ½ \{ [\sin (\{ \beta+[\beta+ (\omega_{(t)} * \Delta t)] /2 \} - \alpha)] – [\sin (\{ \beta+[\beta+ (\omega_{(t)} * \Delta t)] /2 \} + \alpha)] * \Delta t^2 \}$$

Using the (3.7_2) for longitude(η): $\eta = \mathrm{asin} [s_{east} / r_{equatorial}) / \cos(\gamma)] \rightarrow$

$$\eta = \mathrm{asin} (v_{cas} * ½ \{ [\sin (\{ \beta+[\beta+ (\omega_{(t)} * \Delta t)] /2 \} - \alpha)] – [\sin (\{ \beta+[\beta+ (\omega_{(t)} * \Delta t)] /2 \} + \alpha)] * \Delta t^2 \}) / \cos(\gamma)$$

CHAPTER 10: IMPLEMENTATION AND TESTS

10.1 Introduction

Flight Gear is not developed for multiplayer so this functionality must be add as separate modules. The multipilot project of Curis Olson (see chapter 1) did not lead to complete, portable and reliable modules. The sources that were developed for this project are free available but could not be used for the Flight Gear Multiplayer Project described in this thesis. The Flight Gear ATC (Air Traffic Control) module (see chapter 1) does provide functionality to visualize 3D models but does not support the visualization of 'other' aircrafts.

The MPE modules were first developed as network prototypes. These network prototypes were created, tested and debugged separate from Flight Gear. This is because of two reasons:
1. testing is more easy;
2. it takes a lot of time to compile the Flight Gear sources.

10.2 First prototype

With the first prototype, the PLIB network module was tested as well as compiling on different platforms. Testing this prototype made clear that the PLIB network module did work because the functions to write and read data to and from the socket (getData and setData), are using a null terminated string (see question 1 at Appendix C) For this reason the PLIB network module is not appropriate to send packets. This means that new MPE specific network modules had to be developed. Those were tested as the second prototype.

10.3 Second prototype

The purpose of the second prototype was to test the protocol levels 0, 1 and 2 (see section 7.5) and the portability. With this prototype connections could be established between server and clients and some information could be send. At that phase the information itself was totally unimportant because the data analyzing was done parallel.

The prototype was tested on Solaris and Linux and Windows NT. For Windows NT the compiler Cygwin was used. During compilation on Windows NT, the problem arose that is some cases the Win socket did not support NT Socket. This problem is solved by the configure script: configure checks if the NT Socket is supported, if not the required object (Socketlen) is searched and coupled.

Portability was tested using a Sparc processor (big endian) and an Intel processor (little endian). The prototype was using a library for bit manipulation, which seemed to be not endian proof (see section 5.2). For this reason, the bit manipulation library had to be made 'endian proof.' The endian proof bit manipulation routines can be found in Appendix D.

Since the network modules are being developed separate from Flight Gear (see chapter 10), compiling could be done fast and testing was relatively easy.

Integration in Flight Gear

CHAPTER 11: CONCLUSIONS

11.1 Multiplayer possibilities in Flight Gear

The multiplayer possibilities of Flight Gear are studied and the conclusion is made that although Flight Gear has a modular structure it is still complex and to add multiplayer functionality a lot of problems has to be solved. When the Flight Gear developers designed the flight simulator for more than one user from the beginning, it had been much more easy to visualize 'other' aircrafts. Also the provided network module was not really appropriate (see chapter 10). Flight Gear can run on different platforms but compiling with Cygwin under Windows can lead to unexpected problems especially when new code is added. The position data used in Flight Gear is using three different systems (see chapter 3). For a lot of properties, more than one unit is used like: radians/degrees, feet/meter/nautical mile. For this, Flight Gear does provide conversion functions and constants via Sim Gear (see chapter 2) but conversions decrease the performance.

11.2 Design of a multiplayer engine

A design for a multiplayer engine is made (see chapter 7, 8 and 9). For this design a lot of things a lot of researched has been done. The design is reviewed several times and the final design is rarely complex but does contain all essential functionality.

11.3 Development and implementation

During the development lots of porblems concerned with sockets and portability (see chapter 10) are solved. The network modules reached their goal and can be used for data transmission on three different platforms.

For the implementation of the MPE in Flight Gear, a complex makefile is developed that handles portability.

11.4 Testing

The network modules were tested both as separate modules and as integrated modules in Flight Gear. It was very difficult to test and debug the visualization of other aircrafts as this required to recompile Flight Gear after every modification.

APPENDIX A: REFERENCES

[BAS98] M. Basler, The Flight Simulator for free, http://Flight Gear.org/Papers/Basler-1998/FDFD3-98.html, 1998.

[FGD98] Flight Gear Developers, *proposal-3.0.1*, http://www.Flight Gear.org/proposal-3.0.1

[SOU00] SourceForge, *State Threads for Internet Applications*, 2000 http://state-threads.sourceforge.net/docs/st.html

[CAR91] Carlo Ghezzi, Mehdi Jazayeri & Dino Mandrioli, *Fundamentals of Software Engineering*, PRETENCE HALL, Upper Saddle River, 1991.

[EEO96] Dr. William T. Verts, *An Essay on Endian Order*, April 1996 http://www.cs.umass.edu/~verts/cs32/endian.html

[RFC1014]
http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1014.html

[RES01] Research Systems, *Multi-Threading in IDL 5.5*, 2001 http://www.researchsystems.com/idl/mthread55.pdf

[SOU00] Sourceforge, *State Threads for Internet Applications*, 2000 http://state-threads.sourceforge.net/docs/st.html

[CLO99] Curtis L. Olson,
Flight Gear Internal Scenery Coordinate Systems and Representations, 1999 http://www.Flight Gear.org/Docs/Scenery/CoordinateSystem/CoordinateSystem.html

[WON02] R. Wonnacott, *World Geodetic System 1984 (WGS 84)*, 2002 http://w3sli.wcape.gov.za/Surveys/Mapping/wgs84.htm

[CLU98] M.J. Clugston, *The New Penguin Dictionary of Science*, 1998 http://www.xrefer.com/entry/639777

[SDS02] Bipin Sehgal, Robert W. Deters and Michael S. Selig,
Department of Aeronautical and Astonautical Engineering University of Illinois,
Icing Encounter Flight Simulator, 2002
http://amber.aae.uiuc.edu/~m-selig/apasim/pubs/AIAA_Paper_2002-0817.pdf

[JPW01] Jean Paul van Waveren, The Quake III Arena Bot, Delft University of Technology, June 2001, http://www.kbs.twi.tudelft.nl/Publications/MSc/2001-VanWaveren-MSc.html

[SGS93] Sylvia Goldsmith, *A practical guide to Real Time Systems Development*, Prentice Hall Europe, 1993, ISBN 0 13 718503 0

[TAN97] Andrew S. Tanenbaum, Albert S. Woordhull,
Operating Systems, Design and Implementation, Second Edition, Prentice-Hall, New Jersey 1997, ISBN 0 13 630195 9

[THO90] Trevor Thom & Robert Johnson, *The air pilot's manual vol. 3, Air Navigation,*
First edition, St. John's Hill, Shrewsbury, England, 1990, ISBN 1 85310 016 1

APPENDIX B: CONFIRMATION HANDLING

Note: DTE stands for Data Terminal Equipment.

Ideal situation

DTE A
The confirmation flag is set.
The confirmation handler detects that the confirmation flag is set and adds a request id to the packet.
A copy of the packet is stored an the packet will be sent to DTE B.

DTE B
The packet from DTE A is received and the confirmation flag is read.
If the packet is not yet confirmed, it will be stored in a list of confirmed packets.
A packet will be sent to DTE A with the first confirmation slot filled with the request id.

DTE A
The packet from DTE B is received and the confirmation handler checks the confirmation slots for valid request id's. Valid request id's will be compared with the id's of the stored packets. Packets on the list with the same request id are confirmed and therefore removed from the list.

First exception situation

Exception: DTE B does not receive a packet sent by DTE A

DTE A
As before:
The confirmation flag is set.
The confirmation handler detects that the confirmation flag is set and adds a request id to the packet.
A copy of the packet is stored and the packet will be sent to DTE B.

DTE B
The packet from DTE A is not received.

DTE A
The packet is not confirmed. The packet is send to DTE B again. After 4 retransmissions the packet is too old and will be considered as definitely lost.

Second exception situation:

Exception: A confirmation is sent by DTE B but not received by DTE A.

DTE A
As before:
The confirmation flag is set.
The confirmation handler detects that the confirmation flag is set and adds a request id to the packet.
A copy of the packet is stored an the packet will be send to DTE B.

DTE B
The packet from DTE A is received and the confirmation flag is read.
If the packet is not yet confirmed, it will be stored in a list of confirmed packets.
A packet is sent to DTE A with one of its confirmation slots filled with the request id.

DTE A
The packet from DTE B is not received. DTE A sends the next packet to DTE B.

DTE B
The next packet from DTE A is received and the confirmation flag is read.
The request ID will be stored.
A packet is send to DTE A with the first confirmation slot filled with the request id and the second confirmation slot filled with the request id of the previous packet.
Because a packet only has 4 confirmation slots, a packet can be reconfirmed for 3 times.

Third exception situation

Exception: the confirmation is send by DTE but delayed.

DTE A
The confirmation flag is set.
The confirmation handler detects that the confirmation flag is set and adds a request id to the packet.
A copy of the packet is stored an the packet will be send to DTE B.

DTE B
The packet from DTE A is received and the confirmation flag is read.
If the packet is not yet confirmed, it will be stored in a list of confirmed packets.
A packet will be sent to DTE A with one of its confirmation slots filled with the request id.

DTE A
The packet from DTE B is not yet received.

DTE B
The packet from DTE A is received and the confirmation flag is read.
The packet is already confirmed so it will be thrown away.

Appendix C: Questions to the Flight Gear mailing list

This appendix contains a question and the answers to it using the Flight Gear mailing list. It is nice to see that the one question leads to questions by others as with question 2.

note: the authors of this thesis are identified by 'ace project.'

Question 1

```
Is anybody still working on de ./net part of PLIB ?
```

1st Reply

```
yes.  i'm still tracking net library development.  And regarding the
netBuffer data, here's a snippet from the include file...

class netBuffer
{
public:

   int getLength() const { return length ; }
  int getMaxLength() const { return max_length ; }

  /*
  **  getData() returns a pointer to the data
  **  Note: a zero (0) byte is appended for convenience
  **  but the data may have internal zero (0) bytes already
  */
  char* getData() { data [length] = 0 ; return data ; }
  const char* getData() const { ((char*)data) [length] = 0 ; return
data ; }

 };

Many internet protocols are text-based.  For games, binary protocols
may be used and in that case you'll want to use getLength() and getData()
together.  A memcpy could bite into performance in some applications,
but I think you could implement that on top of netBuffer's API:

void getData(char *data,int len)
{
  int n = getLength();
  if (n > len)
    n = len;  // what happens to lost bytes?
  memcpy(data,getData(),n);
}

So if there is a bug, I'm not sure I understand correctly.
```

2nd reply

```
There are some predictable bugs in the netBuffer
class. The most important one is:

const char* getData()

Its returns a string till the next '\0'-pointer. Which
is odd/not good for a buffer that encapsulated binary
data (such as arrays and integers) when used as
superclass for netMessage.

I've fixed it for my purpose to make it:
void getData(char *data,int len)

and then memcopy the data (after some checks).
My function isn't tested yet, but I think this bug
must be mentioned.
```

Question 2

On 12/09/02 at 02:18 ace project wrote:

>Hello,
>
>For the development of our Flight Gear Multiplayer
>Engine, we need to know if there are allready classes
>in Flight Gear that provide the possibility to display
>another aeroplane than ones own simultaneously. We
>found a module called 'FGViewer' that will probrabely
>draw only ones own aeroplane (or doesn't it?). However
>we would be very happy if someone can answer our
>question and/or tell more about the displaying of
>aeroplanes in Flight Gear.

1st reply

Hi Jeroen,

You can quite easily add another plane simultaneously, thanks to some
3D modelling support recently added by David Megginson. The attached
files demonstrate one way to do it - if you look at AIEntity.hxx this defines a class containing the entity's position, orientation, and an instance of FGModelPlacement. Calling the Transform method of an AIEntity derived class simply causes it to update its FGModelPlacement with its current position and orientation, and then run the update method of FGModelPlacement. FGModelPlacement (David M's class) then handles all the hard work of transforming the model correctly into Flight Gear coordinates wherever on the globe it is situated (thus replacing all the code after #if 0 in AIEntity.cxx). You need to look in AILocalTraffic.cxx (derived from AIEntity) to see where the FGModelPlacement is initialised with the correct model:

```
(from AIEntity.hxx):
protected:
   char* model_path;          //Path to the 3D model
   FGModelPlacement aip;

(from AILocalTraffic.cxx):
void FGAILocalTraffic::Init() {
   // Hack alert - Hardwired path!!
   string planepath = "Aircraft/c172/Models/c172-dpm.ac";
   SGPath path = globals->get_fg_root();
   path.append(planepath);
   aip.init(planepath.c_str());
   aip.setVisible(true);
```

That initialised the model - now to place it or update its placement:

```
(from AIEntity.cxx):
// Run the internal calculations
//void FGAIEntity::Update() {
void FGAIEntity::Transform() {
   aip.setPosition(pos.lon(), pos.lat(), pos.elev() *
SG_METER_TO_FEET);
   aip.setOrientation(roll, pitch, hdg);
   aip.update();
}
```

The model should now appear at the correct position in Flight Gear if
You look at it.

This is how I used David's 3D model support, I'm not sure if this is
The 'official' or best way to do it, I'm sure David will point out any
Gross misuse of his API once Canadians awake. This isn't in Flight Gear at the moment (the current AIEntity/AILocalTraffic has a nasty bug and puts multiple planes on top of each other until Flight Gear crashes which is why its not called) but if you unzip all the attached files into src/atc, and uncomment the following lines in main.cxx and fg_init.cxx

```
fg_init.cxx(962):    // globals->set_AI_mgr(new FGAIMgr);
fg_init.cxx(963):    // globals->get_AI_mgr()->init();
main.cxx(1044):    // globals->get_AI_mgr()->update(delta_time_sec);
```

Change to:

```
fg_init.cxx(962):    globals->set_AI_mgr(new FGAIMgr);
fg_init.cxx(963):    globals->get_AI_mgr()->init();
main.cxx(1044):    globals->get_AI_mgr()->update(delta_time_sec);
```

(the lines numbers might be slightly out)

And then start Flight Gear with

```
FGFS.EXE --airport-id=KEMT --heading=030
--prop:"/radios/comm[0]/frequencies/selected-mhz"=121.2
```

then you should be able to follow another plane round a traffic
pattern.

(You need w120n30 scenery though)
HTH

Cheers - Dave

2nd reply

Hi Jeroen,

My previous reply has been siezed by the moderator for being too large
So I've resent without attached files - the attachment mentioned in the
Mail can be found at:
http://www.nottingham.ac.uk/~eazdluf/AICircuits.zip
and this mail will probably reappear on the list later.

You can quite easily add another plane simultaneously, thanks to some
3D modelling support recently added by David Megginson. The attached
files demonstrate one way to do it - if you look at AIEntity.hxx this defines a class containing the entity's position, orientation, and an instance of FGModelPlacement. Calling the Transform method of an AIEntity derived class simply causes it to update its FGModelPlacement with its current position and orientation, and then run the update method of FGModelPlacement. FGModelPlacement (David M's class) then handles all the hard work of transforming the model correctly into Flight Gear coordinates wherever on the globe it is situated (thus replacing all the code after
#if0 in AIEntity.cxx).
You need to look in AILocalTraffic.cxx (derived from AIEntity) to see where the FGModelPlacement is initialised with the correct model:

```
(from AIEntity.hxx):
protected:
   char* model_path;          //Path to the 3D model
   FGModelPlacement aip;

(from AILocalTraffic.cxx):
void FGAILocalTraffic::Init() {
   // Hack alert - Hardwired path!!
   string planepath = "Aircraft/c172/Models/c172-dpm.ac";
   SGPath path = globals->get_fg_root();
   path.append(planepath);
   aip.init(planepath.c_str());
   aip.setVisible(true);
```

That initialised the model - now to place it or update its placement:

```
(from AIEntity.cxx):
// Run the internal calculations
//void FGAIEntity::Update() {
void FGAIEntity::Transform() {
   aip.setPosition(pos.lon(), pos.lat(), pos.elev() *
SG_METER_TO_FEET);
   aip.setOrientation(roll, pitch, hdg);
   aip.update();
}
```

The model should now appear at the correct position in Flight Gear if
you
look at it.

This is how I used David's 3D model support, I'm not sure if this is
the
'official' or best way to do it, I'm sure David will point out any
gross
misuse of his API once Canadians awake. This isn't in Flight Gear at
the
moment (the current AIEntity/AILocalTraffic has a nasty bug and puts
multiple planes on top of each other until Flight Gear crashes which is
why
its not called) but if you unzip all the attached files into src/atc
(its
safe to overwrite what's already there), and uncomment the following
lines
in main.cxx and fg_init.cxx

```
fg_init.cxx(962):    // globals->set_AI_mgr(new FGAIMgr);
fg_init.cxx(963):    // globals->get_AI_mgr()->init();
main.cxx(1044):    // globals->get_AI_mgr()->update(delta_time_sec);
```

Change to:

```
fg_init.cxx(962):    globals->set_AI_mgr(new FGAIMgr);
fg_init.cxx(963):    globals->get_AI_mgr()->init();
main.cxx(1044):    globals->get_AI_mgr()->update(delta_time_sec);
```

(the lines numbers might be slightly out)

And then start Flight Gear with

```
FGFS.EXE --airport-id=KEMT --heading=030
--prop:"/radios/comm[0]/frequencies/selected-mhz"=121.2
```

then you should be able to follow another plane round a traffic
pattern.

(You need w120n30 scenery though)

HTH

I've also got a question though - last time I looked through the code I
couldn't find a way to get the current terrain elevation for an
arbitrary
location - it appeared the only funtion available would return the
terrain
elev at the current (user) location. Am I missing something - is there

a
way to get scenery elev for an arbitrary location?

Cheers - Dave

3rd reply

Look in the Models directory for the code that draws the model(s).
FGViewer
doesn't actually draw anything, but it references the FDM Model (the
one
that's being flown) for positional data in order to set the camera
location in
certain views (e.g. pilot view).

You should be able to find it very easy to output other models. Also
take a look at the FGLocation class and the viewer configuration doc on the web page.
In a nutshell all you need to do is maintain a property structure for
Each aircraft that contains the position (lon,lat,alt) and orientation data (pitch,roll,heading). Take a look at Main/location.cxx and the viewer doc to see how these properties are used for drawing models. Then instantiate a model for each of those property structure entities.

E.G.

```
<multi-player>
  <plane>
    <position>
      ...lon, lat, alt
    </position>
    <orientation>
      ...pitch, roll, heading
    </orientation>
  </plane>
  <plane>
    <position>
      ...lon, lat, alt
    </position>
    <orientation>
      ...pitch, roll, heading
    </orientation>
  </plane>
  <plane>
    <position>
      ...lon, lat, alt
    </position>
    <orientation>
      ...pitch, roll, heading
    </orientation>
  </plane>
  ...etc
</multi-player>
```

Best,

Jim
On 12/09/02 at 11:49 DCL wrote:
>You can quite easily add another plane simultaneously, thanks to some
3D
<snip>

```
>(from AIEntity.hxx):
>protected:
>    char* model_path;          //Path to the 3D model
>    FGModelPlacement aip;
>
>(from AILocalTraffic.cxx):
>void FGAILocalTraffic::Init() {
>    // Hack alert - Hardwired path!!
>    string planepath = "Aircraft/c172/Models/c172-dpm.ac";
>    SGPath path = globals->get_fg_root();
>    path.append(planepath);
>    aip.init(planepath.c_str());
>    aip.setVisible(true);
```

Having just read whats come through, can everyone just ignore the fact
That I declare a variable to hold the model path in the base class and then declare another one in the derived class instead of using it - I'll just crawl under a rock for a while! It is just a work-in-progress that I pulled off my hard drive and zipped up.

Other issues are of course polygon count with large numbers of models -
would it be advantageous to have seperate versions of the models
without the interior (ie do those polygons count when they're not drawn), and can we dynamically switch to lower polygon models as the distance from the viewer increases.

Cheers - Dave

Questions of others and the answers to them

DCL writes:

> Other issues are of course polygon count with large numbers of
models -
> would it be advantageous to have seperate versions of the models
without
> the interior (ie do those polygons count when they're not drawn),
and can
> we dynamically switch to lower polygon models as the distance from
the
> viewer increases.

The current code allows switching to a lower-poly version at any arbitrary distance, but it also allows hiding parts of a model at a distance -- for example, the panel can disappear at 10m, the seats can disappear at 20m, the propeller can disappear at 100m, and so.

The main problem is that we will need alternative paint jobs for each model so that every J3Cub or 172 doesn't have the same colours and call sign.

All the best,

David

Just wanted to clarify that when I suggest looking at the Viewer config
docs,
it is so that you can see a method for managing placement through
properties.
The viewer code is concerned with placing "cameras", but it uses the
same
class (FGLocation) as the model code to get and manage location data.

You will probably want to look at viewmgr.cxx as well, since you will
probably
want to add views to get a close look at other players from time to
time.
Note that the current viewmgr.cxx doesn't allow adding and deleting
views
(cameras) on the fly, so that would need to be changed.

David Luff's suggestions on adding and updating models should help. It
would
be best to maintain model position in the property tree so that AI
plane
models and Multi-player plane models could be managed by the same code

without
any direct dependencies. In otherwords the code that displays the
model will
read position and orientation data from a place in the property tree.
As the
aircraft moves, the AI routine, or the Multi=player routine would
update that
location in the property tree.   This technique ensures a good deal of
versitility for all sorts of applications.

Best,

Jim

Jim Wilson wrote:
>David Luff's suggestions on adding and updating models should help.
It
>would
>be best to maintain model position in the property tree so that AI
plane
>models and Multi-player plane models could be managed by the same code
>without
>any direct dependencies. In otherwords the code that displays the
model
>will
>read position and orientation data from a place in the property tree.
As
>the
>aircraft moves, the AI routine, or the Multi=player routine would
update
>that
>location in the property tree.   This technique ensures a good deal of
>versitility for all sorts of applications.

Ahhh - it looks like I've been doing it all wrong. When the new stuff
appeared in the model directory my first instinct was to look through
the
header files to see which interface was the most suitable, and the only
one
I could make head nor tail of was FGModelPlacement so that's what I
used.
It appears though from your post and reading the modelmgr
implementation
that all I need to do is put a suitable node in the /model property
tree
for each plane, update the position etc property whenever the plane
moves,
and modelmgr will handle the rest. Basically I should maintain a
pointer
to a property node in each entity instead of a pointer to a

FGModelPlacement class. Fantastic!!

David Megginson wrote:
>The current code allows switching to a lower-poly version at any
>arbitrary distance, but it also allows hiding parts of a model at a
>distance -- for example, the panel can disappear at 10m, the seats can
>disappear at 20m, the propeller can disappear at 100m, and so.

Wow, I'm behind the times. How do I use this? Is it agreed between
the
actual 3D model itself and the modelmgr/model code such that I do not
need
to worry about it, or does it require me to specify which models to
switch
to at which distance, and which bits of the model to start
disappearing?

>The main problem is that we will need alternative paint jobs for each
>model so that every J3Cub or 172 doesn't have the same colours and
>call sign.

Judging from the MSFS world, repaints seem to be a very popular way for
people to get started in modelling. I suspect there'll soon be plenty
of
paint jobs for the available planes.

Cheers - Dave

DCL writes:

> Wow, I'm behind the times. How do I use this? Is it agreed between
the
> actual 3D model itself and the modelmgr/model code such that I do not
need
> to worry about it, or does it require me to specify which models to
switch
> to at which distance, and which bits of the model to start
> disappearing?

Here's an example from Aircraft/c172/Models/c172-dpm.xml that makes
the cabin interior and seats disappear at >=50m viewing distance (I
should include more here, like the yokes and rudder pedals, not to
mention the gauges):

```
 <animation>
  <type>range</type>
  <object-name>Cabin</object-name>
  <object-name>Seat.1</object-name>
  <object-name>Seat.2</object-name>
  <object-name>Seat.3</object-name>
  <object-name>Seat.4</object-name>
  <min-m>0</min-m>
  <max-m>50</max-m>
 </animation>
```

Since *.ac models are plain text, it is a simple matter to open them in an editor to find the object names.

All the best,

David

* DCL -- Thursday 12 September 2002 16:41:
> Judging from the MSFS world, repaints seem to be a very popular way
for
> people to get started in modelling. I suspect there'll soon be
plenty of
> paint jobs for the available planes.

Plenty of them can become a problem, given our traditional, but
very inefficient texture format:

```
                 .rgb              .png
  j3cub-01     178717 Bytes     96867 Bytes  (54%)
  j3cub-02     259222 Bytes    102459 Bytes  (39%) !!
```

m.   :-]

Melchior FRANZ writes:

```
 > Plenty of them can become a problem, given our traditional, but
 > very inefficient texture format:
 >
 >                 .rgb              .png
 >   j3cub-01     178717 Bytes     96867 Bytes  (54%)
 >   j3cub-02     259222 Bytes    102459 Bytes  (39%) !!
```

Note that that's only a disk-storage problem, not a texture-memory problem; textures are always fully uncompressed when they're being used by OpenGL.

All the best,

David

I know. A disk space and download band width problem.
m.

I checked it out: The difference between rgb(a) and png in the
current base package is 24.5 MB, less than I had thought.
(Whereby the few rgba's were obviously wrongly converted
without alpha layer and hence too small.)

```
  rgb(a)   45.8 MB
  png      22.4 MB
```

m.   :-)

On Thursday 12 September 2002 2:36 pm, Melchior FRANZ wrote:
> * Melchior FRANZ -- Thursday 12 September 2002 19:55:
> > I know. A disk space and download band width problem.
>
> I checked it out: The difference between rgb(a) and png in the
> current base package is 24.5 MB, less than I had thought.
> (Whereby the few rgba's were obviously wrongly converted
> without alpha layer and hence too small.)
>
>   rgb(a)   45.8 MB
>   png   22.4 MB
>
> m.   :-)
>

[rgb(a) vs. png/savings of about 23.2 MB]
* John Check -- Friday 13 September 2002 00:19:
> Thats still pretty substantial.
> Is png up to the job?

Yes. plib supports png and png is a lossless format that
supports alpha transparency. ImageMagick's "convert" seems
to do a good job converting rgb(a) to png. But ...

> Would we get any savings on tarball size?

Not much. So it is obviously really just disk space.
(Even cvs is gzipped for most people.)

```
                     tar.gz     tar.bz2
  ----------------------------------------
  rgb(a)    45.8       26.0       20.7
  png       22.6       21.2       20.8
  ----------------------------------------
            23.2        4.9       0.031
```

APPENDIX D: ENDIAN PROOF BIT MANIPULATION ROUTINES

Introduction
This header contains the interface for my BIT manipulation routines.
These routines were written so that I can write integers (or other 4 byte values) to a bitfield. Another design demand was that it had to be portable between big and little Endian machines.

The why
I wrote these functions for use in a project to create a multiplayer server for Flight Gear. At that time I could not find a portable bitmanipulation function, only "bfix" by Dick Hogaboom, which interface I copied, but his routines did not properly handle different mixed Endians.

License
Your free to copy, distribute the code in whole or in part as long as you keep my credits intact ("(C)2002 Leon Otte E-mail: s0meb0dy_else@yahoo.com"). I do not guarantee that it works for you, even though it worked for me ! And ofcourse, I'm not responsible for any damages, loss of data, etc.

Donations :)
If you really want to thank me somehow for writing these routines, plz send me a copy of the application your using it for (if it runs on a normal PC/MAC), you can use my E-mail to ask for my post-address.

How to use it
It write all the binary numbers from left to right. So if you would set 'value' to 128 and 'len' to 8 it would write 0000.0001 and NOT 1000.000 as you might expect ! a small example:

```
 0) u8 buf[4]={0,0,255,0};
    Result: 0000.0000 0000.0000 1111.1111 0000.0000
 1) bfi(buf,7,2,2);
    Result: 0000.0001 0000.0000 1111.1111 0000.0000
 2) bfx(buf,7,2,value);
    Result: 2
 3) bfi(buf,17,8,0);
    Result: 0000.0001 0000.0000 0000.0000 0000.0000
 4) bfi(buf,7,2,1);
    Result: 0000.0010 0000.0000 0000.0000 0000.0000
 5) bfi(buf,1,32,MAX_UINT);
    Result: 1111.1111 1111.1111 1111.1111 1111.1111
 6) bfi(buf,1,32,0);
    Result: 0000.0000 0000.0000 0000.0000 0000.0000
```

Amount of bits thats a integer has.

```
#ifdef HAVE_CONFIG_H
#       include "config.h"
#endif
```

// Here you can decide whether you want to use my default for BITS_PER_BYTE and BITS_PER_INT or that you want to use those of limits.h. The one out of limits.h is more portable, but happen to be incorrect on my platform (Solaris 2.6).

```
#ifdef HAVE_LIMITS_H // This block has been commented out !
#       include <limits.h>
#       ifndef BITS_PER_BYTE
#               define BITS_PER_BYTE CHAR_BIT
#       endif
#       ifndef BITS_PER_INT
#               define BITS_PER_INT LONG_BIT
#       endif
#endif // End comment block

 Amount of bits that my Byte has (used if not previously set by limits.h
#ifndef BITS_PER_BYTE
#       define BITS_PER_BYTE 8
#endif
 Amount of bits that my Integer has (used if not previously set by limits.h
#ifndef BITS_PER_INT
#       define BITS_PER_INT 32
#endif


// Error codes

 All went well (or the error was not catchable.
#define BF_E_OK      0
 I found some kind of error in the parameters supplied.
#define BF_E_ERROR (-1)

// Version bla bla
#define BF_VERSION_MAJOR 1
#define BF_VERSION_MINOR 0
#define BF_VERSION_PATCH 0

// Make sure it compiles as C code (so that nobody overwrites the << operators)
#ifdef __cplusplus
extern "C" {
#endif // __cplusplus

 Redefines unsigned char so that everybody can agree that its 8bits wide.
typedef unsigned char u8;
```

Redefines unsigned integer so that everybody can agree that its 32bits wide. Otherwise they should change 'int' to 'long' if needed. They could also change BITS_PER_INT to be correct.

```
typedef unsigned int u32;
```

Insert 'len' bits in a byte-array at the given position. (int) bit f insert:

```
 dst    = Destination Buffer
 pos    = Position offset (in bits), starting from 1
```

len   = # of bits to be filled
value   = Value to be set
returns = BF_E_OK (0) for good, BF_E_ERROR (-1) for error

int bfi   (u8 dst, u32 pos, u8 len, u32 value);

Extract 'len' bits from a byte-array. This version returns a error-value.
(int) bit f extract (wrapped):

- src   = Source Buffer
- pos   = Position offset (in bits), starting from 1
- len   = # of bits to be filled
- value   = Value to be set//is retrieved
returns = BF_E_OK (0) for good, BF_E_ERROR (-1) for error

int bfx_wrapped(const u8 src, u32 pos, u8 len, u32 value);

Extract 'len' bits from a byte-array. (u32) bit f extract:
- src   = Source Buffer
- pos   = Position offset (in bits), starting from 1
- len   = # of bits to be filled
returns = value that is retrieved

PS. ERRORS ARE NOT HANDLED HERE, THAT IS THE RESPONSIBILITY OF THE PROGRAMMER !
Use (int) bit f extract wrapped if you want some basic error handling.

u32 bfx   (const u8 srv, u32 pos, u8 len);   // Return the retrieved value

#ifdef __cplusplus
}
#endif // __cplusplus

#endif // _BITMANIP_H_