TRABAJO FINAL DE CARRERA

TÍTULO DEL TFC : AutoNAV4D. A co-simulator for Unmanned Aircraft Systems

TITULACIÓN: Ingeniería Técnica Aeronáutica, especialidad Aeronavegación

AUTOR: Joshua Martínez Tristancho

DIRECTOR: Juan López Rubio

SUPERVISOR: Pablo Royo Chic

FECHA: 21 de julio de 2008

Título : AutoNAV4D. A co-simulator for Unmanned Aircraft Systems

Autor: Joshua Martínez Tristancho

Director: Juan López Rubio

Supervisor: Pablo Royo Chic

Fecha: 21 de julio de 2008

Resumen

Este documento muestra los requerimientos y componentes necesarios para integrar en un mismo escenario, sistemas de avión no tripulados (UAS) y sistemas simulados. A esto se le llama co-simulación. Un UAS es una serie de sistemas aviónicos tanto a bordo del avión como en la estación de tierra, que podría incluso despegar, volar una misión y aterrizar seguro sin la intervención humana.

La capa de abstracción de servicios para el UAS (USAL) son servicios gestionados y comunicados a través de un programa llamado MAREA (Middleware Architecture for Remote Embedded Applications) basado en el modelo de publicación/suscripción a fin de poder mandar datos, eventos y comandos relacionados con servicios del UAS. La plataforma integrada de simulación de Icarus (ISIS) es una colección de servicios reusables que comprenden un mínimo juego común de los elementos que más se suelen usar en misiones civiles para aviones no tripulados. Algunos de estos servicios forman el co-simulador. Este escenario integrado es útil para probar la plataforma antes de que vuele el avión. El equipo ICARUS ha desarrollado su plataforma orientada a servicios (SOA) dentro de la cual está el UAS real.

El co-simulador es una arquitectura abierta que podría tener un servicio visor (Visor), algunos vehículos virtuales (Virtual Vehicle) y sistemas virtuales (Virtual System) y el gestor de virtualidad (Virtuality Manager). Este co-simulador da la capacidad de prototipado rápido y simulación a la capa de servicios USAL. El prototipado rápido se consigue a través del uso de estándares y componentes que permiten un rápido diseño e implementación de nuevas funcionalidades. La simulación se usa para incrementar la seguridad y reducir el coste de diseño. Cuando se usa esta capa de servicios, una implementación virtual viene a tener efectos similares que una real. Este UAS es parte de una misión, la cual lleva una arquitectura orientada a servicios (SOA)

Mi trabajo en este TFC consistió en diseñar e implementar algunas soluciones que formaron un co-simulador para sistemas de aviones no tripulados. Este co-simulador ha sido llamado AutoNAV4D93. Hay muchas cosas por añadir al co-simulador las cuales serán realizadas en un futuro como resultado del trabajo de desarrollo del escenario integrado ISIS el cual será presentado en la conferencia de exhibición el 9 de enero de 2009.

Palabras clave: Co-simulador, Virtual Vehicle, Virtual System y AutoNAV4D

Title : AutoNAV4D. A co-simulator for Unmanned Aircraft Systems

Author: Joshua Martínez Tristancho

Director: Juan López Rubio

Supervisor: Pablo Royo Chic

Date: July 21, 2008

Overview

This TFC document shows the requirements and the components needed to integrate in the same scenario, real Unmanned Aircraft Systems (UAS) and simulated systems. This is called co-simulation. An UAS is a series of onboard avionics systems and an on ground platform, which might takeoff, fly a mission and land safe without a human intervention.

UAS Service Abstraction Layer (USAL) is a set of available services running on top of the UAV system architecture to give support to most types of civil UAS missions. These services are managed and communicated by a thin software layer called Architecture for Remote Embedded Applications (MAREA). This Middleware promotes a publish/subscribe model for sending and receiving data, events and commands among the services of the UAS. The Icarus Simulation Integrated Scenario (ISIS) is a collection of reusable services that comprises a minimum common set of elements that are needed in most UAV missions. Some of these services conform the co-simulator. This ISIS integrated scenario is useful to test the platform before the UAS flies. The ICARUS team has developed his Service Oriented Architecture (SOA) platform where is set the real UAS.

The co-simulator is an Open Architecture. It may have a Visor Service, some Virtual Vehicles and Virtual Services and a Manager of Virtuality. This co-simulator gives capabilities in fast prototyping and simulation to the UAS Service Abstraction Layer (USAL). Fast prototyping is reached by the use of standards and components. It allows a fast design and implementation of new functionalities. Simulation is used for increase safety and reduce design cost. When a service abstraction layer is used, a virtual implementation has similar effects like the real ones.

My job for this TFC was to design and to implement some solutions which conform a co-simulator for Unmanned Aircraft Systems. This co-simulator has been called AutoNAV4D93. There are many things to add to the co-simulator and it may be accomplished in a future as a result of ISIS developing work. The ISIS integrated scenarios will be presented in the AIAA'09 Meeting and exhibit in 9 of January 2009.

Keywords: Co-simulator, Virtual Vehicle, Virtual System y AutoNAV4D

A Sonia

Quiero agradecer a Dios, a mi iglesia y a mi familia el apoyo recibido durante estos años de trabajo en el proyecto PANS8168. Sin ellos hubiera sido imposible llegar hasta aquí.

Agradecer también el incondicional apoyo de los profesores de la EPSC:
Cristina Barrado Muxí
Dagoberto José Salazar Hernández
Daniel Crespo Artiaga
Enric Pastor Llorens
Enrique Cargía-Berro Montilla
Francisco Javier Mora Serrano
F. Xavier Estopà Mulet
Jordi Gutiérrez Cabello
José Luis Andrés Yebra
Juan López Rubio
M. Angélica Reyes Muñoz
Marcos Quílez Figuerola
Miguel Valero García
Oscar Casas Piedrafita
Pablo Royo Chic
Pilar Gil Pons
Ricard González Cinca
Santiago Torres Gil
Xavier Prats Menéndez
Yuri Koubychine

Y a los miembros del equipo PANS8168 que formaron la MOLECULE:
Ángel Gomáriz
Anabel González
Andrea Jaime
Carles Gonzalez
Carlos Pérez
Christian Schneeberger
Carlos Zamora
Daniel De Miguel
Ignacio Valero
Josep Bonet
Josep Montolio
Juan Martínez
María Teresa Marín
Miriam Sánchez
María Victoria Torres
Sergio Fraile
Sara González
Toni Bardia
Verónica Herrero
Xavier Borrell

Carles, mai t'oblidarem.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

An *Unmanned Aerial Vehicle (UAV)* is an airplane without a pilot on board but monitored on ground by an operator which is the responsible for the mission. This kind of vehicles are suitable for civil missions and *D-cube* (Dangerous-Dirty-Dull) operations. *UAVs* are able to navigate through non-commercial aviation airspace but future airworthiness may consider integration of both commercial aviation and Unmanned Aerial Vehicles when those vehicles will be safe enough. So, an UAS is a series of on board avionics systems and an on ground platform, which might to takeoff, to fly a mission and to land safe without an human intervention.

There are many options and situations to fly a flight plan depending on mission's requirements but not all options are valid and efficient. In fact, some of them raises risk situations and even the crash. This is why we need to simulate the entire flight in order to validate them or find the most efficient flight path. Other important action is to add a variation in the flight which require a consequence's analysis for the previous validated flight plan. In this sense, complete simulation is no feasible due to an expensive or not accurate results.

A co-simulator may be able to put together real systems and simulated systems. The co-simulator is distributed, so it increases the performance and allows system's scalability. There are many flight simulators but not all of them are distributed. Some of them are able to simulate an airspace with several airplanes navigating together but, not all of them are able to integrate real airplanes with virtual airplanes even integrate real systems and simulated systems.

My job for this TFC is to design and to implement some solutions which conform a co-simulator for Unmanned Aircraft Systems called AutoNAV4D93.

Auto The term *Auto* is because some level of automatism will be implemented, for example, we always will provide a default value for each case, no fault messages will be showed in case of uncritical wrong commands, etc.

NAV The term *NAV* is because simulations must be optimized for a navigation model. The level of detail will be adjusted to this premise. The physic model implementation may be a basic kinetic model in terms of coordinates, speed direction and time elapsed.

4D The term *4D* is for showing a capability of not only works in three-dimensions but predicts trajectories. This is the concept of: know where it will be and when it will arrive.

93 The term *93* is because the PANS8168 team develops at least 92 versions. We will mark this implementation as the 93 version in the *AutoNAV* simulator line work.

CHAPTER 1. DEFINITION, REQUIREMENTS AND PREVIOUS WORK

1.1.. Co-simulation definition

Co-simulation is a term sometimes used to denote that a simulation is able to validate more than one thing. For embedded systems, software-hardware co-simulation is used to validate both the software and the hardware components, and can be used to gain information about the system before a prototype is actually built. A distributed co-simulator is formed by a group of components working together, and its modular design depends on the performance required and the accuracy of the simulation model.

The present co-simulator is a group of components and services integrated in a testbed platform able to integrate in the same scenario, real Unmanned Aircraft Systems (UAS) and simulated systems. The co-simulator helps to validate software, hardware and flight plans before its definitive implementation. This co-simulator is used for testing some UAV's missions in order to minimize both the development effort and risk, as well as to provide a feasible migration of the software from a testbed platform to a real platform.

1.2.. Requirement list for the co-simulator

In this section main requirements for the co-simulator are showed. These requirements will be explained in more detail in future chapters. This requirement list are as follows:

Embedded support. The co-simulator's components has to be supported by embedded systems because is the technology commonly used in civil UAS applications. In this sense we have found that *.NET framework* and *JAVA* are the most easy development environment to be used.

Multiplatform. We have decided to use *.NET framework 2.0* because it is supported by a multiplatform implementation, several common libraries and an easy translation from older *AutoNAV* versions. We will use *C#* because is an object oriented programming language usable on .NET framework for this aim.

Distributed. Other important requirement is that the co-simulator must be distributed. It is a set of separated components easy to be reused with other implementations. It allows high performance and future scalability.

Open Graphical Libraries. Some specific libraries are required for the co-simulator like *OpenGL*. We have found that the freeware *Tao Framework*[10] is the most adequate for the co-simulator requirements because it gives support to *.NET* and *Mono* developers. The *.NET* framework runs in *Windows* operating systems and the *Mono* framework runs in *Linux* operating systems.

Implemented under a Middleware. The interconnection between co-simulator components and services are done by the *Middleware Architecture for Remote Embedded Applications (MAREA)* developed by the Icarus team through the communication primitives of variables, events, remote evocation and file transmission.

Integrated as a service. The co-simulator's components and services are designed to be integrated as a part of the *UAS Service Abstraction Layer (USAL)*. The *USAL* is a set of available services running on top of the UAV and designed by the Icarus Team.

1.3.. Previous work

In this section we want to present the previous work which make us possible to design and to implement the co-simulator for Unmanned Aircraft Systems. You will see that, in fact, this is a new *AutoNAV* version integrated in the ISIS Icarus testbed to be explained later[1]. This previous work is considered in order to understand how many people have been involved in many of the concepts used in this co-simulator.

1.3.1.. Icarus team presentation

The *Icarus research group* is composed by personnel such as researchers from the *Computer Architecture Department (DAC)* of the *Technical University of Catalonia*, and belongs to the *Aeronautic and Aerospace Research Group (CRAE)* of the *UPC*.

Figure 1.1: Intelligent Communications and Avionics for Robust Unmanned aerial Systems.

The research of the ICARUS group is focused on the topic of Unmanned Aerial Vehicles (UAV). This type of airplanes fly with no humans on board but monitoring on ground is mandatory. The target of the research are technologies that allow to build low cost UAVs and to manage then for several civil missions as autonomous as possible[6].

One of the research lines of Unmanned Aircraft System (UAS) is to develop a technology testbed able to test and design civil missions. This testbed platform is called *Icarus Simulation Integrated Scenario (ISIS)* where it is located the co-simulator *AutoNAV4D93*.

[1] See chapter 2.4. for definition

1.3.2.. AutoNAV. Four years ago

Personal interest of the author was focused on team work in software development. In this sense many people were involved around of the *AutoNAV4D* application from four years ago to the present day. This history starts in a first course subject called *Introducció als computadors (IC)* when one of the class group implements a navigation scheduler project. This project required a 'shell' design (console text based implementation). They do, but also they implement a Windows application in 3D environment with spanish airways, waypoints and a camera which follows an airplane. This work, called AutoNAV, was based on knowledge gain in others first course subjects like *Transport Aéri (TA)* and *Tecnologia Aeroespacial (TAe)*. The teacher *Xavier Prats* was interested in our work in order to be used with an Air Procedure Designer Project called *RAPIT* where *Mr. Josep Montolio*[9] and *Mr. Carlos González* from our team were involved. This was the first PANS8168 team example of technology transfer to the aeronautical industry. Now we will transfer some of this technology to the Icarus team.

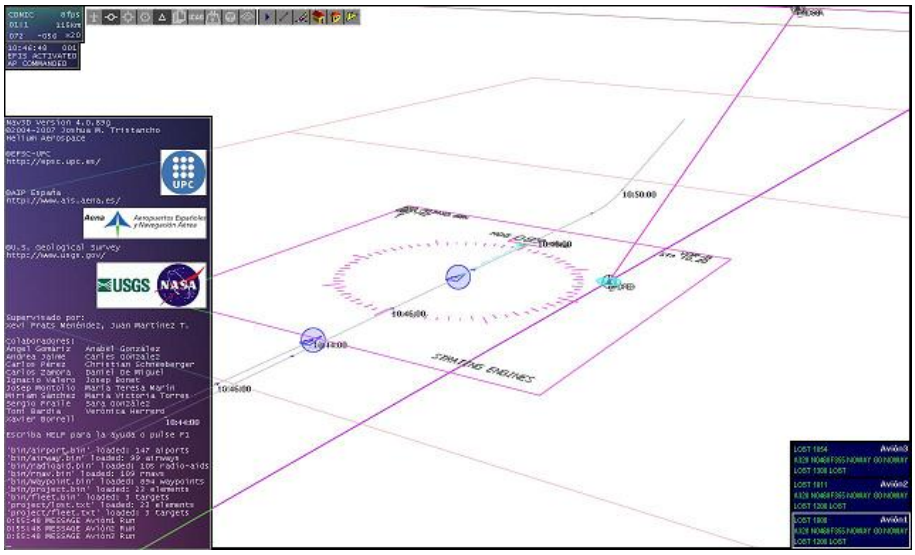


Figure 1.2: A screen shoot from the AutoNAV4D version 89. It was our first future prediction implementation.

During the summer July'05 the *Delegació d'Alumnes del Baix Llobregat (DABL)* in collaboration with the *Escola Politècnica Superior de Castelldefels (EPSC)* gave a Java Course oriented to catch interested personnel in order to build a developer team. Some alumni worked, as alumni's association *AeroUPC*[3], in several university projects like *CESNA cabin* project, *PANS8168* project, *Wind tunnel* project and the newly UAV's team called *ICARUS*. Those projects, for two years, was a testbed of young engineers. Some of them develop the application *AutoNAV3D* using *OpenGL* graphic libraries and a basic model of air traffic navigation simulator. This application was used in practice lessons of the second course in the subject *Gestió de l'Espaci Aeri (GEA)* teach by *Mr. Xavier Prats Menéndez* and *Mr. Lucas García Serrano*.

The team was structured like a molecule of *2-amino-triaminopropane*[7] where the carbon skeleton (colored yellow in the figure 1.3) is a propane and each radical is formed by an amino function of three components. Each hydrogen (colored in black) is a specialized

programmer, each nitrogen (colored in red) is a programmer but also responsible for a specific subject and each carbon (colored in yellow) is a programmer and also a manager of two or three subjects. This molecule may change in size in order to adapts his capacity to the project's phases. They implement the en-route environment with Spanish airways, waypoints, airports, radio-aids, etc. Also they implement the designer environment like CAD designer as learned in the first course subject *Expressió Gràfica (EG)* and the *3D Electronic Flight Instruments System (3DEFIS)* or the remote console as learned in the second course subject *Tècniques de Computació i Programació (TCP)*.

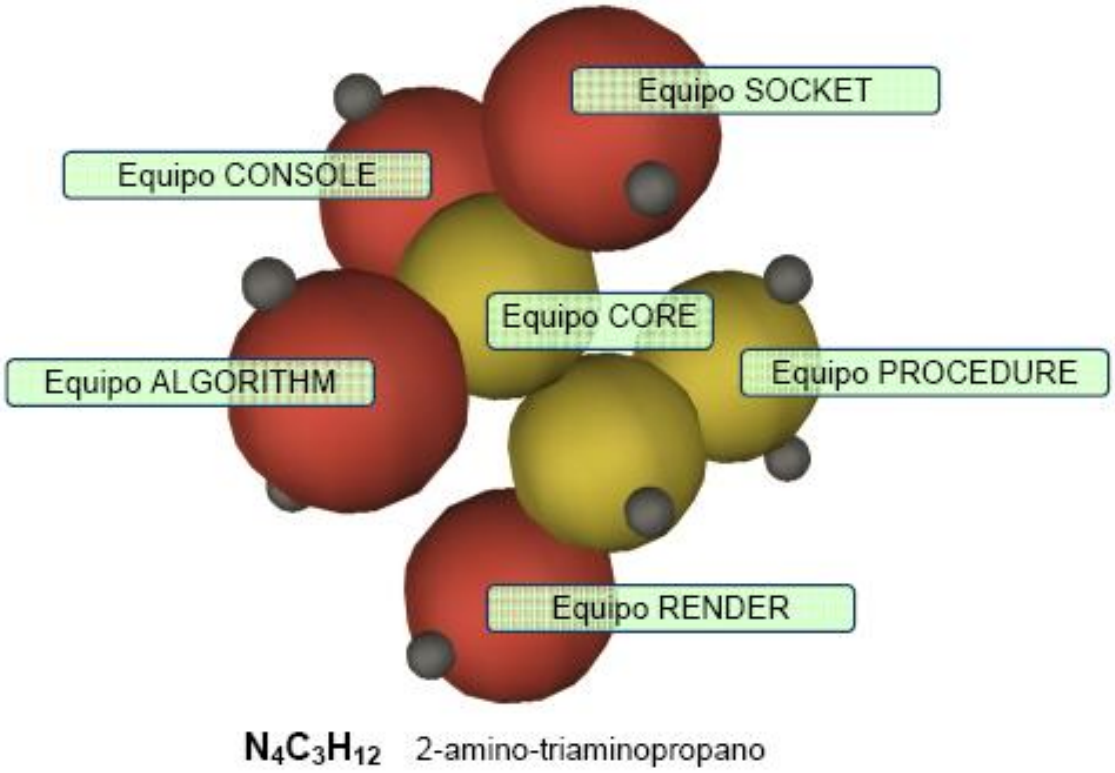


Figure 1.3: 2-amino-triaminopropane molecule was the distribution of PANS8168 project's team in 2006.

During the summer July'07, some mechanical works are done in the *Wind tunnel* and the UAV *Shadow* by the Icarus team. A rudder servo control surface and a cradle with three degrees of movement was build for the UAV *Shadow*, and a panoramic three window projections in the *CESNA cabin*. Several simulations are done in this fuselage related to air traffic management or procedure design. Also we have used two flight simulators; *MS FS2002* and the open source *Flight Gear* which will be use on 2008 by Icarus team.

1.3.3.. Additional uses for the AutoNAV simulator

Last year, the author has been worked on pico-satellites design, showed in the figure 1.4, as learned in the third course subject *Sistemes Espacials (SE)*, and the prediction of trajectories as learned in the third course subject *Tècniques Experimentals d'Aerofísica (TEA)* and the subject *Model Rocket Workshop (MWR)*, developing the *Interplanetary Navigation Display (IND)* for the new application *AutoNAV4D*.

Finally as a result of more involved effort with the *Computer Architecture Department (DAC)*, the author has been worked in collaboration with *Mr. Pablo Royo Chic* and *Mr. Juan López Rubio* in the *Icarus Simulation Integrated Scenario (ISIS)* testbed platform as covered by this *Treball de Fi de Carrera (TFC)* document. We will present some of this work in the *AIAA'09 Meeting and exhibit*[17].

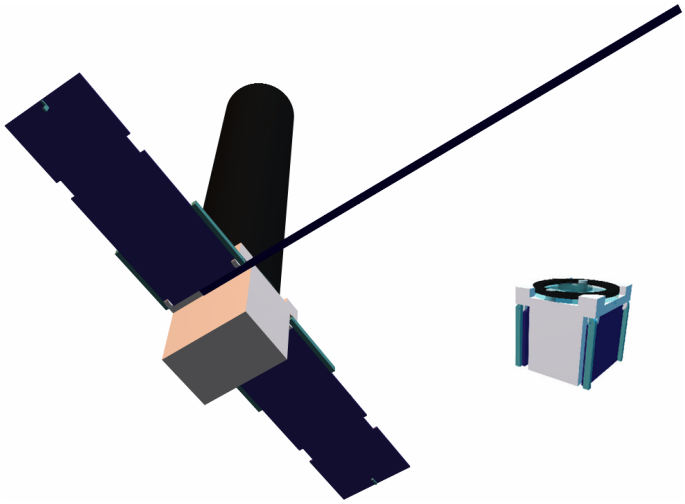


Figure 1.4: Pico-satellite developed by alumni from the Castelldefels School of Technology (EPSC).

CHAPTER 2. TECHNOLOGIES FOR THE CO-SIMULATOR

In this chapter we will introduce some important technologies used in this project in order to reach a comprehensive understanding for future chapters. These technologies are: The *Unmanned Aerial Vehicles (UAV)*, The *UAS Service Abstraction Layer*, the *Middleware Architecture for Remote Embedded Applications*, the *Icarus Simulation Integrated Scenario* testbed, the *.NET framework* and the *Open Graphic Libraries*.

2.1.. UAV. Unmanned Aerial Vehicles

An Unmanned Aerial Vehicle (UAV) is a non-piloted airplane designed to operate in D-cube (Dangerous-Dirty-Dull) situations; for example situations in which the utilization of a traditional airplane could be dangerous, or the environment too rough, or the operation too repetitive. Focusing on civil applications a wide range of application scenarios do exist: remote environmental research, pollution assessment and monitoring, fire-fighting management, security for example border monitoring, agricultural and fishery applications, oceanography, communication relays for wide-band applications, etc.

UAVs are automatically guided by an embedded system named Flight Control System or FCS. The goal of a FCS is to guarantee the stable flight of the UAV through a predefined flight-plan. Many FCS are commercially available today, however no commercial system exists nowadays that provides support to the actual mission and/or application that the UAV should perform, what we call the mission management.

Nowadays and after many years of development, UAVs technology is reaching the critical point in which it can be applied in a civil/commercial scenario. Many types of UAVs exist today; however the class of mini/micro UAVs is emerging as the valid option if this civil commercialization scenario is kept in mind. This type of UAV has the same limitations as most embedded systems: limited space, limited power resources, increasing computation requirements, complexity of the applications, time to market requirements, etc. All these stringent requirements are amplified in civil/commercial applications. In that context, the same platform should be able to implement a large variety of missions and operate with many types of payload; all of it with little reconfiguration effort and overhead if the system has to be economically viable. For this reason we believe that the effective application of UAVs in civil operations requires implementing new hardware/software systems that provide specific support to automatically control the actual missions to be carried out by the UAV[4].

Icarus team have some large *Unmanned Aerial Vehicle (UAV)*: The *Megastar* and the *Shadow*.

The Unmanned Aerial Vehicle *Megastar* showed in figure 2.1 is a giant radio-controlled trainer manufactured by Protech[13]. It has 94.49 inches (2.4 meters), length 71.65 inches (1.82 meters) and weight 13.01 lbs (5.9 kilograms). The RC has 4 channels and 4 servos (Throttle, Elevator, Rudder and Ailerons) and is a steerable tail gear. The power class is Glow engine 1.5 size (26 cubic centimeters). The transmitter is a 7 channels model *ECLIPSE7*. It works in the frequency range from 72 to 73 MHz. It has a receiver and a servo-amplifier connected to each servo with out a shielded lead.

Figure 2.1: Megastar XL120 a giant RC model converted in UAV by Icarus team.

The UAV *Shadow* is a push propeller airplane with two tails and two rudders as shows the figure 2.2. It have 17 feet (5.2 meters) of wingspan. The power-plant is a twin cylinder glow engine. The autopilot *AP04* manufactured by *UAV Navigations*, is a fully automatic, multi-waypoint, 3D flight-plan following; having throttle, elevator, rudder and ailerons control.

Figure 2.2: Shadow UAV is 17 feet (5.2 meters) of wingspan by Icarus team.

2.2.. USAL. Abstraction layer

The *UAS Service Abstraction Layer (USAL)*, developed by the Icarus team, is a set of available services running on top of the UAV system architecture to give support to most types of UAS missions and give facilities to end-users programs in order to access the UAS payload; reducing ”time to market” when creating a new UAS system and simplifying the development. The co-simulator's components and services are suitable to be integrated as a part of the *USAL*[17].

2.2.1.. SOA. Service Oriented Architecture

The USAL is *Service Oriented Architecture (SOA)*. Service Oriented Architectures is getting common in several domains. The main goal of *SOA* is to achieve loose coupling among interacting components, called services. A service is an unit of work, implemented and offered by a service provider, to achieve desired final results for a service consumer. The benefits of this architecture are the increment of interoperability, flexibility and extensibility of the designed system and of their individual services. In the implementation of a system, we want to be able to reuse existing services. *SOA* facilitates the services reuse, while trying to minimize their dependencies by using loosely coupled services. Following this service oriented vision, we propose the usage of services to represent the different components and functionalities that conform the complete UAS operation. These services are managed and communicated by a thin software layer[17].

2.2.2.. Services of the USAL

The USAL is composed by a large set of available services as showed in figure 2.3. Not all of these services have to be present in every UAS or in every mission. Only those services required for a given configuration or mission constraints, should be present and/or activated in the UAS. Available USAL services are classified in four categories:

Flight services: all services in charge of basic UAS flight operations: autopilot, basic monitoring, contingency management, etc.

Mission services: all services in charge of developing the actual UAS mission.

Payload services: specialized services interfacing with the input/output capabilities provided by the actual payload carried by the UAS.

Awareness services: all services in charge of the safe operation of the UAS with respect terrain avoidance and integration with shared airspace.

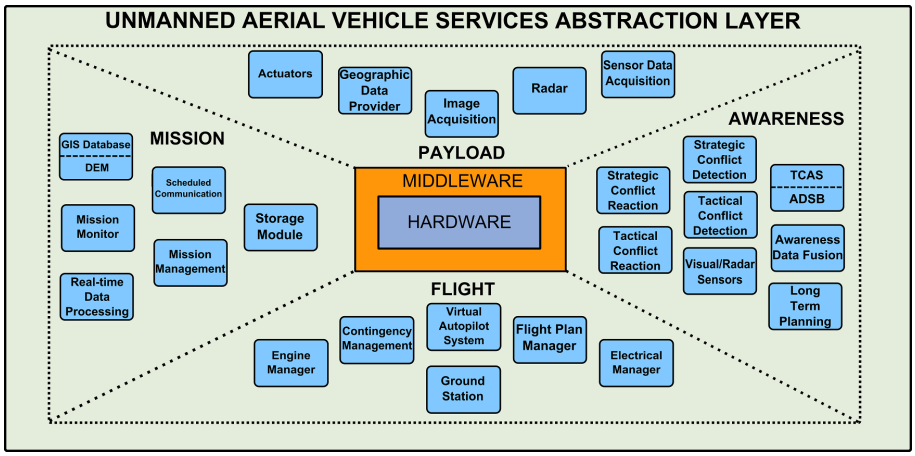


Figure 2.3: USAL architecture global view and categories by Pablo Royo.

2.3.. MAREA. Middleware

The data flow between USAL components and services are done by the *Middleware Architecture for Remote Embedded Applications (MAREA)*. This middleware was developed by the Icarus team and uses an architecture based on communication primitives showed in figure 2.4.

2.3.1.. Middleware description

The Middleware Architecture for Remote Embedded Applications (MAREA) is a middleware-based software systems which consist of a network of cooperating services. MAREA provides an execution environment with communication channels and common functionalities. The role of each service is expressed by the action of publish, subscribe, or both simultaneously; in that way, the publish-subscribe model eliminates complex network programming for distributed applications that makes easy to implement an embedded service. This middleware offers the localization of the other services and manages their discovery in the network; handles all the transfer chores, message addressing and retransmission, data delivery, flow control, etc. Also, this middleware handles marshaling and demarshalling (depending on different hardware platforms that are running a service) like bays where resources support, as a container, the cooperative services[4].

Service Management. The middleware is responsible for service life's cycle starting, communicating, monitoring, reporting changes and stopping each service.

Resource management. The middleware also centralizes the management of the shared resources of each computational node such as memory, processors, input/output devices, etc. A dispatcher may define those resources in a validated option of *Line Replacement Units (LRUs)* like engines, avionics bays, etc. The *Resources Management* optimizes the needs during the mission due to changes in the number of radio-links that are available or loses of UAV performances, range or mission time.

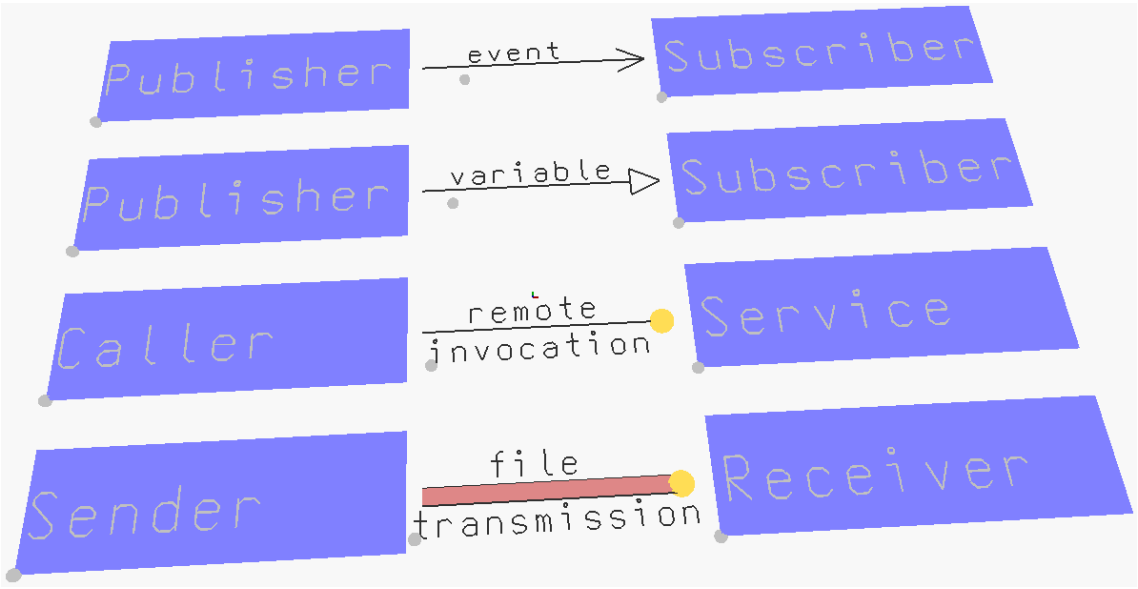


Figure 2.4: Data flow between services through MAREA communications primitives.

Name management. The services are addressed by name, and the middleware discovers the real location in the network of the named service, it makes very easy to reuse existing components by just adding new dedicated services to the new requirement.

Network Management. The middleware access the network instead the services, allowing the services to be deployed in different nodes depending on the UAS configuration.

2.3.2.. Communication Primitives

MAREA promotes a publish/subscribe model for sending and receiving data, events and commands among the services of the UAS. Services that are producing valuable data publish that information while other services may subscribe them. MAREA takes care of delivering the information to all subscribers that declare an interest in that topic. Next, we will describe the used communication primitives, which have been named as *Variables*, *Events*, *Remote Invocations* and *File Transmissions*.

Variables. Variables are the transmission of structured, and generally short, information from a service to one or more services of the distributed system. A Variable may be sent at regular intervals or each time a substantial change in its value occurs. The relative expiry of the Variable information allows to send it in a best-effort way. The system should be able to tolerate the lost of one or more of these data transmissions. The Variable communication primitive follows the publication subscription paradigm.

Events. Like Variables, Events also follow the publication-subscription paradigm. The main difference in front of Variables is that Events guarantee the reception of the sent information to all the subscribed services. The utility of Events is to inform of punctual and important facts to all the services that care about them. Some

examples can be error alarms or warnings, indication of arrival at specific points of the mission, etc.

Remote Invocation. The Remote Invocation is an intuitive way to model one-to-one of interactions between services. Some examples can be the activation and deactivation of actuators, or calling a service for some form of calculation. Thus, in addition to Variables and Events, the services can expose a set of functions that other services can invoke or call remotely.

File Transmission. The File Transmission primitive is used basically to transfer long file-structured information from a node to another when exists the need to transfer continuous media with safety. This continuous media includes generated photography images, configuration files or services program code to be uploaded to the middleware. For example this primitive is used in the thermal camera transmission to the ground station.

2.4.. ISIS. Integrated Scenario

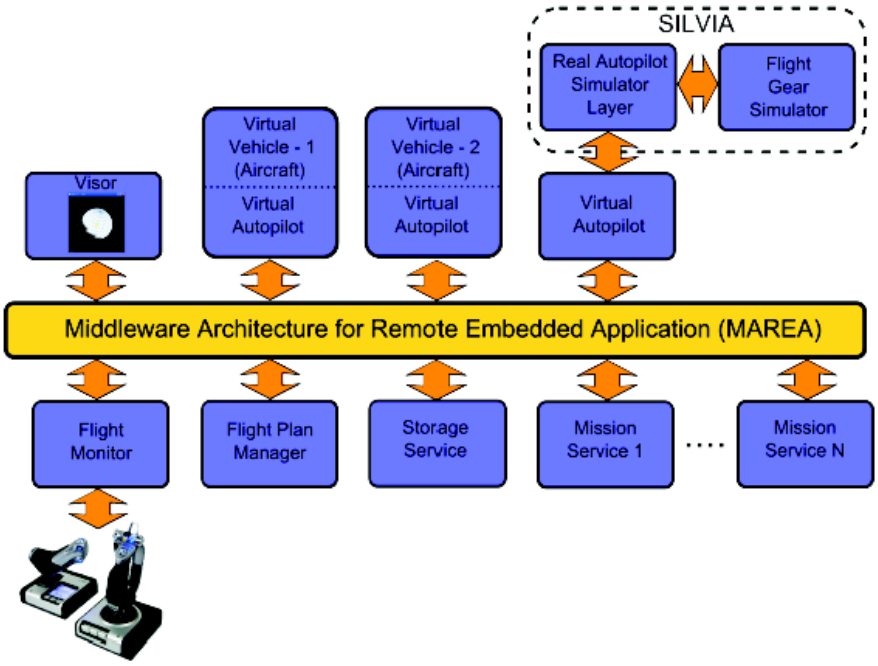


Figure 2.5: Icarus Simulation Integrated Scenario by Pablo Royo

The *Icarus Simulation Integrated Scenario (ISIS)* [17] is a collection of reusable services that comprises a minimum common set of elements that could be needed in most UAV missions. A number of specific services have been identified as ”a must” in any real life application of UAVs. The idea is to provide an abstraction layer that allows the mission developer to reuse these components and that provides guiding directives on how the services should interchange avionics information with each other. The available services cover an important part of the generic functionalities present in many missions. Therefore, to adapt our aircraft for a new mission it will be enough to reconfigure the services deployed in the UAV boards.

As showed in figure 2.5, the aim of ISIS is to minimize both the test development and validation cost, as well as to provide an easy migration of the software from the testbed platform to the real flight platform[17]. On ground, we have found the *Flight Monitor (FM)* which is intended to be used by the operator. The *Flight Plan Manager* manages the flight plan of each UAV for each mission. The *Storage Service* provide massive information like cartographic maps or stores the multimedia information produced for the camera’s mission. In order to control test the UAV’s autopilot, there is the *Software In Loop VAS Interface Adapter (SILVIA)* consist of a *Virtual Autopilot Service*, a *Autopilot Gateway* and a connection to the flight simulator *Flight Gear*. Finally, *ISIS* incorporates *Co-simulator* consisting of a *Visor Service*, some *Virtual Vehicles* and Virtual Services and a Manager of Virtuality used mainly in fast simulations or different epoch[1] to the present time. We have installed a version of the ISIS in the cabin simulator of *Escola Politècnica Superior de Castelldefels (EPSC)* called *CESNA*.

[1] Epoch means not only a local time but also a different time speed

2.5.. .NET technology

The co-simulator must be created as a set of separated components to be reused with other implementations. This concept is called distributed. The C# is the most desired object oriented programming language. Visual Studio 2005 (Windows) and Mono (Linux) support this language. The *Container* is where the service is running. Many embedded containers are available. There are containers where only *C* are supported, like a Programmable System on Chip (PSoC) or Field Programmable Gate Array (FPGA). Other containers commonly used in UAS applications supports the .NET Framework 2.0 in which we are interested in.

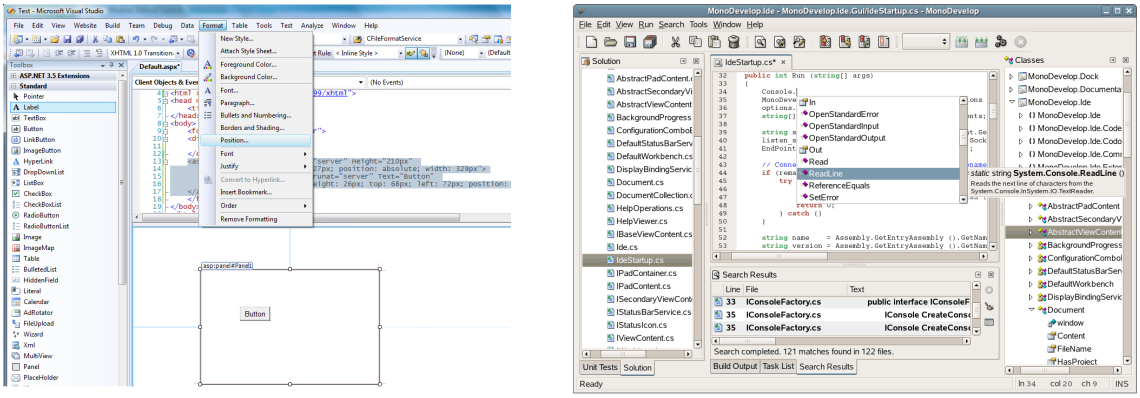


Figure 2.6: .NET framework for Windows and Mono for Linux.

2.5.1.. .NET definition

.NET Framework 2.0 is the Microsoft's managed code programming model for building applications that have visually stunning user experiences, similar and secure communication, and the ability to model a range of business processes. The following information has been extracted from the Microsoft Developer Network *MSDN library*[14]. Some design features are:

Interoperability or interaction between new and older applications require means in order to access external functionalities like the Invoke COM components.

Common Runtime Engine having an intermediate language (CIL) not interpreted but compiled just-in-time. This implementation is called Common Language Runtime (CLR).

Language Independence introduces a Common Type System of all possible datatypes that supports the exchange of instances of types between programs written in any of the .NET languages.

Base Class Library (BCL) is a library of functionality available to all languages using the .NET Framework like file reading, writing, XML documentation manipulation, etc.

Security provides a common security model for all applications.

Portability allows theoretically to run in any development platform, also cross-platform compatible, not only the *MS version Visual Studio 2005*[12] for Windows, also the Common Language Intermediate (CLI) specification is open and makes it possible for third parties entities to create compatible implementations like the *Mono*[11] for Linux.

2.6.. OpenGL. Graphic libraries

The co-simulator may show many visual information like level contours, flight path, virtual flight camera view, etc. For these reasons we may access to the potential of the video card through the use of *Open Graphic libraries (OpenGL)*. We have found that the freeware *Tao Framework*[10] is the most adequate for the co-simulator requirements because is supported by *Visual Studio* and *Mono* developers.

2.6.1.. OpenGL history

In 1992 the *OpenGL Architectural Review Board (OpenGL ARB)* was established. The *OpenGL ARB* is a group of companies that maintain and update the *OpenGL* standard and is widely used in scientific visualization, virtual reality (see *VRML*), *Computer Aid Design (CAD)*, and flight simulation. In 1994 *SGI* include elements such as a *scene-graph* but in 1995, *Microsoft* released the *Direct3D* which is the main competitor of *OpenGL*. *OpenGL 2.1* was released on 2006 and is backward compatible with all prior *OpenGL* versions, and incorporates the *OpenGL Shading Language (GLSL*, also known as *slang*, which have pixel buffer objects for efficient image transfers. *OpenGL 3.0*, formerly known as the codename *Longs Peak* will be revealed during the *OpenGL BoF* at *SIGGRAPH 2008*[16] and have an asynchronous object creation. About *C#* implementation of *OpenGL* libraries, many versions are done like the *CsGL OpenGL/C# library*[2] but its development has essentially stopped. The freeware *Tao Framework* is a most supported *C#* implementation of *OpenGL* nowadays. The *Tao Framework* for *.NET* is a collection of bindings to facilitate cross-platform media application development utilizing the *.NET* (Windows) and *Mono* (Linux) platforms[10].

Figure 2.7: Open Graphics Library. Source http://www.brandsoftheworld.com/

2.6.2.. OpenGL standard

OpenGL was developed by *Silicon Graphics Inc. (SGI)* and is a standard specification defining a cross-language cross-platform *Application Programming Interface (API)* for applications that need *2D* and *3D* computer graphics. A cross-platform is an interface between two implementations; in our case is an interface between a software implementation (our co-simulator) and a hardware implementation (video card). Older video cards do not support many functions. *OpenGL* overcame this problem by providing support in software for features unsupported by hardware. This function allows to any application to use advanced graphics on relatively low-powered systems. The interface consists of more than 200 different function calls which can be used to draw complex three-dimensional scenes from simple primitives.

OpenGL hides the complexities of interfacing with different 3D accelerators, by presenting the programmer with a single, uniform API. Also it hides the different capabilities of hardware platforms, by requiring that all implementations support the full *OpenGL* feature set (using software emulation if necessary).

CHAPTER 3. SCIENTIFIC BASES FOR THE CO-SIMULATOR

In this chapter we will introduce some important scientific bases used in the co-simulator. These scientific bases are: *Quaternions*, *Orbits*, *Keplerian elements*, *Reference systems*, *Datums* and *Geo-positioning*. Some of them are called *Addons* and are implemented as a class or structure in the co-simulator source as we see in the chapter 4.4..

3.1.. Quaternions

We have used quaternions for make rotations. Quaternions avoid the problem of "gimbal-lock". When we use Euler angles like pitch, roll, yaw there is no movement for the gimbal angle. Quaternions extend the concept of rotation in three dimensions to rotation in four dimensions. Instead of rotating an object through a series of successive rotations, quaternions allow the programmer to rotate an object through an arbitrary rotation axis and angle. This is an AutoNAV4D93 add-on called *Quaternion*.

The quaternions are defined as the ring[1]:

$$\mathbb{H} = \{Q = w + xi + yj + zk \quad \forall\, w,\, x,\, y,\, z \in \mathbb{R}\} \tag{3.1}$$

where the addition operation is defined by:

$$(w + xi + yj + zk) + (a + bi + cj + dk) = (w + a) + (x + b)i + (y + c)j + (z + d)k \tag{3.2}$$

and the multiplication operation is defined by:

$$(w + xi + yj + zk)(a + bi + cj + dk) \tag{3.3}$$

$$(wa - xb - yc - zd) + (wb + ax + yd - zc)i + (wc + ye + zb - xd)j + (xd + za + xc - yb)k \tag{3.4}$$

The defining relations are:

$$i^2 = j^2 = k^2 = ijk = -1 \quad \forall\, i,\, j,\, k \in \mathbb{I} \tag{3.5}$$

Quaternions are attributed to Sir William Rowan Hamilton on the 16th of October 1843.

[1] A *ring* is an algebraic structure with addition and multiplication

3.2.. Orbits

We use the *orbit* concept in order to follow a straight path over the planet. When you follow a constant heading, you describe a path called *loxodromic* path. The shorter path between too far points over a planet surface is the *orthodromic* path as showed in figure 3.1. Near the equator, the paths are closer. This is an AutoNAV4D93 add-on called *Orbit*.

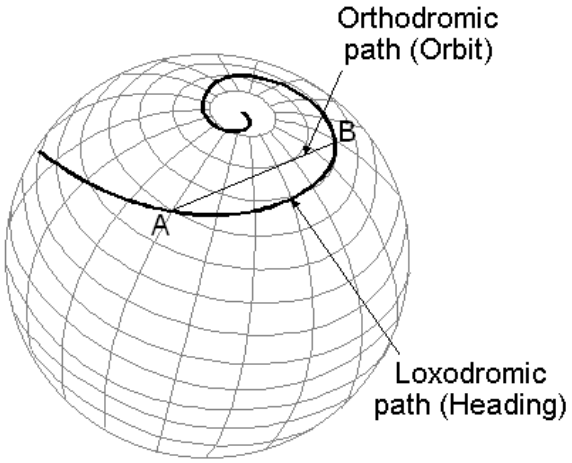


Figure 3.1: Reference http://www.navworld.com/navcerebrations/flights.htm Loxodromic path.

3.3.. Keplerian elements

We define the Keplerian elements as a set of six variables which establish the satellite trajectory. These elements are showed in figure 3.2. Keplerian elements are implemented in the AutoNAV4D93 add-on called *Orbit*. We assume that epoch is known because is the local time of the satellite.

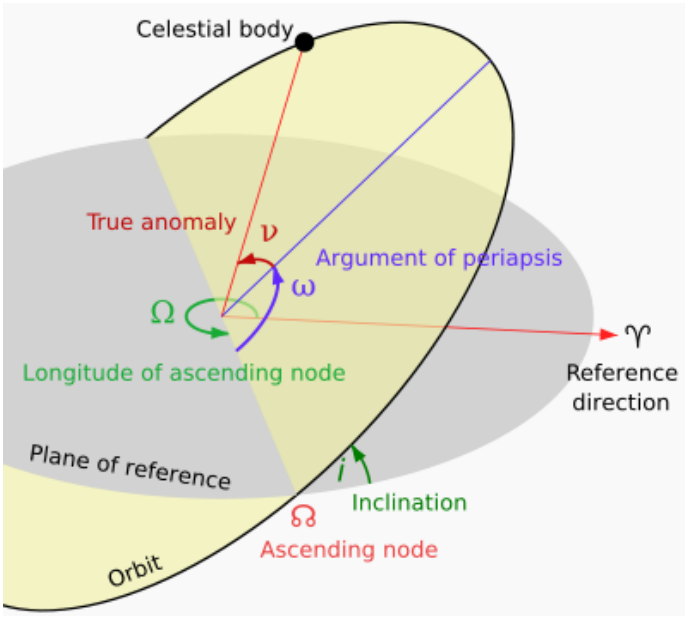


Figure 3.2: Reference http://en.wikipedia.org/wiki/Image:Orbit1.svg Keplerian parameters.

a0 Orbit's semimajor axis measured in meters (m).

e0 Orbit's eccentricity, where 0 is a circle, an ellipse is between 0 and 1, a parabola is equal to 1 and more than one is a hyperbola. This is a non dimensional factor.

i0 Orbit's inclination measured in radians (rad).

omega0 Orbit's ascending node measured in radians (rad).

v0 True anomaly at epoch measured in radians (rad).

W0 Orbit's argument of periapsis measured in radians (rad).

3.4.. Reference systems

In the AutoNAV4D93 co-simulator, we have used three different reference system for different purposes.

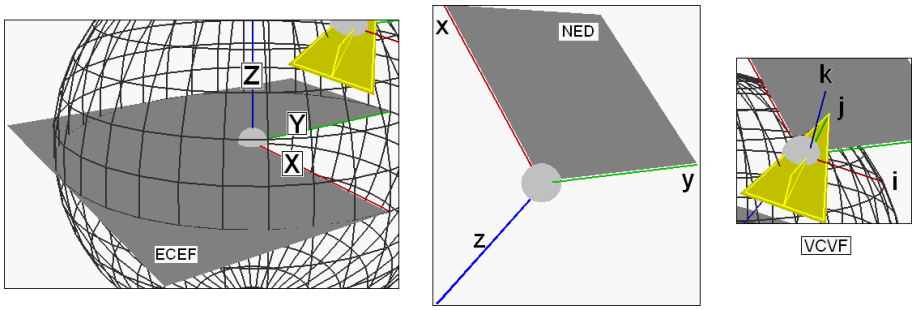


Figure 3.3: The three reference systems used in the AutoNAV4D93

Earth Centered Earth Fixed The ECEF axis (X, Y, Z) is the planetary reference system. Using these axis we omit external effects like earth rotating and orbiting. We define the ECEF axis as an orthogonal vector system, 90 degrees between each axis and right handed. The *Z* axis is the earth's rotation axis. The *X* axis is pointing to Aries star in the sky. The *Y* axis is defined by the vectorial product $Y=Z\cdot X$. This reference system is used for geo-positioning any vehicle with Cartesian coordinates. This is an AutoNAV4D93 add-on called *XYZ*. There is a version of this reference system called *Latitude Longitude Altitude (LLA)* which uses spheric coordinates and depends on the datum. Spheric coordinates are very often used for *Global Positioning System (GPS)*. The same coordinates has different projections depending on datum parameters. This is an AutoNAV4D93 add-on called *LLA*. These coordinates are explained in the chapter 3.6..

North East Down The NED axis (x, y, z) is the *center of gravity* reference system. Using these axis we omit external interactions like forces and rotations. The NED axis is also an orthogonal vector system. The *z* axis is always pointing to the earth's center. The *x* axis is always pointing to the earth's north, so the *y* axis is always pointing to the East. This reference system is used to define local interactions with the Vehicles's center of gravity. There is a version of this reference system called *Euler angles (Pitch, Roll, Yaw)* which uses spheric coordinates. We have used this reference system for compatibility with flight simulators.

Vehicle Centered Vehicle Fixed The VCVF axis (i, j, k) is the *Vehicle* reference system. Using these axis we omit internal movements like vehicle's parts displacements and rotations. The VCVF axis is also an orthogonal vector system. The *i* axis is always pointing to the vehicle's nose which is normally the flight direction. The *j* axis is always pointing to the vehicle's right side. The *k* axis is always pointing to the vehicle's up side. This reference system is used to define internal movements like control surface angles or antenna deployment. There is a version of this reference system called *Flight control angles (elevator, aileron, rudder)* which uses spheric coordinates.

3.5.. Datums

A *Datum* are the minimum parameters which define an origin of reference for a local area and a planetary surface. This surface is defined by a mathematical equation in order to simplify the calculations. This is an AutoNAV4D93 add-on called *Datum*. A reference *Datum* is a known and constant surface. It can be used to describe the location of unknown points. On Earth, the normal reference datum is sea level. On other planets, such as the Moon or Mars, the datum could be the average radius of the planet. The real surface around the entire planet showed in figure 3.4 by the ITC[5] is called *Geoid*. We have not used this model for compatibility purposes.

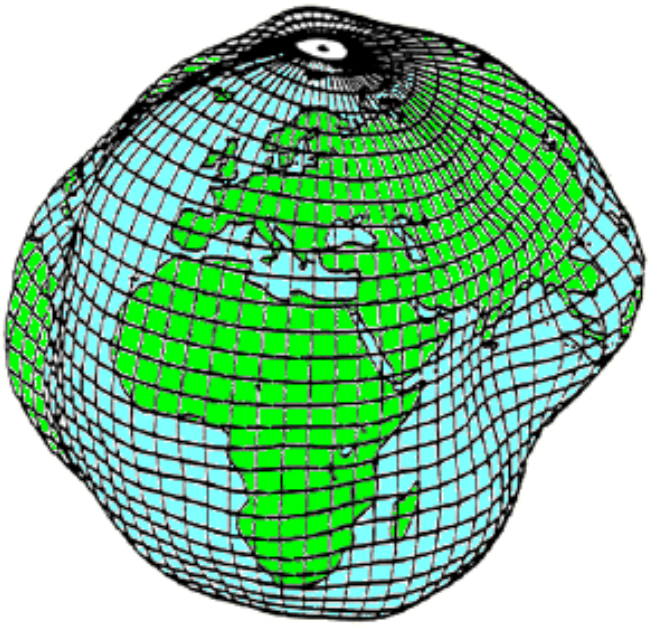


Figure 3.4: Perspective view of the Geoid (Geoid undulations 15000:1).

There is an universal *Datum* which is used by *GPS* (Global Positioning System) and also used by our *Virtual Vehicles*. This universal *Datum* is called *WGS-84* that means *World Geodetic System* defined in the year 1984. Nevertheless, a visualizer may uses a local *Datum* in order to show a correct projection in a near to ground view. For example if a vehicle, which uses the *WGS-84 Datum*, is located in Barcelona city (Spain), then the visualizer's projection will uses the *ED-50 Datum*.

3.6.. Geo-positioning

We use *Geo-positioning* in order to set the vehicle's position over the planet surface. In our case, the position over the earth is commonly defined by four ways:

XYZ These parameters are the Cartesian coordinates in the ECEF reference system explained before. This is an AutoNAV4D93 add-on called *XYZ*

LLA These parameters are the spheric coordinates in the ECEF reference system also explained before. LLA means *Latitude Longitude Altitude*. This is an AutoNAV4D93 add-on called *LLA*

UTM These parameters are defined inside a geographic grid. UTM means Universal Transverse Mercator. You can see the "DMA TECHNICAL MANUAL"[1] for more details. This is an AutoNAV4D93 add-on called *UTM*

UPS These parameters are defined inside a polar stereographic grid. UPS means Universal Polar Stereographic. Also you can see the "DMA TECHNICAL MANUAL"[1] for more details. This is an AutoNAV4D93 add-on called *UPS*

Every coordinate system has transformations between each other. It is important to understand that transformations depends on datum parameters. For this reason, everybody in the AutoNAV4D93 co-simulator uses the *WGS84* datum except when we need to visualize information. When we use same coordinates for two different datums there are two different projections. This is why we use a local datum for each projection.

CHAPTER 4. AUTONAV4D93. THE CO-SIMULATOR

4.1.. AutoNAV4D93. Architecture

We define co-simulation to integrate in a same scenario both, real Unmanned Aircraft Systems (UAS) and a simulated systems. The co-simulator, called *AutoNAV4D93*, works in the same platform as the Icarus's real UAS does. If the co-simulator *AutoNAV4D93* worked alone, it could be just a simulator but if it works inside the *UAS Service Abstraction Layer (USAL)*[1] it will constitute a powerful tool. Following we will explain how powerful could be the co-simulator *AutoNAV4D93* when it works with the USAL services. Also we will explain how a co-simulator could be configured as the old *AutoNAV* simulator in the chapter 4.1.4..

4.1.1.. Co-simulator architecture

The co-simulator is a set of components and services, added to the *UAS Service Abstraction Layer (USAL)*, which gives capabilities in fast prototyping and virtuality to the *USAL*. Virtuality in this case means that virtual's effects are similar like the real one.

Fast prototyping is reached by the use of USAL standards and reusable components. It allows a fast design and implementation of new functionalities. We will see in the chapter 4.2, some examples of services or components related to the co-simulator *AutoNAV4D93* as a part of the *UAS Service Abstraction Layer (USAL)*.

Virtuality is used for increase *safety* and reduce *design cost*. It Increases *safety* because we work with virtual services, like the real ones. A virtual UAV allows a safety testing process because it is not real but, a real UAV in test could be dangerous because it could crash. The system is designed, implemented and tested before it is definitively constructed. It reduces *design cost* because design and test phases are done in the ISIS integrated scenario instead the real life.

The co-simulator is an *Open Architecture*. It may have a Visor Service, a Commander, some Virtual Vehicles and Virtual Services and a Manager of Virtuality. Most of these services are used in the ISIS integrated scenario showed in the chapter 2.4..

[1] See chapter 2.2.2. for definition

4.1.2.. Components and services

The Co-simulator architecture, showed in figure 4.1 is composed by *Virtual Vehicles*, *Virtual Systems* and the *User Interface*:

- The *Virtual Vehicle (VV)* simulates the kinetic behavior of an UAS or mobile element, depending on the level of detail; simulation model could be modified with a more accurate results.
- The *Virtual System (VS)* useful for simulate small subsystem, could be associated to an UAS or mobile element. Is used for cover subsystems under development or just needed but not implemented.
- The *User Interface* is divided in three components: The *Visor Service* show a global vision of the fleet but also could be used as a component inside other service. The *Commander Service* is an on build service interface which allow interaction between the user and services through the visor.

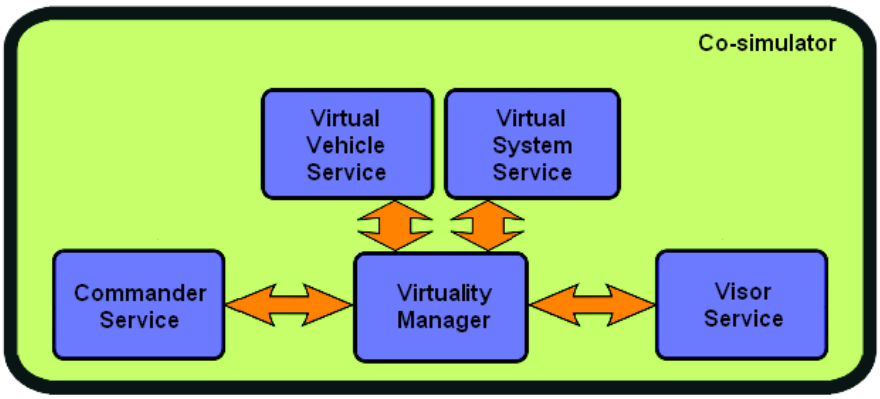


Figure 4.1: Five component and services constitute the co-simulator.

4.1.3.. Integrated architecture

In the *co-simulator*, some of these components are services and have the capability of publish and subscribe variables, events, etc. For this reason, these services require a communication layer called MAREA, explained before[2]. The *co-simulator*'s services use an USAL standard based on human-standing configuration files. This feature reduces the source weight because many repeated functions in the USAL are implemented in the same way. For example: A service can publish and register variables. All they have: the same source for all USAL services, same definition for public source variables in order to have an easy initial configuration of each service, etc. As an USAL standard, small modifications are easy changing just the XML configuration file. Complex modifications are implemented in the service's source by a programmer. Specific documentation are provided inside the source through XML format. Also, some user manuals are available

[2]See chapter 2.3. for definition

in each service, component and add-on. For capacity reasons of this TFC document, we do not include manuals here. *Visual Studio* or *Mono* allows the use of this documentation when programming. ICARUS team is glad to provide a free copy of the *ICARUS Development Kit (IDK)* on demand. Feel free to contact with ICARUS team for this purpose. As we

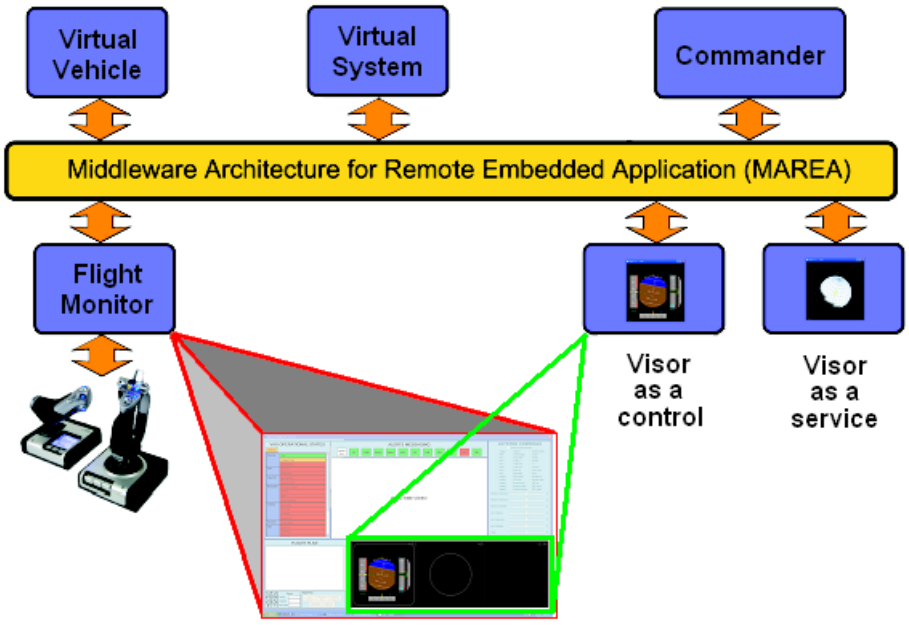


Figure 4.2: Some services and controls as a part of the USAL.

discussed before, the *co-simulator* itself could be useful as a simulator, but there is more usability when is integrated in the abstraction layer called USAL because could be used associated to any UAV mission. Reusability are available, for example using the *Visor* as a component inside the Flight Monitor which is an USAL's flight service (see chapter 2.2.2.). Also, when the *co-simulator* is used inside the ISIS integrated scenario, fast prototyping and safety are reached for new mission design. The *Visor* could be used as a fleet monitoring. *Virtual Vehicles* could be used as a way to predicts the imminent future of the UAV because is using virtuality, flying some minutes before the real time (see chapter 4.3.2.). Following we show an USAL's service list, classified by categories, and some examples of services or components related to the co-simulator *AutoNAV4D93*:

Flight services are all services in charge of basic UAS flight operations: autopilot, basic monitoring, contingency management, etc. As an example, the co-simulator's *Visor* used as a control inside the ISIS's Flight Manager.

Mission services are all services in charge of developing the actual UAS mission. As an example, the co-simulator's *Visor* service as fleet viewer.

Payload services are specialized services interfacing with the input/output capabilities provided by the actual payload carried by the UAS. As an example, the co-simulator's *Virtual System* which detects 'hot-spots' from a real time camera.

Awareness services are all services in charge of the safe operation of the UAS with respect terrain avoidance and integration with shared airspace. As example, the co-simulator's *Virtual Vehicle* used in order to predicts the future trajectory of a real UAV.

4.1.4.. Reusing the old design

Finally, we will explain how a co-simulator could be configured as the old *AutoNAV* in order to work alone as a simulator. Nevertheless, *AutoNAV* simulator details are far from this TFC document's scope. You may refer to the *PANS8168* main web page for further details[8].

A main *Visor* could be configured with a series of drawing layers as the old simulator, which allows to switch on/off airports, runways, airways, radio-aids, airplanes and vehicles, etc. Also a *Commander* could implements several commands as the old *AutoNAV* through a text *Console* and a *Tool-Bar*. The fleet could be implemented through *Virtual Vehicles* with a *Flight Plan* each one.

Older functionalities may be implemented as a series of *Virtual Systems* like the *Collision's detector*. Other simplest functionalities could be implemented as a *Widgets* which it will explained in following chapters. This new service, called *NAV4D*, could be started like other service through the middleware MAREA, but is far from this TFC document's scope.

4.2.. AutoNAV4D93. User interface

We have divided the interaction between a service and an user in three parts: the *Commander*, the *Visor* and the *Virtual Manager*. The *Commander* is responsible of flow information from user to services. The *Visor* is responsible of flow information from service to user. The *Virtuality Manager* is responsible of the time to deliver this flow when virtual elements are in different epoch[3].

4.2.1.. The Visor as a Service

The *Visor*, as a service, is able to shows a global vision of the fleet but also could be reused as a component inside other service. The *Visor* represents visual and acoustic information, in a specific format. We use human-standing XML configuration files in order to allow an easy user interface design and adaptation to new mission's requirements. Complex modifications may be done by adding modular components to the source. For this reason, source documentation may be provided in order to make feasible this new programmer's implementations. These modular components are called *Addons*.

The *Visor* is formed by added-on components; grouped together, forms the reusable user interface. Some add-on standards are accessible for *USAL* services in order to allow an information's interchangeability. In one hand, we have found in the main common needs, for many UAS civil missions, some similar used displays; these are flight or system information monitoring in a popular appearance, image processing as cameras, global positioning with 3D references, mapping projections for a defined *datum*[4], etc. In the other hand, interaction with the operator requires an easy control technique, far from this TFC document's scope. You may refer to the GEDIS-UAV guide[15] by *Mr. Salvador Lorite* for further details.

The *Visor* has a point of view, represented by the camera, which is automatically associated to any vehicle with the same name. This is an AutoNAV4D93 add-on called *Camera*. The *Camera Manager* manages all defined cameras by the *Visor*'s configuration XML file. It could be more than a camera associated to the same vehicle. The *Camera Manager* allows a multiple scenario's view with an independent context of view. This is an AutoNAV4D93 add-on called *CameraManager*.

We have found that visual process design is drawing layer oriented. Many applications have the capability of switch visual layers, so drawing layers will group items or functions, easy to show or hide and easy to interchange between other mission designs. This is an AutoNAV4D93 add-on called *Layer*. When a drawing layer changes with a subscribed variable, his dependence is recorded in a *register*. When this subscribed variable changes, only the associated layers are retraced. This is an AutoNAV4D93 add-on called *Reg*.

We have called *Tracers* to the elemental trace primitives like *Draw*, *Arc*, etc. We have called *Widgets* to the instruments or graphics, because they have more complex functions. Some examples could be a *Compass*, a *HSI* (Horizontal Situation Indicator) or a *Map*. These are

[3] Epoch means not only a local time but also a different time speed

[4] A *Datum* is a set of parameters which defines a local ellipsoid reference

built by elemental trace primitives and they are able to change the *Visor*'s aspect easily depending on values of some variables subscribed by the *Visor*; as is specified in the *Visor*s configuration XML file. The *Tracers* primitives introduces known defaults intelligent constants in case of out of range or incorrect values sending by a registered variable. Designer not to be worry about both, communication and format, just needs to know which services variable may be visualized by the *Visor*. System colors could be defined for some known *Widgets* in order to allow the same aspect with drawing *Layer*'s independence. This is an AutoNAV4D93 add-on called *ColorNAV*.

In Figure 4.3 you can see in the left side, a planetary captured view of a vehicle and his geo-position. In the right side you can see a local view of a fleet composed by three virtual vehicles.

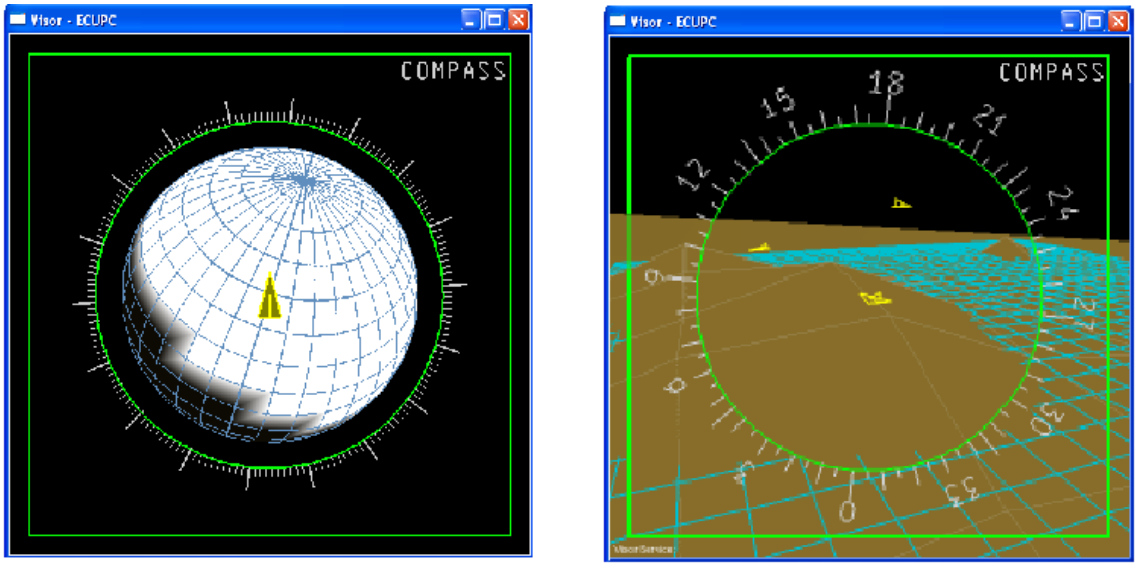


Figure 4.3: Visor with a planetary view and a local view

4.2.2.. The Visor as an User Control

Other powerful *Visor*'s use is when is an *User Control*. An example of this is the *Primary Flight Display (PFD)* showed in figure 4.4. This control (background color in black) is inside the *Flight Monitor*. This control manages as the service does, all drawing layers and variables. The *Flight Monitor* is a service inside the integrated scenario (ISIS) presented in chapter 2.4.. This service has been implemented by Borja López who also implemented some *Widgets* for this purpose.

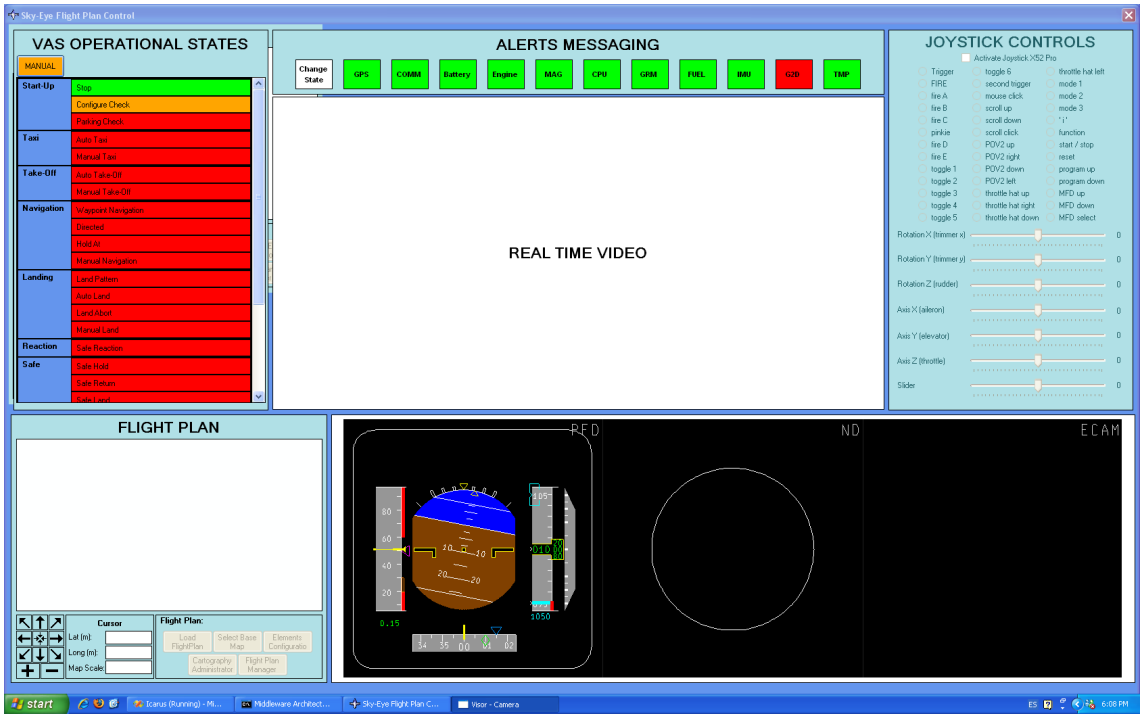


Figure 4.4: An example of Visor as a control inside the Flight Monitor on ground station.

4.2.3.. Commander Service

A *Commander Service* is required in order to separate the visualization process from the command process, the commander uses *Remote Invocation* primitives to send commands to the services. Allow an interaction with the showed widgets to take an action associated to any UAS visualized by the Visor. These commands, out from the TFC document'scope, could be:

- Words with parameters showed by the *Visor*'s console
- A button pressed in the *Visor*'s toolbar
- A mouse click over a *Widget* showed by the *Visor*'s camera
- A talking command used on the *see-and-avoid*[5]

[5]The see-and-avoid is a surveillance system for UAS

4.2.4.. Virtuality Manager

The *Virtuality Manager* creates or modifies a *Virtual Vehicles* or *Virtual Systems*. The *Virtuality Manager* will be useful when *Virtual Vehicles* or *Virtual Systems* work in an accelerated time simulation or a different epoch from the Network's time. The *Virtuality Manager* segregate the traffic information in each service local time. Is important to understand that the *Virtuality Manager* may be able to mange information from the same virtual item in more than an epoch in the same scenario. Virtual components such as *Virtual Vehicles* or *Virtual Systems* may be able to allow the information in any required epoch instantly. This is because real services only are able to allow the information in the Network's time (because they are real, of course). At the moment, only a *Visor* will be able to show information of a future *Virtual Vehicle* even if the *Visor* has an associated camera to him with the same local time. Otherwise this camera will show a different moment of this *Virtual Vehicle*. This feature may be used to predicts the future inside a virtual world similar to the real one. We have tested this feature in the old *AutoNAV4D* simulator version 89.

These capabilities may be implemented in the Middleware layer to make a more efficient use of the channel bandwidth. In this aspect, the middleware must add a *Virtuality Manager* which makes predictions in order to reduce the transmission load and dispatch the information on time for the local service epoch. Specifications about the *Virtuality Manager* is far from the scope of this TFC document.

4.3.. AutoNAV4D93. Virtual services

In this chapter we introduce the concept of *Virtual Services*. Two cases of virtual services inside the co-simulator are: *Virtual Vehicles* and Virtual Systems.

4.3.1.. Definition of Virtual Services

Virtual Services are services of the *UAS Service Abstraction Layer (USAL)*. A *Virtual Services* has a function inside the ISIS testbed, but also, as a *Virtual* it may be predictable. Because *Virtuality Manager* could ask about the state of a *Virtual Service* in any epoch[6], It may be capable of deliver information as a function of time and it may be a fast response. A prediction tolerance is also considered.

This fast response is reached extending the concept of life's cycle. *Virtual Services* have defined their life's cycle if is possible. Some times this information depends on other *Virtual Services* and it is able to allow quickly and for any epoch. Other times, this information depends on real time services. In this case, information is calculated in the moment but only in the actual time; *Virtuality* is reduced to a short period of time covered by the prediction tolerance. Otherwise, life's cycle may be partially defined and *Virtuality* is deprecated. Due to short capacity of this TFC document, we will not explain *Virtuality Services* in deeply.

4.3.2.. Virtual Vehicle

A *Virtual Vehicle* is a virtual service. *Virtual Vehicles* generate kinetic information for any epoch, like flight telemetry, but without the existence of a real vehicle. A *Virtual Vehicle* could be useful for add vehicles to our mission in order to test them. The *Virtual Vehicle* is intended to be used in *fast simulations*[7], capability which is not implemented by most of the commercial flight simulators. *Virtual Vehicles* are useful when in addition we want simulate some mobile services, far than an UAV airplane, like mobile ground antennas on a truck, or a firefighting walking near a hot spot which needs a radio-coverage, etc.

Virtual Vehicles have the capability of allow instantly the geo-position[8] for a required time. This is possible, because *Virtual Vehicles* have defined all his life's cycle before they starts. If any event is produced over this *Virtual Vehicle*, it may recalculates his life's cycle again. We have called *patterns* to each vehicle behavior. The life's cycle is composed through a list of patterns which starts at an epoch and finished when the next *Pattern* starts. The *Virtuality Manager* uses this capability in order to allow different epoch in the same scenario. For this reason a real vehicle only is able to allow the geo-position in the network's epoch. Also, a *Virtual Vehicle* may have implemented a *Virtual Autopilot Layer* which allows an standard autopilot to the USAL, but these features are far from the scope of this TFC document.

[6] Epoch means not only a local time but also a different time speed

[7] Accelerated time or different epoch to the network's epoch

[8] geo-position: the position over the planet's surface

Table 4.1: Summary of Virtual Vehicle Primitives

Interaction plane	attribute	command	units
Physical plane			
Kinetic	posXYZ	IllegalXYZmove(x0,y0,z0)	meters
Physics	cog	Move(XYZ)	meters
Cognitive plane			
Flight Director	direction	GoTo(tx,ty,tz)	meters
Flight Plan	waypoint	Next(LLA)	degrees
Strategic plane			
Navigation	standbyFP	NewFlightPlan(FP)	-

4.3.2.1.. Vehicle's standardized behavior

We have standardized the behavior of any mobile vehicle through a series of functions and attributes as we summarize in the basic table 4.1. However, refer to especial documentation in the component's source. Also you can refer to chapter A.1. for additional details.

This summarized analysis is divided in three interaction's planes: The physical plane, the cognitive plane and the strategic plane. The *Virtual Vehicle* implements an autopilot based on a standard flight director and navigation directives but far from this TFC document's scope.

Physical plane is used for kinetics, physics and autopilot interactions whit out any decision. We will not explain more details about these interactions.

Cognitive plane is used for make decisions in order to maintain a safe flight through flight director and flight plan interactions. We will not explain more details about these interactions.

Strategic plane is done in order to reach a strategic flow efficiency for a fixed scenario through navigation and traffic management interactions. We will not explain more details about these interactions.

4.3.2.2.. Vehicle's reference systems

Real vehicles have only one reference system, but *Virtual Vehicles* have, for different purposes a dual reference systems plus the vehicle's reference system. One is to be used for geo-positioning like every mobile vehicle but in addition we introduce a local reference system in order to interact with the environment. Any service could interacts with the *Virtual Vehicle* in each plane: In the physical plane through 3D commands, in the cognitive plane through target commands and finally, in the strategic plane through navigation's directives. Refer to chapter A.1. for more details about *Virtual Vehicle*'s commands.

ECEF The ECEF reference system is fixed to the Earth's axis rotation and is suitable for geo-positioning the UAS. Geo-position is protected from any external modification with an 'Illegal change message'.

NED The NED reference system is attached to the UAV but his Y axis is normal to the ground surface and is used to introduce local parametric disturbances, to the vehicle's center of gravity, in the scope to be added to the ECEF geo-position by itself.

VCVF The VCVF is fixed to the vehicle's fuselage and is used for reference any internal movement like the control surfaces.

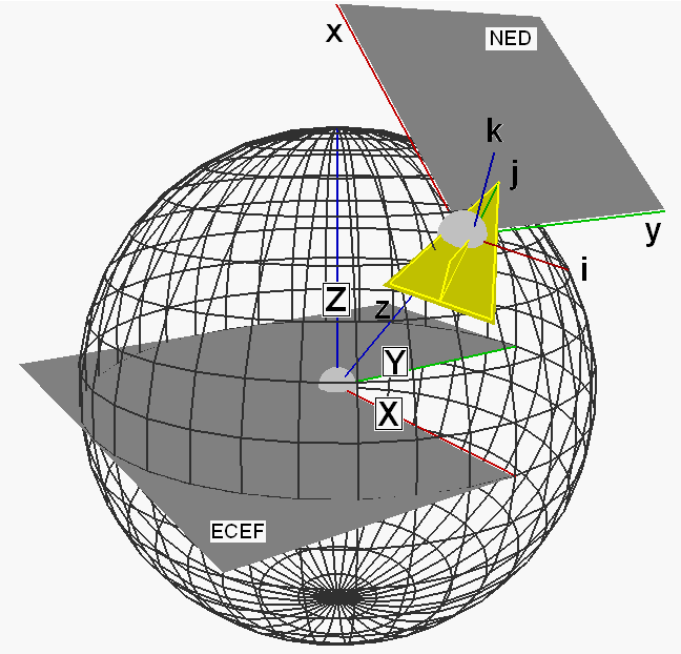


Figure 4.5: Three different reference systems for three purposes: ECEF, NED, VCVF.

4.3.3.. Virtual System

The *Virtual System* is a virtual service. A *Virtual System* simulates the existence of a small subsystem for any epoch. The *Virtual System* abstracts the designer from the rest of the USAL platform in order to solve a singular function, normally associated with an UAS. USAL platform could be reused from a previous tested missions.

Virtual Systems are useful when we are developing new subsystems. *Virtual Systems* allows fast prototyping: no expensive implementations are required because the work is done directly in the design phase. Simulations are done as real as needed in the aim of reach the best and faster solution valid for the mission requirements. After of them, a real implementation is able; interchange between *Virtual System* and a real system is transparent to the networks services. *Virtual Systems* allow an easy configuration way like human-standing configuration lists like XML format files. This is an standard which is able for *USAL* services. A *Virtual System* as a service, has many source implemented. The programmer only may add his functions to the *Virtual Vehicle Process*.

4.3.3.1.. Virtual Radar. An example of Virtual System

We have implemented an example of *Virtual System* which consist on a radar function. Figure 4.6 shows a *Visor* and the *MAREA*'s console. Inside the *Visor*, a 3D drawing *Layer* shows a conflict between two virtual vehicles. The *Radar Service*, which is a *Virtual Service*, is subscribed to each *Virtual Vehicle* position. Every two seconds, the *Radar Service* calculates distances between vehicles. The *Radar Service* may publish a variable as a *Resolution Advisory* with warnings for each vehicle which is implied in a possible collision. Any other services like *Virtual Vehicles*, could subscribe to this variable if they need a surveillance service.

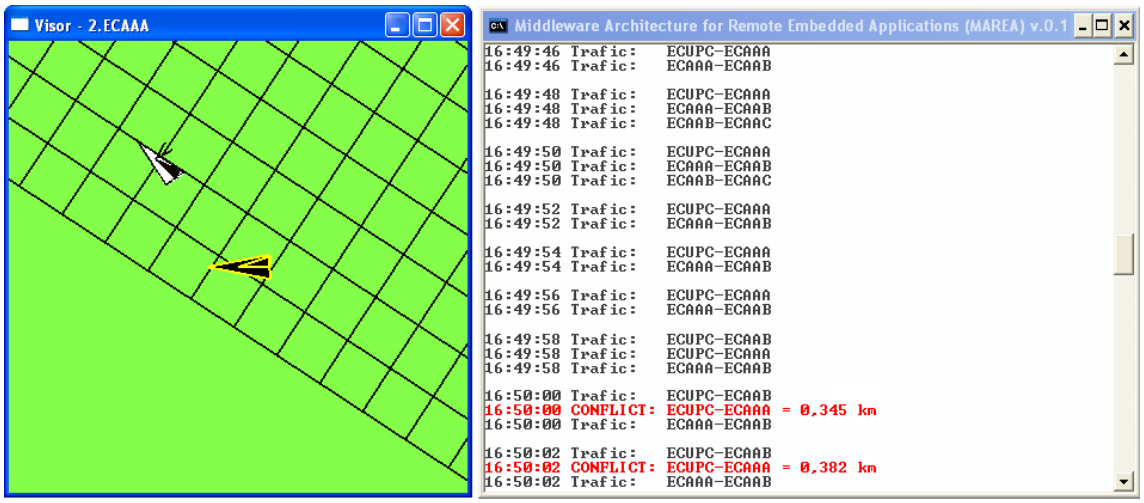


Figure 4.6: An example of Virtual System: a virtual radar service.

4.3.3.2.. Virtual Engine Monitoring. An example of fast prototyping

We present an example of cooperative fast prototyping. It was the *Engine Monitoring* designed by Raquel García from the ICARUS team. The *Engine Monitoring* is a par of the ISIS integrated platform explained before in the chapter 2.4.. We use a *Virtual System* in order to develop the main functionality: *to monitor the engine health*. The *Engine Monitoring* was subscribed to some kinetics variables like *Airspeed*, *Vertical Speed* and *Altitude*. Temporally, *Engine Monitoring* simulates the *rpm*[9] and publish this value through a registered variable called *rpm*. This value was showed in a *Visor* inside a layer called *Electronic Centralized Aircraft Monitor (ECAM)*. We have developed a dedicated indicator using a *Widget* called *RPM*. Now, platform is adapted to the new service and is easy to implement a real *Engine Monitoring* which will acquire the real *rpm* value instead a simulated one. Also, the *ECAM* layer could be integrated inside the *Flight Monitor* using a *Visor* as a component instead a dedicated service. This example is showed in chapter 4.2.2..

[9] rpm: Revolutions per minute

4.4.. AutoNAV4D93. Diagrams

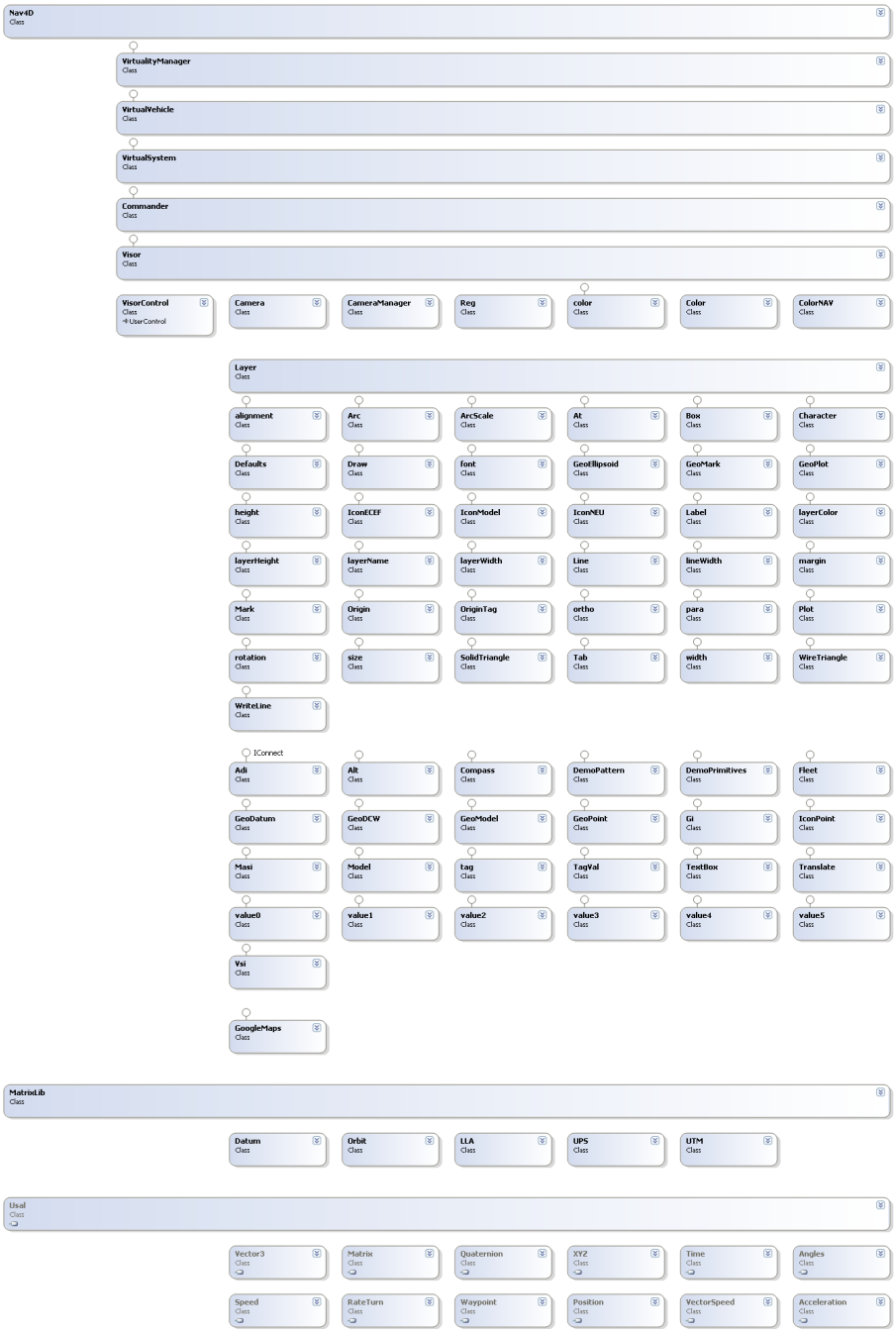


Figure 4.7: Main class diagram for the AutoNAV4D93: Nav4D, MatrixLib and Usal

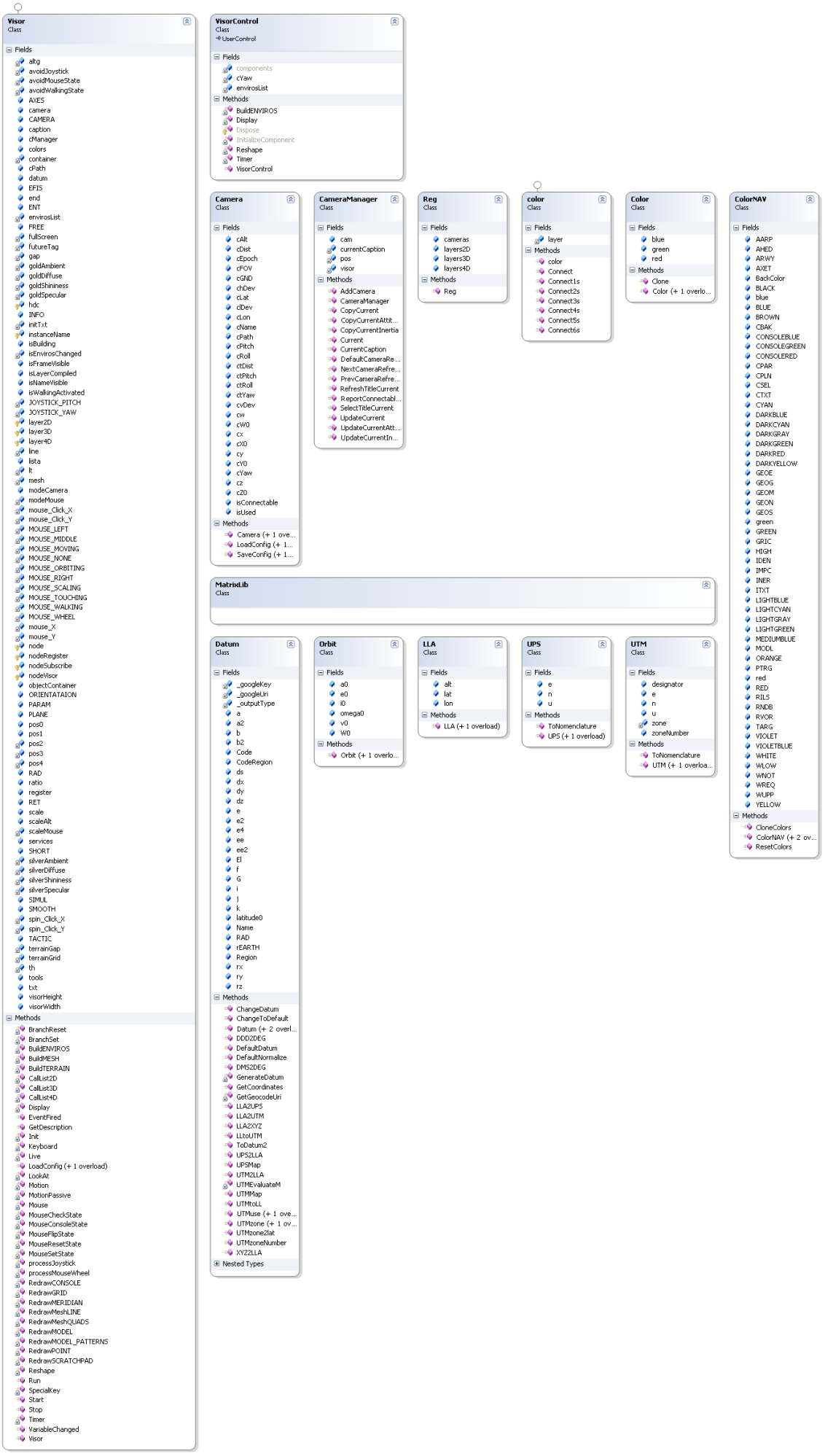
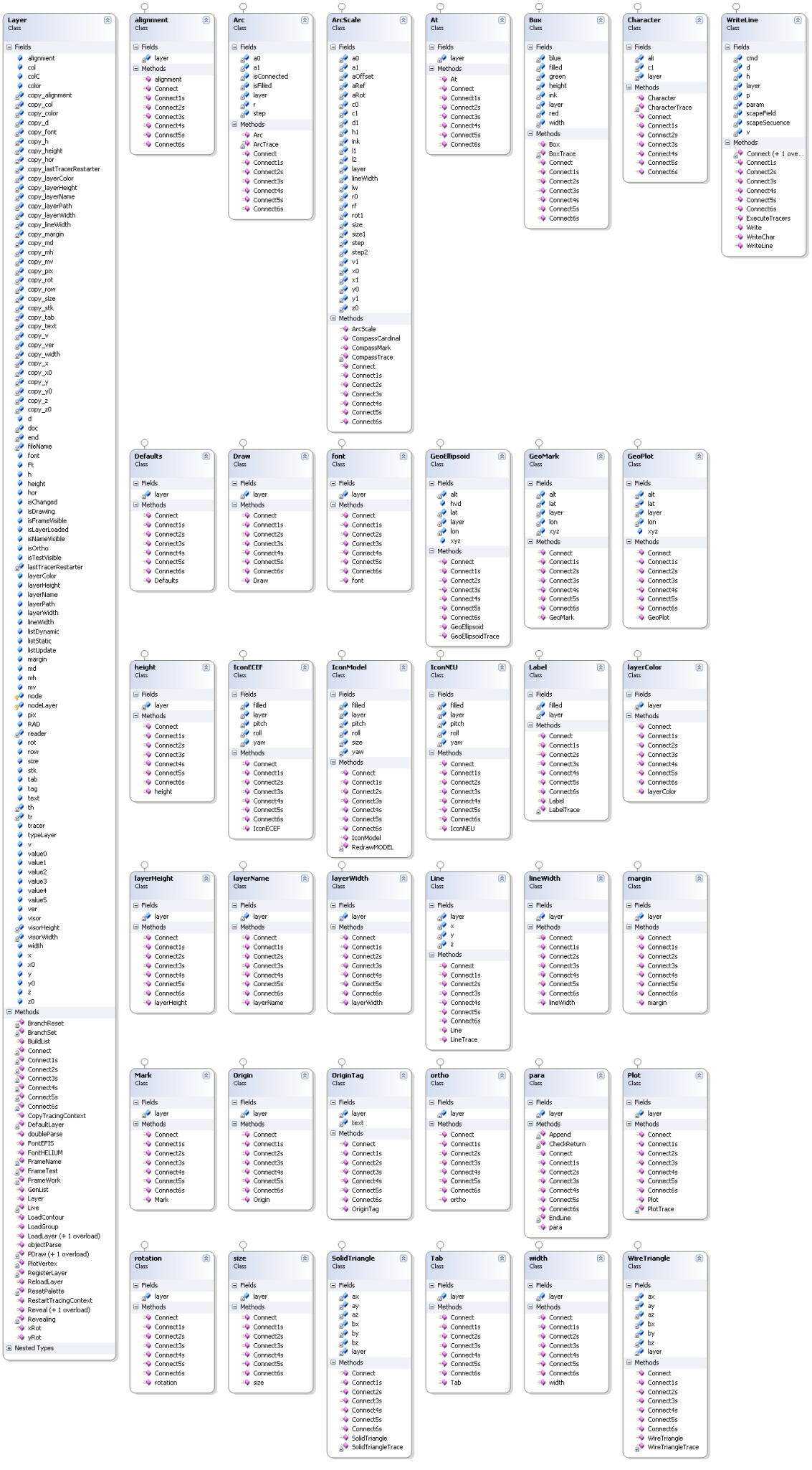


Figure 4.8: Class diagram for the Visor


Layer
Class
Fields
alignment
col
colC
color
copy_alignment
copy_col
copy_color
copy_d
copy_font
copy_h
copy_height
copy_hor
copy_lastTracerRestarter
copy_layerColor
copy_layerHeight
copy_layerName
copy_layerPath
copy_layerWidth
copy_lineWidth
copy_margin
copy_md
copy_mh
copy_mv
copy_pix
copy_rot
copy_row
copy_size
copy_stk
copy_tab
copy_text
copy_v
copy_ver
copy_width
copy_x
copy_x0
copy_y
copy_y0
copy_z
copy_z0
d
doc
end
fileName
font
Ft
h
height
hor
isChanged
isDrawing
isFrameVisible
isLayerLoaded
isNameVisible
isOrtho
isTestVisible
lastTracerRestarter
layerColor
layerHeight
layerName
layerPath
layerWidth
lineWidth
listDynamic
listStatic
listUpdate
margin
md
mh
mv
node
nodeLayer
pix
RAD
reader
rot
row
size
stk
tab
tag
text
th
tr
tracer
typeLayer
v
value0
value1
value2
value3
value4
value5
ver
visor
visorHeight
visorWidth
width
x
x0
y
y0
z
z0
Methods
BranchReset
BranchSet
BuildList
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
CopyTracingContext
DefaultLayer
doubleParse
FontEFIS
FontHELIUM
FrameName
FrameTest
FrameWork
GenList
Layer
Live
LoadContour
LoadGroup
LoadLayer (+ 1 overload)
objectParse
PDraw (+ 1 overload)
PlotVertex
RegisterLayer
ReloadLayer
ResetPalette
RestartTracingContext
Reveal (+ 1 overload)
Revealing
xRot
yRot
Nested Types
alignment
Class
Fields
layer
Methods
alignment
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
Arc
Class
Fields
a0
a1
isConnected
isFilled
layer
r
step
Methods
Arc
ArcTrace
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
ArcScale
Class
Fields
a0
a1
aOffset
aRef
aRot
c0
c1
d1
h1
ink
l1
l2
layer
lineWidth
lw
r0
rf
rot1
size
size1
step
step2
y1
x0
x1
y0
y1
z0
Methods
ArcScale
CompassCardinal
CompassMark
CompassTrace
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
At
Class
Fields
layer
Methods
At
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
Box
Class
Fields
blue
filled
green
height
ink
layer
red
width
Methods
Box
BoxTrace
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
Character
Class
Fields
ali
c1
layer
Methods
Character
CharacterTrace
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
WriteLine
Class
Fields
cmd
d
h
layer
p
param
scapeField
scapeSecuence
v
Methods
Connect (+ 1 ove...
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
ExecuteTracers
Write
WriteChar
WriteLine
Defaults
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
Defaults
Draw
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
Draw
font
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
font
GeoEllipsoid
Class
Fields
alt
hvd
lat
layer
lon
xyz
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
GeoEllipsoid
GeoEllipsoidTrace
GeoMark
Class
Fields
alt
lat
layer
lon
xyz
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
GeoMark
GeoPlot
Class
Fields
alt
lat
layer
lon
xyz
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
GeoPlot
height
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
height
IconECEF
Class
Fields
filled
layer
pitch
roll
yaw
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
IconECEF
IconModel
Class
Fields
filled
layer
pitch
roll
size
yaw
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
IconModel
RedrawMODEL
IconNEU
Class
Fields
filled
layer
pitch
roll
yaw
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
IconNEU
Label
Class
Fields
filled
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
Label
LabelTrace
layerColor
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
layerColor
layerHeight
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
layerHeight
layerName
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
layerName
layerWidth
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
layerWidth
Line
Class
Fields
layer
x
y
z
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
Line
LineTrace
lineWidth
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
lineWidth
margin
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
margin
Mark
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
Mark
Origin
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
Origin
OriginTag
Class
Fields
layer
text
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
OriginTag
ortho
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
ortho
para
Class
Fields
layer
Methods
Append
CheckReturn
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
EndLine
para
Plot
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
Plot
PlotTrace
rotation
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
rotation
size
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
size
SolidTriangle
Class
Fields
ax
ay
az
bx
by
bz
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
SolidTriangle
SolidTriangleTrace
Tab
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
Tab
width
Class
Fields
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
width
WireTriangle
Class
Fields
ax
ay
az
bx
by
bz
layer
Methods
Connect
Connect1s
Connect2s
Connect3s
Connect4s
Connect5s
Connect6s
WireTriangle
WireTriangleTrace

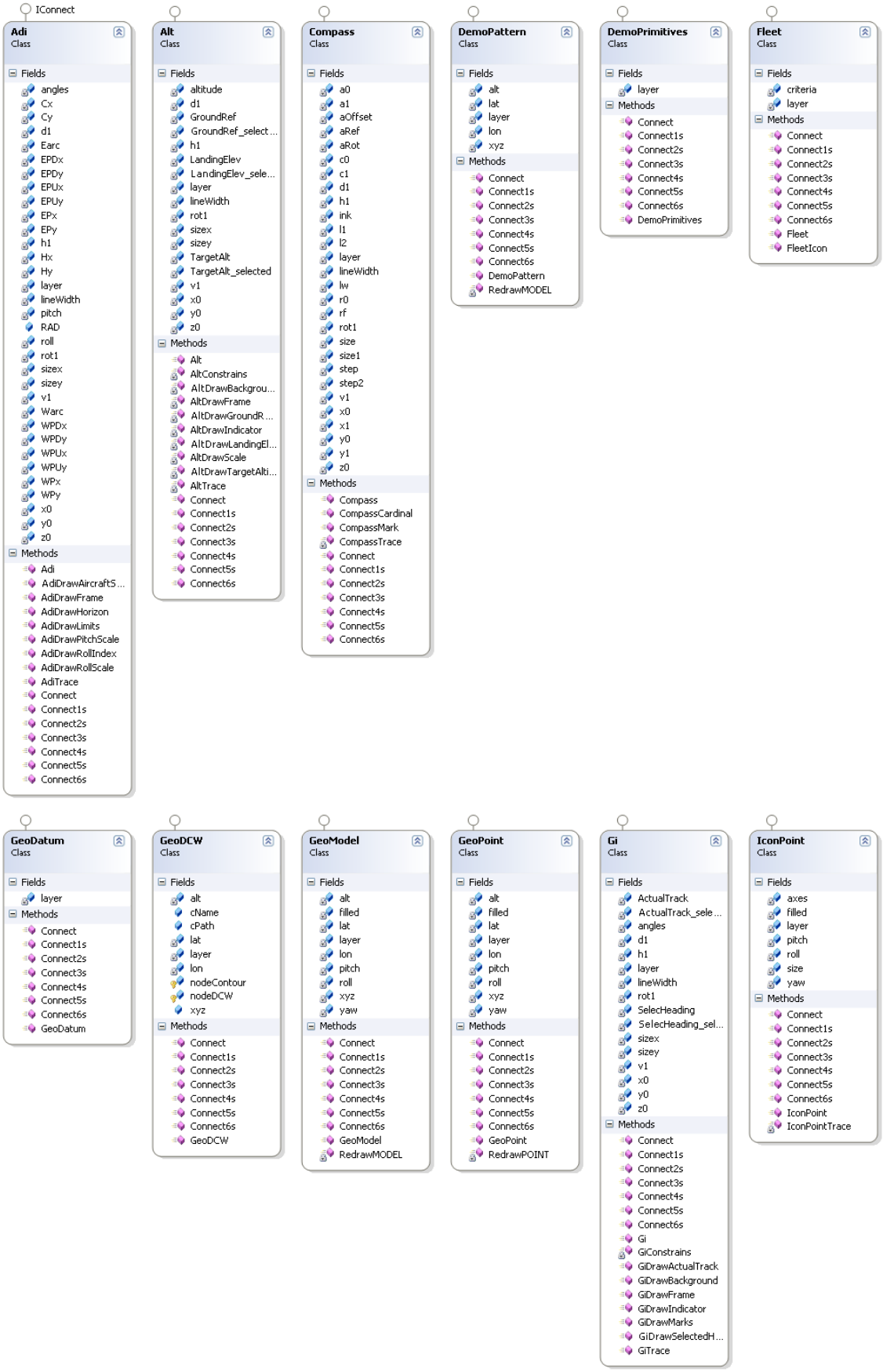


Figure 4.10: Class diagram for the Widgets (I)

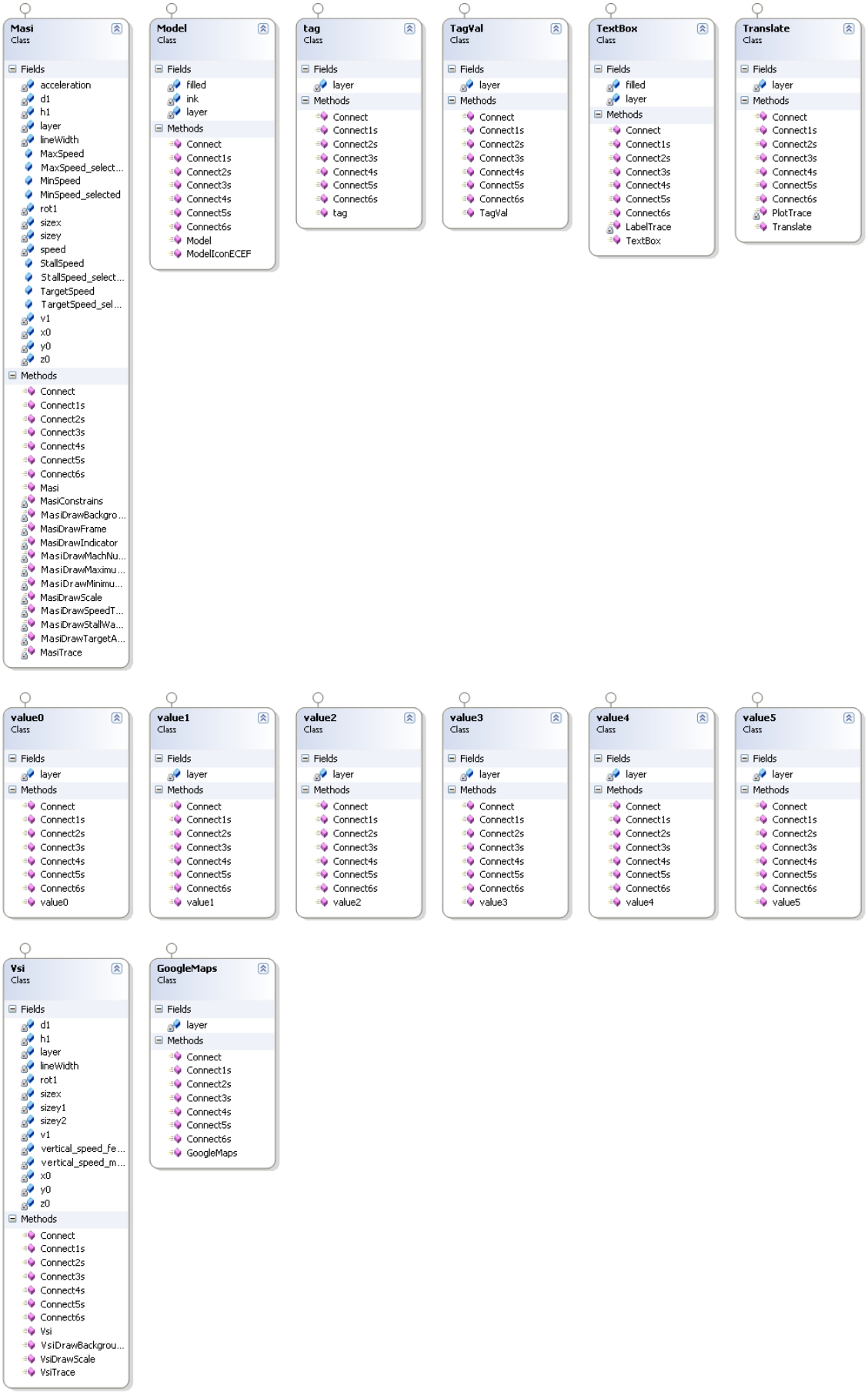


Figure 4.11: Class diagram for the Widgets (II) and Updaters

CHAPTER 5. FUTURE WORK

5.1.. Conclusions

In this Career Final Work (TFC) we use a lot of knowledge gained through this university career in this co-simulator. As a result of an ICARUS team collaborative work, using components an services together, we are able to improve the ISIS integrated scenario in which the co-simulator is an important piece of them. Many bugs and faults have been corrected thanks to this collaborative work. This co-simulator is a large project and this TFC work is only a piece of the *AutoNAV4D93* project. So, many targets to be accomplished in a future as a result of ISIS developing work.

5.2.. Future improvements for the ISIS integrated scenario

This is a list of some tasks not yet completed:

- *Virtual Vehicle*'s *Patterns* may be implemented. *Virtual Vehicles* only make holding patterns. Also Flight Plans are not yet implemented.
- *Virtuality Manager* is not yet implemented. Only real time is available.
- Some improvements for optimizing the network's channel may be implemented like *Serialization* and telemetry trend prediction.
- Improvement in the quality of the telemetry reducing the noise of the inaccurate values.
- Test *Virtual Vehicles* with different epoch in the same scenario.
- Test accelerated simulations and the future predictions for *Virtual Vehicles*.
- The XML standardization sometimes uses *Reflection*. We may consider to use a closed list of source's variables.

5.3.. Pending work for the user interface

Widgets are powerful if we have a large list of them such as topographic, terrain, highways, etc. In order to make easy to use *Layers*, some icons are required like Boeing and Airbus appearance *Widgets*. *Commander* is not yet completed. *Visor*'s *Widgets* may be associated to *Commands*.

We have implemented only the *ICARUS Development Kit (IDK)* with a basic version of MAREA and two simple services, one as a producer and other as a consumer. Other development kits are required like a *Layer Development Kit* and a *Virtual Development Kit*.

BIBLIOGRAPHY

[1] Stanley O., S. “The universal grids: UTM and UPS”. *TECHNICAL MANUAL 8358.2*. Defense mapping agency (Distribution is unlimited). (September, 1989)

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

C sharp Graphics Library (April, 2003)

[3] http://aeroupc.upc.es/

AeroUPC (July, 2004)

[4] López, J.; Royo, P.; Pastor, E.; Barrado, C.; Santamaria, E. “A Middleware Architecture for Unmanned Aircraft Avionics”. *Computer Architecture Dept.* Technical University of Catalonia, Spain (August, 2004)

[5] http://kartoweb.itc.nl/geometrics/

International Institute for Geo-Information (ITC), Enschede (March, 2006)

[6] http://icarus.upc.es/

Icarus Group (October, 2006)

[7] http://nix.upc.es/aero/wiki/index.php/MOL%C3%89CULA

MOLECULE (May, 2007)

[8] http://nix.upc.es/aero/wiki/index.php/PANS8168

PANS8168 (August, 2007)

[9] Alvarez, V.; Prats, X.; Soley, S.; Montolio, J. “EGNOS: A big system for small aerodromes”. *European Navigation Conference 2008*. Toulouse: Centre National d’Etudes Spatials. (May, 2008)

[10] http://www.taoframework.com/

The Tao Framework (May, 2008)

[11] http://www.mono-project.com/Main_Page/
http://www.mono-project.com/Image:Md1.png

Mono (Software) (May, 2008)

[12] http://msdn.microsoft.com/en-gb/vs2005/default.aspx
http://blogs.msdn.com/mikhailarkhipov/

Visual Studio 2005 (May, 2008)

[13] http://www.protech.be/

Protech Air Division (Promodels) (June, 2008)

[14] http://msdn.microsoft.com/en-gb/netframework/default.aspx

.NET Framework Development Center (June, 2008)

[15] Lorite Antezana, S. “DISEÑO DE INTERFACES DE SUPERVISIÓN DE VEHÍCULOS AÉREOS NO TRIPULADOS”. *Master thesis*. Technical University of Catalonia (June, 2008)

[16] `http://www.siggraph.org/s2008/`

Computer graphics and interactive techniques community (August, 2008)

[17] Pablo Royo, Juan López, Joshua Tristancho, Juan Manuel Lema, Borja López, and Enric Pastor. “Service Oriented Fast Prototyping Environment for UAS Missions”. *AIAA’09 Conference of Guidance, Navigation, and Control*. American Institute of Aeronautics and Astronautics. (August, 2009)

APPENDIX

APPENDIX A. PROTOTYPING

A.1.. Virtual Vehicle Specification

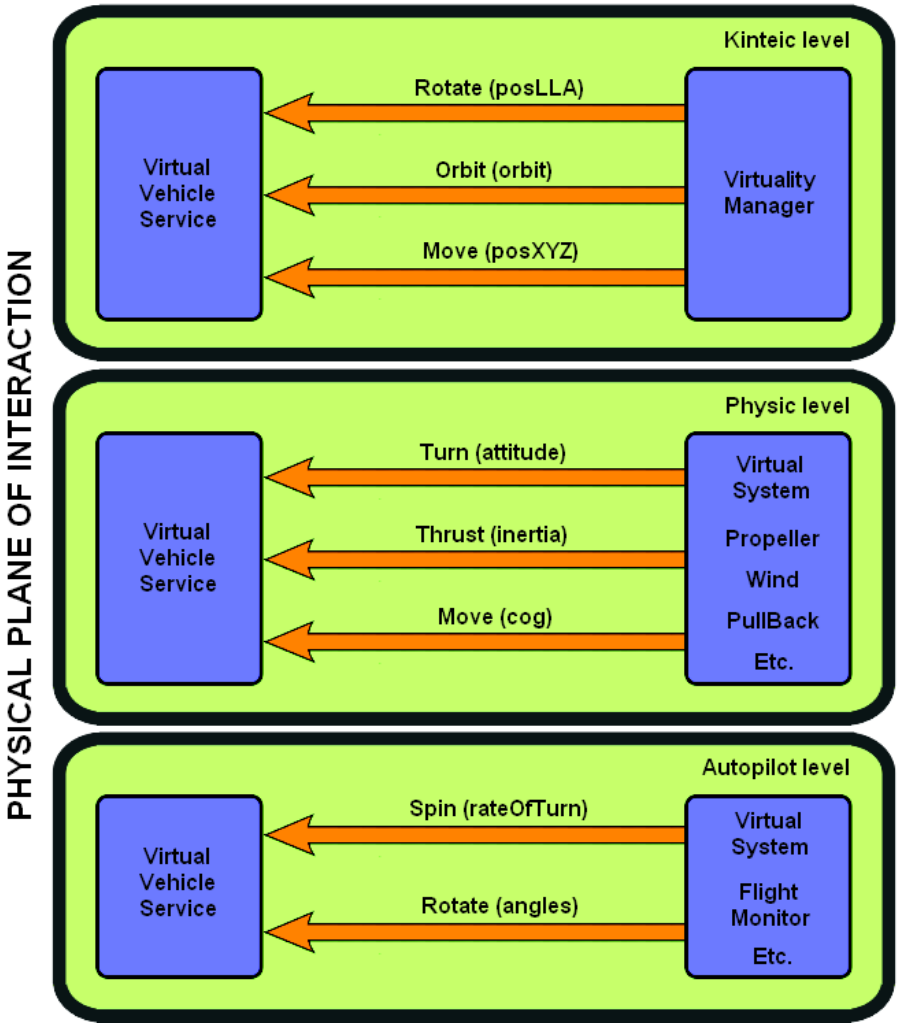


Figure A.1: Physical plane. Interactions between a Virtual Vehicle and his environment.

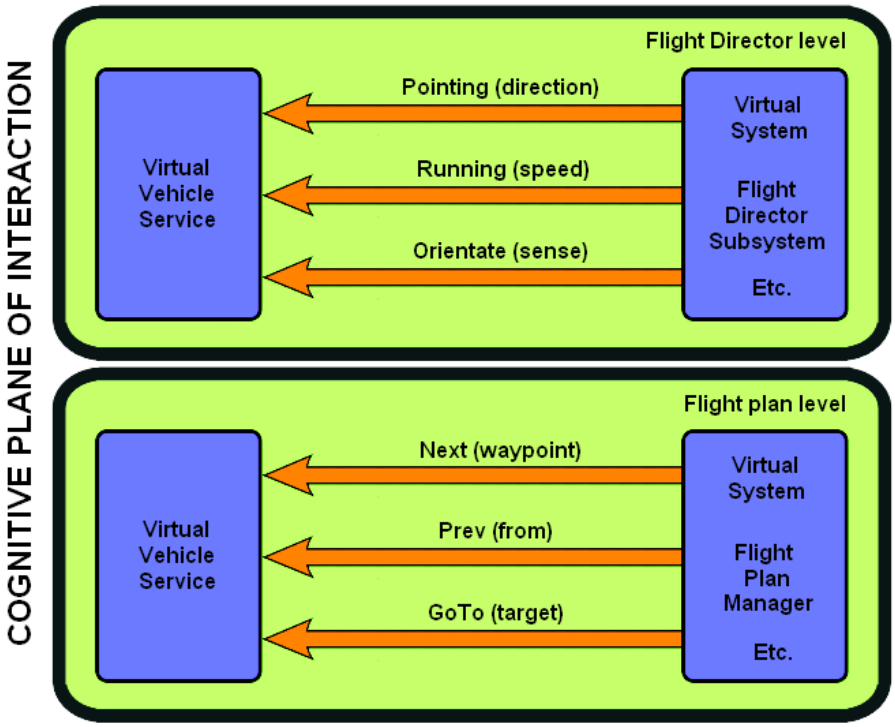


Figure A.2: Cognitive plane. Interactions between a Virtual Vehicle and cognitives services.

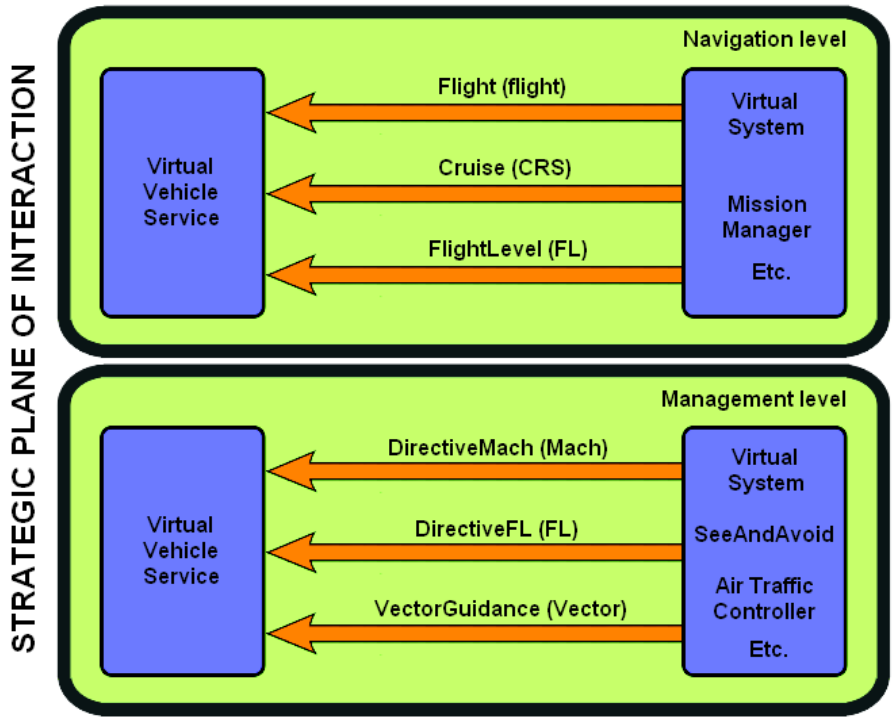


Figure A.3: Strategic plane. Interactions between a Virtual Vehicle and a Controller.