VIDESUPRA

AN ASTROPHYSICS SIMULATOR

by

Grant Jared Nelson

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2011

APPROVAL

of a thesis submitted by

Grant Jared Nelson

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citation, bibliographic style, and consistency, and is ready for submission to The Graduate School.

Dr. Denbigh Starkey

Approved for the Department of Computer Science

Dr. John Paxton

Approved for The Graduate School

Dr. Carl A. Fox

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Grant Jared Nelson

April 2011

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

LIST OF FIGURES

LIST OF FIGURES – CONTINUED

LIST OF EQUATIONS

Equation Page

LIST OF CODE SNIPPETS

ABSTRACT

For small satellite development groups building, testing and launching satellites can be very expensive. Since satellites can have no physical contact while in space testing satellites before launch is very important. To test satellites at low cost software simulation should be used. Several professional software simulation packages exist but they can be expensive to licensed and be trained for. The goal of this paper is to describe a free open source astrophysics simulator that can be used for testing and prototyping. The simulator must be flexible and customizable so that it can be modified for each satellite project. The solution is VideSupra. VideSupra is a software simulator with a built in package for astrophysics simulation. VideSupra is a plug and play application so can be quickly modified and extended as needed. For small satellite development groups VideSupra will provide the needed low cost testing tool.

PROBLEM AND GOALS

Satellite development is costly in both time and money. Additionally, because of the nature of satellites, testing them is extremely important but also extremely complicated. Many satellite groups develop satellites between 1 and 200 kg and typically less than 50 cm per side, called smallsats or nanosats. These groups are usually noncommercial, such as college satellite programs. For these groups most testing methods are not feasible because the test cost compared to the satellite's cost is so great. For such groups a simulated testing environment would greatly reduce the overall cost of development.

The goal of VideSupra is to provide a flexible, extendible, free, open source software simulator which can be used to test satellites. Although no software testing solution will ever be able to replace all of the more expensive physical tests, for many development groups VideSupra will be an effective means to minimize the number of more expensive tests needed to debug a satellite. VideSupra can also be used purely as a simulator, while not testing a satellite. In this mode VideSupra can be used to aid in the research, design, development, and launch trajectory selection of a potential satellite.

VideSupra's capabilities include:

- Written in the C# with .Net Framework 4.0.
- Dynamically loads via an easily extendable plug and play system.
- Contains the PhySim2 library for generating a model view controller physics simulation.
- Provides a toolset of data structures used by most simulations.
- Predefines an astrophysics simulation plug-in that comes with several controls.
- Predefines plug-in views for displaying raw data, data in 2D graphs, and 3D data

modeling.

- Flexible scheme for creating and customizing scenarios to suit the users' needs.
- Scenario setup is done via an intuitive graphical user interface.

This report contains the following sections:

- Section 2, Background: An overview of the challenges that satellite development groups face.
- Section 3, Attitude Determination And Control Simulation: Montana State University's satellite development leading to the creation of VideSupra.
- Section 4, PhySim2: A look at the core libraries for running simulations.
- Section 5, VideSupra: An explanation of the built-in data types and how to extend VideSupra with dynamically loaded libraries.
- Section 6, PhySim2AstroPlugin: A detailed breakdown of the Astrophysics plug-in, what is provided and how it works together to create satellite simulation scenarios.
- Section 7, Running an Astrophysics Simulation: An instruction manual for setting up, running and reviewing a typical astrophysics simulation.
- Section 8, Conclusions: A review of the goals of VideSupra and a discussion of future development options.
- Section 9, Resources: The list of books, papers, websites, and software used during the development of VideSupra.

BACKGROUND

As wireless technology and global connection become more prevalent, more man-made satellites will be needed. Not just for the global network and triangulation systems, but more research satellites and garbage collection satellites must also be flown. Currently only government agencies, commercial groups, and educational institutions can afford to create satellites. The cost for a satellite project is split four ways between the development, testing, launch, and data collection. For research satellites data collection is the cheapest section.

The development process costs so much because of the man hours required to design and assemble space resilient hardware and software. The satellite must endure severe fluctuations in temperature, extreme vibrations, and harsh radiation. The satellite also must run without any physical contact for the entire duration of the mission. If the system doesn't turn on or won't listen to the commands being transmitted to it, the mission is a failure. Even just transmitting a signal to it is complicated. If the satellite has an equatorial orbit, the satellite would be travelling at around 7km / sec. and it would orbit the Earth about every 90 minutes. The development team would have to design the satellite and ground station to handle the Doppler shift and constant loss of signal. Even if the antenna could broadcast from horizon to horizon the air density near the edges would distort the signal leaving a 60 degree window straight above the antenna for communication. The cheaper systems don't have a network of antennas and can't afford to pay for time on an existing network. They only have the connection at their mission command center. Therefore the satellite can only be reached for 10 minutes every 90 minutes when the orbit syncs up with the altitude and latitude of the command center. The communication scheme must be resilient towards a bad connection without having too much redundancy checking and handshaking so that as much information as possible can be transferred in the limited available time.

On top of all that the design of the satellite must handle the high velocity oxygen ions which burn holes straight through any conductive material on the satellite and free radicals flipping bits on and off in the electronics. Also there is no air, so the heat generated by the components could cause them to overheat. A thermal conductive circuit has to be added on top of the electrical system in order to keep the parts from cooking or freezing. Newer hardware can either use less power and therefore produces less heat or provide more processing speed using more power and producing more heat. Newer hardware also use smaller transistors which are more vulnerable to the ions and electrons. Typically space tested and proven hardware has to be used. Unfortunately they are older and provide less computation power therefore the software must be very lightweight but still affective.

If that wasn't enough the satellite will be violently shaken by the launch vehicle before the mission even begins. Structural teams need to design the satellite to be lightweight and very sturdy. Obviously the design and development of a new satellite cannot be simplified, otherwise the mission is a failure before it even begins.

The launch of the satellite is expensive because a small group will not have a rocket powerful enough to launch even a small satellite into space, nor the launch pad and clearance. The cheapest method to date is to buy a flight from Kazakhstan on a decommissioned warhead launch vehicle. If the group is American then they will have to get the US government to help transport the satellite to Kazakhstan since satellites are considered munitions when they are crossing international borders. NASA will launch satellites but for the smaller groups NASA is too expensive or to difficult to get a flight with. There are some companies which are willing to launch a satellite for free if the group enters into a development competition and wins. Typically this means that the satellite has to provide a desired functionality for the use of the company sponsoring the competition. Either way the satellite will only be given a specific orbital inclination for a steep price. The cheaper the fight the more likely a failure to launch will occur.

This wastes the cost of the building materials and assembly costs. There is insurance that can be provided for such a failure. The insurance is ideal for large expensive satellites but usually costs more than the smaller satellites cost themselves. The price is dependent on weight and size so small satellites do get an advantage there.

That leaves testing. Satellites can be tested in several ways and should be tested in as many as possible. Launching an incompetent satellite will cost much more than spending a little more in testing and a little more in development. There are several physical tests which can be performed. A prototype of the satellite can be attached to a high altitude balloon to test communication and some sensors. The high altitude balloon has thermal problems because the air is very cold but not thin enough to have the components or the sun provide the same thermal conditions as in a flight. A cheap and temporary solution is to include a heater in the test to keep the components warm. There are drop zones, used by NASA, which will provide about a minute of free fall down a dark mine shaft or a drop from a high altitude air craft. Both are expensive, provide a short test, the satellite must be small, and the air provides enough drag that the terminal velocity is reached. The satellite should be placed in a smooth sphere so that it can tumble freely without the air catching corners. Even then the moment of inertial isn't enough to overcome the roll of the sphere in the air to provide a good clean test. The small group can also buy or rent a "shaker and baker" to test the satellite under extreme temperatures and violent vibrations. The group could also buy or build a vacuum chamber to test out gassing and thermal. The better the chamber the more expensive. Most vacuum chambers will not be able to even get close to the vacuum of space, but some of the nicer ones will be able to change the atmosphere to Nitrogen or Helium rich, or change the temperature of the remaining atmosphere.

The nonphysical tests are performed by removing the sensors and replacing them with a hardware-in-the-loop system (HIL). This setup convinces the satellite that it is in space and the satellite can then be tested realistically before spending money on rigorous physical tests.

Depending on how realistic the HIL is depends on how much time and man power is put into developing it. Therefore the better the HIL is typically the more expensive it is. A HIL should simulate Doppler shifts, thermal reading, magnetic readings, GPS, lunar gravitational pull, etc., as needed to test the satellite. Doppler shifts and GPS can be hard to simulate unless a large chunk of the hardware below the antenna is removed and only the result from the hardware is sent in via serial. Since these nonphysical tests can be done in house with a "flat sat" (a prototype satellite with the hardware just laid out flat across a table) they can be done multiple times without increased cost, other than paying the group members who are running the tests. The tests are repeatable with lots of detailed results. Once the satellite matures it can be run on an alpha prototype, the full satellite put together but with sensors still replaced by a HIL via a port. Once the alpha passes the HIL tests it should be put into beta and start physical testing. Some groups will even leave in the HIL port on the final release so that on the way to the launch site the satellite can run a simple diagnostics to make sure all parts are functioning correctly. These diagnostics are simply a short HIL test. For the HIL to work it must be a physics simulator which can predict, using the satellites output, the feedback response the satellite would receive from the environment.

ATTITUDE DETERMINATION AND CONTROL SIMULATION

VideSupra is based off of several software packages. The software packages were originally designed as a HIL and physics simulators for the Maia nanosatellite project at Montana State University (MSU). The Maia project was designed to characterize near earth variations in the energetic charged particle of the magnetosphere with a new solid state particle detector. The ADC (Attitude Determination and Control) team for the Maia project needed to determine the desired orbital paths and satellite rotation. The whole project had very little budget for development so software solutions were desirable for testing in the preliminary stages. The ADC team needed to check that their algorithms could correctly determine the satellite's attitude and determine a correct solution for controlling the rotation. The ADC system had a minimal power budget and a slow processor. The tests that were created would have to be repeatable so that once a solution was designed it could be optimized to run in real-time on the hardware. The team selected one member, Grant Nelson, to design and build a simulator that would feed in sensor information and simulate the resulting response to the control output from the satellite while the rest developed the embedded software for the satellite.

The software packages evolved over three years as the Maia project was in development. The Maia mission lost several of its lead developers and the mission got put on long term hold. The parts, personal, and development was shifted to a similar project called SpaceBuoy. Through the transition the requirements changed and the specialized software packages were deprecated. After the shift the deprecated software packages were redesigned as a single application that would to work on all of MSU's satellite projects. This new application was named VideSupra.

Requirements

Even though VideSupra is flexible enough to work for almost any simulation and HIL, its main purpose was to test SpaceBuoy and, although no longer being worked on, still perform the Maia tests. The requirement for VideSupra was that it must be able to simulate most, if not all, of the satellites' requirements and environment for the individual missions.

Maia's orbital path and rotation had to maximize the payload sensor's data. The sensor was mounted on the top of the satellite and the top and sides were fitted with solar panels. The solar panels had to face the sun as much of the time as possible without affecting the collection. The satellite could also be sent into a collection mode where the sensor requirement could be ignored so that more power could be gained by the solar panels. That mode was only initiated if the battery power became critically low.

SpaceBuoy's orbital path and rotation has to maximize the payload sensors' data as well. These sensor's are positioned on two of the sides of the satellite along with the solar panels. To maximize the amount of data collected the sensors' field of view has to sweep through the ram vector whilst also keeping the solar panels facing the sun. The bottom of the satellite, with thermal reflector, must never face the sun or the bottom of satellite would overheat causing damage to the circuitry and batteries. The top of the satellite has auxiliary antennas for high bandwidth and ham radio. The different requirements were weighted such that the system could determine which was the most important at a given moment. SpaceBouy could be put into the same solar collection mode as Maia by changing the weightings such that the solar gain was greater than the sensor's requirement weighting. This could also be done so that the antenna weighting could be raised to put SpaceBouy into a higher quality transmission mode.

The full requirements are much more exhaustively described in the mission

documentation however the full requirements are not needed for the scope of this paper. The mission documentation is available under the NASA-AFRL space scholars program agreement. Contact MSU-SSEL for more information.

Solutions

The ADC team determined that the solution for the Maia mission was to get the satellite to rotate in a ram pointing motion (see Figure 10, page 29). This motion would best fulfill the sensor and solar panel requirements. The team determined the easiest method for this control would be three perpendicular magnetorquers. These magnetorquers, or torque coils, could cause rotational acceleration when turned on perpendicular to the earth's magnetic field. This would provide the means of rotation when they were turned on at the proper time. To determine the proper coil and strength required a magnetic sensor. The sensors could only be used when the coils aren't causing a magnetic field of their own.

For SpaceBouy the team determined that a cartwheel motion (see Figure 12) would work best. They also determined that along with the magnetic sensors, to get proper rotational speed and determination of relation to earth, a blackbody sensor should be added. The system would still use the torque coils for movement.

This meant that the simulator had to determine the Earth's magnetic field that the satellite would experience at a specific time and location. The location had to be calculated given the flight path and time. The desired attitude would have to be created and the simulated attitude would have to be calculated given the magnetic force of the coils and the Earth's magnetic field. The coils could be calculated given the configuration and applied voltage. The blackbody sensor could be simulated with the position of the Sun and Earth. Since the Sun's relative location would be found the amount of solar gain for the satellite could also be calculated.

PHYSIM2

The solution for the ADC team is a dynamic physic simulator which is flexible enough to handle a variety of configurations. To handle this a library was created called PhySim, short for physics simulator. PhySim was a set of software tools which another program could use. The PhySim could run a set of controls which would create the desired results for the simulation, the problem was that the software interface was very complicated to use and setup. PhySim2 was designed to handle the setup for the programmer and simplify the configuration of the simulation.

PhySim2 is a model view controller design. The base simulator does not know what it is calculating nor what the results will be, otherwise flexibility would be lost. The simulator just has to run a set of controls, but before it can run them it will determine the order in which the controls can be called. This determination stage (the main difference between PhySim and PhySim2) provided the organization and flexibility sought for. A group of controls were written to calculate the location of the Earth relative to the Sun, the rotation of the satellite, the magnetic field, and so on. When these controls are added to PhySim2 a simulation would be run that could be used in the HIL for a satellite or just to investigate the configuration of the satellite. One set of controls would simulate the Maia mission and another set would simulate SpaceBouy. Most of the controls were the same between these two satellites. PhySim2 is flexible enough that controls to describe a table with pockets, a pool cue, billiard balls, and collision detection could be added to simulate a round in a game of pool.

PhySim2 has a generator class which is used to create the simulator. The controls can be added and removed freely from the generator. When the configuration is ready the generator is told to create the simulation. The generator performs checks to make sure that all the proper controls are received. These checks include initializing the controls to let them know that they are

about to be used and to give them the opportunity to request all the variables that they'll need. The requests can be made in any order and the generator can call the controls in any order.

Variables

The data for the simulation used by each control has to be typed and have correct units. Access to the data had to be restricted to maintain data correctness. The hardest part is that the data is unknown to the simulator but the controls had to be able to look up data and write data quickly. The views of the data had to be able to look up data but not be able to write.

During the generation of the simulator the controls get a chance to request variables that they will need. Not all variables used by a control need to be declared, only the ones that are accessible from outside the control. The requests for variables are returned to the control as the variable which they will use when running, however they cannot use it at this time. The variable uses a proxy pattern to protect from data being set before writing data is allowed. Figure 1 shows the design for this.

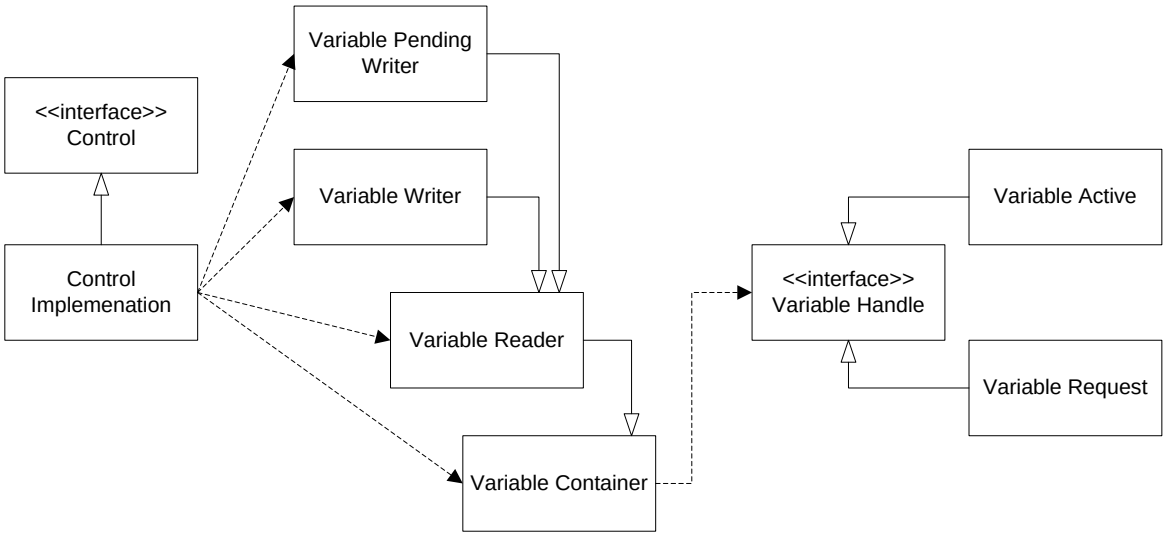


Figure 1: Variable Design for PhySim

The variables can be requested to either write normally, suspend the write until the end of the simulation cycle, or write a constant variable which then cannot be changed. The variables are multiple read, single write (MRSW) so that only one control is allowed to write to a variable to keep conflicts from occurring. The variable writer provides access to modify the value and the guarantee that the variable will not be read from in that simulation cycle until after the value is set. The pending writer provides access to modify the value at the end of the cycle so that the read value is from the previous cycle. The controls can read a pending variable at any time since the value during a cycle is the value that was set at the end of the previous cycle. When a control writes to the pending variable the new value is not actually set until the end of the cycle regardless of when it was written. If a constant variable is requested a reader variable is returned. The value is set during the activation of the variables and never allowed to be changed again. The variable containers just redirect calls to the variable request class or the variable active class. During the generation of the simulation the variables are all just request classes, but once the simulation is finished being created the last step is to replace all the requests with active variables.

Since the container variable is just a proxy for the current variable, either the request or active variable, many containers can be created for the same variable. Even with many containers for the same variable the memory overhead is low and no value propagation is needed to set the value to many controls. The containers always return the newest value, except in the case of a pending writer where the newest value is not set until the end of the cycle. Also by using multiple containers all the controls which asked for just a specific type get just that specific type. Even though the writer extends the reader, just the reader is given to those controls which request it. That way the control cannot cast up to another type of variable but they can cast down towards the container. The container is the lowest form of variable. The reader returns a typed value where as the container only returns a non-typed value. This makes reading values easy for views which don’t know or don’t care about the type of the value.

The request variables keep track of which controls asked to read from them and which controls asked to write to them. They allow default values to be set and attributes to be modified. Both the request variables and active variables keep the name, type and units, but only the active variables keeps the value of the variable. The use of the request variable provides the generator the ability to initialize the controls in any order. Once the controls are all initialized the list of requests are used to determine the dependency order.

The Variable Pool

Since the simulation can be customized, the program needed a way to keep the large number of active variables without knowing what they are. Some simulations could run much faster than real-time so the simulation should keep the data for review and playback. This also proves very useful when the simulation runs slower than real-time. The results can be collected for a long time, then the simulation can be reviewed at normal speed. To solve this problem the variable pool was designed for PhySim.

Type	Name	Value	Units
System.Types.DateTime	Previous Time	12/31/2007 23:59:59.900	UTC
System.Types.DateTime	Current Time	1/ 1/2008 0:00:00.000	UTC
System.Types.DateTime	Next Time	1/ 1/2008 0:00:00.099	UTC
System.Double	Previous Julian Day	2454466.49999884	JD
System.Double	Current Julian Day	2454466.5	JD
System.Double	Next Julian Day	2454466.50000116	JD
System.Double	Delta Time	0.1	secs
System.Types._3D.Matrix4x4	Convert ECIR To HE	[[1.0000000, 0.000000000000, 0.000000000000, 0.0] [0.0000000, 0.917482062069, 0.397777155932, 0.0] [0.0000000, -0.397777155932, 0.917482062069, 0.0] [30,824,178.5555746, -146,387,949.027145000000, 0.000000000000, 1.0]]	km
System.Types._3D.Matrix4x4	Convert HE To ECIR	[[1.0000000, 0.000000000000, 0.000000000000, 0.0] [0.0000000, 0.917482062069, -0.397777155932, 0.0] [0.0000000, 0.397777155932, 0.917482062069, 0.0] [-30,824,178.5555746, 134,308,317.335503000000, -58,229,782.026723800000, 1.0]]	km
System.Types._3D.VecPnt	Sun's Location In HE	[0.0, 0.0, 0.0, 1.0]	km
System.Types._3D.VecPnt	Sun's Location In ECIR	[-30,824,178.5555746, 134,308,317.335503, -58,229,782.0267238, 1.0]	km
System.Types._3D.VecPnt	Earth's Location In ECIR	[0.0, 0.0, 0.0, 1.0]	km
System.Types._3D.VecPnt	Earth's Location In HE	[30,824,178.5555746, -146,387,949.027145, 0.0, 1.0]	km
System.String	Sun Coordinate System	HE	km
System.Double	Sun Radius	695000	km
System.Types._3D.Matrix4x4	Convert ECIR To ECEF	[[0.211253164627, 0.977431378888, 0.0, 0.0] [-0.977431378888, 0.211253164627, 0.0, 0.0] [0.000000000000, 0.000000000000, 1.0, 0.0] [0.000000000000, 0.000000000000, 0.0, 1.0]]	km
System.Types._3D.Matrix4x4	Convert ECEF To ECIR	[[0.211253164627, -0.977431378888, 0.0, 0.0] [0.977431378888, 0.211253164627, 0.0, 0.0] [0.000000000000, 0.000000000000, 1.0, 0.0] [0.000000000000, 0.000000000000, 0.0, 1.0]]	km
System.Types._3D.Matrix4x4	Convert ECEF To HE	[[0.211253164627, -0.896775757033, -0.388799874013, 0.0] [0.977431378888, 0.193820989100, 0.084031683007, 0.0] [0.000000000000, -0.397777155932, 0.917482062069, 0.0] [30,824,178.555574600000, -146,387,949.027145000000, 0.000000000000, 1.0]]	km

Figure 2: Variable Pool Example

When a control or view sets a variable from the variable pool, a variable container or reader is returned. To get a variable the control must provide the name, the units, and the type. This helps diminish unit mismatch problems. Too many errors occur from not having the correct units. Once a control has gets a variable class it can use the variable without ever having to look the variable up again.

When the simulation is being created the controls must declare the variables that it intends to use. The variable pool is then created with those variables in it. The variable pool may not have variables added nor removed for the rest of the simulation. At the end of each cycle of the simulator the variable pool will have the new value for that cycle.

At the end of each cycle the values for the changed variables are copied and put into a frame that is added to the variable lake. The variable lake is a collection of changes to a pool over time. The lake can be used to replay the simulation as well as fast forward, rewind, or playback at faster or slower than real-time. To playback the lake, a variable pool is gotten from it, then as requests for different times is sent in to the lake, that pool will be properly updated with the values for the nearest time to that request. Figure 2 shows the viewer, provided with VideSupra, of the variable pool. The viewer will show the variable pool as a lake is played back.

Determining Control Dependencies

Once all the variable requests have been collected the simulation generator must activate them and determine the dependencies before the simulation is finished being built. The simulation generator determines the dependencies of the controls using the requested variables while activating them. When a variable is activated all the variable requests for the same variable are connected and activated with it. The following rules are run to determine the dependencies while activating the variables.

1. All variables are checked to have one and only one writer. All the variable requests are checked against that writer to make sure that they have the correct type and units.
2. All the constants and pending variables are activated.
3. Some controls won't have any read requests because none were made for that control or because all of the ones which were made have been activated. For all the controls which don't have any read requests, it puts them into the current level of dependency and activates all of its write requests, which activates all of the read requests that match the write requests.
4. Increase the dependency level. Repeat step 3 until no controls were added during that step.
5. If all the controls have been put into dependency levels then the simulation has been activated. If any controls are not in a dependency level then those controls have a dependency loop. A dependency loop is when a control writes a variable another needs, while at the same time the other control writes what the first one needs. Neither can be run first because they depend on the each other. Dependency loops can be fixed by making one of the controls request a pending variable instead of a write variable.

VIDESUPRA

VideSupra, called VideSupra2 in the code after the update from PhySim to PhySim2. VideSupra combines PhySim with a GUI and plug-in loader to provide a setup tool which beginning users will be able to use while still providing the full functionality of PhySim. The plug-ins provide a way to quickly develop and extend VideSupra.

Physics and Mathematics

The standard units are radians, kilometers, kilograms, seconds, nanoteslas and watts. The time zone used is Coordinated Universal Time (UTC) as defined in ISO 8601. However, to simplify some equations and to meet requirements, other units may be used; degrees, meters, days, or minutes. Even though the variables help enforce units, it may not solve all unit mismatch problems. Always keep a close watch when dealing with units. The user interface may use an alternative time zone for convenience. No parts may use, nor should use, United States customary system, as default units.

The simulator uses IEEE 745, a standard for binary floating-point arithmetic, 64 bit double-precision storage for fractional values and 32 bit integer storage using twos-complement for whole numbers. As with the units, for some parts of the simulator other storage types are used.

VideSupra is written entirely in C#, therefore it is compatible with Windows XP with the .Net 2 Framework or greater, Vista, Windows 7, or above. The current libraries are compiled so that they can be included into, but not limited to, C, C++, C#, J# and all CLR based languages. VideSupra will load any CLR based language plug-in, so new libraries do not have to be written in C#.

Types

The types group is made up of seven main classes with some helper classes. Each main class is a storage type for holding some type of data. This data should be useful in creating a functional simulator. Using these types, doubles, strings, and integers will make future controls and maintenance easier to do than each control defining unique types. As always common features make a system more compatible with future designs.

DateTime and TimeZone

The DateTime and TimeZone classes are designed to accurately handle time. Both are designed to keep the values accurate and only use one double to store the value. They use algorithms to calculate the equivalent seconds, minutes, hours, days, etc..

The TimeZone class stores a name and an hour offset. It also includes the 76 main time zones predefined. There are two methods to note for the TimeZone. They are ToUTC and FromUTC. Both take in a DateTime and return a DateTime. The methods change the given time from UTC to the TimeZone whose method was called or vice versa. This is simply done by adding or subtracting the hour offset from the given DateTime. As stated this is fairly straight forward.

The DateTime is slightly more complex. DateTime stores the ticks (10,000.0 ticks per millisecond) since the date and time $1/1/0_{AD}$ 00:00:00.000. The time can contain fractions of ticks for very precise simulations, such as light diffraction simulations. The date can be negative to represent BC dates. The precision for the date and time is 52 bits with an 11 bit mantissa, as defined by IEEE 754, which means that as the value goes up the precision becomes less. For most applications this is great because, if the simulation is in centuries typically the steps will only be at smallest days, and the precision would be in plus minus hours, whereas if the

simulation is in days typically the steps will only be in seconds, and the precision is in ticks. The DateTime can even measure in milliseconds with a precision much smaller than picoseconds. High precision simulations should ignore the day, month, and year. Set them to $1/1/0_{AD}$ for best results.

The DateTime contains methods for getting difference and sum between other dates. It also contains conversion methods from System.DateTime and methods for getting days, months, years, hours, minutes, seconds, and so on. It handles the Julian and Gregorian reform. It also converts to and from Julian days.

Mass

The Mass class is designed to handle a complex rigid body of mass. Mass has three main fields, the first is the total mass in any unit, the second is the center of mass, and the last is moment of inertia tensor matrix and its inverse. The moment of inertia tensor is a Matrix3x3 designed to handle only rotational inertia. Linear momentum can be calculated with the total mass. The center point is a VecPnt. The Mass class contains methods for adding other masses to it, translating the mass, and rotating relative to its own coordinate system.

The Mass can be set directly or it can be calculated using a list of presets including a spherical mass, cylindrical mass, cube mass, cone mass, etc. The original mass design has been tested and proven in the open source ODE (Open Dynamics Engine)[9]. The design has been changed to better suit the PhySim, and rewritten in C#.

Matrix3x3, Matrix4x4, and VecPnt

The matrix classes, Matrix3x3 and Matrix4x4, are designed to handle fast matrix rotations, scales, and coordinate setup. The 4x4 matrices are 3x3 minors padded with a vector in the fourth row. The fourth column is all zeros save the lower right, which is typically a one. The 4x4 matrices use the fourth row and column to perform translations, therefore some of the math

operations for the 4x4 matrices leave out the fourth row and column to maintain consistency. The fourth row and column are also used to handle orthogonal and perspective camera projections for 3D graphics or shadow casting algorithms.

The VecPnt is a vector or point class. The first three values represent the X, Y, and Z. They are used for all vector mathematics (length, cross product, dot product, etc.) and act as typical 3x1 matrices. The fourth value, W, is left out of most of the vector mathematics. The W is designed to determine if the VecPnt is a vector or a point by being set a 0 or a 1, respectively. Any value other than 0 or 1 will cause the VecPnt to scale its own translation. The VecPnt is designed to work with matrix classes. For 3x3 matrices the W value is ignored while being translated. For the 4x4 matrices the matrix multiplication is performed fully to cause the VecPnt to translate properly, whether the VecPnt is a vector or a point. Equation 1 and 2 show how this multiplication performs for a vector and a point. Note: The transpose on the VecPnts are just to make it easier to read. The matrices and VecPnt is stored in doubles to represent the formats shown in Figure 3.

$$\begin{pmatrix} X \\ Y \\ Z \\ 0 \end{pmatrix}^T \begin{pmatrix} R_{00} & R_{01} & R_{02} & 0 \\ R_{10} & R_{11} & R_{12} & 0 \\ R_{20} & R_{21} & R_{22} & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix} = \begin{pmatrix} R_{00}X + R_{10}Y + R_{20}Z \\ R_{01}X + R_{11}Y + R_{21}Z \\ R_{02}X + R_{12}Y + R_{22}Z \\ 0 \end{pmatrix}^T$$

Equation 1: Vector Translation

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}^T \begin{pmatrix} R_{00} & R_{01} & R_{02} & 0 \\ R_{10} & R_{11} & R_{12} & 0 \\ R_{20} & R_{21} & R_{22} & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix} = \begin{pmatrix} R_{00}X + R_{10}Y + R_{20}Z + T_x \\ R_{01}X + R_{11}Y + R_{21}Z + T_y \\ R_{02}X + R_{12}Y + R_{22}Z + T_z \\ 1 \end{pmatrix}^T$$

Equation 2: Point Translation

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} \\ M_{10} & M_{11} & M_{12} \\ M_{20} & M_{21} & M_{22} \end{bmatrix} \begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ M_{30} & M_{31} & M_{32} & M_{33} \end{bmatrix} \equiv \begin{bmatrix} R_{00} & R_{01} & R_{02} & T'_x \\ R_{10} & R_{11} & R_{12} & T'_y \\ R_{20} & R_{21} & R_{22} & T'_z \\ T_x & T_y & T_z & 1 \end{bmatrix} \quad \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}^T \qquad \begin{bmatrix} T \\ x \\ y \\ z \end{bmatrix}$$

Matrix3x3 Matrix4x4 (coordinate conversion matrix) VecPnt Quaternion

Figure 3: Storage Formats for Select PhySim Types.

Quaternion

In some of the more delicate physics calculations a quaternion is used in place of a 3x3 matrix. The Quaternion class handles quaternion mathematics for three dimensional rotations without the fear of gimbal lock. Gimbal lock occurs while using matrices. If any rotation is very close to or at 90° an entire axis is lost so that all control falls to the remaining plane. Quaternions are designed to handle the physics of orientation, rotational velocity, and angular acceleration. A quaternion used four doubles to store its data, as shown in Figure 3. The first value, 'T', is represents the real part of the quaternion where as 'x', 'y', and 'z' represent the fourth dimensional imaginary number parts. Therefore the quaternion as a single number would look like, $Q = T + xi + yj + zk$, where 'i', 'j', and 'k' are the imaginary dimensions. Quaternions fix problems that otherwise wouldn't be solved, and they simplify several algorithms.

Since they aren't as intuitive as matrices, quaternions are rarely used in VideSupra. Where they are used, they are the best method or only method for solving that particular algorithm. One such place is the rigid body dynamics in the control SatelliteRBD found in the library PhySimAstronomicalControls. In the billiard table example each ball would require the use of a quaternion to calculate its location given the friction, momentum, and rotational velocity. To get a better understanding of quaternions read "Quaternions and Rotational Sequences"[5].

Plug-ins and Pages

VideSupra is a plug and play system so that it can be extended to handle new simulations, controls, and views. When VideSupra starts up it will search the directory it was started from and all subdirectories of that directory for plug-ins. Plug-ins are loaded as assemblies from COM dynamic-link library (dll) files. The assemblies are checked for specific class implementations then use the C# activator to create an instance from that assembly. When a plug-ins is created it is

wrapped in a shell class that makes all the calls to and from that plug-in through exception handlers so that unknown plug-ins will not crash the whole project. During development these shells can be turned off so that the development environment can properly navigate to the code in the plug-in which throws an exception. Communication between plug-ins should be protected as well by the plug-ins themselves since they will not necessarily know the other plug-in. The other plug-in could be the expected plug-in, a new version or even a different plug-in as long as it implements the correct interface and name. Not necessarily knowing the other plug-ins is a desired feature. It allows for fast development, fast maintenance, and extendibility for supporting new features.

Currently VideSupra supports only three plug-in types through the VideSupra2.PluginCommons library. The PluginCommons require the use of some other libraries such as PhySim2 and Types. There are four interfaces and a class in PluginCommons. One of the interfaces is not a plug-in interface. It is the MainControlInterface. This interface is implemented by the main control for VideSupra. This interface provides a way for plug-ins to call back to the main control. The main control provides methods to add and remove tool strips and pages from the main form. It also has methods to post for or show a popup for errors, warnings, notices, and comments.

The only class in the PluginCommons is called CollectionResult. This class is used by the simulation during a collection to mark a pool returned from the simulation. This class can be extended to carry other frame information as needed by the scenario and is used when placing a pool in the lake. Typically this won't need to be extended. Extra information can be put into the pool instead. A scenario plug-in will require the use of this class.

The Scenario interface allows new scenarios and scenario formats to be created. VideSupra already implements an abstract scenario designed to run real-time and collection scenarios, as seen in Code Snippet 1.

```
   VideSupra2.CommonTools.Scenarios
   {
      /// <summary>
      /// This is an abstract scenario for collection and real-time scenarios.
      /// </summary>
1     abstract public class CollectionRealTimeScenario:
2        VideSupra2.PluginCommons.Scenario
      {
         ...
      }
   }
```

Code Snippet 1: CollectionRealTimeScenario
(line 28 in Libraries\CommonTools\Scenarios\CollectionRealTimeScenario.cs)

As with the other plug-ins, extending, inheriting and implementing the scenario can be done to make a common class for other scenarios. The plug-in loader will not load these because of two things. First it will not load abstracts, statics or interfaces because they cannot be constructed. Similarly, scenarios with private or protected constructors or those which don't have the public default constructor, a constructor with no parameters. Secondly it will not load the scenario implementation just because it inherits the scenario. The class must have a special attribute tag on it too. Only when the attribute is on a class which inherits the plug-in interface will it be loaded. Code Snippet 2 shows the scenario plug-in for the astrophysics package.

```
   /// <summary>
   /// This is the main scenario plug-in for the Astronomical Simulator.
   /// </summary>
1  [VideSupra2.PluginCommons.ScenarioAttribute]
2  public class Scenario:
      VideSupra2.CommonTools.Scenarios.CollectionRealTimeScenario
   {
      ...
3     public Scenario()
      {
         ...
      }
      ...
   }
```

Code Snippet 2: Astrophysics Scenario
(line 27 in Plugins\PhySim2AstroPlugin\Scenario.cs)

The second plug-in interface is for a scenario object called ScenarioObject. Scenario objects are used to build a tree representing a specific setup to collect data for. The root scenario

may not be removed from the tree. The root is provided by the scenario. The objects can be added as needed to the tree but they can also accept or reject being used in some tree formations. For example, the astrophysics scenario provides a root object called the universe. The universe can have the Sun, the Earth, or a satellite added to it but not all three. If the Sun has been added to the universe, the universe will keep an Earth, a satellite or another Sun object from being added. The Sun will only accept an Earth object yet only one can be added. The Earth will accept any number of satellites. When the tree is finished being built the objects define which controls are added to the simulator and set them with some values. The object is how controls get added, the controls themselves can be in the same library as the object or in any other. It doesn't matter where the controls are as long as the plug-in object can create them.

The scenario object is presented to the user in a property grid (System.Windows.Forms. PropertyGrid). This means that class attributes are directly visible, unless otherwise marked, to the user. No extra user interface code is needed unless the type of the value being displayed is unique. All of the built in C# types and all the types in the VideSupra's Types library already have user interface controls for the property grid created for them. Code Snippet 3 shows how to customize the user interface for a double value for an object. The attribute tags aren't required but it will provide more information to the user when they are using the property grid. The description is shown with the value. The display name can make complicated identifiers into simple to read text. For example "rAscension" could be renamed so that the user sees "Right Ascension Angle". The category attribute will allow the user to group values in the same category for convenience. For more information about attributes see the MSDN documentation on the property grid attributes.

```
    /// <summary>This gets or sets the longitude, in degees.</summary>
1   [SCM.Category("Properties")]
2   [SCM.DisplayName("Longitude")]
3   [UIT.NoDefaultValue()]
4   [SCM.Description("This is the longitude in degrees.")]
5   public double Longitude
    {
6       get { return Math.ToDeg(this.longitude); }
7       set { this.longitude=Math.ToRad(value); }
    }
```

Code Snippet 3: ObjGroundStation
(line 311 in Plugins\PhySim2AstroPlugin\Objects\ObjGroundStation.cs)

The third and last plug-in interface is the page interface called, PageInterface. The page interface is fairly simple. Each scenario implementation should extend a page that will work properly with it. Both the real-time and collection implementations of the scenario pages were designed for specifically to be used with it, VideSupra2.CommonTools.Scenarios.Collection. CollectionPage and VideSupra2.CommonTools.Scenarios.RealTime.RealTimePage. These extended pages to have new methods added to them that the scenario can use to give a page data, usually in the form of a variable pool. Simple pages which aren't associated with a specific scenario can be added as well. The page interface can put a page icon onto the main menu when necessary and they can add their own menu strips as needed. The scenario may choose to activate and remove certain pages for different conditions. The suggested method is that interfaces or abstract pages are created then the scenario activates or removes all the pages of that type for certain conditions. This will allow new pages to be added to a scenario without the scenario having to be updated or even aware of that page implementation.

For more information on scenarios, scenario objects, and pages read the commenting provided with the interfaces and the commenting on the abstract classes which extend the functionality for a specific purpose. Some of the interfaces are large (14 methods) but they are quick to implement. Most of the methods return a constant value used as an identifier, an image, or a title. The interfaces were designed such that C# knowledge is required (or any other COM development language) but beginner programmers should be able to handle creating plug-ins.

This is needed because although the scenario and the pages are usually written by a programming team, the scenario objects should be able to be written by a physics team.

The following sections discuss in detail the controls for the astrophysics scenario. These controls are added as plug-ins through scenario objects. The objects share the same name. A few of the scenario objects will create multiple controls but for the most part they are one to one. The control goes into the simulation and the object provides methods for setting and creating the controls.

PHYSIM2ASTROPLUGIN

The PhySim2Astro plug-in provides controls, objects, and a scenario for astrophysics simulation specifically to simulate the environment required for developing and testing satellites. The following sections discuss the different parts of the plug-in.

Coordinate Systems Conversions

In order for a simulation, specifically VideSupra, to work right, coordinate systems, as frames of reference, have been designed and built in as the backbone. PhySim2AstroControls defines several conversions between coordinate systems. VideSupra will implicitly and automatically generate the coordinate conversions not defined explicitly by the PhySim2AstroControls package. To prove the correctness of these coordinate conversions, first the definition of the coordinate systems will be presented followed by the mathematical representation and implementation for a specific conversion defined by the PhySim2AstroControls package. Every auto-generated coordinate conversion will not be mathematically derived, however the correctness of the auto-generation process will be scrutinized.

Note that some minor changes from the standard coordinate systems may be present but those changes still maintain the validity of the data in that coordinate system. Some of the changes are to handle the discreet nature of the software and others are for enforcing compatibility. The main practice is to use matrices and vectors using the right hand rule for a three dimensional Cartesian coordinate system. These coordinate systems typically use the XY plane to define the movement path and the Z vector points up.

Heliocentric-Ecliptic

The Heliocentric-Ecliptic Coordinate System (HECS) has the sun in the center and uses the earth as a point of reference (Figure 4). The X vector is pointing at the Vernal Equinox direction, when the days and nights on earth are the same duration and the northern hemisphere is heading into winter, also known as the first day of autumn. Since the Vernal Equinox occurs at noon in spring the vector is extended towards the sun and out the other side, hence it is out the autumn side of the orbit. The X vector points towards the constellation Aries, the ram, therefore the astronomy symbol for Aries will sometimes replace X in different documentation. The X vector is the guiding vector such that when the vectors become homogenous the X vector remains in the same direction. Since the Earth's axis wobbles slightly the Vernal Equinox will vary slightly from year to year. The X vector is associated with an epoch or year in which the Vernal Equinox is based off. The XY plane is defined by the ecliptic in which the earth moves around the sun hence the coordinate system is called the Heliocentric-Ecliptic Coordinate System. The Y vector points towards the northern hemisphere's winter solstice, the shortest day of the year. The planet then rotates counter clockwise around the sun on the Z axis. The length between two points in this coordinate system is in kilometers. However, the standard outside of VideSupra is in

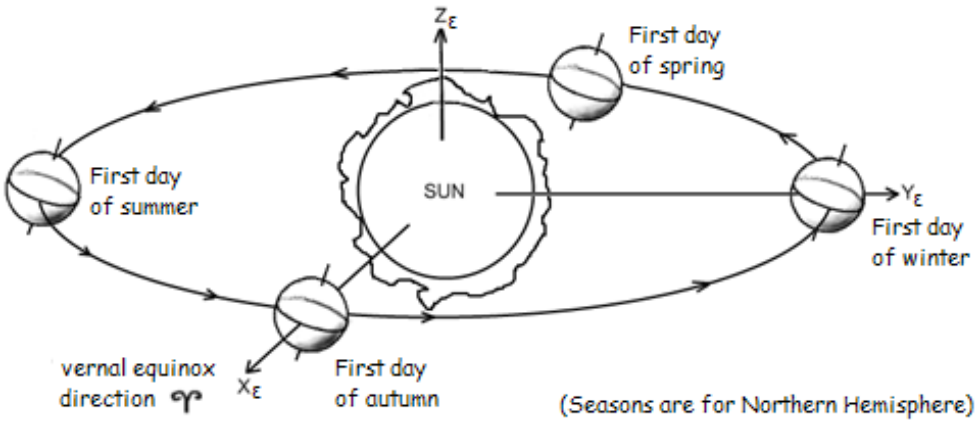


Figure 4: Heliocentric-Ecliptic Coordinate System Diagram

Astronomical Units (AU), the average distance from the sun to earth, 149,598,000 kilometers.

Earth-Centric Inertial Reference

The Earth-Centric Inertial Reference Coordinate System (ECIR) is closely related to the HECS. This coordinate system is also called the Geocentric-Equatorial (GECS) or Earth Centered Coordinate System (ECCS). The X is still pointing in the same direction as the HECS, towards the vernal equinox direction. However, the Z points out of the Earth's Geographic North Pole, forcing the Y to point out the side according to the right hand rule (see Figure 5). This coordinate system is not fixed to the surface of the planet, therefore no matter the hour of day nor day of year the X, Y and Z will always point in the same direction. In fact, this coordinate system stays relative to the stars with the exception of the precession of the equinoxes and parallax of the orbit. The XY plane bisects the planet through the equator, hence the reason this coordinate system is called Geocentric-Equatorial. The I, J, and K vectors are unit vectors, in kilometers, corresponding to the X, Y, and Z axis respectively. This is the most common coordinate system for astrodynamics. It can be used to locate the stars, satellites, and other planets.

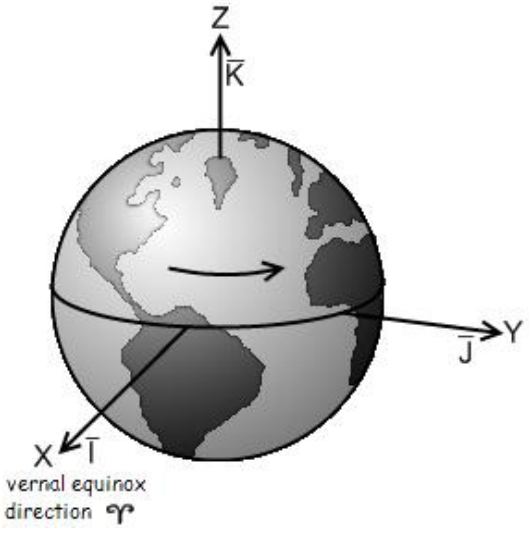


Figure 5: ECIR Coordinate System Diagram

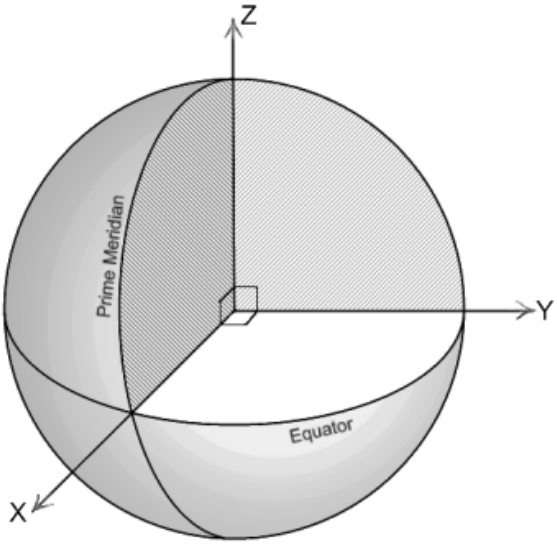


Figure 6: ECEF Coordinate System Diagram

Earth Centered Earth Fixed

The Earth Centered Earth Fixed (ECEF) is also known as the Absolute Coordinate System for Earth (ACSE), shown in Figure 6. The ACS is similar to ECIR except for the XY plane rotates with the planet, such that X is the point where the Equator and Prime Meridian meet (0.0°N, 0.0°E). The Z is still out the Geographic North Pole and Y is at 90° Longitude so that the coordinate system is right handed. The ECEF is in kilometers causing the earth's surface to be roughly 6371.2km from the origin (average sea level, WGS84). It is usually paired with the equivalent ECIR Coordinate System for quick conversion. This coordinate system is designed for locating ground stations and handling geomagnetic data.

Satellite-Centric Inertial Reference

The Satellite-Centric Inertial Reference Coordinate System (SCIR) is identical to the ECIR except that the origin of the coordinate system has been translated to the center of mass for the satellite (see Figure 7). When no torque is applied to satellite which is not currently rotating, the satellite will remain stationary relative to this coordinate system. The drag coefficient, B*, is considered negligible as a torque on most low budget. The Satellite is located with the VideSupra package SpaceTrack.Net.

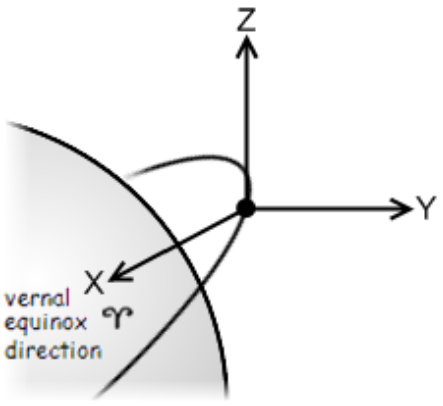


Figure 7: SCIR Coordinate System Diagram

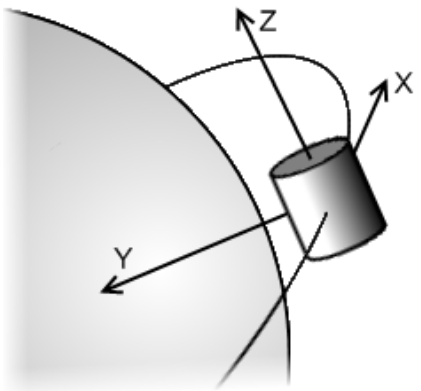


Figure 8: Ram/Nadir Coordinate System Diagram

Ram/Nadir

The Ram/Nadir Coordinate System (RN) is similar to a Perifocal Coordinate System for the satellite. The main difference is that the RN is centered on the satellite and the X and Y axes are defined by the velocity and the direction to the earth (nadir), shown in Figure 8. Unless the elasticity is equal to zero, the velocity vector and nadir vector are not typically perpendicular; the X remains unchanged and the Y is shifted to a homogenous vector. The Z axis is parallel to the Perifocal Coordinate System's Z axis. The Perifocal Z axis is the normal to the orbital plane. The unit for this axis is kilometers. The coordinate system is not fixed to the satellite's orientation. This coordinate system is specialized for an ADCS which must keep a payload fixed relative to the ram direction, as required by Montana State University's – Space Buoy Project.

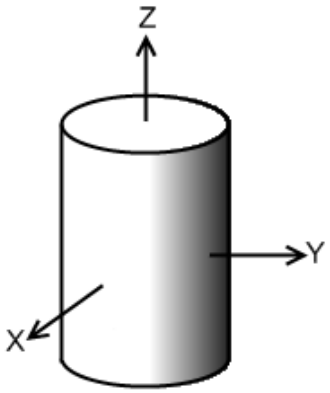


Figure 9: Housing Coordinate System Diagram

Figure 10: Ram pointing

Figure 11: Nadir pointing

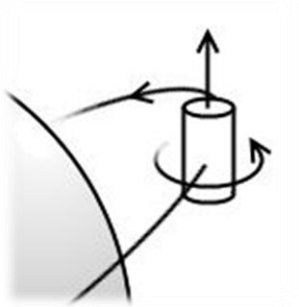


Figure 12: Cartwheel

Housing

The Housing Coordinate System (HCS) is how the ADCS and anything else on the satellite view the universe. This coordinate system is fixed to the housing such that the front (which may be arbitrarily decided) is the X axis, the top (typically opposite side of the light band) is the Z axis and the Y keeps the coordinate system proper to the right hand rule (see Figure 9). The ADCS sensors and actuators are all stationary in the HCS. For many magnetically driven satellites, the X and Y line up with the side magnetotorquers and the Z is opposite the base plane (light band connection) magnetotorquer.

Typically the HCS's Z axis is the axis of rotation for the satellite. The light band can assist with an initial rotation. If the HCS's rotational axis lines up with the RN's Z axis the satellite is in a cartwheel or skid orientation. When the satellite is rotating counter-clockwise around the Z axis, as viewed towards negative Z, then the satellite is in a cartwheel orientation, otherwise it is in a skid. If the HCS's rotational axis aligns with the RN's Y axis then the satellite is nadir pointing, and finally, if the HCS's rotational axis is lined up with the RN's X axis then the satellite is ram pointing.

Earth Fixed Ground Station

The Earth Fixed Ground Station (EFGS) coordinate system is a location on the earth given an altitude, longitude, and latitude. The coordinate system starts with Z directly away from the center of the planet, the Y axis points towards the rotational axis of the Earth, North (towards the Z axis of ECEF), and the X axis points East. The altitude is the distance, in kilometers, from sea level (WGS84 ≈ 6378.15 Km), the latitude and longitude are in degrees, as standard. The coordinate system is then rotated given an elevation and an azimuth. The elevation is the angle, in radians, from the XY plane towards the normal (Z axis) of this coordinate system. The azimuth is the angle, in radians, from the North vector (Y axis) clockwise around the XY plane of this

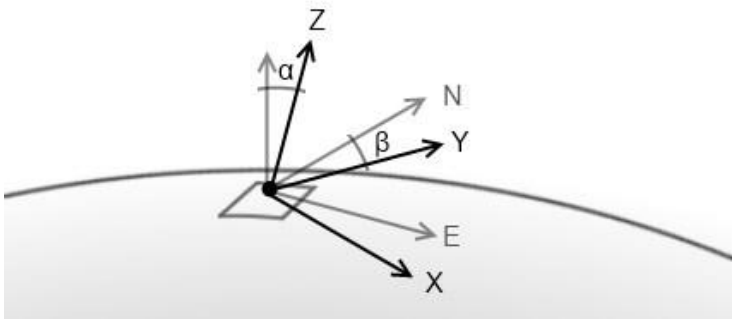


Figure 13: Earth Fixed Ground Station

coordinate system. For most ground stations the elevation and azimuth are zero. The ground station can also be used as a satellite dish location, in which case the elevation and azimuth is set to the tilt of the dish.

Coordinate Conversions

All VideSupra Cartesian coordinate conversions are performed with 4x4 matrices, which are formed from a 3x3 rotational matrix with the fourth dimension being the translation vector. The format for this type of 4x4 matrices is shown in Figure 3. The current version (v1.2) of VideSupra does not accept coordinate conversions to change units; both sides must be in the same units, typically kilometers. This scheme allows for simple vector and point conversions using the VecPnt structure.

ECIR to HE

The coordinate conversion from ECIR to HE requires only the current UTC. This conversion has two tasks. The first is to locate where the Earth is at the given time and the second is to tilt the Earth as shown in Figure 4. Since both HE and ECIR base the X axis off of the Vernal Equinox the X vector must remain unchanged. The resolution required by SSEL for the conversion is low. The Earth's orbit is considered circular and the variance in Vernal Equinox

and Earth's obliquity is ignored. Greater resolution is not required since the only reason SSEL needs the HE is so that the simulation can be used to simulate the SSEL sensors. The sensor's requirements at top resolution is ±1° of accuracy and the Sun will only appear as a 0.5° width object as seen by the sun sensors. For more specifics about the resolution requirements, read about the simulated sensors in the Algorithms Section. The code found in Code Snippet 4 is the method for calculating the ECIR to HE conversion.

$$SunsLoc = [0km \quad 0km \quad 0km \quad 1] \text{ in HE}$$
$$EarthsLoc = [0km \quad 0km \quad 0km \quad 1] \text{ in ECIR}$$

$$\measuredangle yearRot = 2\pi\left(\frac{curTime - UTC(March\ 20, 2000\ 7{:}25{:}0.0)}{365.242199days}\right)$$

$$ECIRtoHE_{4x4} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(-23°26'21.448") & -sin(-23°26'21.448") & 0 \\ 0 & sin(-23°26'21.448") & cos(-23°26'21.448") & 0 \\ cos(yearRot) * 149{,}598{,}000.0km & sin(yearRot) * 149{,}598{,}000.0km & 0km & 1 \end{bmatrix}$$

Equation 3: Symbolic Representation of ECIR to HE

```
   /// <summary>
   /// This calculates the conversion from Earth Centered
   /// Inertial Referenced (ECIR) to Heliocentric Ecliptic (HE).
   /// </summary>
   public override void Calculate(PS.SimulatorArgs args)
   {
      // The Sun's location in HE and Earth's location in ECIR is always zero.
1     this.sunsLoc.Point=VecPnt.Point();
2     this.earthsLoc.Point=VecPnt.Point();
      // Get the days and fraction of days for the Earth since Earth's epoch...
3     double days=(this.curTime.Value-Const.VernalEquinox).TotalDays;
      // Get the rotational angle for the year...
4     double yearRot=Math.PI2*((days/Const.EarthOrbitalPeriod)%1.0);
      // Get Earth's location in HE...
5     VecPnt eLocHE=VecPnt.Point(Const.EarthSemiMajorAxis*Math.Cos(yearRot),
         Const.EarthSemiMajorAxis*Math.Sin(yearRot), 0.0);
      // Rotate by Earth's tilt and offset to location...
6     this.CoordValue=
         Mat4x4.PadMinor(Mat3x3.XRotation(-Const.EarthAxisTilt), eLocHE);
      // Run base calculate methods...
7     this.sunsLoc.Calculate(args);
8     this.earthsLoc.Calculate(args);
9     base.Calculate(args);
   }
```

Code Snippet 4: ECIR to HE
(line 104 in Libraries\PhySim2AstroControls\SolarSystem\ECIRtoHE.cs)

Line 1 in Code Snippet 4 sets the location of the center of the Sun in HE to the origin. The variable sunsLoc is a global point. It is easier to set it to all zeros in this coordinate system and let the auto-generators distribute it to all other coordinate systems, then try to calculate it out of other conversion matrices. This does cause redundant data to be stored but makes the Sun's location easier to get.

Line 2 is the same as line 1 except it sets the center of the Earth in ECIR to the origin. Again this is a global point so it will be propagated to all other coordinate systems. In line 5 the Earth's location is calculated in the HE, but by setting this value to zero in the ECIR the location is slightly more accurate since it will require one less matrix multiplication and inversion before reaching the coordinate systems based off of the ECIR.

Line 3 takes the current time for the simulator then subtracts the Vernal Equinox from it. This will get the change in time since the Earth was directly on the X axis of the HE. This change is given in days and fraction of days.

Line 4 will take the days and fraction of days since the Vernal Equinox and convert it to the angle in radians for the Earth around the Sun. This is done by converting the days and fraction of days into years and fractions of years since the given Vernal Equinox date. The whole part is removed leaving just the fraction of the year since the Vernal Equinox. That fraction is multiplied by 2π to get the angle.

Line 5 uses the angle calculated on the previous line to get the location of the Earth in the HE. Since the orbital plane is only in the X and Y axis, the Z offset for the Earth's location will be defaulted to zero. The X and Y are scaled by the mean distance between the Sun and Earth. The X offset is calculated by taking the cosine of the angle and the Y offset is calculated by taking the sine of the angle. This will cause the Earth to be positive X during the Vernal Equinox the move counter clockwise to positive Y during the Winter Solstice.

Line 6 creates a rotational matrix, tilted clockwise down the X axis, for the Earth's obliquity. The rotational matrix will then be padded by the Earth's location. Remember that this matrix is for converting from ECIR to HE for any point or vector. That means that the frame of reference is the Sun as if it was locating the Earth, as shown in Figure 5. If the X, Y, and Z vectors for the Earth are sent through the ECIR to HE, the X axis would remain the same but the Y and Z would have to tilt clockwise around the X axis as stated above. If the center of the Earth is sent through the ECIR to HE it would have to shift the Earth's location also as state above. The lines 7, 8, and 9 execute the inherited code for the auto-generators.

```
/// <summary>Distance From Sun To Earth = 149,598,000.0 km</summary>
static public readonly double EarthSemiMajorAxis = 149598000.0;

/// <summary>Radius Of Earth = 6,378.15 km</summary>
static public readonly double EarthRadius = 6378.15;

/// <summary>
/// Earth's Obliquity = 23 deg 26 min 21.448 sec = 0.4090928 rad
/// </summary>
static public readonly double EarthAxisTilt = 0.409092804222329;

/// <summary>Days In An Earth Year = 365.242199 days</summary>
static public readonly double EarthOrbitalPeriod = 365.242199;

/// <summary>Earth's Sidereal Days = 0.997269566 days</summary>
static public readonly double SiderealDays = 0.997269566;

/// <summary>
/// Near March 21 (northern hemisphere) when night and
/// day are nearly the same length.
/// </summary>
static public readonly Time VernalEquinox = Time.UTC(2000, 3, 20, 7, 25, 0, 0);
```

Code Snippet 5: Earth Constants
(line 30 in Libraries\PhySim2AstroControls\Tools\Constants.cs)

ECIR to ECEF

Just like the ECIR to HE the ECIR to ECEF coordinate conversion only requires the current simulation time in UTC. As shown in conversion Figures 5 and 6, this conversion is required to take the inertial referenced coordinate system ECIR and rotate it around the Z axis to create the ECEF coordinate system. The first requirement is that on any day at 12:00 UTC the

vector formed from the prime meridian and the orbital plane points towards the Sun given some leniency for Earth obliquity. The other requirement is that the frequency of the rotation around the Z axis is once per sidereal day in a counter-clockwise motion as viewed from the top (positive Z looking towards negative Z) of the Earth. This also means that the Z axis must remain unchanged during the conversion from ECIR to ECEF. The following code also uses the constants shown in Code Snippet 5.

```
    /// <summary>
    /// This calculates the conversion from Earth Centered Inertial Referenced
    /// (ECIR) to Earth Centered Earth Fixed (ECEF).
    /// </summary>
    public override void Calculate(PS.SimulatorArgs args)
    {
       // Get days and fractions of days since earth's epoch...
1      double days=(this.curTime.Value-Const.VernalEquinox).TotalDays;
       // Get vernal equinox day off set in fractions of a day...
2      double offset=Const.VernalEquinox.TotalDays%1.0;
       // Get fraction of sidereal days then the rotational angle for the day...
3      double dayRot=Math.PI2*(((days+offset)/Const.SiderealDays)%1.0);
       // Rotate earth for sidereal day and get the conversion matrix...
4      this.CoordValue=Mat4x4.ZRotation(dayRot);
5      base.Calculate(args);
    }
```

Code Snippet 6: ECIR to ECEF
(line 94 in Libraries\PhySim2AstroControls\SolarSystem\ECIRtoECEF.cs)

Line 1 in Code Snippet 6 calculates the days and fraction of days since the Vernal Equinox. Line 2 gets the fraction of a day since midnight on the day of the Vernal Equinox. This is when the vector formed from the Prime Meridian and the orbital plane points towards the X axis in the ECIR.Line 3 gets the angle of rotation around the North Pole (Z axis) for the day. The offset for the Vernal Equinox is added to the total days of the year then divided by the fraction of a day for a sidereal day. This causes the Earth to make one full rotation every sidereal day causing noon (12:00:00.000 UTC) to be pointing towards the sun each and every day as seen in Figure 14. Line 4 converts the angle into the rotational matrix and line 5 calls the base method causing the auto-generators to run.

$$\measuredangle dayRot = 2\pi\left(\frac{curTime - UTC(March\ 20, 2000\ 0{:}0{:}0.0)}{0.997269566days}\right)$$

$$ECIRtoECEF_{4x4} = \begin{bmatrix} cos(dayRot) & -sin(dayRot) & 0 & 0 \\ sin(dayRot) & cos(dayRot) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0km & 0km & 0km & 1 \end{bmatrix}$$

Equation 4: Symbolic Representation of ECIR to ECEF

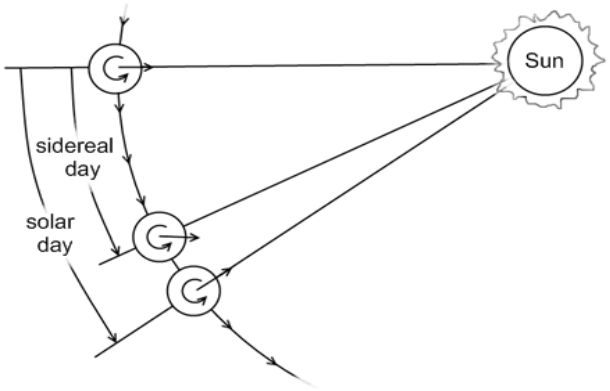


Figure 14: Sidereal and Solar day diagram

ECIR to SCIR

The ECIR to SCIR is simple. It only requires the satellite's location in the ECIR. The ECIR to SCIR translates by the negative of the satellite's location. The satellite's location is calculated using SpaceTrack.Net library and stored as a global point.

```
    /// <summary>This calculates the coordinate conversion matrix.</summary>
    public override void Calculate(PS.SimulatorArgs args)
    {
1       this.CoordValue=Mat4x4.Translation(-this.hndlLoc.Value);
2       base.Calculate(args);
    }
```

Code Snippet 7: ECIR to SCIR
(line 93 in Libraries\PhySim2AstroControls\Satellites\ECIRtoSCIR.cs)

$$ECIRtoSCIR_{4x4} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -SatLoc_x & -SatLoc_y & -SatLoc_z & 1 \end{bmatrix}$$

Equation 5: Symbolic Representation of ECIR to SCIR

SCIR to RN

The SCIR to RN coordinate conversion requires the Satellite's Location in ECIR and the Satellite's Velocity in ECIR. The velocity when normalized is the ram direction and the negative location is the nadir vector. Using these two vectors a coordinate space can be calculated using the following methods.

```
    /// <summary>This calculates the coordinate convertion matrix.</summary>
    public override void Calculate(PS.SimulatorArgs args)
    {
1       Mat3x3 orient=Mat3x3.Transpose(Mat3x3.CoordinateSpaceXY(
           this.hndlVel.Value, -this.hndlLoc.Value));
2       this.CoordValue=Mat4x4.PadMinor(orient);
3       base.Calculate(args);
    }
```

Code Snippet 8: SCIR to RN
(line 94 in Libraries\PhySim2AstroControls\SolarSystem\SCIRtoRN.cs)

Line 1 in Code Snippet 8 gets the coordinate space using the velocity (hndlVel) and the negative of the location (hndlLoc). The coordinate space is a 3x3 matrix. Since all the coordinate conversions have to be 4x4 matrices, line 2 adds on the fourth dimension with a translation of zero. There is no translation from SCIR to RN and the rotation is caused by the coordinate space.

```
    /// <summary>
    /// This transforms from identity coordinate space into a different
    coordinate
    /// space. The x axis remains directionally unchanged.
    /// </summary>
    /// <param name="Xaxis">This is the new x axis.</param>
    /// <param name="Yaxis">This is the new y axis.</param>
    /// <returns>The coordinate space transformation matrix.</returns>
    static public Matrix3x3 CoordinateSpaceXY(VecPnt Xaxis, VecPnt Yaxis)
1   {
2       VecPnt x = VecPnt.Normalize(Xaxis);
3       VecPnt y = VecPnt.Normalize(Yaxis);
4       VecPnt z = VecPnt.Normalize(VecPnt.Cross(x, y));
5       y = VecPnt.Normalize(VecPnt.Cross(z, x));
        return CoordinateSpace(x, y, z);
    }
```

Code Snippet 9: CoordinateSpaceXY
(line 564 in Libraries\System.Types_3D\Matrix3x3.cs)

$$\overline{Vel} = \|SatVel\|$$
$$\overline{Loc} = \|SatLoc\|$$
$$\overline{BiN} = \left\|\overline{Vel} \times \overline{Loc}\right\|$$
$$\overline{Nrm} = \left\|\overline{BiN} \times \overline{Vel}\right\|$$
$$\therefore \overline{Vel} \perp \overline{BiN} \perp \overline{Nrm} \perp \overline{Vel} \text{ and } \left|\overline{Vel}\right| = \left|\overline{BiN}\right| = \left|\overline{Nrm}\right| = 1$$

$$ECIRtoRN_{4x4} = \begin{bmatrix} \overline{Vel}_x & \overline{Vel}_y & \overline{Vel}_z & 0 \\ \overline{Nrm}_x & \overline{Nrm}_y & \overline{Nrm}_z & 0 \\ \overline{BiN}_x & \overline{BiN}_y & \overline{BiN}_z & 0 \\ 0km & 0km & 0km & 1 \end{bmatrix}$$

Equation 6: Symbolic Representation of ECIR to RN

The coordinate space matrix requires three vectors. Code Snippet 9 shows how to use two vectors, specified as the new coordinate space X axis and Y axis, to get all three vectors orthonormal while keeping the X axis unchanged directionally.

```
    /// <summary>
    /// This transforms from identity coordinate space into a scaled skewed
    coordinate /// space. Note: The given axii should be homogeneous orthogonal.
    /// </summary>
    /// <param name="Xaxis">This is the new x axis.</param>
    /// <param name="Yaxis">This is the new y axis.</param>
    /// <param name="Zaxis">This is the new z axis.</param>
    /// <returns>The coordinate space transformation matrix.</returns>
    static public Matrix3x3 CoordinateSpace(VecPnt Xaxis, VecPnt Yaxis, VecPnt
    Zaxis)
1   {
2      Matrix3x3 mat = new Matrix3x3();
3      mat.m[0, 0]=Xaxis.X; mat.m[0, 1]=Xaxis.Y; mat.m[0, 2]=Xaxis.Z;
4      mat.m[1, 0]=Yaxis.X; mat.m[1, 1]=Yaxis.Y; mat.m[1, 2]=Yaxis.Z;
5      mat.m[2, 0]=Zaxis.X; mat.m[2, 1]=Zaxis.Y; mat.m[2, 2]=Zaxis.Z;
       return mat;
    }
```

Code Snippet 10: CoordinateSpace
(line 545 in Libraries\System.Types_3D\Matrix3x3.cs)

Once all three vectors are orthonormal they are entered into the coordinate space matrix as shown in Code Snippet 10. The coordinate space will make the satellite's velocity in RN project directly down the X axis. The earth's location will be mostly on the Y axis with a little on the X axis caused by a non-circular orbit.

To HCS

This conversion can be done by many different methods. The three implemented in the PhySim2Astro-Controls are the identity orientation, standard orientation and RBD orientation. The identity orientation uses the SCIR as the HCS. If the satellite was positioned in the SCIR and no external forces are applied the satellite will stay aligned with the SCIR. The B* drag coefficient and gravity gradient is ignored by SCIR. This is simply a 4x4 identity matrix as a coordinate conversion matrix. The RBD (Rigid Body Dynamics) orientation is discussed under the algorithms section, since it handles much more than just the conversion from SCIR to HCS. The standard orientation provides methods to rotate around a stable axis in the RN coordinate system. This means if rotating counter-clockwise around the Z axis the standard orientation simulates a desired cartwheel orientation. Code Snippet 11 shows how the standard orientations are calculated. It requires the change in time, the rpms (rotations per minutes), and the rotational axis.

```
   /// <summary>This calculates the orientation of this satellite.</summary>
   public override void Calculate(PS.SimulatorArgs args)
   {
1     double freq=this.rpm*Math.PI2/Time.secPerMin;
2     this.rotVel.Vector=this.rotAxis*freq;
3     this.rotation+=(this.hndlDt.Value*freq);
4     this.Orientation=Mat3x3.RotateOnPlane(VecPnt.ZAxis(), this.rotAxis)*
         Mat3x3.ZRotation(this.rotation);
5     this.rotVel.Calculate(args);
6     base.Calculate(args);
   }
```

Code Snippet 11: OrientationRN
(line 195 in Libraries\PhySim2AstroControls\Satellites\OrientationRN.cs)

Line 1 in Code Snippet 11 calculates the radians per second, or frequency of rotation. Line 2 sets a global vector where the length is the frequency and the direction is the rotational axis. The rotational axis was normalized before this method. Line 3 uses delta time and the frequency to determine the angle swept in the last time step. This angle is then added to the

current accumulated rotational angle. Line 4 creates a 3x3 rotation matrix which rotates the first given vector (X axis) to the second given vector (rotational axis) using the cross product and angle between them, as shown in Code Snippet 12. This causes an arbitrary starting rotational angle but typically the housing's rotational angle will have no direct correlation to time, only the change in the rotational angle will correlate to time.

$$freq = rpm\frac{2\pi}{60}$$
$$rotVel = [rotAxis_x * freq \quad rotAxis_y * freq \quad rotAxis_z * freq \quad 0]$$
$$\theta' = \theta_0 + \Delta t * freq$$
$$OrientationRN_{4x4} = \mathcal{R}_{(X \times rotAxis)}\measuredangle(X, rotAxis) * \mathcal{R}_X(\theta')$$

Equation 7: Symbolic Representation of OrientationRN

```
    /// <summary>
    /// This rotates the start axis onto the end axis
    /// on the plane from both axes.
    /// </summary>
    /// <param name="startAxis">This is the axis to start with.</param>
    /// <param name="endAxis">This is the axis to end up on.</param>
    /// <returns>This is the rotational matrix.</returns>
    static public Matrix3x3 RotateOnPlane(VecPnt startAxis, VecPnt endAxis)
1   {
2      if(startAxis==endAxis)
3         return Identity();
4      if(startAxis==-endAxis)
5         return Rotate(VecPnt.ArbitraryPerpendicular(startAxis), Math.PI);
       return Rotate(VecPnt.Cross(startAxis, endAxis),
          VecPnt.Angle(startAxis, endAxis));
    }
```

Code Snippet 12: RotateOnPlane
(line 397 in Libraries\System.Types_3D\Matrix3x3.cs)

Code Snippet 12 computes the rotation on a plane given the start axis and the end axis. If the start axis and end axis are the same (line 1 and 2) then a 3x3 identity matrix is returned. If the start axis is opposite the end axis (line 3 and 4) then an arbitrary perpendicular axis, defined as $[v_y - v_z \quad v_z - v_x \quad v_x - v_y \quad v_w]$ where v is the start axis, is rotated around by π. Otherwise (line 5) the rotation matrix is created by using the cross product as the rotational axis and the angle between the start and end axis as the angle to rotate by. This causes all points along the start axis to be mapped to the end axis. The other points will be mapped similarly except that there is a

rotation around the end axis caused by the rotation around the cross product.

ECEF to EFGS

The ECEF to EFGS creates a point at the correct altitude away from the center of ECEF then rotates it around the center of ECEF so that the point is in the correct location and the axes are rotated correctly. The conversion uses CoordinateSpaceZX which is very similar to CoordinateSpaceXY, see Code Snippet 9. The difference is it uses the Z and X vectors as the basis for the coordinate system instead of X and Y. If Z and X are not perpendicular Z will remain unchanged and X will be orthogonalized. When the location is at either pole where the Y axis and Z axis will be collinear, the Z axis will still point away from the center but the Y axis

```
    /// <summary>This calculates the coordinate converter.</summary>
    public override void Calculate(PS.SimulatorArgs args)
    {
1       double earthSeaLevel = 6378.15; // Earth's Sea Level (WGS84) in Km
2       double d=Math.Abs(this.altitude+earthSeaLevel);
3       double r=Math.Cos(Math.ToRad(this.latitude))*d;
4       double x=Math.Cos(Math.ToRad(this.longitude))*r;
5       double y=Math.Sin(Math.ToRad(this.longitude))*r;
6       double z=Math.Sin(Math.ToRad(this.latitude))*d;
7       this.point=new VecPnt(x, y, z, 1.0);
8       this.normMat=Mat3x3.CoordinateSpaceZX(this.point, VecPnt.ZAxis());
9       this.rotMat=Mat3x3.YRotation(this.elevation)*
                    Mat3x3.ZRotation(this.azimuth);
10      this.CoordValue=Mat4x4.PadMinor(this.normMat*this.rotMat, this.point);
    }
```

Code Snippet 13: ECEF to EFGS
(line 227 in Libraries\PhySim2AstroControls\SolarSystem\ECEFtoLLA.cs)

$$d = |altitude + 6378.15km|$$
$$r = \cos\left(\frac{\pi}{180}latitude\right) d$$
$$point_{3x1} = \left[\cos\left(\frac{\pi}{180}longitude\right) r \quad \sin\left(\frac{\pi}{180}longitude\right) r \quad \sin\left(\frac{\pi}{180}latitude\right) d\right]$$
$$norm_{3x3} = \text{CoordinateSpace}_{\text{ZX}}\left(point, \begin{bmatrix}0\\0\\1\end{bmatrix}\right)$$
$$rot_{3x3} = \mathcal{R}_Y\measuredangle(elevation) * \mathcal{R}_Z\measuredangle(azimuth)$$
$$ECEFtoEFGS_{4x4} = \begin{bmatrix} & & & 0\\ & norm * rot & & 0\\ & & & 0\\ point & & & 1\end{bmatrix}$$

Equation 8: Symbolic Representation of ECEF to EFGS

will point towards the prime meridian. When the location is the center of the Earth the coordinate system is the same as ECEF. Once the correct location and initial rotation have been determined the coordinate system is rotated around its center point with the elevation and azimuth.

Auto-Generated Conversions

The auto-generated conversions implement graph theory and matrix multiplication to create the entire set of relative conversions given as a small group of defined conversions. This allows the user to only implement the basic and simple conversions but make requests of the simulation using conversions not specifically defined.

Conversion Multiplication

The auto-generators are based off of one concept: to solve for a missing conversion two defined conversions may be multiplied together. For example if "A to C" is needed but all that is given is "A to B" and "B to C" then multiply "A to B" by "B to C" to get "A to C". The validation that is required here is to prove that since matrix multiplication is not commutative that the auto-generators will multiply in the correct order.

As shown in Figure 3, the vectors and points are represented as 1x4 matrices. This means that during multiplication the vector or point must be at the left of the equation. This is important to the order of multiplication. A 4x1 would require the reverse order.

Given that P_A is a point in the coordinate system A, M_{AB} is the matrix for "A to B" coordinate conversions, and M_{BC} is the matrix for "B to C" coordinate conversions. The matrix required is M_{AC}.

$$P_A M_{AB} = P_B$$
$$P_B M_{BC} = P_C$$
$$P_A M_{AB} M_{BC} = P_B M_{BC} = P_C$$
$$M_{AB} M_{BC} = M_{AC}$$
$$P_A M_{AC} = P_C$$

Equation 9: Proof of Matrix Multiplication Order.

By definition the first two equations are true. By replacing the P_B in the second equation with the equivalent found in the first equation, the resulting third equation is found to also be true. Using the associative property of matrices, the forth equation can be derived from the third. Using the forth equation, the third equation can be simplified into the final equation. That equation is the wanted result of the auto-generators. Therefore the forth equation can be used when acquiring required coordinate conversions which have not been defined.

Code Snippet 14 and 15 are how these matrices and vectors get multiplied together. Notice that the vector matrix multiplication only allows the vector to be on the left.

```
    /// <summary>This multiplies two matrices together.</summary>
    /// <param name="a">This is the left matrix.</param>
    /// <param name="b">This is the right matrix.</param>
    /// <returns>This is the product of the two matrices.</returns>
    static public Matrix4x4 operator*(Matrix4x4 a, Matrix4x4 b)
    {
1       Matrix4x4 mat=new Matrix4x4();
        // C[x,y]=Sum(A[x,i]*B[y,i]; i=0..3);
2       mat[0,0] = a[0,0]*b[0,0] + a[0,1]*b[1,0] + a[0,2]*b[2,0] + a[0,3]*b[3,0];
3       mat[0,1] = a[0,0]*b[0,1] + a[0,1]*b[1,1] + a[0,2]*b[2,1] + a[0,3]*b[3,1];
4       mat[0,2] = a[0,0]*b[0,2] + a[0,1]*b[1,2] + a[0,2]*b[2,2] + a[0,3]*b[3,2];
5       mat[0,3] = a[0,0]*b[0,3] + a[0,1]*b[1,3] + a[0,2]*b[2,3] + a[0,3]*b[3,3];
6       mat[1,0] = a[1,0]*b[0,0] + a[1,1]*b[1,0] + a[1,2]*b[2,0] + a[1,3]*b[3,0];
7       mat[1,1] = a[1,0]*b[0,1] + a[1,1]*b[1,1] + a[1,2]*b[2,1] + a[1,3]*b[3,1];
8       mat[1,2] = a[1,0]*b[0,2] + a[1,1]*b[1,2] + a[1,2]*b[2,2] + a[1,3]*b[3,2];
9       mat[1,3] = a[1,0]*b[0,3] + a[1,1]*b[1,3] + a[1,2]*b[2,3] + a[1,3]*b[3,3];
10      mat[2,0] = a[2,0]*b[0,0] + a[2,1]*b[1,0] + a[2,2]*b[2,0] + a[2,3]*b[3,0];
11      mat[2,1] = a[2,0]*b[0,1] + a[2,1]*b[1,1] + a[2,2]*b[2,1] + a[2,3]*b[3,1];
12      mat[2,2] = a[2,0]*b[0,2] + a[2,1]*b[1,2] + a[2,2]*b[2,2] + a[2,3]*b[3,2];
13      mat[2,3] = a[2,0]*b[0,3] + a[2,1]*b[1,3] + a[2,2]*b[2,3] + a[2,3]*b[3,3];
14      mat[3,0] = a[3,0]*b[0,0] + a[3,1]*b[1,0] + a[3,2]*b[2,0] + a[3,3]*b[3,0];
15      mat[3,1] = a[3,0]*b[0,1] + a[3,1]*b[1,1] + a[3,2]*b[2,1] + a[3,3]*b[3,1];
16      mat[3,2] = a[3,0]*b[0,2] + a[3,1]*b[1,2] + a[3,2]*b[2,2] + a[3,3]*b[3,2];
17      mat[3,3] = a[3,0]*b[0,3] + a[3,1]*b[1,3] + a[3,2]*b[2,3] + a[3,3]*b[3,3];
18      return mat;
    }
```

Code Snippet 14: Matrix Multiplication
(line 61 in Libraries\System.Types_3D\Matrix4x4.cs)

```
    /// <summary>This multiplies a vector by a matrix.</summary>
    /// <param name="a">This is the vector</param>
    /// <param name="b">This is the matrix.</param>
    /// <returns>This is the resulting vector.</returns>
    static public VecPnt operator*(VecPnt a, Matrix4x4 b)
    {
1       VecPnt vec=new VecPnt();
2       vec.X = b.m[0, 0]*a.X + b.m[1, 0]*a.Y + b.m[2, 0]*a.Z + b.m[3, 0]*a.W;
3       vec.Y = b.m[0, 1]*a.X + b.m[1, 1]*a.Y + b.m[2, 1]*a.Z + b.m[3, 1]*a.W;
4       vec.Z = b.m[0, 2]*a.X + b.m[1, 2]*a.Y + b.m[2, 2]*a.Z + b.m[3, 2]*a.W;
5       vec.W = b.m[0, 3]*a.X + b.m[1, 3]*a.Y + b.m[2, 3]*a.Z + b.m[3, 3]*a.W;
6       return vec;
    }
```

Code Snippet 15: Matrix/Vector Multiplication
(line 91 in Libraries\System.Types_3D\Matrix4x4.cs)

Completing the Graph

Given a set of edges, e', in a finite graph G, where $e' \subseteq E$ and E is all the possible connections in G, as defined by $\forall e_{x,y} \in E,\ e_{x,y} = \{x \leftrightarrow y | \ x, y \in G, x \neq y\}$, the auto-generators will create $E - e' = \bar{e}$, this causes G to be a complete graph. An edge is the connection between two coordinate systems, where the coordinate system is known as a vertex or node in the graph. Note that if the defined conversions do not fully connect, that is that they create separate graphs, then the auto-generators will treat them as separate graphs. The auto-generators will not make guesses or assumptions of any kind, only create the $\bar{e} \in E$ for all graphs. The auto-generators take advantage of the fact that each e is added in individually, therefore the auto-generators only have to run after another edge is added to a graph. Coordinate conversions are one directional but by definition the conversion matrix must be invertible to be an actual conversion, therefore the first thing that the auto-generators do is generate the inverse matrix and add it to the simulator. Since the inverse is immediately added to the simulator these connections may be treated as undirected edge by the auto-generator, $\exists e_{x,y} \in E \ \therefore \exists e_{y,x} \in E$, however the output equations will use only one direction. The conversion matrix for the edge going one direction is always the inverse of the conversion matrix for the edge going the opposite direction between the same two nodes.

For example, given $\{ e_{A,B}, e_{B,E}, e_{D,C}, e_{E,F}, e_{E,D}\} = e'$ the following occurs assuming that they are added in the order as stated in the list. This example will demonstrate how the code, on a high level, performs the auto-generation. Note that the number of connections in a graph, for the graph to be complete, is $|E| = \frac{(n-1)n}{2}$, where n is the number of vertices in G.

- When $e_{A,B}$ is added the auto-generators only create the inverse $e_{B,A}$ which causes the graph to be complete, see Figure 15. Since there will always be both directions the drawing can simplify to look like Figure 16.
- When $e_{B,E}$ is added to the graph, the auto-generator notices the vertex B is shared between two edges. It creates $e_{A,B} * e_{B,E} = e_{A,E}$ and $e_{E,B} * e_{B,A} = e_{E,A}$, as shown in Figure 17, to complete the graph. $n = 3 \ \therefore |E| = 3$.
- Next $e_{D,C}$ is added to the simulation. The auto-generations cannot find a shared point so it doesn't add anything to the simulation, other than the inverse matrix for $e_{C,D}$ of course. See Figure 18.
- $e_{E,F}$ is added to the simulation. The auto-generators notice that the vertex E is shared by $e_{E,F}$ with both $e_{B,E}$ and the auto-generated $e_{A,E}$. Therefore it creates the connections $e_{B,F}$, $e_{A,F}$ and the complement inverse connections, as shown in Figure 19.
- When $e_{E,D}$ is added the auto-generator notices several connections since $e_{E,D}$ bridged the two graphs together. For the shared vertex E, it located $e_{E,F}$, $e_{E,A}$, and $e_{E,B}$ which it uses to create $e_{D,F}$, $e_{D,A}$, and $e_{D,B}$ seen in Figure 20. It also notices that the vertex D is shared, not only by $e_{D,C}$ but also by the newly created edges, $e_{D,F}$, $e_{D,A}$, and $e_{D,B}$. It adds the four new connections, shown in Figure 21.

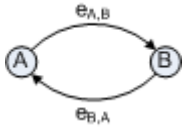


Figure 15

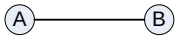
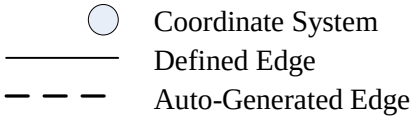


Figure 16



Figures 15-24 Key

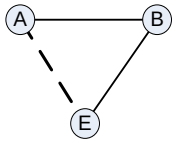


Figure 17

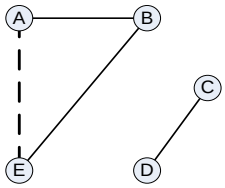


Figure 18

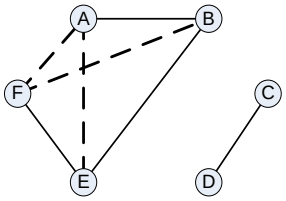


Figure 19

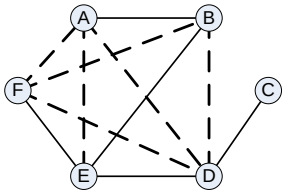


Figure 20

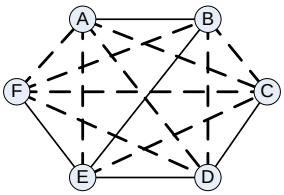


Figure 21

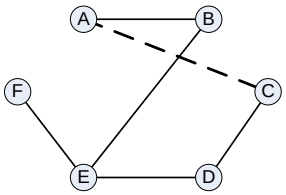


Figure 22

In this example the edge count is 15, which since $n = 6$ the graph is again complete. The total coordinate conversions defined are 5 meaning 25 are auto-generated. For each complete graph defined by n coordinate conversions, n^2 coordinate conversions are auto-generated. The resulting coordinate conversions will be added to the simulation in the following order: $e'_{A,B}, e_{B,A},\ e'_{B,E}, e_{E,B},\ e_{E,A}, e_{A,E}, e'_{D,C}, e_{C,D},\ e_{A,F}, e_{F,A},\ e'_{E,F}, e_{F,E},\ e_{F,B}, e_{B,F},\ e_{F,A}, e_{A,F},$ $e'_{E,D}, e_{D,E},\ e_{D,B}, e_{B,D},\ e_{D,A}, e_{A,D},\ e_{D,F}, e_{F,D}, e_{E,C}, e_{C,E},\ e_{B,C}, e_{C,B},\ e_{A,C}, e_{C,A},\ e_{F,C}, e_{C,F}$. Note that defined conversions are marked as $e'_{x,y} \in e'$.

The dependency of the conversion is important so that the multiplication of auto-generated conversions occur only after the conversions it relies on have been updated. Even though the inverse is dependent on the original conversion they are considered both on the same dependency level because the inverse is always calculated immediately after the original is calculated. The maximum dependency level for a graph will be one less than the maximum path

length of the graph when only the defined connections are present, as shown in Figure 22. In this example, the maximum path length is 4, following the path {A, B, E, D, C}. This means that the dependency level of $e_{A,C}$ is 3. The auto-generators built $e_{A,C} = e_{A,D} * e'_{D,C}$. $e'_{D,C}$ is defined, therefore its dependency level is 0, however $e_{A,D} = e_{A,E} * e'_{E,D}$ and $e_{A,E} = e'_{A,B} * e'_{B,E}$. It follows that $e_{A,C} = e_{A,D} * e_{D,C} = (e_{A,E} * e_{E,D}) * e_{D,C} = ((e_{A,B} * e_{B,E}) * e_{E,D}) * e_{D,C}$, therefore showing why the dependency is 3.

The dependencies built by the auto-generators, assuming all defined conversions have a dependency level of 0, for all edges are as follows for this example:

- Level 0: $e'_{A,B}, e'_{B,E}, e'_{D,C}, e'_{E,F}, e'_{E,D}$
- Level 1: $e_{E,A}, e_{F,B}, e_{D,B}, e_{D,F}, e_{E,C}$
- Level 2: $e_{F,A}, e_{D,A}, e_{B,C}, e_{F,C}$
- Level 3: $e_{A,C}$

PhySim will determine the final dependencies base off of the variable requests from each controller. The auto-generators will add a new control for each conversion and inverse pair. The new control will request the four conversion matrices required by the new conversion and inverse. By making these requests PhySim will enforce the dependencies, to guarantee that the new conversion and inverse will be updated only after the four conversions they need are updated.

The reason four conversions are requested instead of two is because making the extra two requests and extra multiplication takes less time than performing an inverse. The only inversions performed will be on the original defined matrices. Since the graph is complete it is guaranteed that all four conversions will exist before a new conversion is created.

Global Points and Vectors

The auto-generators will also create all global points and vectors in all related coordinate systems. When a global vector or point is added that point or vector checks for all conversions which start at the coordinate system they are in, then, using those conversions creates new points or vectors in the ending coordinate systems. When a new coordinate conversion is added, and all the subsequent coordinate conversions have been added, the auto-generators check the list of defined global points and vectors for any which are in effected coordinate system. If a point or vector exists the auto-generators will create a new point for each new coordinate conversion starting from that point or vectors base coordinate system.

Given a graph, G, a set of connections, $e_{xy} \in E$ for that graph and a point or vector P_v where $v \in V$ and V is the set of vertices in the graph P can be propagated as follows: $\forall P_x;\ \exists P_y\ \leftrightarrow \exists e_{xy} \in E;\ x, y \in V;$ Simply stated, for each point or vector with the base coordinate system of x, there exists a complement point or vector with the base coordinate system of y, if and only if there exists an edge between both coordinate systems. Since the graph is complete the auto-generators will distribute a global point or vector into all connected coordinate systems by multiplying the vector or point by the conversion matrix. Once again the auto-generators will not make guesses or assumptions of any kind. If there is no connection between a group of coordinate systems and another the global vector or point will remain confined in the group in which its base coordinate system exists.

The Specific Conversions

As a more specific example of how the auto-generators work with the PhySim2AstroControls, assume the given conversions, vector, and points are created in the following order and the satellite is called “Sat1”:

1. Convert ECIR to HE
2. Sun’s Location in HE
3. Earth’s Location in ECIR
4. Convert ECIR to ECEF
5. Sat1’s Location in ECIR
6. Sat1’s Velocity in ECIR
7. Convert ECIR to Sat1 SCIR
8. Convert Sat1 SCIR to Sat1 RN
9. Sat1’s Rotational Velocity in Sat1 RN
10. Sat1 RN to Sat1 Housing

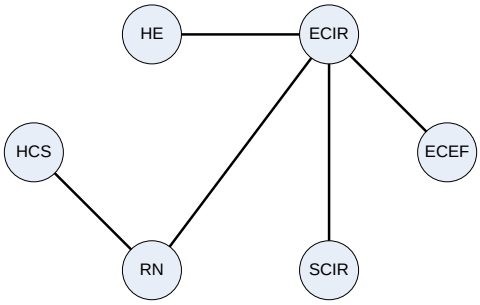


Figure 23

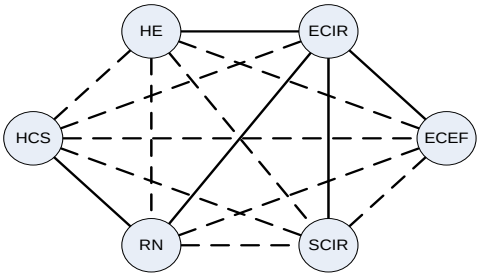


Figure 24

The defined graph is shown in Figure 23. The auto-generators create the graph shown in Figure 24 as well as distribute the global vectors and points to all other coordinate systems. The following list shows the defined value being added and the resulting auto-generated values. It can be seen that all 30 connections are made as well as all 18 global points and 12 global vectors as expected.

1. Convert ECIR To HE
    a. Convert HE To ECIR
2. Sun's Location In HE
    a. Sun's Location In ECIR
3. Earth's Location In ECIR
    a. Earth's Location In HE
4. Convert ECIR To ECEF
    a. Convert ECEF To ECIR
    b. Convert ECEF To HE
    c. Convert HE To ECEF
    d. Earth's Location In ECEF
    e. Sun's Location In ECEF
5. Sat1's Location In ECIR
    a. Sat1's Location In HE
    b. Sat1's Location In ECEF
6. Sat1's Velocity In ECIR
    a. Sat1's Velocity In HE

   b. Sat1's Velocity In ECEF
7. Convert ECIR To Sat1 SCIR
   a. Convert Sat1 SCIR To ECIR
   b. Convert Sat1 SCIR To HE
   c. Convert HE To Sat1 SCIR
   d. Convert Sat1 SCIR To ECEF
   e. Convert ECEF To Sat1 SCIR
   f. Earth's Location In Sat1 SCIR
   g. Sat1's Location In Sat1 SCIR
   h. Sun's Location In Sat1 SCIR
   i. Sat1's Velocity In Sat1 SCIR
8. Convert Sat1 SCIR To Sat1 RN
   a. Convert Sat1 RN To Sat1 SCIR
   b. Convert Sat1 RN To ECIR
   c. Convert ECIR To Sat1 RN
   d. Convert Sat1 RN To HE
   e. Convert HE To Sat1 RN
   f. Convert Sat1 RN To ECEF
   g. Convert ECEF To Sat1 RN
   h. Earth's Location In Sat1 RN
   i. Sat1 's Location In Sat1 RN
   j. Sun's Location In Sat1 RN
   k. Sat1 's Velocity In Sat1 RN
9. Sat1 Rotational Velocity In Sat1 RN
   a. Sat1 Rotational Velocity In Sat1 SCIR
   b. Sat1 Rotational Velocity In ECIR
   c. Sat1 Rotational Velocity In HE
   d. Sat1 Rotational Velocity In ECEF
10. Convert Sat1 RN To Sat1 Housing
   a. Convert Sat1 Housing To Sat1 RN
   b. Convert Sat1 Housing To Sat1 SCIR
   c. Convert Sat1 SCIR To Sat1 Housing
   d. Convert Sat1 Housing To ECIR
   e. Convert ECIR To Sat1 Housing
   f. Convert Sat1 Housing To HE
   g. Convert HE To Sat1 Housing
   h. Convert Sat1 Housing To ECEF
   i. Convert ECEF To Sat1 Housing
   j. Earth's Location In Sat1 Housing
   k. Sat1 's Location In Sat1 Housing
   l. Sun's Location In Sat1 Housing
   m. Sat1 Rotational Velocity In Sat1 Housing
   n. Sat1 's Velocity In Sat1 Housing

Algorithms

There are two very important algorithms for VideSupra. The first determines where the satellite’s location is based on the time and orbital information. The second is an algorithm for

determining the magnetic force from the Earth given a time and location. These algorithms are in SpaceTrack.Net and GeoMag.Net respectively. They have been included into VideSupra as individual libraries for general purpose access to the algorithms. They also have been wrapped in controls which can be easily added to the simulator. The algorithms were originally created by third parties, Spacetrack[16] and WMM[11], but had to be translated into C# and updated to work properly with VideSupra, Spacetrack.Net[17] and GeoMag.Net[12].

SpaceTrack.Net

In SpaceTrack.Net is a managed implementation of the Space Track Report Number 3 [15]. Just as the GeoMag.Net is extended from the original so is the SpaceTrack.Net. The original takes only a TLE and returns six values representing the ECIR's x, y, and z, in kilometers, and dx, dy, dz, in kilometers per second. The extension allows the space track solver to be changed out so that the simulation can quickly change between SGP, SGP4, SGP8, SDP4, and SDP8. Also the extension parses a TLE into a Data storage type, however the Data storage type can be filled without using the TLE allowing other forms of sending the data to the solver. This also allows the use of the managed System.DateTime to handle the time. Finally, the constants are stored in an external interchangeable file so that the planet description can be changed.

The values required by the solvers to calculate the location and velocity of a satellite are all stored in the Data storage type. These values are as follows, where the names are inherited from the FORTRAN IV implementation:

- EPOCH: YYDDD.FFFF, 2 digit year, 3 digit day of year, fractional period of day.
- XNDT2O: The time rate of change of "mean" mean motion at epoch.
- XNDD6O: The second time rate of change of "mean" mean motion at epoch.
- BSTAR: The drag coefficient (B*).
- XINCL: The "mean" inclination at epoch (i0).

- XNODEO: Right Ascension of the Ascending Node (RAAN in degrees).
- EO: The "mean" eccentricity at epoch (e0).
- OMEGAO: The "mean" argument of perigee at epoch (w0).
- XMO: The "mean" mean anomaly at epoch (M0).
- XNO: The "mean" mean motion at epoch (n0).
- XNODP: The orbital period.

For more information about the SpaceTrack satellite propagation algorithms see [15] or the included library in the VideSupra project package, SpaceTrack.Net.

GeoMag.Net

The GeoMag.Net library was created for the Astrophysics Plug-in. GetMag.Net is a C# implementation of the algorithm defined in the "US/UK World Magnetic Model for 2005-2010" [11] (WMM). It also adds methods to make using the library easier for the simulator.

During initialization the GeoMag.Net model loads a set of magnetic readings from a file provided by the NOAA's (National Oceanic and Atmospheric Administration) NGDC (National Geophysical Data Center). Then given a time and location the library will calculate the Earth's magnetic field for that point at that time.

The following diagrams, see Figure 25, are excerpts from the WWM document paired with the corresponding image captures from the GeoMag.Net library. The excerpts are from pages 48 to 58 of the WMM document. The annual rate of change is not show in this document since the library is not currently setup to calculate the rate of change. The libraries accurately and precisely predicted all the test values located on page 45 and 46 of the WWM document. The simulator gathered 15480 points of data at 700km above sea-level (WGS 84) for 01-01-2008 using the 2005 WMM.cof file.

The image captures show this data with the minimum value as black and maximum value as white. The maximum and minimum values matched the test values provided with the document. They use the set of discreet points taken from the simulator and are plotted using a simple rendering tool (the rendering tool is not provided with VideSupra). Any inconsistent artifacts in the images are due to the rendering process, such as the black line in the upper left corner of all Mercator (rectangular) images and two white dots in most of the polar (round) images. These artifacts do not exist in the data. Finally, note that the pole images show from the equator to the pole, the entire hemisphere, whereas the excerpt shows only about 60 degrees from the pole.

It can clearly be seen in the Figures 25-31 images that the GeoMag.Net is a valid implementation of the WMM algorithm. The results from GeoMag.Net library match the results in the WMM documentation. The GeoMag.Net is a managed version of the National Geophysical Data Center's geomag.c algorithms that use the same WMM.cof file as a data source. It determines the magnetic field at a given location and time.

The GeoMag.Net has been extended from the geomag.c so that it can give a variety of output formats and take a variety of input formats as well. The original geomag.c could only take longitude, latitude, altitude and decimalized year. It would only return a declination, inclination, horizontal strength and total field strength. The GeoMag.Net allows the use of the managed System.DateTime for its time input and output and on top of the original input/output types, it can also give absolute coordinates in ECEP. With these extensions the results from the SpaceTrack.Net can be piped into the GeoMag.Net to get the nanotulsas (nT) at the satellites current location.

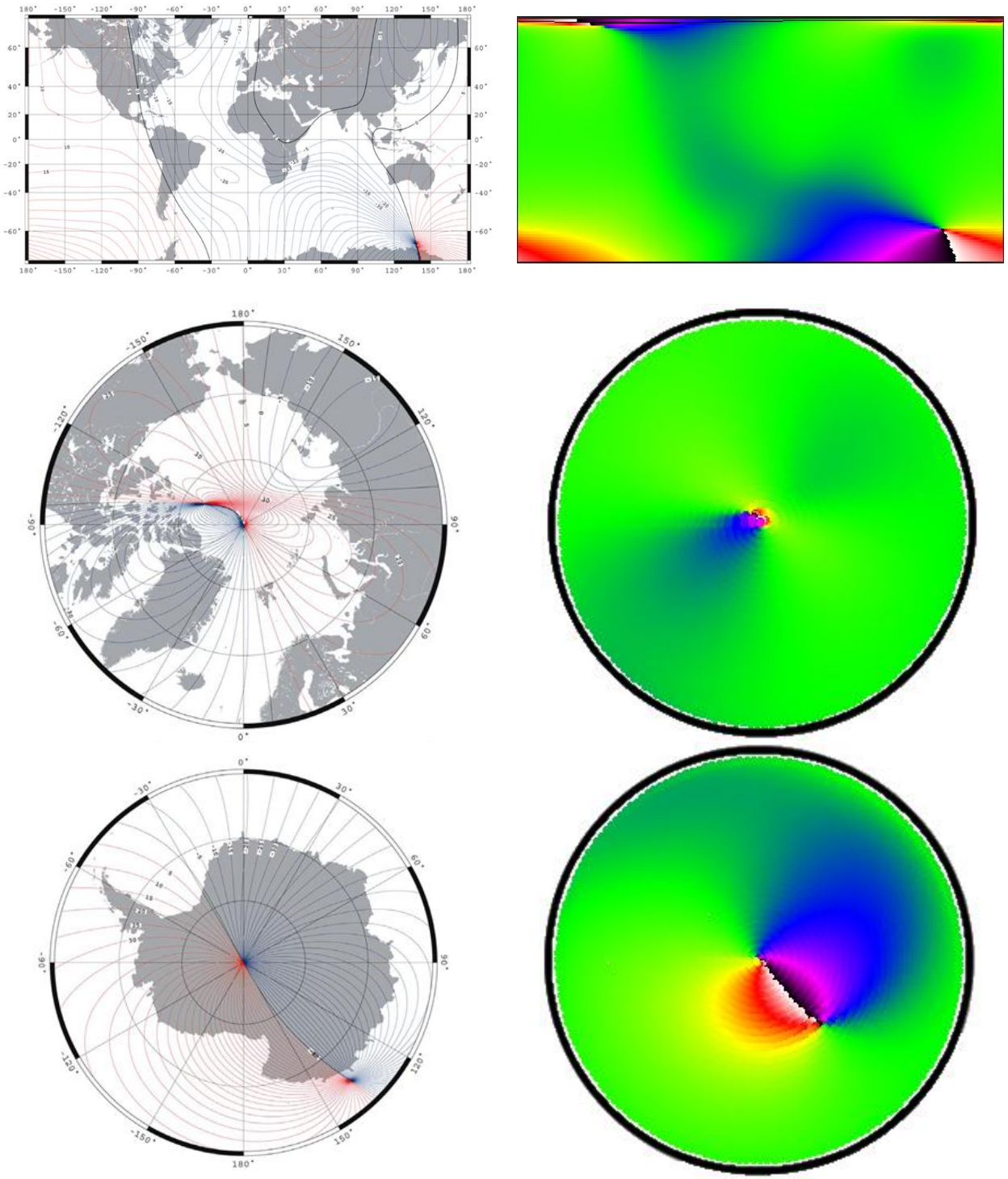


Figure 25: Magnetic Declination (D)
Mercator, North polar region, and South polar region.
Contour interval is 5 degrees, red contours positive (east); blue negative (west); black – zero (agonic) line.
Mercator and polar stereographic projections at 2005.0 for the World Magnetic Model 2005.

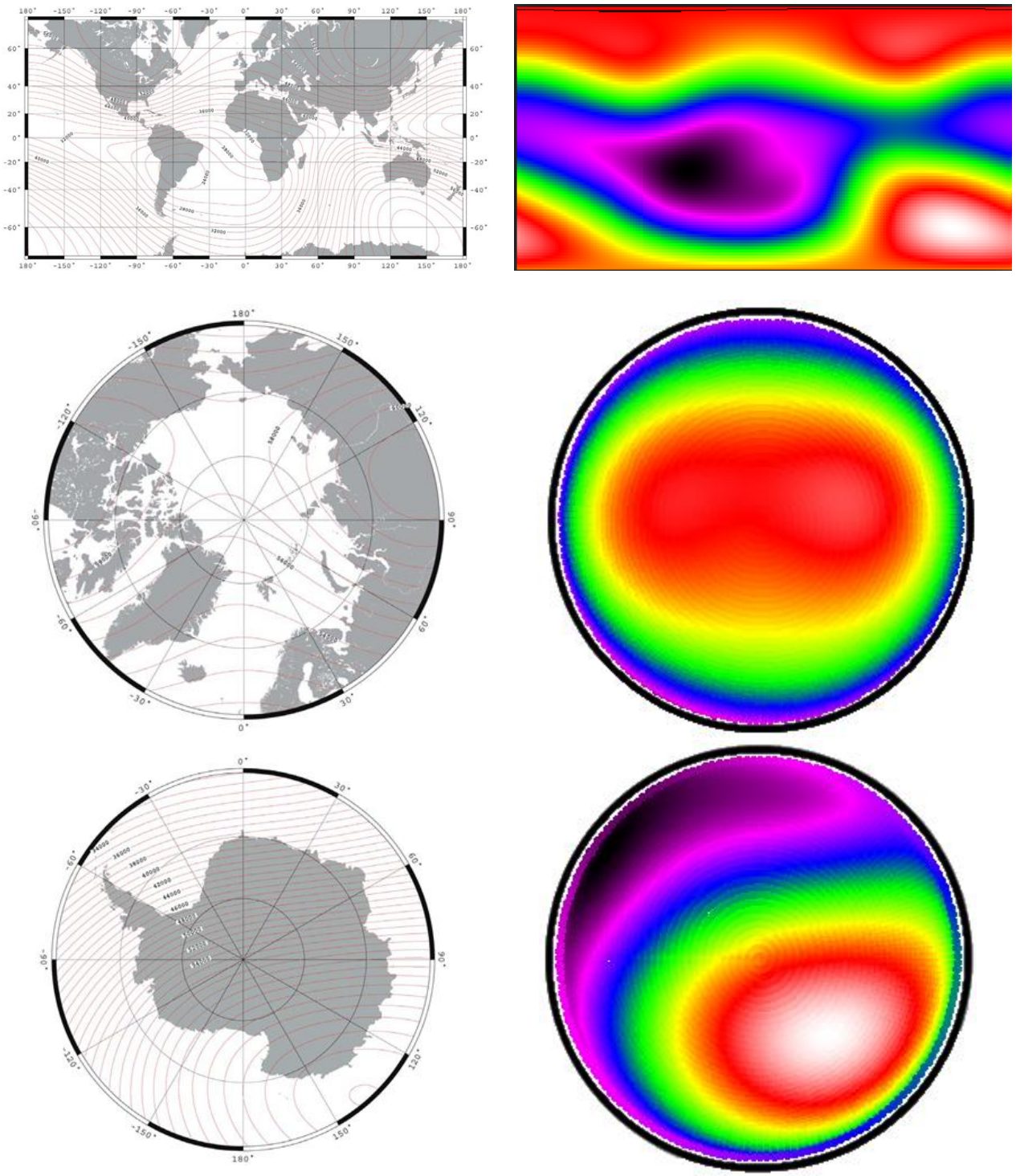


Figure 26: Magnetic Total Intensity (F)
Mercator, North polar region, and South polar region.
Mercator projection contour interval is 2000 nT. North and South polar region contour interval is 1000 nT.
Mercator and polar stereographic projections at 2005.0 for the World Magnetic Model 2005.

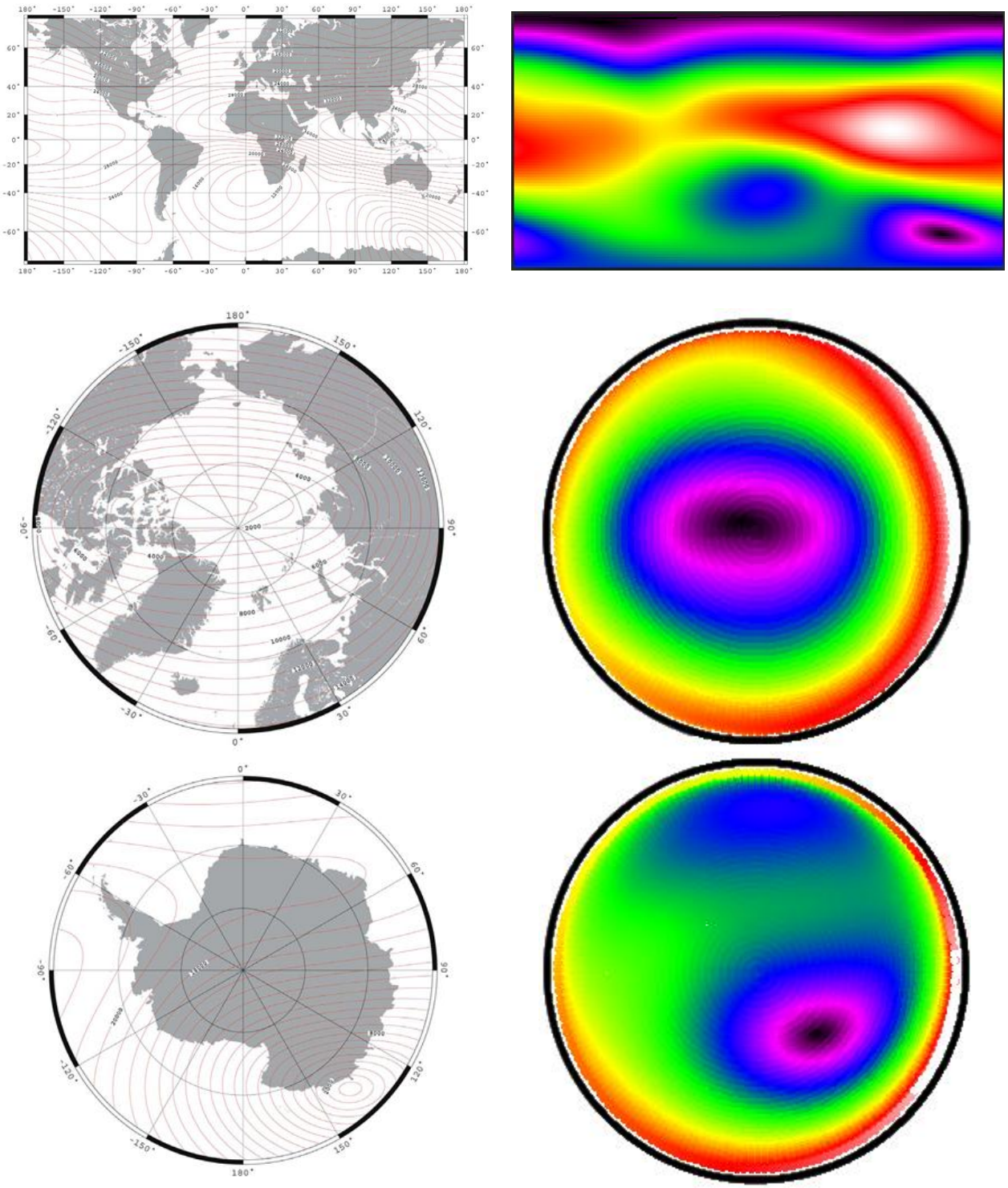


Figure 27: Magnetic Horizontal Intensity (H)
Mercator, North polar region, and South polar region.
Mercator projection contour interval is 2000 nT. North and South polar region contour interval is 1000 nT.
Mercator and polar stereographic projections at 2005.0 for the World Magnetic Model 2005..

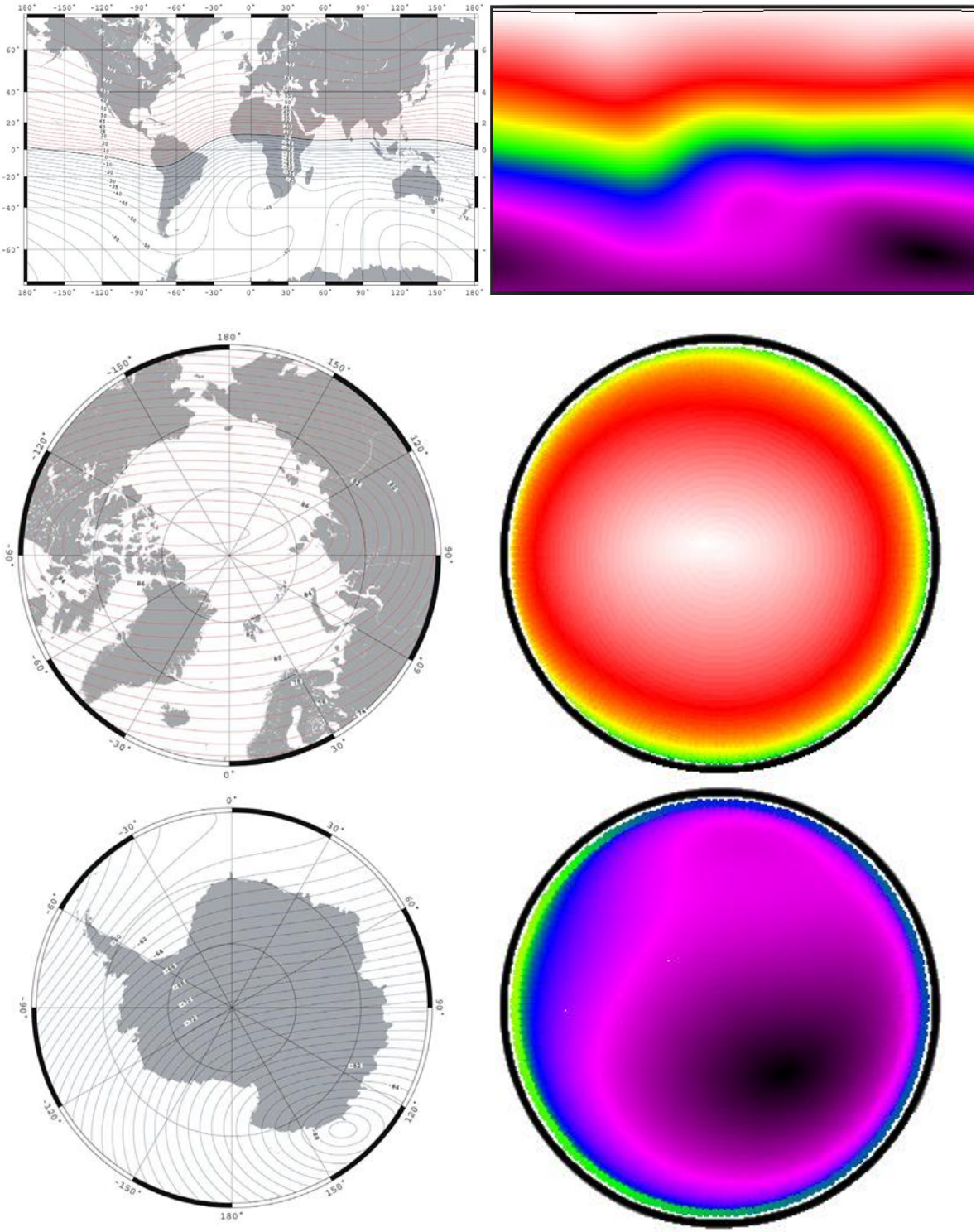


Figure 28: Magnetic Inclination (I)

Mercator, North polar region, and South polar region. Mercator projection contour interval is 5 degrees, red contours positive (down); blue negative (up); black – zero (agonic) line. North and South polar region contour interval is 1 degree. Mercator and polar stereographic projections at 2005.0 for the World Magnetic Model 2005

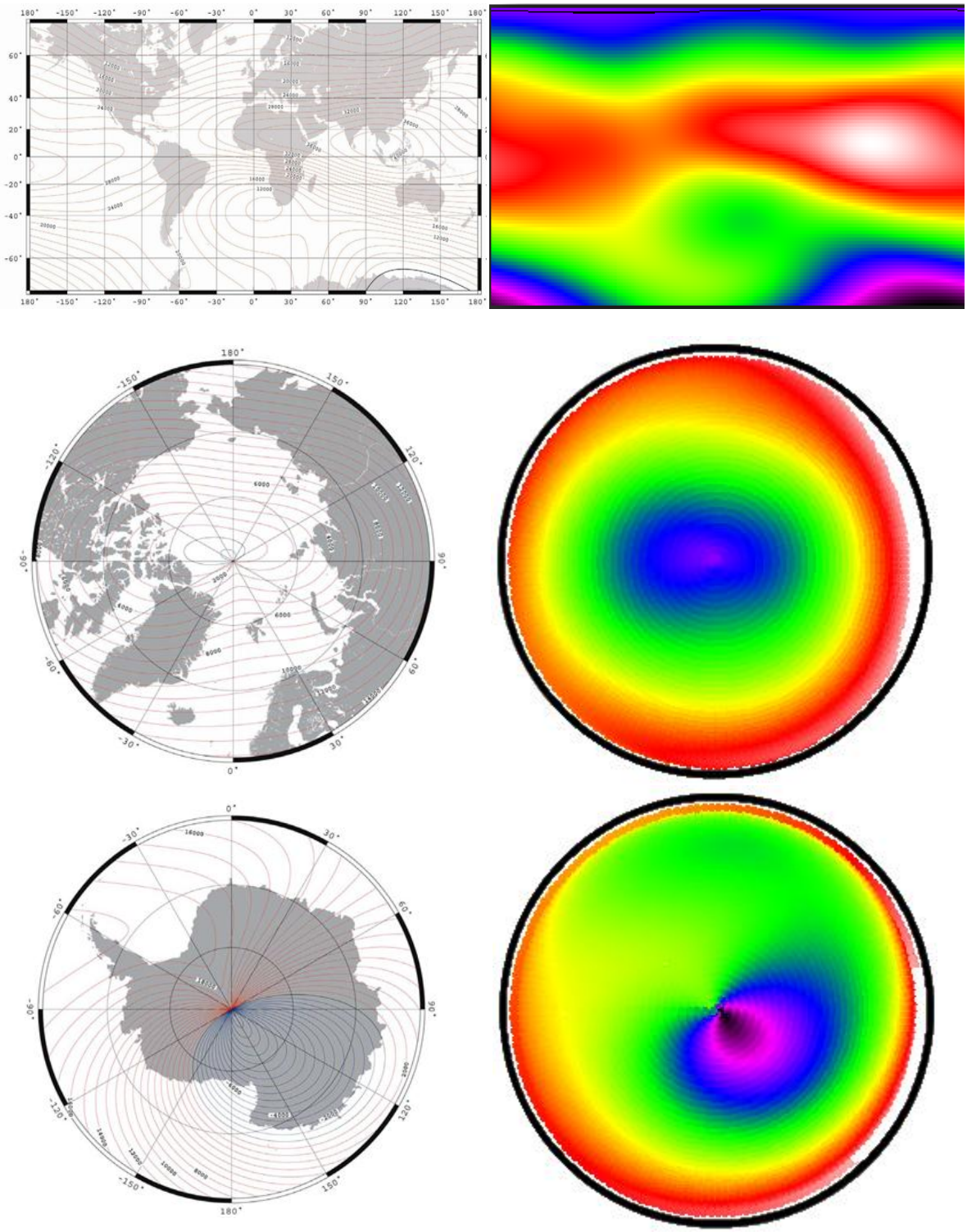


Figure 29: Magnetic North Component (X)

Mercator, North polar region, and South polar region. Mercator projection contour interval is 2000 nT, North and South polar region contour interval is 1000 nT, red positive (north), blue negative (south), black – zero. Mercator and polar stereographic projections at 2005.0 for the World Magnetic Model 2005

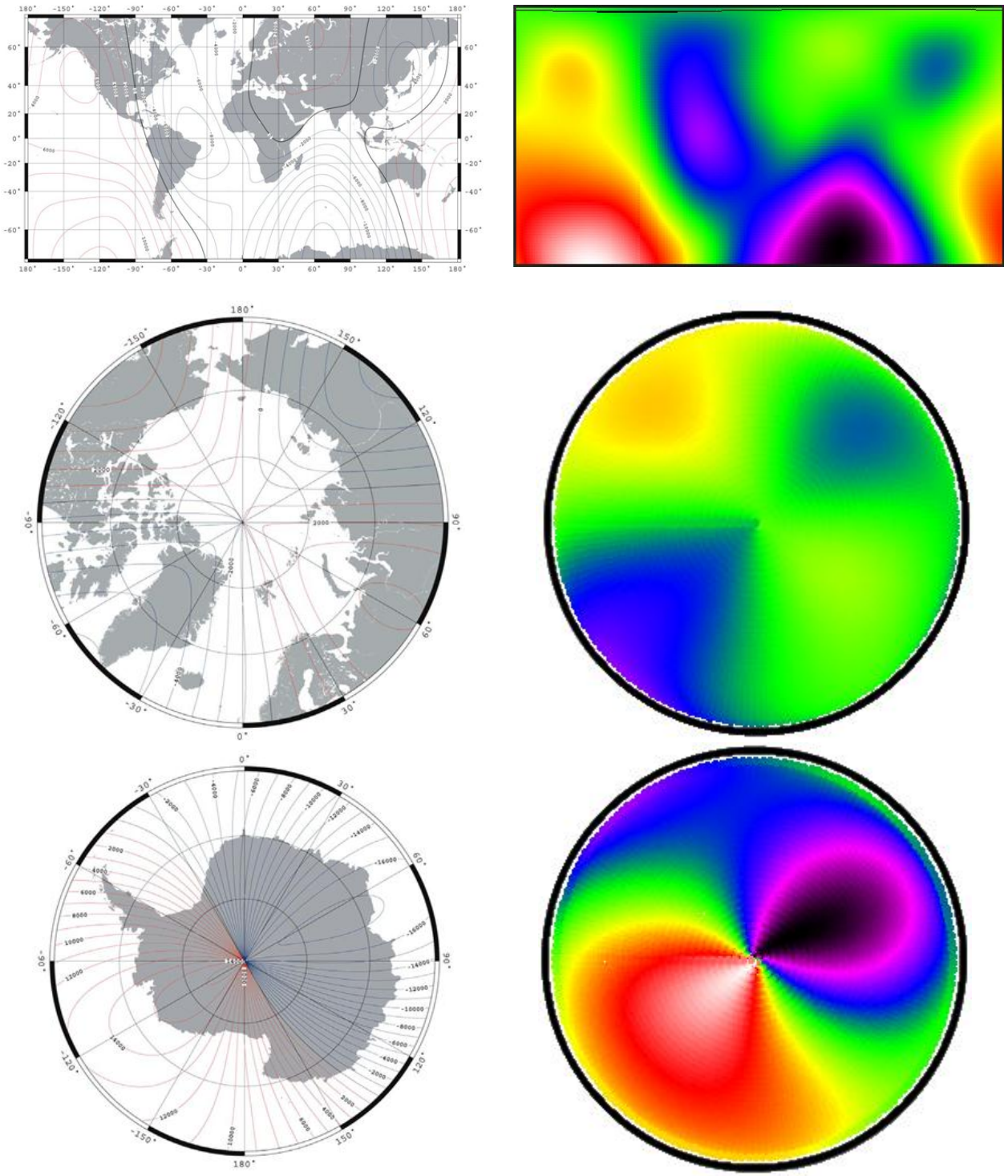


Figure 30: Magnetic East Component (Y)

Mercator, North polar region, and South polar region. Mercator projection contour interval is 2000 nT, North and South polar region contour interval is 1000 nT, red positive (north), blue negative (south), black – zero. Mercator and polar stereographic projections at 2005.0 for the World Magnetic Model 2005

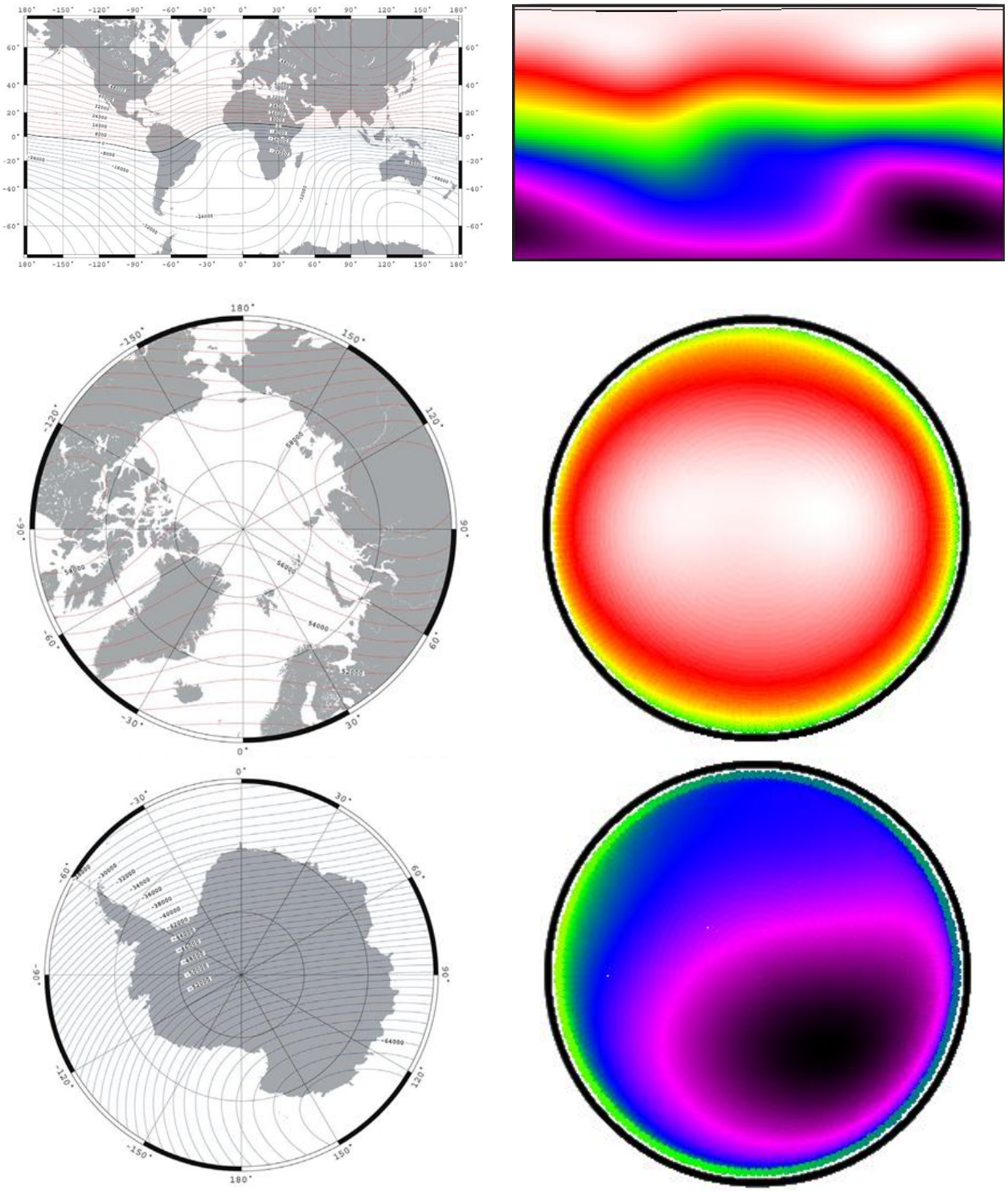


Figure 31: Magnetic Vertical Intensity (Z)

Mercator, North polar region, and South polar region. Mercator projection contour interval is 2000 nT, North and South polar region contour interval is 1000 nT, red positive (north), blue negative (south), black – zero. Mercator and polar stereographic projections at 2005.0 for the World Magnetic Model 2005

Rigid Body Dynamics

The Rigid Body Dynamics control is designed for handling rigid body dynamics calculations for frictionless inertial reference frame. This control takes the previous orientation, the applied torque, the rotational vector and the inertia tensor of the satellite. It will update the rotational vector and orientation.

```
   /// <summary>This calculates the values to set.</summary>
   /// <param name="deltaT">This is the time step, in seconds, to take.</param>
   protected virtual void SubCalculate(double deltaT)
   {
      // Strip forth row and column from orientation. Location doesn't matter
      // here since this is in an inertial reference frame.
1     Mat3x3 orientation=Mat3x3.Minor(this.CoordValue);

      // Get momentum in inertial frame...
2     this.hndlMomentum.Value+=deltaT*this.torque;

      // Compute inverse inertia tensor in inertial frame...
3     Mat3x3 invI=orientation*Mat3x3.Transpose(orientation*
                  this.rigidBody.InverseOfInertiaTensor);

      // Get rotational velocity in inertial frame...
4     this.rotation.Vector=this.hndlMomentum.Value*invI;

      // Add rotational difference to orientation matrix...
5     orientation+=(deltaT*Mat3x3.Cross(this.rotation.Vector)*orientation);

      // Normalize the orientation...
      // Add forth row and column from orientation...
6     orientation=Mat3x3.Orthogonalise(orientation);
7     this.CoordValue=Mat4x4.PadMinor(orientation);
   }
```

Code Snippet 16: Rigid Body Dynamics
(line 118 in Libraries\PhySim2AstroControls\RBD\BaseRigidBodyDynamics.cs)

$$momentum_{1x3}^{i} = momentum_{1x3}^{i-1} + dt * torque_{1x3}^{i-1}$$
$$invI_{3x3}^{i} = orientation_{3x3}^{i-1} * \left[orientation_{3x3}^{i-1} * inertialTensor^{-1}\right]^{T}$$
$$rotation_{1x3}^{i} = momentum_{1x3}^{i} * invI_{3x3}^{i}$$
$$crossRotation_{3x3}^{i} = \begin{bmatrix} 0 & -Z_{rotation}^{i} & Y_{rotation}^{i} \\ Z_{rotation}^{i} & 0 & -X_{rotation}^{i} \\ -Y_{rotation}^{i} & X_{rotation}^{i} & 0 \end{bmatrix}$$
$$orientation_{3x3}^{i} = \left\| orientation_{3x3}^{i-1} * \left(1 + dt * crossRotation_{3x3}^{i}\right) \right\|$$

Equation 10: Symbolic Representation of Rigid Body Dynamics

Code Snippet 16 shows a single step in the rigid body dynamics calculations. The total time step of the simulation can be broken up so that more smaller steps can be made. This control is designed to be inherited such that before each smaller step the torque can be updated. This also gives the possibility to add a drag to the rotational vector.

One of the controls, Magnetic RDB, included in VideSupra, inherits the rigid body dynamics. This control overrides the "SubCalculate" method to calculate the torque given the magnetic moment of the torque coils and the B field. Code Snippet 17 shows the overridden class.

```
    /// <summary>This is a subcalculation step.</summary>
    /// <param name="deltaT">
    /// This is the change in time, in seconds, for this step.
    /// </param>
    protected override void SubCalculate(double deltaT)
    {
        // Calculate magnetic moment in inertial reference frame.
1       VecPnt moment=this.hndlMoment.Value*this.CoordValue;

        // The torque is multiplied by "deltaT" inside the base method.
2       this.torque=VecPnt.Cross(moment, this.bfield.Value);
3       base.SubCalculate(deltaT);
    }
```

Code Snippet 17: Magnetic Rigid Body Dynamics
(line 70 in Libraries\PhySim2AstroControls\RBD\MagneticRBD.cs)

Other Sensors

The Magnetometer control simply uses a sensor vector in a coordinate system and the magnetic vector in the same coordinate system. The magnitude of the projection of the magnetic vector onto the sensor vector is the sensor reading. The raw reading is in nanotulsas (nT) but it can be scaled and converted to simulate different binary outputs from specific magnetometer hardware. Some of the hardware has multiple vectors, for these multiple results are returned by combining the readings of several individual sensors. Typically the sensor is in HCS and the magnetic vector is set in SCIR but because of the auto-generators for vectors the sensor can simply ask for the magnetic vector in HCS.

Another simple control is the Doppler shift sensor which outputs the predicted Doppler shift in a given frequency for any two coordinate systems. This can be used between a satellite and ground station or between two satellites. This sensor calculated the shift using the distance in the coordinate system conversion matrix. As the distance changes from cycle to cycle a velocity is calculated. This was later updated to only work between a satellite and ground station or satellite and satellite. This newer version uses the relative velocity of the objects to get the Doppler shift. This proved to give more accurate results.

One control is the HIL output. This is a control which creates a com port to the hardware which simulates the actual sensors for the actual satellite hardware. This sensor simply collects the results from the variable pool, formats them for the com port and sends them to the HIL. This control could be extended to include the torque coil voltages thereby not only making it a sensor but an actuator.

The last simple sensor is the communication connection prediction control. This will test to see if a satellite is within line of sight and if that line is within a cone. This determines if a satellite is within the cone of broadcast from a satellite dish or antenna in a EFGS. From this the duration of predicted communication connection could be measured. Not only is this useful in determining an orbit but it helps develop and test the software driving the satellite dish alignment motors.

The sun sensor performs just like the communication connection prediction control. A line of sight is taken to the sun and tested against the conical view of the sensor. The only difference is that the sensor is mounted in a HCS instead of in a EFGS. For most projects the moon will intersect with the sun for so little time that the sun sensor will not have to take the moon into account. With a satellite orbiting at 7 Km/s, the satellite will only be in the umbra of the moon for about 1 to 2 seconds per orbit and only when the satellite can pass through the umbra.

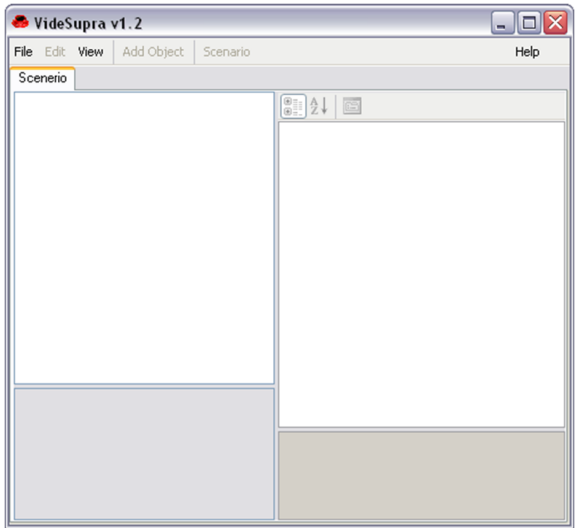
The results from a sun sensor can be run through a thermal conductive equation for simulating the temperatures of top and bottom thermal plates. The difference can be used as a black body sensor result. The scaling for black body radiation on the top plate and rate of conduction to the bottom plate can be tuned for the specific sensor hardware that is being simulated.

Since VideSupra is easily extendable many other controls can be added. VideSupra is designed for this so that controls specific to a task could be quickly written and added. A few of them were designed but not implemented, like a control for tracking the moon. For most use the controls defined will be accurate enough but some groups may want higher precision. For example the SCIR to ECIR control treats the Earth's orbit as perfectly circular. Also their ECIR to ECEF doesn't include the planetary wobble. The VideSupra package comes with most but not all of these controls and it has a few controls not discussed in the scope of this paper. The controls that aren't included with the package and shared as open source contained 3rd party software, proprietary software, or software for military grade hardware. VideSupra could never support all the requirements for each case. It expects that the user of this software will have to create or modify the controls for their own usage.

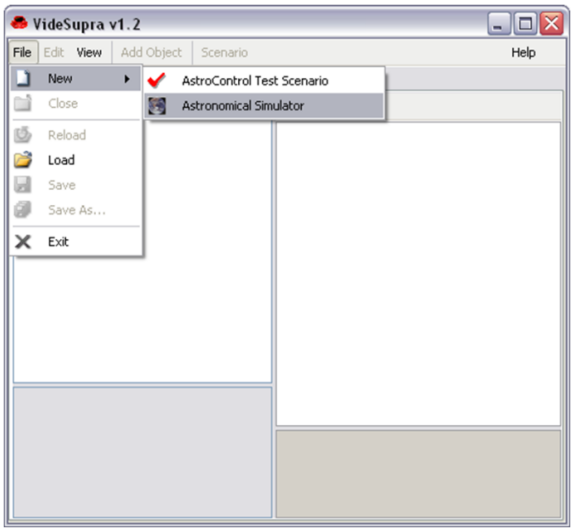
RUNNING AN ASTROPHYSICS SIMULATION

If this is the first time VideSupra is run or that last project was closed before VideSupra was closed then VideSupra will start as seen in Screen Capture 1. Otherwise the project which was open when VideSupra was closed is reloaded. To close a project that is already loaded click “File” then “Close”.

To create a new astronomical simulation, first select “Astronomical Simulator” from the “File” - “New” menu (See Screen Capture 2). This is the given name for the PhySim2AstroPlugin package. This package will allow the simulation of one or more satellites.



Screen Capture 1



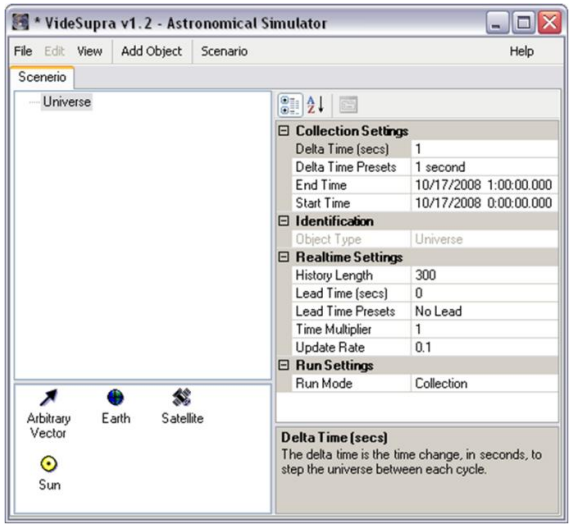
Screen Capture 2

If more or less options appear in the menus this is because of different plug-ins which are provided. The icon menu (not shown in screen captures) provides quick access to common controls in the menu. When the user clicks on the tree list (left panel) with the left mouse button a context menu will popup showing the currently available commands. It is recommended that VideSupra is run maximized so that all the data can be better viewed. The size and location that

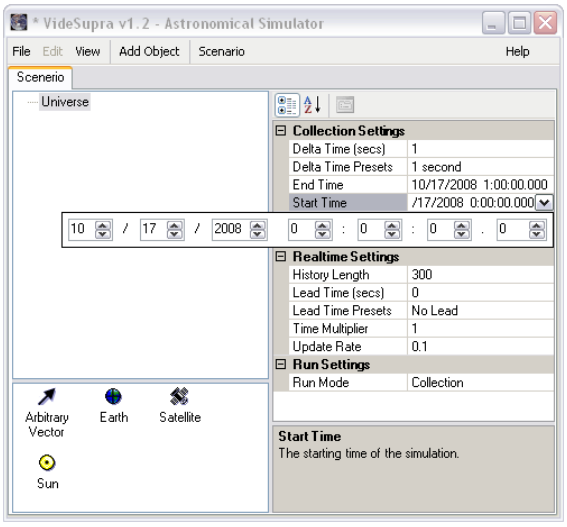
VideSupra is closed at is restored when it is restarted.

The left panel in Screen Capture 3 shows the simulation tree. The simulation tree is made up of objects describing the setup of the simulation. Each object represents zero or more controls that will be added to the simulation. The objects help define which controls can be used together by defining a hieratical configuration. For example a universe may not contain a universe. A universe can contain one Sun and the Sun can contain one Earth. This keeps conflicting controls or repeat controls from being added.

Below the tree is a list of objects which can be added to the currently selected object. Also when an object is selected the right panel will show the properties for that object. The bottom of the property grid is a description of the selected object's selected property. If a property is grayed out, as "Object Type", under the "Identification" category, is, the property cannot be modified. A read-only property is used to given more information about the currently selected object. In this case, if the "Object Type: Universe" property is clicked on, then the property description about the Universe object will be shown in the property description panel. It is possible that the property description panel isn't showing the whole description because the panel is too small. It can be resized by clicking and dragging the border between the property grid and



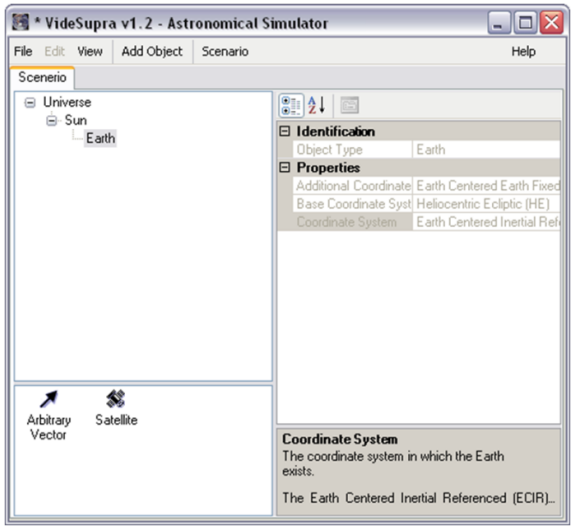
Screen Capture 3



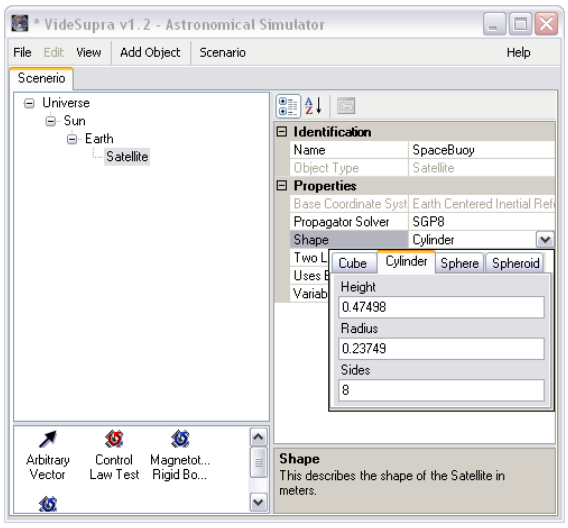
Screen Capture 4

the property description panel. When the astronomical simulation is selected the “Universe” object is initially added. This object will determine how the simulation will be run. There are two simulation modes. The “Real-time” mode and the “Collection” mode. Real-time will run the simulation in relation to the actual time, where the collection mode will calculate a range of time as quickly as it can then the collection can be reviewed and replayed, as many times as needed, at any desired speed, without needing to be recalculated. To test the software simulated firmware sets the “Run Mode”, under the “Run Settings” category, to “Collection” and includes the firmware simulation object that needs to be tested. To test the firmware with HIL set the “Run Mode” to “Real-Time” and includes objects to output the simulated sensor data and input the actuators from the hardware. In Screen Capture 4 the “Delta Time” has been set to 1 second, the start time to midnight on Jan 1, 2007 and the end time one hour later. If the simulator is run with only the “Universe” object the simulation will only show the time stamp for each step of the universe. More objects are needed for more complex simulations. Screen Capture 5 shows the “Sun” object added to the “Universe” object and the “Earth” object added to the “Sun”. The “Sun” and “Earth” objects have no options to it. All the properties are read-only descriptions about the controls the object will add to the simulation.

Screen Capture 6 shows a “Satellite” object added to the “Earth” object. For testing for the SpaceBuoy project set the name of the satellite to “SpaceBuoy”. Set the shape of SpaceBouy to the measurements of the housing, 18.7” right cylinder (0.23749 meters in radius and 0.47498 meters high). Set the “Uses Earth’s Magnetic Field” to “True” the “Variable Write Mode” to “Full”. This configuration will add the controls to create the Sun coordinate systems, Earth’s coordinate systems, and SpaceBuoy’s coordinate systems. Normally only the minimal required controls would be added but by setting the “Variable Write Mode” to “Full” all compatible controls are added. This provides the most amount of simulated and collected data.



Screen Capture 5

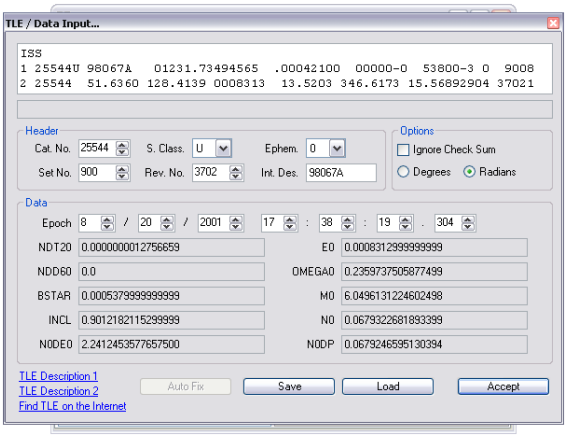


Screen Capture 6

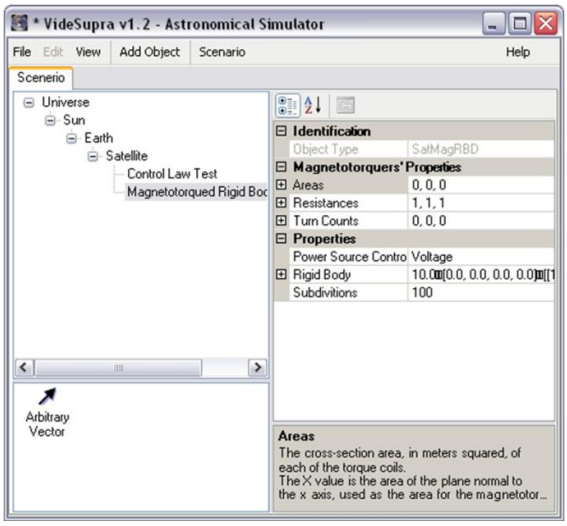
Since the SpaceBuoy project was only in development, like most satellites using VideSupra, no TLE (Two Line Element) was available. SpaceBouy was tested using the ISS orbit for 2001. To change the TLE click on the "Two Line Element" property for the satellite. The form shown in Screen Capture 7 will be brought up. If provided with a TLE from Norad or NASA it can simply be copied to the raw TLE text. Alternately each value can be set individually. Since NASA and Norad's TLE use different checksums both will be tested, Norad first, but checksums can be ignored too. When done click the "Accept" button. To discard changes click the close button in the top right corner.

Add the control law object that should be tested. No control law for the magnetotorquers are included wuth the VideSupra package. The control law object shown isn't included because it contains a proprietary algorithm. For more information about SSEL's control law algorithms read "Control Law Documentation for SSEL-ADCS" by Bryan Ostdiek.

The "Control Law Test" object usually requires a rigid body model with magnetotorquer actuators. Screen Capture 8 show a "Magnetotorquer Rigid Body Dynamics" object added to the satellite along with a control law test. The control law test usually controls the current (amps) of



Screen Capture 7



Screen Capture 8

the magnetotorquers. If it does switch the “Power Source Control” to “Current” and it’ll calculate the voltage and magnetic moment using the current, otherwise set the “Power Source Control” to “Voltage” and the current will be calculated.

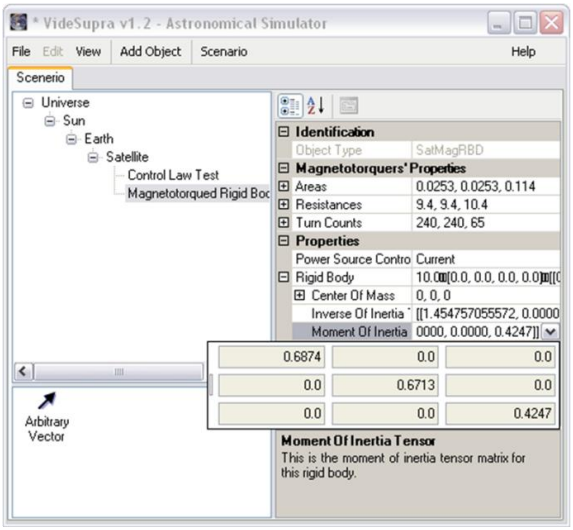
SpaceBuoy has three magnetotorquers on it. Two of them are identical, the square X and Y torque coils, and the third is the circular Z torque coil. To let the simulator know how to use these coils set the areas, resistances, and turn counts for each. Figure 32 shows the values for SpaceBuoy’s setup.

	X	Y	Z
Area	0.0253	0.0253	0.114
Resistance	9.4	9.4	10.4
Turn Count	240	240	65

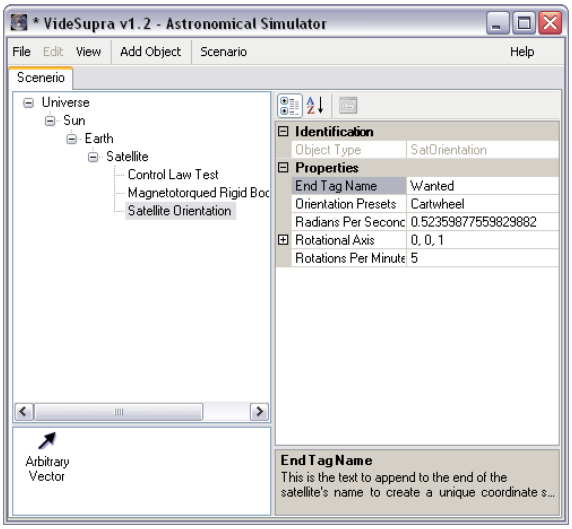
Figure 32: SpaceBuoy Magnetotorquers Value

The rigid body dynamics will simulate the movement of the satellite in frictionless zero-G environment. To do this the mass and moment of inertia tensor matrix must be set. First set the mass, for SpaceBuoy the mass is 29kg. The moment of inertia tensor for SpaceBouy is shown in

Screen Capture 9. If the tensor is set before the mass when the mass is set the tensor will be scaled according to the mass change.



Screen Capture 9



Screen Capture 10

As is, this would test the control law but have no desired data on which to compare the test against. To do this comparison a “Satellite Orientation” object is added to the satellite, shown in Screen Capture 10. This object will calculate an ideal orbit. So that there isn’t a conflict in the controls, set the “End Tag Name” to “Wanted”. This will cause the satellite to create two housing structures as well as other duplicated value, just with “Wanted” in the name of the ideal orientation.

For the SpaceBouy project the first structure would be “SpaceBuoy Housing” and the second structure would be “SpaceBuoy Wanted Housing”. SpaceBuoy was required to fly in a cartwheel orientation at 5 rpms, so the wanted orientation should reflect that requirement.

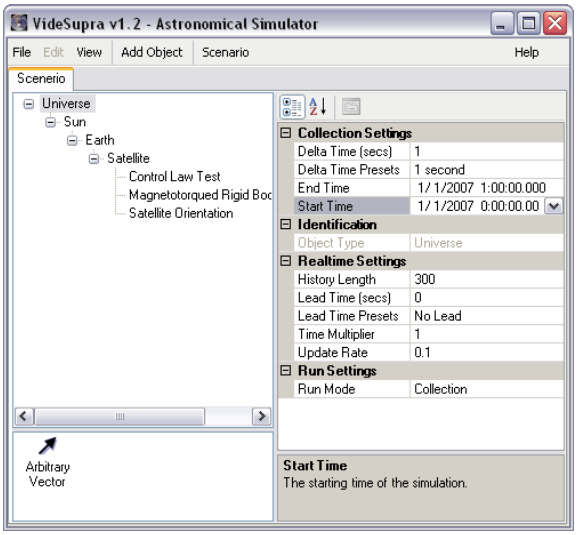
Once the simulation is prepared to test the satellite or satellites, as desired by the requirements, just like with any project, the setup should be saved. VideSupra automatically saves a backup when a simulation is started, but just to be safe save it anyway. If the project isn’t closed when VideSupra is closed that opened project will be open when VideSupra is restarted. Any

changes, saved or unsaved, will also be reopened. Again, it is better to save than to accidently lose the data. The auto-saves are to help protect the project but shouldn't be used as the only means of keeping data. When changes to the data occur an asterisk is shown next to the icon in the form's title bar. When the project is saved the asterisk will be removed. Screen Capture 11 shows the project after being saved.

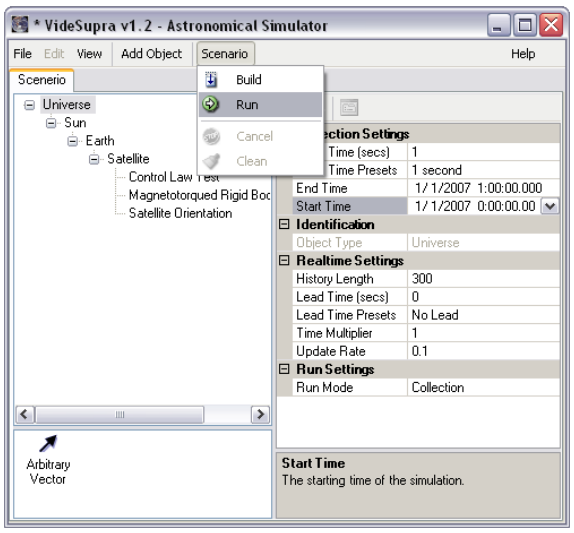
With the setup for the simulator finished the simulator can now be built and run. Click on "Simulator" then "Build" to test the setup and prepare the simulator for running. If any errors occur they will be posted to the console and the console will be shown. If the simulation builds correctly then click "Simulator" then "Run". This will run the simulation. Clicking "Run" will automatically build the simulator unless the simulator has already been built and no changes since that build have been made. If the simulation is run two or more times the simulation will only be created the first time or whenever a change has been made. Build is useful to test the simulation to determine what still needs to be setup without it running the first time the build is successful. Screen Capture 12 shows the project about to be run.

Some simulations may take a long time to build and even a longer time to run. When the simulation is running it will be started in multiple separate threads to help keep the computer responsive whilst completing the simulation as fast as possible. To lower simulation time, close any other programs that use a large amount of memory or CPU power. VideSupra is not distributive across a network in this version. However the separate threads for running the simulation are designed to take advantage of multiple core computers. If the simulation is left to run overnight remember to turn the screen saver to a simple blank screen to keep it from stealing process time from the simulation.

When the simulation starts running in collection mode four new bars are added to the top of VideSupra. The first bar is a list of views that were determined capable of working with the current configuration of the simulation. The other three are the playback controls, the collection



Screen Capture 11



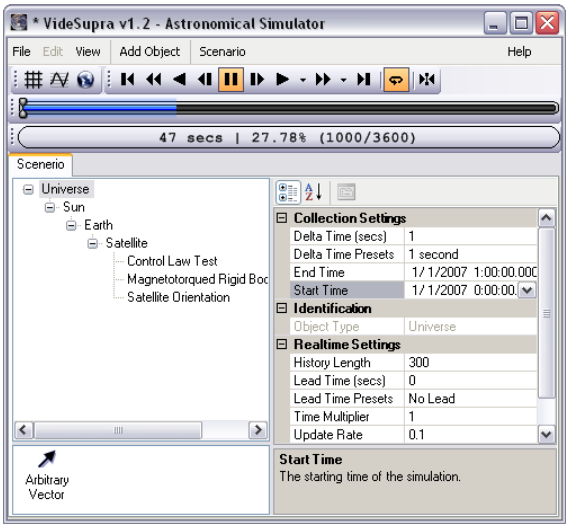
Screen Capture 12

progress bar, and the progress status bar. The progress status will show the amount of results collected from the simulation as well as the estimated time until completion. Although the currently collected results can be reviewed the moment they start being collected it is recommended that the simulation is allowed to finish before reviewing results. The collection has finished when the progress bar turns gray and the status bar shows stops showing the percentage of completion. Screen Capture 13 shows the simulation performing a collection.

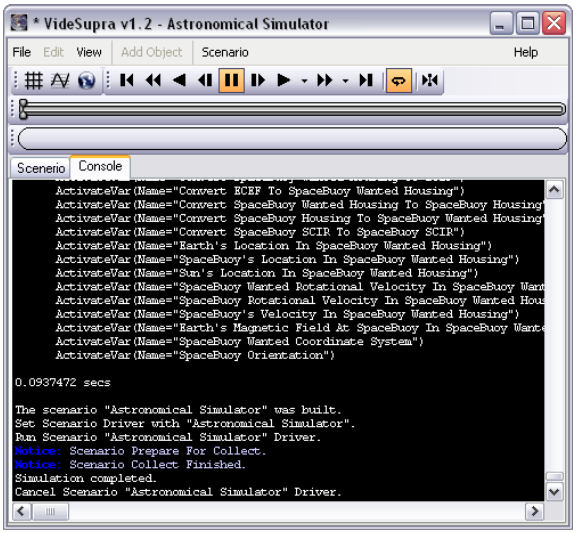
To stop a collection once one has been started click on “Scenario” then “Stop”. The simulation will halt and show the data which was collected before the simulation was stopped.

As mentioned before, if an error occurs while building the error will be posted to the console. This is also true for any error which occurs during the collection process. The console will also show the status of the build and events which have occurred while VideSupra has been running. It is a running history. If the results from the simulation are not as expected use the console to check that all the expected variables were created, that the controls making up the simulation were assigned to the correct order, and that the simulation was assembled correctly. Screen Capture 14 shows what the console would look like after the current simulation has been

built, run and let finish its collection. If the simulation has been running and appears unfinished when it should have been finished, check the console. It will indicate if the simulation crashed, had an error, and even if a user had clicked stop. Any well known error with a known cause will also contain a suggestion to correct for this error or a tip to keep it from occurring again. In the rare case that VideSupra has crashed or is not running in the morning, all the text from the console will have been written to DebugResults.txt in the same path as the executable. To view the console click on “View” then on “Console”. To close the console click on “View” then on “Console” again.



Screen Capture 13



Screen Capture 14

Figure 33: Playback Control Bar

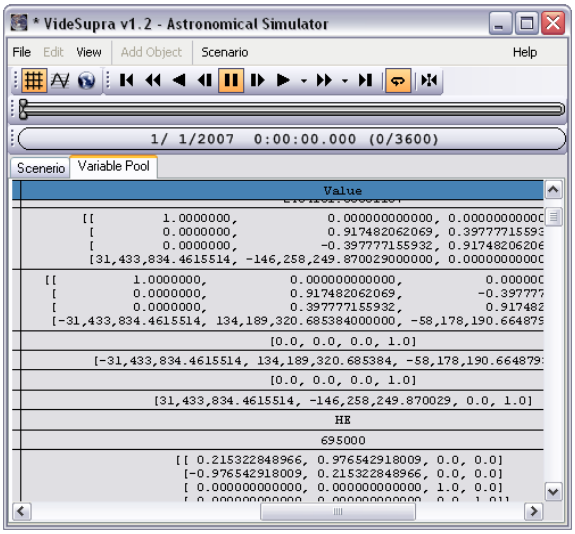
To review the data use the playback controls shown in Figure 33. The controls from left to right are jump to start, rewind, play in reverse, step backwards, pause, step forward, play, fast forward, jump to end, continues play, and go to. Jump to start will go to the first time frame (variable pool) loaded, similarly jump to end will go to the last. Step back and forward will go to

the previous or next time frame from the current one. Play will step forward through the data at a specific frames per second speed. The down arrow to the left of the play button selects the amount of simulated time to pass in actual time. For example, if the play back is one second per second then the data will play back in actual time but with one minute per second the data will play back 60 times faster than actual time. The play in reverse will run at the same speed as play but backwards. Fast forward and rewind review the data at a magnitude of the play speed. The multiplier can be selected by the drop down arrow to the right of the fast forward button. The continues play button will make the playback jump back to the first frame when it reaches the end. Similarly when playing backward it will jump to the end when it reaches the start. The go to button brings up a control where a time can be entered in and the closest time frame is jumped to.

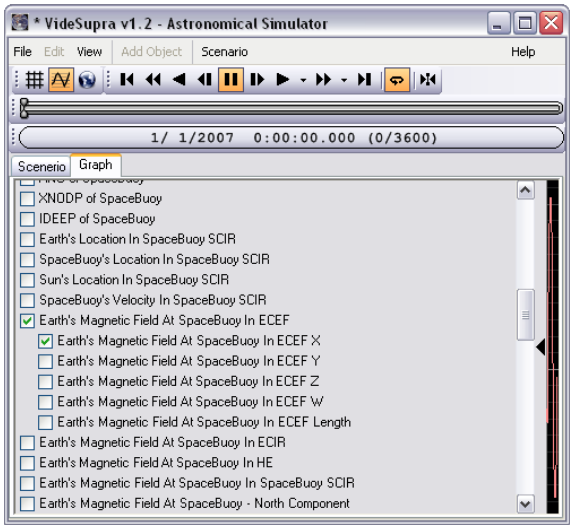
The views which have been placed into the view bar must meet a minimum set of requirements provided by the created simulation. That way only the views which will be able to show anything will be added to the list. Click the view's icon to show or hide the view. When the view is clicked on a new bar is added to VideSupra to provide tools for that view (not shown in the Screen Captures).

One view which will always pass the set of requirements is the variable pool view. As mentioned before the variable pool is how the simulator stores all the values it worked on and collected during the simulation. As the playback slider is moved or the simulation is played the variable pool can be viewed for the currently selected time frame. This variable pool view is a grid which displays the name of the variable, the type, the value at the given time, the units and some other information. For a detailed description, including the attributes associated with the variable, just hover the mouse over a variable's row and the description will popup. The variable pool view provides a toolbar to output the pool in a comma separated values (csv) file which can then be opened by a data grid tool such as Excel or Open Office Calc. The variable pool also provides a way to sort the pool by name, type, order of creation and a few other ways. Screen

Capture 15 shows the variable pool.



Screen Capture 15

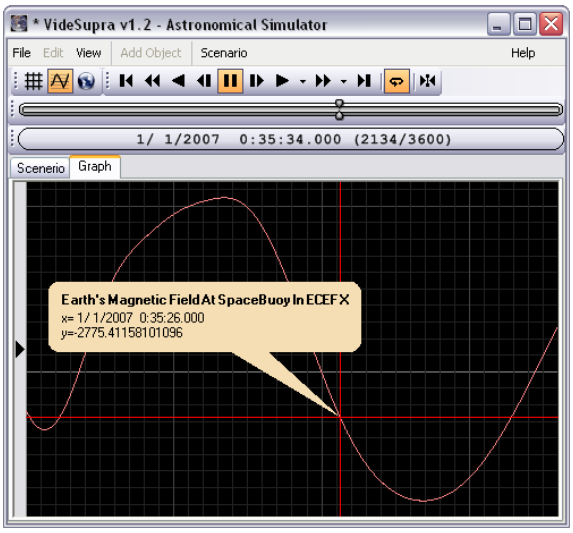


Screen Capture 16

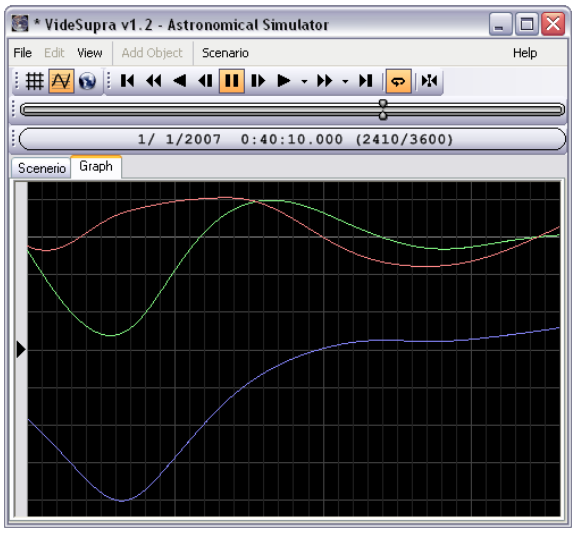
Another view which will typically be shown is the graph view. To be shown the simulation must have a double, float, vector, matrix, mass, or a quaternion variable. All practical simulations will have at least one. The graph view will create a list of variables which can be graphed relative to time. Screen Capture 16 shows the list of variables that can be graphed for the current simulation. When a multiple component variable, such as a vector, matrix, mass, or quaternion, is selected a sub-list is shown with the components exposed for selection. Click on the divider for the variable selection to resize or hide it. Screen Capture 17 shows a variable selected and the selection hidden. When the user clicks on the graph the time frame will be moved to the nearest frame and a bubble will popup showing the value of the variable at that time frame. The scroll wheels can be used to zoom in and out on a portion of the graph. Use shift and control modifier keys with the scroll wheel to compress or expand the graph in only one dimension. Use click and drag with the right mouse button to shift the graph around. The graph can show many variables at the same time as seen in Screen Capture 18. For the best results select variables with the same units and similar ranges. The variables are automatically colored to help

distinguish between the different variables.

As with the variable pool view, a bar is added when the graph view is shown. This bar provides options to save the image of the graph or to export the selected variables into a comma separated value (csv) file for further processing.

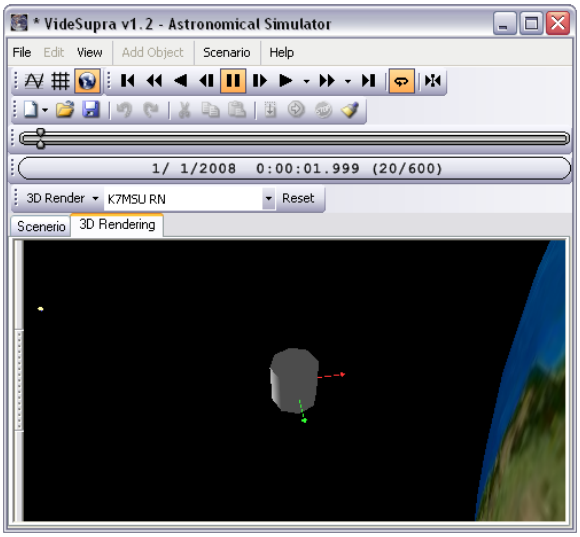


Screen Capture 17

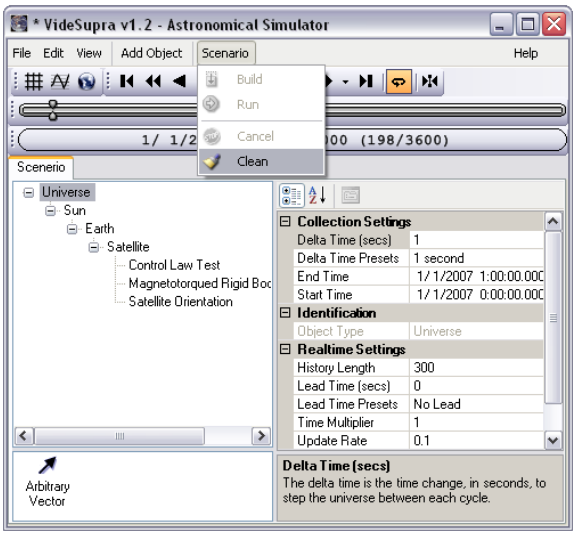


Screen Capture 18

The 3D render view, written with OpenGL, is only available when the simulation contains render tagged controls, such as the Sun, Earth or a satellite, or any matrix, vector or point. This view will provide a visual representation of the 3D data. Select a base coordinate system from the drop down menu. During play back the camera will stay relative to the base coordinate system so that relevant conclusions about the data can be drawn. Like the graph view, the 3D render view provides a variable selection which allows components to be turned on and off. The difference is that the 3D render view's selection will provide different rendering modes and presentation modes. Polar and Cartesian grids can be turned on and off. Click and drag the mouse with different mouse button combinations to rotate the camera around the base coordinate system. Use the mouse wheel to zoom in and out on the base coordinate system's center point. Screen Capture 19 shows the 3D render looking at a satellite with both texturing and lighting on.



Screen Capture 19



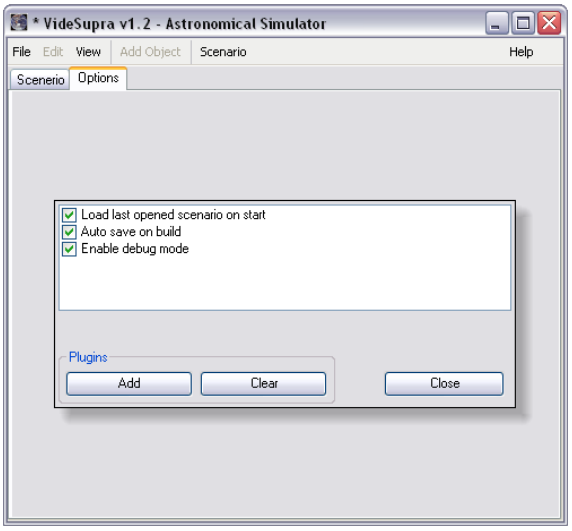
Screen Capture 20

When done viewing and reviewing the results of the collection click Scenario then Clean. This will remove the data and immediately free up the memory it used. If clean isn't clicked it might take a minute or two (depending on the computer) after starting a new collection or project before the data from the previous one has been released. To close the current project and start a new one click "File" then "New". To just close the project click "File" then "Close". Screen Capture 20 shows the clean menu for freeing the data and resetting the simulator.

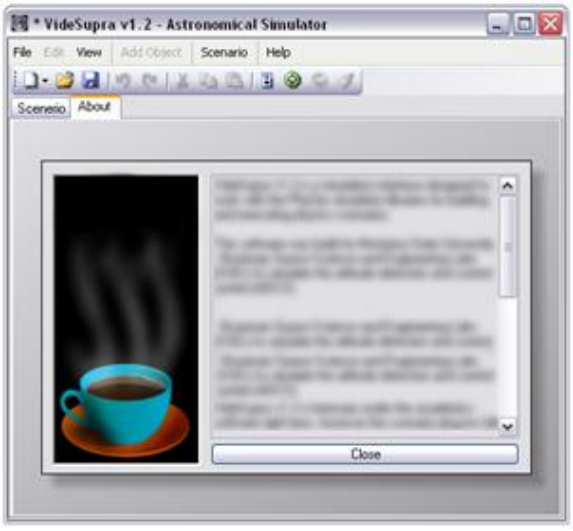
To view the options for VideSupra click on "View" then on "Options". From the options page plug-ins which weren't automatically loaded can be added or all the plug-ins can be removed. The options also provide some customizations on how it loads and saves projects. While developing a plug-in for VideSupra, run VideSupra with the argument "-dev" after the executable name (VideSupra.exe –dev). This will show the "Enable debug mode" option. By default VideSupra will attempt to catch and gracefully handle errors in plug-ins as well as in the main code. When the debug mode is enabled VideSupra will not catch any errors which allows the IDE to trace to the line of code which caused the error. VideSupra was developed with VisualStudio .Net 2005 then later upgraded to VS 2008 and VS 2010. It is recommended that all

plug-ins are developed in VS 2010 or newer. Screen Capture 21 shows the options page.

To view the “About” information or go to the website for more help information click on the “Help” menu. Based on the plug-ins available more or less menus, pages, views and controls will be available.



Screen Capture 21



Screen Capture 22

CONCLUSION

VideSupra is an extendable physics simulation tool which is provided with a physics simulation plug-in. It is designed to help development and testing of satellites. VideSupra isn't the only software available for satellite simulation, but can be used in design, development, and testing both simulated and HIL. VideSupra can be used for tracking and assistance during a satellite flight and it is completely open-source and free. This makes it the ideal tool for any small group satellite development team, such as colleges and individual usage. Without VideSupra these small groups will be paying more than they can afford for testing utilities, tracking and possible loss of poorly tested satellites. VideSupra can be used for other projects as well with no limitations on the type of simulations that can be added as plug-ins. It supports fast development methods with modern programming languages to provide the most amount of control and data per simulation.

Future development on VideSupra could include better attribute tagging of variables, a unit conversion tool that would auto-generate unit conversions similar to how the auto-generator for the coordinate conversions currently works, and an online repository of custom controls created and shared by the ever growing small satellite development community. VideSupra is designed for advanced dockable tab pages where multiple views can be seen at the same time. This feature is shut off because the 3D graphics interferes with the MDI (Microsoft's Multiple Document Interface) the dockable tab page controller is built on. In future versions this interference needs to be eliminated so that dockable pages and 3D graphics can cooperate.

REFERENCES CITED

[1] VideSupra Design, Development, and Programming By:
Grant Nelson
Version 1.0.0.0, July 2007 – March 2008
Version 1.2.0.0, May 2008 – October 2010
<http://www.snowgremlin.com/main.php?d=proj_vs>
Email: grantnelson1@gmail.com
Montana State University
Space Science and Engineering Labs (SSEL)

Code written by Brian Ostdiek, Sam Gardner, and Chad Bohannon has been removed from the released copy of VideSupra in cooperation with the copyrights and restrictions of the usage of their algorithms and source code.

[2] Bate, Roger R.; Mueller, Donald D.; White, Jerry E. "Fundamentals of Astrodynamics"
New York, New York: Dover, 1971
ISBN 0-486-60061-0, Library of Congress Cat. No: 73-157430

[3] Dunn, Fletcher; Parberry, Ian "3D Math Primer, for Graphics and Game Development"
Plano, Texas: Wordware Publishing, Inc., 2002
ISBN 1-55622-911-9

[4] Hughes, Peter C. "Spacecraft attitude dynamics"
New York, New York: Dover, 2004
ISBN 0-486-43925-9 (pbk.)

[5] Kuipers, Jack B. "Quaternions and rotation sequences"
Princeton, New Jersey: Princeton University Press, 2002
ISBN 0-691-10298-8 (pbk.)

[6] Meeus, Jean "Astronomical Algorithms" Second English Edition.
Richmond, Virginia: Willmann-Bell, Inc., 1998
ISBN 0-943396-61-1

[7] Larson, Wiley J.; Wertz, James R. "Space Mission Analysis and Design" Third Edition.
Portland, Oregon: Microcosm Press, 1999
ISBN: 978-1881883104

[8] Applied Physics & Physics Engines:
Euclidean Space by Martin Baker
<http://www.euclideanspace.com/>

[9] Rigid Body Dynamics From:
ODE, Copyright (C) 2001-2004, By Russell L. Smith

[10] Applied Mass and Inertial Momentum Tensor:
Opensource Dynamics Engine (ODE)
ODE is Copyright © 2001-2004 Russell L. Smith. All rights reserved.
<http://www.ode.org/>
<http://odedotnet.sourceforge.net/index.php/Main_Page>

[11] World Magnetic Model (WMM)

Software and Model Support:
National Geophysical Data Center
NOAA EGC/2
325 Broadway
Boulder, CO 80303 USA
Attn: Susan McLean or Stefan Maus
Phone: (303)497-6478 or -6522
Email: Susan.McLean@noaa.gov or Stefan.Maus@noaa.gov
Web: <http://www.ngdc.noaa.gov/seg/WMM>

Sponsoring Government Agency:
National Geospatial-Intelligence Agency
PRG/CSAT, M.S. L-41
3838 Vogel Road
Arnold, MO 63010
Attn: Craig Rollins
Phone: (314)263-4186
Email: Craig.M.Rollings@Nga.Mil

Original Program By:
Dr. John Quinn
Fleet Products Division, Code N342
Naval Oceanographic Office (NAVOCEANO)
Stennis Space Center (SSC), MS 39522-5001

Code Maintained By:
Stefan Maus
Version 2.0, Sept 2005

[12] GeoMag.Net v2.1 of WMM:
<http://www.snowgremlin.com/main.php?d=proj_vs>
Code Rewrite and Conversion From C To C# By:
Grant Nelson
Version 2.1, July 2007
Email: grantnelson1@gmail.com
Montana State University
Space Science and Engineering Labs (SSEL)

[13] Geomag Paper:
McLean, S., S. Macmillan, S. Maus, V. Lesur, A. Thomson, and D. Dater, December 2004, The US/UK World Magnetic Model for 2005-2010, NOAA Technical Report NESDIS/NGDC-1.

[14] Geomagnetism:
National Geophysical Data Center (NGDC)
NOAA Satellite and Information service
<http://www.ngdc.noaa.gov/seg/geomag/>

[15] Satellite Coordinate Systems:
Penn State, College of Engineering, Surveying
<http://surveying.wb. psu.edu/sur351/SatCoords/SatCoords.html>

[16] Spacetrack Report No. 3
Models for Propagation of NORAD Element Sets
Felix R. Hoots and Ronald L. Roehrich
December 1980
Package Compiled by TS Kelso 31 December 1988

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED
Requests for additional copies by agencies of the Department of Defense, their contractors, and other government agencies should be directed to the:
Defense Documentation Center
Cameron Station
Alexandria VA 22314

All other persons and organizations should apply to the:
Department of Commerce
National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Address special inquiries to:
Project Spacetrack Reports
Office of Astrodynamics
Aerospace Defense Center
ADC/DO6
Peterson AFB CO 80914
SGP, was developed by HILon & Kuhlman (1966)
Uses a simplification of the work of Kozai (1959) for its gravitational model

SGP4, was developed by Ken Cranford in 1970 (Lane and Hoots 1979)
Obtained by simplification of the more extensive analytical theory of Lane and Cranford (1969) which uses the solution of Brouwer (1959) for its gravitational model and a power density function for its atmospheric model (Lane, et al. 1962)

SDP4, is an extension of SGP4 to be used for deep-space satellites developed by Hujsak (1979)

SGP8 model is used for near-Earth satellites and is obtained by simplification of an extensive analytical theory of Hoots 1980.

SDP8 model is an extension of SGP8 to be used for deep-space satellites.

[17] SpaceTrack.Net:
<http://www.snowgremlin.com/main.php?d=proj_vs>
Code Rewrite and Conversion From FORTRAN IV To C# By:
Grant Nelson
Version 2.0, October 2006
Email: grantnelson1@gmail.com
Montana State University - Bozeman
Space Science and Engineering Labs (SSEL)

[18] TLE Description:
CelesTrak
Center for Space Standards & Innovation (CSSI)
<http://celestrak.com/NORAD/documentation/tle-fmt.asp>