A HYBRID HARDWARE & SOFTWARE SIMULATION ENVIRONMENT FOR RELATIVE ORBIT MOTION STUDIES

by

DANIEL DUNN

B.S., Auburn University, 2006

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Aerospace Engineering Sciences

2011

This thesis entitled:
A Hybrid Hardware & Software Simulation Environment for Relative Orbit Motion Studies
written by Daniel Dunn
has been approved for the Department of Aerospace Engineering Sciences

Hanspeter Schaub

Prof. George Born

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Dunn, Daniel (M.S., Aerospace Engineering)

A Hybrid Hardware & Software Simulation Environment for Relative Orbit Motion Studies

Thesis directed by Prof. Hanspeter Schaub

Abstract:

Relative orbital motion is a topic of interest to programs developing vehicles for formation flight applications such as remote inspection or autonomous rendezvous and docking. High-accuracy laboratory systems for such simulations can be costly to operate, requiring expensive hardware and extensive development for a specific simulation. This thesis details the development of a framework at the Autonomous Vehicle Systems Laboratory (AVSLab) at the University of Colorado at Boulder which utilizes relatively low-cost commercial hardware for medium-fidelity low-cost relative orbit simulation. The simulation framework provides a modular approach to building simulations along with tools to reduce the required simulation development effort. The framework integrates existing software modules to control a wheeled robotic vehicle and two-axis camera mount to simulate space vehicles in near-planar motion, and provides emulation of hardware, including hardware for which no previous virtual equivalent was available, in a hybrid real/virtual environment for faster simulation and scalability to larger formations. A demonstration simulation of 3D visual tracking and relative orbit propagation and navigation around a target is created to showcase the capabilities of the framework and typical usage. Results from the simulation are presented, showing the performance of implementing the simulated physics as well as providing insight into the performance of the simulated satellite's control algorithm.

DEDICATION & THANKS

To Dr. Schaub, for all the guidance laid out.

To Ann Brookover, who knows everything that counts.

To friends and co-workers, for patience amid madness.

To Amber, who knows more about polishing than I ever will.

To Mom and Dad, for the endless support.

To Kristen, for believing it was worth the wait.

To God, for everything else and then some.

ACKNOWLEDGEMENTS

This work is based upon work supported by Sandia National Laboratories. Special thanks to the UMBRA team for their continued technical support.

This thesis was supported by funding from the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program.

CONTENTS

TABLES

FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Relative orbital motion, owing to the construction in a non-inertial frame, introduces a number of complications for vehicles and systems attempting to navigate in close proximity. The cyclical motions of one vehicle relative to another make safe rendezvous & docking maneuvers difficult, as a loss of power has the potential to place the vehicles on a collision course, requiring odd approach paths to ensure safety. Maneuvering thrusters compound the issues, as their mass expulsion plumes can adversely affect the other vehicles, leaving deposits on surfaces or damaging components.

Systems designed to operate under such limitations must often operate in non-intuitive ways, making interaction and control more difficult. Given the present costs involved in manned operations, automation is an obvious avenue for augmenting human presences in space. Maintaining robust responses while pushing toward greater autonomous operation necessitates a great deal of testing in order to ensure the safety of the vehicles and potentially their occupants. Such testing often comes at great expense with precision hardware, slowing advancement in design.

The Orbital Motion & Control Simulation (OMCS) group of the Autonomous Vehicle Systems (AVS) Laboratory is primarily concerned with developing the capacity to fill this gap using relatively inexpensive hardware and software. The hardware allows for validation of the robustness of the control schemes under realistic conditions, while the software permits simulations in regimes that would be difficult to conduct using real hardware.

1.2 Potential Applications

Any application that involves on-orbit proximity maneuvers has the potential to benefit from reduced-cost simulation capabilities. Some potential applications include remote sensing, autonomous rendezvous & docking, external manipulation & inspection, and increased productivity.

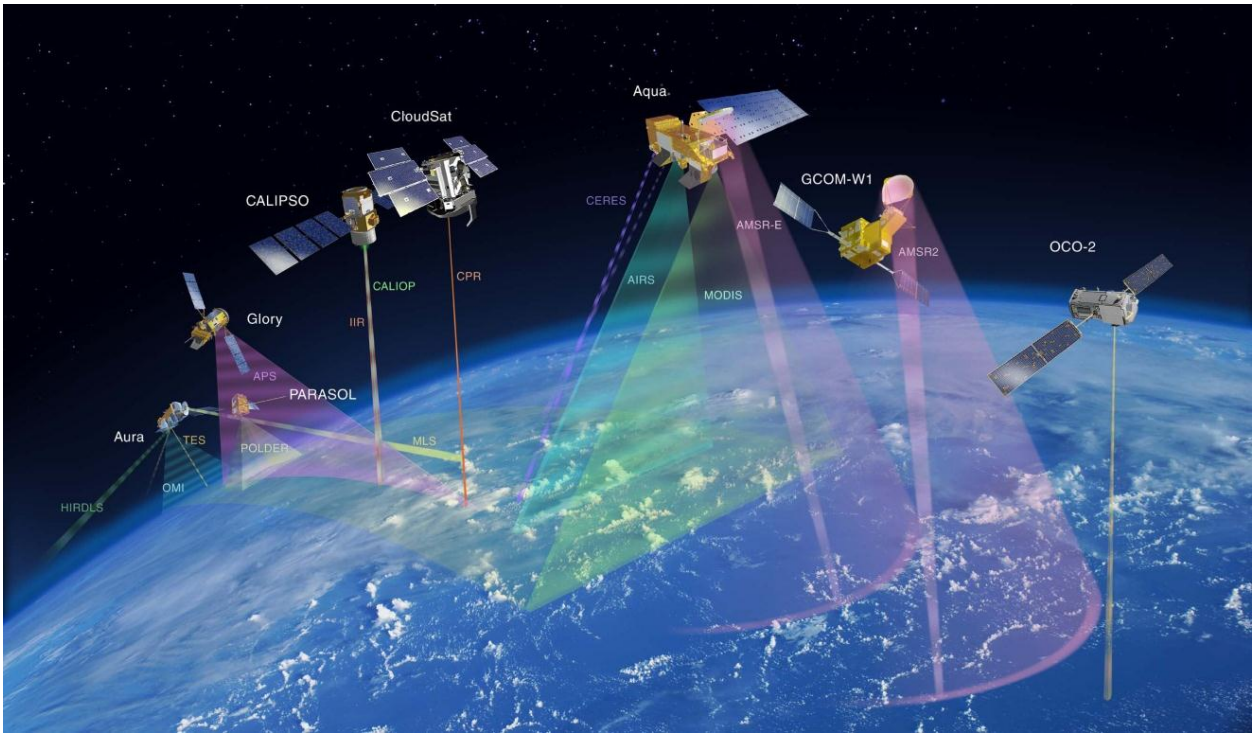

Figure 1.1: Formation of satellites engaged in combined remote sensing

Formations of spacecraft can be ideal for remote sensing applications, an example of which is depicted in figure 1.1[1] , the 'A-Train' group. [1] Deploying multiple satellites for sensing has potential benefits over a single satellite in providing physical separation of instruments, such as would be required for laser interferometry or ultra-precise gravitational measurements. In addition to technical benefits, there are programmatic ones as well, including redundancy and reduced program startup costs through incremental rollout.

[1] `http://atrain.nasa.gov/images/A-TrainEOS_wv.png`

Figure 1.2: Orbital Express autonomous rendezvous, docking, and service system

Costliness of environmental support systems and mass make supporting automation for manned operations mission-critical. Autonomous rendezvous and docking procedures are becoming more commonplace than their fully-manned counterparts, with unmanned resupply ships launched to the international space station routinely. Figure 1.2[2] shows an artist rendering of the Orbital Express autonomous rendezvous, docking, and service system. In light of the Columbia tragedy, additional emphasis has been placed on the ability to rapidly externally inspect and assess vehicle health, a task well-suited to readily-deployed external craft.

[2] http://sm.mdacorporation.com/images/what_we_do/OE_high.jpg

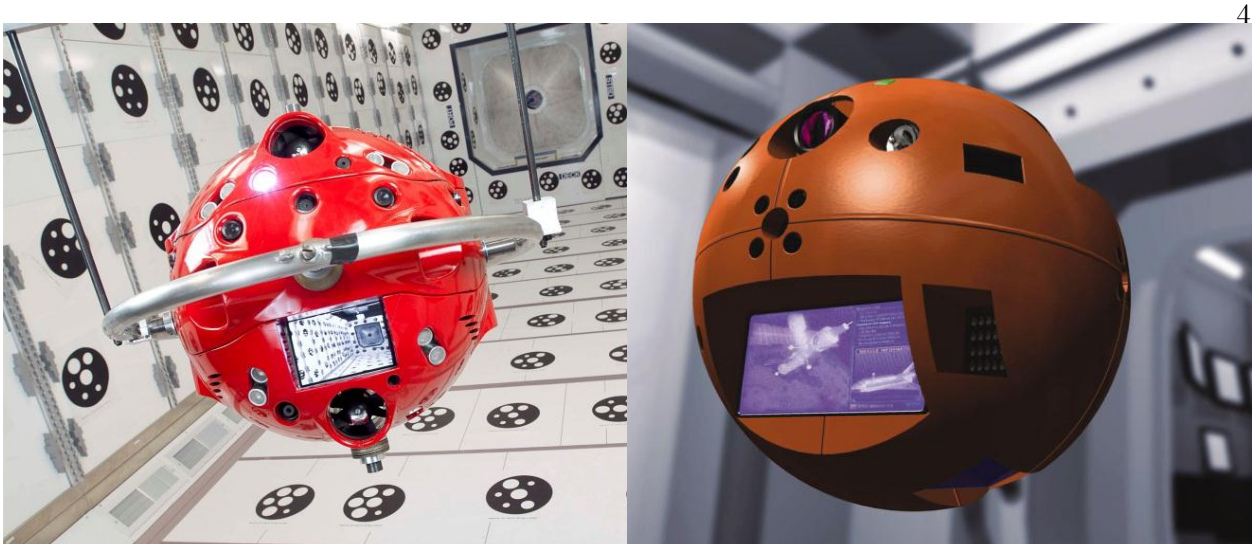

Figure 1.3: Personal assistant for on-orbit activities

Although human presence is capable of greatly expediting a scientific mission, man-hours on orbit are always in short supply. Autonomous vehicles can multiply their productivity if applied properly. One example under development is the Personal Satellite Assistant, shown in figure 1.3[3] [4] , designed to automate a number of tasks which would ordinarily require the presence of a human. The vehicle as-designed makes extensive use of visual navigation for the interior of the international space station.

1.3 Literature Review

A number of existing facilities are currently capable of conducting relative orbital simulation with a widely varying range of limitations, purposes, and accuracies. The favored type of device for these facilities is the air-bearing, which provides a near-frictionless movement capability in two dimensions. [2] These types of simulators often combine flat-surface air bearings with rotational joints which permit attitude simulation in two or three degrees of freedom.

Larger scale facilities are usually operated by government or industrial organizations, such as the Flight Robotics Laboratory at NASA's Marshall Space Flight Center. These facilities are often

[3] http://i.cmpnet.com/ddj/sdmagazine/images/sdm0309a/sdm0309a_2_lg.jpg
[4] http://www.nasa.gov/centers/ames/images/content/77419main_ACD04-0106-002.jpg

used for near full-scale testing of dynamics of orbital maneuvers, such as docking and rendezvous operations. [3] However, larger facilities are often tremendously expensive to operate, and cannot react quickly to accept requests for simulations from outside entities, with priority going to the owning organization's activities.

Smaller-scale simulation facilities have become more abundant in recent years. [2] They are often funded by organizations with much smaller budgets, and permit a somewhat faster testing cycle for simulations due to easier setup associated with objects that can be readily manipulated manually. The Aerospace Robotics Laboratory at Standford University is a good example, having conducted research into a number of orbital maneuvering problems such as inexpensive decommissioning of satellites [4] and robust visual navigation. [5]

Although air bearings are favored for their properties force-reaction, the lower cost of wheeled vehicles makes them an attractive alternative. The Vehicle Systems and Control Laboratory at Texas A&M has developed a control algorithm to create translational bases atop caster wheels. [6]. These have a high accuracy, and and an external system is used to make objective measurements of the craft. However, the measurement system, though less expensive than other air-bearing facilities of similar accuracy, is still a significant barrier to organizations with small budgets. It also is limited in ability to mix simulated and real components.

In contrast, the system described in this thesis, is designed to permit much lower cost simulation capability by using consumer-grade hardware at the expense of extremely high accuracy. The goal is to create a system which can be used for validation of control theory and implementation during the development phase of a project, reducing the amount of costlier high-accuracy simulations required.

The primary type of control system presently being investigated is visual navigation controls. Visual navigation controls have a number of applications, as indicated in section 1.2. The growing ubiquity and rapidly increasing capacity of visual sensing hardware, combined with dramatically shrinking costs, make them an attractive option that can potentially lead to more affordable missions. Visual navigation systems are already under development by a number of entities. An

example is NASA's NGAVGS sensor being developed primarily to facilitate autonomous rendezvous and docking. [7]

The design of the system described in this thesis builds upon, incorporates, and generalizes a set of related projects built by previous faculty of the laboratory. Earlier work by Dr. Hanspeter Schaub led to the development of robust statistical color pressure snake ('visual snakes') algorithms for identifying targets in a video. [8] Subsequent work was done at the first AVS Laboratory at Virginia Polytechnic Institute. From a desire to expand upon and test this work, Master's student Mark Monda integrated a commercially-available education robotics platform with a two degree-of-freedom rotational mount, creating the ability to navigate using the visual snakes algorithm. [9] At the same time, to alleviate some of the difficulties associated with software development on full hardware, Master's student Christopher Romanelli created a virtual version of the robotic platform, along with an relative orbital motion simulation. [10] As Monda and Romanelli's work was wrapping up, Master's student Donald Shrewsbury developed an early formulation of a control law that isolates the rotation of the rotational mount from the rotation of the robotic platform along with a provisional implementation, allowing the vehicle to move with limited independence in heading. [11] Soon thereafter, the AVS Laboratory was moved to the University of Colorado, and work continued on various tangentially-related and other projects. Although the works of Monda, Romanelli, and Shewsbury were developed in tandem, they were still somewhat isolated and targeted rather specifically at desired end simulation products. This creates the motivation to generalize the work into a more modular, reusable, and complete system with greater applicability and scope.

CHAPTER 2

SIMULATION FRAMEWORK

2.1 Concept

The concept of the simulation framework is an integrated workspace containing all of the tools necessary to quickly assemble and test a simulation by combining new and prior work, encouraging code re-use and cutting development time. The framework is designed to tailor the UMBRA software package to the usage scenario of the laboratory, while remaining sufficiently generalized to be extensible for whatever purposes the simulations require. It encompasses a number of practical aspects, including simulation tools, coherent mathematical environment, user interface definition/creation, system hooks, and organization.

2.2 Overview of UMBRA

UMBRA is the toolkit and framework upon which the AVSLab Framework builds. UMBRA is a system of C++ meta-typing, dependency resolution, and bindings to the Tcl scripting language which allow one to build C++ classes called "modules" and dynamically join and manipulate them at runtime. The modules, if written sufficiently generically, can be joined together as building blocks to make progressively larger and more complex simulations. The modules are joined to one another with connections, which are one-to-many relationships that share values of pre-defined types. Feedback connections permit resolution of circular dependencies by retaining the value from the previous time-step, and UMBRA automatically infers the proper order of update execution for each module's connections.

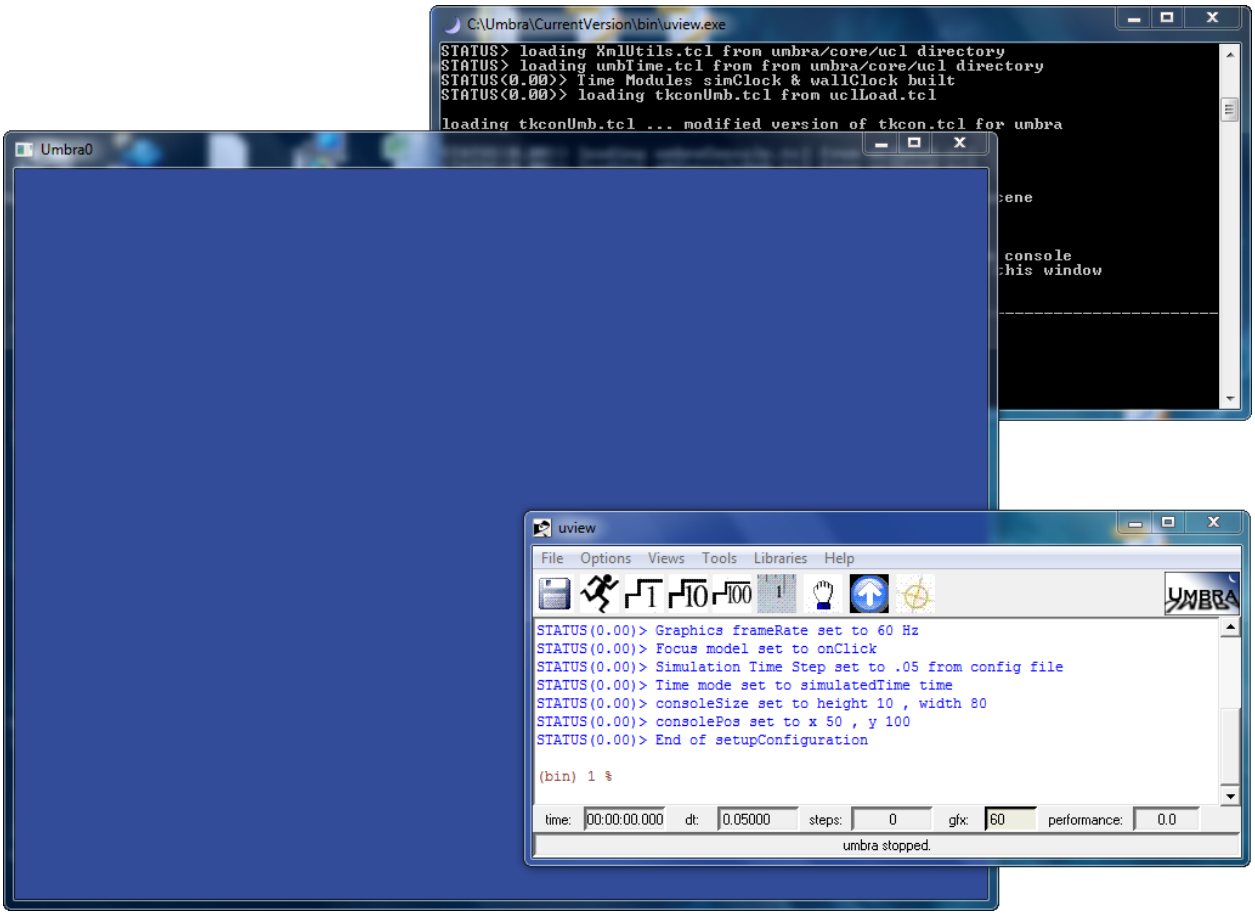

Figure 2.1: Default UMBRA user interface

In addition to modules, UMBRA also provides 3D visualization capabilities using OpenGL and the OpenSceneGraph toolkit. It uses the Tcl scripting language to provide interactive, dynamic control of the connected modules of the simulation, and the Tk widget system to provide custom user interfaces. UMBRA also has a patented 'multiple world' system for modeling interactions in realms not normally covered by physics simulations.

2.3 Design

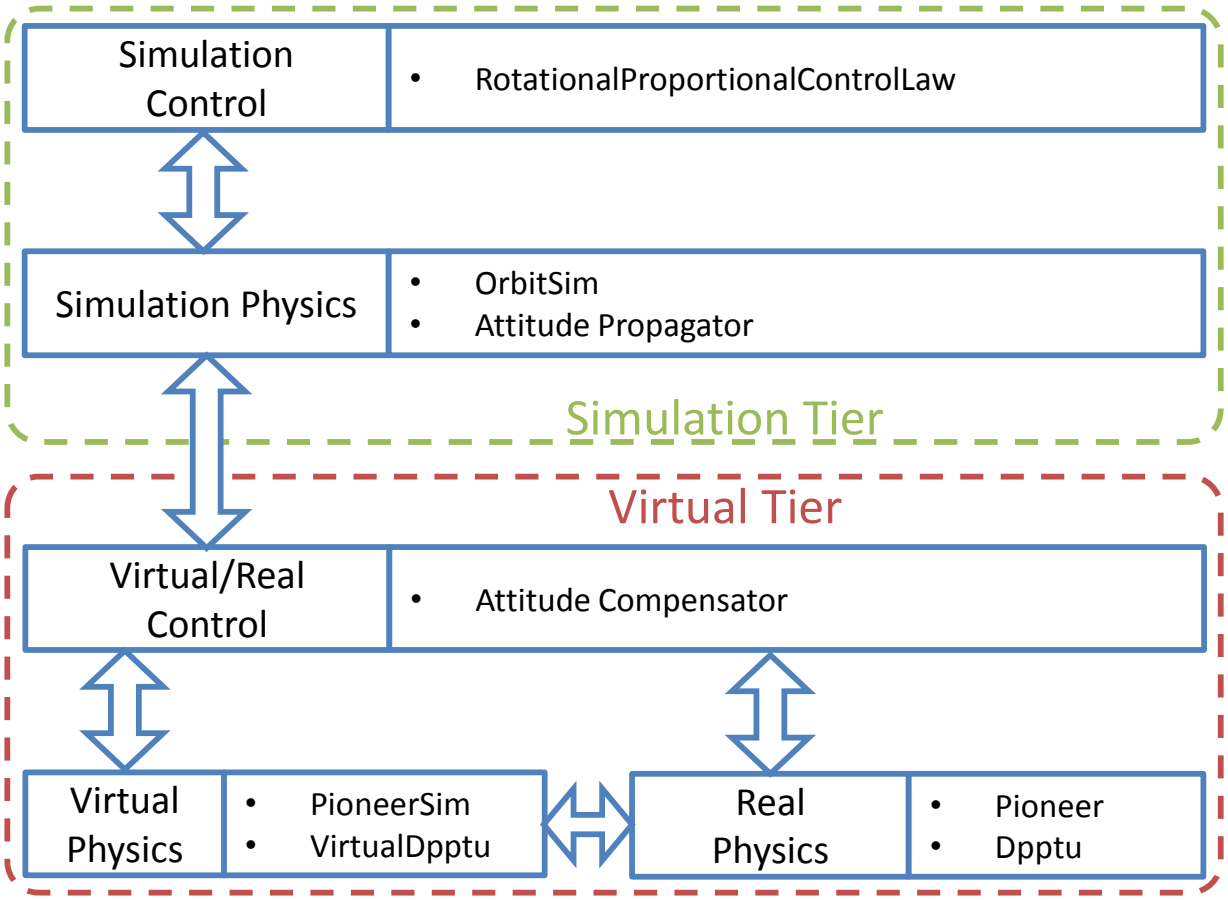

Figure 2.2: Block diagram illustrating the generalized strata of the nested simulation paradigm

The simulation framework is host to two nested simulations, one within the other. This layering approach is illustrated in Figure 2.2. The hybrid virtual/real environment is the first tier of simulation, in which virtual hardware components are simulated in software and intermixed with real hardware. This can manifest in a number of ways. Virtual hardware may be mounted atop real hardware, virtual signals may be imposed over real ones, real hardware may lead virtual hardware. The only limitations to free interchangeability are physical mounting considerations. (For example, it would be impossible to mount a real camera on top of a virtual pan-tilt unit and actually cause the camera to move without further intervention.)

The second tier of simulation is carried out by the hybrid hardware layer, and is host to the more generalized research mission of the laboratory. In the second tier, the hardware represents the translational and rotational motion of on-orbit hardware, or other roles, depending on the purpose of the simulation. Typically, the virtual layer's function is to implement the physics of the simulation layer.

The dual simulation layers can give rise to confusion, necessitating a specific nomenclature for each tier, which is also observed in this document. The simulations of the hybrid hardware layer are known as 'virtual', while the mission simulations of the second tier are called 'simulations'. That is, the use of the word simulation is restricted from the general meaning to refer only to the research simulations that the hybrid virtual/real hardware environment enables.

Both tiers of the simulation may be separated into physics modeling or implementing components, and control components. Components are not packaged or sorted by tier, and instances of a given component may serve more than one role in both tiers. Where a component exists in this paradigm is a function of its role. Understanding where a component exists in this layering is a key step to understanding what data should be available to it. Typically, simulation components are more restricted in what data they may access, while virtual components may use almost any data necessary to implement the physics of the simulation layer.

Objects in the Tcl scripts are created in a form called 'bundling'. A 'bundle' is a group of variables, functions, and named modules, all of which share the same naming structure. The naming structure is period-delimited, e.g. `Object.SubObject.Component`. While expansions to the basic Tcl packages do support a full object-oriented system, the static names used by UMBRA to bind Tcl scripts to C++ modules do not readily adapt to such a paradigm, and would necessitate using arbitrary strings as pointers for the modules. A pointer-based system for handling module bindings would hamper developers in querying the modules at the command line, leading to the decision to use bundles as a lightweight alternative to full object-oriented design.

2.3.1 Frames

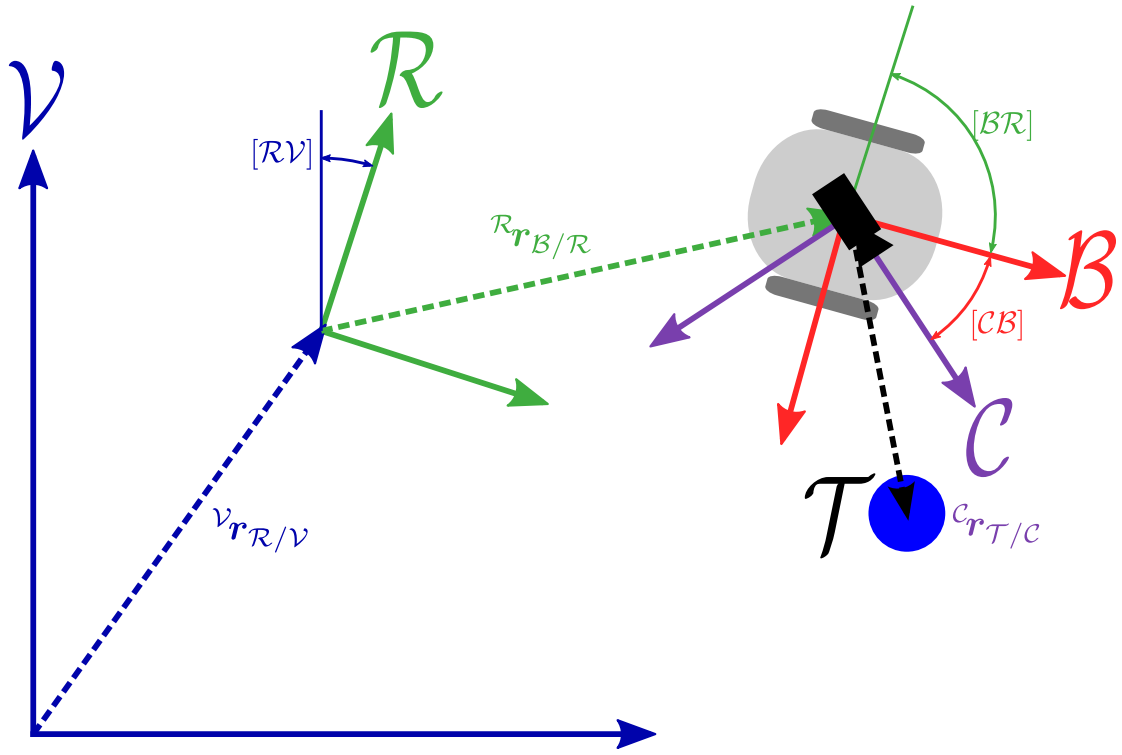

Figure 2.3: A series of sequentially-defined frames

Figure 2.3 shows a typical example of frames as they would be found in a visual tracking and control simulation, with a two-wheeled robotic vehicle and top-mounted camera tracking a target. Frames in the simulation framework are grouped by the naming convention `Frames.FrameName`. Two of the frames depicted are standard parts of the framework, and are created by default in any simulation:

The Virtual Frame ($\mathcal{V}$, `Frames.Virtual`) is the root frame for the entire framework and hybrid environment. Every other frame should be defined, directly or indirectly, against the Virtual Frame, solving inverse transformations if necessary. Positions and rotations with respect to the Virtual Frame are the default representation, and for this reason, the Virtual frame should never

be moved, so that it will always be both inertial and co-incident with any implicit coordinate systems, such as those used for the OpenGL graphics.

The Real Frame ($\mathcal{R}$, `Frames.Real`) is the second automatically-created frame, representing the chosen origin within the laboratory setting. The real frame functions similarly to the virtual frame for real hardware. However, the real frame is subordinate to the virtual frame, which allows the laboratory to be placed at convenience with respect to the simulation, as indicated by the arbitrary position vector $\boldsymbol{r}_{\mathcal{R}/\mathcal{V}}$ and the rotation matrix $[\mathcal{RV}]$. The Real frame may move if needed, and is not necessarily unique. Later expansions of the scope of simulations to networked laboratories may require additional Real frames for physically distant hardware. (Note that unique names for the frames are required.)

In addition to the root frames for the hybrid environment illustrated, the Simulation Frame ($\mathcal{S}$, `Frames.Simulation`) is the root frame for the simulation tier and is automatically created by the framework. The Simulation Frame usually represents the largest outer frame of convenience used to describe the events being simulated, such as the Hill Frame for an orbiting formation. Like the Virtual frame, the Simulation frame is the unique root for its tier and should not be duplicated.

The other frames illustrated in Figure 2.3 are not created by the framework itself, but rather are frames of convenience for calculations. The $\mathcal{B}$ frame is the body coordinate frame of the robotic vehicle as described in section 3.1, here defined relative to the $\mathcal{R}$ frame ($\boldsymbol{r}_{\mathcal{B}/\mathcal{R}}$, $[\mathcal{BR}]$). The $\mathcal{C}$ frame is the frame of the camera, as described in section 3.5. And finally, the $\mathcal{T}$ frame represents a target being tracked visually by the others.

Note that the $\mathcal{C}$ frame has no position change relative to $\mathcal{B}$ frame ($\boldsymbol{r}_{\mathcal{C}/\mathcal{B}} = \mathbf{0}$) as seen from overhead: there is no requirement that frames be distinct in position and/or rotation. In fact, it can often be useful from the perspective of coding interfaces to have frame modules linked to one another with no distinction in math, but as software components of different systems. For instance, one frame may represent the mounting bracket of one unit, while the other represents the corresponding mounting surface of the mounted component.

Frames and frame calculations are implemented using UMBRA's `umb::Frame` modules. Each

frame module records a position vector and a rotation. For a child frame $\mathcal{B}$ defined relative to some other parent frame $\mathcal{A}$, these would be the position vector ${}^{\mathcal{A}}\boldsymbol{r}_{\mathcal{B}/\mathcal{A}}$ and the rotation represented as the direction cosine matrix $[\mathcal{BA}]$. This position & rotation are the origin and orientation of the frame relative to the parent frame, and are implemented as a position/rotation input connector, called the `offset`.

The frame module also accepts a position & rotation pair, ${}^{\mathcal{V}}\boldsymbol{r}_{\mathcal{B}/\mathcal{V}}$,$[\mathcal{BV}]$ that defines the parent frame in relation to the root frame, usually $\mathcal{V}$. Direction cosine matrix representations of rotations need only be multiplied together to form successive transformations.

$$[\mathcal{BV}] = [\mathcal{BA}]\,[\mathcal{AV}] \tag{2.1}$$

For the positions, the module uses a technique known as a heterogeneous transformations, piecewise constructing a 4×4 transformation matrix:

$$\begin{matrix}\mathcal{V}\\ \mathcal{B}\end{matrix}\boldsymbol{T} = \begin{bmatrix} [\mathcal{BV}]^T_{(3\times3)} & {}^{\mathcal{V}}\boldsymbol{r}_{\mathcal{B}/\mathcal{V}(3\times1)} \\ 0_{1\times3} & 1 \end{bmatrix} \tag{2.2}$$

This matrix can be used to transform an augmented position vector from one frame to another:

$$\begin{bmatrix} {}^{\mathcal{V}}\boldsymbol{r} \\ 1 \end{bmatrix} = \begin{bmatrix}\mathcal{V}\\ \mathcal{B}\end{bmatrix}\boldsymbol{T}\Big] \begin{bmatrix} {}^{\mathcal{B}}\boldsymbol{r} \\ 1 \end{bmatrix} = \begin{bmatrix} [\mathcal{BV}]^T_{(3\times3)} & {}^{\mathcal{V}}\boldsymbol{r}_{\mathcal{B}/\mathcal{V}(3\times1)} \\ 0_{1\times3} & 1 \end{bmatrix} \begin{bmatrix} {}^{\mathcal{B}}\boldsymbol{r} \\ 1 \end{bmatrix} \tag{2.3}$$

Applying this transformation to the zero position vector in the $\mathcal{B}$ frame, results in the position of the $\mathcal{B}$ frame relative to the $\mathcal{V}$:

$$\begin{bmatrix} {}^{\mathcal{V}}\boldsymbol{r}_{\mathcal{B}/\mathcal{B}} \\ 1 \end{bmatrix} = \begin{bmatrix}\mathcal{V}\\ \mathcal{B}\end{bmatrix}\boldsymbol{T}\Big] \begin{bmatrix} {}^{\mathcal{B}}\boldsymbol{r}_{\mathcal{B}/\mathcal{V}} \\ 1 \end{bmatrix} \tag{2.4}$$

This transformation is applied against the position component of the `input` connector, and the result given to the `output` connector. By chaining the input and output connectors of successive frame modules, each one is related back to the original frame: the operation only results in trivial information for the first child frame, in which the output connector will be the same as the offset connector. For notational convenience, these frame modules are given a similar nomenclature to

the homogeneous transform. For the module that defines $\mathcal{B}$ relative to $\mathcal{A}$ with root $\mathcal{V}$, the module can be compactly expressed as ${}^{\mathcal{A}}_{\mathcal{B}}\mathcal{V}$.

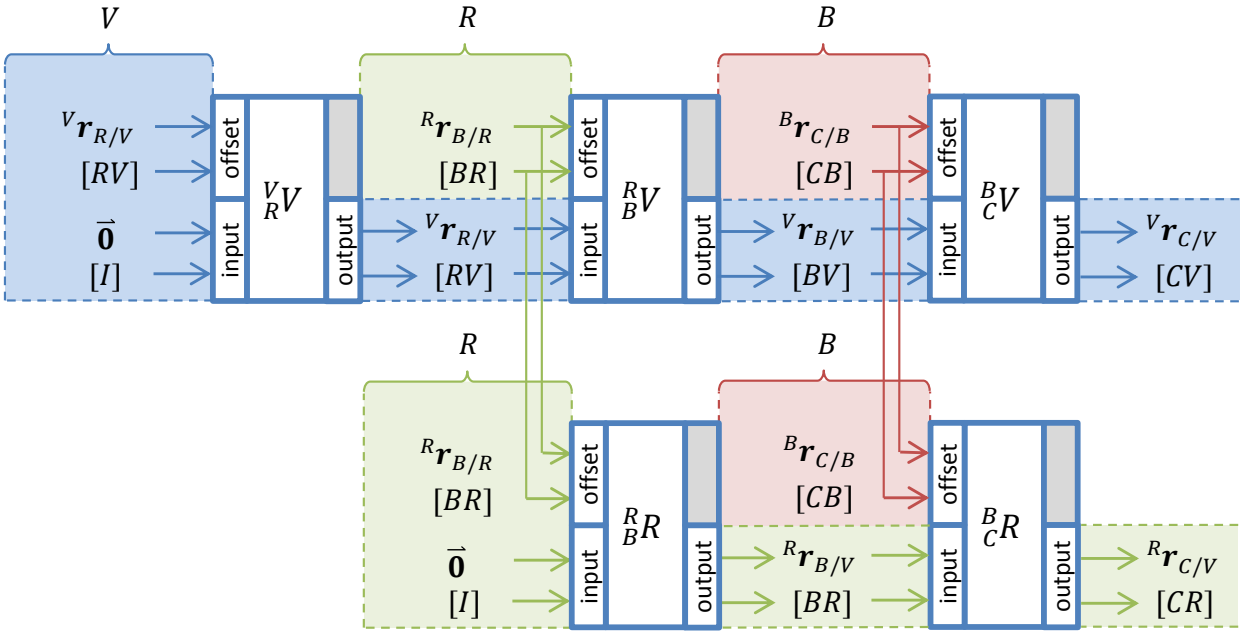

Figure 2.4: A connected tree of modules representing frames

Figure 2.4 shows a typical definition of frames, in which each successive frame is defined in terms of the previous frame. The chain at the top of the diagram is representative of the standard frame chain which the framework automatically creates. The virtual frame ($\mathcal{V}$, `Frames.Virtual`), shown in blue, at the root of the chain has a fixed input and offset of zero, setting it as the root frame against which other frames in the chain are measured. That is, the input transformation of the root frame module in a given chain is identity, so that it maps to itself. The module for the $\mathcal{V}$ frame (${}^{\mathcal{V}}_{\mathcal{V}}\mathcal{V}$) is not shown in figure 2.4 as its inputs and outputs are all identity transformations, but would be to the immediate left of the space shown for $\mathcal{V}$ frame calculations. (In fact, a $\mathcal{V}$ frame module is not technically required, but provides a logistical anchor point for other modules, and is generally good practice.)

The real frame ($\mathcal{R}$,`Frames.Real`), shown in green and defined by module ${}^{\mathcal{V}}_{\mathcal{R}}\mathcal{V}$ takes in the position and rotation of the frame it is defined against in the `input` connector. Since this is only the

second frame in the chain, it will always receive an identity transformation. The `offset` connector defines this frame in terms of the previous frame, with the position vector ${}^{\mathcal{V}}\boldsymbol{r}_{\mathcal{R}/\mathcal{V}}$ and rotation $[\mathcal{RV}]$ relative to (and expressed in the components of) the $\mathcal{V}$ frame. The output of the frame module is the frame's position and orientation relative to the root frame in the chain, thus, for the second frame in the chain, the `output` will always be the same as the `offset`.

The body frame, $\mathcal{B}$, is a user-defined frame implemented by module ${}^{\mathcal{R}}_{\mathcal{B}}\mathcal{V}$. This frame is defined by its position (${}^{\mathcal{V}}\boldsymbol{r}_{\mathcal{R}/\mathcal{V}}$) and orientation ($[\mathcal{BR}]$) relative to (and expressed in components of) the $\mathcal{R}$ frame. Note that the `output` of this module is still expressed in terms of the root frame $\mathcal{V}$. The camera frame (${}^{\mathcal{B}}_{\mathcal{C}},\mathcal{C}\mathcal{V}$) is another user-defined frame and demonstrates the logical extension of the chain.

The lower half of figure 2.4 demonstrates the creation of a second frame tree using a different root frame, $\mathcal{R}$. A second module is needed to represent the $\mathcal{R}$ frame as the root: ${}^{\mathcal{R}}_{\mathcal{R}}\mathcal{R}$. The modules ${}^{\mathcal{R}}_{\mathcal{B}}\mathcal{R}$ & ${}^{\mathcal{B}}_{\mathcal{C}}\mathcal{R}$ implement the $\mathcal{B}$ & $\mathcal{C}$ frames, respectively on this chain. However, note that the output of this chain at every step is referenced to the root frame of the chain, $\mathcal{R}$.

Using the fact that UMBRA connections are one to many, with one output being linkable to many inputs, the frames can be organized into a tree structure, with multiple child frames being defined against a single parent frame. The output of the parent frame module is connected to the inputs of each of the child frames, while each child frame module has a separate offset.

2.3.2 Visualization Spaces

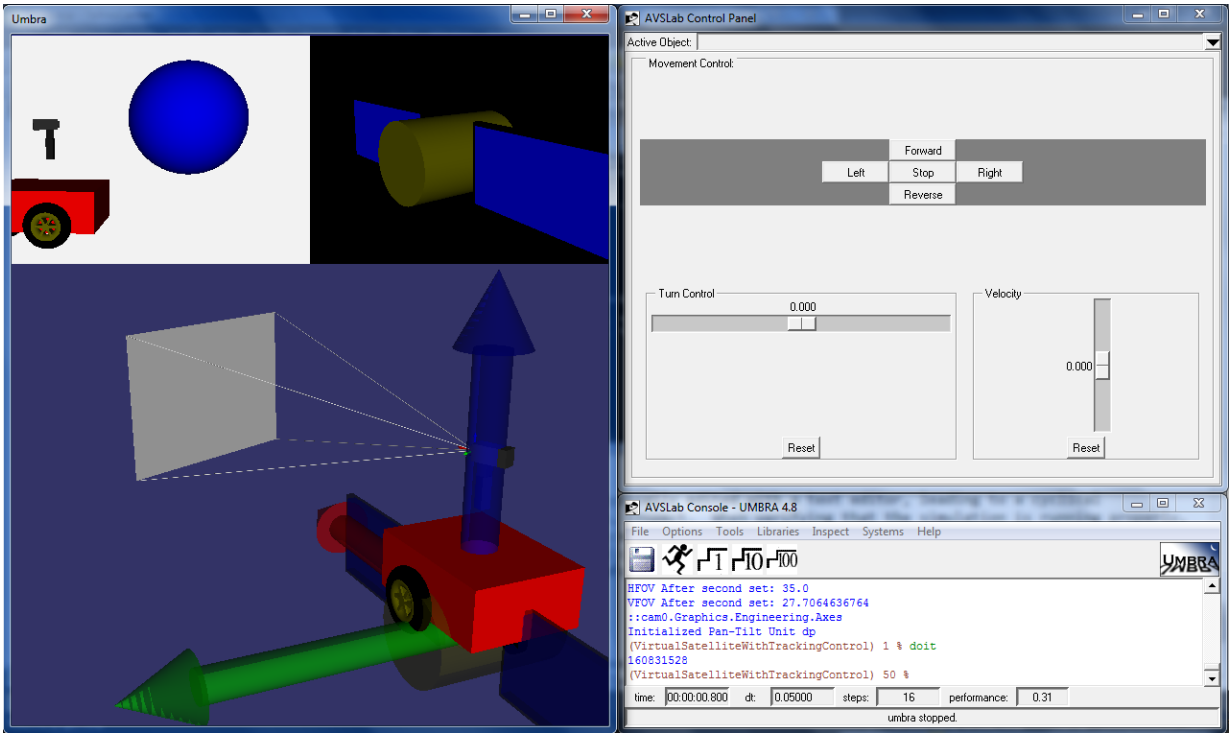

Figure 2.5: Screenshot of a simulation running with the AVSLab Framework

UMBRA utilizes OpenGL and OpenSceneGraph to provide three-dimensional representations of objects. Each collection of objects exists in a renderable space called a 'scene', implemented by the class `osg::scene`. Because of the multi-tiered approach outlined in section 2.3, it is useful to have multiple scenes which capture different tiers or aspects. By default, the framework creates three such scenes automatically. Figure 2.5 shows a screenshot of a typical simulation. The visualization spaces are shown in the 3D graphics window on the left side of the screen.

The Virtual Scene, `Scenes.Virtual`, shown in the top-left corner of the 3D window, depicts the real/virtual hybrid tier of the framework. It contains 3D models of virtual hardware, as well as representations of real hardware operating in the laboratory. The Virtual Scene is intended to roughly approximate the appearance of the real world, allowing virtual visual sensing techniques to see 'real' hardware and respond accordingly.

The Simulation Scene, `Scenes.Simulation`, shown in the top-right corner of the 3D window, is similar to the Virtual Scene, but depicts the Simulation Tier of the framework. It should as nearly as possible approximate what a piece of (space-borne) hardware in the simulation would see, such as other satellites, the Earth, or the Sun.

The Engineering Scene, `Scenes.Engineering`, occupying the lower portion of the 3D window, is different from the tier scenes in that it exists primarily to provide a meaningful overview to someone watching the simulation. While the Virtual and Simulation scenes are devoid of markers, which would be rendered by any virtual cameras looking into the scene, the Engineering scene contains axes, translucent components, and mixed components from the Virtual and Hybrid scenes. This allows the Engineering Scene to add a wealth of information about the status of the simulation which would not otherwise be apparent. In figure 2.5, the Engineering Scene shows both the robotic vehicle and the satellite it simulates superimposed, allowing the user to visually confirm that the simulation is working properly and understand the relationship between the two tiers of the simulation at a glance.

2.3.3 Component Projects

In order to make understanding each component library as easy as possible, the files in the library should follow a standard layout pattern. Each project should exist as a sub-directory under the `Share` directory (or as directed by Sandia's outlined best practices in future versions of UMBRA). The root of the project directory is where the header & code file for the compiled library should reside, as well as a project-level CMake file. A library TCL file of the same name as the project should also be found here, and is used by UMBRA to load all of the script components of the library as well as the compiled libraries. A debug script (by convention, the name of the Project with 'Debug' appended) may also reside here which would assist in debugging the project, usually by instantiating the modules within UMBRA and running them through specific conditions.

The remainder of the files should be stored in subdirectories of the project root. The C++ code and header files for the UMBRA modules should be located in the `Modules` subdirectory, and

Tcl scripts that are part of the library should reside in `Scripts`. Other supporting libraries may be accorded their own subdirectory if they are being built solely for the use of the one project, or may be kept in a different location and referenced. Optionally, verification Tcl test-scripts may be stored in the subdirectory `Tests`.

2.3.4 Framework Code Structure

The Tcl scripts used in the AVSLab Framework are located in a component project directory named `AVSLabFramework`. Primary scripts reside in the root of the folder, and are used to launch the framework for a given simulation. `Setup.tcl` is the core script file that loads all of the other scripts, and `Debug.tcl` launches the framework itself in an empty simulation, with some settings changed to better aid debugging of framework features. The remaining scripts are further subdivided into category directories based on function.

The `\Utilities` subdirectory contains scripts which provide libraries of related Tcl methods and implement framework systems. For example, the script `Display.tcl` implements the creation & layout engine for the various types of user interface elements seen when a simulation is started, while `Events.tcl` implements the system of event hooks and callbacks. Of particular interest to users is `Inspection.tcl`, which contains methods for inspecting bundles in live simulations at the Tcl prompt.

The `\Interfaces` subdirectory contains definitions for Tcl-bundle (structured grouping of variables and Tcl code used in lieu of true object-oriented programming) interfaces. Each interface defines a set of variables that the bundle must have and code it must define, similar to multiple inheritance in C++. An interface captures a useful aspect that a bundle can present, and allows other code to work with that bundle agnostically. For instance, the `IPauseable` interface (`IPauseable.tcl`) allows other code to pause and resume any bundle implementing `IPauseable` without any additional knowledge of what the bundle is. In this way, `IPauseable` permits pausing of the simulation even with real hardware, provided that all of the bundles representing the hardware implement `IPauseable`. Used properly, interfaces can provide a very powerful technique for

achieving otherwise difficult results.

The `\Graphics` subdirectory provides common graphics for the framework, including generic graphics for simulation items and visualizations. Graphics are generally built on `GeoObject` Tcl objects, provided by the System of Systems (`SoS`) UMBRA package, which provide an easy means to build up and manipulate complex trees of 3D shapes in a visualization space. Note that graphics related to specific component projects should be included with the projects' script folders rather than with the full framework.

The `\Settings` subdirectory contains script files which control the various default settings and parameters available for the framework. Most users will likely find little need to alter most of the settings, but they are provided for completeness and adaptability. These settings can be overridden in specific simulations simply by setting the values listed before initializing the framework. A local settings file, `SettingsLocal.tcl` may be created to impose local defaults for a specific machine and will be automatically loaded by the framework. This can be useful for display settings, which may need to vary to accommodate differing sizes of operating system user interface elements, such as the task bar. (The controlling computer for the hardware, for example, uses a default GUI layout better-suited to the smaller screen.) Because this can vary across machines, users are advised to include settings that are critical to a simulation in the files for that simulation. The script `Alias.tcl` provides typing-friendly alias wrappers for the more verbose full names of library functions.

The `\Templates` subdirectory contains template files for using and expanding the framework, such as a template library file, or a template simulation launcher script. The `\Tests` subdirectory is a folder for tests of individual systems of the framework.

2.3.5 Loop Cycle

Figure 2.6: Update loop cycle and simulation flow

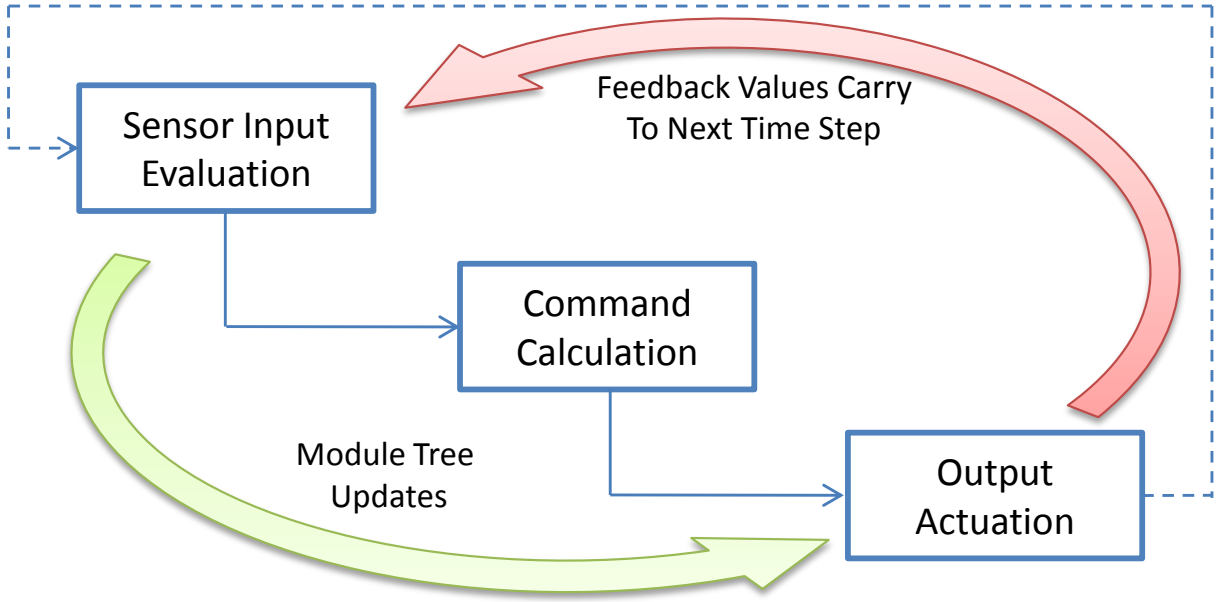

Figure 2.6 demonstrates the looping features of UMBRA. UMBRA modules are sorted into a non-circular dependency tree, which is then updated for one time step. Modules whose inputs depend on the outputs of another module are calculated after the module that supplies their inputs. However, most simulations of interest have circular dependencies which evolve simultaneously in time in the real world. The solution for discretizing this into finite time steps is the feedback connection, shown at the top of figure 2.6, and adopted as a general notation in this work, as a dashed line. Feedback connections allow a module input to use the output value from the previous time step, thus providing a way to define the breaking point in the loop where the current time is updated.

Figure 2.6 illustrates the general pattern for choosing where this break should occur, which in turn dictates the general structure of a simulation's connections. A natural breaking point occurs

between actuation and sensing, as with real hardware this part of the relationship is implicit. Thus, sensing should occur first in a given time-step, then calculation of commands based upon the information, followed by actuation of thrusters, motors, and so forth. This forms a reactionary pattern in which each time step is a reaction to the outcome of the previous.

2.4 Usage

2.4.1 Tools

The primary tools for using the framework are:

- UMBRA, the framework published by Sandia National Laboratories for dynamic integration of C++ modules in simulations.
- CMake, a cross-platform makefile builder capable of supporting multiple compilers from one script. CMake files allow compilation of most codes on other platforms or with other tools if properly maintained.
- A Tcl distribution, typically ActiveState's ActiveTcl, of a version matching the required version for UMBRA.
- An IDE, typically a version of Microsoft Visual C++ Express Edition (made available to the public at no cost) matching the version of compiler used to compile UMBRA. Other IDEs or text editors may be used in place of this, and the compiler invoked separately.
- A full-featured text editor. For coding in Tcl, a more advanced text editor than the basic ones usually packaged with an operating system is desired. Many are available; popular choices with explicit coding support include Notepad++, PSPad, KomodoEdit, emacs & vi.
- A subversion version-control system client, such as TortoiseSVN. Version control for coding is invaluable for preventing cross-breaks in modifications between developers, accidental code loss, and keeping track of stable versions.

2.4.2 Workflow

The workflow for developing in or with the framework varies somewhat depending on the specific type of development. This can generally be broken down into development of simulations, components, and the framework itself. Best practices and style guidelines for writing the code itself are contained in appendix A.

2.4.2.1 Simulation Development

Simulations generally grow from small, simple scripts. The templates subdirectory of the framework contains a script launcher template which can be used to start a simulation script in the framework. A good pattern for repeatable, flexible simulations is to reuse the same simulation script with multiple launcher scripts, each launcher containing different initial conditions, settings, or other variables used by the simulation script to produce different outputs.

The launcher scripts can be activated with only one or two clicks by creating a shortcut to the UMBRA executable (`uwish.exe` with the `-load` parameter and the name of the launcher script, using the simulation's directory for the working directory. This allows for repeated, rapid startup of the simulation.

The Tcl files of the simulation are typically edited with a text editor, leading to a cyclical procedure for development: Edit, run, inspect. When verifying that the simulation is running properly, the framework's inspection namespace can be invaluable for rapidly discerning what bundles exist. The logging tools can be used to generate comma-separated value files (CSV) which can then be repeatedly imported into a spreadsheet program after each run to provide visual time-histories of important quantities.

2.4.2.2 Component Development

New modules are often required for developing simulations. Modules should be grouped into related component projects, each with a compiled C++ library and optionally accompanying Tcl scripts as described in section 2.3.3. Typically, these are edited from the Visual C++ IDE, with

a small test Tcl script for the UMBRA module. A script launcher may be used if desired, but it is not necessary to launch the framework for these tests unless it interacts with the framework in some way. Visual C++ can be used to launch the code directly in a debug mode by inputting the same properties as the shortcut in section 2.4.2.1 as the run action for the IDE's debug. The workflow for this becomes: Edit, run, inspect.

2.4.2.3 Framework Development

Developing the framework itself is very similar to developing a simulation in workflow, and can in some cases be done in tandem. For developing or debugging the framework, some additional tools are available. First is the debug mode of the framework, which runs only the framework itself, invoked from the `Debug.tcl` file in the framework's directory. The second is the `ReloadAllLibraries` command, aliased to `ral`, which reloads all of the libraries of the

`Utilities`,

`Interfaces` and

`Graphics` subdirectories. This allows development of the framework in real-time, especially useful for libraries which are not state-dependent. Libraries that cache information or require some types of initialization may need a restart of the framework.

CHAPTER 3

LABORATORY HARDWARE

The AVS Laboratory is host to many pieces of hardware which are represented in the mixed-reality 'virtual' tier of the framework. This equipment is used to test pit theoretical control performance against realistic sensing conditions, such as variable target lighting for visual tracking.

3.1 Robotic Motion Platform

The robotic motion platform (interchangeably referred to as the vehicle or the robot) is a Mobile Robotics model Pioneer 3-DX, pictured in Figure 3.2 and as part of the integrated hardware stack in Figure 3.1 (outlined in blue). The vehicle has two primary wheels situated on either along the center of the vehicle body, with a rear caster for support. By varying the speed at which each wheel rotates, the vehicle can drive forward or backward axially, or turn left or right. Combining these motions allows it to produce smooth curvilinear motion in a plane. This makes the vehicle well-suited for simulation of relative orbital trajectories in close proximity, which are very nearly planar and naturally smooth curves. The wheels are equipped with optical encoders which detect wheel movement with high precision (2000 positions per revolution), which, along with an embedded IMU sensor, allows the vehicle to determine movement with precision on the order of millimeters, including accounting for wheel slippage. The vehicle can accept either position or velocity commands, with an embedded controller enforcing these commands.

Power is supplied to the unit (and optionally to the connected devices) by one to three rechargeable lead-acid batteries stored in the rear of the body. A control panel on the left rear

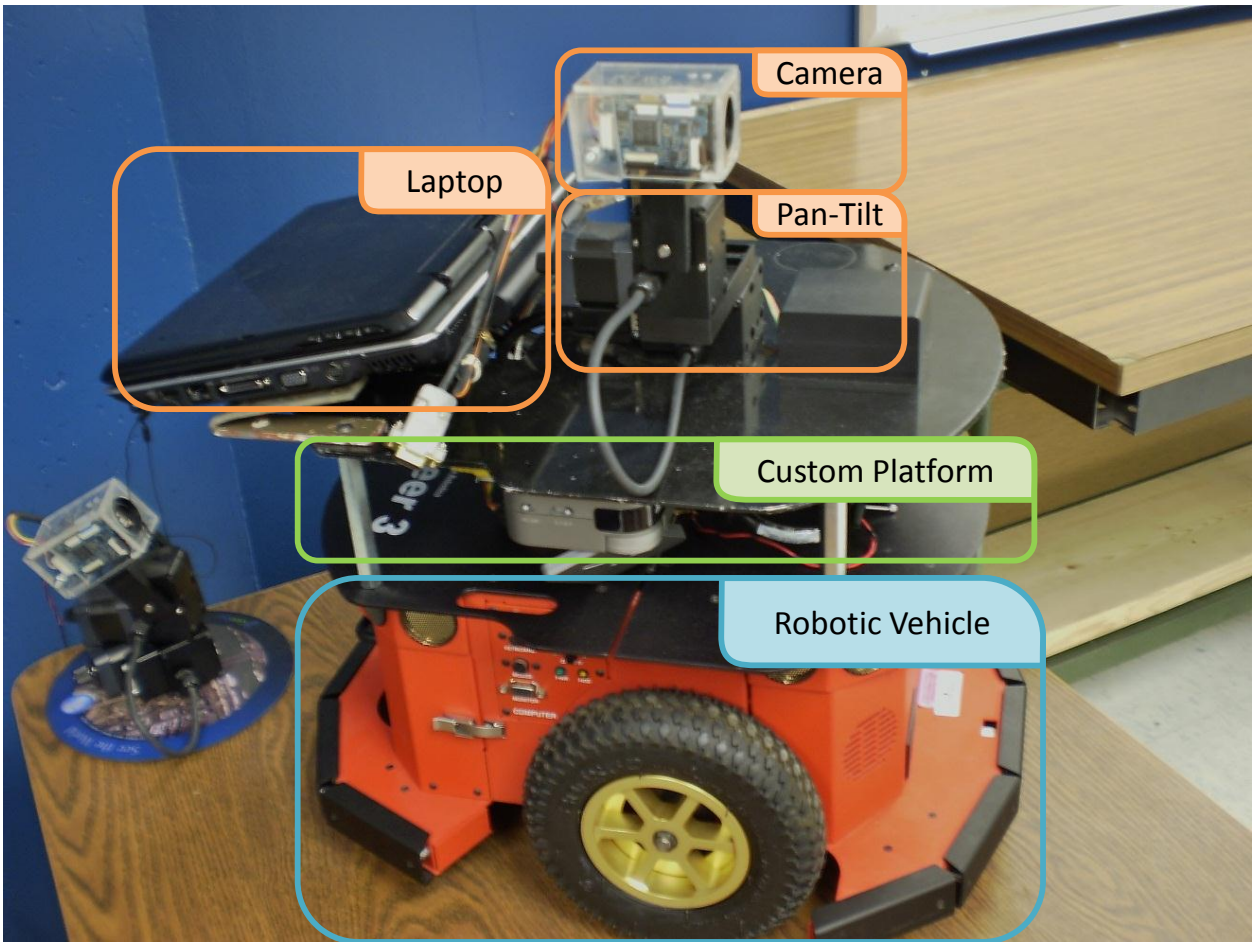

Figure 3.1: Annotated view of the robotic motion platform integrated with all external devices

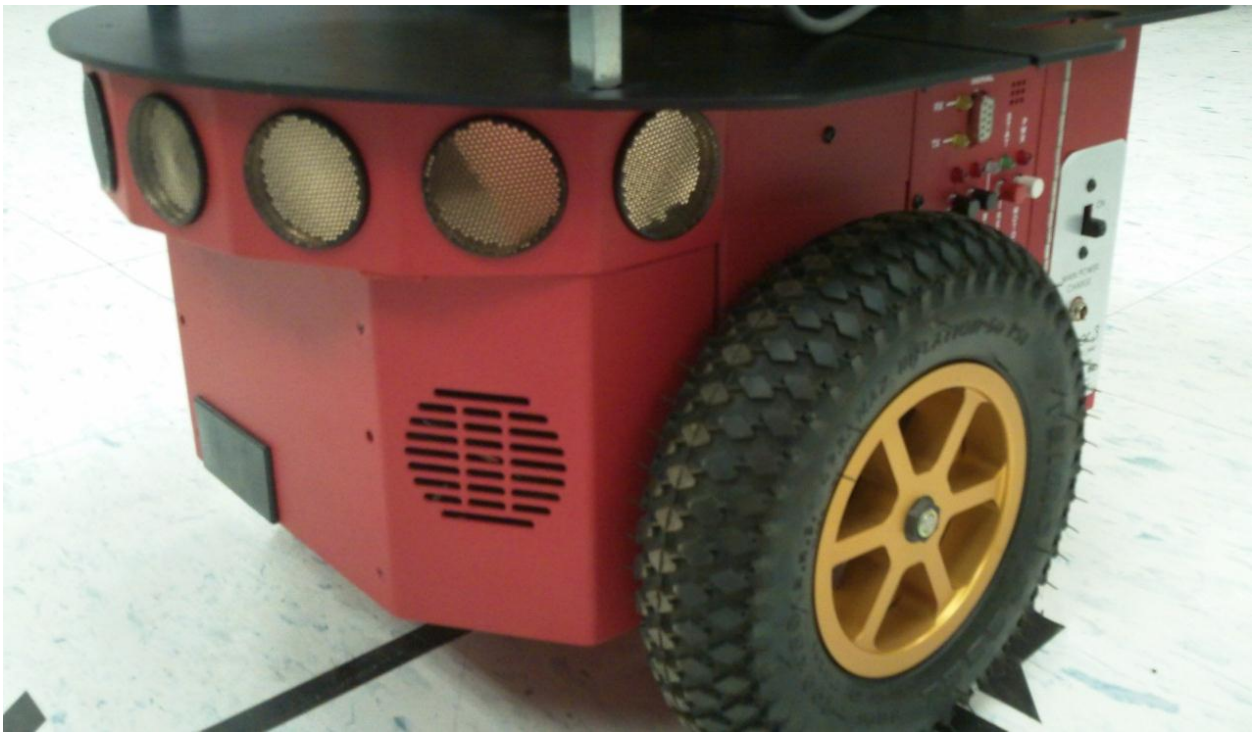

Figure 3.2: The Pioneer 3-DX robotic motion platform

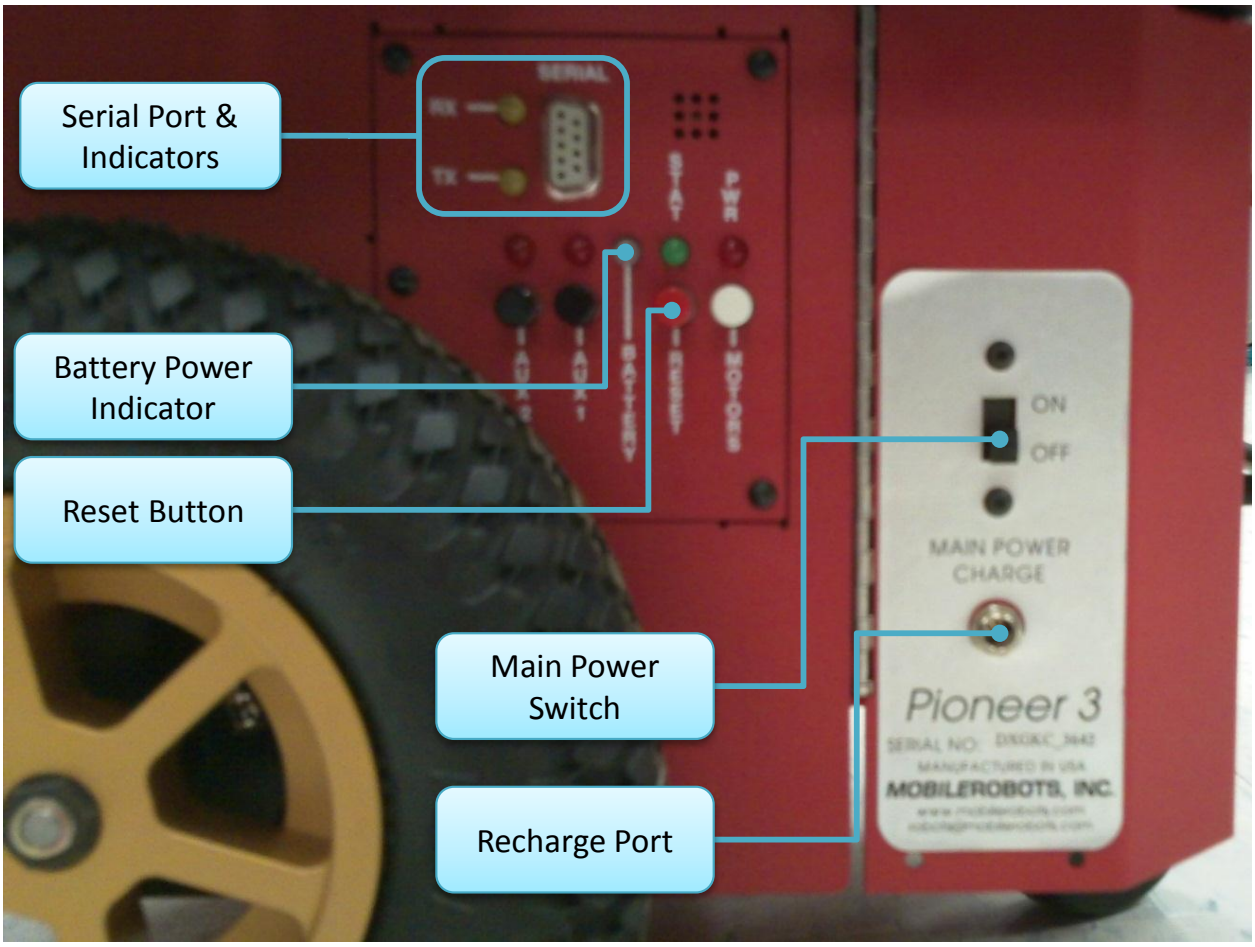

Figure 3.3: Control panel of the robot

corner of the vehicle provides basic power and reset controls as well as a serial port for an external computer to control the hardware. Power terminals inside the unit provide power from 5V and 12V buses, although peripherals attached in this manner may run low on power before the audible low-power chime is activated.

The vehicle tracks, to the best of its ability, and reports its current position and orientation relative to the position and orientation in which it was last initialized. The coordinate system used by the interface libraries may be thought of as a 'compass' system, in that the vehicle faces the positive Y-axis at a heading of zero. However, this is not a very readily generalized coordinate system, and so is mapped in software to a 'proper' coordinate system in which the vehicle's direction of forward travel is the positive X axis and the Y axis points to the left of the vehicle. This frame is considered to be the 'body' frame of the vehicle. In both frames, the Z axis is centered between the two wheels (co-incident with the turn axis of the vehicle), perpendicular to the plane of movement, with positive corresponding to the top of the vehicle and the origin at the ground level. The vehicle should be re-initialized at the beginning of each simulation run by use of the reset button.

3.2 Hardware Support Platform

The top of the robotic vehicle has a mounting surface upon which various pieces of external equipment may be carried. In the interest of making it easy to manage external devices and to detach them from the vehicle, a custom mounting platform was created from plate steel and painted black, seen in Figure 3.1 outlined in green. Four four-inch risers vertically offset the platform to allow room for devices to be mounted without obstructing the view of active components on top. The plate is covered with hook material for attaching strap or dot type hook-and-loop fasteners, which makes mounting devices as simple as attaching loop material which can double as softer feet for the device. Cable management is also easily accomplished with a few straps.

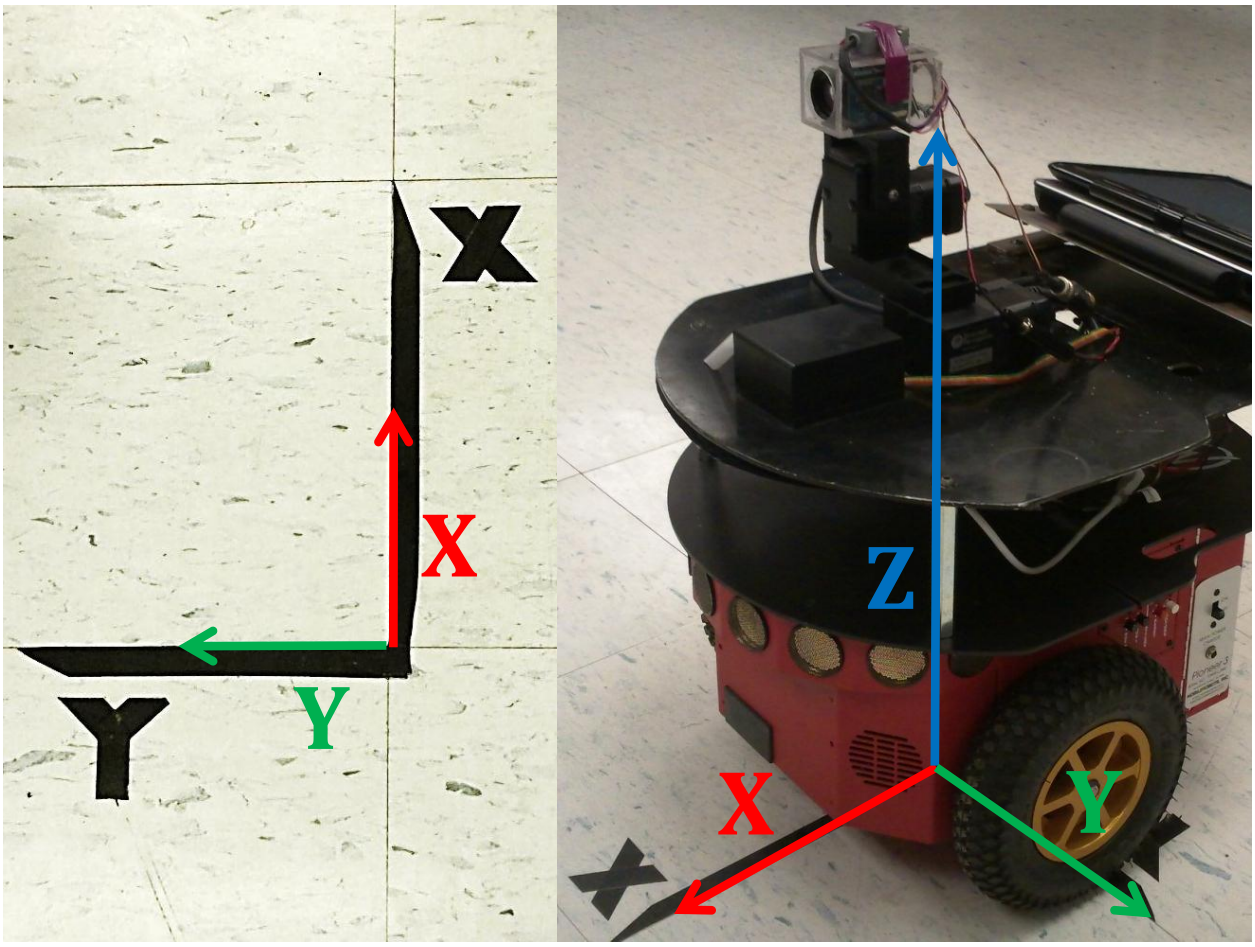

Figure 3.4: Robot's coordinate system

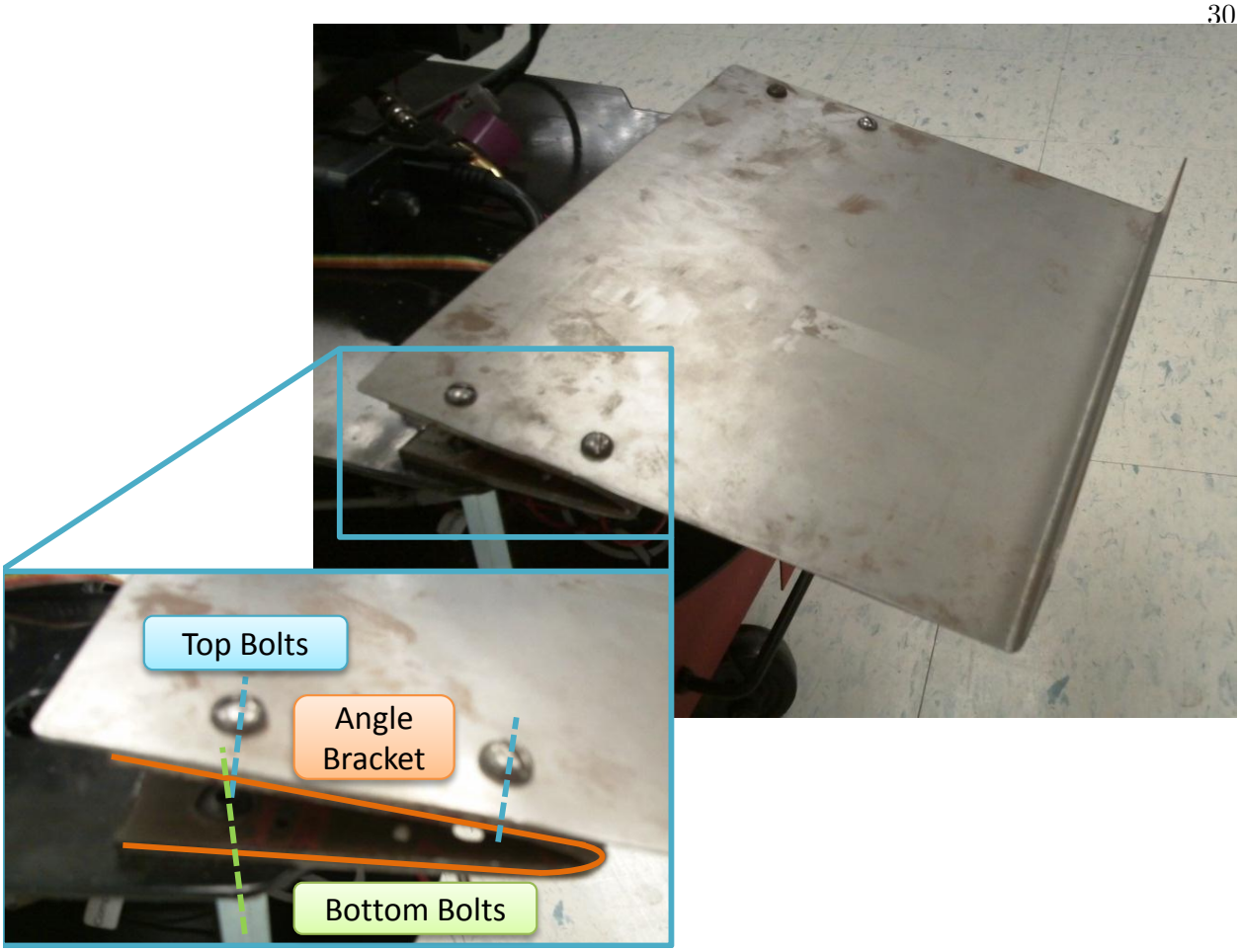

Figure 3.5: Mounting bracket for the controlling computer

At the rear, a pair of angle brackets support a custom cradle which serves as a mount for the external laptop. The support devices attached to the underside of the mounting platform include video capture, USB hub, serial port adapters, pan-tilt mount control, and external video broadcast. Taken together, these unify the robotic vehicle and all of its supporting devices into one single USB plug for the external computer.

3.3 Controlling Computer

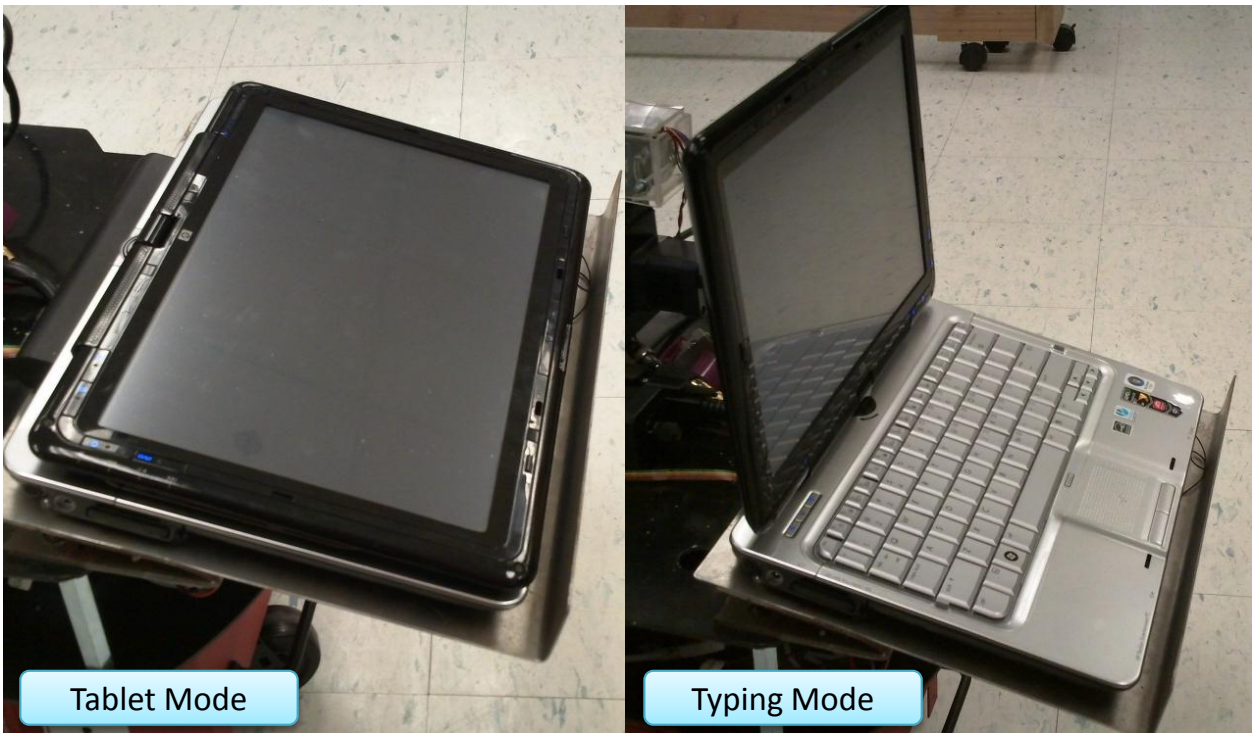

Figure 3.6: The controlling computer in both tablet and typing modes

Although the Pioneer 3-DX is capable of supporting an embedded computer, early trade studies determined that a laptop (outlined in orange) with an external mounting bracket on top of the mounting platform was more economical, adaptable, and powerful. The computer selected for this task is a Hewlett-Packard Pavilion TX2000 tablet PC, shown in figure 3.6 in both tablet and laptop modes. The device was selected for the flexible input modes, video processing capacity, and small (12-inch diagonal) form-factor. More detailed specifications for the device are presented in Table 3.1.

Processor	AMD Turion X2 Mobile, 2.1 GHz
Memory	3GB DDR2
Graphics Processor	ATI Radeon 3200 HD (Integrated)

Table 3.1: Processing specifications of the controlling computer

3.4 Pan-Tilt Unit

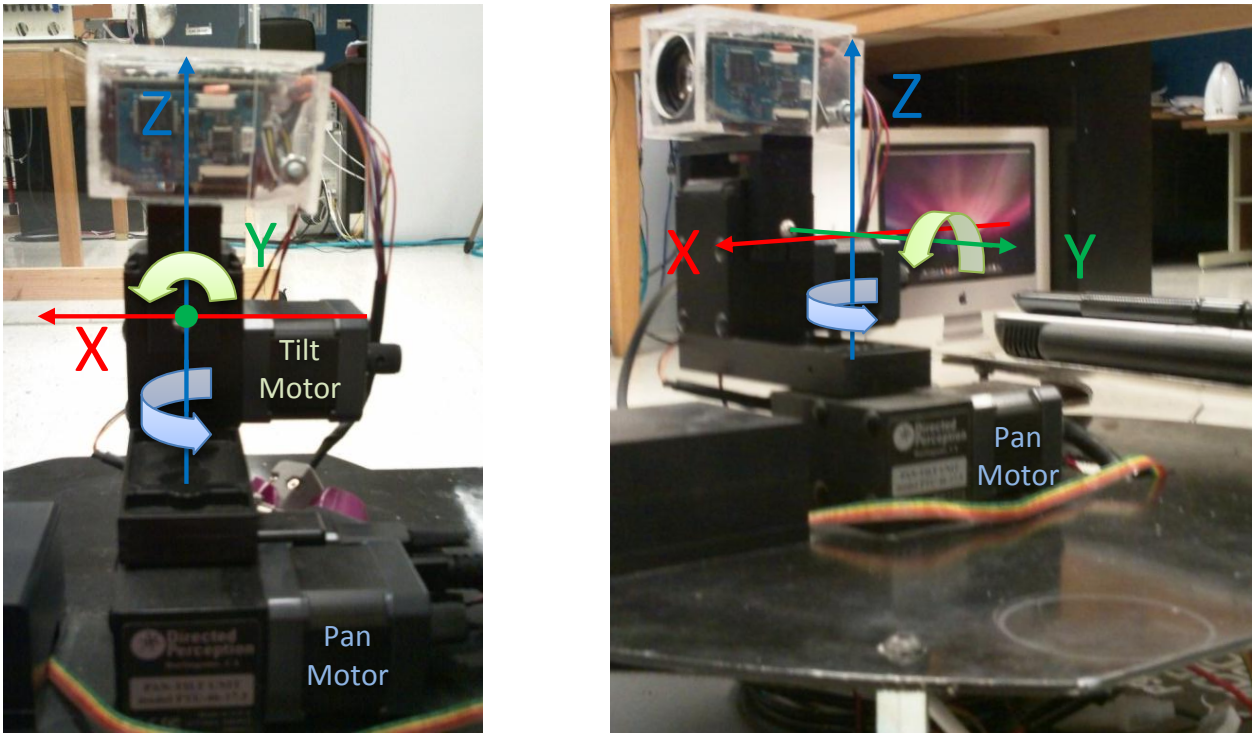

Figure 3.7: The Directed Perception pan-tilt unit

While the robotic motion platform implements planar translational motion, simulation of orientation is accomplished by a pan-tilt unit mounted on the hardware support platform with the pan axis aligned with the turn axis of the vehicle. The unit is a Directed Perception (now FLIR Motion Control Systems) model PTU-D46-17 pan-tilt mount unit, intended for use with a camera. The unit is comprised of two motorized axes and an external controller module. Each of the motors is capable of high-precision, variable-speed movement, providing two rotational degrees of freedom

in a classic Euler angle sequence 3-2 combination. The external controller is mounted beneath the hardware support platform, where a serial to usb converter cable makes the unit available to the computer.

The pan-tilt unit motors are mounted such that the axes of each motor's rotation are perpendicular and intersect. Hardware mounted at the point of intersection experiences pure rotation with two degrees of freedom. The unit may be upgraded at a later date to incorporate a third degree of freedom, completing the Euler 3-2-1 angle sequence with a 'roll' axis.

3.5 Camera

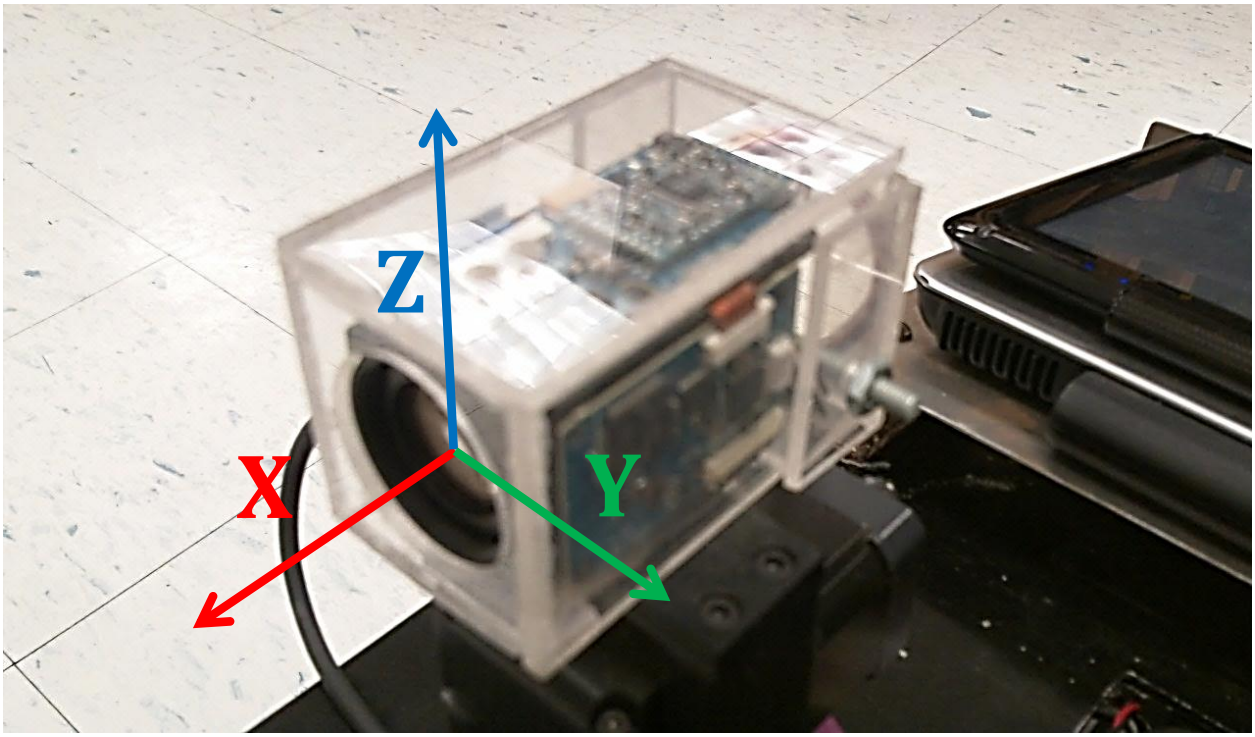

Figure 3.8: Sony FCB-I10A mounted on the Pan-Tilt Unit

For limited visual control purposes, a camera mounted at the intersection of the pan & tilt axes of the pan-tilt mount captures the essential motion. The camera currently being used is a Sony model FCB-I10A, which provides standard-definition NTSC video (640x480 pixels) via BNC

connector. The camera's features include automatic white-balance, auto-focus, and access to the internal hardware pins along with thorough documentation. A BNC-to-RCA adapter permits usage of the more standardized RCA composite video connectors found on most consumer-grade NTSC equipment, and a Y-splitter for video permits future potential video access for a second device, such as a wireless video transmitter.

The camera is presently mounted on top of the pan-tilt unit's tilt actuator block directly, as seen in figure 3.7. This places the camera's focal frame at an offset from the rotation point, introducing a small error for attitude control simulations. This problem which will be resolved with the future addition of a new mounting bracket, possibly in tandem with a roll control system. (See section 6.1)

3.6 Video Capture

Figure 3.9: Pinnacle Dazzle DVC-100 Video Capture Device

Video is captured with a Pinnacle Systems Dazzle Video Capture device (model DVC-100). The device accepts composite video (NTSC) and stereo audio, and communicates with (and draws power from) the computer via USB 2.0. The drivers make the captured video stream available to programs through Microsoft's DirectX graphics framework, where it is listed as a standard camera.

Video quality is observed to be a good reproduction of the original. The device mounts on the underside of the hardware support platform (shown mounted on the right side of figure 3.9), and connects to the camera described in section 3.5 by a standard RCA composite video cable.

CHAPTER 4

SOFTWARE COMPONENTS

A number of software component modules populate the framework. Pre-existing modules from earlier work had to be ported forward from older versions of the UMBRA framework. Real hardware modules without a virtual complement required the design of a hardware emulator. Numerous integration modules have been created to bridge the core modules. Implemenations of mathematical constructs such as integrators and control laws allow for simulation of the performance of control algorithms and physics.

4.1 Math Libraries

The MathLibrary project is a set of newly-created modules that implement a number of mathematical descriptions, operations, and simulations. In addition, it contains a number of interfacing modules which allow for manipulation & conversion of the various types in the UMBRA environment.

4.1.1 Attitude Propagator

The Attitude Propagator module (**MathLibrary::AttitudePropagator**) integrates the rotational equations of motion of a rigid body. It uses both current attitude ($\boldsymbol{\sigma}_{\mathcal{B}/\mathcal{N}}$) with respect to an inertial frame ($\mathcal{N}$), in modified Rodriguez parameters and the current body-frame ($\mathcal{B}$) angular velocity (${}^{\mathcal{B}}\boldsymbol{\omega}_{\mathcal{B}/\mathcal{N}}$) in radians per second as the state of the object. It accepts a mass-moment of inertia matrix to define the mass distribution of the object, and a body-frame torque. As inputs

are constant over a given time step, the simple Euler's method of integration is used.

Modified Rodriguez paramters (MRPs,$\boldsymbol{\sigma}$) are a rotational representation defined in terms of the Euler parameters (Quaternion components). For the Euler parameters (β_0,β_1,β_2,β_3), the corresponding MRP representation is:

$$\sigma_i = \frac{\beta_i}{1+\beta_0} \qquad (i = 1, 2, 3) \tag{4.1}$$

with inverse transformation

$$\beta_0 = \frac{1-\sigma^2}{1+\sigma^2} \qquad \beta_i = \frac{2\sigma_i}{1+\sigma^2} \qquad (i = 1, 2, 3) \tag{4.2}$$

where $\sigma^2 = \boldsymbol{\sigma}^T\boldsymbol{\sigma}$. Or, in terms of the principle rotation vector ($\hat{\boldsymbol{e}}$,Φ):

$$\boldsymbol{\sigma} = \tan\frac{\Phi}{4}\hat{\boldsymbol{e}} \tag{4.3}$$

Modified Rodriguez parameters are used for the integration and control laws because they are resilient to singularity issues, as a singularity can only occur when describing a full revolution. As with Euler parameters, which are non-unique, every MRP set has a shadow set, which represents the longer rotation to return to the same point. (Thus, when the singular MRP representation would be encountered, the shadow MRP set is $\sigma_I = 0$) Better still, the set which meets the criteria $\sigma^2 \leq 1$ is the shorter rotation of the two sets, making it simple to determine when shadow set switching should be used.

On each update of the module, the attitude is integrated from the velocity at the previous time step, using the differential kinematic equation for MRPs,

$$\dot{\boldsymbol{\sigma}} = \frac{1}{4}\left[\left(1-\sigma^2\right)[I_{3\times3}] + 2\left[\tilde{\boldsymbol{\sigma}}\right] + 2\boldsymbol{\sigma}\boldsymbol{\sigma}^T\right]\boldsymbol{\omega} = \frac{1}{4}\left[B\left(\boldsymbol{\sigma}\right)\right]\boldsymbol{\omega} \tag{4.4}$$

to find the MPR rates from the angular velocity. The Euler rotational equations of motion,

$$[I]\,\dot{\boldsymbol{\omega}} = -\left[\tilde{\boldsymbol{\omega}}\right][I]\,\boldsymbol{\omega} + \boldsymbol{L} \tag{4.5}$$

are used to find the rate of change of the rotational velocity vector, and then integrated to obtain the new rotational velocity vector.

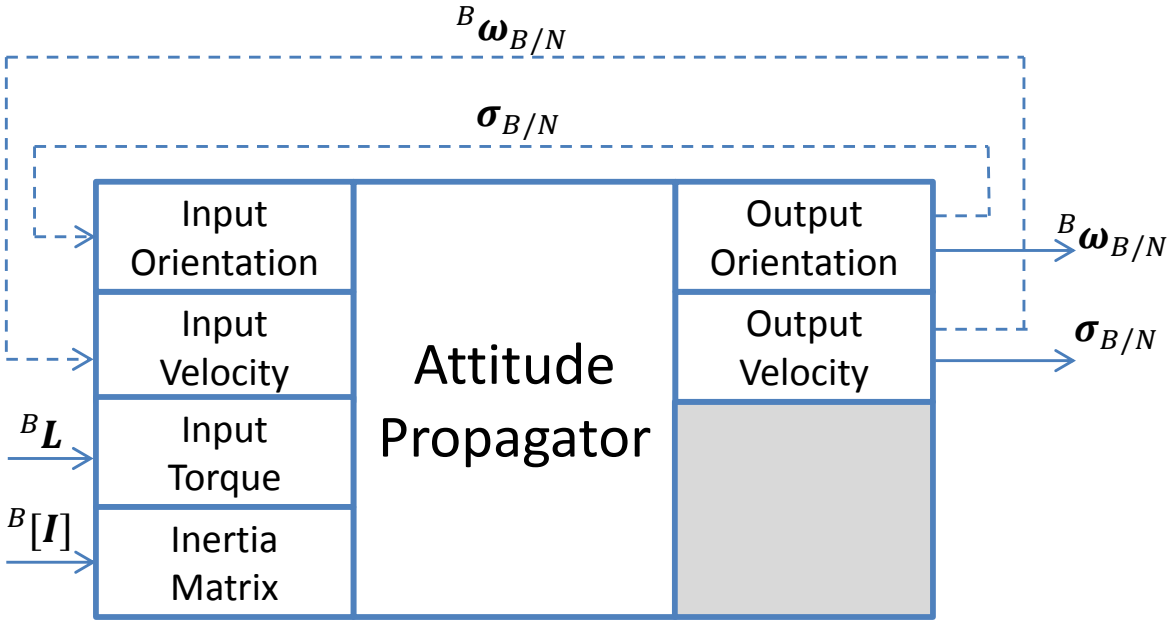

Figure 4.1: Block diagram demonstrating external state of the attitude propagator

The state of the propagator is not kept internally, but utilizes UMBRA's native feedback connections to update the input state at the beginning of each time step with the previous value of the output state. This, in effect, causes the module to operate in lockstep with the whole simulation. The external state has the advantage of permitting manipulation of the object's state, including logging and repeatability. Integration at finer time scales than the global time step is unnecessary, as input torques are constant over the duration of the time step.

4.1.2 Rotation Converter

The Rotation Converter is one of a number of modules designed to make connections more fluid and interoperable between different types of inputs and outputs on various modules. The Rotation Converter converts between many different formats of expressing rotations. Internally, the module uses Euler Parameters as a 'universal currency' due to their lack of singularity behavior. Inputs are converted to Euler Parameters, and then to the desired output format.

4.1.3 Attitude Compensator

Although the robotic platform is intended primarily for translation, due to the two-wheeled design, the attitude of the platform is not decoupled from the translational motion. That is, it must turn in order to change translational direction, whereas a satellite in orbit would not. Because the pan axis of the camera is mounted directly above the turn axis of the platform, it is possible to use the pan motor of the camera's pan-tilt mount to compensate and retain a fixed (or proscribed) motion for the camera independent of the platform's orientation.

The Attitude Compensator module, based on an early Tcl-script implemenation [11], corrects these errors, allowing the combined hardware to implement a proscribed translation and orientation. In essence, the Attitude Compensator maps the simulation physics onto real or virtual hardware.

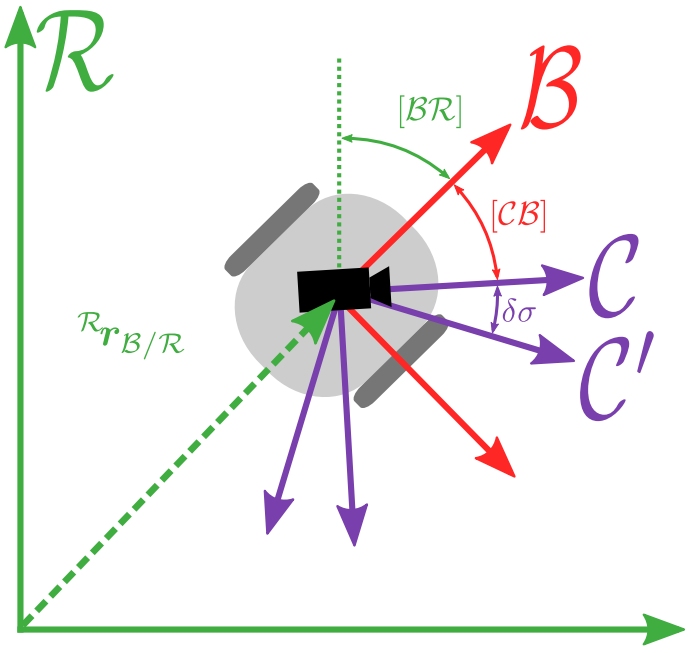

Figure 4.2: Frames of reference for the Attitude Compensator

The Attitude Compensator component uses a proportional feedback/feedfoward control law

to determine the required commands for the pan/tilt unit and remove any introduced errors. Figure 4.2 shows the construction of the frames, with an outer inertial frame (assumed here to be the real frame R), and the body frame of the platform, B. There are two co-located camera frames. The frame co-incident with the camera's focal frame is $\mathcal{C}$, while the desired frame orientation is $\mathcal{C}'$. For convenience, we define the rotation and rotational velocity of $\mathcal{C}$ relative to $\mathcal{C}'$ as:

$$\delta\boldsymbol{\sigma} = \boldsymbol{\sigma}_{\mathcal{C}/\mathcal{C}'} \tag{4.6}$$

$$\delta\boldsymbol{\omega} = \boldsymbol{\omega}_{\mathcal{C}/\mathcal{C}'} \tag{4.7}$$

And attempt to design a control law which will drive both quantities to zero.

In order to find a control law which will prove Lyapunov stable, a candidate Lyapunov function of a form known to produce desirable results [12] is selected:

$$V\left(\delta\boldsymbol{\sigma}\right) = 2\left[K_1\right]\ln\left(1 + \delta\boldsymbol{\sigma}^T\delta\boldsymbol{\sigma}\right) \tag{4.8}$$

And the derivative is taken:

$$\dot{V}\left(\delta\boldsymbol{\sigma}\right) = \frac{4\left[K_1\right]}{1+\delta\sigma^2}\delta\dot{\boldsymbol{\sigma}}^T\delta\boldsymbol{\sigma} \tag{4.9}$$

We use the differential kinematic equation for MRPs in Equation 4.4 and substitute it into Equation 4.9. Simplifying, we obtain the elegant result:

$$\dot{V}\left(\delta\boldsymbol{\sigma}\right) = \left[K_1\right]\delta\boldsymbol{\omega}^T\delta\boldsymbol{\sigma} \tag{4.10}$$

This can then be set equal to a known negative-definite function in $\delta\boldsymbol{\sigma}$ in order to create a control law that will satisfy the asymptotic stability criteria:

$$\dot{V}\left(\delta\boldsymbol{\sigma}\right) = \left[K_1\right]\delta\boldsymbol{\omega}^T\delta\boldsymbol{\sigma} \equiv -\delta\boldsymbol{\sigma}^T\delta\boldsymbol{\sigma} \tag{4.11}$$

From which we can readily solve for $\delta\boldsymbol{\omega}$

$$\delta\boldsymbol{\omega} = -\left[K\right]\delta\boldsymbol{\sigma} \tag{4.12}$$

Equation 4.12 is the closed-loop form of the control law, which must be related back to the measurable quantities of the hardware, input of the simulation, and command outputs before it

can be implemented. Plugging eq. 4.12 back into the definition in eq 4.7, and recognizing that $\boldsymbol{\omega}_{\mathcal{C}/\mathcal{C}'} = \boldsymbol{\omega}_{\mathcal{C}/\mathcal{B}} - \boldsymbol{\omega}_{\mathcal{C}'/\mathcal{B}}$, we obtain a control law that outputs commands that a velocity-controlled, body-mounted attitude control unit (such as the pan-tilt unit described in section 3.4 can follow):

$$\boldsymbol{\omega}_{\mathcal{C}/\mathcal{B}} = \boldsymbol{\omega}_{\mathcal{C}'/\mathcal{B}} - [K]\,\delta\boldsymbol{\sigma} \tag{4.13}$$

However, the simulation specifies attitudes and rotational velocities relative to the outer inertial frame, shown in figure 4.2 as the real frame, $\mathcal{R}$. Notionally, we must 'subtract out' the motion of the robotic platform in order to command body-relative attitude control unit(s). For the rotational velocities, this is literally true ($\boldsymbol{\omega}_{\mathcal{C}'/\mathcal{R}} = \boldsymbol{\omega}_{\mathcal{C}'/\mathcal{B}} + \boldsymbol{\omega}_{\mathcal{B}/\mathcal{R}}$), but for attitudes we must use a formulation of the successive rotation property for MRPs which relates two successive rotations, $\boldsymbol{\sigma}'\&\boldsymbol{\sigma}''$ with their combined rotation, $\boldsymbol{\sigma}$, performing an effective 'subtraction'.

$$\boldsymbol{\sigma}'' = \frac{(1-|\boldsymbol{\sigma}'|^2)\boldsymbol{\sigma} - (1+|\boldsymbol{\sigma}|^2)\boldsymbol{\sigma}' + 2\boldsymbol{\sigma}\times\boldsymbol{\sigma}'}{1+|\boldsymbol{\sigma}'|^2|\boldsymbol{\sigma}|^2 + 2\boldsymbol{\sigma}'\cdot\boldsymbol{\sigma}} \tag{4.14}$$

To preserve compactness, this expression is not substituted into the control law, but should be understood to be a requirement for computing $\delta\boldsymbol{\sigma}$ ($\boldsymbol{\sigma}_{\mathcal{C}/\mathcal{C}'}$).

The feed-forward term on the left side of eq. 4.13 compensates for the induced error of the platform's rotation, while the feed-back term on the right removes errors in attitude with an adjustable gain matrix. Taking into account that the simulation and platform rotational velocities are expressed in inertial (in this case, $\mathcal{R}$-frame) components, whereas the camera rotational velocity output must be expressed $\mathcal{B}$-frame components, the corresponding matrix equation for the control law is:

$${}^{\mathcal{B}}\boldsymbol{\omega}_{C/B} = [\mathcal{BR}]\left({}^{\mathcal{R}}\boldsymbol{\omega}_{\mathcal{C}\prime/\mathcal{R}} - {}^{\mathcal{R}}\boldsymbol{\omega}_{\mathcal{B}/\mathcal{R}}\right) - [K]\,\boldsymbol{\sigma}_{\mathcal{C}/\mathcal{C}'}) \tag{4.15}$$

Where $[BR]$ is the rotation directional cosine matrix from the $\mathcal{R}$-frame to the $\mathcal{B}$-frame.

Implementing the control law in eqn. 4.15 in an UMBRA module is relatively simple. The equation requires the rotational state information of both the pan-tilt unit, and the robotic platform, as well as the desired simulated motion. The current rotational velocity of the pan-tilt unit ($\boldsymbol{\omega}_{\mathcal{C}/\mathcal{B}}$) is

not required, for a total of five input connectors and one output connector. Note that the Attitude Compensator may call upon any information available to reduce error, as any error from attitude compensation is error in laboratory approximation of simulated motion, as opposed to simulated error.

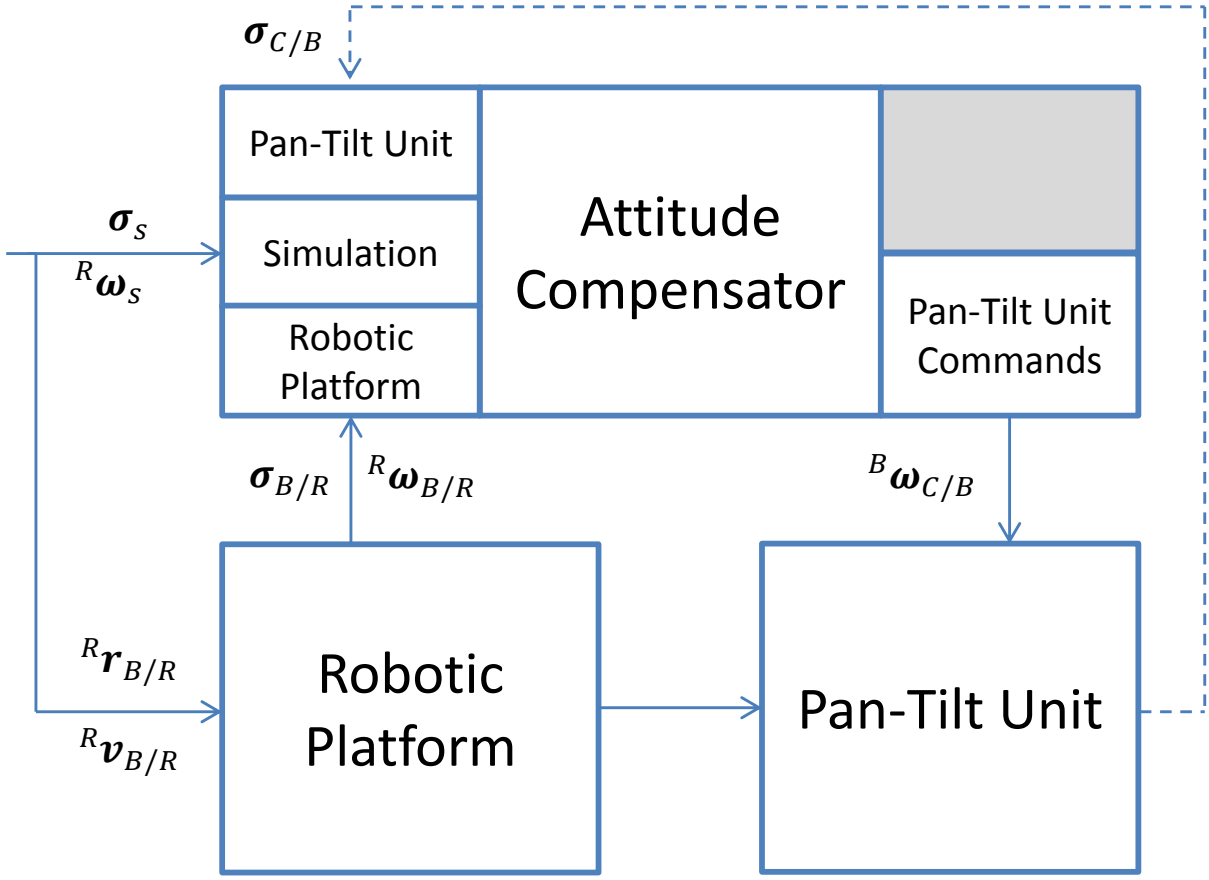

Figure 4.3: Block diagram demonstrating designed usage of the Attitude Compensator module

Figure 4.3 illustrates the usage of the Attitude Compensator module with the robotic platform and pan-tilt mount unit. The dashed line indicates a feedback connection. That is, the value used in calculating a given time step is the value from the previous time step.

4.1.4 Rotational Control Law

The rotational-proportional control law (`MathLibrary::RotationalProportional ControlLaw`) is a generic control law for causing a rigid body's attitude to follow a proscribed attitude and rotational velocity time-history. It implements a number of different feed-back and feed-forward types and mechanisms with differing properties and uses, which may be disabled to reduce the control law to a simpler form. The control law produces a commanded control torque, $\boldsymbol{u}$. In its simplest form, the control law is purely proportional to the attitude error:

$$\boldsymbol{u} = -K\delta\boldsymbol{\sigma} \tag{4.16}$$

With a scalar, positive attitude error gain K. The proof of stability for this is essentially the same as that of the attitude compensator, and is not repeated here. Note that, in this form, the control vector, u is in fact a commanded rotational velocity rather than a torque, even though the output connector for the control is named `ControlTorque`. This meets the criteria for global, asymptotic stability. However, this assumes that velocity command is possible. For most physical systems, a commanded velocity would require a secondary lower-level control loop to enforce the speed by commanding torques. This is not uncommon in robotics, however, and as such is worthy of note as an application of the module.

If a non-zero positive-definite gain matrix $[P]$ and valid inertia matrix $[I]$ are supplied, the module computes a control torque which accounts for rotational physics of the module, including other known external torques on the system ($\boldsymbol{L}$):

$$\boldsymbol{u} = -K\delta\boldsymbol{\sigma} - [P]\,\delta\boldsymbol{\omega} + [I]\,(\dot{\boldsymbol{\omega}}_r - [\tilde{\boldsymbol{\omega}}]\,\boldsymbol{\omega}_r) + [\tilde{\boldsymbol{\omega}}]\,[I]\,\boldsymbol{\omega} - \boldsymbol{L} \tag{4.17}$$

Which results from the Lyapunov function

$$V(\boldsymbol{\sigma}, \boldsymbol{\omega}) = \frac{1}{2}\delta\boldsymbol{\omega}^T\,[I]\,\delta\boldsymbol{\omega} + 2K\log\left(1 + \delta\boldsymbol{\omega}^T\delta\boldsymbol{\omega}\right) \tag{4.18}$$

differentiated in the body-frame and set equal to a negative-semidefinite function of $\boldsymbol{\omega}$:

$$\dot{V}(\boldsymbol{\sigma}, \boldsymbol{\omega}) = \delta\boldsymbol{\omega}^T\left([I]\,\frac{\mathcal{B}_d}{dt}(\delta\boldsymbol{\omega}) + K\boldsymbol{\omega}\right) \equiv -\delta\boldsymbol{\omega}^T\,[I]\,\delta\boldsymbol{\omega} \tag{4.19}$$

Since eq. 4.19 says nothing regarding $\delta\boldsymbol{\sigma}$, investigation of higher-order derivatives of V where $\delta\boldsymbol{\omega} = 0$ is needed to determine stability. [13] The second derivative of V is zero for $\delta\boldsymbol{\omega} = 0$, however the third derivative of V,

$$\dddot{V}(\boldsymbol{\sigma}, \delta\boldsymbol{\omega} = 0) = -2K^2 \delta\boldsymbol{\sigma}^T [I]^{-1} [P] [I]^{-1} \delta\boldsymbol{\sigma} \tag{4.20}$$

is negative-definite on the range $\delta\boldsymbol{\omega} = 0$ because the inertia matrix ($[I]$) and the velocity error gain matrix ($[P]$) are positive-definite. Thus, the control is globally, asymptotically stabilizing. Note that this stability guaranteed only holds if the external torque, $\boldsymbol{L}$, is known. If an unmodeled torque (that is, not accounted for by u) is applied to the system, then the Lyapunov function in equation 4.19 gains a term for the difference in torque, $\Delta\boldsymbol{L}$, and is no longer negative semidefinite, and is no longer globally stabilizing. However, if $\Delta\boldsymbol{L}$ is bounded, then as it causes increases in velocity errors, the negative-definite term will dominate the equation, driving the errors back down. Thus, the control will retain Lagrange stability characteristics and will remain within some neighborhood of the desired state.

In order to make the control more resilient to unmodeled torques, an additional state vector can be introduced:

$$\boldsymbol{z}(t) = \int_0^t (K\delta\boldsymbol{\sigma} + [I]\,\delta\dot{\boldsymbol{\omega}})\, dt \tag{4.21}$$

which will increase over time without bound if $\delta\boldsymbol{\sigma}$ remains nonzero. If the inertia matrix is constant, then this can be simplified:

$$\boldsymbol{z}(t) = \int_0^t K\delta\boldsymbol{\sigma} dt + [I](\delta\boldsymbol{\omega} - \delta\boldsymbol{\omega}_0) \tag{4.22}$$

This new state vector is then added to the Lyapunov function:

$$V(\boldsymbol{\sigma}, \boldsymbol{\omega}, \boldsymbol{z}) = \frac{1}{2}\delta\boldsymbol{\omega}^T [I]\,\delta\boldsymbol{\omega} + 2K\log\left(1 + \delta\boldsymbol{\omega}^T\delta\boldsymbol{\omega}\right) + \frac{1}{2}\boldsymbol{z}^T [K_I]\,\boldsymbol{z} \tag{4.23}$$

where $[K_I]$ is a positive definite gain matrix. Again, the derivative is taken and set equal to a negative-semidefinite quantity:

$$\begin{aligned}\dot{V}(\boldsymbol{\sigma}, \boldsymbol{\omega}, \boldsymbol{z}) &= (\delta\boldsymbol{\omega} + [K_I]\,\boldsymbol{z})^T ([I]\,\delta\dot{\boldsymbol{\omega}} + K\boldsymbol{\sigma}) \\ &\equiv -(\delta\boldsymbol{\omega} + [K_I]\,\boldsymbol{z})^T [P](\delta\boldsymbol{\omega} + [K_I]\,\boldsymbol{z})\end{aligned} \tag{4.24}$$

where $[P]$ is the same positive-definite velocity error gain matrix as before. Once more, it is necessary to investigate the higher order derivatives to determine the extent of stability for the range in which $\dot{V} = 0$: $\delta\boldsymbol{\omega} + \boldsymbol{z} = 0$. The third derivative of the Lyapunov function is:

$$\dddot{V}\left(\boldsymbol{\sigma}, \delta\boldsymbol{\omega} + \boldsymbol{z} = 0\right) = -2K^2\boldsymbol{\sigma}^T\left[I\right]^{-1}\left[P\right]\left[I\right]^{-1}\boldsymbol{\sigma} \tag{4.25}$$

which is negative-definite on the remaining state vectors. Hence, solving this for a control law, then, will guarantee global asymptotic stability. The control law derived from this Lyapunov function is:

$$\begin{aligned}\boldsymbol{u} \quad &= -K\delta\boldsymbol{\sigma} - \left(\left[P\right] + \left[P\right]\left[K_I\right]\left[I\right]\right)\delta\boldsymbol{\omega} - K\left[P\right]\left[K_I\right]\int_0^t \boldsymbol{\sigma}dt \\ &\quad + \left[P\right]\left[K_I\right]\left[I\right]\delta\boldsymbol{\omega}_0 + \left[I\right]\left(\dot{\boldsymbol{\omega}}_r - \left[\tilde{\boldsymbol{\omega}}\right]\boldsymbol{\omega}_r\right) + \left[\tilde{\boldsymbol{\omega}}\right]\left[I\right]\boldsymbol{\omega} - \boldsymbol{L}\end{aligned} \tag{4.26}$$

The new terms added to the control law are called integral feedback.

Adding an unmodeled constant torque (ΔL) to this system once again makes the Lyapunov derivative of eqn. 4.23 not negative-semidefinite. The same argument applies that for bounded unmodeled torques, the error term will dominate and stabilize, so the control law retrains Lagrange stability, and none of the states may become unbound. However, $\boldsymbol{z}$ must grow without bound unless $\delta\boldsymbol{\sigma} = 0$. Thus, the state vector grows until it compensates for the unmodeled torque, counteracting it.

4.1.5 Splitters, Joiners & Basic Math

In addition to the more math-centric modules, a number of utility modules are provided to make working with inputs and outputs of modules easier. Splitter modules allow access to individual components of vector outputs, such as standard vectors (`std::vector<>`) or UMBRA vectors (`umb::Vec3d`). Joiner modules permit assembly of various components back into vectors. Basic math modules permit simplistic operations on vectors and attitudes often needed to join module groups, such as addition or subtraction. While somewhat trivial in scope, these modules are nevertheless important for building up more complex systems. They are employed frequently in the bundles described in the following sections, but are usually left implicit.

4.2 Hardware Interfaces

The hardware interface components are libraries which wrap low-level hardware control of the equipment described in chapter 3 into UMBRA modules and systems. Typically, they are compiled against a vendor-specific control library, though some may use direct serial-port communication or other interfaces.

4.2.1 Robot Module

The robot module, contained in the component library `Pioneer`, connects to the Pioneer 3-DX described in section 3.1. A pre-existing component of the laboratory's software [9], it uses the vendor-supplied ARIA libraries to communicate with the robot via serial port. The module forwards the position or velocity input commands to the robot, and reports the robot's position, orientation, and velocity by querying the on-board sensors. The associated Tcl scripts create bundles which wrap the module with control laws, convenient inputs, and provide programmatic capabilities such as pausing or user interface panels. (These are described in greater detail in section 4.3.1.)

4.2.2 Pan & Tilt Unit Module

The interface to the real pan-tilt unit, contained in the component library `Dpptu`, is a C++ module which accesses the serial port and uses codes in a vendor-supplied C header file to communicate with the unit. The unit is intended to be commanded by specified pan and tilt position values, with speeds set less frequently. The UMBRA wrapper module exposes the position drive mode and creates a velocity drive mode by commanding the limit position in the direction of turn and resetting the speed for each time step. The module also queries for, and then respects the physical limitations of the pan-tilt unit.

4.2.3 Camera Module

The camera module is a component library maintained and distributed by Sandia National Labs' UMBRA group. It interfaces with Microsoft's DirectX graphics libraries to make any standard camera attached to the computer available to UMBRA. The library also includes modules for a few other limited camera types in frequent use by their customers and in their own labs. The AVSLab Framework extends this library by creating wrapper Tcl scripts that create bundles with cameras as positionable objects, described in section 4.3.3.

4.3 Virtual Hardware

The virtual hardware components are designed to emulate the hardware described in chapter 3. The virtual hardware can be, within physical limitations, intermixed with the real hardware in the virtual tier of the framework. The virtual hardware libraries carry all of the scripts and modules for working with that type of hardware which are not specifically tied to interfacing with real hardware. The corresponding hardware interface libraries load the virtual hardware components implicitly, so that loading a hardware library loads both real and virtual versions, while loading the virtual version explicitly circumvents loading of hardware interface code.

4.3.1 Virtual Robot

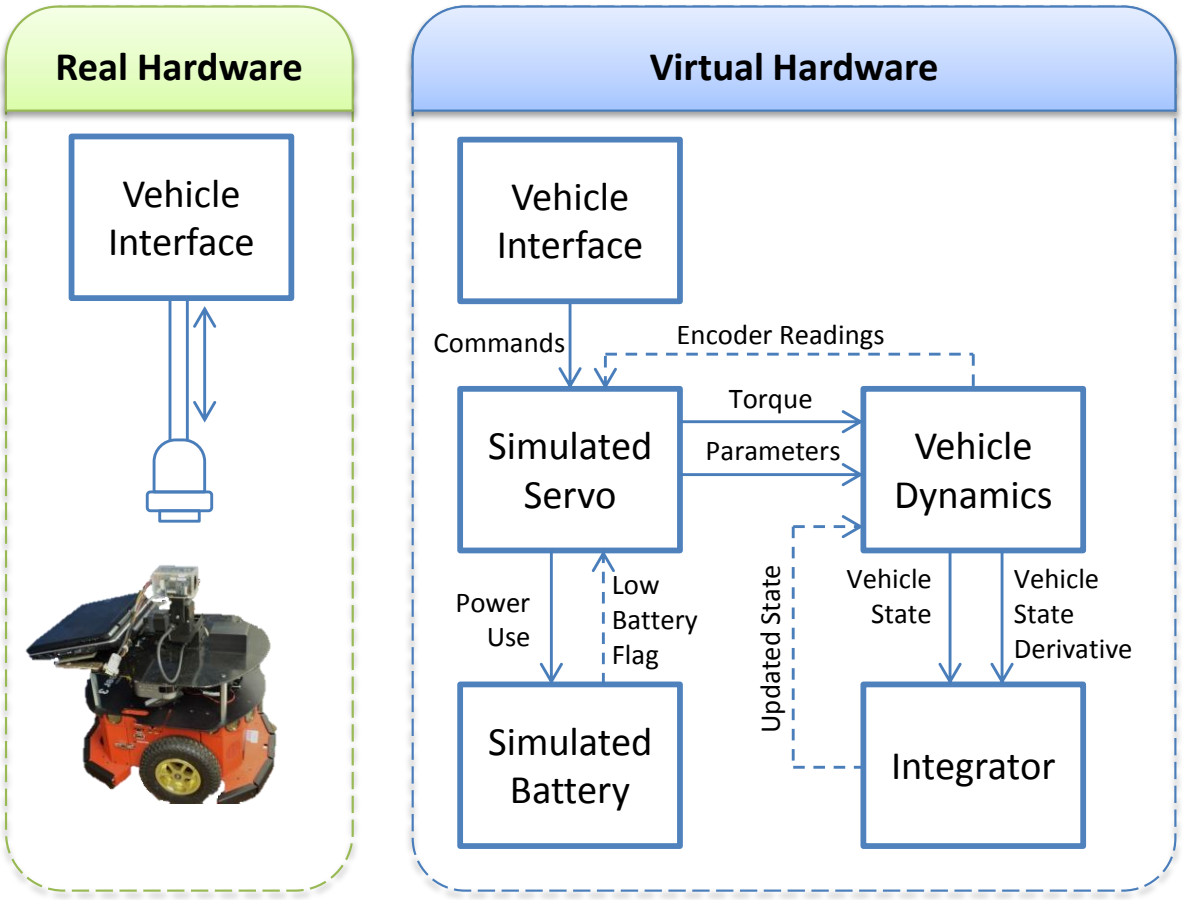

Figure 4.4: Block diagram showing the differences between the real hardware and virtual hardware

The virtual robot, a pre-existing [10] virtual-layer library in the component project `PioneerSim`, provides a software emulation of the Pioneer 3-DX described in section 3.1. Figure 4.4 illustrates the difference between the virtual robot and the real robot. Whereas the real robot is a single UMBRA module which talks to the hardware over a serial cable, the virtual robot is split up into a number of emulation modules, concealed (i.e., unconnected to outside modules) as a system behind the vehicle interface module.

The vehicle interface module (`PioneerSim::Pioneer`) mimics as closely as possible the real hardware interface, aiming for near drop-in compatibility. (In practice, separate, though largely sim-

ilar, wrapper scripts are still required for each.) The simulated servo module (`PioneerSim::SimServo`) acts as a simulation of the electronic controller and servo motors on-board the vehicle, issuing generated torques and reading encoder returns from the wheels. The vehicle dynamics module (`PioneerSim::CartDynamics`) contains a kinetic vehicle model with equations of motion for the body and wheels, as well as a friction model for floor slippage. A specifically-tailored integrator module (`PioneerSim::Integrator`), integrates the equations of motion, and a simulated battery module (`PioneerSim::BatterySim`) tracks power drain from the servo motors.

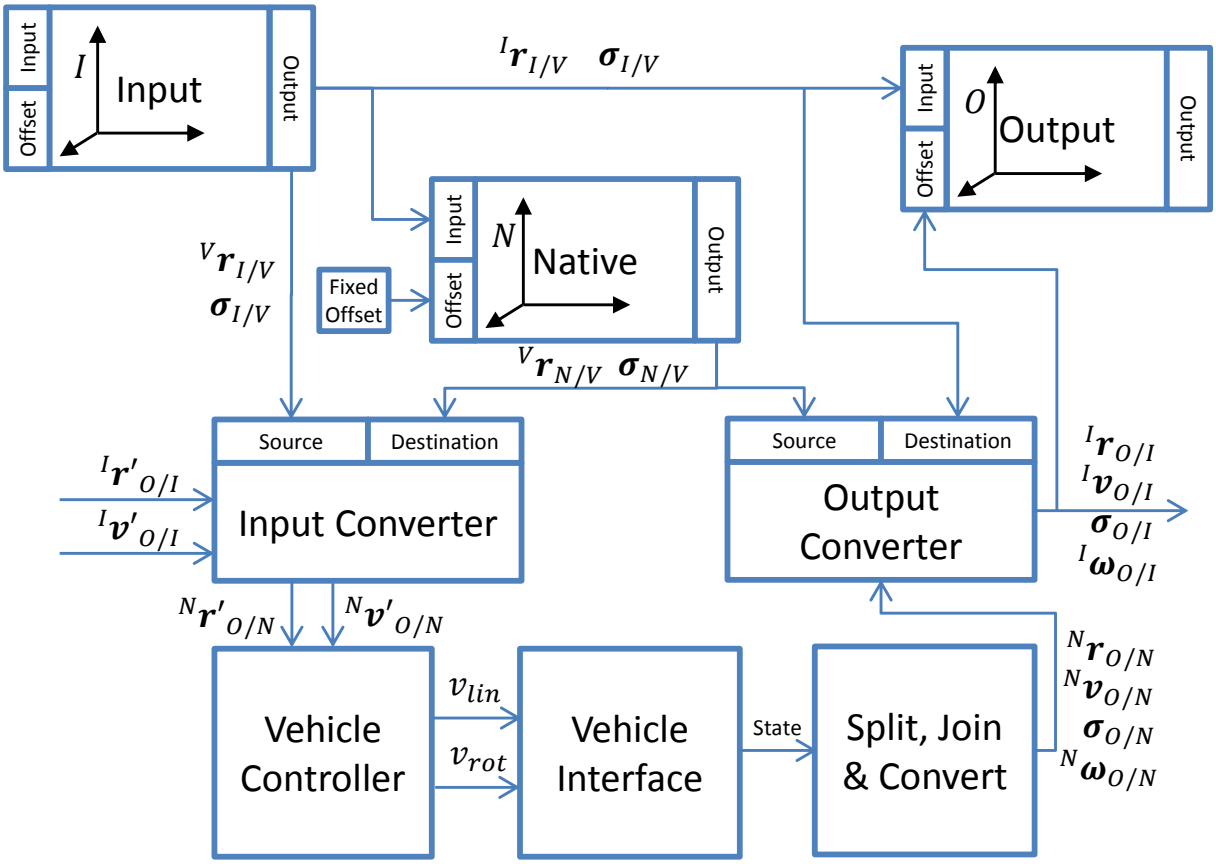

Figure 4.5: Block diagram of the wrapping bundle around the virtual/real vehicle interface

The wrapper scripts create a bundle around the vehicle interface, real or virtual, which translates the expected inputs into the proper coordinate frames, provides a control law for the

vehicle, and parses the outputs of the module back into standard forms. Three frames are used for the vehicle. As part of the `IHardware` interface, it has an input and output frame, with the input frame being the starting point at which the robot is initialized and the output being the current position and orientation. The present configuration presumes the input frame is inertial, as it would essentially be for any lab simulation. Modifications would be necessary to handle the effects of a rotating frame, such as if the vehicle were used on a turning table. The native frame is at a fixed offset from the input frame, and provides a means to translate the coordinates the vehicle uses into the standard coordinates used by the rest of the simulation. Two frame converter modules are used to do most of the translation between the native frame and the input frame.

The vehicle is commanded with a desired position and/or desired velocity. The vehicle controller module (`PioneerSim::VehicleController`) uses a control law previously developed for use with the virtual robot to home in on the correct trajectory. [10] The vehicle interface may be either the real vehicle interface, or the virtual vehicle interface along with its hardware emulation. Outputs are collected from the virtual vehicle interface, and a series of splitter, joiner, and converter modules use essentially trivial operations to put the measurements into the appropriate formats for output. Note that specifying position and velocity dictates orientation and rotational velocity, which must be compensated by the attitude compensator module, described in section 4.1.3.

4.3.2 Virtual Pan & Tilt Unit

The virtual pan-tilt unit module, contained in the `VirtualDpptu` library, emulates the hardware of the Directed Perception pan-tilt mount unit. It keeps track of the orientations of the pan and tilt motors independently, integrating the speed commands it receives for each direction. The unit is a medium-fidelity simulation. Due to the relatively slow operational speed regimes and the high torque capabilities of the motors, accelerations are not modeled - speed changes are assumed to be instantaneous. However, in precision applications, discretization effects are presumed non-trivial, both in speed and in position. Although the interface accepts double-precision commands, the commands are first discretized to match the serial interface and internal capabilities of the real

pan-tilt unit. Limitations on the orientation, minimum and maximum speeds of the motors are also applied, and then internal integration is performed with double-precision.

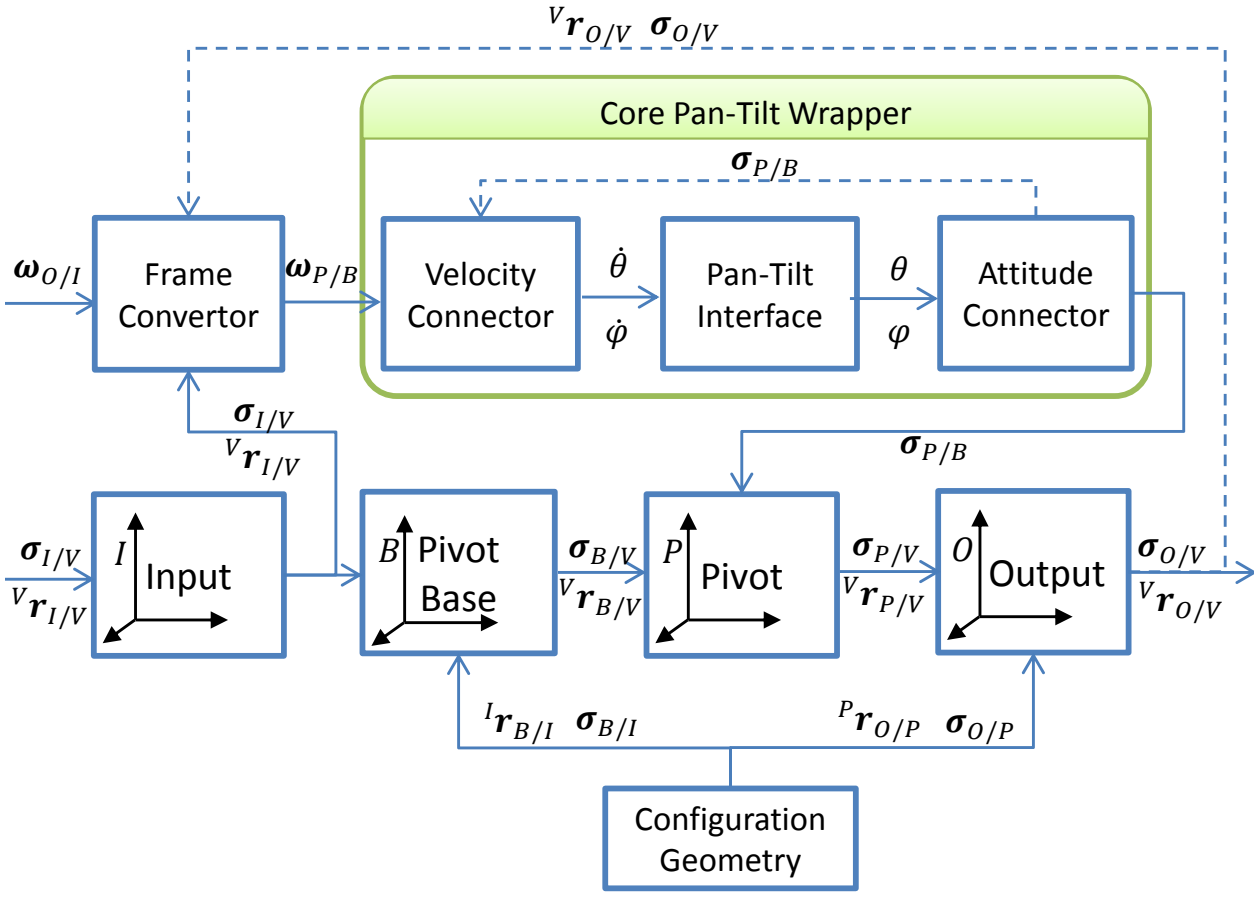

Figure 4.6: Block diagram of the pan-tilt unit interface and wrapper bundle

Figure 4.6 is a block-diagram of the wrapping of the hardware. The pan-tilt interface (`VirtualDpptu::Dp`) module represents the real or virtual hardware, which is driven by pan & tilt speed commands. The attitude connector (`VirtualDpptu::AttitudeConnector`) module serves to convert the reported pan and tilt positions into the more generalized rotation measurements used by the rest of the simulation. The velocity connector module (`VirtualDpptu::VelocityConnector`) works in much the opposite way, converting the supplied rotational velocity vector into pan and tilt direction speed commands. In order to make the conversion, the velocity converter needs to

know the current attitude of the pan-tilt unit, and thus has a feedback connection from the output attitude. This is the 'core' wrapping which exposes the hardware as a point rotation actuator, without any shape or size.

The remainder of Figure 4.6 shows the fully-wrapped pan-tilt unit, as it is intended to be used in simulations. The wrapped core as described above is, in turn, connected to a frame chain which relates the output mounting plate of the unit to the pivot point of the unit to the input base of the unit. The converted output of the hardware controls the offset rotation of the pivot frame with respect to the pivot base frame. The pivot base frame represents the default (zero) orientation of the hardware after calibration. In order that the hardware may be mounted in an arbitrary position and orientation, a frame converter is used to convert the rotational velocity components in the base frame to the pivot base frame used by the velocity connector to the hardware.

4.3.3 Virtual Camera

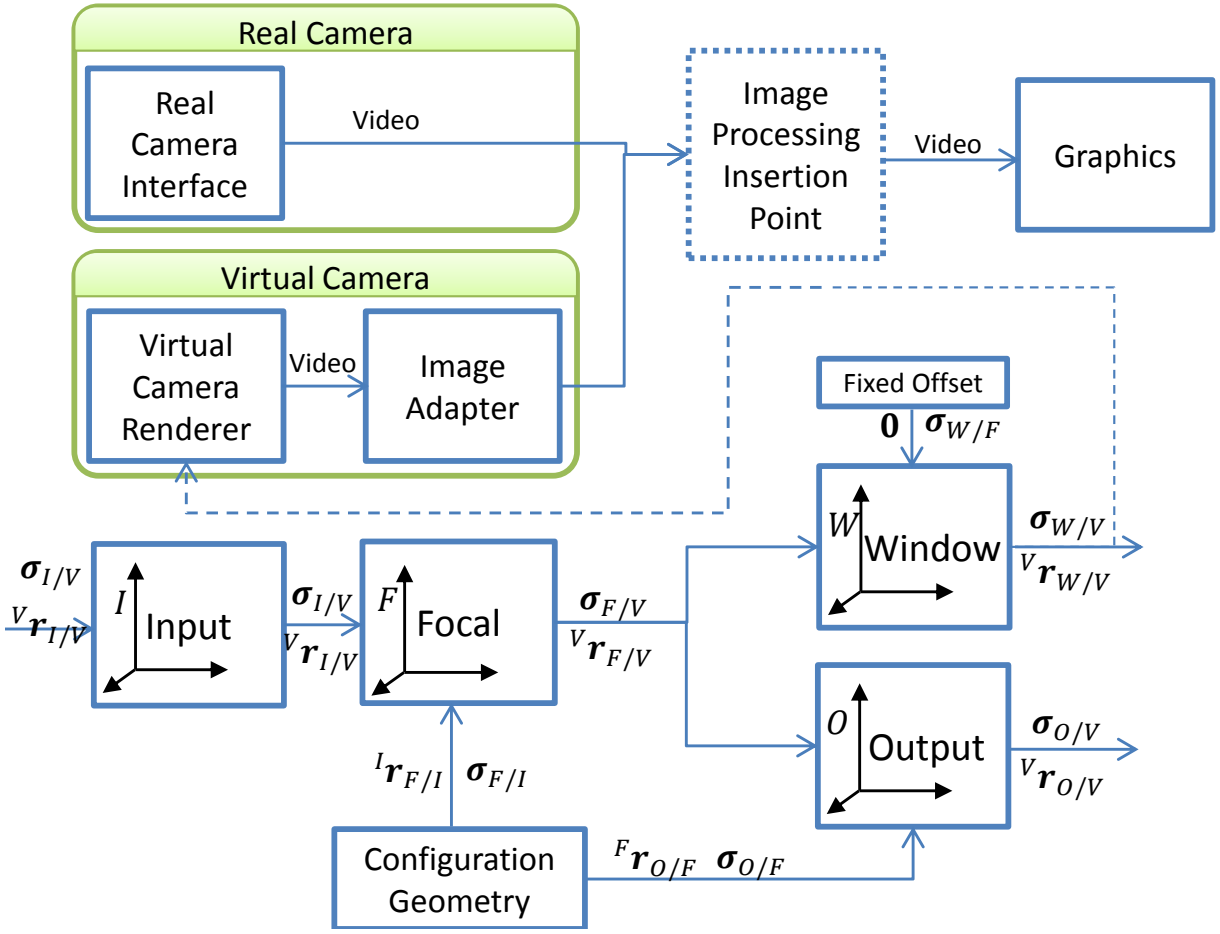

Figure 4.7: Block diagram of the real and virtual cameras

The virtual camera is a new module which wraps an OpenGL rendering window attached to one of the visualization spaces described in section 2.3.2. (Due to oddities in the setup process for the composite window hosting all 3D drawing windows, the rendering windows for the cameras must be reserved at GUI creation time.) By default, the rendering windows attach to the virtual laboratory visualization space, but may be changed to view the simulation if desired. The video is passed as a series of images rendered from the scene and held in memory. The camera uses an `ImageFilter::OsgImageAdapter` to convert the in-memory images into a format used by the other video processing modules available in the framework.

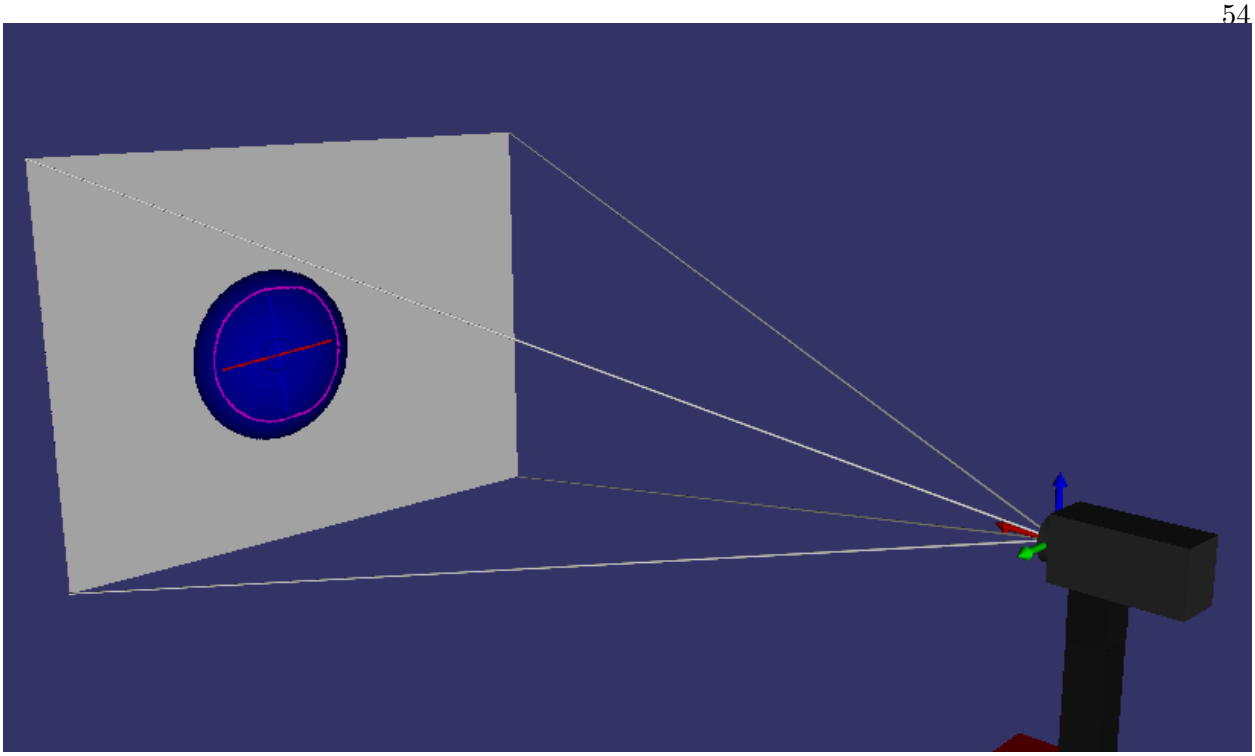

Figure 4.8: Visualization of the virtual camera in action, with visual snakes tracking an object

For both the virtual camera and the hardware interface to the real camera, wrapper scripts create bundles which include representations of the camera itself drawn in all of the visualization spaces. The video output is also rendered on a 'screen' object in front of the camera in the overview visualization space to give feedback on what the camera is currently viewing. Tcl script functions allow for easy insertion of (or splitting for additional) video processing (such as the color pressure snakes described in section 4.5.1). The input frame of the camera is considered the be the mounting point on the lower surface of the camera housing, while the output frame is the focal frame of the camera. An additional output frame allows other OpenGL windows to be attached to the viewpoint of the camera for examining the differences between virtual and real camera output or monitoring orientation of the camera in a separate visualization space or window.

4.4 Orbit Simulation

The relative orbital simulation library, contained in the component library `OrbitSim`, calculates the dynamics of satellites in close orbits relative to one satellite in the formation, designated the 'chief'. (Other satellites in the formation are considered the 'deputy' satellites.) Calculation of dynamics between the satellites is is handled by the `OrbitSim::RelOrbitSim` module, which calculates and outputs the full states of all vehicles as a `std::vector<double>`.

Early versions of the component library tied this simulation directly to the robotic motion platform with the `OrbitSim::ROPI` module serving as a mediator, however this was deemed insufficiently scalable for more generalized simulations. The functionality of `OrbitSim::ROPI` was refactored into three modules. `OrbitSim::CraftSelector` extracts the state information from the output of `OrbitSim::RelOrbitSim` and converts it into standardized outputs of position, velocity, and acceleration. `OrbitSim::LocalFrameConverter` calculates the instantaneous local orbital plane of a deputy satellite relative to the chief, as well as the inverse transformation. A calculation flag determines whether the local instantaneous plane is calculated at every time step, or only at one time step and held constant.

The final module created from `OrbitSim::ROPI` was a position and velocity control law to cause the robotic vehicle to approach and then follow a simulated position & velocity history. This module was moved to the virtual robot component library, as it is a key part of controlling the vehicles and more naturally belongs with them.

4.5 Visual Tracking

The visual tracking library, contained in component library `VisualSnake`, provides modules related to tracking targets with visual sensing equipment. The components of the visual tracking library may be considered a part of either the virtual tier or simulation tier depending on their role in a specific simulation.

4.5.1 Visual Snakes

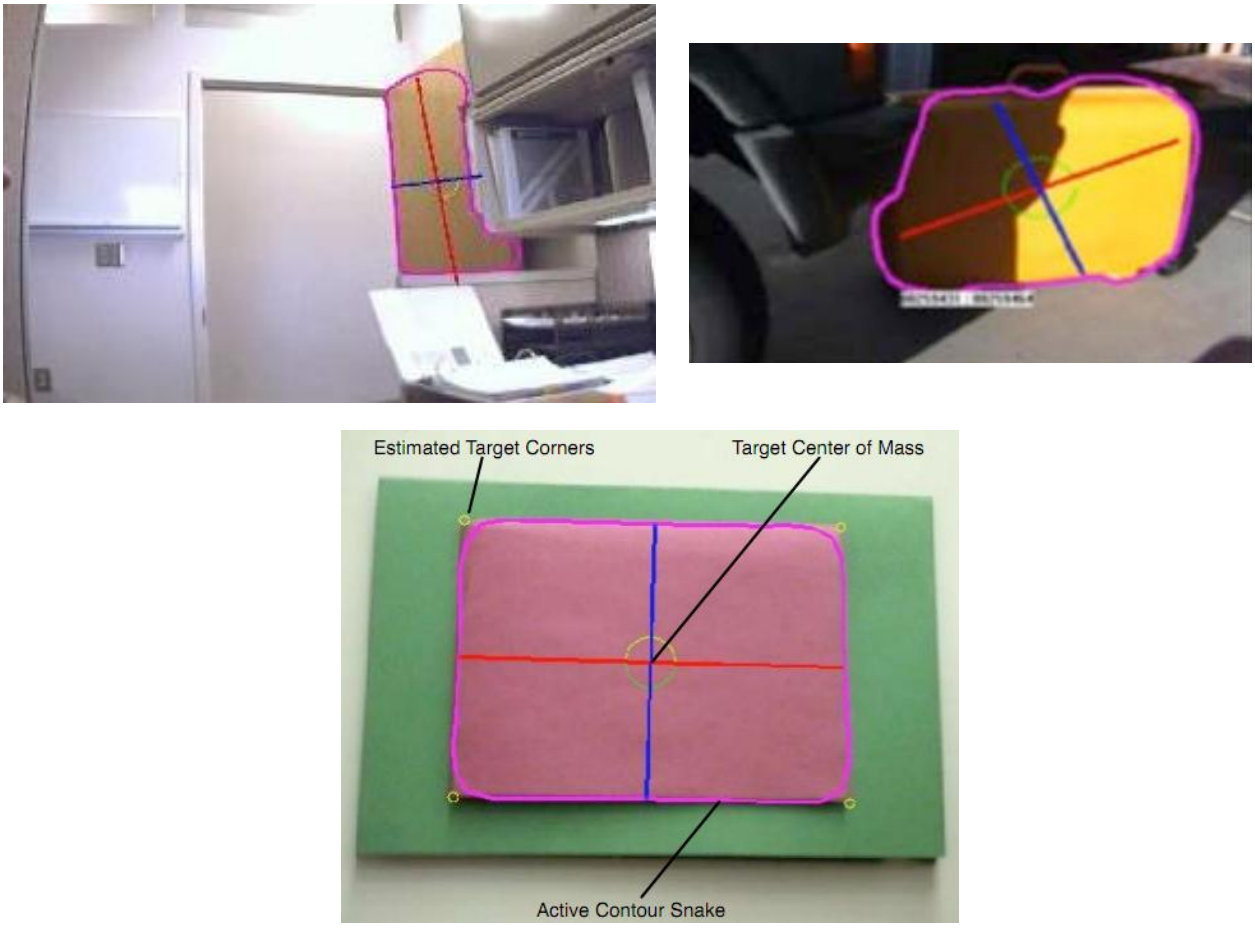

Figure 4.9: Visual snakes tracking targets in various conditions

The primary means of visual tracking is the `VisualSnake::SnakeMod` module, which implements a statistical color-pressure gradient snake algorithm [8]. The module takes in a video feed as a series of image frames, and breaks them down into a color space using Intel's OpenCV library. A single point is selected as belonging to an object to track, by a human operator or by a separate algorithm, and around that point a small set of points connected in a loop (the 'snake') are created. An artificial 'pressure' force is added to cause the snake to expand, while differences in color space components are treated as an opposing pressure gradient. In essence, the snake expands until it reaches the visible edges of the object being tracked.

The snake algorithm is robust to partial occlusions, smoothly adapting to fill in where blocked and then expanding when the blocking object is removed. Sensitivity gains can be used to increase or decrease the aggressiveness of the algorithm. The apparent size and center of the object are the primary outputs of the module. By studying the curvature of the snake, a priori information about the shape and size of the target can be used to make more detailed estimations of the target's true location in the image frame, orientation, and distance. Different choices of color space decomposition provide differing behaviors, with corresponding tracking strengths and weaknesses, in different types of lighting conditions, making the algorithm very flexible in application.

4.5.2 Virtual Snakes

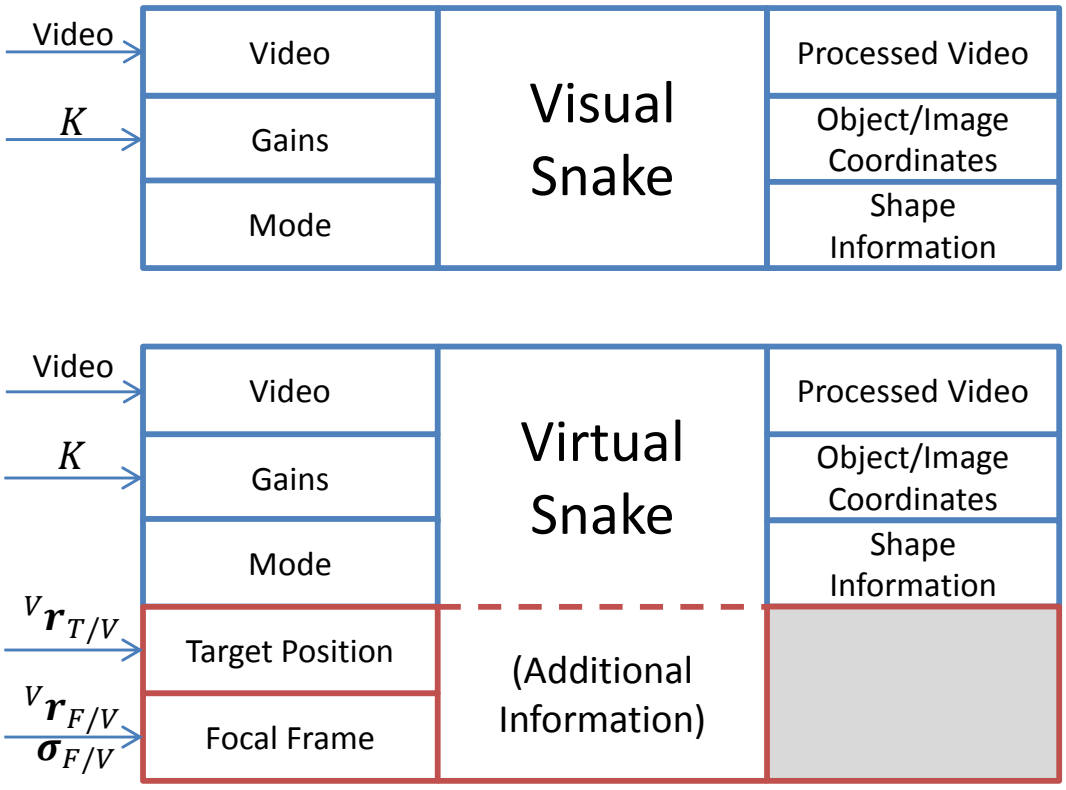

Figure 4.10: Comparison of the visual and virtual tracking snake implementations

The virtual snake module, `VisualSnake::VirtualSnakeMod` is a new fast-running emulation of the operation of the actual virtual snake module. Although the visual snakes module is purely software, the amount of image processing required to use it presents some limitations for multi-craft scenarios. By contrast, the virtual snakes module uses state information of the target to calculate the perfect output of the visual snakes algorithms. For larger simulations involving multi-vehicle formations with multiple visual tracking, virtual snakes can be used for most of the vehicles and actual visual snakes used for spot-checking of control behaviors with imperfect information. The virtual snake also provides a point of reference for performance of the visual snakes.

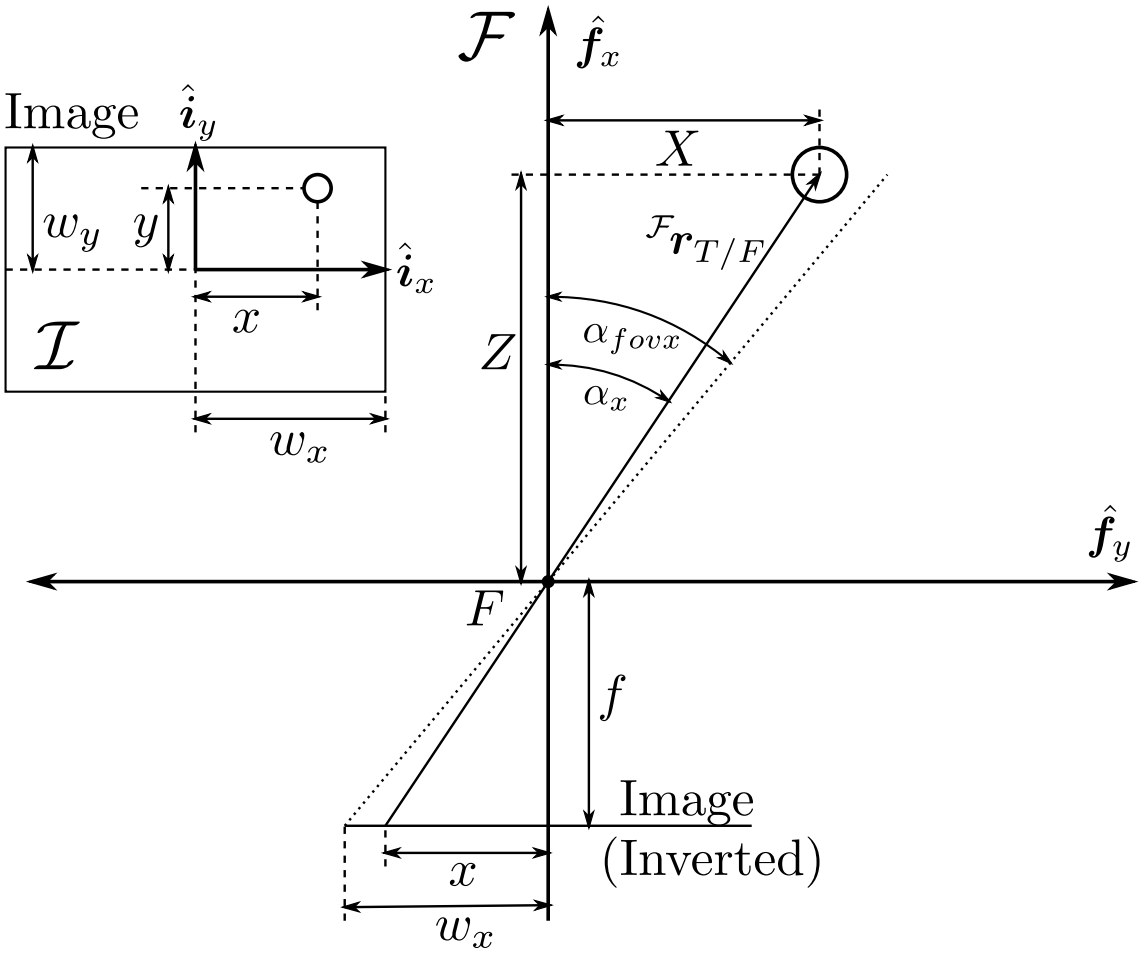

Figure 4.11: Illustration of the focal frame breakdown of coordinate information for the Virtual Snake

Figure 4.11 shows the focal frame, $\mathcal{F}$, constructed about the focal point of the camera, F. A target (depicted as a sphere) exists at a position ${}^{\mathcal{F}}\boldsymbol{r}_{T/F}$. A pinhole camera model with horizontal field of view angle α_{fovx} is used to construct the inverse image at a focal length f behind the focal point. The goal of the virtual snakes module is to obtain the normalized coordinates $x_n = \frac{x}{w_x}$ and $y_n = \frac{y}{w_y}$ which may then be transformed into pixel coordinates.

We begin by investigating the relationship between the tangents of the angles

$$\tan \alpha_x = \frac{x}{f} \tag{4.27}$$

$$\tan \alpha_{fovx} = \frac{w_x}{f} \tag{4.28}$$

and then substituting them into the definition of the normalized coordinate

$$x_n = \frac{x}{w_x} = \frac{\tan \alpha_x}{\tan \alpha_{fovx}} \tag{4.29}$$

Then, we apply a second definition of $\tan \alpha_x$:

$$\tan \alpha_{fovx} = \frac{X}{Z} \tag{4.30}$$

Which leads to

$$\rightarrow x_n = \frac{X}{Z} \cot \alpha_{fovx} \tag{4.31}$$

The lengths X and Z can be found by using the position vector and the frame unit vectors:

$$x_n = \frac{{}^{\mathcal{F}}\boldsymbol{r}_{T/F} \cdot \hat{\boldsymbol{f}}_2}{{}^{\mathcal{F}}\boldsymbol{r}_{T/F} \cdot \hat{\boldsymbol{f}}_1} \cot\left(\alpha_{fovx}\right) \tag{4.32}$$

A similar equation can be derived for the image-vertical ($\hat{\boldsymbol{i}}_y$) direction:

$$y_n = \frac{{}^{\mathcal{F}}\boldsymbol{r}_{T/F} \cdot \hat{\boldsymbol{f}}_3}{{}^{\mathcal{F}}\boldsymbol{r}_{T/F} \cdot \hat{\boldsymbol{f}}_1} \cot\left(\alpha_{fovy}\right) \tag{4.33}$$

More commonly, however, only one field of view is specified. With hardware, this is usually the horizontal field of view, α_{fovx}, along with an aspect ratio. The relationship between the two is:

$$a_r = \frac{w_x}{w_y} = \frac{w_x}{f} \cdot \frac{f}{w_y} = \tan\left(\alpha_{fovx}\right) \cot\left(\alpha_{fovy}\right) \tag{4.34}$$

thus,

$$y_n = \frac{{}^{\mathcal{F}}\boldsymbol{r}_{T/F} \cdot \hat{\boldsymbol{f}}_3}{{}^{\mathcal{F}}\boldsymbol{r}_{T/F} \cdot \hat{\boldsymbol{f}}_1} \cot\left(\alpha_{fovx}\right) \cdot a_r \tag{4.35}$$

Knowing the size of the desired, emulated image in pixels (s_x,s_y), the pixel coordinates can be calculated:

$$x_p = \left(\frac{s_x}{2}\right) x_n \tag{4.36}$$

$$y_p = \left(\frac{s_y}{2}\right) y_n \tag{4.37}$$

Other attributes of the real snake output may also be emulated by setting corresponding input values which will be passed through, though they are not calculated by the virtual snakes. This limits the virtual snakes' usefulness to that of tracking the center of an object. Distance or roll calculations based on shape or size would not be correctly emulated without further development of the module specific to such an application.

4.5.3 Attitude Error Estimator

The attitude error estimator module, `VisualSnake::AttitudeErrorEstimator`, uses the position data from the visual or virtual snake modules, combined with knowledge about the image frame and the camera that acquired it to determine the relative attitude difference between the current attitude of the camera and the vector from the camera to the tracked target. This difference may be used as an error signal which a rotational control law can drive to zero, causing the camera (and attached equipment) to follow the object being tracked.

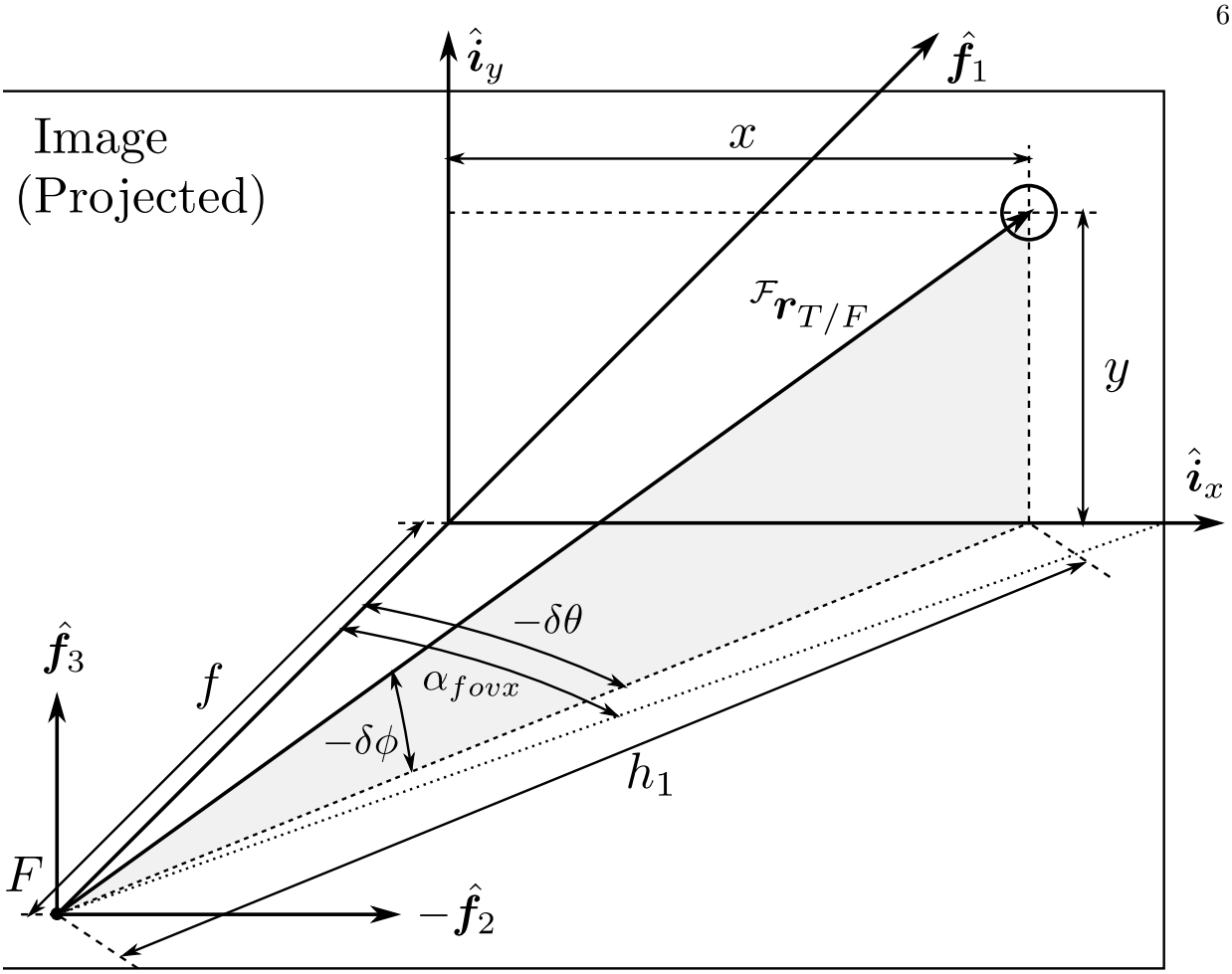

Figure 4.12: Illustration of the formulation of the attitude error estimator

Because a vector is not a full attitude, lacking orientation about itself as an axis, only two degrees of freedom can be obtained from center point information. Figure 4.12 shows the setup for deriving the formulas that relate image position and attitude error. The rotation is constructed relative to the focal frame, $\mathcal{F}$ with coordinate unit vectors $\hat{\boldsymbol{f}}_1$,$\hat{\boldsymbol{f}}_2$ and $\hat{\boldsymbol{f}}_3$. The rotation is constructed by imagining a second frame, $\mathcal{F}'$ (not pictured), whose 1-axis is coincident with the vector to the target, representing the focal frame if it were pointed at the target. The natural choice for breaking down the rotation between the two frames is an Euler 3-2-1 sequence rotation, which closely mirrors the action of the pan-tilt unit being driven ($\delta\theta$,$\delta\phi$,$\delta\psi$).

The 3-axis error, $\delta\theta$, can be derived from examining its tangent,

$$\tan\left(-\delta\theta\right) = \frac{x}{f} \tag{4.38}$$

The tangent of the horizontal field of view,

$$\tan\left(\alpha_{fovx}\right) = \frac{w_x}{f} \tag{4.39}$$

can be plugged into eq. 4.38 to yield,

$$\tan\left(-\delta\theta\right) = \frac{x}{w_x}\tan\left(\alpha_{fovx}\right) \tag{4.40}$$

And using the definition of the normalized coordinates ($x_n = \frac{x}{w_x}$), we can solve for $\delta\theta$:

$$\delta\theta = -\tan^{-1}\left(x_n \tan\left(\alpha_{fovx}\right)\right) \tag{4.41}$$

The 2-axis rotation, $\delta\phi$, can be found by investigating the cosine of $\delta\theta$,

$$\cos\left(\delta\phi\right) = \frac{f}{h_1} \tag{4.42}$$

which can be solved for h_1 by substituting in eq. 4.39,

$$h_1 = \frac{w_x}{\tan\left(\alpha_{fovx}\right)\cos\left(\delta\theta\right)} \tag{4.43}$$

and plugged into the tangent of $\delta\phi$:

$$\tan\left(-\delta\phi\right) = \frac{y}{h_1} = \frac{y}{w_x}\tan\left(\alpha_{fovx}\right)\cos\left(\delta\theta\right) \tag{4.44}$$

Recalling from eq. 4.34 the definition of the aspect ratio and the definition of the normalized coordinate, the eq 4.44 can be simplified to:

$$\delta\phi = \tan^{-1}\left(y_n \tan\left(\alpha_{fovx}\right)\frac{\cos\left(\delta\theta\right)}{a_r}\right) \tag{4.45}$$

The 1-axis rotation, $\delta\psi$, cannot be determined purely from center-point information. Thus, for the purposes of this module:

$$\delta\psi = 0 \tag{4.46}$$

This error signal is then converted from an Euler 3-2-1 sequence rotation to MRPs (and other formats).

CHAPTER 5

DEMONSTRATION SIMULATION

In order to fully demonstrate the capability and readiness of the framework and component integration, an example simulation was prepared for execution with both real hardware and virtual hardware. The simulation uses most of the components already present in the framework, establishing a baseline simulation for later expansion.

5.1 Concept

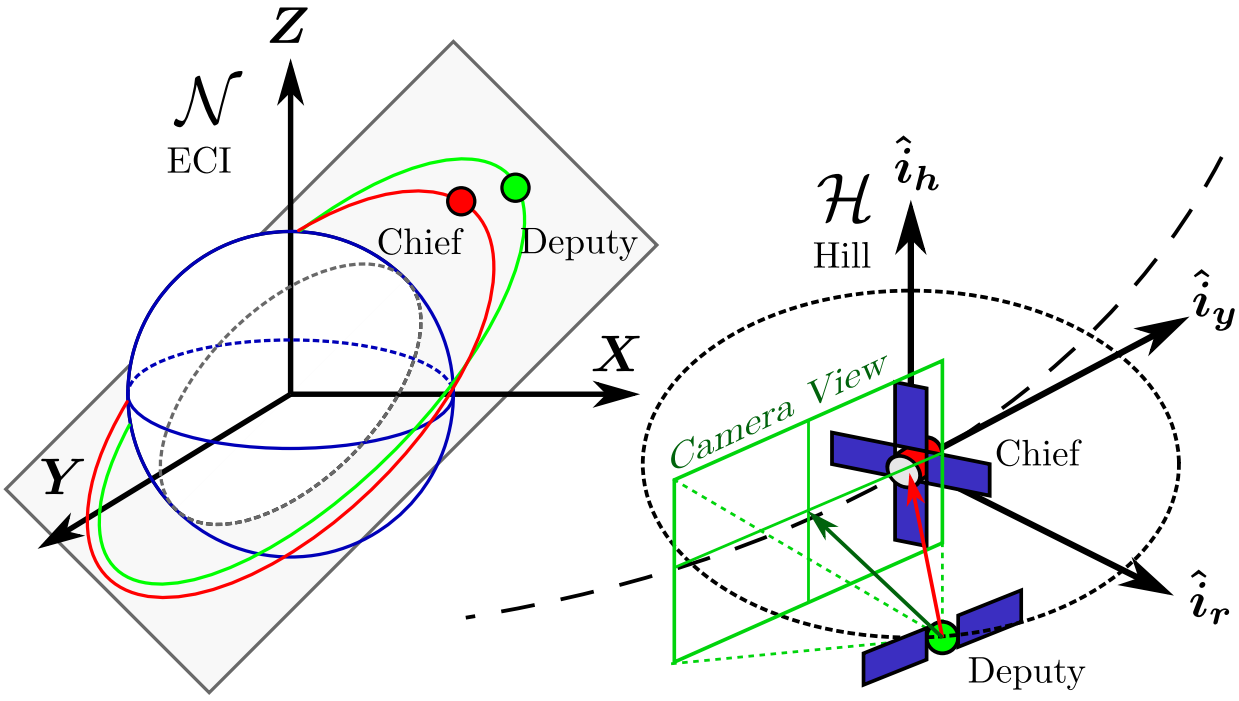

Figure 5.1: Demonstration visual tracking simulation with a two-satellite formation

The demonstration simulation features two satellites, shown in Figure 5.1 as the "chief" (red) and "deputy" (green). Choice of deputy and chief is arbitrary in a given formation and may be chosen for convenience; here a larger, more central satellite is chosen to be the chief. The satellites and their orbits are depicted in inertial space (Earth-Centered Inertial, $\mathcal{N}$) on the left side of the figure. The right side of the figure shows the same scenario cast in the more intuitive chief-satellite Hill frame ($\mathcal{H}$). The deputy satellite's eccentric orbit produces an elliptical relative orbit in the $\hat{\boldsymbol{i}}_{\boldsymbol{r}}$-$\hat{\boldsymbol{i}}_{\boldsymbol{y}}$ plane.

The green projection from the deputy satellite shows an example field of view for an on-board, body-fixed camera. The green arrow is the focal axis of the camera. The deputy satellite is here charged with maintaining a centered view of the chief satellite for external inspection purposes. Because the camera is body-fixed, the deputy must re-orient itself as the chief satellite begins to wander away from the center of the field of view. The red arrow is the estimated unit position vector from the deputy to the chief. The deputy satellite must then determine a two degree-of-freedom orientation difference between this and the current focal axis, then calculate a corresponding command torque to bring the two vectors into alignment. For this simulation, rotation about the focal axis is to be uncontrolled.

The simulated deputy satellite may be thought of as a 'perfect' satellite. It exists without simulated hardware limitations. The hardware is assumed to be sufficiently sized to handle any command issued without saturation, and receives no external effects due to environment. In this way, it provides a point of reference for theoretical conditions, illustrates the errors due solely to control algorithms, and serves as a basic lower-fidelity model which may be later expanded into a much higher-fidelity model.

5.2 Assembly

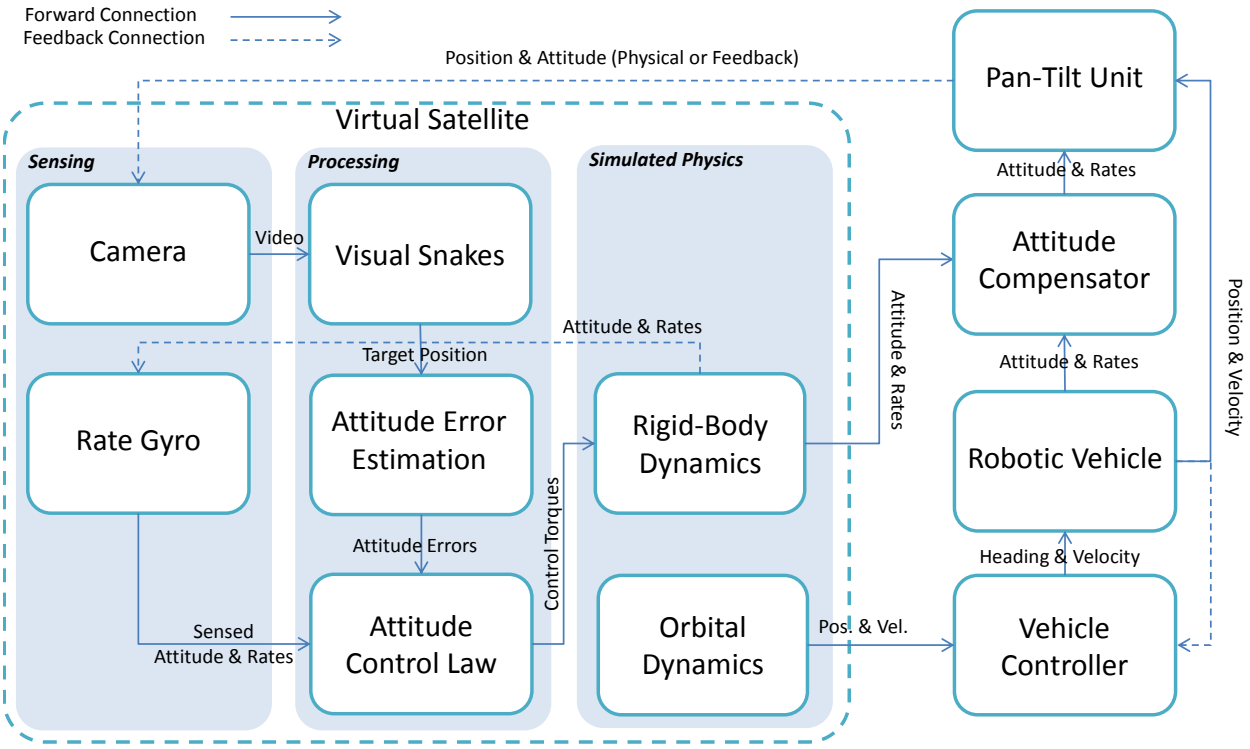

Figure 5.2: Block diagram for the demonstration simulation

Figure 5.2 depicts the block diagram for the assembly of the demonstration simulation from the software and hardware components previously discussed. Items on the left side of the figure are grouped into components of a virtual satellite. Despite the name 'virtual satellite', these modules are grouped within the simulation tier. Note the progression within the simulation tier of sensing, command, and then physics, reflecting the data flow model discussed in section 2.3.5. To the right side of the figure are modules that exist in the virtual tier. They are either interfaces to hardware or controls built on the hardware. Note that figure 5.2 is a high-level representation: many modules are required to implement the blocks shown and translate between them.

The assembly of the simulation is conducted in steps, each of which builds on the previous one to a usable demonstration. The steps of assembly are presented here as an example of incrementally

building and testing a complex simulation.

5.2.1 Controlled Rigid-Body Rotation

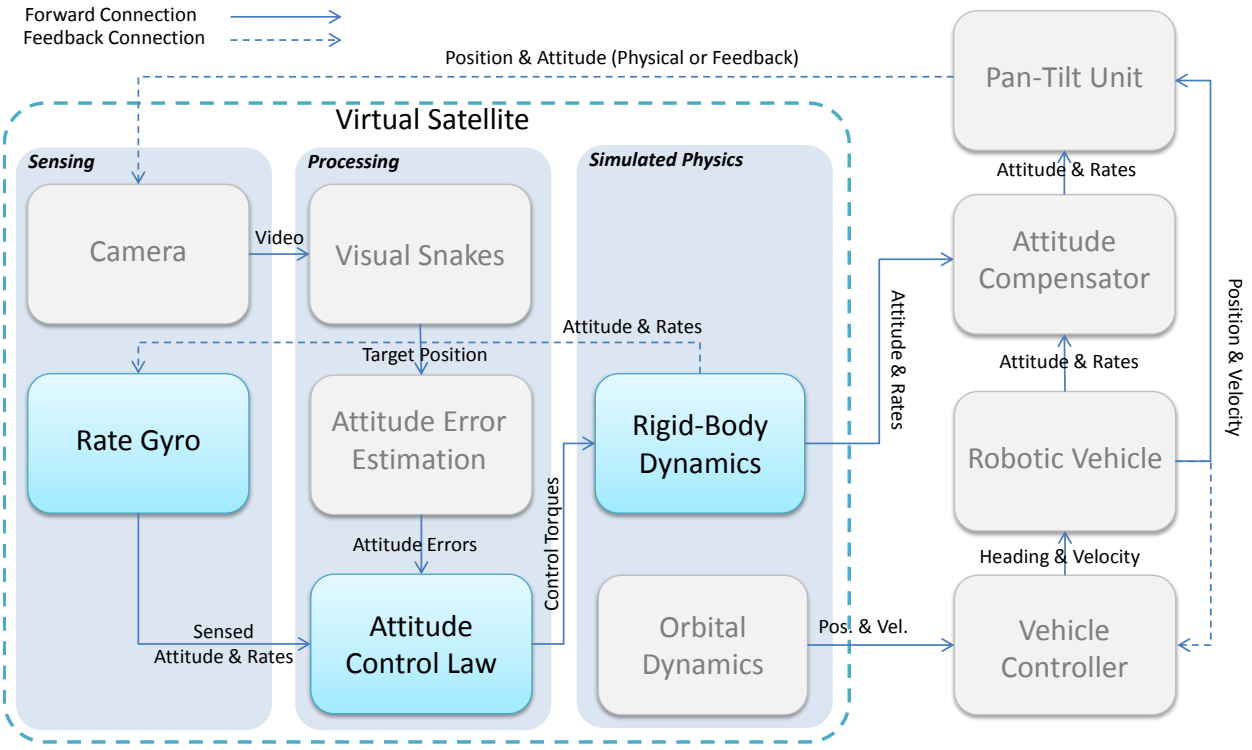

Figure 5.3: Demonstration Simulation Step 1: Core attitude propagator and control law

The first step of building the simulation involves testing the core components of the simulation, the highlighted portions of figure 5.3. The primary components are the Attitude Propagator (section 4.1.1) and the Rotational-Proportional Control Law (section 4.1.4).

The attitude propagator models the rotational physics of a tumbling rigid body, which will represent the satellite as it rotates within a localized non-rotating frame. Because the model is a numerical integrator, and accepts a general torque, this model is quite flexible and allows for later addition of other effects, such as gravitational gradients or solar pressures based on exposed profile, which would allow greater realism to be injected. However, for this simulation, the physics model

only receives control torques as dictated by the control law.

The rotational-proportional control law represents a basic attitude-stabilization algorithm and attitude control hardware. It 'senses' the attitude and rotational velocity of the satellite through a simulated gyroscope. The gyroscope, in implementation, is simply a feedback connection which passes 'perfect' sensing information, but can be expanded to include noise corruption and drift. Because this simulation does not model orbital hardware limitations, the commanded torque is added to any external torques and passed to the physics model unchanged. These three parts, taken together, form the basic core of the simulation and fit the most basic requirements of the loop cycle presented in section 2.3.5.

Quantity	Value
Initial Attitude ($\boldsymbol{\sigma}_0$)	-0.30, -0.40, 0.20
Desired Attitude ($\boldsymbol{\sigma}_d$)	0.00, 0.00, 0.00
Initial Velocity ($\boldsymbol{\omega}_0$)	0.20, 0.20, 0.20 rad/s
Desired Velocity ($\boldsymbol{\omega}_d$)	0.00, 0.00, 0.00 rad/s
Modeled Torque (L)	0.00, 0.00, 0.00 N·m
Unmodeled Torque (ΔL)	0.05, 0.00, -0.10 N·m
Attitude Error Gain (K)	1.00
Velocity Error Gain ($[P]$)	3.00$\cdot$ $[I_{3\times3}]$
Integral Gain ($[K_I]$)	0.01$\cdot$ $[I_{3\times3}]$

Table 5.1: Initial conditions for the first assembly step of the demonstration simulation

Verification at this stage is relatively straightforward. Only the attitude propagator and control law modules are created, and the attitude is simply referenced to the virtual frame. The input conditions and output of a reference implementation are shown and discussed in reference [14], in Example 8.10. The initial conditions are those of an initially tumbling body, detailed in table 5.1. The control is set to arrest the motion, and resist an unmodeled external torque. By adding a logger

to the simulation and recording the time-history of the attitude propagator from the same initial conditions and gains, identical outputs are obtained. The graphs clearly show the rotational velocity being quickly suppressed and the attitude returning to zero. The control torque vector reaches a non-zero steady state value due to the integral wind-up term of the control law, counteracting the external torque.

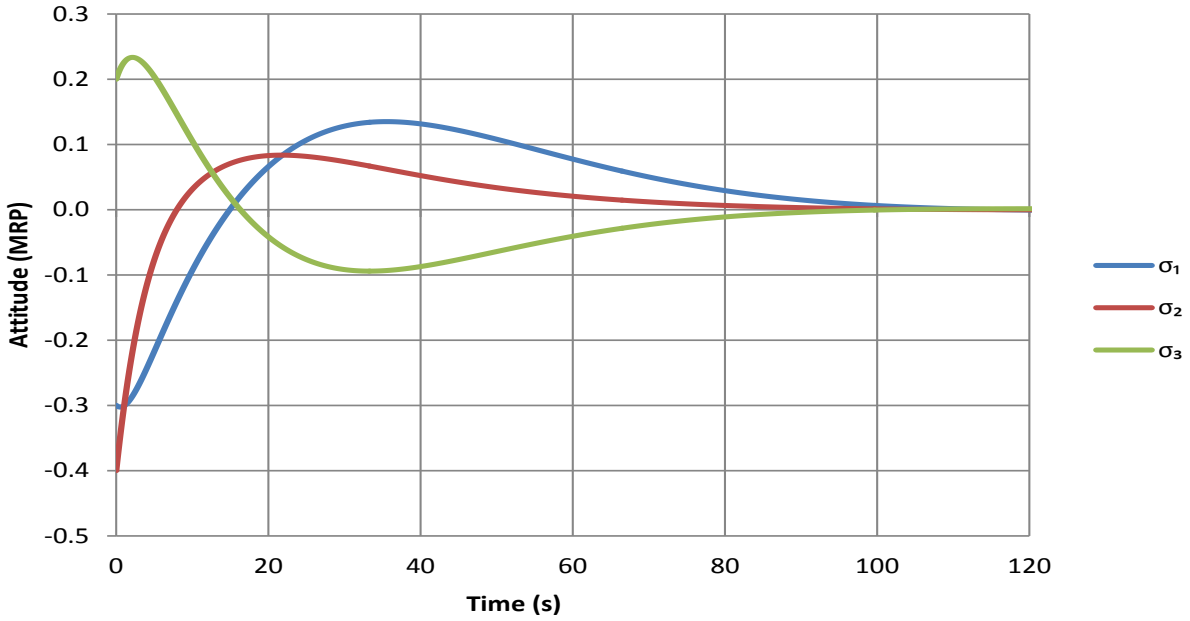

Figure 5.4: Demonstration simulation step 1 verification: Attitude in MRP components vs. Time.

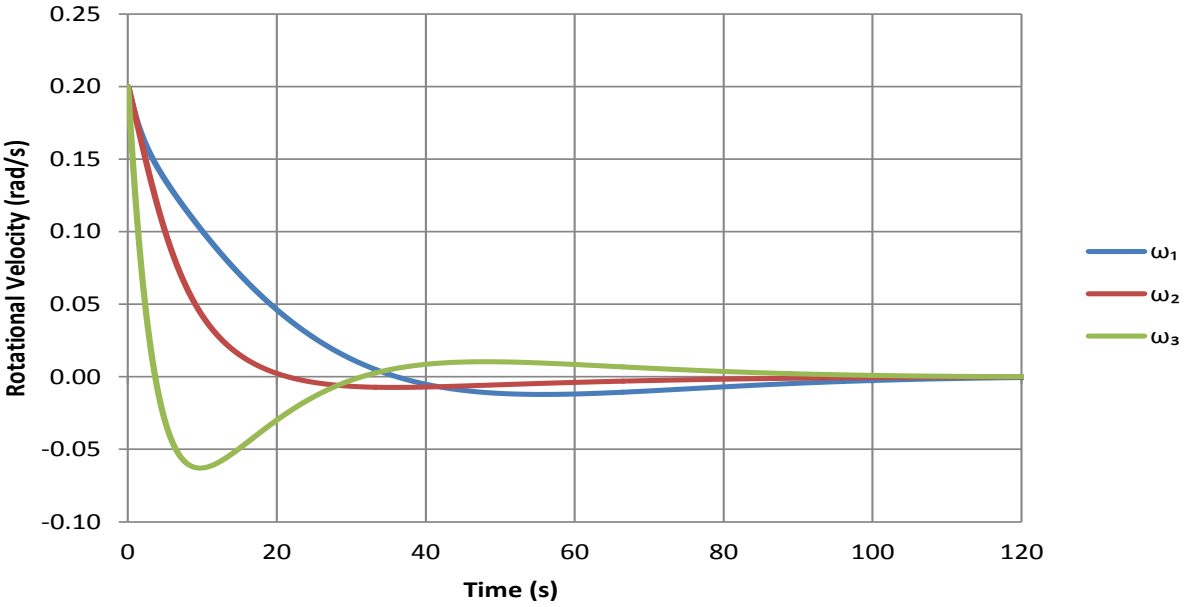

Figure 5.5: Demonstration simulation step 1 verification: Rotational Velocity vs. Time.

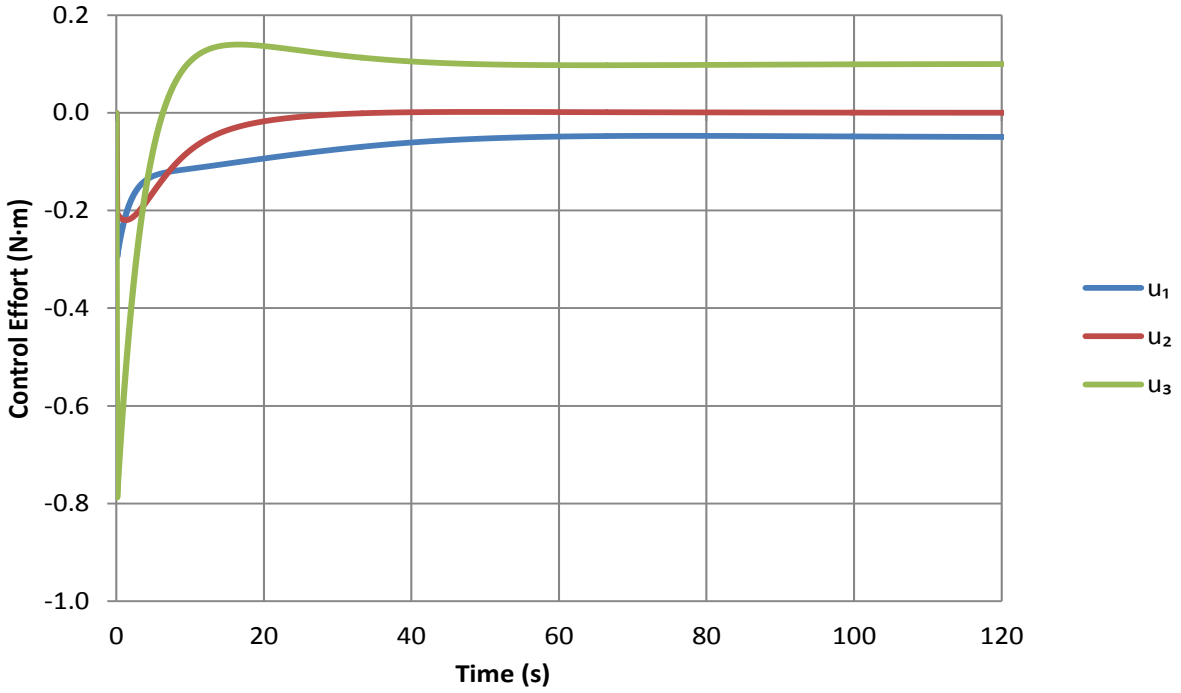

Figure 5.6: Demonstration simulation step 1 verification: Control Effort vs. Time.

5.2.2 Visual Tracking

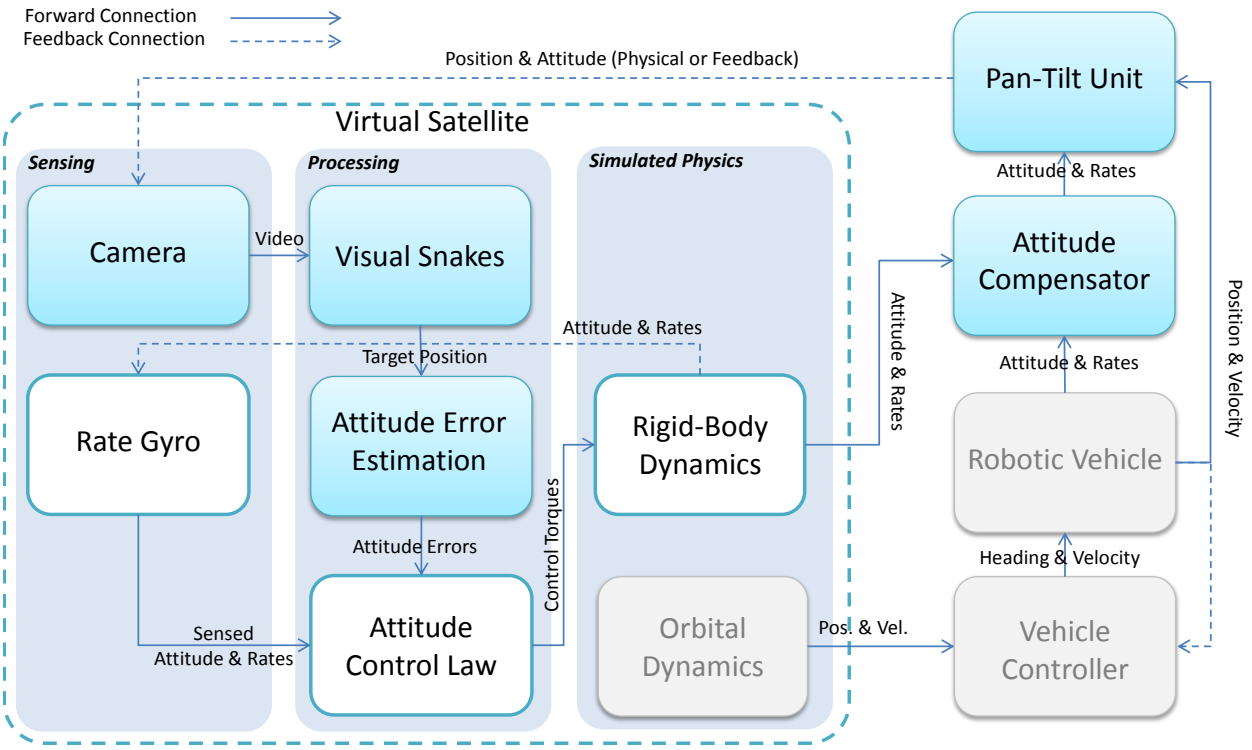

Figure 5.7: Demonstration Simulation Step 2: Introduction of visual tracking to drive control law

The second step of assembly builds upon the first by including the segments highlighted in figure 5.7. Taken together, these modules provide a visual tracking capability and expand from the simulation tier into the virtual tier. The camera components (sections 4.2.3 & 4.3.3) look at the area containing the target. The camera's position and orientation are taken from the results of the last time step's actuation. When using physical hardware, this is an implicit feedback connection that occurs simply because the hardware is bolted together. However, when using virtual hardware, this connection is explicit. However, because other simulation elements benefit from knowing the position of the camera, an explicit feedback connection is always created by the script wrappers for the camera bundle.

The visual sensing is accomplished with the visual snakes algorithm (section 4.5). The visual

snakes must at present be initialized by either a human operator or a known set of initial conditions which cause the target to be visible and located at a certain area within the frame. Ongoing research at Sandia Laboratories is improving the means by which the snakes can automatically acquire a target without human intervention.

The attitude error estimator uses the visual sensing's location of the target to determine a two degree-of-freedom relative attitude difference between the target and the current focal frame. Because the attitude error estimator assess no attitude difference about the focal axis of the camera (the 'roll' degree of freedom in an Euler 3-2-1 'yaw-pitch-roll' angle sequence), the craft does not waste control effort in attempting to align the roll axis. This is useful for applications which are insensitive to roll, such as low-powered directional antennas or some remote-sensing applications. This error signal creates a meaningful attitude time-history for the control law to implement, as opposed to an arbitrary null-rotation.

The physics model of the first step then produces an attitude which must be implemented upon the camera. Here, the pan-tilt unit (sections 4.2.2 & 4.3.2) and attitude compensator (section 4.1.3) come into play. The attitude isolator breaks down the physics simulation's attitude and rotational velocity, using a control law to turn this into a relative rotational velocity command for the pan-tilt unit. During this stage, no movement of the pan-tilt unit's base occurs, so the isolation effect of the compensator is not seen. The pan-tilt unit's bundle further decomposes the commanded rotational velocity into commanded pan and tilt rates, then issues these to either real or emulated hardware, moving the camera.

Verification of this setup is more qualitative than the first. Robustness and performance are determined by causing the visual snakes to lock on to a target, then moving the target around through various motions and visibility conditions. For the real camera, this is usually a sheet of paper with two concentric rectangles of highly-contrasting colors, while in the virtual environment the target of choice is a blue sphere against the white background. The target is moved through positions which cause the pan-tilt unit to follow to its extremities of motion to verify that no obvious outlier conditions exist. The target is partially occluded and moved about rapidly to ensure that

gains for the visual tracking algorithm have been well-tuned.

5.2.3 Robotic Translation & Orbital Motion

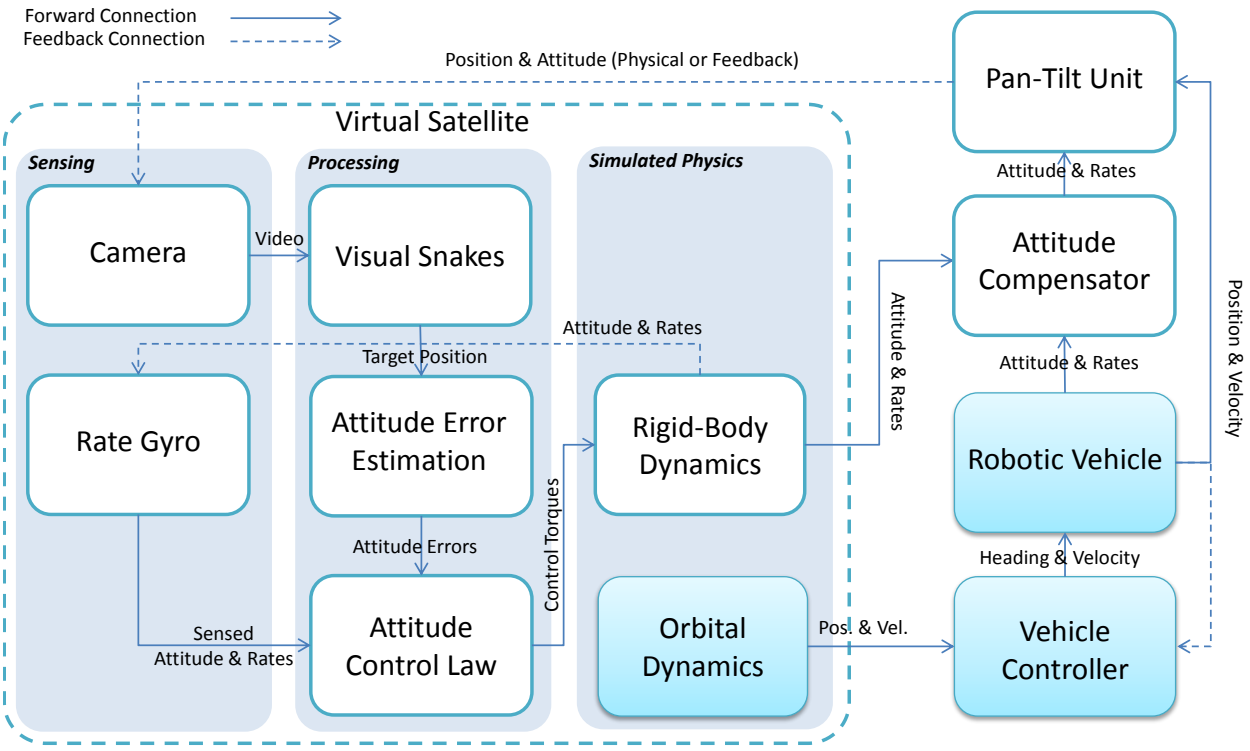

Figure 5.8: Demonstration Simulation Step 3: Addition of robotic translation platform and orbital simulation

The third and final step in the assembly of the simulation is the addition of the robotic vehicle and orbital motion, shown by the highlighted sections in figure 5.8. This is similar to the second step in that it also provides a virtual-layer implementation of physics, however this step deals with translational motion rather than rotational. The robot (sections 4.2.1 & 4.3.1) implements the simulation of translational motion for the vehicle, giving it the ability to follow a time-history of position and velocity in a plane. The vehicle controller module, created as part of the vehicle's bundle, translates the position and velocity into turn velocity and linear movement

velocity commands for the vehicle to follow, using a control law to close errors in position and velocity over time.

Orbit simulation is initialized by calculating a desired orbital motion from the analytical solution to the Clohessy-Wiltshire approximate equations of relative orbital motion. The equations of motion are

$$\begin{aligned} \ddot{x} - 2n\dot{y} - 3n^2 = 0 \\ \ddot{y} + 2n\dot{x} = 0 \\ \ddot{z} + n^2 z = 0 \end{aligned} \tag{5.1}$$

where x, y and z refer to position of the deputy spacecraft in the chief spacecraft's Hill frame, and n is the mean orbital motion. Their analytical solution is:

$$x(t) = A_0 \cos(nt + \alpha) \tag{5.2}$$

$$y(t) = -2A_0 \sin(nt + \alpha) + y_{off} \tag{5.3}$$

$$z(t) = B_0 \cos(nt + \beta) \tag{5.4}$$

Choosing the constants A_0, B_0, α, and β results in different closed orbits. Note that $y(t)$ also has an x_offset term in the full analytical solution, but this must be zero or the relative orbit will not be closed.

Although the Clohessy-Wiltshire equations are an approximation, the calculations of the orbital simulator module are exact, so no loss of accuracy is introduced by using them as a rough design tool for initial conditions.

At this point in the evolution of the simulation, additional frame structure must be created to accurately reflect the movement. The orbit simulator modules (section 4.4) can calculate orbital movement in a non-rotating ('static') frame centered at the chief satellite. A static frame is desirable as a base because calculation in non-rotating frames is computationally simpler, requiring no rotational differentiation. It is, however, far more convenient to have the deputy centered at the origin of the real & virtual frames at simulation start. More arbitrary placement of the deputy is functionally identical, making the setup a worthwhile exercise in controlling the relationship

between the vehicles and the virtual environment.

The orbital simulator library includes a module, `OrbitSim::LocalFrameConverter`, which can calculate the forward and inverse transformations of the Hill frame of one object moving about another. The Hill frame uses the unit radius vector from the central vehicle as the 1-axis ('x') and the unit angular momentum vector as the 3-axis ('z'), with the 2-axis ('y') completing a right-handed system. However, the Hill frame is a rotating frame. Given the knowledge that relative orbits are near-planar for diameters on the order of 100m, a useful construct is the Static Hill frame - that is, the Hill frame at one instant, and retained statically. Objects do move within the static hill frame, though they are co-incident with the origin at the instant of calculation. The Static Hill frame is non-rotating, and is useful for measurements regarding in-plane travel.

In order to position the Deputy's Static Hill frame (`Frames.DeputyHillStatic`), the local frame converter is used to calculate the inverse transformation to the chief's frame, allowing the chief frame to be a child frame of the deputy's static hill frame. Frame converters must then be used to translate between the output of the orbital simulator, which measures in the chief's non-rotating frame, and the deputy's static frame. The chief and deputy body frames are then child frames of their respective static frames, allowing them to move about in formation. As all of these frames are calculated against the deputy's static hill frame, the entire simulation may be then given a convenient placement.

Verification of the simulation's construction is done in two parts. A qualitative assessment is made before orbital motion is included that the vehicle has been properly connected and the pan-tilt unit rotationally isolated by driving it with direct velocity commands from its included UI panel. Verification of the orbital motion's connection is also easily accomplished by observing the behavior of the virtual hardware for longer run times, that it does indeed follow the designed relative orbit. Full verification of the completed simulation is discussed in detail in section 5.3.

5.3 Results

Quantity	Value
Chief Orbital Elements	
Semimajor Axis (a)	6800.0 km
Eccentricity (e)	0.0000
Inclination (i)	0.7854 rad
R.A.A.N. (Ω)	0.3491 rad
Argument of Periapsis (ω)	0.2618 rad
True Anomaly (f)	0.0000 rad
Chief Spacecraft	
Drag Coefficient (C_D)	2.6000
Cross-Section Area (A)	0.7854 m^2
Initial Position (${}^{\mathcal{C}}\boldsymbol{r}_{\mathcal{C}/\mathcal{C}}$)	0.00000, 0.00000, 0.00000 km
Initial Velocity (${}^{\mathcal{C}}\boldsymbol{v}_{\mathcal{C}/\mathcal{C}}$)	0.00000, 0.00000, 0.00000 m/s
Deputy Spacecraft	
Drag Coefficient (C_D)	2.0000
Cross-Section Area (A)	1.5000 m^2
Initial Position (${}^{\mathcal{C}}\boldsymbol{r}_{\mathcal{D}/\mathcal{C}}$)	-0.02000, 0.000000, -0.04000 km
Initial Velocity (${}^{\mathcal{C}}\boldsymbol{v}_{\mathcal{D}/\mathcal{C}}$)	0.00000, 0.045037, 0.00000 m/s
Initial Attitude ($\boldsymbol{\sigma}_{\mathcal{D}/\mathcal{C}}$)	0.00000, 0.000000, 1.00000
Initial Velocity (${}^{\mathcal{C}}\boldsymbol{\omega}_{\mathcal{D}/\mathcal{C}}$)	0.00000, 0.000000, 0.00000 rad/s

Table 5.2: Initial conditions for the relative orbit of the demonstration simulation

Testing of the simulation, for purposes of practicality, was done primarily in the virtual environment, as the simulation required considerably more floor space to cover a significant portion

of the relative orbital arc than was available.

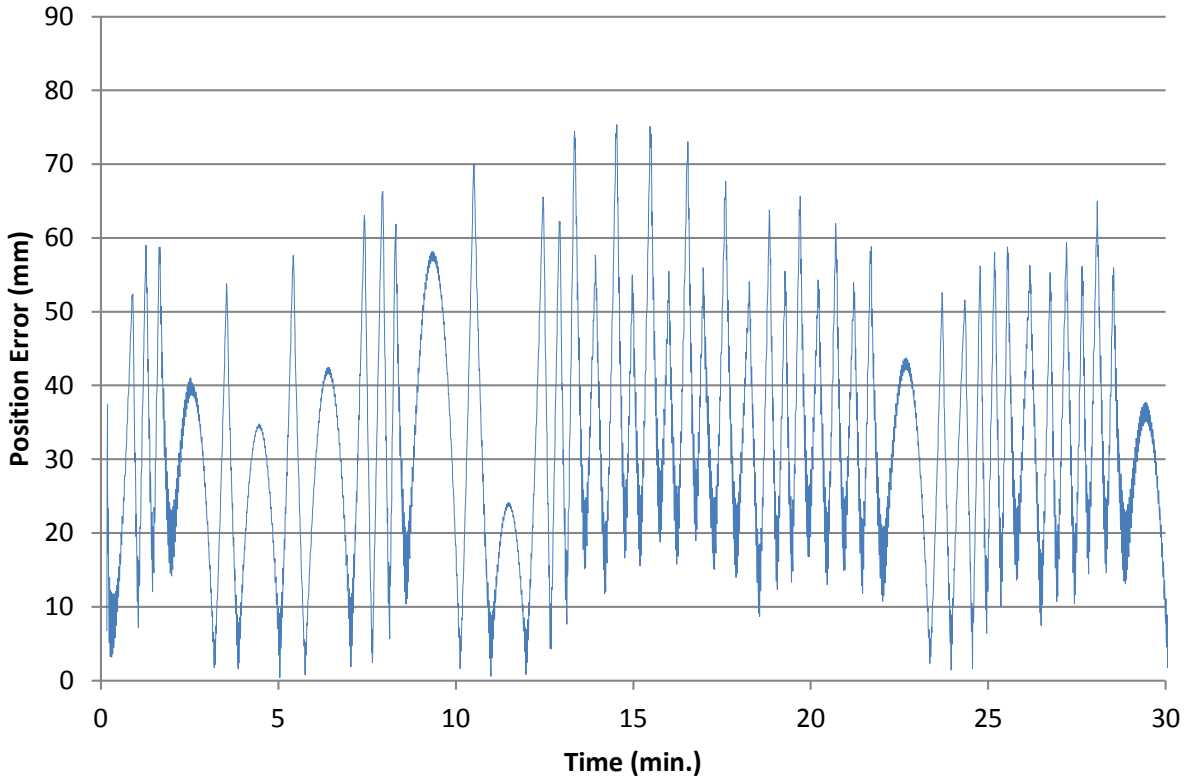

Figure 5.9: Absolute position error of the vehicle vs. time

Figure 5.9 shows the absolute position error over time of the robotic vehicle. That is, this is the error in mapping the simulated physics of the relative orbital motion to the virtual physics of the virtual laboratory. The behavior of the errors is reasonable consistent, and the total error is bounded by 75 millimeters. The pattern of the position/velocity control bringing the errors back down after drifting is readily apparent.

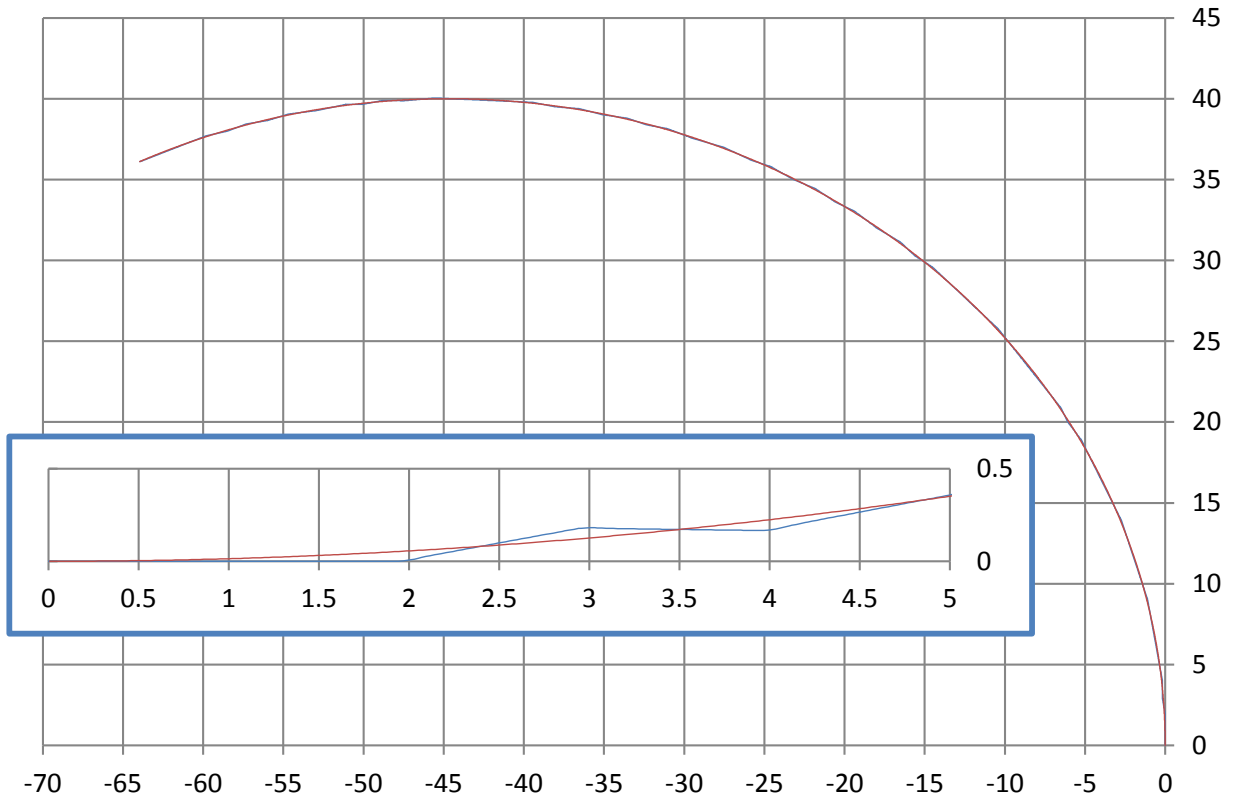

Figure 5.10: In-plane path of the robotic vehicle

Figure 5.10 shows the in-plane path of the vehicle over time. The vertical axis of the main graph corresponds to the y axis of the deputy static Hill frame, and the horizontal axis corresponds to the x. Though the two paths overlap in the fuller view of the orbit, the zoomed inset at bottom-right reveals a clear pattern of correction and re-correction by the control. Better performance could likely be achieved through fine-tuning of the control gains for the robot.

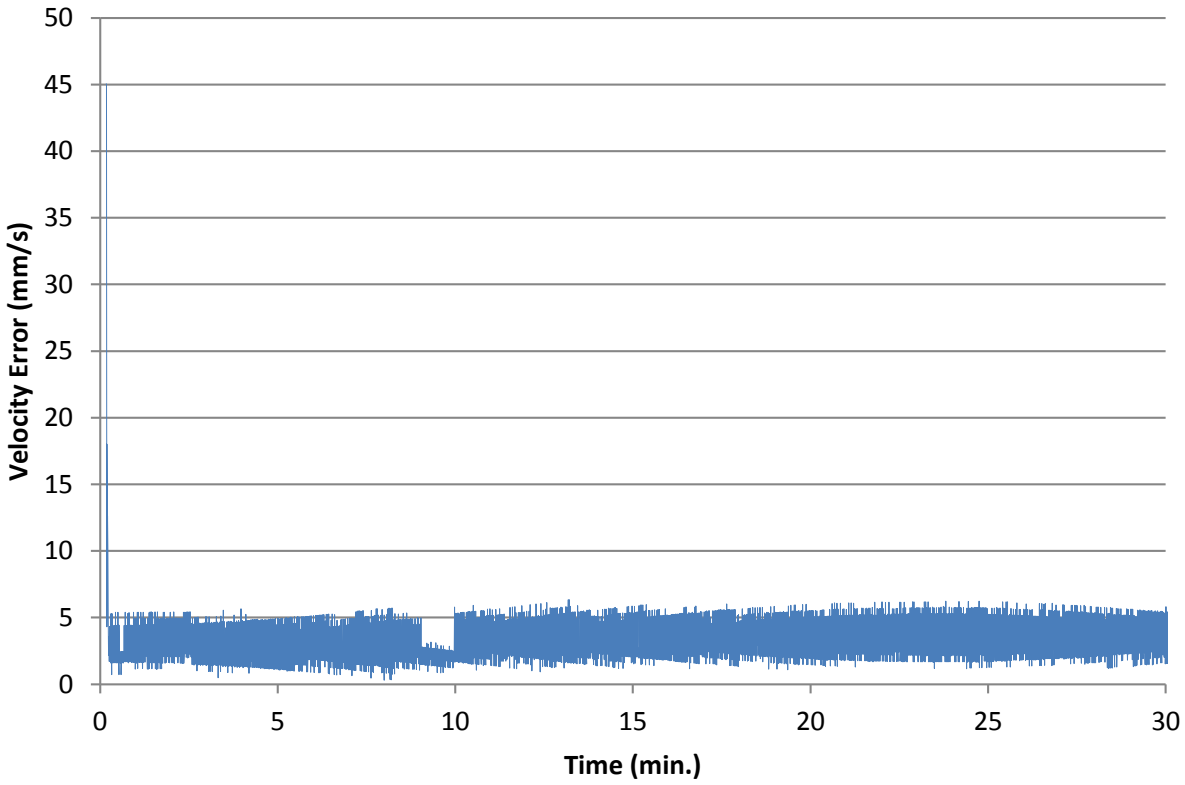

Figure 5.11: Velocity error of the robotic vehicle vs. time

Figure 5.11 shows the velocity error of the robotic vehicle over time. The velocity errors, although noisy, are reasonably low, bounded by 7 millimeters per second.

Figure 5.12 shows the time history of sensed attitude errors. These errors are the ones measured by the attitude error estimator module. The attitude error is broken down into pan and tilt directions due to the rather intuitive nature for small rotations. The tilt axis, as can be expected, has almost no error at all, with the few errors found being most likely due to small amounts of noise in the visual tracking algorithm. The pan axis error, however, displays two interesting features. The first is the spiking behavior. This is due to the small lag between the beginning of the vehicle's correctional turn and the counter-turn from the attitude compensator.

The second phenomena is the small steady-state error observed on the pan axis, which is most likely due to the continual apparent movement of the chief satellite as the deputy circles it.

The satellite's rotational control law neither models nor estimates the rotational velocity it needs to maintain the target in the center of the field of view, simply seeking on a desired zero rotational velocity. This mismatch between the actual velocity error and the desired velocity causes a small steady-state error as the velocity and attitude error terms compete. This is the type of insight that the simulation environment is designed to provide.

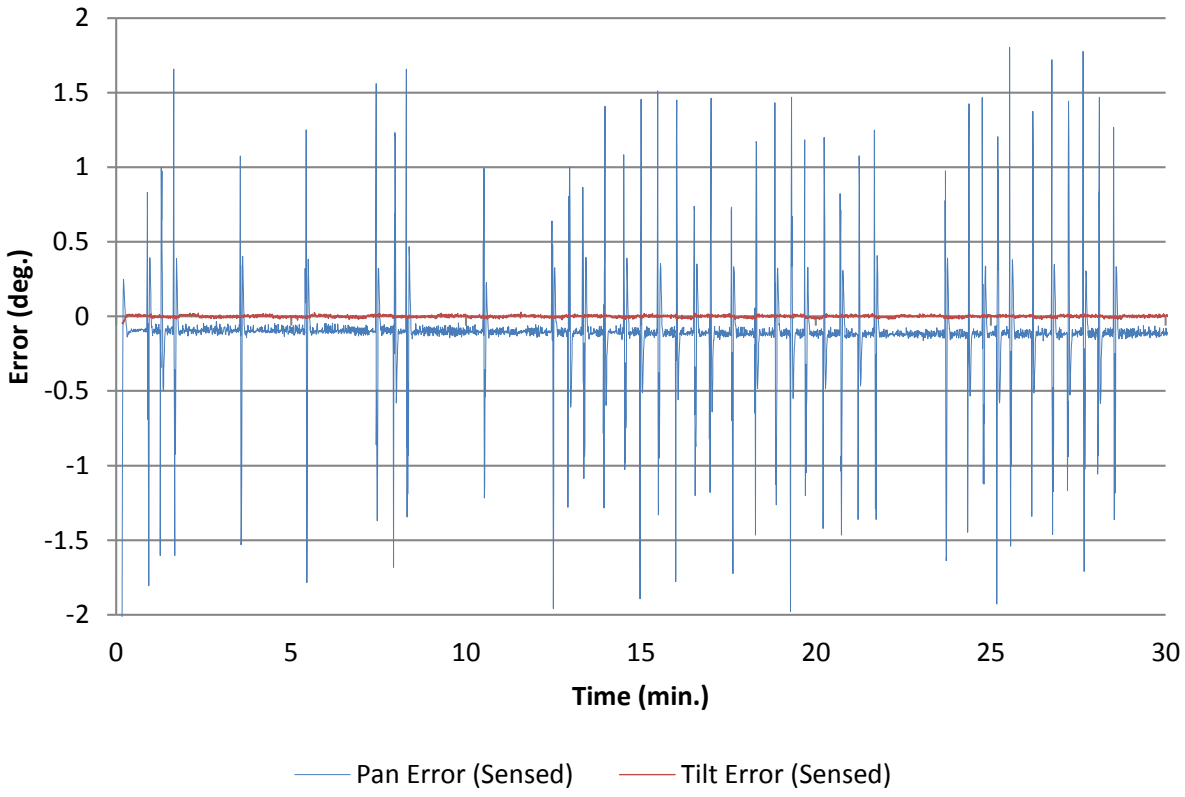

Figure 5.12: Error in attitude vs. time, as determined by the visual tracking algorithms

CHAPTER 6

CONCLUSIONS & FUTURE WORK

6.1 Future Work

6.1.1 Hardware Components

A number of potential hardware additions could be made to increase the capability of the laboratory's motion simulation, presented here in approximately increasing order of difficulty or complexity.

- A rotational servo could be mounted against the top block of the pan-tilt unit, providing a third degree of freedom for camera control. This would permit simulation of spacecraft roll with real hardware.
- A linear actuator mounted between the pan-tilt unit and the robotic vehicle could permit near-planar out-of-plane motion by moving the entire pan-tilt unit up and down.

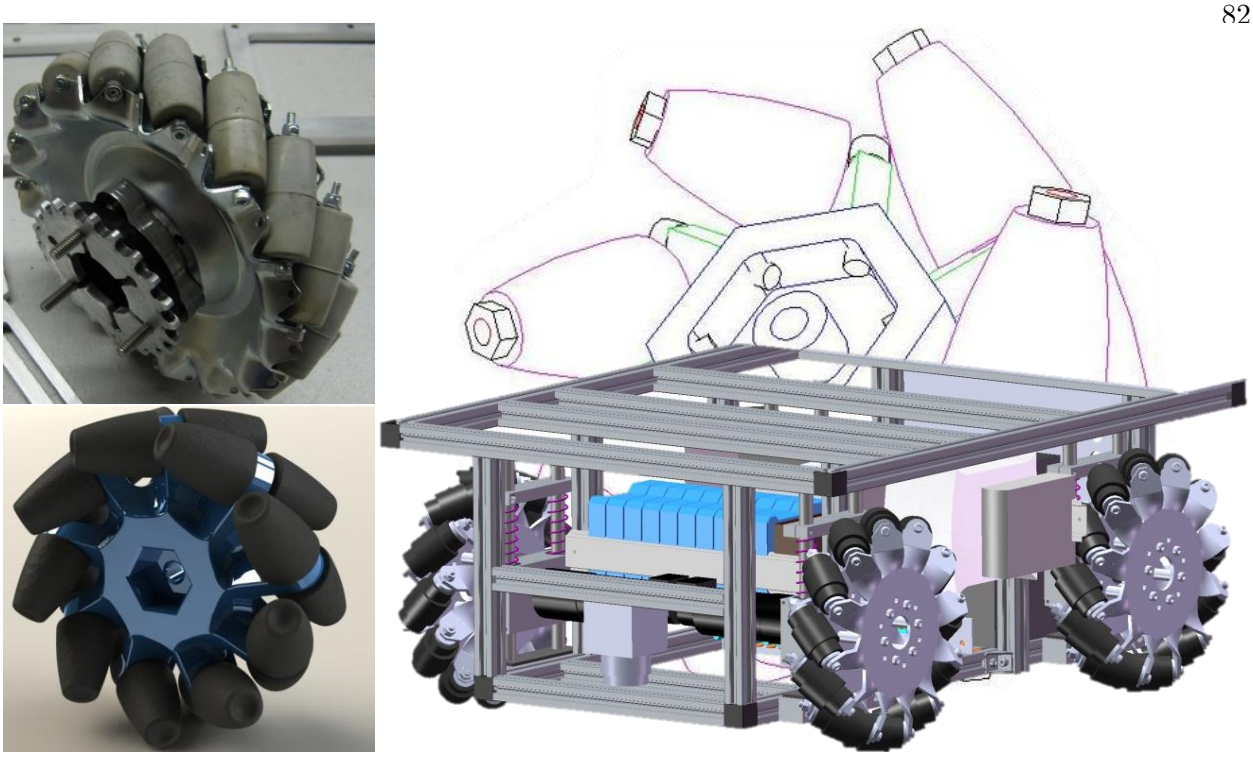

Figure 6.1: Examples of mecanum wheels and omni-directional vehicles

- An additional type of robotic platform could be constructed using mecanum wheels, which provide the capability for orientation-independent planar movement. This would carry an advantage over the current robotic vehicle when attempting to apply translational thrust to the simulated satellite at angles close to perpendicular to the direction of relative flight, as it would not require the vehicle to turn abruptly.

- With the advent of inexpensive consumer-grade quad-rotor aerial vehicles, such a vehicle could be added to the robotic vehicle collection, permitting exploration of large departures from the orbital plane, potentially over much larger scales than the laboratory floor.

6.1.2 Software Components

Owing to the extensible design of the framework, a large number of opportunities exist to expand capability through additional or improved software.

- The relative orbital motion simulation would be an ideal candidate for upgrading to a

full UMBRA interaction world, allowing for more generalized expansion of simulations to include more entities with fewer connections.

- Distributed simulation models could permit multiple labs to work in unison to simulate more complex events using local area networks or high-speed internet connections. This could potentially allow simulations to be conducted remotely by other researchers, increasing the utility of the laboratory.

- A more generalized image processing module could permit greater interaction between the real and virtual world, such as superimposing virtual objects, drawings, and targets onto the field of view of real cameras or pre-recorded video streams.

- Further interfaces could be created to make additional useful traits for simulation bundles. Examples could include `IPerspective` for objects which have a useful viewpoint that could be followed with a 3D view, or `IReplayable` for objects whose output could be recorded and then replayed to re-examine the simulation.

- UI & visualization improvements, such as native data graphing windows and video output windows, could add more rapid analysis capabilities for end-users and simulation designers.

6.1.3 Simulations

- Additional realism of satellite response can be injected into the demonstration simulation by creating a module which would emulate satellite hardware, including control saturation and minimum thrust dead-bands.

- Further realism can be obtained by adding sensor signal corruption, such as noise or calibration drift, and filtering techniques to counter it.

- Translational controls can be added to the simulation to test impacts of translational orbital maneuvers.

6.2 Conclusion

Robotic simulations are widely-recognized as an effective means of testing orbital hardware and software during development. The prohibitive costs associated with high-precision development have been shown to be mitigable with the increasing capacity and lowered cost of consumer-grade hardware, given the appropriate control laws and supporting software. A modular approach to the software, combined with immediate feedback and inspection capabilities, can further reduce development time for simulations.

The framework developed in this thesis represents an extensible platform which can scale to meet the growing needs of the laboratory and potential customers and partners. Software can be used to provide rapid testing of theory, while hardware can interchangeably verify the software results. The wide array of options for future work, features and improvement are deliberate facets of reusability and a forward-looking design. End-users of the framework are already adapting it to their needs and using the framework for verification of control algorithms. [15]

BIBLIOGRAPHY

[1] Jonathan H. Jiang Tristan S. L'Ecuyer. Touring the atmosphere aboard the a-train. Physics Today, July 2010.

[2] Jana L Schwartz and Christopher D Hall. Historical Review of Air-Bearing Spacecraft Simulators Introduction. AIAA Journal of Guidance, Control and Dynamics, 26(4):513–522, 2003.

[3] Linda L. Brewster, Connie Carrington, Richard T. Howard, Jennifer D. Mitchell, a. S. Johnston, and Scott P. Cryan. Multi-Sensor Testing for Automated Rendezvous and Docking Sensor Testing at the Flight Robotics Lab, March 2008.

[4] Daniel Sheinfeld and Stephen Rock. Optimal Despin of a Tumbling Satellite with an Arbitrary Thruster Configuration, Inertia Matrix, and Cost Functional. In Proceedings of the 20th AAS/AIAA Space Flight Mechanics Meeting, San Deigo, California, 2010. AAS,AIAA.

[5] Jacob Langelaan. State Estimation for Autonomous Flight in Cluttered Environments. PhD thesis, Stanford University, 2006.

[6] James Doebbler, Jeremy Davis, John Valasek, and J. Junkins. Mobile robotic system for ground-testing of multi-spacecraft proximity operations. In AIAA Modelling and Simulation Technologies Conference and Exhibit, number August, Honolulu, Hawaii, 2008.

[7] Richard T. Howard, Thomas C. Bryan, Linda L. Brewster, and James E. Lee. Proximity operations and docking sensor development. 2009 IEEE Aerospace conference, pages 1–10, March 2009.

[8] Christopher Smith Hanspeter Schaub. Color snakes for dynamic lighting conditions on mobile manipulation platforms. In IEEE/RJS International Conference on Intelligent Robots and Systems, Las Vegas, Nevada, October 2003. IEEE.

[9] Mark J. Monda. Hardware testbed for relative navigation of unmanned vehicles using visual servoing. Master's thesis, Virginia Polytechnic Institute, April 2006.

[10] Christopher C. Romanelli. Software simulation of an unmanned vehicle performing relative spacecraft orbits. Master's thesis, Virginia Polytechnic Institute, April 2006.

[11] Donald T. Shrewsbury. Providing a camera sensor with pointing capabilities independent of an unmanned ground vehicle. Master's project report, Virginia Polytechnic Institute, November 2006.

[12] P. Tsiotras. A passivity approach to attitude stabilization using nonredundant kinematic parameterizations. Proceedings of 1995 34th IEEE Conference on Decision and Control, pages 515–520, 1995.

[13] Ranjan Mukherjee and Degang Chen. Asymptotic Stability Theorem for Autonomous Systems. AIAA Journal of Guidance, Control and Dynamics, 16(5):961–963, 1993.

[14] Hanspeter Schaub and John L. Junkins. Analytical Mechanics of Space Systems. AIAA Education Series. American Institute of Aeronautics and Astronautics, Reston, VA, 2nd edition, October 2009.

[15] Samantha Krening. Visual spacecraft relative motion control using higher order geometric moments. Master's thesis, University of Colorado at Boulder, April 2011.

APPENDIX A

CODING PRACTICES

- Prefer verbose names to shorthand in scripts and code This allows for unambiguous identification of what is happening in the code by someone reading it who is not immediately familiar with the code and context. While this means more writing in a script, generally scripts need to be read with greater frequency than they are written. Shorthand should be reserved for the command line, where speed of typing is more important. Note that the alias script file is a good place to create global shorthand commands. For demonstrations, shorthand commands should ideally be located in a clearly-marked section toward the end of the script.

- Prefer PASCAL case to Camel case for public interfaces. PASCAL case consists of every word begining with a capital letter. Camel case consists of first word first letter lower-case, followed by subsequent words' first letter capitalized. Acronyms lose lower-case on second and subsequent letters.

Examples:	
Case Scheme	Sample Code
PASCAL Case:	`InputRotd HtmlDocument`
Camel Case:	`inputRotd htmlDocument`

Table A.1: Examples of PASCAL & Camel casings

PASCAL case is (arguably) easier to use when doing global search-replace strings in compounded words. Although this is a small consideration, the choice of case is largely arbitrary, and mainly should be kept as a guideline simply for consistency. Caution must always be exercised, regardless of casing scheme, to prevent inadvertent changes when using find/replace. Prepending of types is discouraged, with the exception of input/output connectors inside of C++ modules. Inside of non-public code, casing is at the author's discretion.

- Prefer consistent naming schemes Consistent naming schemes aid in recollection and prediction of members and api names, speeding development by requiring fewer interruptions to check documentation. Consistency in naming also helps suggest names for future additions to code.

- Prefer the following verbs In line with consistency, the following verbs are designated corresponding precise meanings, and should be preferred above synonyms. Note that many of the verbs exist in pairs which imply one another's existence.

Verb	**Meaning**
Create	Create something that does not currently exist.
Delete	Remove something that does currently exist.
Get	Obtain something that does currently exist.
Set	Change the value of something that does currently exist.
Start	Start a process or task which has not been previously running.
Stop	Stop a process or task with no intent to resume.
Resume	Start a process or task which had previously been running and paused.
Pause	Stop a process or task with intent to resume.

Table A.2: Preferred verbs and specific meanings

Table A.2 should be expanded as new verbs or verb pairs are added for one purpose or another.

- Prefer creative alignment whenever it would aid clairity One of the advantages of monospaced fonts typically employed in text editors is the ability to perform on-the-spot custom text alignments which can greatly aid readability of large blocks of slightly-differing code. Formatted properly, these draw the eye toward the differences in near-columnar format and de-emphasize the similarities.

- Prefer spaces to tab characters for alignment This rule is somewhat flexible to the needs of the author, but be advised that the framework's code is not guaranteed to gracefully handle tab characters when processing text.

- Prefer double-slash comments in C & C++ code to slash-star comments The C family of languages offers two types of comments, a single-line comment which begins with a backslash pair (`//`), and a multi-line comment which begins with a backslash & asterisk (`/*`) and ends with the pair reversed (`*/`). As might be expected upon reflection, attempting to nest two of the multi-line comments will cause early termination of the outer comment at a place which was not intended. As such, regular code comments should use single-line comments, one per line, reserving multi-line comments for deactivating large segments of code during debugging or alteration so that normal code comments do not interfere with workflow.

- Prefer multiple outputs of native datatypes over `std::vector<>` templates While standard C++ vectors are a useful way to handle sets of numbers, resist the temptation to use them to group sets of numbers which are not part of the same measurement. Seperating outputs makes the information more readily available for consumption by other modules without the gratuitous use of splitter and joiner modules.