Virginia Tech Department of Aerospace and Ocean Engineering
AOE 4994 Undergraduate Research
Final Report
Alternative Methods of Spacecraft Control Using Space Systems Simulation Laboratory and VT-CAVE

Michael Shoemaker

May 7, 2003

1 Introduction

The goal of this research is to design a preliminary system utilizing the Space Systems Simulation Laboratory (SSSL) and the VT-CAVE to explore alternative ways of controlling spacecraft simulators from within a virtual environment (VE). This research is conducted over the course of one semester with the intent of gaining background knowledge related to this area and building a framework for possible future research.

This goal is accomplished with the following tasks:

- A review of the literature regarding manual control of satellite systems, computer generated displays used in spacecraft control, and other research or applications using non-conventional display methods is conducted.
- The C++ code needed to visualize an orbital environment in the CAVE using OpenGL Performer is written.
- The DIVERSE application programming interface (API), developed previously[1] by the University Visualization and Animation Group of Virginia Tech, is used to interface the CAVE simulation with the existing software for the spacecraft simulator.
- Preliminary experiments are conducted to test the effectiveness of the system.
- Documentation of experiments, including results, ways to improve the system, and recommendations for further research are given.

This report is divided into the following sections. First is a section highlighting the literature review conducted thus far. Next is a section describing the CAVE graphical simulations developed, and issues associated with them. This is followed by a section describing the two main methods of animating the CAVE simulation using attitude data, and the experimental results. Lastly is a discussion of the achievements and limitations of the current system, and ideas for further research.

2 Literature Review

The initial topics considered in the literature review deal primarily with manual spacecraft control and associated display methods; however, while conducting the review, other applicable areas become apparent. Examples of immersive or interactive displays are found in fields besides spacecraft control, such as undersea vehicles and remotely operated planetary rovers. Also, such displays are used not only for direct control, but mission planning, observation, and supervisory control. These additional topics are included to illustrate the potential for any future VE control system using the CAVE and the SSSL spacecraft simulators.

Typical manual control tasks of interest often involve maneuvering a spacecraft

in close relation to another object in orbit, such as rendezvous, docking, rescue, and repair[2]. These kinds of tasks, called proximity operations (PROX-OPS) when occurring within a 1-km radius of the space station, can be difficult for a human operator to perform [3]. Trajectories viewed between two moving bodies, combined with the effects of orbital mechanics, result in counterintuitive motion with a lack of stable reference points. Numerous burns are required to complete a single task: departing, maneuvering, and braking. The operator must also be aware of various safety constraints: clearance from structures, allowable approach velocities, fuel consumption, and thruster plume impingement on sensitive equipment.

Because of their complexity, PROX-OPS are usually planned well in advance. However, unanticipated situations will undoubtedly arise, and often they require quick decision making or control schemes that cannot be implemented by computer automation. This is one argument in favor of direct manual control. A hypothetical example would be a crew member or piece of equipment becoming separated from the space station, and a recovery craft responding quickly on an improvised trajectory. A real-life example shows another necessity of manual control; during the Gemini program, astronaut John Glenn was able to save himself and his spacecraft using manual override when the automated system failed[4].

It should be noted that there are cases when automated control is preferred, and direct human control could even be considered a hinderance[5]. Such examples include precise payload pointing for remote sensing or communication satellites, repetitive tasks like attitude correction or station keeping, and unmanned missions with prohibitively large time lags in communication.

Of interest in this research is the graphical interface used by the operator in the cases of manual control, and how it can add or detract from the mission. The difficulties in the PROX-OPS scenarios described above "may be substantially overcome by visualizing the orbital trajectories and constraints in a pictorial, perspective display."[4] Even though most of these systems described in the literature deal with orbital motion, the lessons learned can be applied to attitude dynamics and control, and space mission analysis as a whole.

One such system[6] was an interactive PROX-OPS planning tool using a traditional 2-D monitor. Rather than specifying thrust direction and magnitude, the operator selected trajectory waypoints in a graphical representation of the spacecrafts in orbit (Fig. 1). An "inverse dynamics" algorithm then calculated the burns required to reach those waypoints. Experimental results showed that with minimal training, "non-astronaut aerospace professionals" were able to design complex maneuvers subject to constraints to recover simulated objects released from the space station in under 2 min. Even though this was an iterative process that could conceivably be accomplished through computer automation, at the very least this experiment provides a benchmark to evaluate such an automatic system.

In the system described above, increased performance was achieved through 3-D graphics displayed on a standard 2-D monitor. While this is sufficient in some circumstances, others missions require an interface capable of conveying more information. For the constraints listed above in the PROX-OPS missions, such as clearance from structures and the directions of approach velocities and thruster plumes, this infor-

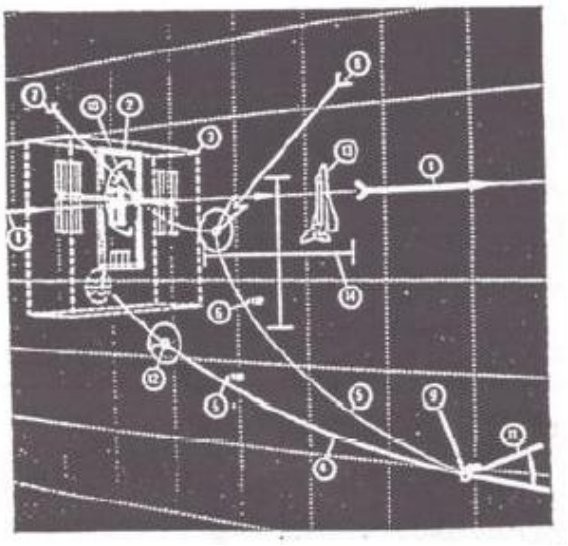

Figure 1: Reproduction of screenshot of interactive trajectory planning tool[6], showing planning of PROX-OPS recovery mission.

mation would take the form of an increased awareness of the geometry of the area in question. Immersive 3-D graphical interfaces are often used to achieve this so called "telepresence", described as "a form of electronically mediated presence providing high-fidelity remote control by projecting natural human capabilities to distant work sites".[7]

One example in the literature of such a system was the Telepresence-controlled Remotely Operated underwater Vehicle (TROV)[8]. This system consisted of an undersea vehicle piloted remotely to explore the marine environment beneath Antarctic ice. The motivation was to test future space systems and their control interfaces in environments and situations analogous to those which they might encounter during operation. One method of remotely controlling the TROV was accomplished with the Virtual Reality Vehicle Interface (VEVI), in which an operator located at NASA Ames Research Center in California controlled the actual TROV from within the graphical simulation (Fig. 2). The undersea terrain models in VEVI were generated using real sensor data from the TROV, and the operator viewed the simulation using either a Head-Mounted Display (HMD) or a stereo monitor. The results gathered from this system showed that the immersive VE allowed the operator to achieve a level of spatial orientation and understanding despite little or no foreknowledge of the operating environment.

Another example of an unconventional display system used to control space-related hardware is the Mars Pathfinder mission[9]. The martian lander used a high-resolution narrow field of view stereo-camera called the Imager for Mars Pathfinder to send images of the landing site back to Earth. Then a 3-D terrain map was generated by comparing the stereo images, and the data feed into a computer generated graphical environment3. The controllers could then navigate in this VE and specify

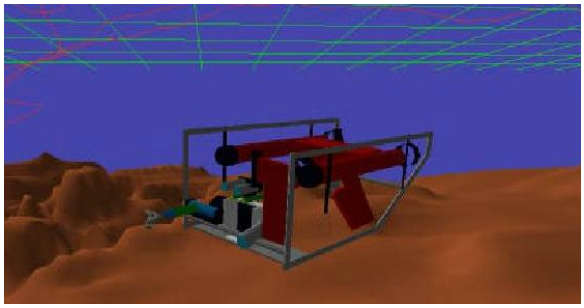

Figure 2: Screenshot of VEVI showing computer generated models of TROV and undersea environment.

science targets or movement commands to the rover. Like the TROV, this system allowed the controllers increased situational awareness despite the extreme distances and unknown environment. Another noteworthy result was that even though the rover was not directly controlled, the interface allowed high-level commands to be issued to the rover with confidence while the onboard autonomous control system managed the low-level control tasks.

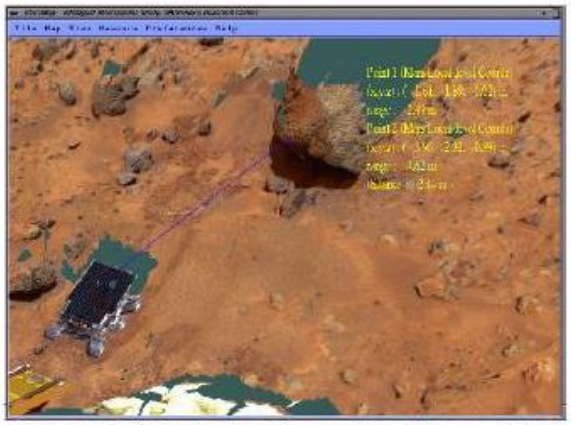

Figure 3: Screenshot of the VE used by Mars Pathfinder mission.

As a final example, researchers at Caltech developed an interactive trajectory planning tool to design orbits related to the Terrestrial Planet Finder mission[10]. This astronomy mission called for transfer orbits from Earth to the L2 Lagrange point, where a formation of 5 satellites would enter a halo orbit and form an infrared interferometer. In order to better visualize and design these complex transfer orbits and satellite formations, a semi-immersive VE was created for use on a Responsive Workbench (Fig. 4). The designer would view a 3-D representation of a region of low-energy transfer orbits, and select different orbits using a tracked input device.

Using this system allowed better understanding of the complex geometry associated with both the transfer orbit and the formation flying. Additionally, this tool was useful for conveying these unfamiliar orbits to others in industry.

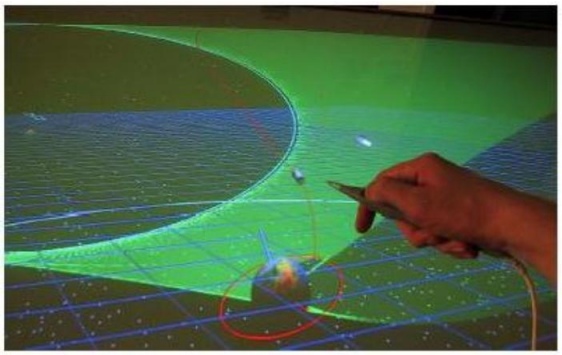

Figure 4: Screenshot of the VE used by Terrestrial Planet Finder mission.

While these examples represent only some of the various applications of VEs related to spacecraft control, they highlight some important features. The intent of the system currently under consideration is the direct manual control of the spacecraft simulators from within the CAVE; however, in the future the system could certainly find other uses related to research in the SSSL. For example, the VE could be used to evaluate an autonomous satellite control scheme, or visualize a formation of spacecraft simulators. Regardless of the specific application, "the most successful visual interface will be the one that maximizes the operator's ability to apply his or her superior perceptual and judgement abilities to the control problem at hand."[11]. Thus, the CAVE is merely one option, which under given circumstances, could be a tool which allows the user to apply these judgemental abilities.

3 CAVE Graphical Simulation

This section contains a brief description of the programming interfaces used to visualize the graphical simulation in the CAVE, followed by descriptions of the two separate VEs created.

OpenGL Performer from SGI is a software API that is built on the OpenGL graphics library. The code used in the current system was written using C++, but C versions of Performer also exist. Since Performer is considered a high-level programming language, the programmer is relatively shielded from the low-level OpenGL interaction with graphics hardware[12]. This makes development of high performance graphical simulations possible without a steep learning curve.

Another benefit of Performer is that an open source evaluation version is available to run on a PC using GNU/Linux[1], provided the PC has powerful enough graphics hardware. The graphical simulation seen here was developed on the author's PC running Red Hat Linux 7.3 and Performer 2.5, equipped with an NVidia GeForce4 Ti 4200 graphics card.

The general procedure to load Performer graphics into the CAVE is to first break up the desired scene into logical components, compile each separately as a different graphics file, then combine the components into the final Performer CAVE simulation. For example, the orbital scene is divided into the earth, reference frames, planes, etc. The geometry and appearance of each is written in Performer's C++ API, then the scene graph is compiled and exported as a *.pfb* file, which is the native database format used by Performer[13]. Each of these *.pfb* files is then attached to a coordinate system in the main DPF program to be run in the CAVE.

3.1 Orbital Environment

The initial scene created for the CAVE consists of a preliminary version of an orbital environment (Fig. 5). The main features include a textureless Earth, the earth-centered inertial and perifocal reference frames, the orbital and equatorial planes, and a star field. The star field came as a pre-made file included with Performer. The properties of the orbit are determined by the orbital elements: semi-major axis (a), eccentricity (e), inclination (i), right-ascension of the ascending node (Ω), and argument or periapsis (ω). Currently this simulation is entirely static; if the orbital elements need to be changed, the program must be re-compiled.

This scene served two primary purposes: to test writing Performer simulations and loading them in the CAVE, and to evaluate how an orbital environment would look in an immersive VE.

Even though this is a preliminary version of an orbital environment represented in

[1] As of Performer 3.0, a Windows version is also available. However, DIVERSE currently only runs on GNU/Linux or IRIS. For more information on DIVERSE, DIVERSE Graphics Interface to Performer (DPF), or DIVERSE ToolKit (DTK), please see www.diverse.sourceforge.net

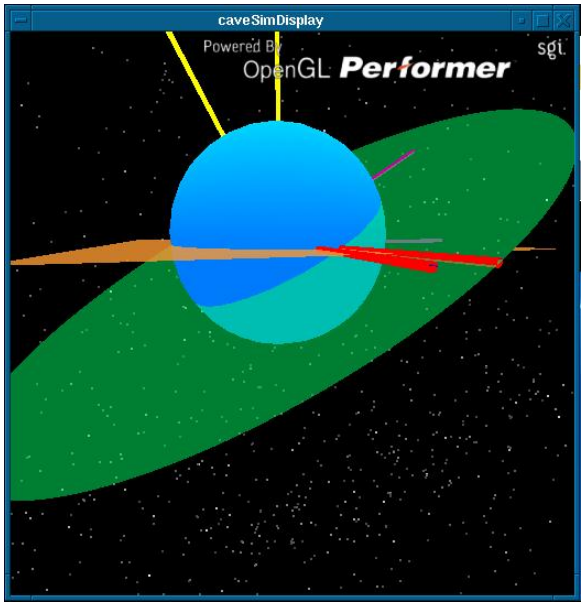

Figure 5: Screenshot of orbital environment, shown in DPF desktop CAVE simulator.

a VE, some traits become apparent. For users who have little previous experience with astrodynamics or orbital elements, the simulation provides a readily understandable representation of the orbit in 3-D space.

3.2 Spacecraft Simulator

The next scene created for the CAVE is a graphical representation of the Whorl1 spacecraft simulator located in the SSSL (Fig. 6). This model consists of some simplified components of the simulator mounted on a hexagonal test-bed, as well as body-fixed and inertial reference frames. This early model does not contain much detail, but serves the purpose of the reminding the user that what he or she is viewing represents the actual test-bed and simulator hardware, rather than a general model of a spacecraft.

Unlike the orbital scene, this graphical simulation is dynamic; the model is attached to a coordinate system which can change its position and orientation with respect to the origin of the CAVE, or the so called "world" coordinates. This allows the model of Whorl1 to be animated by external data. The current means of rotating models in Performer is with an heading-pitch-roll (HPR) combination of rotations. If the rotations are taken about a reference frame which remains aligned with the world coordinate system, these HPR rotations correspond to a 3-1-2 Euler angle rotation sequence[14], the development of which follows in the next section.

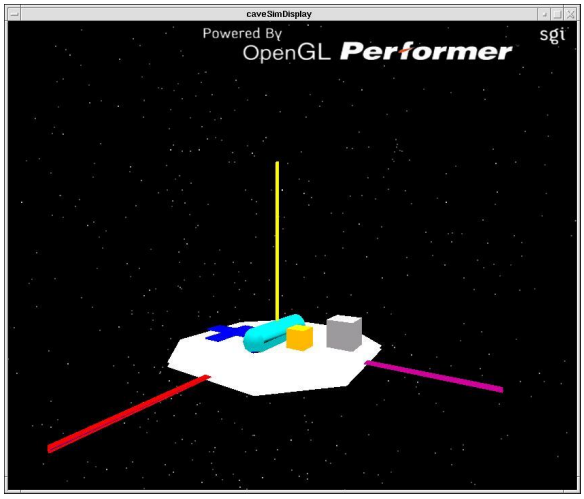

Figure 6: Screenshot of Whorl1 graphical model, shown in DPF desktop CAVE simulator.

4 Animation using Attitude Data

A number of different methods were tested to animate the model of Whorl1, with the two most significant presented here. What follows is a development of the attitude kinematics used to animate the CG model of Whorl1 in the first method. Then a description of the two sources of attitude data used to animate the model will be given. The first involved integrating the kinematic differential equations of motion (EOM) to generate "fake" attitude data off-line, then animating the model using this pre-generated data. The second involved reading the Euler angle rates sensed by the rate gyro located on Whorl1, and sending this attitude data to the application running in the CAVE.

4.1 Attitude Kinematics

For the development of the kinematic differential EOM, the spacecraft simulator is approximated as a simple rigid body, and the attitude is represented by a rotating reference frame. The two frames of interest are the body frame, $\mathcal{F}_b$, which remains fixed to the center of mass of the satellite, and the inertial frame, $\mathcal{F}_i$, which remains fixed in the lab. The inertial frame can be thought of as being fixed on the mounting stand which supports the Whorl1 simulator. It should be noted that this is not a true inertial frame, but this approximation is made for these experiments.

For the reasons stated above concerning the HPR rotations in Performer, a 3-1-2 Euler angle sequence was chosen to rotate $\mathcal{F}_b$ about $\mathcal{F}_i$. The individual rotations are defined as:

$$\mathbf{R}_3(\theta_1) = \begin{bmatrix} c_1 & s_1 & 0 \\ -s_1 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix}; \mathbf{R}_1(\theta_2) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_2 & s_2 \\ 0 & -s_2 & c_2 \end{bmatrix}; \mathbf{R}_2(\theta_3) = \begin{bmatrix} c_3 & 0 & -s_3 \\ 0 & 1 & 0 \\ s_3 & 0 & c_3 \end{bmatrix} \tag{1}$$

where $c_n = \cos(\theta_n)$ and $s_n = \sin(\theta_n)$ for $n = 1, 2, 3$. The subscript on the rotation represents the axis about which it is taken, *i.e.* $\mathbf{R}_3$ is a rotation about the $\hat{\mathbf{i}}_\mathbf{3}$ axis. Likewise, the subscript on the rotation angle represents the order in the sequence, *i.e.* $\mathbf{R}_3(\theta_1)$ signifies the first rotation is a "3"-rotation of the amount θ_1.

The 3-1-2 Euler angle sequence is then determined by multiplying the matrices in Eq. (1):

$$\mathbf{R}_2(\theta_3)\mathbf{R}_1(\theta_2)\mathbf{R}_3(\theta_1) = \begin{bmatrix} c_1c_3 - s_1s_2s_3 & s_1c_3 + c_1s_2s_3 & -s_3c_2 \\ -s_1c_2 & c_1c_2 & s_2 \\ c_1s_3 + s_1s_2c_3 & s_1s_3 - c_1c_3s_2 & c_2c_3 \end{bmatrix} \tag{2}$$

The angular velocity of $\mathcal{F}_b$ with respect to $\mathcal{F}_i$ is expressed in vector notation as:

$$\vec{\omega}^{bi} = \vec{\omega}^{bi''} + \vec{\omega}^{i''i'} + \vec{\omega}^{i'i} \tag{3}$$

where $\vec{\omega}^{i'i}$ is the angular velocity of $\mathcal{F}_{i'}$ about $\mathcal{F}_i$, $\vec{\omega}^{i''i'}$ is the angular velocity of $\mathcal{F}_{i''}$ about $\mathcal{F}_{i'}$, and $\vec{\omega}^{bi''}$ is the angular velocity of $\mathcal{F}_b$ about $\mathcal{F}_{i''}$. These intermediate reference frames are a consequence of using the 3-1-2 Euler angle sequence, *i.e.* $\mathcal{F}_{i'}$ is the intermediate frame that results from $\mathbf{R}_3(\theta_1)$. The vector notion in Eq. (3) is replaced with matrix notion, where the subscript represents the reference frame in which the individual angular velocity vector is expressed:

$$\omega_i^{i'i} = \omega_{i'}^{i'i} = \begin{bmatrix} 0 & 0 & \dot{\theta}_1 \end{bmatrix}^{\mathrm{T}} \tag{4}$$

$$\omega_{i'}^{i''i'} = \omega_{i''}^{i''i'} = \begin{bmatrix} \dot{\theta}_2 & 0 & 0 \end{bmatrix}^{\mathrm{T}} \tag{5}$$

$$\omega_{i''}^{bi''} = \omega_b^{bi''} = \begin{bmatrix} 0 & \dot{\theta}_3 & 0 \end{bmatrix}^{\mathrm{T}} \tag{6}$$

It should be noted that the angular velocity between two reference frames can be expressed in either frame, since the axis of rotation remains the same before and after the rotation is performed.

The angular velocities must be expressed in the same reference frame in order to be added together in Eq. (3); this is done by rotating them all into the $\mathcal{F}_b$ frame using the necessary rotations in the 3-1-2 Euler angle sequence. Eq. (6) is already expressed in $\mathcal{F}_b$. Eqs. (4 & 5) are expressed in $\mathcal{F}_b$ as follows:

$$\omega_b^{i'i} = \mathbf{R}_2(\theta_3)\mathbf{R}_1(\theta_2)\omega_{i'}^{i'i} = \begin{bmatrix} c_3 & 0 & -s_3 \\ 0 & 1 & 0 \\ s_3 & 0 & c_3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_2 & s_2 \\ 0 & -s_2 & c_2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \dot{\theta}_1 \end{bmatrix} = \begin{bmatrix} -s_2c_3\dot{\theta}_1 \\ s_2\dot{\theta}_1 \\ c_2c_3\dot{\theta}_1 \end{bmatrix} \tag{7}$$

$$\omega_b^{i''i'} = \mathbf{R}_2(\theta_3)\omega_{i''}^{i''i'} = \begin{bmatrix} c_3 & 0 & -s_3 \\ 0 & 1 & 0 \\ s_3 & 0 & c_3 \end{bmatrix} \begin{bmatrix} \dot{\theta}_2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} c_3\dot{\theta}_2 \\ 0 \\ s_3\dot{\theta}_2 \end{bmatrix} \tag{8}$$

Thus, Eq. (3) is written as:

$$\omega_b^{bi} = \begin{bmatrix} 0 \\ \dot{\theta}_3 \\ 0 \end{bmatrix} + \begin{bmatrix} c_3\dot{\theta}_2 \\ 0 \\ s_3\dot{\theta}_2 \end{bmatrix} + \begin{bmatrix} -s_2c_3\dot{\theta}_1 \\ s_2\dot{\theta}_1 \\ c_2c_3\dot{\theta}_1 \end{bmatrix} = \begin{bmatrix} c_3\dot{\theta}_2 - s_3c_2\dot{\theta}_1 \\ \dot{\theta}_3 + s_2\dot{\theta}_1 \\ s_3\dot{\theta}_2 + c_2c_3\dot{\theta}_1 \end{bmatrix} \tag{9}$$

Using Eq. (9) to relate angular velocity to the rate of change of Euler angles yields:

$$\omega_b^{bi} = \begin{bmatrix} -c_2s_3 & c_3 & 0 \\ s_2 & 0 & 1 \\ c_2c_3 & s_3 & 0 \end{bmatrix} \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{bmatrix} \tag{10}$$

Eq. (10) is customarily written as:

$$\omega_b^{bi} = \mathbf{S}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}} \tag{11}$$

which can then be rearranged by taking the inverse of the $\mathbf{S}(\boldsymbol{\theta})$ matrix:

$$\dot{\boldsymbol{\theta}} = \mathbf{S}^{-1}\omega_b^{bi} = \begin{bmatrix} -s_3/c_2 & 0 & c_3/c_2 \\ c_3 & 0 & s_3 \\ s_2s_3/c_2 & 1 & -c_3s_3/c_2 \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} \tag{12}$$

Thus, if the angular velocity in the body frame is known as a function of time, the individual Euler angle rates can be calculated.

4.2 Animation Method 1: Pre-generated Attitude Data

The first method used to animate the model of Whorl1 involved numerical integration of the kinematics differential EOM. A schematic of the system can be seen in Fig. (7).

First, Eq. (12) was numerically integrated in MATLAB using the `ode45` function. This built-in function is part of MATLAB's suite of ordinary differential equation solvers, and uses an explicit one-step Runge Kutta integrator of 4th or 5th order [15]. The right-hand side of Eq. (12) was included as part of a function *M-file*, called "RHS.m" in Fig. (7), with an input argument consisting of the state vector at the current time step:

$$\mathbf{X} = \left[\boldsymbol{\theta}^{\mathrm{T}}, \;\; \boldsymbol{\omega}^{\mathrm{T}}\right]^{\mathrm{T}} \tag{13}$$

and a return value consisting of the rate of change of the state vector:

$$\dot{\mathbf{X}} = \left[\dot{\boldsymbol{\theta}}^{\mathrm{T}}, \;\; \dot{\boldsymbol{\omega}}^{\mathrm{T}}\right]^{\mathrm{T}} \tag{14}$$

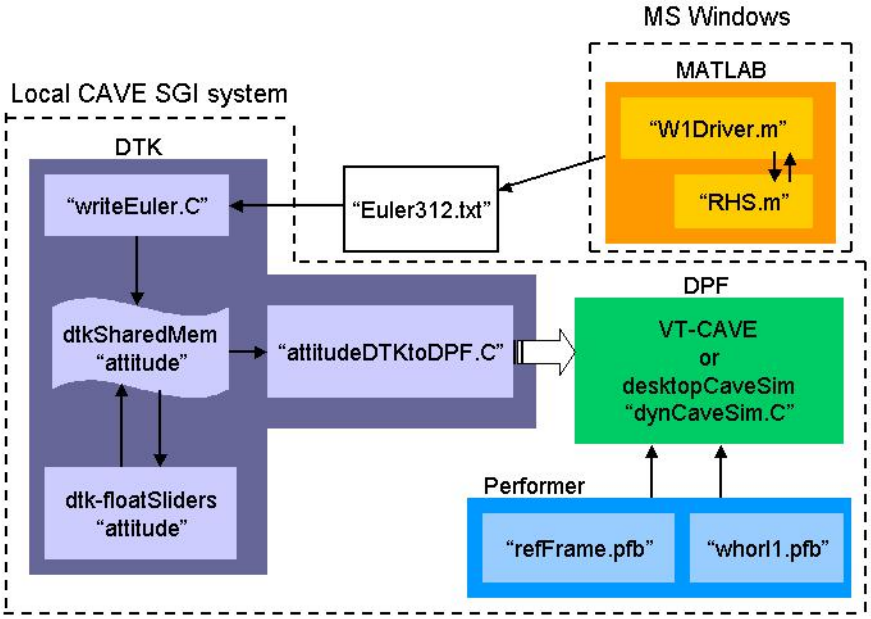

Figure 7: Diagram showing layout of the first system to animate the model using pre-generated attitude data.

The $\dot{\theta}$ vector is specified by right-hand side of Eq. (12), and the $\dot{\omega}$ vector was chosen to replicate the motion of the Whorl1 simulator, *i.e.* steadily increasing rotation about the $\hat{\mathbf{b}}_3$ axis, and oscillatory rotation of small magnitude about the $\hat{\mathbf{b}}_1$ and $\hat{\mathbf{b}}_2$ axes:

$$\dot{\boldsymbol{\omega}}_b^{bi} = \begin{bmatrix} 0.025\sin(t) \\ -0.025\sin(t) \\ \mathrm{e}^{-t} \end{bmatrix} \tag{15}$$

The "W1Driver.m" MATLAB program then calls the "RHS.m" *M-file* through the `ode45` function call, at a time step of 0.05 sec for a time interval of 20 sec. The resulting 3-1-2 Euler angles are output to both a plain text file and a MATLAB plot, seen in Fig. (8). The text file "Euler312.txt" is saved and transferred to the SGI system running the CAVE.

The next part of the system consists of a program to read from the text file containing the pre-generated Euler angles and write to a DTK shared memory segment. This program, called "writeEuler.C" in Fig. (7), reads the rotation angles from the text file at a specified rate using the methods included in the `dtkTime` class, and writes them to the DTK shared memory segment "attitude". In this case, the rate is the same as the time step used for the numerical integration; the Euler angles are written to shared memory once every 0.05 sec.

The "attitudeDTKtoDPF.C" program is based on the "joystickDTKtoDPF.C" sample program included with the 2.1.0 version of DPF. The "attitudeDTKtoDPF.C"

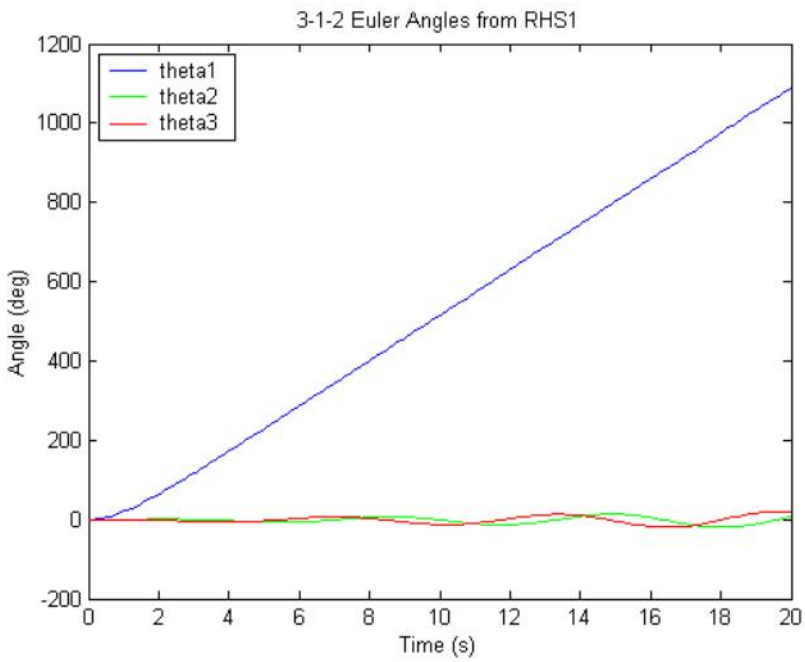

Figure 8: MATLAB plot showing the 3-1-2 Euler angles generated by numerically integrating the kinematic differential EOM.

program contains a modified version of the `dtkInValuator` class, which uses inherited methods from the `dtkAugment` class. Essentially, the `dtkAugment` class contains methods to augment the functionality of DPF using pre-defined callback functions (ref DPF manual). An example of a typical dtkInValuator object would be a joystick or pointer device. In this case, "attitudeDTKtoDPF.C" associates the three `float` values in "attitude" with three variables in the CAVE application to control the HPR rotations of the Whorl1 model. The "attitudeDTKtoDPF.C" program is compiled into a DSO, which can then be loaded to or unloaded from a running DPF application. Once loaded, the DPF application reads the values in the "attitude" shared memory segment at a high rate, such that when the "writeEuler.C" program writes the pre-generated Euler angles at its specified rate, the model in the CAVE appears to be updated at nearly the same rate.

The `dtk-floatSliders` program seen in Fig. (7) is included with DTK, and is a simple GUI to read or write to shared memory. This is used to verify the Euler angles are being written to the "attitude" shared memory properly, as well as manually move the Whorl1 model to verify the "attitudeDTKtoDPF" DSO is working properly.

4.3 Animation Method 2: Rate Gyro Attitude Data

The second method to animate the CG model of Whorl1 in the CAVE uses actual attitude data sensed by the rate gyro located on the spacecraft simulator. The system used here, seen in Fig. (9), is notably more complicated then the previous system. However, since the CAVE application is updated using the actual information from the spacecraft simulator, this system has more potential uses. It should be noted that some of the components in this system are temporary solutions to problems encountered during its development, and more efficient means with fewer separate programs should be used in future implementations.

This section will be divided into descriptions of the system, problems encountered, and early experimentation.

4.3.1 Organization of System

The rate gyros measure the $\dot{\boldsymbol{\theta}}$ expressed in the body frame, with $\mathcal{F}_b$ specified on the inertial measurement unit (IMU) (Fig. (10). The reference frame shown in Fig (6) approximately corresponds to $\mathcal{F}_b$ as defined by the orientations of the rate gyros. However, there are some small discrepancies between the two: the IMU is not located exactly at the center of the body, as shown in the CG model, and the test-bed as shown in Fig (6) is not oriented properly with respect to the $\hat{\mathbf{b}}_\mathbf{1}$ and $\hat{\mathbf{b}}_\mathbf{2}$ axes. Due to other limitations in this system described below, the main interest is rotation about the $\hat{\mathbf{b}}_\mathbf{3}$, and thus these discrepancies are negligible for these preliminary tests. Also, since this early CG model of Whorl1 is a simplified version of the actual test-bed, the orientation of the reference frames in the CAVE simulation is the primary concern.

The first part of this system is the program called "dmuDTKTest.c" running on Whorl1's PC-104 computer. This program is largely based on the code written previously[2] to read the data from the IMU. The additional code written for the current system writes the $\dot{\boldsymbol{\theta}}$ values to a socket connection over the wireless LAN. The socket is read by the "dmu2ShMem.C" program running on a PC located in the SSSL. This PC uses Debian Linux as its operating system, and has a connection to the wireless LAN hub and an ethernet connection to the internet. The main purpose of this program is to write the $\dot{\boldsymbol{\theta}}$ values to the "thetaDot" shared memory segment for use by other local processes.

Next, the "thetaDot2attitude.C" program reads the $\dot{\boldsymbol{\theta}}$ values from the "thetaDot" shared memory, performs a numerical integration, and writes the resulting $\boldsymbol{\theta}$ values to another local shared memory segment, "attitude". The integration is a simple Euler integration, where the average of the previous value and the current value is multiplied by the time interval. This program also uses the methods in the `dtkTime` class to control the time-dependant calculations.

[2]This previous code was written by Cengiz Akinli of the SSSL. All of the calibration and bias values in this code remains the same as determined previously by him.

The `dtk-floatScope` program is another GUI included with DTK, which is useful for evaluating different components in the system. This program simulates an oscilloscope by reading from shared memory and displaying the values in a time-swept graphical form. Fig. (11) shows two `dtk-floatScopes`'s connected to the "thetaDot" and "attitude" shared memory segments. The noisiness of the IMU data can clearly be seen in the "thetaDot" values displayed on the oscilloscope.

One the CAVE-side of the system, there is another local DTK shared memory segment call "attitude". Like the first system using the pre-generated attitude data, the values in "attitude" are read by the "attitudeDTKtoDPF" DSO and the CAVE model is updated accordingly. Thus, what remains is to connect the two shared memory segments using the `dtk-server` program.

4.3.2 Early Problems

This remote shared memory connection leads to one of the major problems encountered during the development of this system, which concerns compatibility issues between the GNU/Linux and SGI IRIS operating systems. Namely, IRIS uses the so called "big-endian" byte ordering convention, whereas GNU/Linux uses the "little-endian" convention. Thus, even after the two shared memory segments are connected, the `float` values read by the program running on IRIS are meaningless since the byte order is different.

Apparently there exist functions to correct this problem, such as the `swab` system call. However, after spending time attempting to alleviate the problem this way to no avail, a "homemade" solution was devised. Thus solution is by no means the most efficient way, and should be replaced in future systems. The basic approach is to break each `float` theta value up into 5 integer fields: 1 field for the sign, the other 4 fields each hold 2 digits, *i.e.* the floating point angle -125.426286654... has the fields $\begin{bmatrix}0 & 01 & 25 & 42 & 63\end{bmatrix}$. The first field is a 1 or 0 depending on the sign. These fields are then converted to type `char`, giving a total of 15 `char`'s for the 3 Euler angles. This is the reason for the "charAttitude" shared memory segments seen in Fig. (system2); since the `char` values in shared memory are only 1-byte long, there are no byte-ordering problems.

However there are disadvantages associated with this method. The obvious one is that the system requires two extra shared memory segments to hold the 'charAttitude" data. Also the "att2charAtt.C" and "read3charTest.C" programs must be used to convert the attitude from 3 `float`'s to 15 `char`'s, and back to 3 `char`'s. Some fidelity is lost in the attitude data as well, since only 4 decimal places are retained. Despite the added complexity, this is a sufficient "quick-fix" for this preliminary version of the system.

Another problem worth mentioning is associated with the data read from the rate gyros. When the angular rates were displayed on the `dtk-floatScope` oscilloscope program while the simulator was rotated by hand, it became apparent that for some

rotations, the angular rates became either discontinuous or fluctuated between extreme values. The only usable values were obtainable through carefully controlled rotations of the simulator. For example, if the simulator was balanced and nearly brought to rest, the angular rates about all three axes fluctuated close to zero. If the simulator was rotated slowly, less than approximately 5 deg/sec in a negative direction about the $\hat{\mathbf{b}}_\mathbf{3}$ axis, with the other two rotations kept at a small rate, the angular rates appeared correct despite slight noise in the readings. However, if the simulator was rotated in the opposite direction, or the rotations about the $\hat{\mathbf{b}}_\mathbf{1}$ and $\hat{\mathbf{b}}_\mathbf{2}$ axes exceeded approximately 1 deg/sec, then the values became discontinuous and unusable.

4.3.3 Experimentation

It was determined that a simple test could be performed if the simulator was rotated by hand within the constraints described above. Once the connection was established between the two sides of the system, and the attitude data was converted successfully, a simple test was conducted to animate the CG model in the CAVE using the actual motion of the simulator. This required one person in the SSSL to move Whorl1 by hand, and another person in the CAVE to view the graphical simulation.

First, Whorl1 was brought as close to rest as possible, so that when the numerical integration began, the initial conditions for the Euler angles could be assumed as zero. Stated another way, $\mathcal{F}_b$ was assumed to be initially aligned with $\mathcal{F}_i$. The simulator was then slowly rotated about the -$\hat{\mathbf{b}}_\mathbf{3}$ axis.

Data from this experiment were output to a text file, which were plotted in MATLAB and appear in Figs. (12 & 13). Similar to the `dtk-floatScope` display, the plots show the variations in $\dot{\boldsymbol{\theta}}$ and $\boldsymbol{\theta}$ over time. Fig. (12) shows that $\dot{\theta}_3$ increased from near 0 deg/sec to about 2 deg/sec during the first 10 seconds, at which point it continued to rotate at that rate until around 40 sec. During this same time, $\dot{\theta}_1$ oscillated between 0 and 0.5 deg/sec and $\dot{\theta}_2$ oscillated between 0 and -1 deg/sec. This indicates the table was likely off balance slightly, combined with the fact that the initial torque given to the simulator was likely not entirely about the -$\hat{\mathbf{b}}_\mathbf{3}$ axis. Likewise, the plot in Fig. (13) for this time span shows a steadily increasing θ_3, whereas the other two rotations remain near zero.

At a time of 40 seconds, an additional toque was applied by hand to the simulator, which is seen in the rise in $\dot{\theta}_3$ from 2 deg/sec to about 4 deg/sec. As expected, θ_3 increases at a steeper rate in Fig. (13). Likewise, the oscillations about the other two axes increase in magnitude, such that beginning at a time of 50 sec, the $\dot{\theta}_2$ value jumps discontinuously to values around 12 deg/sec. Because the numerical integration is still including these extreme values, the θ_2 line begins veering off course, at which point the simulation begins to break down.

4.3.4 Discussion

Despite the noise in the data, and the limited range of movement for which the measured values have meaning, this early experiment serves to show the potential for this system to be used to accurately visualize the motion of the spacecraft simulator in a VE. Some factors that need addressing include: proper selection of biases or other scaling factors for the IMU, further understanding of why the $\dot{\boldsymbol{\theta}}$ become discontinuous at some levels, and the use of other attitude measuring techniques. Ideally, for any future system to be useful, multiple attitude sensors should be included in a Kalman filter.

However, in the short term, there are definite areas in which the current system can be improved. The problem described previously regarding the cross-platform compatibility can be solved in a more elegant way. This will reduce the number of different programs and shared memory segments in the system.

With regard to control, the next step is to attempt to control the simulator from within the VE, as part of some manual control scheme using the compressed air thrusters. While this might prove difficult given the current sensitivity of the system, as illustrated above, it actually presents an interesting opportunity. One of the issues this research hopes to address is how well an operator can control the spacecraft simulator, given certain constraints, from within a VE. As described in the literature review, this normally might take the form of clearance from structures, approach velocities, and other issues related to PROX-OPS. In this case, the constraints involve limitations in the current spacecraft simulator hardware, such that it can only be moved in a certain way in order for the attitude data to have meaning. Thus, even though the constraints are somewhat self-imposed, it still allows for further studies to be conducted.

5 Conclusion

The following topics were presented in this paper. A summary of the literature review was given, highlighting the issues and applications related to manual control of spacecraft, associated display types, and uses for VE's. This review was not limited to spacecraft, but included other remotely operated vehicles and control methods. Two different methods were experimented to animate a model of the spacecraft simulator in the CAVE. One involved numerical integration of the kinematic differential equations of motion to generate rotation angles off-line. The other used attitude data from the rate gyro hardware on the simulator, which were sent over the network at near realtime. The problems associated with the current system were discussed, and ideas to work around these problems in the short term, as well as long term goals, were given.

References

[1] Kelso et. al. DIVERSE: A Framework for Building Extensible and Reconfigurable Device-Independent Virtual Environments and Distributed Asynchronous Simulations. *Presence*, 12(1):19–36, Feb 2003.

[2] A. R. Brody et. al. Interactive displays for trajectory planning and proximity operations. *Journal of Spacecraft and Rockets*, 30(4):514–518, Jul-Aug 1993.

[3] A. J. Grunwald and S. R. Ellis. Visual display aid for orbital maneuvering: Design considerations. *Journal of Guidance, Control, and Dynamics*, 16(1), Jan-Feb 1993.

[4] A. R. Brody and S. R. Ellis. Manual control aspects of space station docking maneuvers. In *Proceedings of the 20th Intersociety Conference on Environmental Systems*, Williamsburg, Virginia, Jul 1990. Society of Automotive Engineers.

[5] Personal discussion, Chris Hall, Professor, Virginia Tech Department of Aerospace and Ocean Engineering, Jan 2003.

[6] A. J. Grunwald and S. R. Ellis. Visual display aid for orbital maneuvering: Experimental evaluation. *Journal of Guidance, Control, and Dynamics*, 16(1), Jan-Feb 1993.

[7] M. W. McGreevy and C. Stoker. Telepresence for planetary exploration. *The International Society of Optical Engineering*, 1387 *Cooperative Intelligent Robotics in Space*:110–123, 1990.

[8] Carol Stoker. From antartica to space: Use of telepresence and virtual reality in control of a remote underwater vehicle. *The International Society of Optical Engineering*, 2352 *Mobile Robots IX*:288–299, 1994.

[9] L. A. Nguyen et. al. Virtual reality interfaces for visualization and control of remote vehicles. *Autonomous Robots*, 11:59–68, 2001. Kluwer Academic Publishers.

[10] K. Museth et. al. Semi-immersive space mission design and visualization: Case study of the terrestrial planet finder mission. In *SIGGRAPH Proceedings on Visualization*, volume 21-26, pages 501–599. IEEE, Oct 2001.

[11] H. L. Alexander. Experiments in teleoperation and autonomous control of space robotic vehicles. In *Proceedings of the 10th American Control Conference*, pages 1474–1477, Boston, Massachusetts, Jun 1991. IEEE.

[12] Eckel et. al. *OpenGL Performer: Getting Started Guide*. Silicon Graphics Inc., 2002.

[13] George Eckel. *OpenGL Performer Programmer's Guide*. Silicon Graphics Inc., 2000.

[14] Chris Hall. Aoe 4140: Spacecraft attitude determination and control, class notes. Spring, 2003.

[15] Bruce Littlefield Duane Hanselman. *Mastering MATLAB6: A Comprehensive Tutorial and Reference*, chapter 24, page 334. Prentice Hall, 2001.

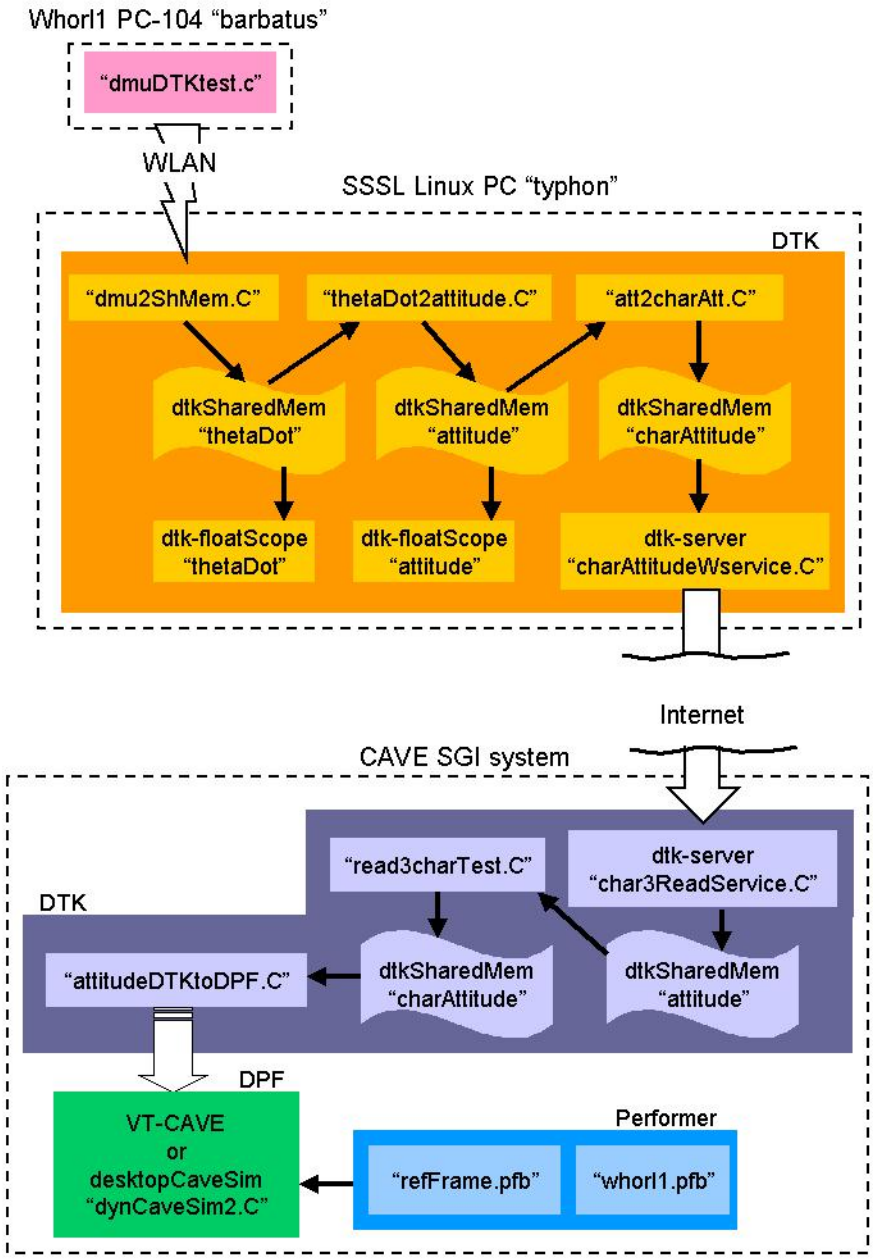

Figure 9: Diagram of system used in second animation method.

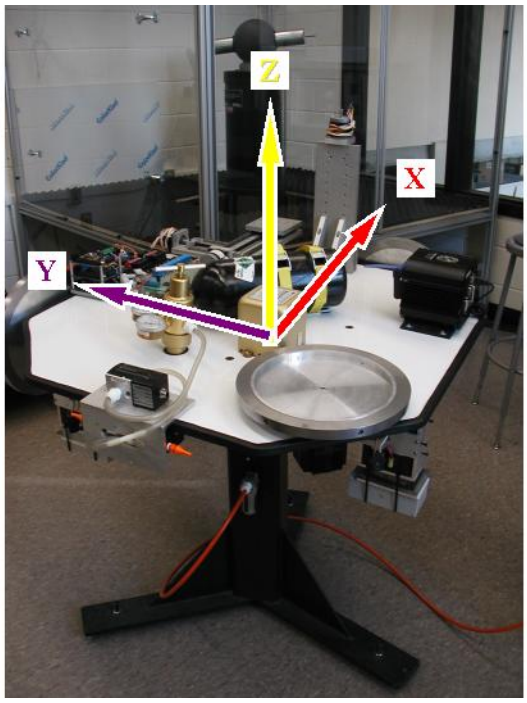

Figure 10: Photo of Whorl1 simulator with illustration of body frame as defined on IMU.

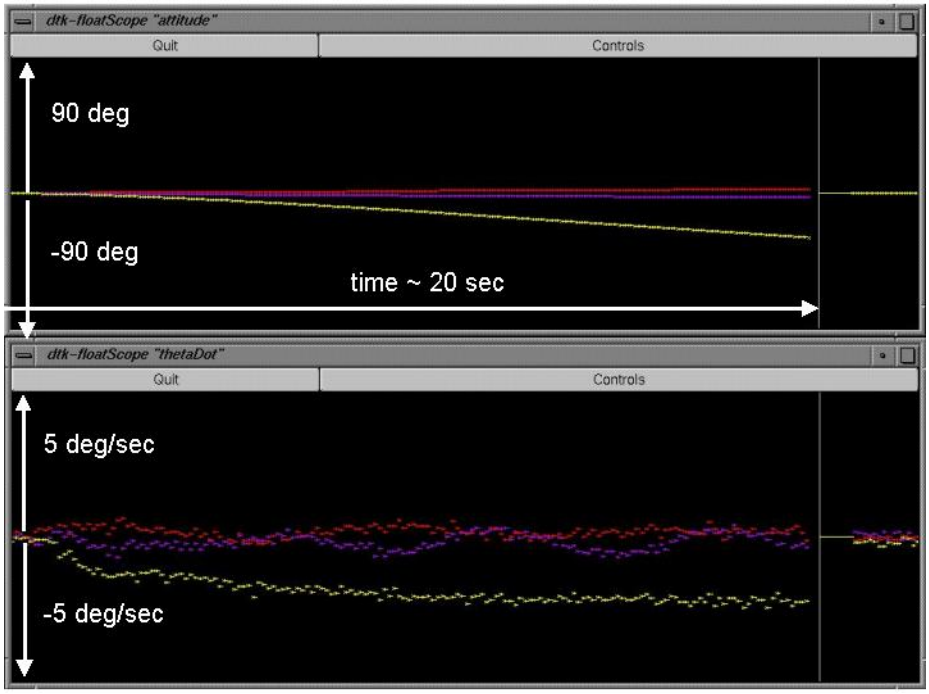

Figure 11: Screen shot of oscilloscope program included with DTK, here displaying "attitude" (top) and "thetaDot" (bottom). Annotations showing scale were added afterwards.

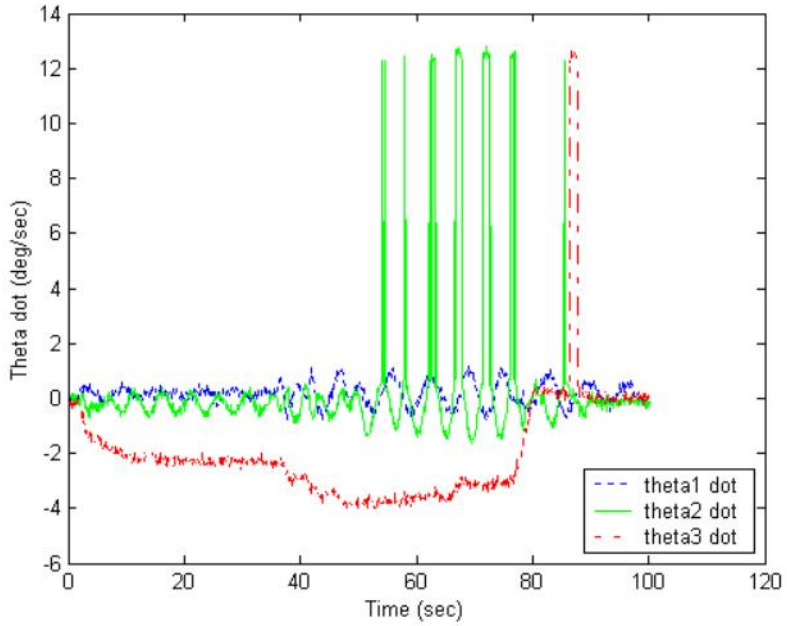

Figure 12: MATLAB plot of Euler angle rates as read from the IMU.

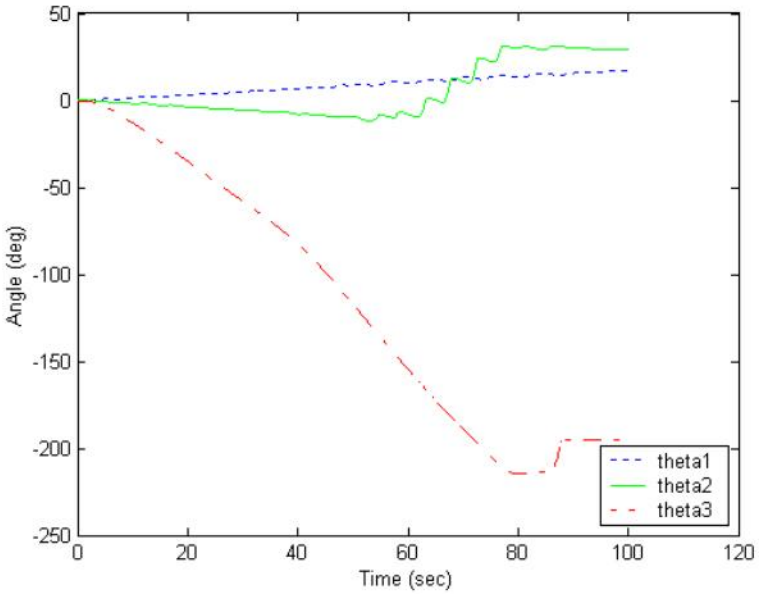

Figure 13: MATLAB plot of Euler angles, obtained from numerical integration.