1-1-2006

A phantom interface for the teleoperation of a mobile platform over the internet

Adam Eugene Bogenrief
Iowa State University

A Phantom interface for the teleoperation of a mobile platform over the internet

by

Adam Eugene Bogenrief

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Mechanical Engineering

Program of Study Committee:
G. R. Luecke (Major Professor)
James Bernard
Yan-Bin Jia

Iowa State University

Ames, Iowa

2006

Graduate College

Iowa State University

This is to certify that the master's thesis of

Adam Bogenrief

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The ability to teleoperate a mobile vehicle over the internet is a difficult task. Many sensory signals must be processed by the user in order to make an informed and safe vehicle-guiding decision. So much so, that the interface design is often the downfall of an otherwise capable mobile robot. A good interface should be able to relieve the user of some of the visual sensor strain and still result in a safely controlled mobile platform. The research detailed in thesis first, describes the construction of a reliable and sensor rich platform for remote vehicle control. Then an interface is developed that adds haptic sensing to divert some of the strain from the operator, resulting in an easy-to-drive remote vehicle application.

CHAPTER 1. OVERVIEW

1.1 Background

Advancements in new technologies are changing the way vehicles are being driven. Good sensor data can allow a person to get enough information about a vehicle's environment in order to drive a vehicle remotely. Some applications where it is necessary to drive a vehicle to destinations remotely already exist. The military has converted many land, sea, and air vehicles into remotely driven vehicles [12, 16]. NASA even uses remote driving to control vehicles on Mars [12, 13]. Search and Rescue teams also use remotely controlled vehicles in environments that are deemed too dangerous for human involvement [8].

Teleoperation is the term used to describe remote vehicle operation. In order to achieve teleoperation, signals garnered from various sensors attached to a mobile platform must be sent to a human at a remote location. The human uses these signals to reconstruct the robot's environment in order to make an informed decision about where to move the robot. The movement commands are then sent back to the robot to be executed. The pieces of hardware and software that combine to implement the previously mentioned steps are known as the interface [8]. Some existing interfaces can be seen in any number of literary robotic papers [8, 12, 15].

1.2 Traditional Issues

There are three main difficulties encountered when designing a teleoperation interface. First, choosing the right sensors can be challenging. If the user is unable to "see" an obstacle, how can the robot be intelligibly controlled away from harm? Second,

communication lags can hinder progress as well. Even if the user "sees" the obstacle, if too much time has passed between sending the motor commands and receiving the motor commands, then how can the robot move out of harms way? Third, the control device can misrepresent the robots actual abilities. Simply because the user can move the joystick, or some other form of control, rapidly away from an object, does not mean that the robot can respond as quickly.

1.3 State of the Art Issues

Even though current teleoperation interfaces have been reported as effective [8, 12, 15], there still exist some problems. One such problem occurs when attempting to solve the first traditional issue. The user surpasses their ability to multitask. Too much sensor data is displayed in such a way that adequate processing is impossible. This leads to poor control over the movement of the mobile robot.

Communication lags are still a big problem in teleoperation interfaces despite the vast capabilities of current electronics. Since instantaneous transfer of data will most likely never exist, this is a problem that will need to be addressed by every teleoperation interface. This can be most problematic when dealing with internet communication, large unpredictable lags can result in poor control over the mobile platform.

The current trend for user input devices is to utilize either a graphical user interface (GUI) system, or a gaming joystick. Admittedly, the two previously mentioned methods for input are somewhat effective, but seem to lack a sense of realism. The user is unable to get a sense of "feel" for the limitations of the mobile robot or its environment. The lack of tactile data can also result in poor control over the mobile robot.

1.4 Assessment of Issues

With so many issues plaguing the teleoperation interface designer, the question may arise as to whether or not it is possible to create a mobile platform that is capable of conveying as much sensory information to the user as possible, without overwhelming the user? To answer that question, this thesis will detail the construction of a mobile platform that gives the user enough sensory data in order to get a good visual approximation of the mobile platform environment.

Another question that may arise is how to overcome the very important lag issues? The answer to that question has two parts. First, an alternate route to the standard video streaming technique is covered. It will be shown that the new method will make it possible to get very near to real time video exchange. Second, the methods proposed by this internet interface are privatized and not freely available to anyone with an internet connection. That being the case, the relative distance that the data needs to travel is shortened significantly, thereby reducing some of the conventional internet lag.

A final question that may arise is how to get the user to process more sensor data input without overwhelming the senses? The answer to this question, as proposed by this thesis, is to use a force feedback device to handle user input device. In this way, the device could be programmed to handle some sensory data and then relay it to the user via the tactile senses. In this way, the amount of visual sensor data could be lessened.

A fortunate side effect to the successful answering of the previously posed questions was discovered. The control of the mobile platform was achieved with adequate enough results to handle a considerable speed of travel increase. Along with this speed increase, there was also the increased possibility of sliding out of control. To protect against slipping

and not reduce the overall top speed, a vehicle slip model will be proposed that manipulates force feedback effects to convey vehicle slippage to the user.

1.5 Thesis Map

Chapter 2 is a literature review. The purpose of this section is to detail the approach taken by some current solutions to the traditional issues. In that way, it could be proven that the there is a need for the work presented in this thesis and that the approach developed in this thesis is competent enough to compete with aspects of other intellectually respected works.

Chapter 3 is a hardware description of the mobile platform. The purpose of this section is two fold. First, it is necessary to let the reader know the overall design of the interface so that they can see the importance of each component in regards to the finished robot. Second, there was a lot of hardware used to create the interface. An introduction to all of the hardware functionality is necessary to understand the technical development.

Chapter 4 is a technical development of the work presented by this thesis. The purpose of this section is to detail how the components work. First, details are needed to explain how the raw sensor values of the environment are converted to a form useable by other components. Second, the communication between components needs to be detailed. And third, it is necessary to detail how the data will be converted into a format that the user can understand.

Chapter 5 is the result attained by implementing chapter 3 and 4. The purpose of this section is to show what was accomplished. This section is only composed of two photos; one photo to show the finished mobile platform and the other to show the finished interface.

Chapter 6 is a conclusion to bring the work of this thesis to a close. The purpose of this section is to explain and convey validity of the results. In other words, to prove that something meaningful and of intellectual value was accomplished.

Chapter 7 is a development of future work that could be done to further this thesis. The purpose of this section is to detail what changes could be performed in order to improve upon the presented work. Nothing is perfect.

CHAPTER 2. LITERATURE REVIEW OF STATE OF THE ART

2.1 Effective Vehicle Teleoperation on the World Wide Web

Grange, Fong, and Baur collaborate to create a unique web-based teleoperation interface. They believe that "a well designed vehicle teleoperation system requires effective and efficient use of the communication link, a clear and compelling operator interface, and an architecture which supports local autonomy." But they put the most emphasis on a lack of internet band width and lag predictability and so, designed their interface with that issue at the forefront of most of their decisions.

The paper goes on to detail the result of their interface implementation named WebDriver. The WebDriver user interface contains a dynamic environment map built from on board sensors and an image manager to handle video. For the dynamic map, everything is color coded and color intensity is used to convey a sense of location certainty. To cut down on internet overhead, the team uses an event driven system to display still images to the user.

The system architecture uses a Java-applet to control a virtual robot. The virtual robot then sends input commands to the physical PioneerAT remote platform; although the group claims the virtual robot can also communicate to any number of different mobile platforms. Since they let the mobile platform handle all the obstacle avoidance, the communication overhead can be limited, resulting in a "quicker" response of their system.

This particular application brought to light many of the concerns that should be addressed by a web-based interface, such as the one proposed by this thesis. First and foremost, there is a need to limit the communication delays in some way. Second, the user can be overwhelmed with sensory data so limit the control of the user. Third, make sure to

still provide enough sensory data so that an accurate estimation of the environment can still be made. And fourth, the interface has to be easy to use. No one wants to read a detailed list of instructions, or spend hours learning how to control the robot.

Some things were found to be lacking in this paper. First, the authors are trying to pitch teleoperation, or remote control, of a vehicle. But, it appears as though the only control that the user has is in the final destination of the robot. The robot has so much local autonomy that the user interface can not be enjoyable. What is the point of controlling a robot that capably drives itself? Second, the user interface only updates the video image when not busy, on request, or after 5 seconds has elapsed. How does this adequately represent the vehicles current environment to the user? These ill conceived designs will be noted and considered in the design of this haptic interface.

2.2 Interface Evaluation for Mobile Robot Teleoperation

Olivares, Zhou, Bodenheimer, and Adams present the results of a study on teleoperation interfaces. The group is concentrating their efforts on the user interface, more specifically on how the sensory information is presented to the user. The authors tested three separate interfaces named Nilas, Express, and Cockpit. Each of the interfaces had a myriad of sensor data spread over many different windows. The first two interfaces allowed manipulations of the windows in various and no doubt confusing manners. The third interface allowed only "limited" manipulations of the windows.

The study was conducted on two groups of people. The first group was composed of 12 "expert" teleoperation users and the second was composed of 24 "novices". The two

groups were trained on the three separate interfaces and then had to complete a questionnaire. The results were presented and discussed, but were not worth mentioning.

Despite the lack of profound knowledge gained from this publication, a valuable lesson was learned. Presenting the user with a vast number of windows is ineffective. A better approach would be to design the interface with only 1 or 2 windows. That way the user would spend most of their time controlling the robot, and not commanding the interface.

2.3 Issues in Vehicle Teleoperation for Tunnel and Sewer Recon

Laird, Bruch, West, Ciccimaro, and Everett present an application for military teleoperation in the form of a tunnel searching robot. The authors detail two generations of design considerations before creating the final interface. The first iteration was created simply to get feedback from users in an effort to then create an improved interface.

There were several lessons learned in the testing phase. First, a control approach was preferred when the user had explicit control over the robot, i.e. there was no local autonomy. Another lesson learned by the researchers was the input device had to be easy to use. If user attention had to be diverted from the video feedback to the hand controls to see where to move the robot next, the result would yield poor control of the robot.

The second generation of the user interface fixed many of the previously discussed issues. To aid the user in feeling the control a tactile "feel" of the input buttons was implemented. This allowed for the full attention of the user to be concentrated on the video feedback.

While this paper did not detail a web-based mobile robotic teleoperation application, many good interface design tips could be taken from it. Most important is the use of a tactile

device to give additional feedback to the user. Second, this paper helped solidify the belief that local autonomy takes away from the teleoperation interface. Attempts should be made to let the user control the mobile robot as much as possible.

2.4 Haptic Teleoperation of a Mobile Robot: A User Study

Lee, Sukhatme, Kim, and Park detail their results of hapticly controlling a mobile platform using a phantom. The same input device as proposed by this thesis. While there are some similarities in the equipment used by the authors of this paper and this student, there are enough differences so as not to warrant copyright fraud. These differences will be detailed below.

First, the authors use a pre-made mobile platform where this thesis will detail the construction of a self-made platform. Second, the authors of this paper develop a shared autonomy approach to handle obstacle avoidance. It has been made clear in the preceding sections that no local autonomy will be used in this research. Third, the authors use the device to control the mobile platform speed and turning rate. This thesis will use the device to control the mobile platform speed and an absolute turn angle, not a rate. Fourth, the authors go into a lengthy development the obstacle avoidance algorithm used to calculate the forces sent to the phantom. This thesis will not detail any such force-aided obstacle avoidance. And fifth, the authors only attempt low speed driving applications and fear for their very expensive equipment. One goal of this thesis is to operate the mobile platform at as fast of a speed as can be "safely" attained.

CHAPTER 3. HARDWARE DEVELOPMENT

3.1 Phantom Interface Design

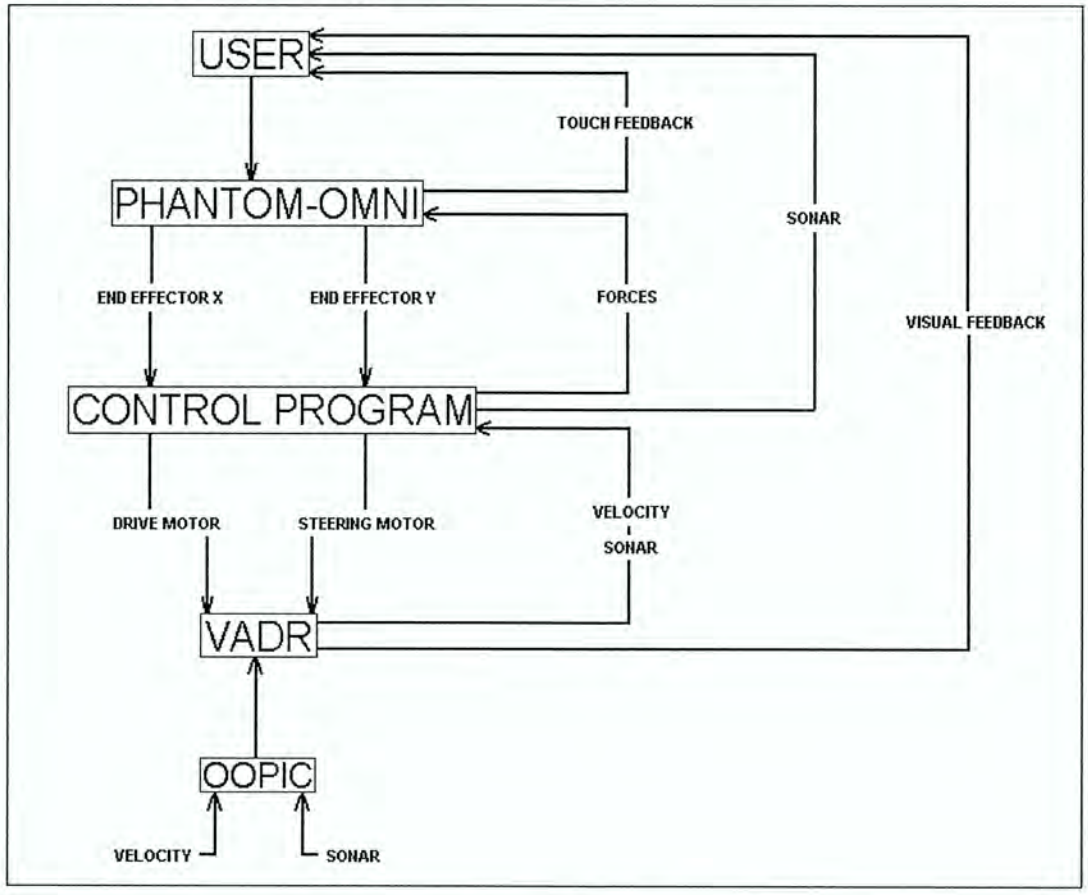


Figure 1. Interface Flow Diagram

Figure 1 above, gives a visual of how the interface works. It is fairly easy to follow. First, the user views the video feedback and sonar readings to get an idea of the mobile platform's environment. The user then decides what to command the robot to do and actually sends the commands by moving the phantom input device. The device X and Y are polled by the control program, converted to commands for the steering and drive motors, and then sent to the mobile platform. The mobile platform executes the motor commands and

records any velocity and sonar changes. The velocity and sonar changes are then sent to the control program. The control program uses the velocity reading to help calculate the forces that are sent to the phantom and at the same time pass on the sonar readings to the user. Next, the phantom input device transmits the forces to the user. And finally, the cycle repeats. Giving the user both visual and tactile data to help describe the environment.

3.2 Mobile Platform

Fully functioning, pre-built mobile robotic platforms like the Pioneer AT , the Pioneer 2-DX , and Khepera [8, 12, 13, 15] can be purchased on the World Wide Web for a substantial amount of money (somewhere in the range of several tens of thousands of dollars). They can have a myriad of sensors and computing power, depending on the platforms intended use; of course, anything is available for a price. However, for this interface application a pre-built mobile, platform was not bought. Instead, a platform was put together from mostly commonly available items. The reason behind this choice was very simple. It was far cheaper to build a platform than to purchase one. Other driving factors behind building a platform were to gain knowledge about design and construction issues encountered during the creation of a fully functioning mobile platform. Also, to create a base platform that could be used in any number of other mobile robotics research projects.

For this reason, the platform received the name Vehicle for Autonomous Driving Research (VADR). From this point on, the mobile platform will be referred to as VADR. It should be noted that VADR has been through two previously unsuccessful design and construction phases where some of the components may have been previously utilized. The knowledge gained from the two previous failures prompted the improved design that will be

detailed in the next section. However, the two previous attempts will not be detailed any further in this paper.

3.2.1 VADR Base

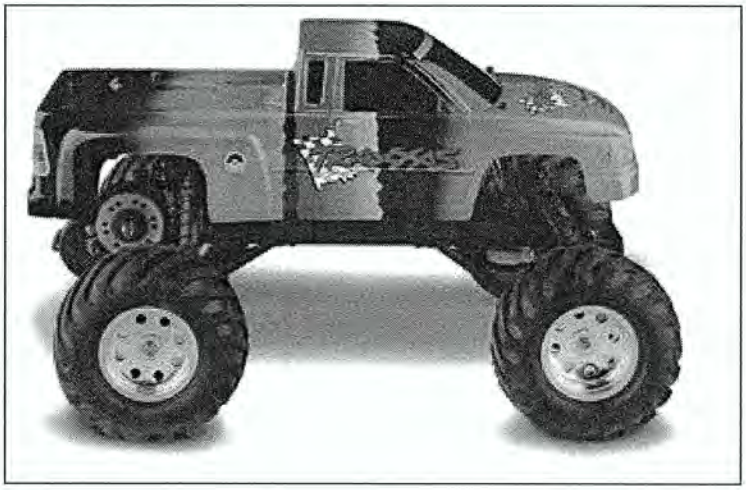


Figure 2. Traxxas Stampede

The base for VADR started out as a Traxxas Stampede RC truck, see figure 2 above. There were two main reasons for choosing an RC truck for the base. First, since the ultimate goal is to be able to teleoperate a real vehicle, the mobile platform should have the same characteristics and responses as an actual vehicle. The RC truck base does this in the form of an Ackerman drive train [6]. The Ackerman drive train takes advantage of 2 motors. A servo motor controls the directional input and a geared DC motor controls the speed input. Second, the plastic frame is easy to modify.

In order to allow for the intended conversions, it was necessary to make some cosmetic changes to the appearance of the original RC truck base. First, the plastic truck cover was removed and discarded, giving access to all the internal electronics and also giving access to a flat base from which to mount the electronics frame. Next, the plastic mounting

posts used to support the truck cover were removed and discarded. After that, the radio frequency (RF) receiver originally used to receive user input to the two motors was removed. And finally, the original shock absorbers were replaced by straight, solid rods. Unfortunately, the shock absorbers were not designed to handle the added weight of the frame and electronics.

3.2.2 VADR Electronics Frame

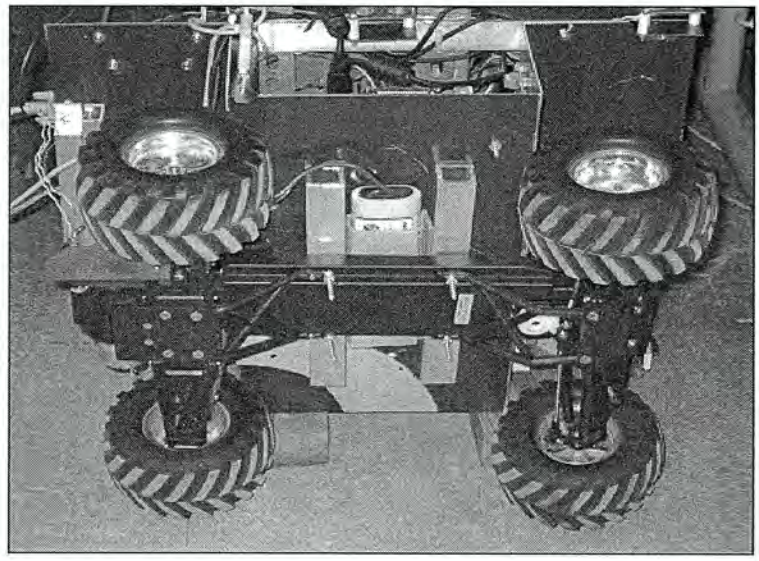


Figure 3. Aluminum Frame

A thin sheet of aluminum was chosen for the frame material. As aluminum has enough strength to support all the added electronics, but at the same time is flexible enough to be bent to the final shape. The final shape, as seen above, was constructed to maximize the amount of space available for mounting all the computing components and sensors and at the same time stay compact enough to not move the center of gravity (CG) into an undesirable alignment that would alter the responsiveness of VADR. Two thin strips of aluminum were then attached to the frame to add rigidity and support for the two extensions.

The frame was then lifted above the RC base by two square aluminum tubes to add spacing for the drive train battery. The final base structure can be seen in figure 3.

3.2.3 VADR Computer

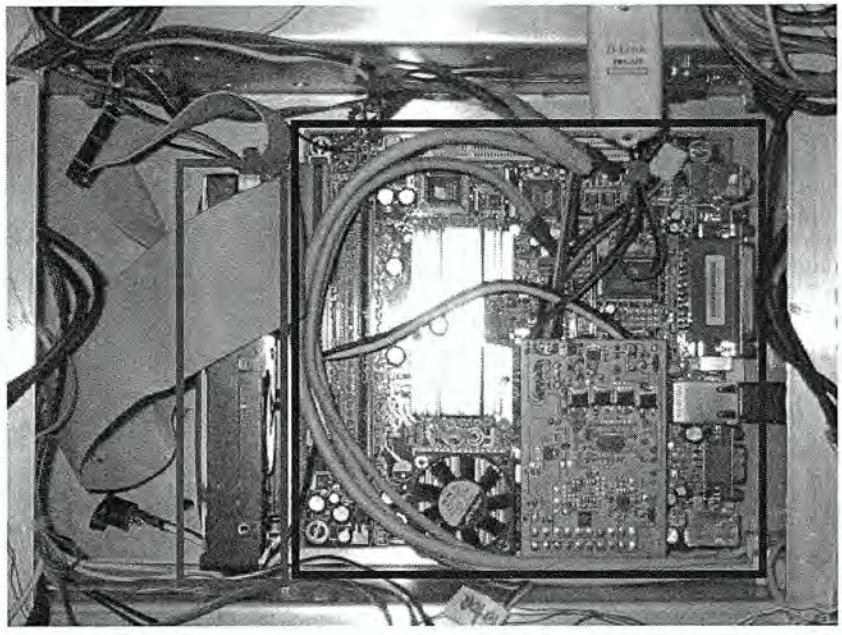


Figure 4. Mini-ITX

It became apparent that running all the sensors, motors, and communications would require quite a bit of processing power. This led to the addition of a computer as one of VADR's components. In an attempt to conserve space, an EPIA Mini-ITX M10000 motherboard was chosen. A detailed specifications list is available in Appendix A, page 50. As evidenced by the blue box in figure 4, the mounted motherboard takes up only slightly more than half of the space available in VADR's recessed center section.

Unfortunately, the motherboard did not come with a hard drive attached, so one had to be added. The hard drive specifications can be seen in Appendix A, page 50. The hard drive was mounted next to the motherboard, as evidenced by the red box in figure 4.

The criteria for choosing an operating system for the VADR computer were very simple. The operating system had to be cheap and easy to use. Ubuntu, a version of the Linux operating system was decided upon and installed on the VADR computer.

3.2.4 VADR Phidgets

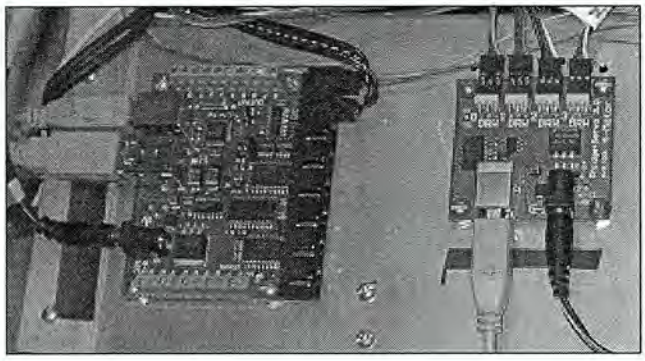


Figure 5. Phidgets

Phidgets were chosen to handle the input output (IO) interface to the computer. A couple of reasons why phidgets were chosen are that they are cost effective and connect to a computer via USB. Another plus for phidgets is that they can be programmed using any of the major programming languages.

Two different phidget boards were chosen for VADR components. To handle sensor IO to or from the computer, a phidget interface kit has 8 digital inputs, 8 digital outputs, and 8 analog inputs. The interface kit was mounted on the rear extension of VADR, as seen on the left hand side of figure 5. A detailed description of the phidget interface can be seen in Appendix A, page 52.

To handle motor control, a phidget motor control kit has 4 pulse-width-modulated (PWM) outputs. The motor control kit was mounted on the rear extension of VADR, as seen

on the right hand side of figure 5. A detailed description of the phidget motor control kit can be seen in Appendix A, page 53.

3.2.5 VADR OOPIC

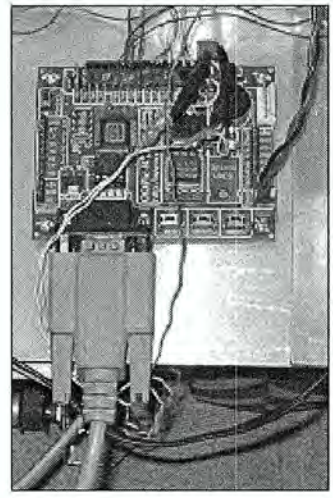


Figure 6. OOPIC

Figure 6 shows an Object-Oriented Programmable Integrate Circuit (OOPIC) microcontroller. Admittedly, the OOPIC and the phidget interface kit have the same hardware functionality; the only significant difference is that the OOPIC communicates serially. A detailed specification of the OOPIC can be seen in Appendix A, page 54.

The advantage of adding an OOPIC as a VADR component comes in software. The OOPIC has some standardized sensor objects that are very useful and easy to implement. More specifically, the OOPIC has an object to control both an SRF-04 sonar module as well as a quadrature encoder. The time saved by not programming the same functionality of the OOPIC object into the phidget interface kit more than justified it's addition to VADR.

3.2.6 VADR Sonar Range Finders

Figure 7. Sonar Range Finder

Figure 7, shown above, is a picture of an SRF-04 Devantech sonar range finder. The SRF-04 uses a sonic ping and echo to determine the distance to an object. There are several sensors that have reliably been used in robotics applications to determine the distance to an object. The list includes infrared light detectors, sonar detectors, and laser range detectors. Although laser range detectors offer more accurate results, they unfortunately cost too much to be a practical solution for VADR. Infrared however, is much more aptly priced, but the limited range of these sensors rendered them not practical. The sonar detectors are moderately priced and operate over a large enough range to be considered a practical choice for the interface.

The sensor is composed of an ultrasonic transducer and receiver pair. The pair is mounted next to each other on an integrated circuit (IC) board. The sonic ping emitted by the sensor flows out in a 3-D cone. A 2-D footprint that models the usable range of the SRF-04 can be seen in Appendix A, page 55. The IC board also provides the necessary IO to access the distance information. The trigger and echo pins were wired to two IO pins on the

OOPIC. This allowed the microcontroller to get an object's distance by giving the sensor a trigger pulse and waiting for the time it takes the echo to come back.

The time of flight of the sonar ping is then recorded in an OOPIC software object. The program necessary to process the sonar information is detailed in the sensor section in the technical development portion of this thesis. Five such sensors were mounted to the mobile platform and wired to the OOPIC, see figure 8 below.

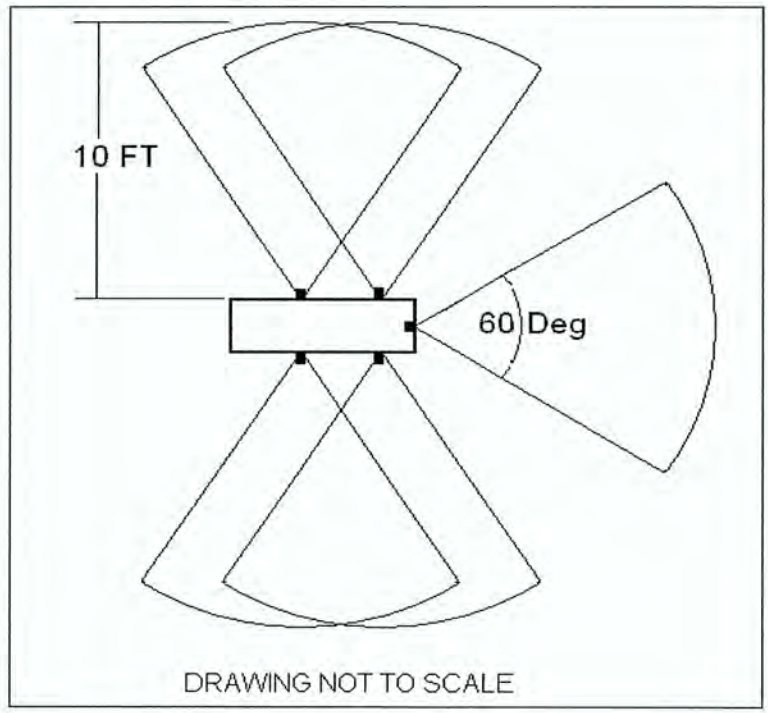


Figure 8. Sonar Locations

3.2.7 VADR Velocity Sensor

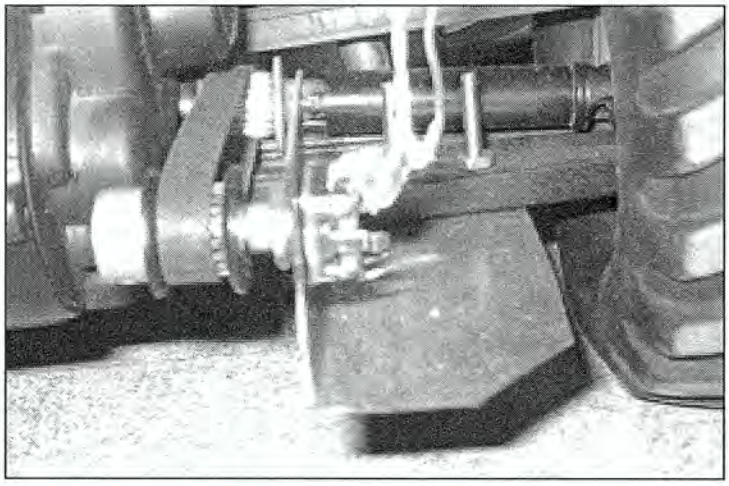


Figure 9. Encoder

Figure 9 above, details the hardware portion of VADR's velocity sensor. The sensor is essentially an incremental encoder. There are several encoders that have been used reliably in robotics applications to count the revolutions of a shaft. They include mechanical contact encoders, optical encoders, and magnetic encoders. Mechanical encoders are fairly cheap. Unfortunately, they are rated for low speed applications and have a short life cycle, rendering them useless to VADR. Magnetic encoders can operate in an acceptable range. However, they are so expensive that they are not a practical choice for VADR. Optical encoders also operate in an acceptable range and are only moderately expensive, making them the only practical choice.

The sensor was composed of a mounting bracket, gear and belt system, and an encoder. A detailed datasheet for the encoder can be seen in Appendix A, page 56. The mounting bracket was constructed by bending a thin piece of sheet metal to the desired angle and then drilling a hole large enough to fit the encoder input shaft. The bracket was fastened to a support strut on VADR and the encoder was mounted. A gear was then mounted to the

encoder input shaft via a set screw. A separate gear was adhered to the drive shaft with J-B weld. Unfortunately, since the space around the shaft was limited, a larger gear could not be attached. So, no increase in resolution could be attained. A belt was then used to transfer the rotation of the shaft to the encoder. The encoder was then wired to the OOPIC. The program necessary to process the velocity information is detailed in the sensor section in the technical development portion of this thesis.

3.2.8 VADR Webcams

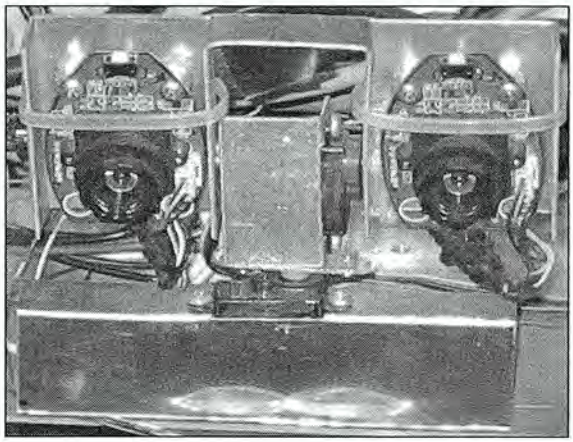


Figure 10. Webcam Setup

Figure 10 details the vision setup used on VADR. Webcams are just a fancy name given to cameras whose output can be hooked to a computer. There are so many different available webcams that it would be monotonous to detail the advantages of one type over another. For the purpose of this thesis, it is only necessary to know that the cameras continuously stream video to the VADR's computer via USB. The camera stand was created by another, and so will not be detailed here. It is only necessary to know that two servo motors, connected to the phidget motor control kit, give the cameras yaw and pitch maneuverability.

The two webcams were centered on the outer most portion of the front extension of VADR. The specifics for the Creative Labs WebCam NX Pros can be seen in Appendix A, page 57. A noteworthy design flaw was encountered after the cameras were installed. The original intent was to have stereo vision. Stereo vision is advantageous because with two cameras, depth calculations can be performed on the video. Unfortunately, the two cameras were powered by USB and streaming both cameras at the same time overloaded the USB server. To solve this dilemma, it was simply much cheaper to choose not to use one of the cameras.

3.2.9 VADR Power

Figure 11. Power Supply

All of the previously mentioned components require electricity, somewhere in the range of 5-12v DC. Initially, a 7 volt battery supplied power to the drive train, a 12 volt battery supplied power to the computer and phidgets, a 9.5 volt battery supplied power to the OOPIC, and a 5 volt battery supplied power to the sonar and velocity sensors. Any component that was not just listed is powered indirectly by one of the previously mentioned power supplies.

Two main problems occurred after the implementation of the 4 power supplies. First, batteries are not light weight solutions. The added battery weight caused the drive motor to

draw more current, thereby shortening the life of the drive train battery. This caused the operation life of VADR to be near the 15 minute mark before a recharge was necessary. Not nearly long enough to do anything practical. Second, utilizing 4 batteries became too expensive, both in the amount of time necessary to recharge and in the actual cost.

To solve this dilemma, a single power supply capable of powering all of the electronic components was mounted to the frame of VADR, above the computer. A mount was created by securing two angled aluminum pieces to the top of the frame, as can be seen in figure 11. When totaled, the added weight of the power supply and mount was actually less than that of the three power supplies that it replaced. The VADR power supply is capable of supplying a continuous voltage at 5 and 12 volts DC. The details of the power supply can be seen in Appendix A, page 58.

The only downside to using the continuous power supply was that it required a connection to a wall socket. A tether was constructed using a 30 foot extension cord. Even though this severely limited the range of VADR, the advantage gained by not recharging every 15 minutes justified the change. It should be noted that this is not a permanent change; it is still possible to drive VADR solely on battery power.

3.2.10 VADR Input Device

Figure 12. Phantom Device

A Phantom Omni Haptic Device (Phantom) is shown in figure 12, above. The phantom is a stationary input device capable of 6 degrees of freedom. It is connected to a control computer via a firewire cable. The pen-like user input device is called the stylus. The stylus has the ability to record user inputs as well as transmit forces back to the user. Those forces are input to the device from a control program running on a separate computer (a standard desktop PC). More detailed specifications for the phantom device can be seen in Appendix A, page 59.

CHAPTER 4. TECHNICAL DEVELOPMENT

4.1 Sensor Design

The OOPIC java program created to control the velocity and sonar sensors can be seen in appendix B, page 60. It is fairly easy to follow. First, 5 sonar objects, 1 encoder object, and 1 serial object are created. Then, the OOPIC is told what pins activate the appropriate IO on the sensors. Next, the OOPIC enters a loop where the sonar and velocity data are gathered. And finally, the resultant values are sent out by the serial object. The loop was set to repeat every half second. The raw sensor data from the OOPIC will eventually reach the control program where it will be converted to useable data.

The sensor data is output from the OOPIC as an unsigned 2 byte number. Bytes are simply binary numbers that computers use to communicate data with. For this application, it is necessary for the control program to convert the unsigned byte into a signed byte. The signed property resulted in the sensors ability to convey the difference between forward and reverse. A two byte number is capable of having the value of 0-65535. In one direction, the encoder starts at 0 and adds up. In the opposite direction, the encoder starts at 65535 and counts down. The count is converted to a signed number by dividing the max possible value by two. This gives a range to check whether the number is positive or negative. For this application, the 0 – 32767 range of numbers related to a reverse direction of VADR. The equation below details the conversion to a negative number.

Eq. 1) Reverse Count = - (Count)

For this application, the 65535-32768 range of numbers related to a positive direction of VADR. The equation below details the conversion to a positive number.

Eq. 2) Positive Count = ABS(Count – 65535)

The count value was also output from the OOPIC in feet divided into 64 increments. The control program simply carries out the necessary unit conversion, detailed below, to get a reading of the distance in feet.

Eq. 3) Distance (ft) = Sensor Data (ft/div) / 64 (div)

Next, the velocity data is output from the OOPIC as a count. The program only takes data for half a second, but the convention is to have data for a full second. Equation 4 is performed to handle the appropriate conversion.

Eq. 4) Velocity (count/s) = Sensor Data(count) * 2 / 1 (s)

Next, it was necessary to convert the count of the encoder into revolutions of the tire. This was achieved by performing the calculation detailed below. The conversion number was determined by experimentally rotating the tire through a full rotation and noting the amount of counts that the encoder recorded.

Eq. 5) Velocity (rev/s) = Velocity(count/s) / 43(count/rev)

Future calculations detailed in the kinematics of vehicle slippage section necessitate a conversion of rev/s to feet/s. The calculation is detailed below. The necessary conversion number was taken from a measurement of the tire circumference.

Eq. 6) Velocity(ft/s) = Velocity(rev/s) * 1.27(ft/rev)

4.2 Communication Design

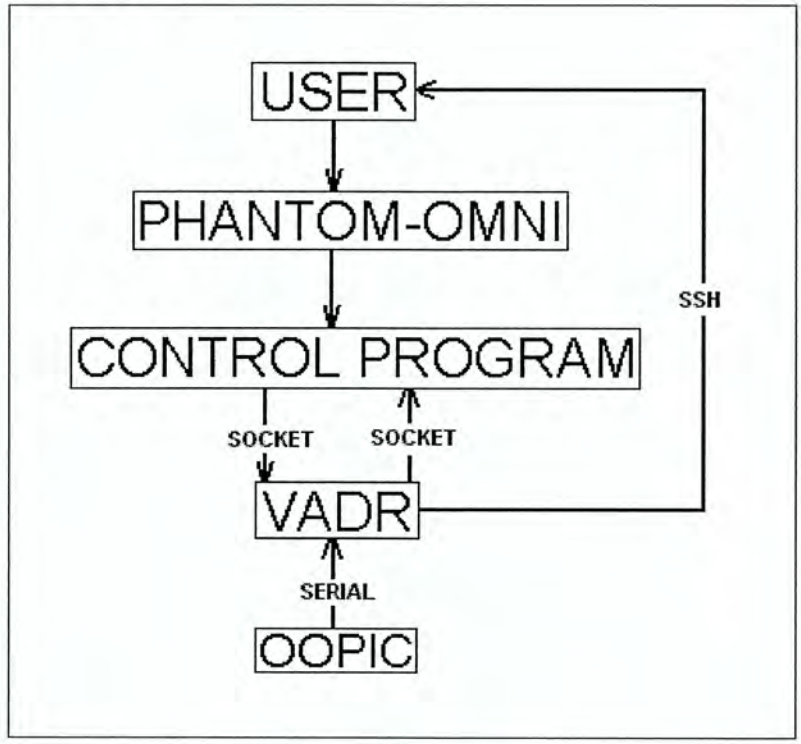


Figure 13. Communication Flow

Figure 13 above, gives a visual representation of how the communication flows throughout the interface. It is fairly easy to follow. Any connection that did not get mentioned in figure 13 did not get mentioned because the communication was handled internally and not necessary to develop in this thesis. The three noteworthy communication connections of figure 13 are detailed in the next few sections.

4.2.1 Serial

Serial communication is a well known standard for communicating data to and from a computer and another electronic device. The serial connection between two devices can be simplified into a single wire connection. Data is sent or received over the wire one bit at a time. Serial communication details a handshake that takes place between the two devices.

The handshake is set up so that each device may know whether it should be sending or receiving data and whether or not the device it is communicating to is ready to send or receive data. Also, a protocol to specify how the data is handled needs to be set up in the form of start and stop designations. The next few paragraphs detail the previously mentioned setup for serial communication between the OOPIC and VADR's computer.

Conveniently, the OOPIC has an object to handle serial communication. The java program that details the control of the OOPIC can be seen in Appendix B, page 60. In this program, a serial object is first created. Then, the object is set to operate at a rate of 9600-baud and asynchronously. The baud setting is a designation for the number of discrete signals sent by the device in one second. The particular baud rate of 9600 was chosen because both devices are capable of that rate of data transfer. The asynchronous setting handles the start and stop bit designation for the serial port. The setting is asynchronous due to the unpredictable rate at which the data for sending is gathered. Finally, a loop is entered where the data to be sent is assigned to the appropriate serial object value and sent out the serial port.

Also, conveniently Java supports a Communications API in which a serial object exists. Unfortunately, the standard version of the object must be modified for the specific application of VADR and this is not as easily done as in the OOPIC object case. Fortunately the object-oriented aspect of java allows for the type of manipulations required to get an object that VADR can use. The method used to achieve the useable object is buried in the code in the program detailed in Appendix B, page 62. A simplified explanation of the code however, can be seen below.

First, a serial object is set up to handle the same hand shaking and data protocol as the OOPIC object so that the computer can talk to the OOPIC. Next, since the serial object was designed to handle events, an event listener was initialized and set to let the serial object know when data is available. Then, the serial object created a buffer to read in the data. The buffer was then parsed into the correct sonar or velocity value. The data was then sent to a socket for internet transfer, as detailed by the next section.

4.2.2 Socket

Internet communication is most commonly handled in the form of sockets. A socket is a software endpoint that establishes two-way communication between a server program and a client program. There are two main protocols used to employ sockets for internet communication, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP and UDP tell the computer how to send and receive data, also known as packets. The TCP protocol is set up to handle guaranteed delivery, no matter how long it takes. Packets are numbered so that specific ordering can be maintained. Packets are also re-transmittable in the event that they are lost during transmission. The UDP protocol is not set up to handle guaranteed delivery, but sends packets very fast. Packets may rarely arrive out of order or never arrive at all.

From a quality stand point, the TCP is clearly the top choice for this application. From a speed stand point, the UDP is clearly the top choice for this application. And, since the speed in which commands can be relayed between the two computers is more important for this application, UDP was chosen and implemented. The next few paragraphs detail the UDP setup for internet communication between VADR and the control computer.

Conveniently, Java handles UDP socket communication in the form of a datagram socket and datagram packet object. However, as in the serial object case, the datagram objects need to be converted for the VADR specific application. The object-oriented approach taken to transform the standard datagram objects is buried in the code in the programs shown in Appendix B, pages 65 and 66. A simplified explanation of the code is still necessary to convey understanding.

A UDP server was constructed using both a standard datagram socket and packet object. The purpose of this object was to receive motor commands from the control program. First, the socket object was constructed with a port parameter that tells the server which port to listen for packets. That is all that was required to convert the object into a server. Then, a receive function was defined to handle incoming packets of data by assigning them to the local packet object. Next, the packet data is parsed into the appropriate form (motor number and position) and sent to a motor control object. All that was left to get data from the control program was to call the receive function of the server object.

A UDP client was constructed using both a standard datagram socket and packet object. The purpose of this object was to send sonar and velocity data to the control program. First, the socket was constructed with an internet protocol (IP) address and a port. That is all that was required to convert the object into a client. Then, a send function was defined to handle the transmittal of packets. To populate the data, the current serial port data was polled and then concatenated into one large byte that was set to the packet object. All that was left to send data to the control program was to call the send function of the client object.

C++ also includes a generalized socket object as a standard C++ object. However, a C++ socket class was already manipulated into a useable format by another individual for a previous application. The individual's consent to use the C++ client and server for the control program was obtained. The C++ source code can be seen in the header files seen in Appendix B, page 69. A simplified explanation of the implementation of the code is still necessary to convey understanding.

A server was created with the appropriate port to bind to. The data the client needed to receive was then added to a list. In the header files, the send function was already created. In the receive function, the received packet is parsed into the appropriate list parameters. All that was left to get the velocity and sonar data from VADR was to call the receive function of the server object.

A client was created with an IP address and a port. The data that the client needed to send was added to a list. In the header files, the send function was already created. In this send function, the list data is assigned to the packet data. All that was left to send motor command data to VADR was to call the send function of the client object.

4.2.3 SSH

The conventional way to send video over to a remote destination is by streaming. However, streaming video over the internet has the tendency to take an unacceptably long time. The length of time is a direct result of the video file format being too large. Several specialized algorithms have been developed to speed up the streaming process. They all follow the generic process detailed below. First, the raw video stream frames are broken down into a more compact video format (MPEG, JPEG, etc) using a codec. The individual

files are then concatenated back into a stream using a muxer. The muxed video stream is then sent over the internet to the specified destination. The target computer then demuxes the video, recodes it, and displays it to the user.

There are several different programs available that stream video over the internet. Unfortunately, none of the algorithms found could do better than a 2 second lag. Simply unacceptable, the lag offset the visual feedback from the motor commands enough to make controlling VADR reliably impossible. Despite this setback, a protocol that could result in near real-time video transmission was discovered.

Secure Shell (SSH) is an internet protocol to establish a secure connection between a remote and local computer. There is nothing special about this specific protocol that makes it faster than the previously described streaming method. It even uses the TCP, slower, internet communication protocol. SSH works by taking advantage of an algorithm instantiated by the X Windowing System (X Windows), as detailed in the next paragraph.

X Windows is the standard toolkit and protocol used to create a graphical user interface (GUI) in the Linux operating system. X Windows uses an algorithm called an X Session to send video from a local computer to a host computer. The reasons why this algorithm appears to operate a significant amount better than any standard streaming method is probably another thesis subject in and of itself and so, is out of the scope of discussion in this thesis. All that is necessary to know is that nearly real-time video streaming was achieved. Also, there is no code to see for implementation of SSH. It is a stand alone program that is set to run in the background of VADR.

4.3 User Interface Design

The user interface can be broken up into three main sections, a graphic interface, a haptic interface and a visual interface. The graphic interface was designed to convey a good sense of how the user input will affect VADR. To get the graphical interface, portions of three different graphics based programs were exploited. Blender, a computer aided design (CAD) package was necessary to create graphical models of parts of VADR. Open Scene Graph (Open SG), a scene graph program was necessary so that motion could be given to the parts. Open Graphics Library (Open GL), a computer graphics program was necessary to manipulate the computer to get the optimal graphics settings for the interface and as a base layer to tie these programs together.

The haptic interface was designed to let the user control the speed and direction of VADR through the phantom device. The haptic interface was also designed to exploit the force feedback aspect of the phantom so that the user could also get a feel for how their input would affect VADR. To get the haptic interface, portions of two programs were exploited. Haptic library application program interface (HLAPI), a haptic programming library was necessary to program forces into the phantom. And the same Open GL program as mentioned in the previous paragraph, was necessary as a base layer to tie the phantom to the graphic interface.

The visual interface was designed to let the user see the video from the webcam mounted to the front of VADR. To get the video from the webcam to the user an intermediate program called CamStream was utilized. The CamStream program was necessary to process the raw video data and display it in a viewable format.

4.3.1 Graphic Interface

4.3.1.1 Blender

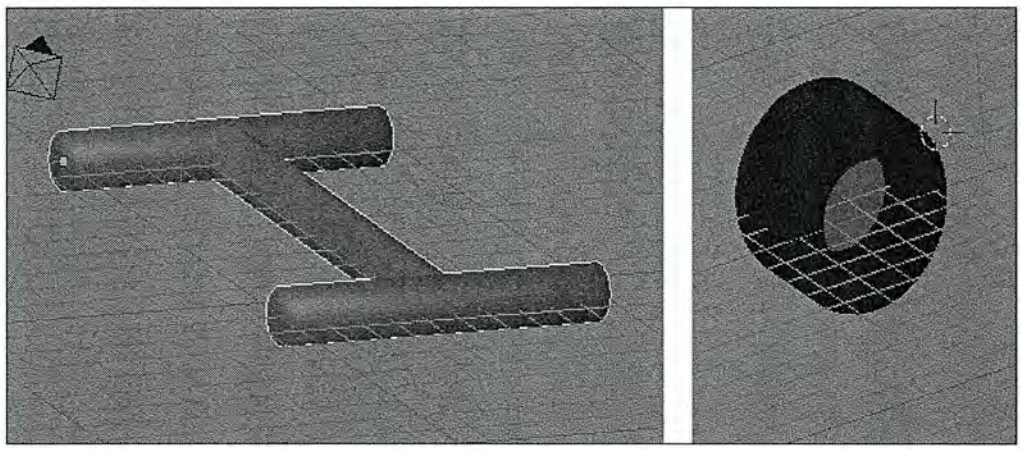


Figure 14. Blender Models

VADR is a complex mobile platform with many parts that could be modeled and of use to the user. However, the only part of VADR that is necessary to give the user a decent visual representation for controlling the movement of the vehicle is the footprint. The footprint can be simply represented by four wheels and a frame. Measurements were taken from VADR and entered as parameters for the Blender geometry. The created models are shown in figure 14, above. The models were then exported into the (XML) file format for porting into Open SG.

4.3.1.2 Open SG

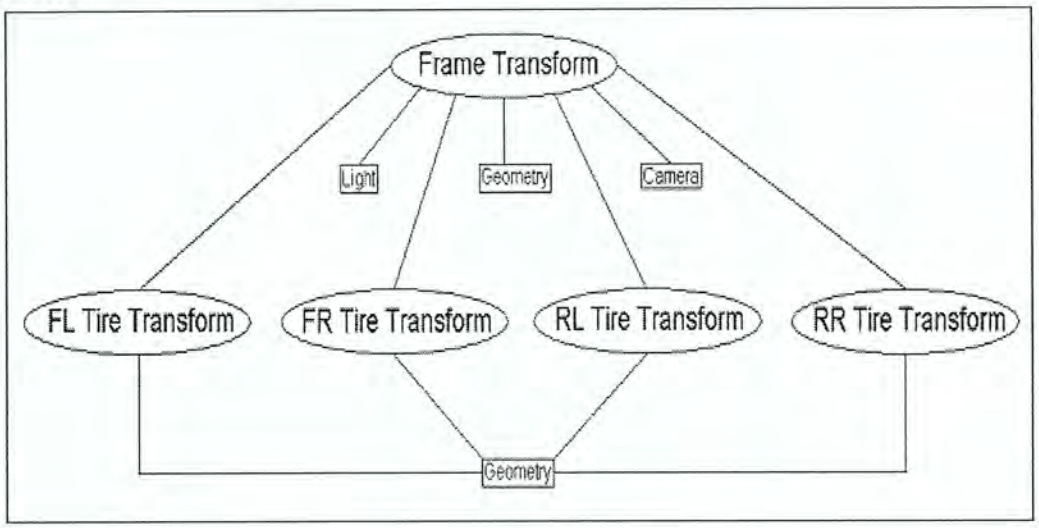


Figure 15. Open SG Scene Graph

The generalized scene graph depicted in figure 15 was designed using the Open SG library. The C++ source code used to generate the actual scene graph is programmatically complex and boring, but may be seen in Appendix B, page 76.

As mentioned earlier, the scene graph allows for animation of the blender geometries. The hierarchy of the scene graph is relatively simple to follow. The frame transform node was set as the root node or parent node. Any translation, rotation, or scaling would not only affect that node, but would also affect the children of that node. The four tire transform nodes were then set as child nodes of the frame transform.

Each of the children could then be translated to correctly match the frame positions and also rotated about an axis perpendicular to the frame. To match the Ackerman steering requirements, only the front two wheels were allowed to rotate about the axis. A trivial mapping connected the user input to the appropriate wheel angle rotations.

The square nodes in figure 15 are still children of the specified parent nodes and follow the rules as described above, but it was necessary to visually represent them as less important. Each of the geometry nodes are pointers to their respective blender models so that Open SG goes to the appropriate location when it is necessary to draw. The final two nodes in the scene graph, light and camera, are merely cosmetic nodes necessary for Open GL rendering.

4.3.1.3 Open GL

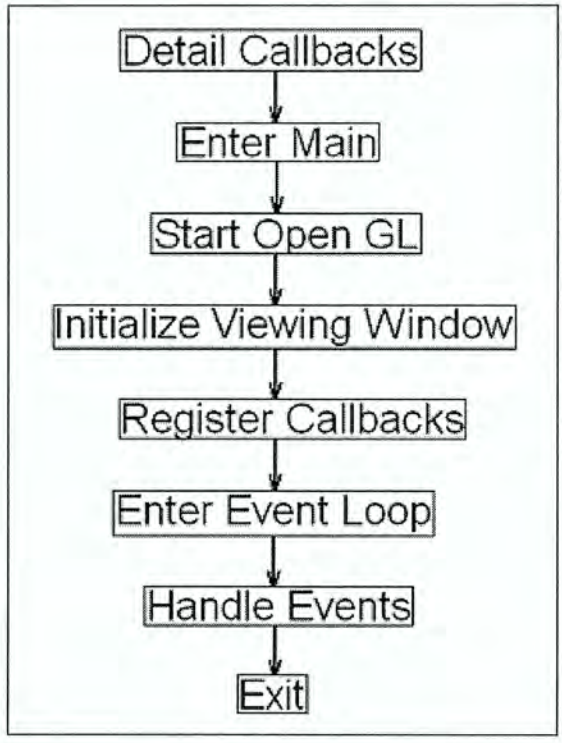


Figure 16. Open GL Program

A generalized Open GL program can be seen in figure 15. The C++ source code used to generate the actual program is, like all the previously mentioned code, programmatically complex and boring, but may be viewed in Appendix B, page 76. However, a simplified overview is necessary.

Open GL is an event listening program that once activated, builds events into a queue and reacts to them in a first come first serve manner. The hierarchy of the Open GL program is set by defining what Open GL should do upon certain events. These functions are known as callback functions. After all the callbacks are defined, the main program is entered. The main program is set up so that once Open GL is started, a graphic display window is initialized and placed on the screen, then the event callbacks that Open GL is required to handle are registered, and finally an event listening loop is entered until the program exits.

4.3.2 Haptic Interface

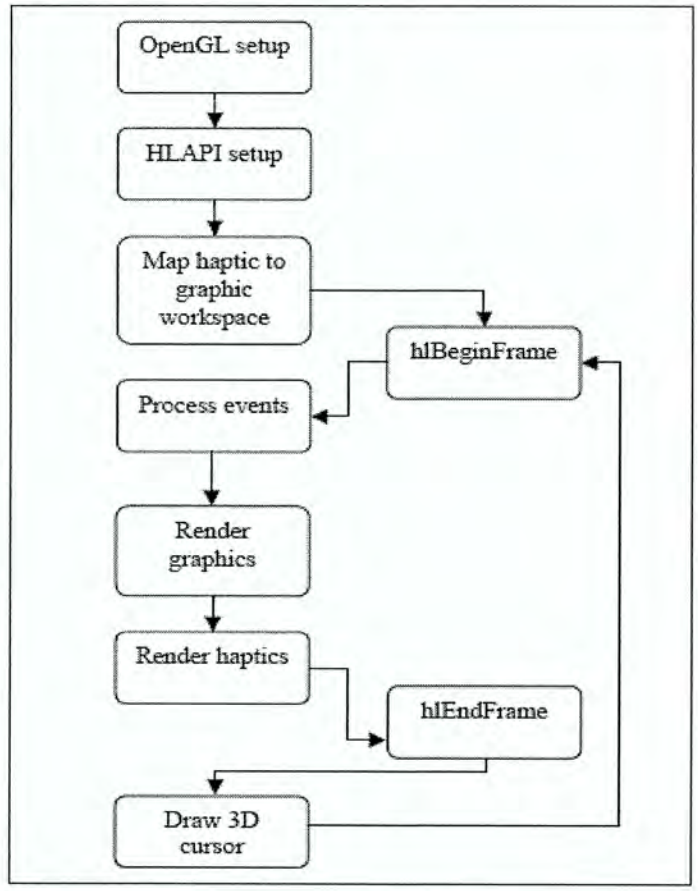


Figure 17. Typical Phantom Program [21]

A generalized haptic program can be seen in figure 17. It was used to program the hardware device so that forces may be sent to the user from the interface. The program is set up to run along with the previously mentioned Open GL program. The C++ source code used to generate the actual phantom program is programmatically complex and boring, but may be seen in Appendix B, page 76. However, a simplified overview is necessary.

In order to make the phantom visible to the graphics, Open GL must be made aware of its existence. To do this, a device must be initialized, a haptic context must be created, and the context must be set to current. Once that is taken care of, then a mapping must be created so that the when the device is manipulated by a user, it can interact within the existing Open GL program. To do this it is necessary to make two function calls, one to handle the physical dimensions of the phantom device and one to establish how those dimensions will be oriented with respect to the world coordinates of the Open GL program.

Next, a haptic frame is started. This is where all the haptics actually get programmed. First, any pertinent events are handled via a similar callback method as described in the Open GL section. For this application, a button-pressed event is of particular interest.

For safety reasons, it would be useful to add an on/off switch functionality that would start and stop sending commands to VADR. Also, at the program start up, the physical device position does not line up with the center of the workspace. In order to correct this, the device was programmed to move itself to the desired starting position. To achieve this, some necessary haptic function calls were made to disallow user input. At the same time that the user is ready to command VADR, it would be fundamental for the phantom to revert back to a state where it will react to user input. So the button-pressed event handles two commands, on/off and motion reset.

After handling the event, any Open GL geometry is rendered. For this application, that means the Open SG scene graph is traversed and drawn. Once that happens, control is returned to the phantom to handle the actual haptics. For this application, it was necessary to create three haptic forces, a bounding volume, a spring force, and a damping force.

First, a bounding volume was necessary to constrain the device to react only within a specified volume of the workspace. A convenient aspect of the haptic library is that it uses the same calls to render haptic shapes as the Open GL program uses to render graphics. It was noticed that since the device has three independent axes, that a cube would make an appropriate bounding volume. The desired dimensions of a bounding cube were chosen and transferred into Open GL draw commands contained in a C++ object. The C++ source code required to do this can be seen in Appendix B, page 73. The phantom calculates all the necessary forces internally and sends them to the appropriate axis of the device.

Second, a spring force was necessary to give a tactile sensation of the user controlled speed. The spring force follows the well known math model shown below.

Eq 7.) $F_{spring} = K_{spring} * (X_{anchor} - X_{current})$

The Kspring value was determined by a simple trial and error method until the desired effect was achieved. Unlike the haptic shape, the spring force gave write access to the phantom forces. Since only the speed input needs to feel the spring effect, only forces in the Y axis of the device were calculated and sent to the user.

Third, a damping force was necessary to give a tactile sensation of resistance to the user controlled steering angle. Admittedly, the addition of this force was not necessary, it only served to smooth the steering input given by the user. By providing a resistance to

motion, the phantom can help eliminate any over correction experienced by quick movements. The damping force also follows a well known mathematical model.

Eq. 8) Fdamp = Kdamp * Velocity Stylus

The Kdamp was also determined by a trial and error method. Much like the spring force, the damping force gave write access to the phantom forces. Since only the angle input needs to feel the damping effect, only forces in the X axis of the device were calculated and sent to the user.

After sending out the forces, the haptic frame is ended. All that is left is to give the user a visual representation of where the device is in the graphical world. To do this, the position of the device is polled and then some Open GL geometry is drawn at that location in the graphic world. This cycle repeats a thousand times a second so that the force feeling is as realistic as possible.

4.3.3 Visual Interface

Simply plugging a webcam into the USB port is not enough to display images on a computer. A capture program is necessary to read in the raw video and open a window to display the video. Unlike streaming, a simple display window does not need to convert the raw video format. There are many different video capture programs and comparing them all would be mundane. A brute force method of trial and error was used to determine that Camstream is an adequate enough video capture program to display the video from VADR's webcam. Like SSH, Camstream is a stand alone program that is fully developed by others, and so there is no code to detail. Camstream is simply told to run in the background of VADR upon start up.

4.4 Device Input Mapping

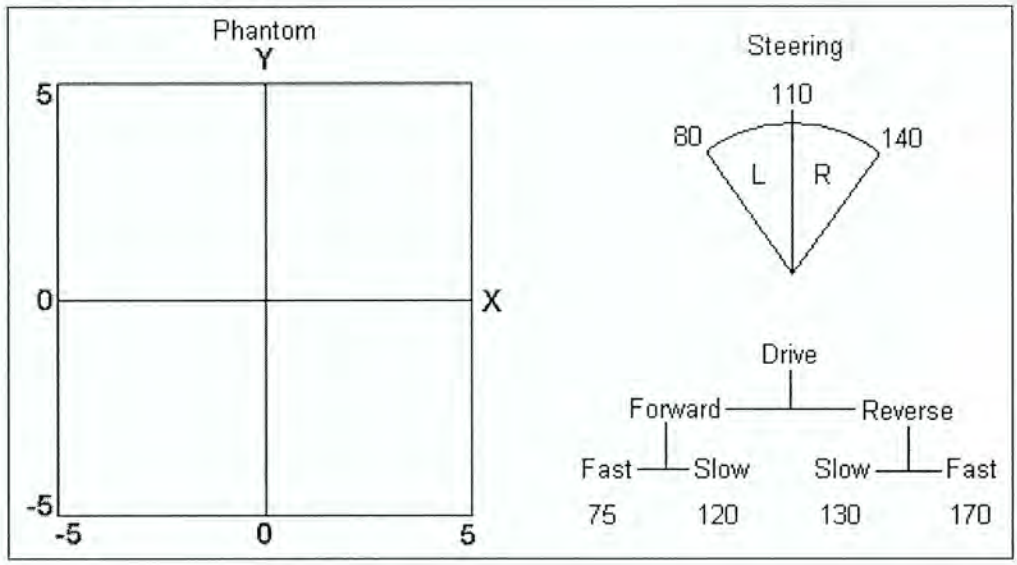


Figure 18. Phantom to Motor Map

There are two input variables that the user needs to be able to input to control VADR. The steering motor is a servo and needs to be commanded to go to a certain position. The drive motor is a direct current (dc) motor and needs to be commanded to go to a certain speed. The phantom is a device that has 3 degrees of linear freedom (X, Y, and Z). As seen in figure 4 above, the X axis was chosen to map to the steering motor position, the Y axis was chosen to map to the drive motor speed, and the Z axis input was simply ignored. To achieve the correct mapping, the motors were tested to find the range of practical operating positions and speeds. These positions and speeds are displayed on the right hand side of figure 18.

To get the steering and drive coordinates ready to map, it was necessary to convert the 10 unit coordinate range into a decimal number between 0 and 1 or 0 and -1 as evidenced by the equation below.

Eq. 9) Linear Number = Steering Coordinate / 5.0

Conveniently, the change in position from the 0 degree turn position to the maximum and minimum turn value is the same, 30. So the map should start at 110, essentially zero, and be incremented or decremented by the change in position times the linear Number. This is detailed by equation 10.

Eq. 10) Steering Position = 110 + ((steer coord / 5) * 30)

Unfortunately, the drive speed mapping was not as perfectly linear as the steering. The approach is still the same, but there are two separate equations to handle a dead band in the DC motor forward and reverse commands. The two equations are shown in equations 11 and 12.

Eq. 11) Forward Drive = 120 - ((drive coord / 5) * 45)

Eq. 12) Reverse Drive = 130 + (abs(drive coord / 5) *40)

4.5 Kinematics of Vehicle Slip

The only problem with using an RC drive motor to propel the mobile platform is that typical RC motors are designed to go fast. This allows the mobile platform to accelerate to speeds that would cause it to slide as it negotiates turns. The most common way to guard against this would be to limit, in software, the speed that the motor can attain. It should be noted that the slip speed limitation reduces the overall top speed attainable by VADR. A feature that is less than desirable.

The phantom allows for a way to solve this dilemma. In order to keep the top speed at an acceptable rate and at the same time safeguard against slipping, forces could be administered by the phantom. As the user directs the phantom to into a turn, the phantom could take the current velocity reading, predict whether or nor a slippage will occur, and then

supply forces that would control VADR to a safer orientation. In order to perform this proposed manipulation, a predictive model must be developed. The following equations detail such a model.

It can be noted in any vehicle dynamics text that a four wheeled vehicle may be simplified into a two wheeled model, also known as a bicycle model [6]. A bicycle model for VADR can be seen in figure 19, below.

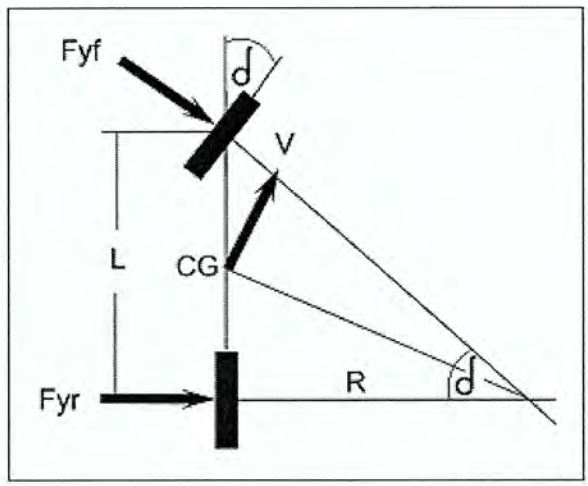


Figure 19. VADR Bicycle Model

From the figure, it can be shown that a simple geometry relationship will yield the Ackerman angle [6].

Eq. 13) Tan(Delta) = L / R

After applying a small angle simplification (Tan (Delta) = Delta) and solving for the radius of curvature, the equation above reduces to the equation below.

Eq. 14) R = L / Delta

Also, if the moments in figure 19 are summed and set equal to zero, as in a static relationship, then equation 15 can be attained.

Eq. 15) Sum of Moments = 0 = Fyf * a - Fyr * b

Solving for Fyf yields the following equation.

Eq. 16) Fyf * a = Fyr * b

Since, in the case of VADR, a roughly equals b the above equation can be reduced to the next equation.

Eq. 17) Fyf = Fyr

Upon inspection of equation 17, it can be ascertained that the front and rear of VADR will slip at the same time. So from this point on, the front and rear designation will be dropped from the Fy's so that only one Fy will be utilized.

The lateral acceleration can be determined by solving for the centripetal acceleration of VADR about the turn.

Eq. 18) Ay = v^2 / R

Also, by using Newton's Second Law and summing forces in the Y direction, the following equation can be developed.

Eq. 19) 2 * Fy = M * Ay

Substituting equation 18 into equation 19 and solving for Fy results in the equation below.

Eq. 20) Fy = (M * V^2) / (2*R)

A final substitution of equation 14 for R can be made to get equation 21.

Eq. 21) Fy = (M*V^2*Delta)/(2*L)

From the study of physics and statics, it can be realized that in order for the lateral force to reach a state of slippage, the coefficient of friction between the tire rubber and the surface of travel must be overcome. The equation to model this can be seen below.

Eq. 22) Slip if Mu * N > Fy

After substituting equation 21 for Fy, the final equation that will be used to determine if VADR will slip while cornering can be seen below.

Eq. 23) Slip if Mu * N > (M*V^2*Delta)/(2*L)

It should be noted that all the variables of the equation are known vehicle parameters, constants, or user inputs as seen in table 1.

Parameter	Value
L	.875 (ft)
M	.559 (LBf)
Mu	.80 Unitless
N	18 (LBf)
Delta	User Input (rad)
Velocity	User Input (ft / s)

Table 1. Vehicle Parameters

To translate the slip information to the user, a new force was added, both to lessen the degree of the turn and slow down VADR. In this application, a constant force was added to the device, both in the X and Y directions. The value for the constant force was determined from a trial and error method. The constant force was chosen over the spring force because a spring force could still be overridden by the user; effectively negating the safety of this safety feature. The code to implement this constant force can be seen in Appendix B, page 76. Since the code is very similar to the previously mentioned force and damping effects, the method to implement it will not be detailed.

CHAPTER 5. RESULTS

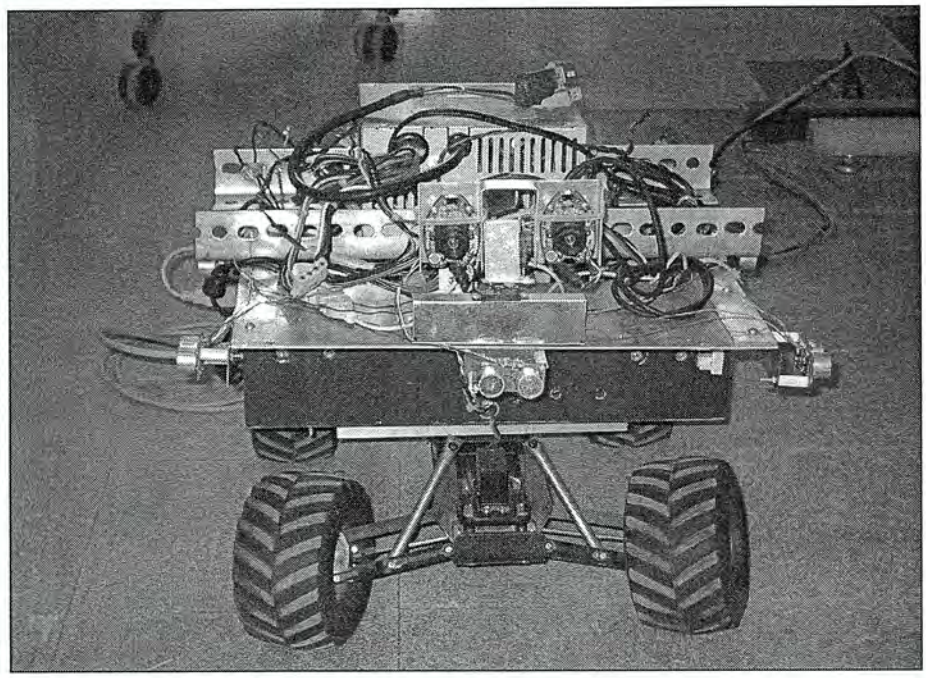


Figure 20. VADR

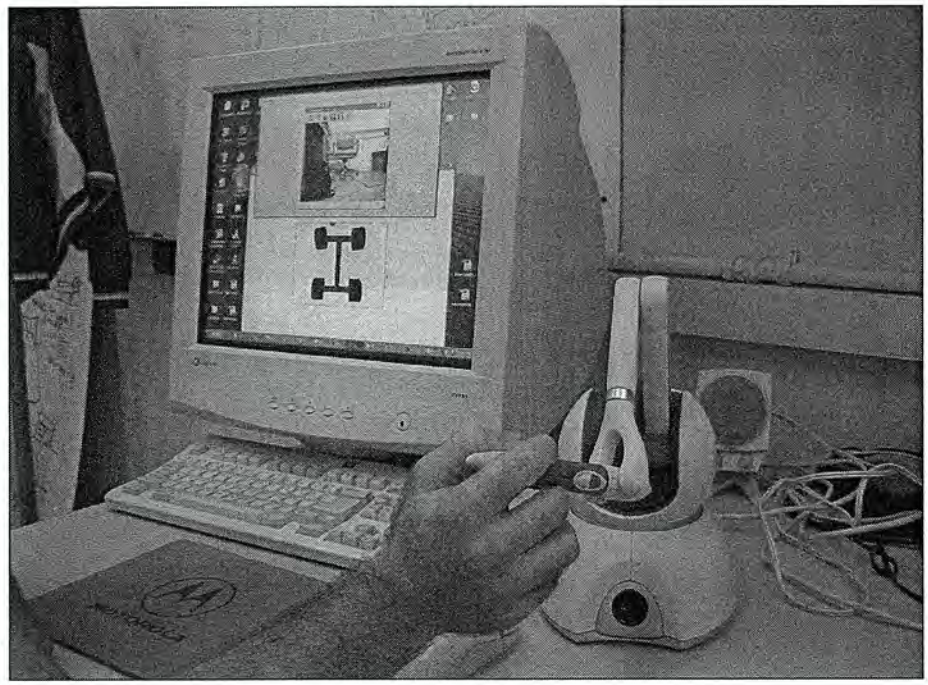


Figure 21. Phantom Interface

CHAPTER 6. CONCLUSION

Figure 20 shows the fully constructed mobile platform VADR. The platform is capable of receiving commands remotely from a user. Not only that, but VADR is also capable of sending the sensor data back to the user. After constant use, it can be seen that there is an occasional errant motor command, but it is rare enough so as to not hinder the overall outcome. It is the claim of this thesis that there was very little lag associated with the mostly reliable use of the UDP sockets. This leads to the belief that UDP sockets were in fact the correct choice for internet communication.

There was an original claim that near real time video feedback could be achieved. After constant use of the system, this claim could not be reliably repeated. While near real time video was often witnessed, it was not always present and so should be noted. The fluctuations were believed to lie with a limited band width in the internet that was being employed. It should be stated that while it was hoped that some work around to internet lag could be found, it was never an expected goal.

The interface design, as shown in figure 21, appears to have been successful also. The ability to not overload the visual senses was a crucial factor in creating an interface that is simple, yet effective. Also, the added haptic aspect of the interface was simple enough so as not to draw too much attention away from the visual feedback display.

The implementation of the force feedback did appear to improve the drive-ability of VADR. A certain realism was added to the interface that, without the haptics, could not have otherwise been achieved. The forces appeared to be just strong enough to cause the desired

effects in the user responses. It is hoped that all future interface designs will not only employ visual feedback, but tactile feedback as well.

CHAPTER 7. FUTURE WORK

A positive side to working with mobile robotics is that there is always something new to learn or try. The field is relatively new in development and there are few set standards. There are so many possible applications that could be investigated now that a successful mobile base has been constructed. But those are for future students to struggle with. The rest of this section details just a few ideas to further this particular haptic interface.

The sensor information that the user receives only gives a limited vision of the environment to the user. Since there is only the one camera view pointing out in front of the vehicle, it is difficult to navigate between close objects. To fix this, a sonar measurement could be read in from each side of VADR and used to calculate the center between the two obstacles that VADR is attempting to pass between. Then the user could press a button on the phantom stylus and VADR would drive to the center automatically. So as not to take away from user control, this would only be implemented for close maneuvering.

It was noticed during some trial runs that it might be more helpful if the haptic device were able to let the user feel obstacles. The force, whatever the type, could get stronger as the vehicle neared the object. Due to the current device set up, the obstacle force would have to either slow the vehicle down or turn the vehicle onto a non-collision path. To enact this functionality, a better sensor design would probably have to be implemented. Note that the forces would still allow the user to control the vehicle and not take control away, as in the case of partial autonomy.

The existing Open SG scene graph could easily be manipulated to handle a translation that would allow the graphical representation of VADR to move around in a virtual world. An advantage is gained in the virtual world because the user has the ability to see much more of the robots surrounding environment. This type of application is somewhat limiting however, due to the fact that the parameters needed to recreate the environment need to be known long enough in advance so that an adequate model can be created. However, this application could become more dynamic and useable if a representation of the environment could be picked up and modeled on the fly, say from a webcam or two.

APPENDIX A. HARDWARE SPECIFICATIONS

Computer

	M10000 Specifications
Processor	VIA 1 GHz C3 with fan
Chipset	VIA CLE266 north bridge VIA VT8235 south bridge
System Memory	1 DDR266 DIMM socket Up to 1GB memory size
VGA	Integrated VIA UniChrome AGP graphics with MPEG-2 Accelerator
Expansion Slots	1 PCI
Onboard IDE	2 Ultra DMA 133 (40 pin)
Onboard USB	4 USB 2.0
Onboard LAN	VIA VT6103 10/100
Onboard Audio	VIA VT1616 6 channel AC'97 Codec
Onboard TV Out	VIA VT1622 TV out
Onboard 1394	VIA VT6307S IEEE 1394 Firewire
Back Panel I/O	1 PS2 mouse port 1 PS2 keyboard port 1 Parallel port 1 LAN port 1 Serial port 2 USB 2.0 ports 1 VGA port 1 TV-out port 1 S-Video port 3 Audio jacks: line-out, line-in, and mic-in
Onboard I/O Connectors	2 IDE 40 pin connectors 1 USB 2.0 connector for 2 USB ports 2 IEEE 1394 connectors 1 Front panel audio connector 1 CD Audio-in connector 1 FIR connector 1 CIR connector (Switchable for KB/MS) 1 Wake-on-LAN connector 1 Fan connector

	1 SMBUS connector 1 Connector for LVDS module 1 Serial port connector ATX Power Connector
BIOS	Award BIOS 2/4Mbit flash memory
System Monitoring & Management	CPU voltage monitoring, Wake-on-LAN, Keyboard-Power-on, Timer-Power-on- System power management, AC power failure recovery
Operating Temperature	0 ~ 50°C
Operating Humidity	0% ~ 93% (relative humidity; non-condensing)
Form Factor	Mini-ITX (17cm x 17cm)
Includes	ATA 133 flat cable (40 wire, 80 conductor, 3 connectors) FDD cable Back plate Installation CD Quick installation guide Dual USB 2.0/Firewire connector

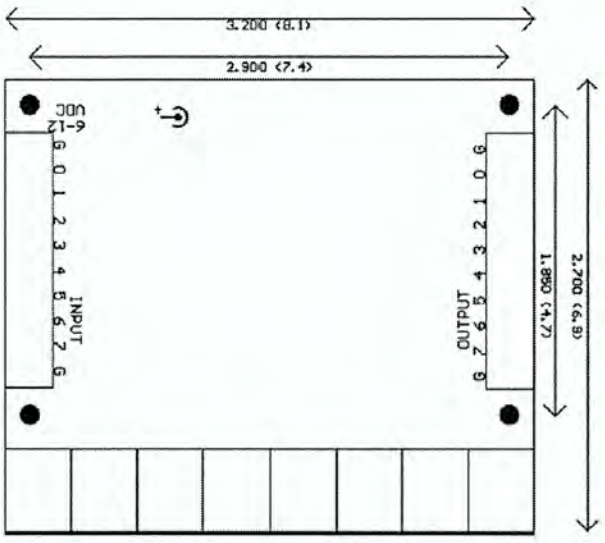
Phidget Interface Kit

Hardware Description

The PhidgetInterfaceKits are the most versatile of the Phidgets currently available. This versatility comes at the price of some complexity. In addition to the USB input port, the PhidgetInterfaceKit 8/8/8 has:

- 8 Analog Inputs,
- 8 Digital Inputs,
- 8 Digital Outputs,
- 2 USB output ports.

An external power supply is required only to operate the USB output ports.



Device Specification

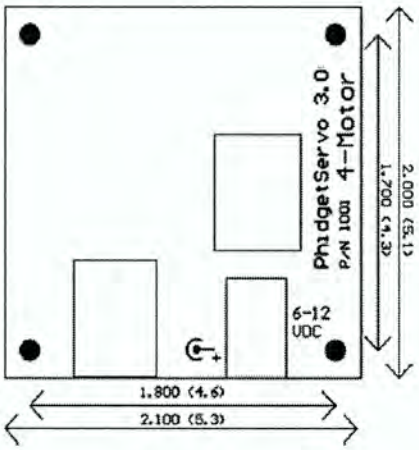
Digital Output Resistance	300 ohm
Digital Output Update Rate	Approx. 125 Hz
Digital Input Update Rate	125 Hz
Analog Input Update Rate	65 Hz
USB Current Consumption	Max. 500 mA
External Power Supply Voltage	6 to 12 V DC
External Power Supply Current	Max. 1000 mA
External Power Supply Rating	Max. 6 to 12 VA

Phidget Motor Controller

Hardware Description

The PhidgetServo 4-Motor controls up to four standard hobby servo motors. It is designed to work with various types of servo motor. The ones we use are relatively inexpensive hobbyist RC servo motors that are commonly available. These include:

- Futaba FP-S-148 Servo (available from HVWTech and Tower Hobbies).
- Tower Hobbies TS-53 (available from Tower Hobbies).



Other hobbyist servo motors are likely to work as long as they take 5 V and use a Pulse Code Modulation (PCM) scheme, but care must be taken that the motor does not operate past its boundaries and overheat.

The labeling on the PhidgetServo board is B – Ground, R – Power (+ 5 V), and W – Signal. The signal wire from the servo motor can be white or yellow depending on the manufacturer. An external power supply is required only if you attach a servo motor to position 1, 2, or 3.

Device Specification

Position Update Rate	Approx. 30 Hz
USB Current Consumption	Max. 500 mA
External Power Supply Voltage	6 to 12 V DC
External Power Supply Current	1500 mA (Min. 500 mA per servo)

OOPIC

Prg Connector.

Connects to the PC's Printer Port. (5 Pins)

Power Connector.

Connects to any power supply of 6-15 Volts DC. (2 Pins)

I/O Connector.

Provides connection to all I/O. (56 Pins)

Memory Socket.

Socket for program EEPROM. (8 Pins)

Network Connector.

Connector for I2C network cables. (5 Pins)

Dual H-Bridge Connector.

Connector for Dual DC Motor H-Bridge. (8-Pins)

LCD Connector.

Connector for Serial LCD (3-Pins)

RS232 Connector.

Connector for RS232 Serial Port

Optional Power Connector.

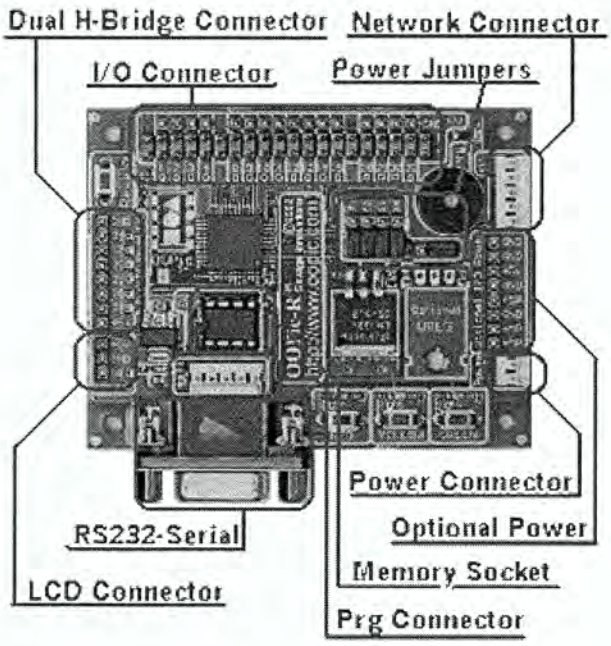
Connector for Optional Power configurations (8-Pins)

Power Jumpers

Jumpers for for Optional Power configurations (15-Pins)

Mechanical layout.

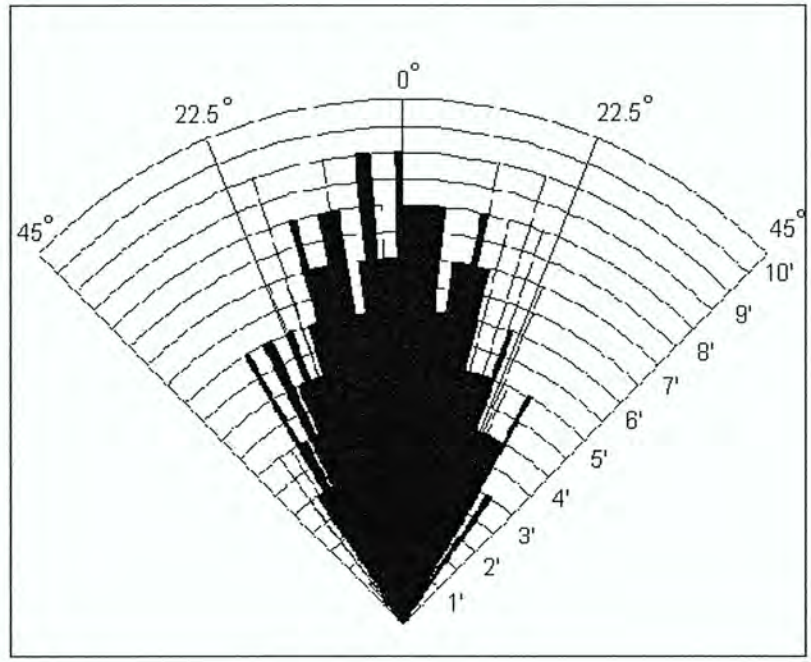
2 inches X 3-1/2 inches x 5/8 inch.



Sonar Range Finder

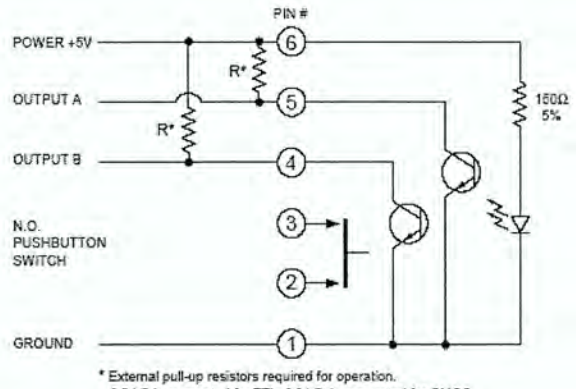
Beam Pattern	see graph
Voltage	5v
Current	30mA Typ. 50mA Max
Frequency	40KHz
Maximum Range	3 m
Minimum Range	3 cm
Sensitivity	Detect a 3cm diameter stick at > 2 m
Input Trigger	10uS Min. TTL level pulse
Echo Pulse	Positive TTL level signal, width proportional to range.
Weight	0.4 oz.
Size	1.75" w x 0.625" h x 0.5" d

Beam Pattern



Encoder

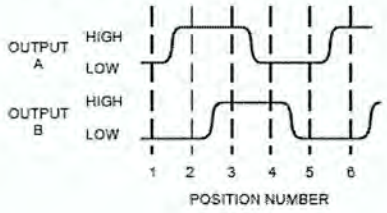
CIRCUITRY, TRUTH TABLE, AND WAVEFORM: Standard Quadrature 2-Bit Code



* External pull-up resistors required for operation.
8.2 kΩ is suggested for TTL; 3.3 kΩ is suggested for CMOS.

Clockwise Rotation		
Position	Output A	Output B
1		
2	•	
3	•	•
4		•

• Indicates logic high, blank indicates logic low. Code repeats every 4 positions.



SPECIFICATIONS

Pushbutton Switch Ratings

Rating: 5 Vdc, 10 mA, resistive
Contact Resistance: less than 10 ohms (TTL or CMOS Compatible)
Voltage Breakdown: 250 Vac between mutually insulated parts.
Contact Bounce: Less than 4 milliseconds at make and less than 10 milliseconds at break.
Actuation Life: 3,000,000 operations.
Actuation Force: maximum actuation force of 330 grams and a minimum actuation force of 250 grams.

Encoder Ratings

Coding: 2-bit quadrature coded output.
Operating Voltage: 5.0 ±.25 Vdc
Supply Current: 30 mA maximum at 5 Vdc
Logic High: 3.8V for CMOS and 2.7V for TTL minimum.
Logic Low: 0.8V maximum
Logic Rise and Fall Times: Rise Time less than 30 mS at 16.6 RPM. Fall Time less than 30 mS at 16.6 RPM.
Operating Torque: 1.5 in-oz ± 30% initial (1.0 in-oz ± 50% after life for 32 position only)
Rotational Life: more than 1,000,000 cycles of operation (1 cycle = 360° rotation and return)
Shaft Push Out Force: 20 lbs minimum
Mounting Torque: 10 in-lbs maximum

Environmental Ratings

Operating Temperature Range: –40°C to 85°C
Storage Temperature Range: –55°C to 100°C
Relative Humidity: 90-95% at 40°C for 96 hours.
Vibration Resistance: Harmonic motion with amplitude of 15g, within a varied 10 to 2000 Hz frequency for 12 hours per MIL-STD-202, Method 204
Shock Resistance: Test 1: Tested at 100g for 6 mS, half sine, 12.3 ft/s Test 2: 100g for 6 mS, sawtooth, 9.7 ft/s

Materials and Finishes

Bushing: Reinforced thermoplastic
Shaft: Reinforced thermoplastic
Detent Balls: Steel, nickel-plated
Detent and Pushbutton Springs: Tinned music wire
Printed Circuit Boards: NEMA grade FR-4
Pushbutton Contact: Stainless steel, gold-plated
Board Terminals: Phosphor bronze, tin-plated
Mounting Hardware: One brass, cadmium-plated nut and lockwasher supplied with each switch. Nut is 0.094 inches thick by 0.562 inches across flats.
Rotor: Reinforced thermoplastic
Aperture/Dome Retainer: Lexan 141, Polycarbonate

ORDERING INFORMATION

61C22-01-04-02

Series
Style: C = Standard
Angle of Throw: 00 = No detent
11 = 11.25° or 32 Positions
22 = 22.25° or 16 Positions
Coding: 01 = Quadrature

Pushbutton Option: 01 = Without pushbutton, 02 = With pushbutton
Number of Changes per Revolution: 04 for no detent and 22.25° angle of throw
08 for no detent and 11.25° angle of throw

Custom knobs available, see page I-57.

Webcam

Key Specifications		Webcam Instant
Manufacturer		Creative
Manufacturer Part #		73VF004000000
Webcam	Image Resolution	640 x 480
	Video Resolution	352 x 288
	Optical Sensor Type	CMOS
	Still Image Format	JPEG, BMP
	Video Capture	30 fps @ 352 x 288
Connectivity	Connector Type	1 x USB 1.1
Miscellaneous	Included Software	Creative WebCam Center ArcSoft PhotoImpression ArcSoft Multimedia Email
System Compatibility	PC	Windows 98 SE, Me, 2000, XP
Warranty		One year limited warranty
Includes in the Box		Creative WebCam Instant Quick Start guide Installation and application CDs: Creative WebCam Center,

Power Supply

Phantom

• Force feedback workspace	~6.4 W x 4.8 H x 2.8 D in. > 160 W x 120 H x 70 D mm.
Footprint (Physical area device base occupies on desk)	6 5/8 W x 8 D in. ~168 W x 203 D mm.
Weight (device only)	3 lbs. 15 oz.
Range of motion	Hand movement pivoting at wrist
Nominal position resolution	> 450 dpi. ~ 0.055 mm.
Backdrive friction	<1 oz. (0.26 N)
Maximum exertable force at nominal (orthogonal arms) position	0.75 lbf. (3.3 N)
Continuous exertable force (24 hrs.)	> 0.2 lbf. (0.88 N)
Stiffness	X axis > 7.3 lbs. / in. (1.26 N / mm.) Y axis > 13.4 lbs. / in. (2.31 N / mm.) Z axis > 5.9 lbs. / in. (1.02 N / mm.)
Inertia (apparent mass at tip)	~0.101 lbm. (45 g)
Force feedback	x, y, z
Position sensing [Stylus gimbal]	x, y, z (digital encoders) [Pitch, roll, yaw (± 5% linearity potentiometers)
Interface	IEEE-1394 FireWire® port
Supported platforms	Intel-based PCs
GHOST® SDK compatibility	No
3D Touch™ SDK compatibility	Yes
Applications	Selected Types of Haptic Research, FreeForm® Concept™ system, ClayTools™ system

APPENDIX B. PROGRAMS

OOPIC

```
//Program by Adam Bogenrief
//OOpic program to read sonar ping
//from Devantech SRF-04 Ultrasonic Range Finder
//And a GreyHill Quadrature Encoder (Velocity)

//Create new Sonar Objects
oSonarDV a = new oSonarDV;
oSonarDV b = new oSonarDV;
oSonarDV c = new oSonarDV;
oSonarDV d = new oSonarDV;
oSonarDV e = new oSonarDV;

//Create a new Serial Port Object
oSerial ser = new oSerial;

//Create a new Quad Encoder Object
oQencode quad = new oQencode;

Sub void Main(void)
{
        //Set the Echo and Pulse pins for sonar
        a.IOLineE = 29;
        a.IOLineP = 28;
        b.IOLineE = 15;
        b.IOLineP = 14;
        c.IOLineE = 13;
        c.IOLineP = 12;
        d.IOLineE = 11;
        d.IOLineP = 10;
        e.IOLineE = 9;
        e.IOLineP = 8;

        //Ensure not pinging till wanted
        a.Operate = 0;
        b.Operate = 0;
        c.Operate = 0;
        d.Operate = 0;
        e.Operate = 0;


        //Set LED's on board to light up
        LEDa.IOLine = 6;
        LEDa.Direction = cvOutput;
        LEDa.Value = 0;
        LEDb.IOLine = 5;
        LEDb.Direction = cvOutput;
        LEDb.Value = 0;
```

```
LEDc.IOLine = 7;
LEDc.Direction = cvOutput;
LEDc.Value = 0;



//Set A and B pins for Quadencoder
quad.IOLine1 = 31; //A
quad.IOLine2 = 30; //B
//Ensure data updates
quad.Operate = 0;
//Make data signed
quad.Signed = 0;


//Activate serial port
ser.Operate = cvTrue;
//Set transmission speed
ser.Baud = cv9600;
//Set mode to asynchronous
ser.Mode = 0;


quad.Value = 0;

while(1)
{
        //Trigger the Sonars
        a.Operate = 0;
        a.Operate = 1;
        b.Operate = 0;
        b.Operate = 1;
        c.Operate = 0;
        c.Operate = 1;
        d.Operate = 0;
        d.Operate = 1;
        e.Operate = 0;
        e.Operate = 1;

        //Reset the Encoder
        quad.Value = 0;
        //Start the Encoder
        quad.Operate = 1;
        //500ms wait for echo and give time for encoder to count
        OOPic.Delay = 50;
        //Stop updating Encoder
        quad.Operate = 0;
        //Multiply by 2 to get counts/sec
        //quad.Value = quad.Value * 2;
        //Divide count by # counts/rev(43) to get rev/s
        //quad.Value = quad.Value / 43;
        //Multiply by circumference to convert rev/s to ft/s
        //quad.Value = quad.Value * 1.27;
        //Send the Data
```

```
                ser.string = Str$(a.Value) + " " + Str$(b.Value) + " " + Str$(c.Value) + " " + Str$(d.Value) + "
" + Str$(e.Value) + " " + Str$(quad.Value) + " x";}}
```

Java Serial

```
//Class to read in from the Serial Port
//Written by: Adam Bogenrief
//March 2006

import java.io.IOException;
import java.io.InputStream;
import java.util.*;
import javax.comm.*;

public class OopicSerialIn
{

        //MEMBER VARIABLES
        SerialPort sPort;
        SerialListener lsr = new SerialListener();
    CommPortIdentifier id;
    InputStream data;

        //CONSTRUCTOR
        public OopicSerialIn()
        {

                //Find A Computer Port To Use, dev/ttyS0 (Linux)
     try
     {

        id = CommPortIdentifier.getPortIdentifier("/dev/ttyS0");


     }
     catch(NoSuchPortException e)
     {

       System.err.println(e);

     }

     //Port Available, So Open Port
     if(id.getPortType() == CommPortIdentifier.PORT_SERIAL)
     {

       try
       {

    //make new serial port object and open it
    //Default settings: 9600 Baud, 8 Data Bits, 1 Stop Bit, NoParity
          sPort = (SerialPort) id.open("VADR", 2000);

          data = sPort.getInputStream();
       }
```

```
   catch (PortInUseException e)
   {

      System.err.println(e);


   }
   catch (IOException e)
   {

     System.err.println(e);


   }
}

//Add Event Listener
try
{

   //enable particular event
   sPort.notifyOnDataAvailable(true);
   //add listener
   sPort.addEventListener(lsr);

}
catch(TooManyListenersException e)
{
   System.err.println(e);
}

     }

    //METHODS
public void closePort(SerialPort serport)//OopicSerialIn.sPort
{
   serport.close();
}

   //Inner class that implements a serialporteventlistener
   public class SerialListener implements SerialPortEventListener
   {

   //MEMBER VARIABLES
   protected String sonar1 = new String();
   protected String sonar2 = new String();
   protected String sonar3 = new String();
   protected String sonar4 = new String();
   protected String sonar5 = new String();
   protected String velocity = new String();
   protected byte[] buffer = new byte[37];

   //CONSTRUCTOR
   SerialListener()
```

```
    {
     super();
    }


    //METHODS
    public void serialEvent(SerialPortEvent evt)
    {
     if(evt.getEventType() == SerialPortEvent.DATA_AVAILABLE)
     {
                                    //Serial data is sent a byte at a time, so need to    //build a bunch of bytes
                        into the appropriate string
       //Clear the buffer
       clear();

                                //for statement to populate buffer with bytes read //in from serial port
              try
                                    {
                              for(int i = 0;i<buffer.length; i++)
                                                  {
                                                   buffer[i] = (byte) data.read();
                                                  }
                              }
              catch (IOException e)
                              {

                                    e.printStackTrace();
                              }

    //Make new strings by calling the appropriate constructor
       sonar1 = new String(buffer,0,5);
       sonar2 = new String(buffer,6,5);
       sonar3 = new String(buffer,12,5);
       sonar4 = new String(buffer,18,5);
       sonar5 = new String(buffer,24,5);
       velocity = new String(buffer,30,5);

      }
    }

    public void clear()
    {
     for(int i=0; i<buffer.length; i++)
     {
      buffer[i] = 0;
     }
    }
  }
}
```

Java Socket Server

```
//Program Written by Jesse Lane and Adam Bogenrief to handle socket //server side
import java.net.DatagramSocket;
import java.net.SocketException;
import java.net.DatagramPacket;
import java.io.IOException;
public class UDPServoServer
{
   PhidgetServoControl controller;
   DatagramSocket socket;
   DatagramPacket packet;

   public UDPServoServer(int port, String path) throws SocketException
   {
   controller = new PhidgetServoControl(path);
   socket = new DatagramSocket(port);
   packet = new DatagramPacket(new byte[100], 100);
   }

            public boolean recv() throws IOException
            {
                  socket.receive(packet);
               String strbuf = new String(packet.getData());
               if(strbuf.trim().equals("die"))
               {
                  return false;
               }
               int[] servoNum = new int[2];
               double[] position = new double[2];

               // Parse packet -- in the format of:
               //   servo (null) position (null) servo (null) position ...
               String[] values = strbuf.split("\0");
               for (int i = 0; i < servoNum.length; i++) {
                    servoNum[i] = Integer.parseInt(values[i*2]);
                    position[i] = Double.parseDouble(values[i*2+1]);
               }

               // Send positions to servos
               for (int i = 0; i < servoNum.length; i++) {
                    controller.writeServo(servoNum[i], position[i]);
               }

               return true;
            }
         }
```

Java Socket Client

```
//Class to send data to server in VADR Haptic Control Program
//Written by: Adam Bogenrief
//March 2006
import java.io.*;  //for IOException
import java.net.*; //for DatagramSocket,DatagramPacket,and InetAddress
public class UDPClient
{

	//Member Variables
	private DatagramSocket sock;
	private DatagramPacket pack;
	private OopicSerialIn ser;
	private String Sonar1;
	private String Sonar2;
	private String Sonar3;
	private String Sonar4;
	private String Sonar5;
	private String Velocity;


	//Constructor
	public UDPClient(int port)throws SocketException
	{

//Make a socket
//Make a packet(byte array, length of array, dest address, dest port)

		sock = new DatagramSocket(port);
	try
		{

			pack = new DatagramPacket(new
byte[100],100,InetAddress.getByName("192.168.0.8"),port);

		}
		catch (UnknownHostException e)
		{

			e.printStackTrace();

		}
		//Make a serial port reader
		ser = new OopicSerialIn();

	}

	//Methods
	public boolean send()throws IOException
	{
		//Get the data from the serial port and add escape //sequences for parsing over socket
		Sonar1 = ser.lsr.sonar1 + "\0";
		Sonar2 = ser.lsr.sonar2 + "\0";
		Sonar3 = ser.lsr.sonar3 + "\0";
```

```
Sonar4 = ser.lsr.sonar4 + "\0";
Sonar5 = ser.lsr.sonar5 + "\0";
Velocity = ser.lsr.velocity + "\0";

//System.out.println(Sonar1);
//System.out.println(Sonar2);
//System.out.println(Sonar3);
//System.out.println(Sonar4);
//System.out.println(Sonar5);
//System.out.println(Velocity);

//Convert strings to Bytes to send
byte[] s1 = Sonar1.getBytes();
byte[] s2 = Sonar2.getBytes();
byte[] s3 = Sonar3.getBytes();
byte[] s4 = Sonar4.getBytes();
byte[] s5 = Sonar5.getBytes();
byte[] v = Velocity.getBytes();

//place holding ints
int one = Sonar1.length();
int two = Sonar1.length()+Sonar2.length();
        int three =  Sonar1.length()+Sonar2.length()+Sonar3.length();
int four = Sonar1.length()+Sonar2.length()+Sonar3.length()+Sonar4.length();
int five = Sonar1.length()+Sonar2.length()+Sonar3.length()+Sonar4.length()+Sonar5.length();
int six =
Sonar1.length()+Sonar2.length()+Sonar3.length()+Sonar4.length()+Sonar5.length()+Velocity.
length();

//Concatenate into a single byte array
byte[] dat = new byte[six];
//Populate data
for(int i = 0; i < Sonar1.length(); i++)
{
        dat[i] = s1[i];
}
for(int i = one; i < one + Sonar2.length(); i++)
{
        dat[i] = s2[i-one];
}
for(int i = two; i < two + Sonar3.length(); i++)
{
        dat[i] = s3[i-two];
}
for(int i = three; i < three + Sonar4.length(); i++)
{
        dat[i] = s4[i-three];
}
for(int i = four; i < four + Sonar5.length(); i++)
{
        dat[i] = s5[i-four];
}
for(int i = five; i < five + Velocity.length(); i++)
{
```

```
                dat[i] = v[i-five];
            }

            //Populate packet with data from above
            pack.setData(dat);

            //Send the data over the socket
            sock.send(pack);

            return true;
        }
}
```

C++ Socket Headers

```
//Written By: Jesse Lane
#include "stringsocketdata.h"
#include <vector>
#include <string>
#include <sstream>
#include <iomanip>
int StringSocketSend::send()
{
	std::vector<char> buffer;
	char nullChar = '\0';

	for(SocketIter = SocketList.begin(); SocketIter != SocketList.end(); SocketIter++)
	{
		std::stringstream ss;
		int size = SocketIter->array_size;
		if(SocketIter->format == CHAR_FORMAT)
		{
			char* tmpbuf = (char*)SocketIter->content;
			for(int i = 0; i < size; i++)
			{
				buffer.push_back(tmpbuf[i]);
			}
			buffer.push_back(nullChar);
		}
		else if(SocketIter->format == INT_FORMAT)
		{
			int* tmpbuf = (int*)SocketIter->content;
			for(int i = 0; i < size; i++)
			{
				ss << tmpbuf[i];
				std::string str = ss.str();
				for(int j = 0; j < str.length(); j++)
				{
					buffer.push_back(str[j]);
				}
				buffer.push_back(nullChar);
				ss.clear();
			}
		}
		else if(SocketIter->format == FLOAT_FORMAT)
		{
			float* tmpbuf = (float*)SocketIter->content;
			for(int i = 0; i < size; i++)
			{
				ss << std::setiosflags(std::ios::fixed) << tmpbuf[i];
				std::string str = ss.str();
				for(int j = 0; j < str.length(); j++)
				{
					buffer.push_back(str[j]);
				}
				buffer.push_back(nullChar);
				ss.clear();
```

```
                }
            }
            else if(SocketIter->format == DOUBLE_FORMAT)
            {
                double* tmpbuf = (double*)SocketIter->content;
                for(int i = 0; i < size; i++)
                {
                    ss << std::setiosflags(std::ios::fixed) << tmpbuf[i];
                    std::string str = ss.str();
                    for(int j = 0; j < str.length(); j++)
                    {
                        buffer.push_back(str[j]);
                    }
                    buffer.push_back(nullChar);
                    ss.clear();
                }
            }
            else if(SocketIter->format == CHAR_ARRAY_FORMAT)
            {
                char** tmpbuf = (char**)SocketIter->content;
                for(int i = 0; i < size; i++)
                {
                    ss << tmpbuf[i];
                    std::string str = ss.str();
                    for(int j = 0; j < str.length(); j++)
                    {
                        buffer.push_back(str[j]);
                    }
                    buffer.push_back(nullChar);
                    ss.clear();
                }
            }
        }
        if(packet != NULL)
        {
            delete packet;
        }
        packet_size = buffer.size();
        packet = new char[packet_size];
        for(int i = 0; i < packet_size; i++)
        {
            packet[i] = buffer[i];
        }
        return sendto(theSocket, packet, packet_size, 0,
         (LPSOCKADDR)&saServer, nFromLen);
}

int StringSocketRecv::recv()
{
        char nullChar = '\0';
        int packetIndex = 0;
        int bufsize = 1024;
        if(packet != NULL)
        {
```

```
		delete packet;
}
packet = new char[bufsize];
if(recvfrom(theSocket, packet, bufsize, 0, (LPSOCKADDR)&saServer, &nLen) < 0)
{
		return -1;
}

for(SocketIter = SocketList.begin(); SocketIter != SocketList.end(); SocketIter++)
{
		std::stringstream ss;
		ss << std::setiosflags(std::ios::fixed);
		int size = SocketIter->array_size;
		if(SocketIter->format == CHAR_FORMAT)
		{
				if(packetIndex + size < bufsize)
				{
						memcpy(SocketIter->content, packet + packetIndex, size);
						packetIndex += size + 1;
				}
		}
		else if(SocketIter->format == INT_FORMAT)
		{
				for(int i = 0; i < size; i++)
				{
						while(packetIndex < bufsize && packet[packetIndex] != nullChar)
						{
								ss << packet[packetIndex];
								packetIndex++;
						}
						packetIndex++;
						ss >> *((int*)SocketIter->content + i);
						ss.clear();
				}
		}
		else if(SocketIter->format == FLOAT_FORMAT)
		{
				for(int i = 0; i < size; i++)
				{
						while(packetIndex < bufsize && packet[packetIndex] != nullChar)
						{
								ss << packet[packetIndex];
								packetIndex++;
						}
						packetIndex++;
						ss >> *((float*)SocketIter->content + i);
						ss.clear();
				}
		}
		else if(SocketIter->format == DOUBLE_FORMAT)
		{
				for(int i = 0; i < size; i++)
				{
						while(packetIndex < bufsize && packet[packetIndex] != nullChar)
```

```
                                {
                                        ss << packet[packetIndex];
                                        packetIndex++;
                                }
                                packetIndex++;
                                ss >> *((double*)SocketIter->content + i);
                                ss.clear();
                        }
                }
                else if(SocketIter->format == CHAR_ARRAY_FORMAT)
                {

                }
        }
        return 0;
}
```

```
//Program Written By  Jesse Lane

#ifndef _STRINGSOCKETDATA_H
#define _STRINGSOCKETDATA_H

#define CHAR_FORMAT 1
#define INT_FORMAT 2
#define FLOAT_FORMAT 3
#define DOUBLE_FORMAT 4
#define CHAR_ARRAY_FORMAT 5
#define OTHER_FORMAT 6

#include "socketdata.h"

class StringSocketSend : public SocketSend
{
public:
        int send();
};

class StringSocketRecv : public SocketRecv
{
public:
        int recv();
};

#endif
```

C++ Shape

```
//Program Written By Adam Bogenrief
//OpenGL Includes
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <GL/gl.h>
#include <GL/glut.h>
#include "Vector.h";
//Class to draw a Square for haptics
class Square
{
	//Declare Member Variables
public:
	double lenx; //Half of size (look at draw method)
	double leny;
	double lenz;
	vec3_t ctr;

	//Default Constructor
	Square()
	{
		ctr.Set(0,0,12.0);
		lenx = 6.0;
		leny = 6.0;
		lenz = 6.5;
	}

	//Methods
	void draw()
	{
		int xbegin=ctr.X()-lenx;
		int ybegin=ctr.Y()-leny;
		int znear=ctr.Z()+lenz;
		int xend=ctr.X()+lenx;
		int yend=ctr.Y()+leny;
		int zfar=ctr.Z()-lenz;

		int hxbegin=ctr.X()-5.0;
		int hxend=ctr.X()+5.0;
		int hybegin=ctr.Y()-5.5;
		int hyend=ctr.Y()+5.5;
		int hznear=ctr.Z()+6.5;
		int hzfar=ctr.Z()-6.5;

		//Set fill Mode
			glPolygonMode(GL_FRONT, GL_FILL);
			glPolygonMode(GL_BACK, GL_FILL);
				//Draw Boundary
				//LeftSide
				glBegin(GL_QUADS);
				glNormal3f(1.0, 0.0, 0.0);
```

```
                glVertex3f(hxbegin, ybegin, znear);
                glVertex3f(hxbegin, ybegin, zfar);
                glVertex3f(hxbegin, yend, zfar);
                glVertex3f(hxbegin, yend, znear);
        glEnd();
        //BackSide
        glBegin(GL_QUADS);
        glNormal3f(0.0, 0.0, 1.0);
                glVertex3f(xbegin, ybegin, hzfar);
                glVertex3f(xend, ybegin, hzfar);
                glVertex3f(xend, yend, zfar);
                glVertex3f(xbegin, yend, hzfar);
        glEnd();
        //FrontSide
        glBegin(GL_QUADS);
        glNormal3f(0.0, 0.0, -1.0);
                glVertex3f(xbegin, ybegin, hznear);
                glVertex3f(xbegin, yend, hznear);
                glVertex3f(xend, yend, hznear);
                glVertex3f(xend, ybegin, hznear);
        glEnd();
        //RightSide
        glBegin(GL_QUADS);
        glNormal3f(-1.0, 0.0, 0.0);
                glVertex3f(hxend, ybegin, znear);
                glVertex3f(hxend, yend, znear);
                glVertex3f(hxend, yend, zfar);
                glVertex3f(hxend, ybegin, zfar);
        glEnd();
        //TopSide
        glBegin(GL_QUADS);
        glNormal3f(0.0, -1.0, 0.0);
                glVertex3f(xbegin, hyend, znear);
                glVertex3f(xbegin, hyend, zfar);
                glVertex3f(xend, hyend, zfar);
                glVertex3f(xend, hyend, znear);
        glEnd();
        //BottomSide
        glBegin(GL_QUADS);
        glNormal3f(0.0, 1.0, 0.0);
                glVertex3f(xend, hybegin, znear);
                glVertex3f(xend, hybegin, zfar);
                glVertex3f(xbegin, hybegin, zfar);
                glVertex3f(xbegin, hybegin, znear);
        glEnd();
}

void line()
{
        int xbegin=ctr.X()-lenx;
        int ybegin=ctr.Y()-leny;
        int znear=ctr.Z()+lenz;
        int xend=ctr.X()+lenx;
        int yend=ctr.Y()+leny;
```

```
                int zfar=ctr.Z()-lenz;

                int hxbegin=ctr.X()-5.0;
                int hxend=ctr.X()+5.0;
                int hybegin=ctr.Y()-5.5;
                int hyend=ctr.Y()+5.5;
                int hznear=ctr.Z()+5.5;
                int hzfar=ctr.Z()-5.5;

                //Set fill Mode
                glPolygonMode(GL_FRONT, GL_LINE);
                glPolygonMode(GL_BACK, GL_LINE);
                //FrontSide
                glBegin(GL_QUADS);
                glNormal3f(0.0, 0.0, -1.0);
                        glVertex3f(xbegin, ybegin, hznear);
                        glVertex3f(xbegin, yend, hznear);
                        glVertex3f(xend, yend, hznear);
                        glVertex3f(xend, ybegin, hznear);
                glEnd();
        }
};
```

C++ Control Program

```
//Program Written By Adam Bogenrief
//OpenGL Includes
#include <windows.h>
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <GL/glui.h>
#include <GL/gl.h>
#include <GL/glut.h>
#include "Vector.h"
#include "Square.h"
//OpenSG Includes
#include <OpenSG/OSGGLUT.h>
#include <OpenSG/OSGConfig.h>
#include <OpenSG/OSGSimpleGeometry.h>
#include <OpenSG/OSGGLUTWindow.h>
#include <OpenSG/OSGSolidBackground.h>
#include <OpenSG/OSGDirectionalLight.h>
#include <OpenSG/OSGPerspectiveCamera.h>
#include <OpenSG/OSGTransform.h>
#include <OpenSG/OSGRenderAction.h>
#include <OpenSG/OSGViewport.h>
#include <OpenSG/OSGImage.h>
#include <OpenSG/OSGSceneFileHandler.h>
#include <OpenSG/OSGSimpleTexturedMaterial.h>
#include <OpenSG/OSGMaterial.h>
#include <OpenSG/OSGMaterialGroup.h>
#include <OpenSG/OSGMatrixUtility.h>
#include <OpenSG/OSGMatrixCamera.h>
#include <OpenSG/OSGComponentTransform.h>
#include <OpenSG/OSGPassiveWindow.h>
#include <OpenSG/OSGPassiveViewport.h>
#include <OpenSG/OSGPassiveBackground.h>
//Haptic Includes
#include <HL/hl.h>
#include <HDU/hduMatrix.h>
#include <HDU/hduError.h>
#include <HLU/hlu.h>
#include <HD/hd.h>
//Socket Includes
#include "stringsocketdata.h"

//dont want to preface everything with OSG::
OSG_USING_NAMESPACE;
//Use standard namespace
using namespace std;

//Globally create some necessary Haptic Handles
HHD hHD;            //Device
HHLRC hHLRC;        //Rendering Context
HDErrorInfo error;  //Error
HLuint myShapeId;   //Shape ID
HLuint spring;      //Effect ID
```

```
//Define Anchor Pos for Spring effect
struct anchorPos
{
   hduVector3Dd a_position;
};
//Globally create an Anchor
anchorPos Anchor;

//Create initial position variable
hduVector3Dd prevPos;

//Globally create some necessary socket pointers and variables
//Socket to send data
StringSocketSend * sender;
////Socket to receive data
StringSocketRecv * receiver;
//Initialize some data to send
int motor2 = 2; //Steering
double pos2 = 110.0; //Center
double steer;
int motor3 = 3; //Drive
double pos3 = 125.0; //Stationary
double drive;
//Initialize some data to receive
float Velocity = 0.0;
double Sonar1 = 0.0;
double Sonar2 = 0.0;
double Sonar3 = 0.0;
double Sonar4 = 0.0;
double Sonar5 = 0.0;
//Setup the destination address
char Server[] = "192.168.0.26";
//Setup the ports
short portin = 1061;
short portout = 1060;

//Make boolean variable to handle On/Off
bool trigger = false;
//Make boolean variable to handle starting point
bool trip = true;

//make a boundary object for Haptics
Square bound;

/*************************************************************
*************************************************************
                                                    OpenSG
*************************************************************
*************************************************************/
//Setup some global variables
NodePtr scene;
RenderAction *renderAction;
PassiveWindowPtr mWin;
PassiveViewportPtr mView;
```

```
PassiveBackgroundPtr mBack;
MatrixCameraPtr mCamera;
//Globally create SG motion pointers so can manipulate anywhere
//ComponentTransformPtr tcVehicle;
ComponentTransformPtr tcFLwheel;
ComponentTransformPtr tcFRwheel;


//Method to Create the SceneGraph
NodePtr createScenegraph()
{
        /*****************Make All Nodes***************/
        //Create Geometry Cores
   NodePtr vehicle = SceneFileHandler::the().read("C:\\Documents and Settings\\me\\Desktop\\Frame.wrl");
   NodePtr FLwheel = SceneFileHandler::the().read("C:\\Documents and Settings\\me\\Desktop\\Wheel.wrl");
        NodePtr FRwheel = SceneFileHandler::the().read("C:\\Documents and
Settings\\me\\Desktop\\Wheel.wrl");
        NodePtr RLwheel = SceneFileHandler::the().read("C:\\Documents and
Settings\\me\\Desktop\\Wheel.wrl");
        NodePtr RRwheel = SceneFileHandler::the().read("C:\\Documents and
Settings\\me\\Desktop\\Wheel.wrl");

        //Create Transform Cores
        ComponentTransformPtr tcVehicle = ComponentTransform::create();
        tcFLwheel = ComponentTransform::create();
        tcFRwheel = ComponentTransform::create();
        TransformPtr tcRLwheel = Transform::create();
        TransformPtr tcRRwheel = Transform::create();
        TransformPtr lightTrans = Transform::create();
        ComponentTransformPtr CamTrans = ComponentTransform::create();

        //Create Group Cores
        GroupPtr gcVehicle = Group::create();

        //Make Nodes
        NodePtr vehicleNode = Node::create();
        NodePtr CamBeacon = Node::create();
        NodePtr lightBeacon = Node::create();
   NodePtr lightNode = Node::create();

        //Make Specialized Nodes
        PerspectiveCameraPtr Camera1 = PerspectiveCamera::create();
        mView = PassiveViewport::create();
        mBack = PassiveBackground::create();
        DirectionalLightPtr dLight = DirectionalLight::create();
        SimpleMaterialPtr m = SimpleMaterial::create();
        MaterialGroupPtr mg = MaterialGroup::create();

        //Make transform nodes
   NodePtr FLwTransformNode = Node::create();
        NodePtr FRwTransformNode = Node::create();
        NodePtr RLwTransformNode = Node::create();
        NodePtr RRwTransformNode = Node::create();
```

```
/*****************SceneGraph Hierarchy***************/
//Setup Vehicle Transform
beginEditCP(vehicleNode);
        //Group Core
        vehicleNode->setCore(gcVehicle);
        //TransformCore
                beginEditCP(tcVehicle);
                        tcVehicle->setTranslation(Vec3f(-5.5,-4.945,2.3625));
                endEditCP(tcVehicle);
        vehicleNode->setCore(tcVehicle);
        //Add Children
        vehicleNode->addChild(vehicle);
        vehicleNode->addChild(lightNode);//Want Light to follow Vehicle
        vehicleNode->addChild(FLwTransformNode);
        vehicleNode->addChild(FRwTransformNode);
        vehicleNode->addChild(RLwTransformNode);
        vehicleNode->addChild(RRwTransformNode);
        vehicleNode->addChild(CamBeacon);
        vehicleNode->addChild(lightBeacon);
        vehicleNode->addChild(lightNode);
endEditCP(vehicleNode);

        //Setup Material for Frame
        beginEditCP(m);
                m->setDiffuse(Color3f(1,0,0));
                m->setAmbient(Color3f(0.797, 0.253, 0.582));
                m->setTransparency(0.0);
        endEditCP(m);
        //Assign Material to Material Node
        beginEditCP(mg);
                mg->setMaterial(m);
        endEditCP(mg);

        //Assign MaterialNode to Geometry Node
        beginEditCP(vehicle);
                vehicle->setCore(mg);
        endEditCP(vehicle);

        //Setup Camera
        beginEditCP(Camera1);
                Camera1->setBeacon(CamBeacon);
                Camera1->setFov(deg2rad(90));
                Camera1->setNear(1.0);
                Camera1->setFar(25);
        endEditCP(Camera1);

        //Setup Light Beacon
        beginEditCP(lightBeacon);
                beginEditCP(lightTrans);
                        Matrix lightM;
                        lightM.setIdentity();
                        lightM.setTransform(Vec3f(1,1,10));
                        lightTrans->setMatrix(lightM);
                endEditCP(lightTrans);
```

```
            lightBeacon->setCore(lightTrans);
        endEditCP(lightBeacon);

        //Setup Camera Beacon
        beginEditCP(CamBeacon);
            beginEditCP(CamTrans);
                Quaternion q1 (Vec3f(0,0,1),-1.5708);
                CamTrans->setTranslation(Vec3f(5.5,4.945,20));
                CamTrans->setRotation(q1);
            endEditCP(CamTrans);
            CamBeacon->setCore(CamTrans);
        endEditCP(CamBeacon);

        //Setup Viewport
        beginEditCP(mView);
            mView->setCamera(Camera1);
            mView->setBackground(mBack);
            mView->setRoot(vehicleNode);
            mView->setSize(0,0,1,1);
        endEditCP(mView);

        //Setup Light Attributes
        beginEditCP(dLight);
            dLight->setDirection(Vec3f(0,0,0));
            //color information
            dLight->setDiffuse(Color4f(1,1,1,1));
            dLight->setAmbient(Color4f(0.2,0.2,0.2,1));
            dLight->setSpecular(Color4f(1,1,1,1));
            //set the beacon
            dLight->setBeacon(lightBeacon);
        endEditCP(dLight);

        //Setup Light
        beginEditCP(lightNode);
          lightNode->setCore(dLight);
        endEditCP(lightNode);

//Setup Wheel Transformations
//FL
beginEditCP(FLwTransformNode);
        //TransformCore
            //Populate Translation Matrix First
            beginEditCP(tcFLwheel);
                tcFLwheel->setTranslation(Vec3f(11.0,9.89,0.0));
            endEditCP(tcFLwheel);
        FLwTransformNode->setCore(tcFLwheel);
        //Add Children
        FLwTransformNode->addChild(FLwheel);
endEditCP(FLwTransformNode);

//FR
beginEditCP(FRwTransformNode);
        //TransformCore
            //Populate Translation Matrix First
```

```
                beginEditCP(tcFRwheel);
                        tcFRwheel->setTranslation(Vec3f(11.0,0.0,0.0));
                endEditCP(tcFRwheel);
        FRwTransformNode->setCore(tcFRwheel);
        //Add Children
        FRwTransformNode->addChild(FRwheel);
    endEditCP(FRwTransformNode);

    //RL
    beginEditCP(RLwTransformNode);
        //TransformCore
                //Populate Translation Matrix First
                Matrix d;
                d.setIdentity();
                d.setTranslate(0.0,9.89,0.0);
                beginEditCP(tcRLwheel);
                        tcRLwheel->setMatrix(d);
                endEditCP(tcRLwheel);
        RLwTransformNode->setCore(tcRLwheel);
        //Add Children
        RLwTransformNode->addChild(RLwheel);
    endEditCP(RLwTransformNode);

    //RR
    beginEditCP(RRwTransformNode);
        //TransformCore
                //Populate Translation Matrix First
                Matrix e;
                e.setIdentity();
                e.setTranslate(0.0,0.0,0.0);
                beginEditCP(tcRRwheel);
                        tcRRwheel->setMatrix(e);
                endEditCP(tcRRwheel);
        RRwTransformNode->setCore(tcRRwheel);
        //Add Children
        RRwTransformNode->addChild(RRwheel);
    endEditCP(RRwTransformNode);

return vehicleNode;

}

/*************************************************************
*************************************************************
                                                        Haptics
*************************************************************
*************************************************************/

/**********************Create Callbacks**********************/
//SPRING EFFECT
void HLCALLBACK startEffectCB(HLcache *cache, void *userdata)
{
    //Not necessary to do anything but make call to this function
}
```

```
void HLCALLBACK computeForceCB(HDdouble force[3], HLcache *cache, void *userdata)
{
        // Get the time delta since the last update
   HDdouble instRate;
   hdGetDoublev(HD_INSTANTANEOUS_UPDATE_RATE, &instRate);
   HDdouble deltaT = 1.0 / instRate;

        //Set the anchor position for the spring
        hduVector3Dd anchorPos;
        anchorPos[0] = 0.0;
        anchorPos[1] = 0.0;
        anchorPos[2] = 12.0;

        //Get current proxy position
        hduVector3Dd currentPos;
        hlCacheGetDoublev(cache, HL_PROXY_POSITION, currentPos);

/*****************Definitely the cause of the problem*****************
        //Get the current position of the device
        hduVector3Dd devicePos;
        hlCacheGetDoublev(cache, HL_DEVICE_POSITION, devicePos);
*********************************************************************/

        //(X - Xold) = Delta X; Delta X / Delta T = Velocity
        //hduVector3Dd velDevice;
        //velDevice = (devicePos - prevPos) / deltaT; //worldcoords/s

        //Declare variables to get max values
        HDdouble spring;
        //HDdouble damp;

        //Get max spring value
        hdGetDoublev(HD_NOMINAL_MAX_STIFFNESS, &spring);

        //Set spring value according to kinematic constraint
        spring *= .05;
        //Set damping value
        //damp = .005;

        //Compute spring force between proxy and anchor
        hduVector3Dd SPRING = spring * (anchorPos - currentPos);
        //printf("SpringForce = %f \n" , SPRING[1]);

        //Compute damping force
        //hduVector3Dd DAMP = -damp * velDevice;
        //printf("DampingForce = %f \n" , DAMP[0]);

        //Send Force Boundary Force plus Spring Force plus Damping Force to Device
        force[0] += 0;//SPRING[0];  //Possibly add spring to lessen turn radius as well
        force[1] += 0;//SPRING[1];//force to slow you down if you try turning too fast
        force[2] += 0;
}
```

```
void HLCALLBACK stopEffectCB(HLcache *cache, void *userdata)
{
        //Not necessary to do anything but make call to this function
}

void HLCALLBACK button1DownCallback(HLenum event, HLuint object, HLenum thread,
                        HLcache *cache, void *userdata)
{
        //Turn off the proxy hold
//      hlEnable(HL_PROXY_RESOLUTION);

   //Toggle the Socket On/Off
        if (trigger == false)
        {
                trigger = true;
        }
        else
        {
                trigger = false;
        }
}

/**********************Setup Device***********************/
void initHL()
{
  //Create a haptic device instance
   hHD = hdInitDevice(HD_DEFAULT_DEVICE);
   if (HD_DEVICE_ERROR(error = hdGetError()))
   {
      hduPrintError(stderr, &error, "Failed to initialize haptic device");
      exit(-1);
   }
  //Create a haptic rendering context and activate it
  hHLRC = hlCreateContext(hHD);
  hlMakeCurrent(hHLRC);
  //Enable optimization of the viewing parameters when rendering geometry for Haptics
  hlEnable(HL_HAPTIC_CAMERA_VIEW);

  //Create handles for haptics
  myShapeId = hlGenShapes(1);
  spring = hlGenEffects(1);

  //Initialize the prevPos
  prevPos[0] = 0.0;
  prevPos[1] = 0.0;
  prevPos[2] = 12.0;

  //Register Event for Button1Down
  hlAddEventCallback(HL_EVENT_1BUTTONDOWN, HL_OBJECT_ANY, HL_CLIENT_THREAD,
&button1DownCallback, NULL);
}

/********************Deallocate Device*******************/
void exitHandler()
```

```
{
  //Free up the haptic rendering context
  hlMakeCurrent(NULL);
  if (hHLRC != NULL)
  {
    hlDeleteContext(hHLRC);
  }

  //Free up the haptic device
  if (hHD != HD_INVALID_HANDLE)
  {
    hdDisableDevice(hHD);
  }

        //Free up shape ID
        hlDeleteShapes(myShapeId, 1);
        //Stop effect
        hlBeginFrame();
        hlStopEffect(spring);
        hlEndFrame();
        //Free up effect ID
        hlDeleteEffects(spring, 1);
}

/*****************************Draw Cursor*********************/
void drawCursor()
{
        //Start Matrix stack
        glPushMatrix();

        // Get the proxy position in world coordinates
        HLdouble proxyPosition[3];
        hlGetDoublev(HL_PROXY_POSITION, proxyPosition);

        //Hook up representation of proxy to end-effector transform
        glTranslatef(proxyPosition[0],proxyPosition[1],proxyPosition[2]);

        //Pick Color
        glColor3f(0,0,1); //Blue

        //Draw representation of proxy as a sphere
        glutSolidSphere(.25,20,20);

        //Stop Matrix stack
        glPopMatrix();
}

/*************************Draw the Graphics*****************************/
void drawSceneGraphics()
{
        //Set up Buffers
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        //Draw Scenegraph
```

```
        glMatrixMode(GL_PROJECTION);
        glPushMatrix();
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        mWin->render(renderAction);
        glPopMatrix();
        glMatrixMode(GL_PROJECTION);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);

        //line.draw();
        bound.line();
        // draw 3D cursor at haptic device position
        drawCursor();
}

/*****************************Draw the Haptics*****************************/
void drawSceneHaptics()
{
        // start haptic frame - must do this before rendering any haptic shapes
        hlBeginFrame();
        //Make Boundary effect First
        // Set material properties for the shapes to be drawn.
  hlMaterialf(HL_FRONT_AND_BACK, HL_STIFFNESS, 0.7f);
  hlMaterialf(HL_FRONT_AND_BACK, HL_DAMPING, 0.1f);
  hlMaterialf(HL_FRONT_AND_BACK, HL_STATIC_FRICTION, 0.2f);
  hlMaterialf(HL_FRONT_AND_BACK, HL_DYNAMIC_FRICTION, 0.3f);
  // Start a new haptic shape.  Use the feedback buffer to capture OpenGL
  // geometry for haptic rendering.
  hlBeginShape(HL_SHAPE_FEEDBACK_BUFFER, myShapeId);
  // Use OpenGL commands to create geometry.
  bound.draw();
  // End the shape.
  hlEndShape();

        //Register Spring effect to add to boundary effect
        hlCallback(HL_EFFECT_COMPUTE_FORCE, (HLcallbackProc) computeForceCB, &Anchor);
        hlCallback(HL_EFFECT_START, (HLcallbackProc) startEffectCB, &Anchor);
        hlCallback(HL_EFFECT_STOP, (HLcallbackProc) stopEffectCB, &Anchor);
        hlStartEffect(HL_EFFECT_CALLBACK, spring);

        //Proxy Position Debugging
        HLdouble proxyPos[3];
        hlGetDoublev(HL_PROXY_POSITION, proxyPos);
        steer = proxyPos[0]; //x
        drive = proxyPos[1]; //y
        //printf("Px = %f \n" , proxyPos[0]);
        //printf("Py = %f \n" , proxyPos[1]);
        //printf("Pz = %f \n" , proxyPos[2]);

        if(trip == true)
        {
                //Necessary to move device
//              hlDisable(HL_PROXY_RESOLUTION);
```

```
                //Desired position to move to
//              HLdouble startpos[3] = {0.0,0.0,12.0};
                //Move device to that position
//              hlProxydv(HL_PROXY_POSITION,startpos);
                //hlEnable(HL_PROXY_RESOLUTION);//Add this to button 1 press
                trip = false;
        }

    // Call any event callbacks that have been triggered.
    hlCheckEvents();

        // end the haptic frame
        hlEndFrame();
}

/*****************Map Device to World Coordinates*************/
void updateWorkspace()
{
//      GLdouble modelview[16];
//    GLdouble projection[16];

 //   glGetDoublev(GL_MODELVIEW_MATRIX, modelview);
//    glGetDoublev(GL_PROJECTION_MATRIX, projection);

        hlWorkspace (-80, -80, -20, 80, 80, 50);

    hlMatrixMode(HL_TOUCHWORKSPACE);
    hlLoadIdentity();
//      hluFeelFrom(0,1,0,0,0,0,0,1,0);
    hlOrtho (-10.0, 10.0, -10.0, 10.0, -10.0, 10.0);

    // Fit haptic workspace to view volume.
 //   hluFitWorkspace(projection);

    /* fit haptic workspace to view volume.
        Specify the boundaries for the workspace of the haptic device
        in millimeters in the coordinates of the haptic device.
        The haptics engine will map the view volume to this workspace.
        NOTE:  It wont clip out the rest of the workspace, simply fits the amount of
        workspace specified in hlworkspace into the bounding box specified by hlOrtho.*/
//      hlWorkspace (-80, -80, -20, 80, 80, 50); //Device Space
        //hlWorkspace (-80, -80, -50, 80, 80, 20);
        //Left,Bottom,Far,Right,Top,Near

        // specify the haptic view volume
//      hlMatrixMode(HL_TOUCHWORKSPACE);
//      hlPushMatrix();
        //Clear the matrix stack
//      hlLoadIdentity();
        //specify the haptic view volume
        //From 0,0,0 cause this doesnt know where the Camera is
//      hlOrtho (-10.0, 10.0, -10.0, 10.0, -10.0, 10.0);  //World Space
        //Left,Right,Bottom,Top,Near,Far
//      hlPopMatrix();
```

```
        // specify the haptic view-Touch Matrix
//      hlMatrixMode(HL_VIEWTOUCH);
        //Rotate so that z is y
//      hluFeelFrom(0,1,0,0,0,0,1,0);
        //(look from, origin, up)

}
/*************************************************************
*************************************************************
                                                    Socket Setup
*************************************************************
*************************************************************/
//Method to initialize the sockets
void initSocket(void)
{
        // create a socket object
        sender = new StringSocketSend();

        //push the data onto the list
        sender->push(&motor2);  //Steering Angle motor command
        sender->push(&pos2);
        sender->push(&motor3);  //Speed motor command
        sender->push(&pos3);

        //Open the socket
        int open1 = sender->open(Server, portout);

        //Output to Verify it worked
        if(open1 == -1)
        {
                std::cout << "\nDidn't connect to Socket!\n";
        }

        std::cout << "\nConnected to Socket. Host is: " << Server << "; port # is: " << portout << std::endl;

        // create a receiver socket
        receiver = new StringSocketRecv();
        //Initialize the kind of data that is wanted
        receiver->push(&Sonar1);  //Sonar1
        receiver->push(&Sonar2);  //Sonar2
        receiver->push(&Sonar3);  //Sonar3
        receiver->push(&Sonar4);  //Sonar4
        receiver->push(&Sonar5);  //Sonar5
        receiver->push(&Velocity);  //Velocity

        //Open the socket
        int open2 = receiver->open(portin,false);  //blocking halts program till all the data is there, which ties
up processesor and dont want that

        //Output to Verify it worked
        if(open2 == -1)
        {
                std::cout << "\nDidn't connect to receiver!\n";
```

```
        }

        std::cout << "\nConnected to Receiver.\n";
}

//Method to update socket data and graphic front tires
void updateDataOut(void)
{

        //X is steering angle
        pos2 = 110.0 + ((steer/5.0)*30.0); //handles + or -
        //printf("steer = %f \n" , pos2);
        //Move graphical representation with wheels
        beginEditCP(tcFLwheel);
        beginEditCP(tcFRwheel);
                tcFLwheel->setRotation(Quaternion(Vec3f(0,0,1),(((-steer/5.0)*35.0)*.017453)));
                tcFRwheel->setRotation(Quaternion(Vec3f(0,0,1),(((-steer/5.0)*35.0)*.017453)));
        endEditCP(tcFRwheel);
        endEditCP(tcFLwheel);

        //Y is speed
        //Reverse
        if(drive < 0)
        {
                pos3 = 130.0 + (abs(drive)/7.0)*40; //7.0
        }
        //Forward
        else
        {
                pos3 = 120.0 - (drive/7.0)*15; //45
        }
        if(pos3 < 110)
        {
                pos3 = 110;
        }
        if(pos3 > 140)
        {
                pos3 = 140;
        }
        //printf("drive = %f \n" , pos3);
}

void updateDataIn(void)
{
        //Need to convert to unsigned word to signed word
        if((Velocity > 0) && (Velocity <= 3268))
        {
                Velocity *= -.05907; //convert to ft/s
        }
        else
        {
                Velocity -= 65534;
                Velocity = abs(Velocity)*.05907; //convert to ft/s
        }
```

```
        //Need to convert to ft
        Sonar1 /= 64.0;
        Sonar2 /= 64.0;
        Sonar3 /= 64.0;
        Sonar4 /= 64.0;
        Sonar5 /= 64.0;

        //update glui variables
//      glui1->sync_live();
//      glui2->sync_live();
//      glui3->sync_live();
}

/************************************************************
************************************************************
                                                OpenGL Setup
************************************************************
************************************************************/
//Method to Display
void display(void)
{
        //Populate socket with appropriate motor commands to VADR
        updateDataOut();
        //Send the data at appropriate time
        if (trigger == true)
        {
                int send = sender->send();
        }

        //Receive the data
        int rec = receiver->recv();
        //Make necessary conversions to the data
        updateDataIn();

        //Draw Graphics
        drawSceneGraphics();

        //Draw Haptics
        drawSceneHaptics();

  glutSwapBuffers();
}

/**********************Haptic Reshape*******************************/
void glutReshape(int width, int height)
{
  glViewport(0, 0, width, height);

  // Compute the viewing parameters based on a fixed fov and viewing
  // a canonical box centered at the origin
  double aspect = width / height;

  glMatrixMode(GL_PROJECTION);
```

```
    glLoadIdentity();
    //gluPerspective(90, aspect,5 ,35 );
            glOrtho(-11.0,11.0,-11.0,11.0,2,35);
    // Place the camera down the Z axis looking at the origin
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0, 0, 25, 0, 0, 0, 0, 1, 0); //Camera that looks down the Z axis
            //gluLookAt(0, 25, 0, 0, 0, 0, 0, 0, -1); //Camera that looks down the Y axis

    updateWorkspace();
}

//Method to handle Idles
void glutIdle()
{
    HLerror error;

    while (HL_ERROR(error = hlGetError()))
    {
        fprintf(stderr, "HL Error: %s\n", error.errorCode);

        if (error.errorCode == HL_DEVICE_ERROR)
        {
            hduPrintError(stderr, &error.errorInfo,
                "Error during haptic rendering\n");
        }
    }

    glutPostRedisplay();
}
//Method to setup OpenGL
void initGL()
{
    static const GLfloat light_model_ambient[] = {.2f, .3f, .1f, 1.0f};
    static const GLfloat light0_diffuse[] = {0.5f, 0.5f, 0.5f, 0.1f};
            static const GLfloat light0_specular[] = {0.2f, 0.2f, 0.2f, 0.2f};
    static const GLfloat light0_direction[] = {0.0f, 0.0f, 0.0f, 1.0f};

            //Make background white
            glClearColor(1.0,1.0,1.0,0.0);

    //Enable depth buffering for hidden surface removal.
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);

    //Set lighting parameters
    glEnable(GL_LIGHTING);
    glEnable(GL_NORMALIZE);
    glShadeModel(GL_SMOOTH);
            //Needed to Enable reflection of colors
            glEnable(GL_COLOR_MATERIAL);

    glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
    glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

```
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, light_model_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
        glLightfv(GL_LIGHT0, GL_SPECULAR, light0_specular);
    glLightfv(GL_LIGHT0, GL_POSITION, light0_direction);
    glEnable(GL_LIGHT0);
}

//Method to handle Inits
void initScene()
{
    initGL();
    initHL();
        initSocket();
}

/*************************************************************
**************************************************************
                                                        Main
**************************************************************
**************************************************************/
int main(int argc, char **argv)
{
  //Start OpenGL
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(600, 600);
  glutInitWindowPosition(100, 100);
  int main_window = glutCreateWindow("Phantom Teleoperation Interface");

                //Set glut callback functions
                glutDisplayFunc(display);
                glutReshapeFunc(glutReshape);
                glutIdleFunc(glutIdle);
                //Provide a cleanup routine for handling application exit
                atexit(exitHandler);

  //Call initializations
  initScene();

  //Start OpenSG
  osgInit(argc,argv);
  //Initialize the OpenSG Scene
  scene = createScenegraph();
  //Setup Renderability
  renderAction = RenderAction::create();
  //Create Passive Window
  mWin = PassiveWindow::create();
  //Add Passive Viewport
  mWin->addPort(mView);


  glutMainLoop();
  return 0;
}
```

Java Motor Control Program

```
//Written By Jesse Lane and Adam Bogenrief

import java.awt.geom.Arc2D.Double;
import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.IOException;


public class PhidgetServoControl
{
   private String[] servos;

   public PhidgetServoControl(String path)
   {
      servos = new String[4];
      for(int i = 0; i < servos.length; i++)
      {
         servos[i] = path + "servo" + Integer.toString(i);
      }
   }

   public void writeServo(int servoNum, double position) throws IOException
   {

         int pos = (int)position;
      if(servoNum >= 0 && servoNum < 4)
      {
         PrintWriter writer = new PrintWriter(new FileWriter(servos[servoNum], false));
         writer.print(pos);
         writer.flush();
         writer.close();

         //System.out.println("Wrote " + pos + " to servo" + servoNum);
      }
   }
}
```

Java Main Program Run on VADR

```
import java.io.IOException;
import java.net.SocketException;

//Program written by: Adam Bogenrief
//April 2006
//Program combines serial, socket communication, and motor control


public class AdamsVadrControl {

	public static void main(String[] args)
	{
		//Two choices for path to be, cause gets reset everytime on bootup
		String path1 = new String("/sys/bus/usb/drivers/phidgetservo/2-2.1:1.0/");
		String path2 = new String("/sys/bus/usb/drivers/phidgetservo/3-2.1:1.0/");


		/**************Make a socket to send stuff******************/
		//Make local variable
		UDPClient sender = null;
		try
		{
			//Assign it to destination
			sender = new UDPClient(1061);
		}
		catch (SocketException e)
		{
			e.printStackTrace();
		}

		/**************Make a socket to receive stuff******************/
		//Make local variable
		UDPServoServer receiver = null;
		try
		{
			//Try Path 1
			receiver = new UDPServoServer(1060,path1); //listening port and path to phidgets
		}
		catch (SocketException e)
		{
			//Reset receiver
			receiver = null;
			try
			{
				//Try path2
				receiver = new UDPServoServer(1060,path2);
			}
			catch (SocketException e1)
			{
				// print error to screen
				e1.printStackTrace();
```

```
                }
        }
        /**************Loop to continuously send and receive******************/

        //boolean to activate while
        boolean value = true;

        //While makes a loop
        while(value)
        {
                //make receive attempt
                //boolean to set equal to receive call
                boolean getr = false;
                try
                {
                        if(receiver != null)
                        {
                                        getr = receiver.recv();
                        }
                        else
                        {
                                System.out.println("receiver is null");
                        }
                }
                catch (IOException e)
                {
                        System.out.println(e);
                }
                //make send attempt
                boolean gets = false;
                try
                {
                        gets = sender.send();
                }
                catch (IOException e)
                {
                        e.printStackTrace();
                }

        }

    }

}
```

BIBLIOGRAPHY

1. Calvert, Kenneth L. (2002). TCP/IP Sockets in Java: Practical Guide for Programmers. San Francisco: Morgan Kaufmann Publishers.
2. Clark, Dennis and Owings, Michael. (2003). Building Robot Drive Trains. New York: McGraw-Hill.
3. Cohoon, James P. and Davidson, Jack W. (2004-2006). Java Program Design 5.0. New York: McGraw-Hill.
4. Devantech Ltd. http://www.robot-electronics.co.uk. accessed 21 January 2006. accessed for SRF-04 sensors.
5. Digikey Corporation. http://www.digikey.com. accessed 21 January 2006. accessed for quadrature encoder hardware.
6. Gillespie, T. D. (1992). Fundamentals of Vehicle Dynamics. Danvers, MA: Society of Automotive Engineers Inc.
7. Gofton, Peter W. (1986). Mastering Serial Communication. Berkeley: SYBEX.
8. Grange, S., Fong, T. and Baur, C. (2000). Effective Vehicle Teleoperation on the World Wide Web. IEEE International Conference on Robotics and Automation, 2, 2007-2012.
9. Harold, Elliotte R. (1999). Java I/O. Sabastopol, CA: O'Reilly.
10. Harold, Elliotte R. (2005). Java Networking Programming. Sabastopol, CA: O'Reilly.
11. Hill, F. S. (2001). OpenGL Programming Guide. Upper Saddle River, NJ: Prentice Hall.
12. Laird, R. T., Bruch, M. H., West, M. B., Ciccimaro, D. A. and Everett, H. R. (2000)

Issues in Vehicle Teleoperation for Tunnel and Sewer Reconnaissance. Proc. of the Vehicle Teleoperation Interfaces Workshop, IEEE International Conference on Robotics and Automation, 2, 1900-1908.

13. Lee, S., Sukhatme, G., Kim, G. and Park, C. (2002). Haptic Teleoperation of a Mobile Robot: A User Study. Intelligent Robots and Systems, 3, 2867-2874.
14. Mini-ITX. http://www.mini-itx.com. accessed 21 January 2006. accessed for computer and computer accessories.
15. Olivares, R., Zhou, C., Bodenheimer, B. and Adams, J. (2003). Interface Evaluation for Mobile Robot Teleoperation. ACM South East Conference.
16. Omnitech Corporation. http://www.omnitech.com. accessed 15 January 2006. accessed for teleoperation examples.
17. Phidgets Inc. http://www.phidgets.com. accessed 20 January 2006. accessed for Phidgets hardware.
18. Savage Innovations. http://www.oopic.com. accessed 20 January 2006. accessed for OOPIC hardware and software guides.
19. SensAble Technologies. http://www.sensable.com. accessed 20 January 2006. accessed for programming guides and Phantom hardware.
20. SensAble Technologies. (1995-2000). Programmers API. Woburn, MA: SensAble Technologies.
21. SensAble Technologies. (1995-2000). Programmers Guide. Woburn, MA: SensAble Technologies.
22. Shreiner, D., Woo, M., Neider, J. and Davis T. (2004). Computer Graphics Using Open GL. Boston: Addison-Wesley.

23. Sun Microsystems Inc. http://java.sun.com. accessed 8 June 2006. accessed for programming guides.

24. Wilks, Ian. (1994). Instant C++ Programming. Chicago: Wrox Press Ltd.

ACKNOWLEDGEMENTS

I guess I had better start off by thanking my parents, to whom this thesis is dedicated to. Most especially thanks to my mother Kathy, for all the support and understanding. I really do love you and wish I could have come home more, but as we all now know, theses are very involved. And to my father Greg, hope I did you proud.

Thanks to Dr. Luecke for funding this little project and helping me with all my many questions along the way. And to give credit where it is due, thanks to my fellow lab nerds. Thanks to Jesse Lane for being my main programming guru and all around computer god. Thanks to Kevin Godby for being my general computer answer guy who seems to know just about everything there is to know about the Linux or Windows operating systems. Thanks to Ethan Slattery for the SSH idea, without it, this project would not perform nearly as well. And finally, thanks to John Burnett for being my moral support guy who never let me quit, even though I wanted to.

And since I am being thorough, I'd better thank VADR. For all the knowledge I just gained. I learned so much about computers, Linux, C++, Java, networking, and interfacing hardware. Well, enough to let me know that I still have a lot more to learn.