Handheld Controller for use in a Virtual Environment

MQP

Final Report

Worcester Polytechnic Institute

Roger Burns

CS 2007

rburns07@wpi.edu

Nick Wirth

ECE 2007

nwirth@wpi.edu

Advisors:

Robert Lindeman (CS)

gogo@wpi.edu

Susan Jarvis (ECE)

sjarvis@wpi.edu

Abstract of the Final Report for the Virtual Environment Handheld Controller

By

Nicholas Wirth

Roger Burns

The purpose of this project was to design a new handheld Virtual Environment controller. The controller was designed based on research regarding previous designs. This research started with the mouse and the first Atari Joystick, and progressed to encompass the new Nintendo Wii controller. This provided a sense of technology to use, where it had been used before, and how it could be applied in the future. From that point it was a matter or designing the controller based upon a set of specifications that were drafted to provide guidelines and direction. These included the functionality of the controller, the specific components needed to provide that functionality as well as how the controller would communicate with a personal computer. To aid in the implementation of the controller, a virtual environment was created to test the various inputs the controller provided. After mapping the controller functions to the various components, these were tested in the environment, looking specifically at navigation and object manipulation. Other aspects were also incorporated, such as a modal design, where based upon button presses, different components on the controller can have different purposes. Future work includes the expansion of user testing to provide an idea of feasibility against competing controllers as found in research. The flexibility of the environment as well as the controller allow for adaptation to certain tasks or alternative mappings of the controls to functions. Because the controller at the moment is not in a hand held form factor, it is also recommended that a housing be provided at some point in the near future to lend to possible market venture.

Table of Contents

List of Figures

List of Tables

1. Introduction

The purpose of this project is to design, create and test a hand held controller that interfaces with a computer. This allows a user to interact with a simulated environment. Just as someone moves through a room by walking, the controller provides a user the ability to navigate through the simulated space. Other actions that can be done in real life are possible in the environment. A person moving through a bedroom may wish to choose amongst a pile of pictures lying on a desk, pick that picture up and examine it. The project's aim is to provide that level of interaction using simple hardware that has been in the hands of gaming enthusiasts for years. The addition of new technology will allow an increased ability to interact with the system and provide the ability to create a life-like experience while interacting with these computer simulations. The technology this project hopes to encapsulate includes a joystick, a trackball, an accelerometer, scroll wheels, as well as buttons. Their purposes and functions will be discussed later.

2. Project Background

2.1 Comparative Products

It was necessary to research other products in the controller market to help create the product specifications to meet expectations of potential users. Looking at the history of virtual reality controllers provided an explanation as to how they developed over time and which of their specific features impacted success. Some controllers have become commonplace while others have been passed over and ignored by the public. Current devices were reviewed to determine benchmarks in performance that must be met to be competitive in the market.

2.1.1 Controller History

In the past, controllers were designed to provide interaction with the computer systems people were using. The first computer mouse was designed and implemented in 1965 by Stanford Research Laboratory [13]. It was a simple device providing the user with a unit that would scroll a selector/manipulating graphic on a computer screen. This added versatility to the computer world and eventually would become a common component when the first desktop environment was created. The concept of playing a game represented in 2D space was seen in the gaming world when in 1972 Atari was founded and released its groundbreaking game, Pong. How does one manipulate objects on a screen so that they can interact with each other? The progression of devices and their uses started with this simple question, and moved on to evolve into the controllers we use today [14].

In 1976, Atari introduced the joystick, a simple box with a stick that provided the ability to distinguish between eight directions (Figure 1), making it possible to move an object selector around along horizontal, vertical and diagonal axes. Later Atari introduced a joystick that provided motion control in 360 degrees. A keypad with button inputs was also added and would allow a game to become more complex, providing a means for multiple inputs. This was soon to be replaced by a common symbol of games and gaming for years to come [14]: the gamepad.

Figure 1 : Atari Joystick

Nintendo, currently one of the largest game console companies, designed a simple rectangular controller to be used with their Nintendo Entertainment System in 1986 (Figure 2). It incorporated a simple four way directional pad. This mimicked the ability of the joystick, but incorporated it into a small button system taking stress away from the wrist and hand. They also used two buttons to manipulate the in-game characters and objects, and two other buttons for menu navigation. This simple layout was a design that would be expanded and upgraded as games required more-complex input [14].

Figure 2 : Nintendo Controller

Companies changed the basic design of the controller as time progressed and the technology available to the gaming market advanced. Designs that succeeded were basic yet provided great functionality. Most controllers soon consisted of some sort of ergonomic curve to better fit the hands of gamers and relieve stress from long durations of use. The world of gaming encapsulated movement through a supposed 3D environment, most often the case with first person shooters, and controllers began using analog joysticks to provide a form of movement and view control. This was apparent with the arrival of the Nintendo 64 controller in 1996 [8] (Figure 3).

Figure 3 : Nintendo 64 Controller

The following generations of controllers provided not only directional pads and buttons, but pressure sensitive triggers, accelerometers and analog sticks. The versatility of these devices is wide, but is limited to an environment largely used by games. Pressure sensitivity was incorporated in the previous generation of controllers in the gaming market, namely the PlayStation 2 controller and the Xbox controller (Figure 4).

Figure 4 : PlayStation 2 and Xbox Controllers

The methods by which someone manipulates perspective or movement with these controllers may not be tailored to specific applications, but because of generalization these controllers provide functionality for a wide range of software supporting different forms of manipulation. This method of creating a controller out of multiple simpler controls is essential for a device to be able to work with hundreds of applications, requiring only remapping for increased functionality.

This segues into the personal desktop world where two devices have dominated the market for years: the keyboard and mouse. They have been used for games, graphics design, and engineering. They are even used to emulate game console controllers in order to play comparable games. They are versatile and provide the necessary inputs to navigate through a 2D environment [9].

2.1.2 Current Competition

Regarding products that are currently on the market, two main categories of controllers are available: high-end, VR-specific devices, and consumer-level controllers aimed at video games. Both groups are used for interaction with a virtual environment, but they serve distinct purposes. The high-end controllers provide a large degree of motional freedom, essential for complex environments, but they will only function with proprietary software and cost a large amount of money. For example, the Flock of Birds motion tracking system provides six degrees of freedom with magnetic sensors, but prices for such a system start at $2,500 [20]. The gaming controllers are reasonably compatible with various systems, and are affordable, but have limited input options to keep price down. Our goal is to bridge the gap between these two product groups and create a device that will provide enough control for a virtual environment, yet will be functional and affordable for the gaming market as well.

One example of a device that attempted to achieve this goal is the Spaceorb 360 [10] (Figure 5).

Figure 5 : Spaceorb 360

The ball on this device allows for six degrees of movement, for traveling within a virtual environment or video game. It combines the functionality of multiple input devices into one, but was only mildly successful. In a review by Jason Bergman, the Spaceorb was tested with multiple computer games and commented on for its functionality and ease of use. Bergman comments, “The strange ball affixed to the SpOrb reacts beautifully in Descent, and really provides a clear advantage over any other input device” [2]. For specific uses, such as this flight-based game, the controller has a distinct benefit over traditional gamepads. However in a first-person shooter game he goes on to say, “When all's said and done...sure you *can* play Quake with the SpOrb...but *why?!?!?*. There really isn't any major advantage to it, it has a really steep learning curve....” This shows that outside of a few specific applications, the controller's difficulty of use outweighs its increased control. While designing our controller it was necessary to keep in mind a wide range of uses so as not to pigeon hole the device for a small number of applications.

Another device that has recently come to market is the Wii Remote (pronounced "wee") from game console manufacturer Nintendo. This controller makes use of a number of features including accelerometers and infrared technology to track its motion in 3D space (Figure 6).

Figure 6 : Nintendo Wii Remote

This device is one of a small number of controller devices using accelerometers to track motion of the device as a user input. Reviews for this device have been positive, generating a wave of new games focused primarily on the use of the new controller. Our project plans to employ a similar accelerometer technology (along with a unique combination of other inputs) in a device that will be usable by any computer with a USB port.

2.2 Component Selection

This section will provide background on the technology behind the various components of the controller. The chosen sensors will be analyzed, providing explanations of their operation, and the signals they will generate. It is necessary to understand how these signals are produced in order to effectively troubleshoot them during testing, as well as determine how to best handle the data they produce for translation into a virtual environment.

2.2.1 Accelerometers

One of the devices that is to be built into the controller is an accelerometer. This device measures its orientation to gravity along three axes as it is turned along the x and z axes, and moved along the y axis. These three movements output a proportional voltage used to measure the relation to gravity or acceleration. This device is useful as an input sensor, because the user can tilt and move the controller around in order to perform a task such as locomotion. Because the accelerometer is internal to the controller, the user's fingers are free to operate other controls simultaneously, and more readily interact with their virtual environment. In our application, the accelerometer will be used to measure the force of gravity. As the controller is tilted, the accelerometer's orientation varies between perpendicular and parallel with the force of gravity. When parallel with the force of gravity, a force of 1g (or $9.8m/s^2$) is registered with the accelerometer. In the perpendicular position, 0g is measured. The variation in this measured force can be used to calculate the angle at which the device is oriented. By employing three orthogonally mounted devices (or one 3-axis device), the controller's attitude comprised of its yaw, pitch, and roll can be calculated (*Figure 7*).

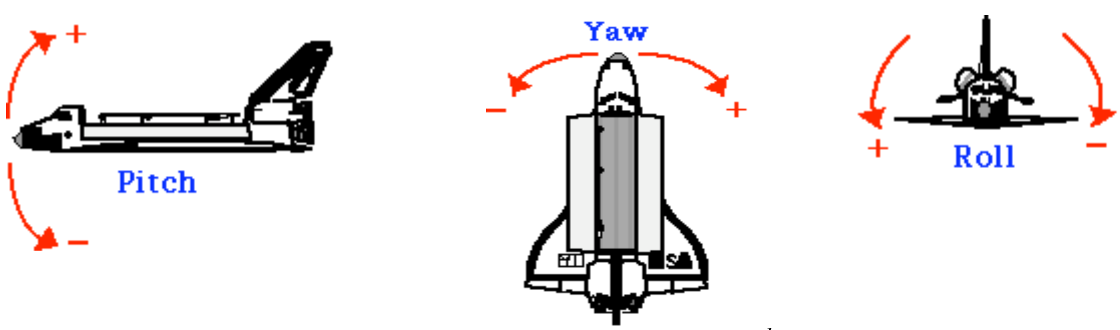


Figure 7 : Pitch, Yaw, Roll[1]

This method of operation allows the accelerometer to output values which describe the tilt of the controller at any time, providing a means for fluid motion control. The use of accelerometers also allows the controller to measure the absolute three-dimensional position of itself with a different interpretation of the data. If the

[1] http://liftoff.msfc.nasa.gov/academy/rocket_sci/shuttle/attitude/pyr.html

acceleration data from the controller is collected and stored over time, the speed, and position of the controller can be obtained by calculating the first and second integral of acceleration with respect to time. This alternative method of operation allows the device to be used in multiple ways with only changes in software.

Over the last decade, Microelectromechanical systems (MEMS) have advanced a great deal, allowing once-bulky mechanical devices to be manufactured into tiny integrated circuits. Currently, multiple MEMS accelerometers are on the market, such as Analog Devices' ADXL330, which simultaneously measures 3-axes of force with a sensitivity of 300mV/g [7], and occupies a footprint of only 4mm x 4mm. With today's mass manufacturing of integrated circuits, this device costs much less than a traditional mechanical accelerometer at $5.45 per unit (at 1,000 units). These advances in MEMS technology will allow the controller to provide more methods of input compared to previous controllers without significantly increasing its price, or complicating its design.

2.2.2 Optical Encoders

One common electromechanical device used for translating rotational movement into an electrical signal is an optical encoder. This device is used both in the operation of a trackball, and a scroll wheel, which are elements found in the design of this controller. An image of a typical optical wheel encoder can be seen in Figure 8.

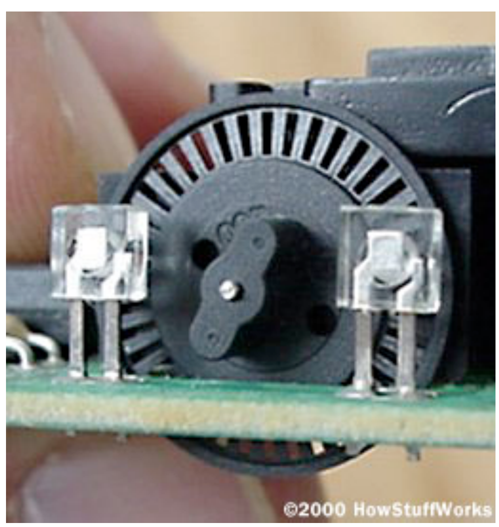


Figure 8: Optical Wheel Encoder

In this trackball example, rotating the ball turns two shafts through friction, one for the X-axis and one for the Y-axis of movement. A slotted disc is located at the end of each shaft (only one wheel is pictured above). As the wheel turns, the slots pass by two optical sensors (the clear plastic squares in Figure 8). On the opposite side of the wheel, two light sources are pointed at the sensors. If a slot lines up with the sensor, it detects light and outputs a digital “1”. If the sensor is blocked by the wheel, it cannot see the light and outputs a digital “0”. By counting the number of light pulses as the wheel turns, the position of the wheel (and indirectly the ball) can be calculated[2].

There are two sensors on each wheel in order to determine the direction of rotation. The sensors are positioned so that when one is lined up with a slot, the other is in transition between slots. This offsets the signals from the two sensors as shown in Figure 9.

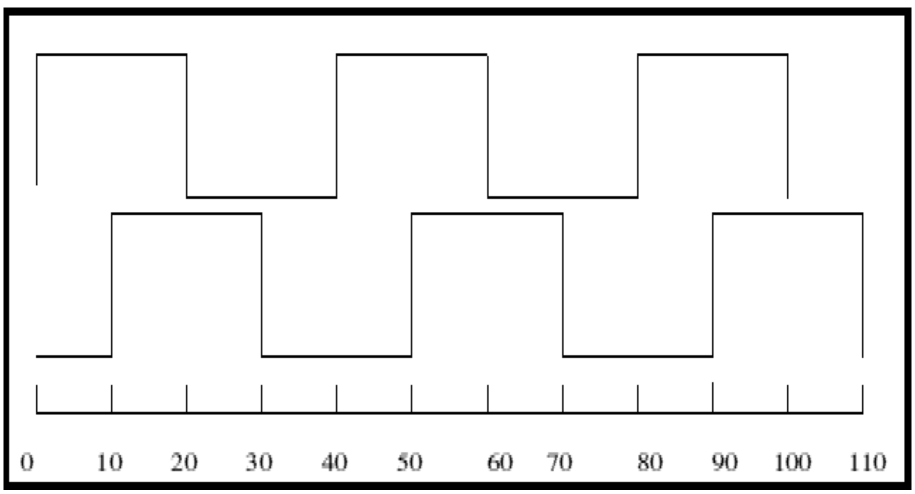


Figure 9: Optical Encoder Waveform[3]

When a “1” is sensed by the controller's processor from the first sensor, an immediate check of the second sensor will determine the direction the wheel is turning. This system necessitates that there be two sensors present for each axis of rotation to be monitored. For a trackball, four optical sensors will be needed. For a scroll wheel, only two sensors will be needed. Optical encoders are simple to implement, because they output a digital signal, which is easier to work with than an analog signal when

2 http://www.4qdtec.com/meece.html

3 ibid

performing logic in a microprocessor.

2.2.3 Potentiometers

Another common electromechanical device for measuring movement is a potentiometer. A potentiometer acts as a variable resistor, whose value changes with the position of a shaft. A typical potentiometer can be seen in Figure 10.

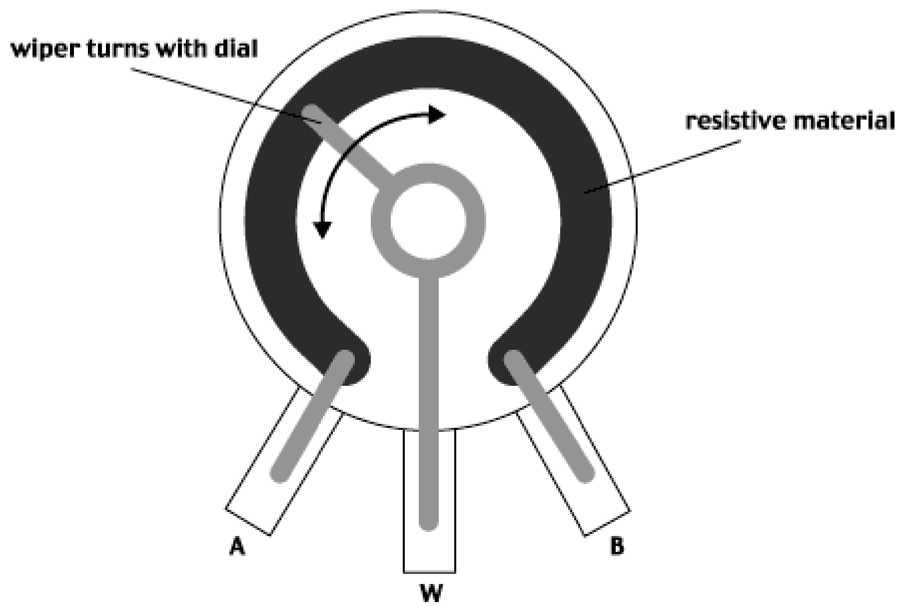


Figure 10: Generic Potentiometer[4]

As the wiper is turned, the resistance from point A to point W (as well as from point B to point W) changes. This can be useful in an analog circuit, where a varying resistance can be translated into a varying voltage, and input into a microprocessor.

This type of input sensor is commonly seen in a joystick. Potentiometers are attached to the ends of two rotating shafts in the base of the joystick. As it is moved, its displacement in the X-axis is recorded by one potentiometer, and its displacement in the Y-axis by the other. This system can be seen in Figure 11.

4 http://www.markallen.com

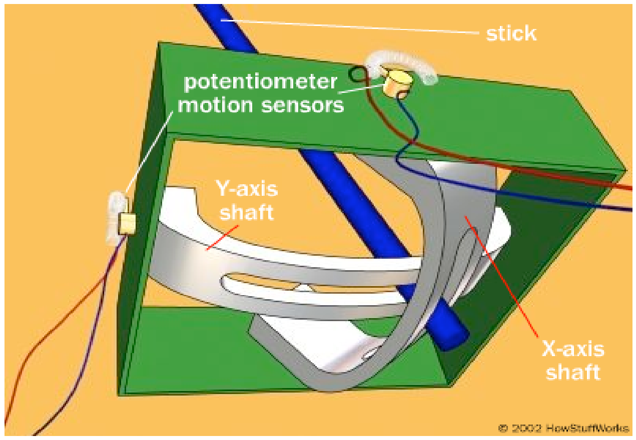


Figure 11: Joystick Mechanics

A joystick also typically has internal springs that force the stick back to its center position when released, and outputs a varying voltage for each axis of rotation. These analog signals require the microprocessor to contain an analog-to-digital converter in order to create usable position data.

2.2.4 Digital Buttons

One of the simplest electromechanical devices that will be integrated into the controller is a digital button. A button works to momentarily complete a circuit, resulting in a digital "1". When the button is released, the signal returns to a digital "0". The symbol for a momentary button is shown below in Figure 12, illustrating its simple operation.

Figure 12: Standard Button Symbol

Buttons are useful for initiating a predefined action in a virtual environment. They are easy to design into a system compared to other components because they require little additional circuitry, and processing logic requires little code. For this application, button debouncing, the act of accounting for jitter in an electrical switch, was not necessary, as the digital buttons do not function on interrupts. The microprocessor is set to poll the buttons' status whenever it sends data to the PC.

3. Project Specifications

In order to be a successful design, a VR controller must meet a set of pre-defined specifications. These specifications will detail the tasks the controller must be able to accomplish. Each of the defined specifications must be clearly measurable in order to determine if it has been met. After the design was completed and the controller had been built, a series of tests were run to verify these specifications.

The overall goal of the controller is to allow a user to interact with a three-dimensional virtual environment. The design of this controller focuses on three actions determined to be the most critical in a virtual environment; 3D navigation, object selection, and object manipulation. The controller should be tailored for these types of actions so that the user can easily multi-task and feel as if he is in the simulated environment. **Table 1** lists a number of tasks to be performed in a virtual environment, and matches each to possible sensors that would be well suited to accept user input.

Task	Possible Hardware
Navigate (Front, Back, Left, Right)	Accelerometers
Navigate (Up, Down)	Possible accelerometers
Object Selection	Scroll wheel / Accelerometer
Object Manipulation	Trackball, Analogue Joystick, Accelerometer
Menu Call	Button
Menu Control (Up, Down)	Scroll Wheel (clicking roll Up, Down)
Menu Control (Left, Right)	Scroll Wheel (left and right click)
Selector Manipulation	Track Ball
Point of View Manipulation	Analogue Joystick

Table 1: Task vs. Hardware

Another way of viewing this information is to start with a sensor type, display its attributes, and determine a task that fits those characteristics. This format is shown in **Table 2**.

Sensor	Input Dimensions	Advantages	Disadvantages	Possible Tasks
Trackball	2D	Precision, absolute position	No continuous movement	Object selection/manipulation
Joystick	2D	Rate of movement, automatically centers	Poor precision	Object selection/manipulation
Scroll Wheel	1D	Discrete points in movement	No continuous (smooth) scrolling	Object "depth" selection
Accelerometer	3D	3D input, doesn't occupy fingers	Poor precision	Movement
Binary Button	0D	On/off functions	Few input dimensions	Selection, map to function
Analog Button	1D	Amplitude Control	Poor precision	Rate control

Table 2: Sensor Attributes

Using Tables 1 & 2, the conclusion was made that multiple device types are necessary to achieve the desired functionality of the controller. By implementing a combination of these inputs in a package with which the user can efficiently and easily accomplish all above-mentioned tasks, the result should be a useful product that will have a distinct place in the PC and VR-controller markets.

The following list of specifications must be met in order to create a competitive controller that allows the user to perform 3D navigation, object selection, and object manipulation:

1. Must provide an input for movement in a 3D environment
2. Must provide an input for selecting of an object in 3D space
3. Must provide an input for interaction with a selected object
4. Should allow at least two actions to be performed simultaneously
5. Usable after five minutes of training
6. USB Compatible
7. Software configurable (input sensors can be remapped for different applications)

In order to assure these specifications have been met, a series of tests was performed that resulted in a yes/no or numerical value for each item on the above list. These procedures are detailed in the results and analysis section of the report.

4. Project Design Overview

The design process and specifications set the background of the project. This section will explore the specific hardware and software components of the design for the prototype controller. An overall system flow will be presented, along with discrete device choices.

4.1 Hardware

The hardware design of the controller begins with a functional block diagram (Figure 13) displaying the major system components and the interactions between each component, the user, and the software.

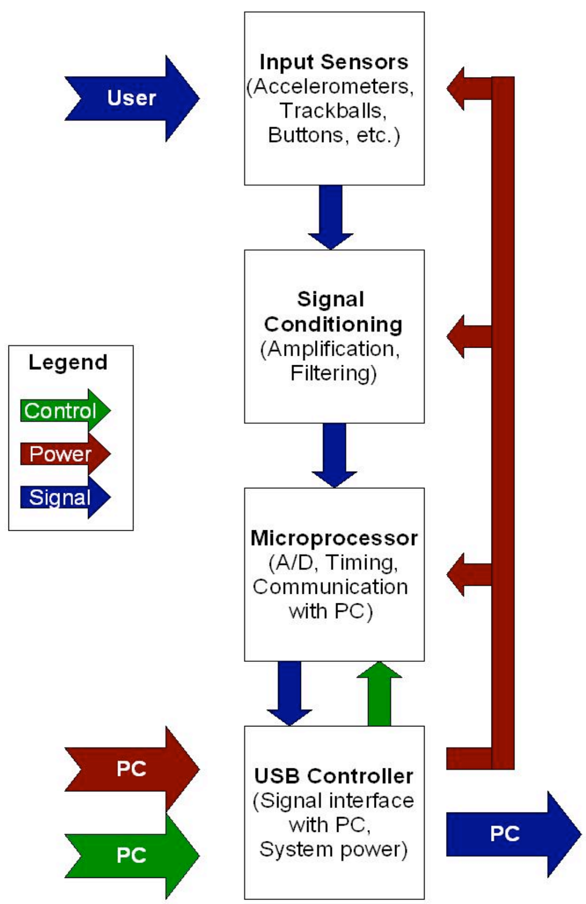


Figure 13 : Hardware Design

The block diagram consists of four major components, the input sensors, signal conditioning, a microprocessor, and a Universal Serial Bus (USB) controller. The arrows represent the flow of information and power. The signal data originates from the user, and is transformed into electrical signals by the sensors. These signals are then conditioned into a form that allows them to be compatible with the microprocessor. The processor takes these analog (and digital) signals and converts them into digital information. The microprocessor synchronizes its output data with the USB controller's clock. The USB controller acts as a translator between the microprocessor and the PC, allowing them to exchange data. The PC sends control data back through the USB controller to the microprocessor, in order to change modes of operation. The PC also supplies power to the entire circuit through the USB connection. All of the blocks shown in this diagram will be contained within the body of the controller, and connected to the PC with an external cable.

4.1.1 Sensors and Signal Conditioning

Through a combination of background research on previous virtual environment controllers and input sensor analysis, it has been determined that to best achieve the specifications for interaction with a virtual environment this hand-held controller will contain the following sensors:

- 1 - 3-axis accelerometer
- 1 - trackball
- 1 - joystick
- 4 - digital buttons
- 2 - scroll wheels

All of the input devices result in a total of 17 separate signals that need to be interpreted by the processor and sent to the computer. Table 3 summarizes these inputs.

Device	Analog Signals	Digital Signals	Total Signals
Accelerometer	3	0	3
Trackball	0	4	4
Joystick	2	0	2
Scroll Wheel (2)	0	4	4
Digital Button (4)	0	4	4
	5	**12**	**17**

Table 3: Device Outputs

The accelerometer will be housed within the body of the controller, and mounted directly to the main Printed Circuit Board (PCB). The product chosen to fulfill this task is the ADXL330 Accelerometer produced by Analog Devices Inc. This MEMS device is capable of measuring up to 3.6g in 3D space, and is contained in a 4mm x 4mm Lead Frame Chip Scale Packaging (LFCSP). A pin out of the device is displayed in Figure 14.

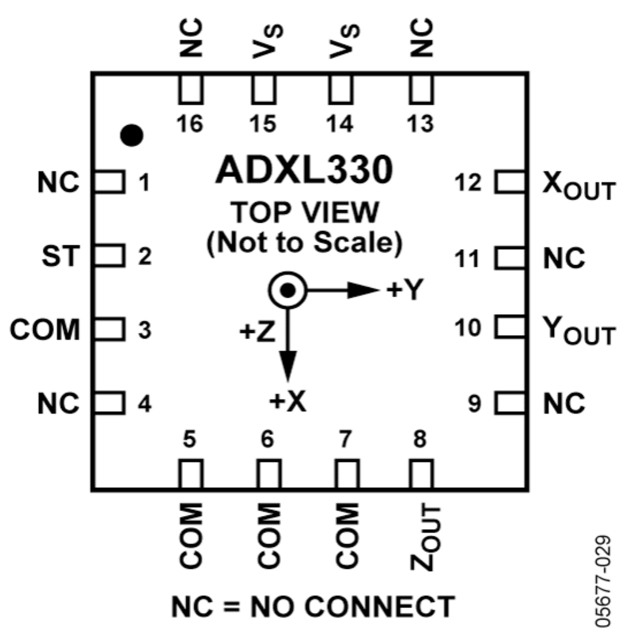


Figure 14: ADXL330 Diagram[5]

The ADXL330 requires a supply voltage between 2.0V and 3.6V. The 5V

5 Analog Devices Inc.

supplied by the USB port will be sufficient after being attenuated by a voltage regulator circuit. The device supplies three analog outputs corresponding to the 3 axes of movement (X,Y,Z) which are interpreted by the analog-to-digital converter in the microprocessor. The manufacturer states common applications of this device to be "Motion and Tilt sensing in Mobile Devices" as well as "Motion-Enabled Gaming Devices", which both closely describe the goal of this controller. At a price of $5.45 (@ 1,000pcs.), this device is easily attainable for a low budget project.

The trackball operates with the use of optical wheel encoders measuring its X and Y axis movement. This results in the device having 4 digital connections to the microprocessor. The joystick is based on potentiometers and provides 2 analog signals to the microprocessor. Each digital button provides one digital input to the microprocessor, resulting in a total of 4 inputs. Finally, the two scroll wheels are based on one-axis optical wheel encoders, and output a sum of 4 digital signals.

For the prototype controller, all of the input sensors except the accelerometer are sourced from existing devices, as complete sensors are generally not available from manufacturers. For example, rotary encoders are available, but scroll wheels are not. Research shows devices such as joysticks, trackballs, and scroll wheels are specially made for products under large quantity contracts. The accelerometer however, is a bare sensor and was readily ordered from the manufacturer.

In order to properly interface with the microprocessor, all of the signals from the input devices must be properly conditioned. For digital signals, the amplitude of a logic "1" will be made sufficiently high to trigger each digital input. This value is the microprocessor's system voltage of 3V, which is higher than the input threshold voltage of 1.9V. If necessary, the signals are filtered to reduce false triggers resulting from overshoot or noise. For example, analog signals from the accelerometer are filtered with a simple RC low-pass filter consisting of a surface mount capacitor of 0.1μF, and the internal resistance of the sensors. This will result in more accurate digital data as the analog input signal will contain less noise. After being adjusted, the data is input to a suitable microprocessor.

4.1.2 Microprocessor

The microprocessor for this controller must meet a number of criteria in order to function properly:

- Minimum of 17 I/O pins for sensors
- Minimum of 5 A/D channels
- Operable on <= 5V supply
- Universal interface to communicate with USB controller.
- Moderate memory volume to hold program and sensor data
- Powerful processor to handle multiple data streams
- Small dimensions to fit in controller
- As few extraneous features as possible

Many solutions were researched and analyzed, ranging from simple 8-bit architecture microcontrollers to high-end DSP (Digital Signal Processing) capable chips. It was determined that a level of performance between these two extremes be chosen for this application. The low-end controllers are simple to program, require fewer resources, and cost less money, but they lack features, memory size, and processing speed. The high-end DSPs provide ample computing power, but are very complex to control, more expensive, and most of the chip's features would go unused.

The middle ground is a mid-range RISC (Reduced Instruction Set) based microcontroller, lacking DSP capability, but still containing the features necessary to monitor all the sensors and process the incoming data. A device based on Texas Instruments' 16-bit MSP430 family contains the required features and complexity necessary for this project. Analysis of the features within this family results in a choice of the MSP430x1xx line. Although smaller models contain enough I/O pins, a 64pin chip is necessary because with all 17 pins occupied (5 of which consist of analog inputs), the UART (universal asynchronous receiver/transmitter) interface pins of the smaller device would be unavailable. At a cost ranging from $5-$8, this processor fits comfortably in the project's budget. An additional benefit of choosing this processor is the availability of comprehensive developer kits that can be used for testing the device and becoming familiar with its interface. The final model chosen is the MSP430F169, as it is included in both the development kit, and the Softbaugh USB interface test board (discussed in the next section). In order to program and test the microprocessor, a Texas Instruments USB FET debugger board and IAR Embedded Workbench Kickstart software suite are utilized.

4.1.3 USB Controller

The USB controller must be capable of taking output data from the microprocessor, converting it into the USB format, and transmitting it to the PC. Additionally, the controller must also be able to convert any control signals sent from the PC to a serial format so that they can be recognized by the microprocessor. This process must occur at a sufficient speed such that the microprocessor can send data to the PC as fast as it is obtained. For example, a typical USB mouse operates at 125Hz as shown through Windows configuration settings.

Research into USB interface devices results in the choice of a USB Peripheral controller. This chip is designed specifically to organize communication between a USB host (such as a PC) and an attached device (such as the VR controller of this project). The FT232BM from Future Technology Devices International Ltd (FTDI) meets all these criteria, and provides additional features such as a low-power mode when the controller is inactive. Figure 15 shows an image of the device.

Figure 15: FT232BM[6]

6 Future Technology Devices International Ltd

This interface chip has the ability to power itself from the USB bus, a 5V source that can supply a maximum of 500mA. With the addition of a 5V voltage regulator, the USB port is capable of powering the VR controller's entire circuit. This USB peripheral device also has an attainable price of about $5.

4.1.4 Layout and Construction

Following the selection of all the controller's components, and the design of any supporting circuitry such as signal filtering, decoupling capacitors, or current limiting resistors, an overall circuit design was created. The system-level schematic of the controller is shown in Figure 16.

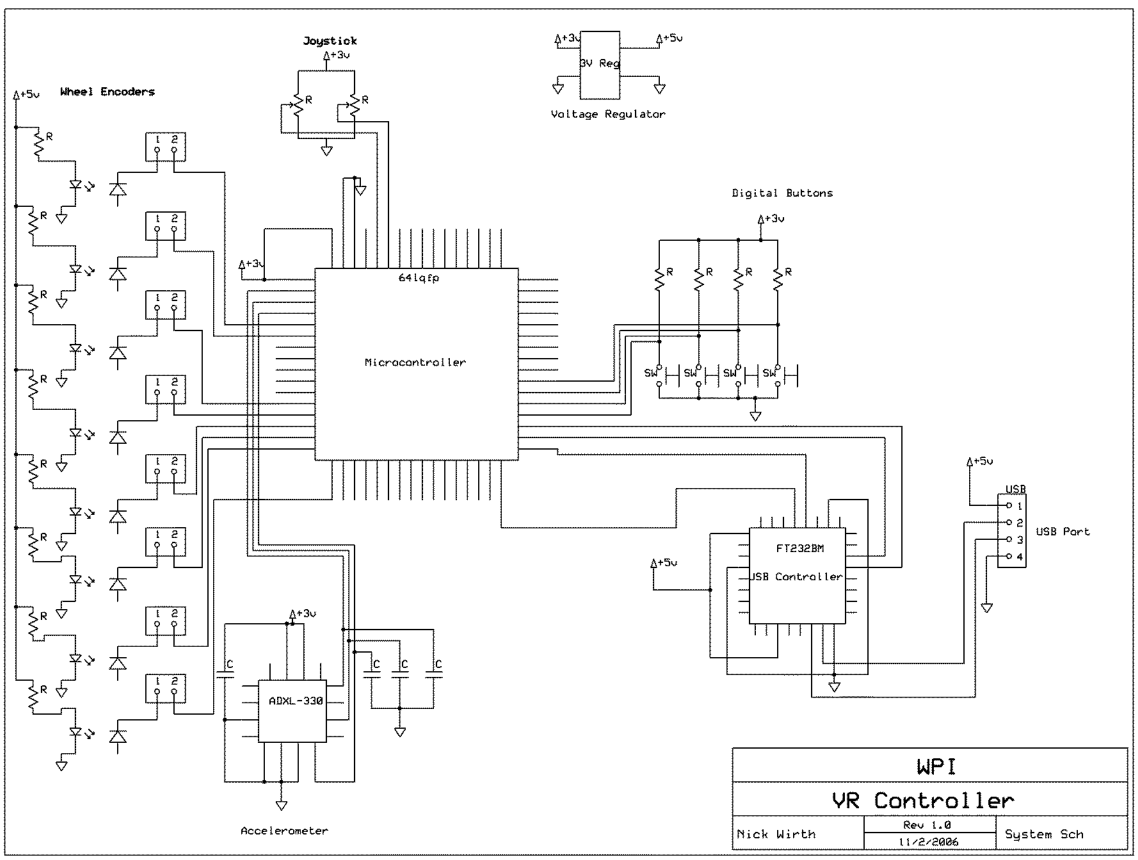


Figure 16: System Schematic

This circuit shows the connections between the sensors, and the microprocessor and includes all major components. The input from the accelerometers and the joystick are input to port 6 of the microprocessor, which are configured as a multi-channel analog to digital converter. The digital signals from the optical encoders are set up on port 2, which allows changing input signals to trigger an interrupt. The digital buttons are set up on port 1. What is not shown in this schematic are the details of the interface between the microprocessor and the USB controller. Each major section of the system-level schematic will now be examined in detail.

4.1.4.1 BFT232U169 test board

The original design of this project involved the use of a Texas Instruments based USB controller solution, detailed in *Appendix A*. After experiencing difficulty with that configuration, a switch to the FTDI chip was made. This allowed the use of the Softbaugh BFT232U169 evaluation board, pictured in figure 17, which formed the core of the system's circuit.

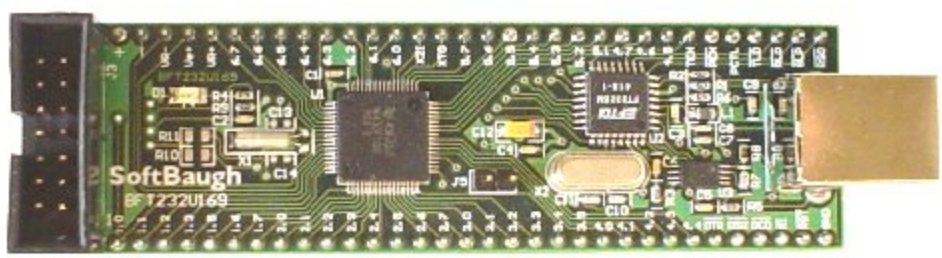


Figure 17: Softbaugh BFT232U169

This board includes both the MSP430F169 and the FT232BM of the design, with the interfacing circuitry fully constructed. A detailed schematic of this test board can be seen in *Appendix B*. This board is set up so that both the MSP430 and the FT232 are supplied with power and a clock crystal for proper operation, (32kHz and 6MHz respectively). A 93LC46B 1kb EEPROM chip is interfaced with the FT232, holding configuration firmware and USB identifier tags. A 5V to 3V voltage regulator is included on the board, creating a power supply for the controller. The black plastic 14-pin connector on the board is a JTAG connector which allows easy programming of the on-board MSP430. Finally, all the I/O pins of the microprocessor are accessible along the edge of the board via header pins.

4.1.4.2 Custom Printed Circuit Boards

The PCB was designed within the program ExpressPCB, as it provides adequate design flexibility, a direct board ordering feature and a relatively inexpensive source of boards. All of the traces, mounting pads, vias, and board layers can be mapped out in a fashion displayed in figure 18.

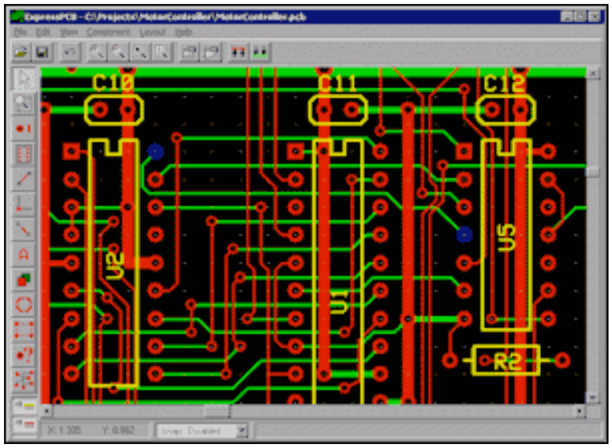


Figure 18: ExpressPCB CAD Software[7]

All dimensions can be directly adjusted by the designer, and custom templates can be created for specific devices to be attached. Once the board is designed, the resulting CAD (Computer-Aided Design) file is sent directly from the program to a manufacturer who creates the boards and ships them to the designer. Three copies of a simple 2-layer board can be purchased for about $50. With the acquisition of a PCB and all the system components, the board can be populated and tested for functionality.

The primary board designed for this project is the test board for the accelerometer, and its supporting passive components. Along with the chip itself, four frequency setting (and filtering) surface mount capacitors are necessary. An array of vias (small metal-plated holes through a PCB) are also placed along the edge of the board to facilitate easy attachment of wires. Some of the components such as the capacitors were laid easily through the use of templates in the software. The accelerometer on the other hand resides in a relatively new package, so it was necessary to manually create the template. Package dimension data was taken from the ADXL330 data sheet and used to determine pad sizes

7 http://www.expresspcb.com

and correct spacing. The layout for this board is shown in Figure 19.

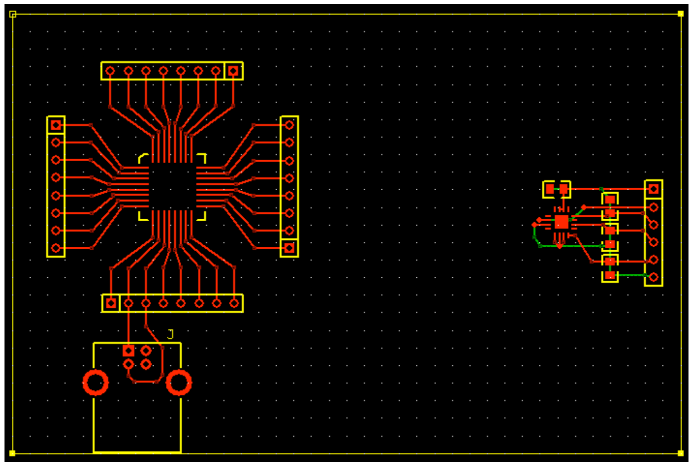


Figure 19: Accelerometer PCB

The accelerometer circuit is located on the right half of the board layout. The red traces represent conductive metal which will be placed on the top layer of the board. The green traces represent the bottom layer of the board. The yellow outlines of components would normally be printed as a silkscreen, but for the low cost manufacturing option, this board has no silk screen mask. The circuit on the left of the board is a pin-out for one of our tested USB solutions. Each pin is sent to a row of headers, and the USB data lines are sent to a USB B-style header for connection to a USB cable. This file was sent out to ExpressPCB, and after a wait time of about one week, the board was received. The fully populated board is shown in figure 20.

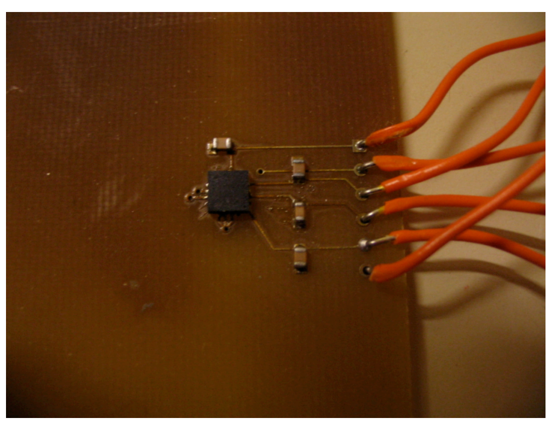


Figure 20: Populated Accelerometer Board

Due to the LFCSP package of the accelerometer and the small size of the capacitors, a non-standard method for populating the test board was used. Instead of using a traditional soldering iron, a water-soluble solder paste, Kester R276 shown in Figure 21 was used.

Figure 21: Solder Paste

Using the heating method found at seattlerobotics.org, the board was brought up to temperature in a small oven. It was held at the following temperatures for each specified amount of time.

- 4 min. 200 deg. Warm up board and allow temperatures to equalize.
- 2 min. 325 deg. Bring temperature up to saturation.
- 30 sec + 450 deg. Temperature raised until solder melts and beads at individual pins, then held for 30 additional seconds.
- Tap the oven before cool down

After this procedure, the water has evaporated from the solder paste, and the components are securely attached to the board. Lead testing reveals solid connections, and no short circuits. Lastly, the wires are manually soldered onto the board for breadboard interfacing.

4.1.4.3 Sensor Wiring

Apart from the accelerometer, the other sensors of the controller are taken from other human interface devices. This section will detail how their printed circuit boards are configured, and how they are connected to the microprocessor. The circuit board holding the scroll wheel is shown in Figure 22.

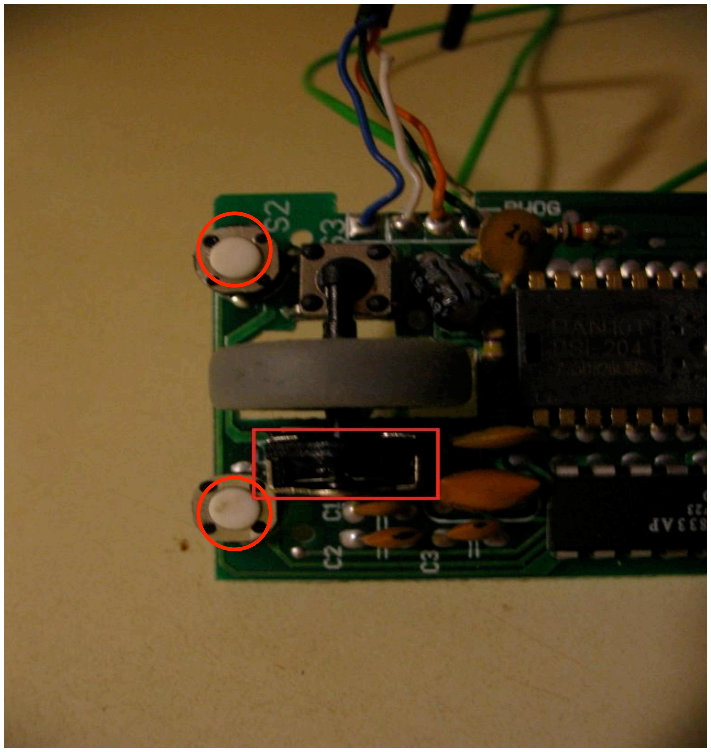


Figure 22: Scroll Wheel

The rotary encoder is highlighted with a red box. Four terminals extend through the printed circuit board, and perform the following functions:

- Power
- Ground
- Signal 1
- Signal 2

The power is provided with 3V from the system rail, and the ground pin is grounded. The signal 1 and 2 pins are routed via wires to the port 2 inputs of the MSP430. These signals represent the two square wave signals, separated by 90 degrees to determine the direction of rotation.

The red circles show examples of the digital buttons. The mechanics of each button are comprised of two conductive metal pieces on each end of the button. The white plastic button contains a metal pellet that connects the circuit. When the button is pressed down, the two halves are connected, and the circuit is completed.

The trackball also operates through the use of rotary encoders, with its board shown in Figure 23.

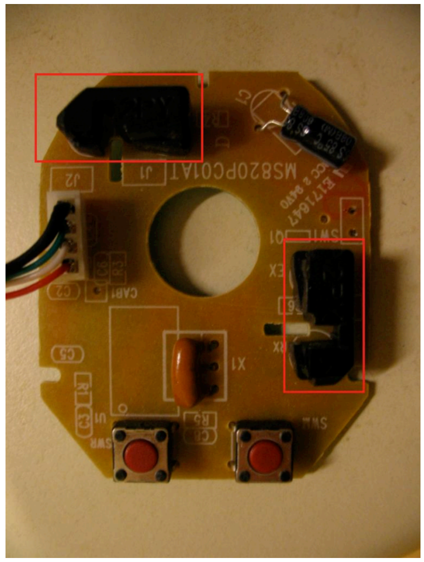


Figure 23: Trackball PCB

The two rotary encoders (one for X-axis and one for Y-axis) are highlighted with red boxes. The notched wheels that are rotated by the trackball fit into the slots in these encoders, and translate rotational movement into a stream of digital pulses. The underside of the board is shown in figure 24 to illustrate the connections.

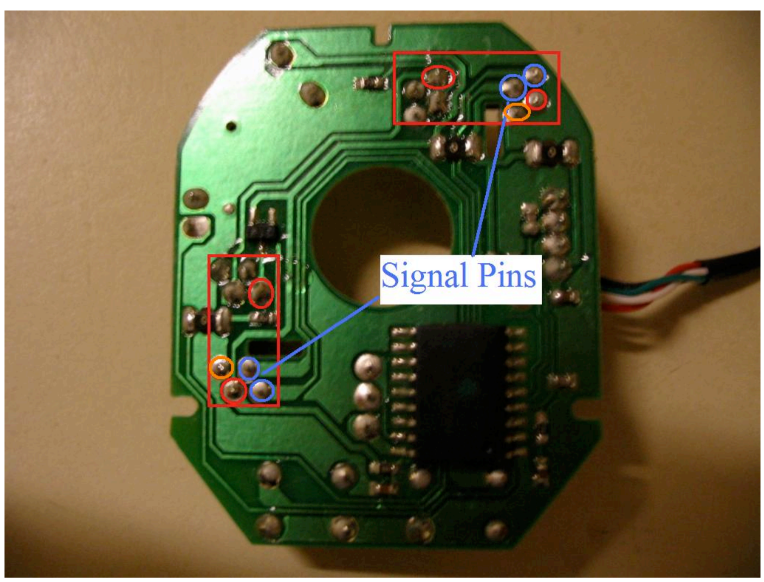


Figure 24: Trackball Wiring

The two red boxes (each containing eight pins) are the two rotary encoders shown in the previous figure. It can be seen that for each encoder, the pins are separated into groups of four. This is because the encoders are comprised of two pieces. An LED on one side of the device emits a constant light, while the other side is an optical receiver. The pins marked in blue are the signal pins for the receivers. The pins marked in red provide power to the encoders. The orange pins are the ground pins for the devices. All four signals pins are routed to the port 2 inputs of the MSP430 as digital signals.

The joystick and digital buttons are sourced from a disassembled X-box controller, shown in Figure 25.

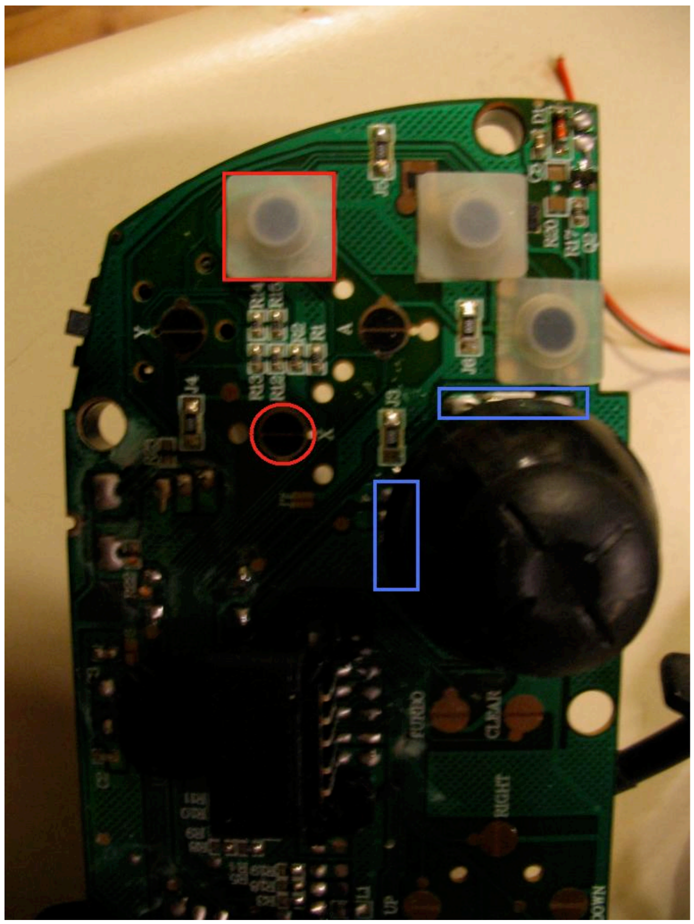


Figure 25: Joystick and Digital Buttons

The joystick is pictured on the right of the image. This device is based on two potentiometers, one for each axis of motion. Each potentiometer has three terminals, highlighted by the blue boxes. The two outer pins of each one are the power and ground pins. Because a potentiometer is essentially a variable resistor, these can be wired with either polarity. The center pins of each are the signal outputs, which are sent to port 6 (analog to digital converter) of the MSP430.

With the signal paths of all the individual sensors identified, a complete circuit was constructed. Each of the sensors was fixed with epoxy to a small sheet of polycarbonate, and connecting wires were soldered to data and power pins. The complete circuit is shown in Figure 26. This circuit allows testing of the devices as a whole, while retaining the ability to re-wire and add or remove components.

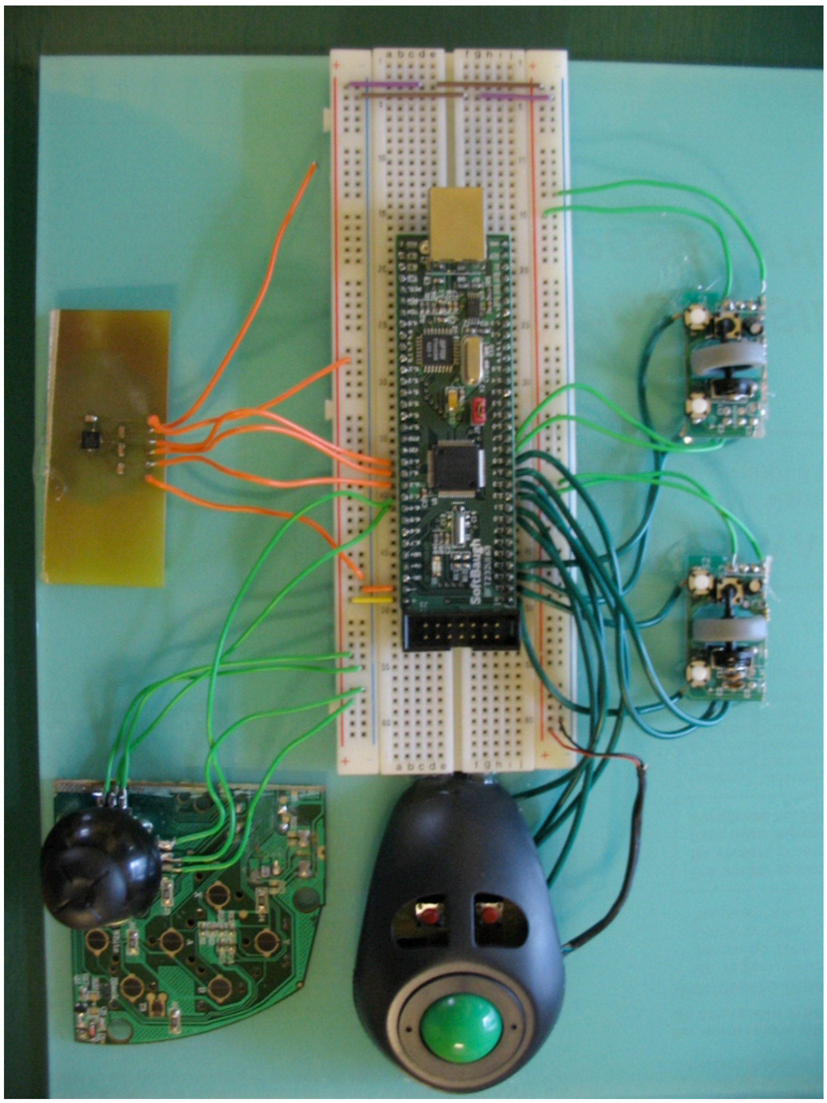


Figure 26: Complete Circuit

4.2 Software

A device must have software to interact with the computer system that it is connected to. This is a device driver. Drivers are files that describe to the operating system, the necessary functions to perform based upon the input received from a specific device. In looking to create a unique device, it was important to do work on a method to gather the information from the device within the software. When determining if the device is acting as expected, it is required to have a way to inspect if the information being provided to the personal computer is being interpreted correctly. The VE will provide the tools to check the information coming in and assist with the determination of hardware capabilities as well as standard input from the devices on the controller. This extends to a visual inspection of the interaction between the device and the computer as well a visual approach to determine the correct mapping of the functions to controller operation.

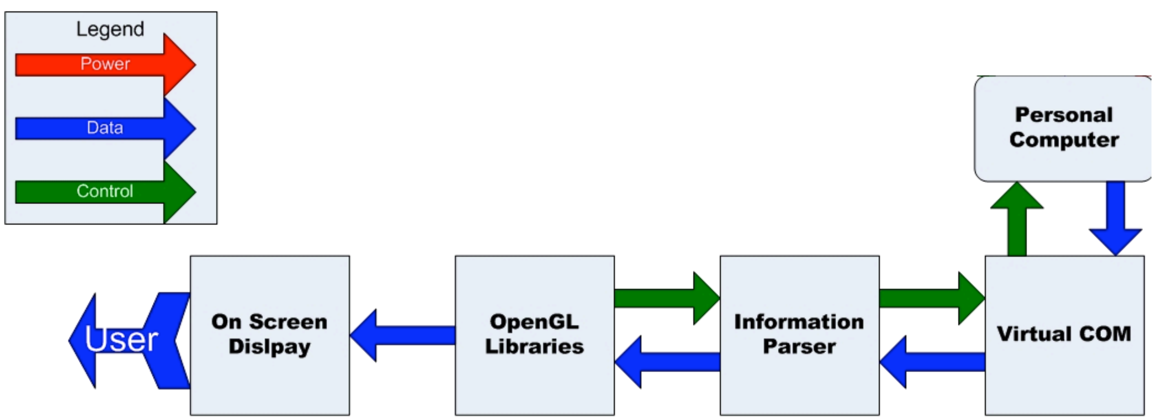


Figure 27: Software Design

Figure 27 represents the flow of information from the computer through a Serial port. The data is passed to the parser and separated into individual data fields that are interpreted by the OpenGL libraries and the VE. The screen displays the interpreted data and is a direct corollary to the input the user signaled by interacting with the device. The specifics of each of these blocks are explained below.

4.2.1 Virtual COM

The communication between the VE and the device will be handled in a serial fashion. While USB is being used for the hardware to communicate as well as provide power, the late implementation of USB software for the computer left communication over a COM port as the reasonable technology to use. The evaluation board used to provide USB communication provides device drivers that mimic a COM port on a personal computer. This is a Virtual COM Port Driver and allows access to the data as if opening a serial communication over COM in the VE. A baud rate of 9600 was set as the standard, but through trial and error with the device, the evaluation board only allowed the VE to receive information at 2400 baud. 2400 baud refers to the number of symbols per second received, in this case, over the Serial port. When connecting to the Serial port the software declares that it receives 1 bit per symbol. This translates to 2,400 bits per second. The information that is being passed is a stream of 19 bytes. On the windows architecture a byte corresponds to 8 bits. The stream then is 152 bits of information. Dividing the amount of information in the steam into the baud rate gives 2,400 / 152. This is the equivalent to a rounded down number of 15 samples of information per second. While this is low for the capabilities of the Serial port, it provides ample enough information to have a rough conveyance of user interaction with the device. Future work to improve this rate will improve the precision that the device can have as a higher baud would allow a greater sample speed.

4.2.3 Information Parsing

The information that is passed to the PC is in the form of a structured stream of data. This is passed whenever the PC sends a control signal to the device. The structured stream is passed to a parser object that splits the information into various data structures that handle the update of variables in the VE. The passed data handles the numeric values of the buttons, joystick, trackball, scroll wheels and the accelerometer. When the parser gets this information and parses it, it returns the updated variables to the VE. In turn the VE updates the visual representation according to the change as specified in the code. Since the information that is being parsed is in byte form, it was important to check the hardware description and determine the actual bits within each byte that are important. The information coming from the device followed the following protocol:

- Buttons (4): xxxxdddd
- Trackball X: dddddddd dddddddd
- Trackball Y: dddddddd dddddddd
- Scroll 1: dddddddd dddddddd
- Scroll 2: dddddddd dddddddd
- Accel X: xxxxdddd dddddddd
- Accel Y: xxxxdddd dddddddd
- Accel Z: xxxxdddd dddddddd
- Joystick X: xxxxdddd dddddddd
- Joystick Y: xxxxdddd dddddddd

The only information that the VE should be concerned with, as depicted above, is the bits where it is represented by the character ‘d’. The character ‘x’ represents zeros. These are not accounted for by the VE. The parser is now concerned with determining the correct addition of the information represented in several bytes. Utilizing bitwise operations in C++, it is possible to ensure the exclusion of any information by masking a byte and retrieving the pertinent information. In the case of the buttons and accelerometers this implies masking the first four bits of the byte and ensuring that they are zeros.

- *Input AND 0x0F* *(AND is equivalent to &)*

The previous statement is a hexadecimal representation of the masked bits as the ‘0’ after the x represents the zeroing of the left most bits while the ‘F’ represents all ones which leaves the information in the right most bits as is. This is a method of ensuring that only the pertinent data in each byte is accounted for. The next step in the parsing process revolves around the shift of information received from the first byte to combine it bitwise with the second byte received (for those components whose information spans more than one byte of information).

- *(Input & 0x0F)<<8*

This statement represents that shift, adding padding while the 8 bits from *input* are shifted. The final step is to apply the second byte that follows, combining the two bytes to represent one numerical value.

- $((Input_{first}\ \&\ 0x0F) << 8)\ |\ (Input_{second}\ \&\ 0xFF)$

This final representation shows the OR operator between the two bytes. This is similar to the below representation.

	00001111 00000000
OR	11111111
	00001111 11111111

This is a binary representation combining the two bytes of information into one value that can be cast as an integer in the VE. Simply adding the two bytes to form a single numeric representation would not give the correct answer, hence the use of the bitwise operators.

4.2.4 OpenGL

OpenGL is a set of graphical libraries often used for games as well as visual representation of data to a user on the screen. The libraries include functions to initialize views into a 3D space, initialize and change objects as well as assist in the definition of interaction. This is done through drawing the display to the screen upon a change in the variables in the environment or upon an explicit redraw. The usage of the OpenGL libraries assisted in the implementation in the following areas: navigation, object manipulation, and object selection.

4.2.4.1 Navigation

Navigation is provided through the interface as is familiar with many games in today's game market. When a person wishes to move, as the user interacts with the controller, the eye, or rather the camera slides through the graphical environment. This is visible by the addition of a floor to the environment. This allows for a permanent point of reference that the user can use to judge the accuracy of the movement based on the interaction with the controller. As proposed earlier in the paper, the mapping of the controller to the navigational portion of the software is something that will be reflected in the VE.

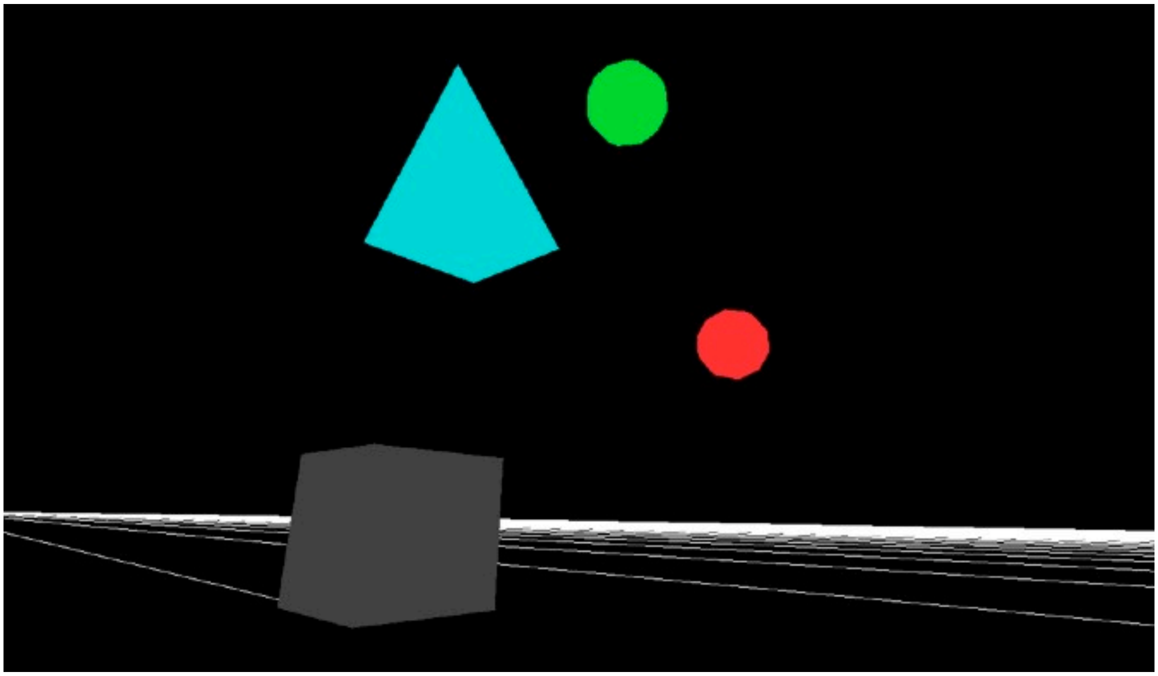


Figure 28: First Person View into the VE

Figure 28 depicts a sample view of the world the VE places the user into. The approach that is being utilized is centered around the use of the accelerometer. The accelerometer uses the three axes to provide forward/backward and left/right motion in a sliding motion in each of those directions. The third axis is used to travel up and down. While this is the current setup in the environment, the particular problem of functional mapping is something that can be addressed by inspection of the expected movement through the VE as well as the actual movement depicted on the screen. This can be adjusted for individual users or a standard can be set based upon future user studies.

4.2.4.2 Object Selection

Object selection is the process by which the user can, upon inspecting the VE, choose an object to manipulate. This reflects upon the ability of the controller to allow the user to distinguish between individual objects in the environment and the software's ability to single out an individual component. The act of selection revolves around the intersection between any given object and a ray. Currently this is not implemented in the VE, but the capabilities certainly exist. The mathematics that concerns the intersection of a ray and a sphere can be found in Appendix D. The ray is a simple line with a start and an end at the far viewpoint. This effectively stretches to "infinity" as the user is only concerned with selecting objects that are within the view space currently presented. The intended use of the ability would be to have a ray interact with the environment based upon camera movement (always in view, similar to a cursor) as well as user input. The user input would change the position of the cursor, moving through the space in front of the user. The final intention was to implement a selection function, mapped to a button, that would allow the user to directly manipulate an object as outlined in the next section.

4.2.4.3 Object Manipulation

The ability to manipulate an object revolves around the functions built around the transformation of the current OpenGL matrix. This allows for the transformation of objects, whether it is a scale, translation or rotation. This covers the basic manipulations one can perform on an object. This allows the device to explore the differences when incorporating various modes into a device.

- Translation:

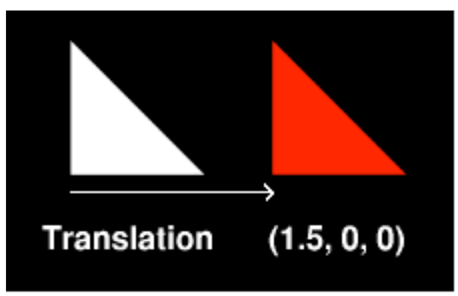


Figure 29: OpenGL Translate

http://www.limsi.fr/Individu/jacquemi/IG-TR-4-5-6/opengl-transf3.png

$$T(dx, dy, dz) = \begin{vmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

The above Matrix represents the calculations to translate (slide) an object in 3-D space (***Figure 29***). It is equivalent to moving an object along an axis. The numerical input to dx, dy, or dz represents the amount the object moves along the respective axes.

- Scaling:

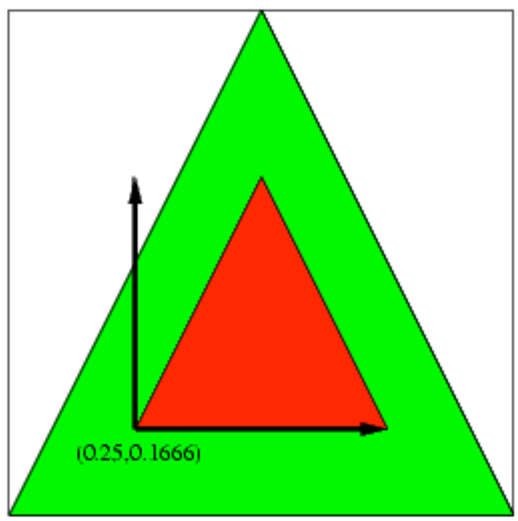


Figure 30: OpenGL Scale

http://www.comp.leeds.ac.uk/marcelo/opengl/transform2d-b.png

$$S(sx,sy,sz) = \begin{vmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

The above matrix represents the scale of a triangle utilizing the OpenGL libraries. The first (green) triangle in ***Figure 30*** is drawn with no change to the current matrix that is currently used as a reference. The function to scale is called and the matrix that represents the current numerical multiplication of the objects represented on the screen. When the scale is applied, the new triangle (red) is drawn over the old according to the new scale factor.

- Rotation:

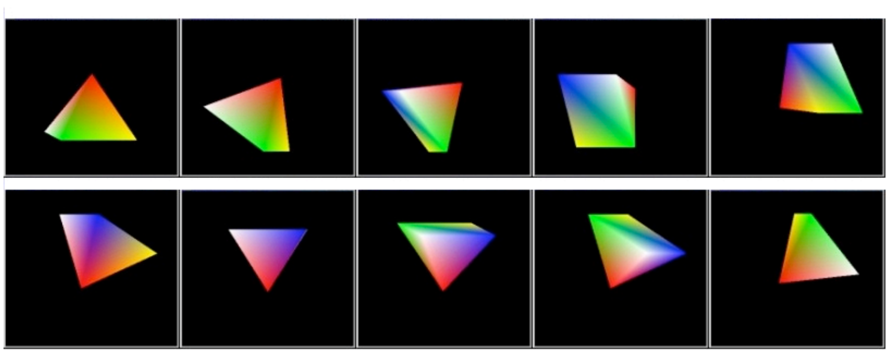


Figure 31: OpenGL Rotation

http://www.naturewizard.com/Tutorials/Tutorial01/images/image010.jpg

$$Rx(A) = \left| \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & \cos A & -\sin A & 0 \\ 0 & \sin A & \cos A & 0 \\ 0 & 0 & 0 & 1 \end{array} \right|$$

$$Ry(A) = \left| \begin{array}{cccc} \cos A & 0 & \sin A & 0 \\ 0 & 1 & 0 & 0 \\ -\sin A & 0 & \cos A & 0 \\ 0 & 0 & 0 & 1 \end{array} \right|$$

$$Rz(A) = \left| \begin{array}{cccc} \cos A & -\sin A & 0 & 0 \\ \sin A & \cos A & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right|$$

Rotation matrices are applied just as the previous two are and can change the rotation of an object in relation to any of the three axes. Figure 31 shows a sample application of the matrices to an object. The pyramid rotates along several axes at once, showing a practical application of these functions.

4.2.5 The Virtual Environment

The Virtual Environment is the compilation of the previous software related sections. It ties together each of the three classes into one cohesive unit. There exists one parser, one environment and one communication module. There exists within the environment any arbitrary number of objects. Currently this is set in the code but is open in the future for expansion. The purpose of the entire package is to simplify the components and define their interaction. Thus, a single executable can be run under Windows allowing the controller to connect over a COM port and update the display.

5. Results and Analysis

5.1 Hardware

After the completion of the system design and the acquisition of components began, each subsystem of the controller was individually tested to ensure proper operation.

5.1.1 Sensors

Testing the controller's sensors involves applying power to each device, moving it through its full range of motion, and comparing the measured output to the expected output at specific points. Once the analog voltage levels from each device are verified, they can be passed to the microprocessor with confidence.

The first device tested was the 3-axis accelerometer. In order to power up the device, the system's 3V is applied to the Vcc pin of the accelerometer, and the ground pin is grounded. The device has three individual analog outputs corresponding to the X, Y, and Z acceleration forces on the package. According to the datasheet for the ADXL330, with 0 gs applied to the device, the output should remain around half of the voltage supply, or 1.5V. With a sensitivity of ~0.3V/g, the outputs should range from 1.2V for -1g, and 1.8V for +1g. Subject to 0 gs of force, the equilibrium voltage of each axis is measured to be:

- X 0g --> 1.52V
- Y 0g --> 1.51V
- Z 0g --> 1.52V

This shows a small bias of about 0.02V given no input. Figure 32 shows the orientation of the package to obtain specific gravitational forces.

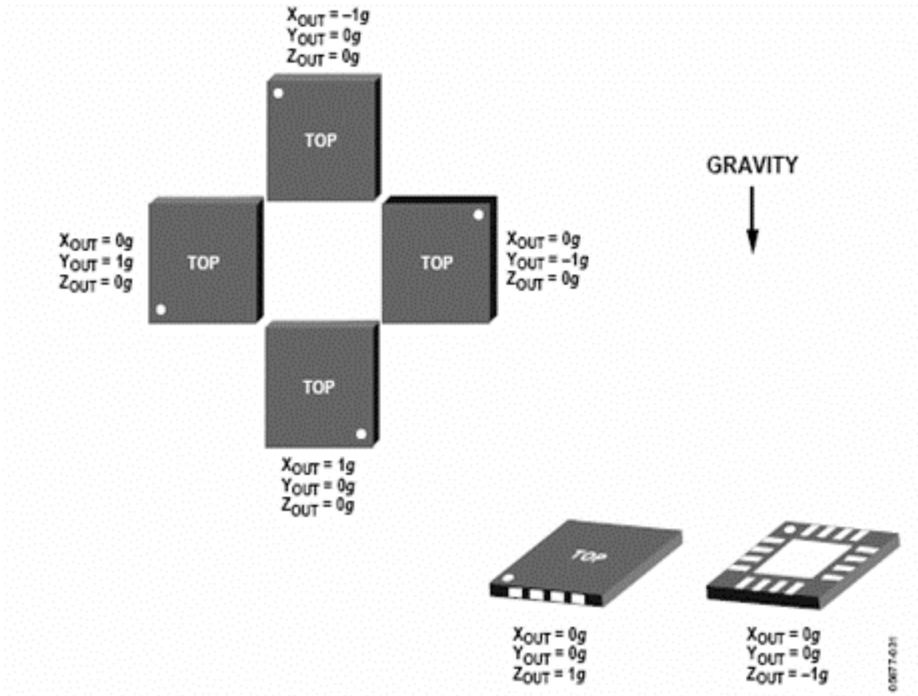


Figure 32: Accelerometer Orientation [1]

For each axis of the accelerometer, a test was performed at 0, 45, 90, 135, and 180 degrees. Using trigonometry to calculate the force of gravity at the various angles, as well as the stated sensitivity of the accelerometer, the anticipated output voltages were calculated.

$$\sin(\theta) = Vout/(V/g)$$

45 degrees X-axis

$$Vout = \sin(\theta) * V/g$$

$$Vout = \sin(45) * 0.3 = 0.212V$$

1.52V at 0 degrees + 0.212V = 1.73V

Actual measured voltage: **1.725V**

Tables 4-6 show the calculated and measured values for each axis of the accelerometer.

Accelerometer X-Axis				
Degrees	ΔV	Calculated V	Measured V	Digital Val
0	0V	1.520	1.520	2045
45	0.212V	1.732	1.725	2280
90	0.3V	1.820	1.812	2450
135	-0.212V	1.308	1.315	1821
180	-0.3V	1.220	1.230	1633

Table 4: Accelerometer X-Axis

Accelerometer Y-Axis				
Degrees	ΔV	Calculated V	Measured V	Digital Val
0	0V	1.510	1.510	2041
45	0.212V	1.722	1.720	2278
90	0.3V	1.810	1.810	2448
135	-0.212V	1.298	1.300	1822
180	-0.3V	1.210	1.215	1645

Table 5: Accelerometer Y-Axis

Accelerometer Z-Axis				
Degrees	ΔV	Calculated V	Measured V	Digital Val
0	0V	1.520	1.520	2050
45	0.212V	1.732	1.730	2280
90	0.3V	1.820	1.820	2445
135	-0.212V	1.308	1.310	1830
180	-0.3V	1.220	1.230	1650

Table 6: Accelerometer Z-Axis

It can be concluded from these results that the accelerometer performs within the specifications listed in its data sheet. Although the measured voltages are not exactly as expected (about 0.01V off in many cases), they are consistently offset, so a correlation between voltage and angle can still be determined. Although this accelerometer does not have very high precision, it is precise enough for the tilt sensing application of this controller. One issue that may have to be resolved in software is jitter. As the accelerometer sits in a static position, its output varies slightly. In order for this variation not to create movement in the virtual environment, an averaging function must be performed on the incoming data.

The next device under test was the scroll wheel. The functionality of this sensor is tested in two ways. First the voltage levels are tested, and second the offset of the two signals is verified. As the wheel is rotated, the two outputs of encoder should provide a digital 1 or 0 as each detent in the wheel is reached. The power and ground pins of the scroll wheel are wired, and the outputs are measured with a digital multi-meter. As expected, the outputs alternate between 0V and 3V as the wheel is rotated. To test the offset of the two signals, the outputs are monitored through a small LED circuit. As each output goes high, it lights its corresponding LED. The center pin is wired to LED #1 and the outer signal pin is wired to LED #2. By slowly rotating the wheel clockwise, it can be seen that LED #1 is enabled slightly before the other. Reversing the direction of rotation (counter clockwise) results in LED #2 lighting up slightly before LED #1. This will allow the microprocessor to determine which direction the wheel is rotating, and therefore whether to increment or decrement the appropriate counter.

The joystick was the next sensor to be tested. This device functions with the use of potentiometers, which vary their resistance as the joystick is moved throughout its range. By applying a constant voltage to the potentiometers, a proportional voltage will be output to the microprocessor. Below is a sample calculation of the voltage output from the potentiometers:

Total (power to ground) resistance: 5.9kΩ

Power to Output resistance: 2.95kΩ

Output voltage = (2.95/5.9) * 3V = **1.5V**

Tables 7 & 8 show the data obtained from each axis of the joystick.

X-Axis				
Degrees	Resistance(kΩ)	Calculated V	Measured V	Digital Val
0	2.95	1.5	1.5	2035
23L	2.25	1.86	1.85	3802
45L	1.56	2.16	2.15	4094
23R	3.65	1.45	1.44	1592
45R	4.34	0.79	0.8	2

Table 7: Joystick X-Axis

Y-Axis				
Degrees	Resistance(kΩ)	Calculated V	Measured V	Digital Val
0	2.85	1.5	1.49	2032
23L	2.15	1.88	1.87	3811
45L	1.51	2.21	2.2	4094
23R	3.58	1.13	1.15	1559
45R	4.19	0.79	0.78	1

Table 8: Joystick Y-Axis

The “R” or “L” in the degrees field indicates if the joystick was pushed right of left of its origin. It can be determined from this data set that the joystick outputs data as expected. One important note, however is that the voltage output is non-linear; it is slightly more sensitive near the center position than it is near the edges of its motion.

The last device to be tested is the trackball. For accuracy purposes, the pulses per revolution of the ball can be calculated given the number of “notches” in the wheel, and the dimension of the wheel and rollers.

30 openings = 60 positions/revolution (each axis)

$$\text{roller diameter} = 2.2\text{mm}$$
$$C = \pi * d = 3.14159*2.2\text{mm} = 6.911\text{mm}$$
$$\text{ball diameter} = 19\text{mm}$$
$$C = \pi * d = 3.14159*19\text{mm} = 59.690\text{mm}$$
$$59.690/6.911 = 8.636 \text{ rev/rev}$$

8.636*60 = **518 positions/revolution of ball**

This means that for one full revolution of the trackball, the microprocessor's counter will be incremented 518 times, assuming the ball remains in full contact with the rollers the entire time. Because the trackball operates on the basis of wheel encoders, its testing procedures follow that of the scroll wheels. Each sensor is provided 3V of power, and its ground terminal is grounded. As the slotted disc is rotated through the sensor, the outputs alternate between the 3V high and 0V low. The two outputs are next wired to LED #1 and #2 used in the previous test. As the wheel is slowly rotated, LED #1 lights up 1/4 of a pulse width before LED #2. When rotated the opposite direction, LED #2 lights up first.

5.1.2 Microprocessor

Now with the sensors providing consistent and documented data, the microprocessor's functions must be tested to ensure the incoming data is properly interpreted. The first test performed to the process is a basic functionality procedure. A simple program that uses a timer to continuously blink an attached LED is programmed onto the chip. When powered up, the program begins automatically, and the LED proceeds to blink. This verifies both the functionality of the MSP430 chip, and the debugging interface.

With the debugging interface correctly working, it was possible to begin testing the different peripherals of the MSP430. Because the wheel encoders of the trackball and scroll wheels will function by causing interrupts in the microprocessor's code, the digital I/O interrupts must be tested. A simple program is created that waits in a lower power mode until an input (high) is recognized on port 2.0. When this occurs, the LED will be enabled and remain lit. Using this program, the interrupt functionality of the MSP430 was successfully verified.

Next to be tested was the analog to digital converter. Using a sample program provided by Texas Instruments, a single channel analog to digital conversion is repeated, with the results stored in a global variable. By setting the reference voltage to Avcc, this test should also verify the minimum and maximum values that can be held in the ADC result buffers. Using the IAR Kickstart software and USB FET debugger, the register values of the microprocessor can be actively monitored as the program is run. Running the program provides the expected results. With an input of ground (0V), an output of 0000 is stored in the ADC results buffer. With the supply (3V) applied at the input, the maximum 4095 value is stored in the buffer. This corresponds to 2^{12} -1, as the device is a 12-bit converter.

5.1.3 Communication

In order for data to be sent from the microprocessor to the PC, the UART functionality of the MSP430, as well as the FT232 chip had to be tested simultaneously. Again, a simple TI-provided program was loaded onto the MSP430. This program simply takes a string “Hello World” and sends it character by character over UART, through the FT232 to the PC. On the PC a serial terminal (Terraterm) is opened, and set to monitor the correct serial port. When the program is initiated, the message “Hello World” is successfully sent every second to the PC.

The next test was to verify two-way communication. Again, a sample program was loaded. The function of this program is to take a keyboard character from the PC, increment its ASCII value, and send it back. Setting up a terminal window on the PC allows a connection to the microprocessor. As expected, typing a character in the terminal window results in a response of the next character in the ASCII code. With the PC to microprocessor connection verified, data can now be successfully sent to the PC with an increased degree of confidence.

5.1.4 Program Flow

With both the sensor output tested, and the proper functioning of the microprocessor and communication links, the overall program that takes data and sends it to the PC can be constructed and tested. This section will provide the primary functions of the microprocessor code, and explain in detail how they function. The full code is located in *Appendix C*.

5.1.5 System Testing

As the microprocessor's code develops, the overall function of the controller can be tested as well. With all, or some of the sensors connected (disconnected sensors result in readings of zero), and the program running, a terminal window can be opened on a PC, and connected to the controller via a virtual communications port driver. When the letter 'u' is typed into the console, the controller responds with all the current sensor information displayed in ASCII (or hexadecimal with a simple converter program). Through the use of the MSP430 debugging interface, the memory data on the processor can be monitored and compared to the data being displayed on the screen of the PC. Repeated polls to the controller show that the data is successfully transmitted to the computer. Data values can be verified in this step as well. For example, with 0 gs applied to an axis of the accelerometer, a reading of half supply or 1.52 V is output to the microprocessor. This in turn is converted to (1.52/3) * 4095 = 2075, which correctly matches the range of values being sent to the PC.

5.2 Software

The testing regarding the software revolves around specifically invoking each of the functions the VE is capable of handling. Individually this means opening an environment and testing navigation, object selection, manipulation as well as the parsing and communication. The parsing and communication require that the software be capable of outputting the information that is being received by the computer. This can be used to check the information coming in against what is expected. Graphically, the VE allows the user to determine if the information coming in is generally correct and whether it is producing the correct transformations to the display the user sees after interacting with the device. Through the early stages of testing, it was also deemed necessary to account for noise in the device.

5.2.1 Component Noise

The testing showed that the accelerometer and the joystick produce noise when sampled. This noise can interfere with the environment and how it portrays the scene to the user. When noise is present, the scene the screen portrays to the user jitters erratically. For the joystick, the cube which it rotates also jerks erratically. The noise is something that can be compensated for, allowing for smooth transitions.

It was important to define a controller class that held the information for a particular component. In this case, the accelerometer will be discussed. Each of the three axes had different information, such as the amount of jitter, the data being passed at its initial and resting state, as well as the maximum and minimum values. The latter was described earlier in the report. In order to account for the noise, the software allows each axis to have a threshold. This threshold accounts for a certain amount of change that must occur before the accelerometer (along any given axis) will produce a valid piece of information. The logic behind the threshold mathematics can be described in a few steps. One checks the current sample of the data from the controller, specifically that of the accelerometer. Once obtained, the difference between the sample and the last non-jitter piece of information received is measured, and compared to the threshold. Should the difference prove to be greater, the current sample is then used as the newest valid numeric data piece. This valid data piece is then used to create a percentage, describing where the current sample is in relation to the maximum and minimum values the accelerometer attains when in use. This was described in the hardware section earlier in the report. The percentage is the division of the current sample by the difference between the max and min values of the accelerometers axis. This percentage is used to adjust the maximum move rate the environment provides to the user. This is depicted on the screen as translation through the environment. An example is provided below.

Current Sample	Last Sample	Max Rate	Threshold	Max Value	Min Value
3020	2048	5.0 Units	50	3072	1024

Table 9: Sample Data from Accelerometer

The absolute value of the difference between the current sample and the last sample is 72. (Absolute value of 3020 – 2048). Seventy-two is compared to the threshold value of 50. Since the calculated value is greater than the threshold, the current data sample is considered valid. Next, the difference of the sample and the mean of the min and max values is calculated. This provides the distance from the resting state the sample is. Dividing the previous calculation by the same mean of the min and max values provides a positive or negative value between 0 and 1 that describes a percentage of movement the accelerometer is providing i.e., ((3020-2048)/2048 =). This provides the sample's relation to the spread of data points the accelerometer is capable of generating. This relation is described as a percentage, in this case 0.47. The max rate is then multiplied by this percentage, in this case (5.0 Units * 0.47) to provide the total value to apply to translate through the environment.

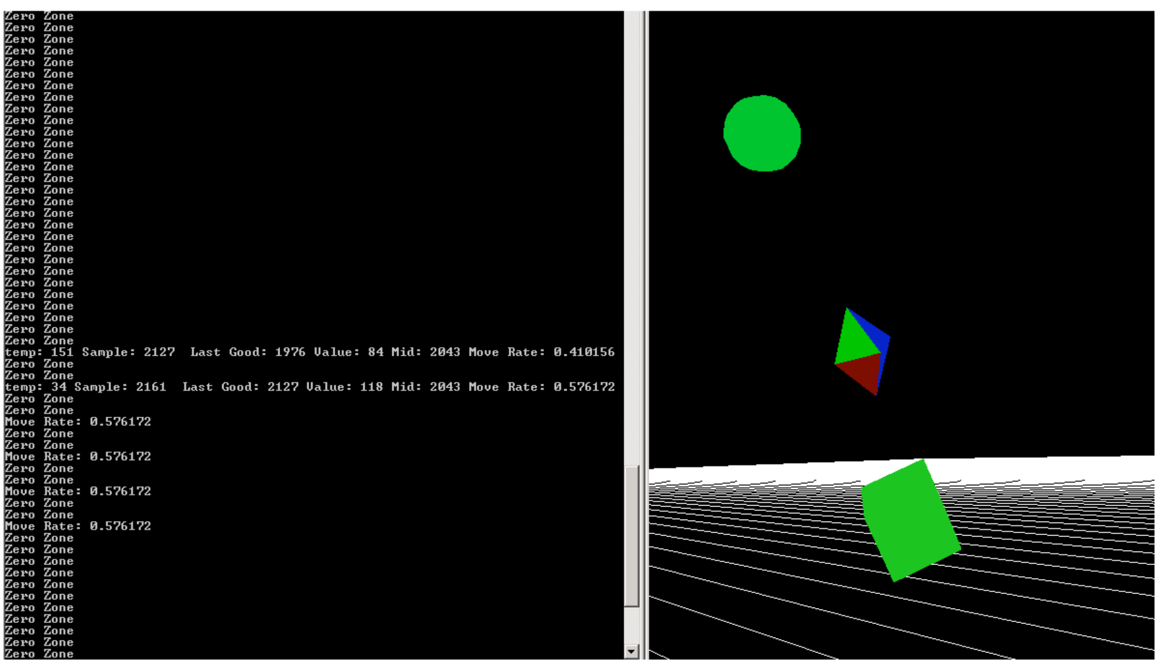


Figure 33: Sample next to Environment

As seen in Figure 33, the sample in the middle of the screen provides information based on the last move, the current sample, the mid point and the current move rate. This is all translated to the screen when a valid sample is sent from the controller to the PC. Surrounding the valid data is the text “Zero Zone” which depicts the controllers level state where it does not provide input to the environment.

6. Conclusions

The project started with an idea of a new virtual reality handheld controller. After researching the current controllers on the market and the past controllers that the gaming industry had seen over the years, we drew conclusions about the components that were most successful for interaction with a virtual environment. Having seen what was available, we turned to look at new possibilities focusing on the accelerometers that are present in the Nintendo Wii controllers. A list of functions was created to explore the requirements of the controller as it would interact with a virtual environment. Control mappings were explored as each component was assigned strengths and weaknesses and finally picked to perform an action. After we outlined our expectations we proceeded with a design and the project took form. We had to design the entire hardware system, drawing out circuits as well as implementing the software to unify the components. The software evolved around the central idea of interacting in a Virtual Environment as input from the hardware was made available. After our design was complete we began to implement and test the different components of our system. Our main conclusions revolve around the two sections of our project. The hardware system successfully transmits data over the USB to the PC. The software takes in that data and can transform the environment on the display of the PC.

If we had more time to pursue the advancement of this project, the two aspects of this project would continue as follows. The team would more thoroughly test and explore the interaction between the software and hardware as well as solidify the components into a hard case to unify them. Closely related, if we could start over again there would be some changes that would be made to ensure this project run more smoothly. One of the changes would relate to the relationship between two students in different majors, utilizing their skills to bring a project to fruition that combined both of their respective fields. This was one of the larger areas where problems occurred due to inexperienced communication between different majors. The setbacks that were experienced also contributed to this feeling of frustration as certain research proved to be misleading, as in the case of certain device driver implementation libraries, or when hardware did not respond as expected as according to the manufacturer's specifications. Although this project may have proved difficult, it has equipped us with the tools necessary to tackle a similar project in the respective fields with additional knowledge and expertise. Although this project did not completely reach its original goals, we were still able to create a functioning system that can be built upon in the future.

6.1 Future Work

As the project progressed and evolved, some aspects presented themselves in both software and hardware that could be expanded upon in the future beyond the scope of this project.

6.1.1 Software

The environment, while adequate for the time being in determining if the device is correctly gathering and outputting the input from the user, could be expanded upon to provide greater functionality as well as become a future testing ground for other devices. In having a standard testing suite to compare multiple products, one can begin to look at the effectiveness in separate controllers and the future marketability of any new device that is being developed. The environment's generic implementation of the basic modes of interaction between a virtual world provide for testing without discrimination between a specific implementation of device, game, or other world that could be used to compare devices.

The communications the environment uses are adequate for its current use, but the project could benefit greatly from the development of specific device drivers. Utilizing the USB capabilities of the device would allow the device to transmit data faster. While this would limit the implementation to the Operating System the USB driver was written for, it would allow the device to take full advantage of the speed that USB offers and allow the environment the option to no longer treat the device as a polling system.

6.1.2 Physical Design

Based upon the project specifications it is possible in the future to draft a physical design, implementing the selected hardware, circuitry, and sensors into a hand-held controller device

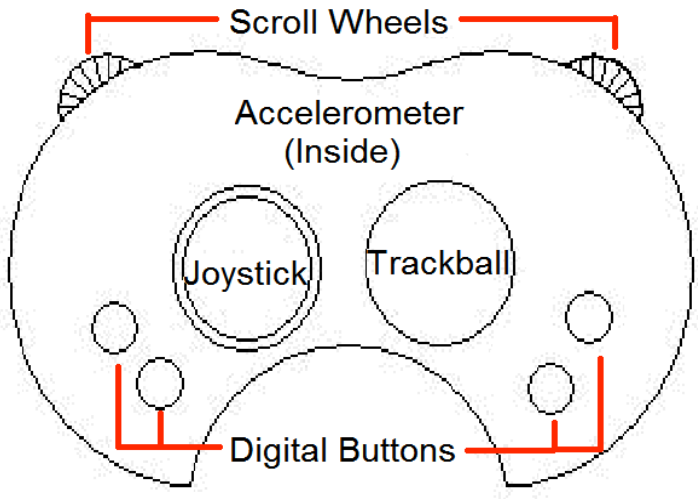


Figure 34: Sample Controller Design

The controller follows a simple ergonomic design that is commonplace in most controllers on the market today. Each shoulder has a scroll wheel. This provides two methods of input along an axis in a segmented manner, and grants the user the ability to move through a series of selections either in a graphical list, or within an environment and easily move between the options. The left half of the controller provides the user with an analog joystick and two buttons. This analog stick provides a joystick with a centering characteristic that can improve the way in which a user moves through an environment. The right side mirrors the left with the exception of the analog stick. It is replaced with a trackball. This configuration provides versatility and numerous ways to manipulate the environment. This is one of many possible configurations. Having implemented the major hardware required to fill this casing, it would be a huge benefit to see this implemented and made real some point in the future.

6.1.3 Continued Testing

It would behoove the project to have continued testing of the device and its development. The original purpose of this device was to improve upon the existing designs by adding some twists and utilizing components that you would not normally find on a hand held game controller device. The continued testing would allow for the evolution and improvement upon the configuration of the mapping of functionality.

References

1. Analog Devices Inc. ADXL330 3-Axis Accelerometer Datasheet. ADXL330.pdf. http://www.analog.com/en/prod/0%2C2877%2CADXL330%2C00.html
2. Bergman, Jason. "SpaceOrb 360 Game Controller." Blue's News. 25 Oct. 1997. <http://www.bluesnews.com/articles/spaceorb360-review.html>.
3. Davis, Tom and Neider, Jackie and Woo, Mason. “OpenGL Programming Guide” 1994,Silicon Graphics, INC.
4. Gyration. <http://www.gyration.com/en-US>.
5. Kopchak, Jeremy. "Nintendo® Wiimote: Technology Limitations." X-Arcade. <http://www.xgaming.com/newsletter/Wii%20Dupe.shtml>.
6. Maxon, Kenneth. “Have you seen my new soldering iron?” Encoder. http://www.seattlerobotics.org/encoder/200006/oven_art.htm
7. Jeff Molofee “NeHe Productions” OpenGL Game Development. http://nehe.gamedev.net/
8. Nintendo 64. CyberiaPC.com
9. "Nintendo Wii - Controllers." Nintendo. <http://wii.nintendo.com/controller.html>.
10. "Products." MINDFLUX. <http://www.mindflux.com.au/products/index.html>.
11. Rick. "Nintendo's Investment in Gyration." Gamecubicle. <http://www.gamecubicle.com/news-nintendo_gyration.htm>.
12. Wisniowski, Howard, ed. "Analog Devices and Nintendo Collaboration Drives Video Game Innovation with the IMEMS Motion Signal Processing Technology." Analog Devices. 9 May 2006. <http://www.analog.com/en/press/0,2890,3%255F%255F99573,00.html>.
13. Brad A. Myers. "A Brief History of Human Computer Interaction Technology." ACM interactions. Vol. 5, no. 2, March, 1998. pp. 44-54.
14. Koushik, Sud. "Evolution of Controllers." Advanced Media Network. 30 Jan. 2006. 20 Sept. 2006 <http://wii.advancedmn.com/article.php?artid=6355>.
15. Texas Instruments Inc. MSP430F169 Datasheet.

http://focus.ti.com/docs/prod/folders/print/msp430f169.html

16. Texas Instruments Inc. MSP430 USB Connectivity Using TUSB3410. slaa276a.pdf
17. Texas Instruments Inc. MSP430x1xx Family Users Guide. Slau049f.pdf
18. Texas Instruments Inc. TUSB3410 USB to Serial Port Controller Datasheet. Tusb3410.pdf.
19. Texas Instruments Inc. TUSB3410_UART Evaluation Board User's Guide. Sllu043.pdf
20. Taylor, Russel M., Thomas Hudson, Adam Seeger, and Hans Webber. VRPN: a Device-Independent, Network-Transparent. University of North Carolina at Chapel Hill. 2 Oct. 2006 <http://www.cs.unc.edu/Research/vrpn/VRST_2001_conference/vrst_vrpn_paper_reprint.pdf>.
21. Flock of Birds – Virtual Reality Motion Tracker. Vrealities.com

Appendix A: TUSB3410 Design

Through the design process of this virtual reality controller, an alternate USB controller was incorporated into the hardware system. The initial design contained a TUSB3410 USB controller from Texas Instruments. This device is very similar to the FT232 that was eventually included. Both devices are designed to interface an RS232 data stream with a USB data stream. The primary difference between the two devices is that the TUSB3410 was purchased as an individual chip, and the FT232 was purchased as part of an evaluation board. The reason for the switch from the TUSB3410 to the FT232 was an inability of the TI part to be properly configured. Although it is common for a USB controller to have an auxiliary EEPROM memory chip to hold configuration and identifier data, the TUSB3410 data sheet claimed that it could also be successfully be implemented without one. After a great deal of testing and troubleshooting time was spent, it has been concluded that this device cannot be used without an EEPROM, or if it can, it requires special configuration not detailed in the data sheet or application paper[8].

The testing of this device began with the construction of a custom printed circuit board (PCB) that would allow easy access to all the pins of the device for wiring a breadboard circuit. Due to the cost ($50) and turn around time (1 week) with creating a professionally made PCB, a custom, home-made board was constructed. It was produced using a transfer and etch method involving toner transfer and a ferric chloride etch. The supplies needed for the construction of this board included:

- Radio Shack PCB Design Kit
    - Copper Clad PCB
    - Bottle of ferric chloride
    - Chemical solvent (isopropyl alcohol)
    - Abrasive pad
    - Plastic tray
- Ink-Jet photo paper
- Laser printer

[8] Texas Instruments Inc. TUSB3410 USB to Serial Port Controller Datasheet. Tusb3410.pdf

- Masking tape
- Household Iron

The first step of this process is creating the layout for the board in any PCB layout software. For this board, the ExpressPCB software was used. The resulting board (previously mentioned in the report) is shown in figure A1.

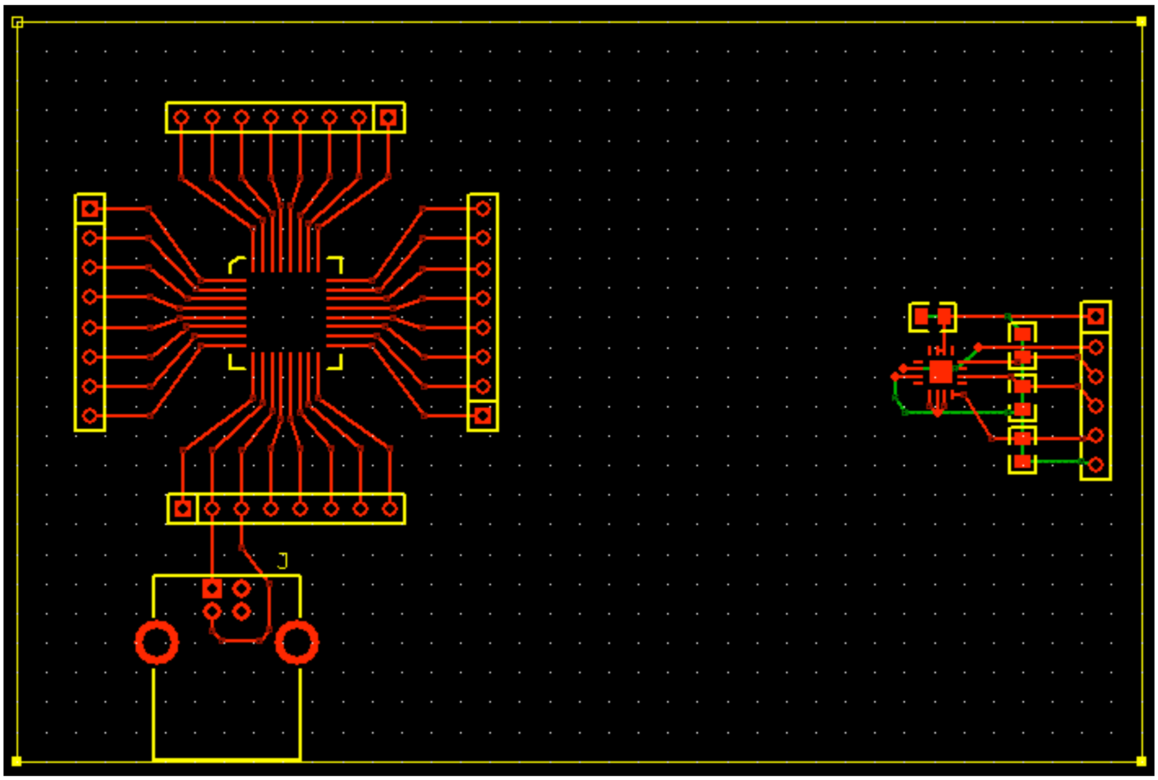


Figure A1: Board Layout

For this board, the accelerometer circuit is not used, and was deleted before printing the layout. It is important when creating the traces to lay them in a "mirrored" fashion. That is the entire circuit should be flipped over to create a mirror image. This is because the transfer will reverse will image when it is applied to the copper PCB. The next step is to print the layout onto a piece of photo paper with a laser printer. It essential that only the copper traces are printed, and not the silkscreen or any other layers as they will all be applied to the board.

The printed schematic should now be taped to the copper PCB, making sure it will not shift while heat is applied. A hot iron is not pressed on the paper for approximately five minutes, transferring the toner to the copper board. After the board is cooled down, it should be placed in a container of water for approximately twenty minutes to soak off the photo paper. After soaking, the paper can be removed by peeling and gentle scrubbing. The resulting board is shown in figure A2.

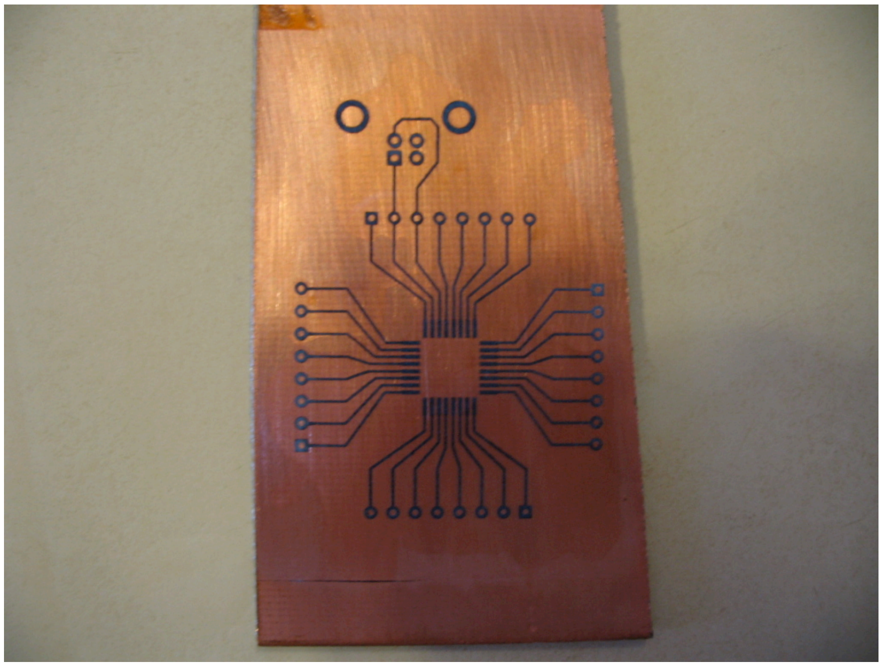


Figure A2: Pre-Etched Board

Here the toner traces can be seen applied to the copper board. The next step is to submerse the board in the ferric chloride until the exposed copper has been removed (approximately one hour). With the excess copper removed, the ink traces can be removed with the solvent and abrasive pad. The fully etched board is shown in figure A3.

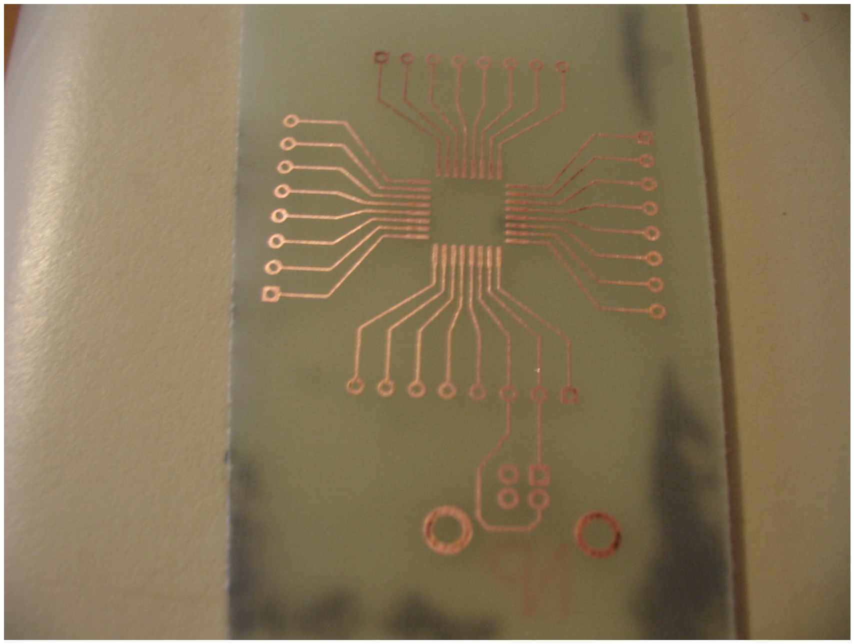


Figure A3: Post-Etched Board

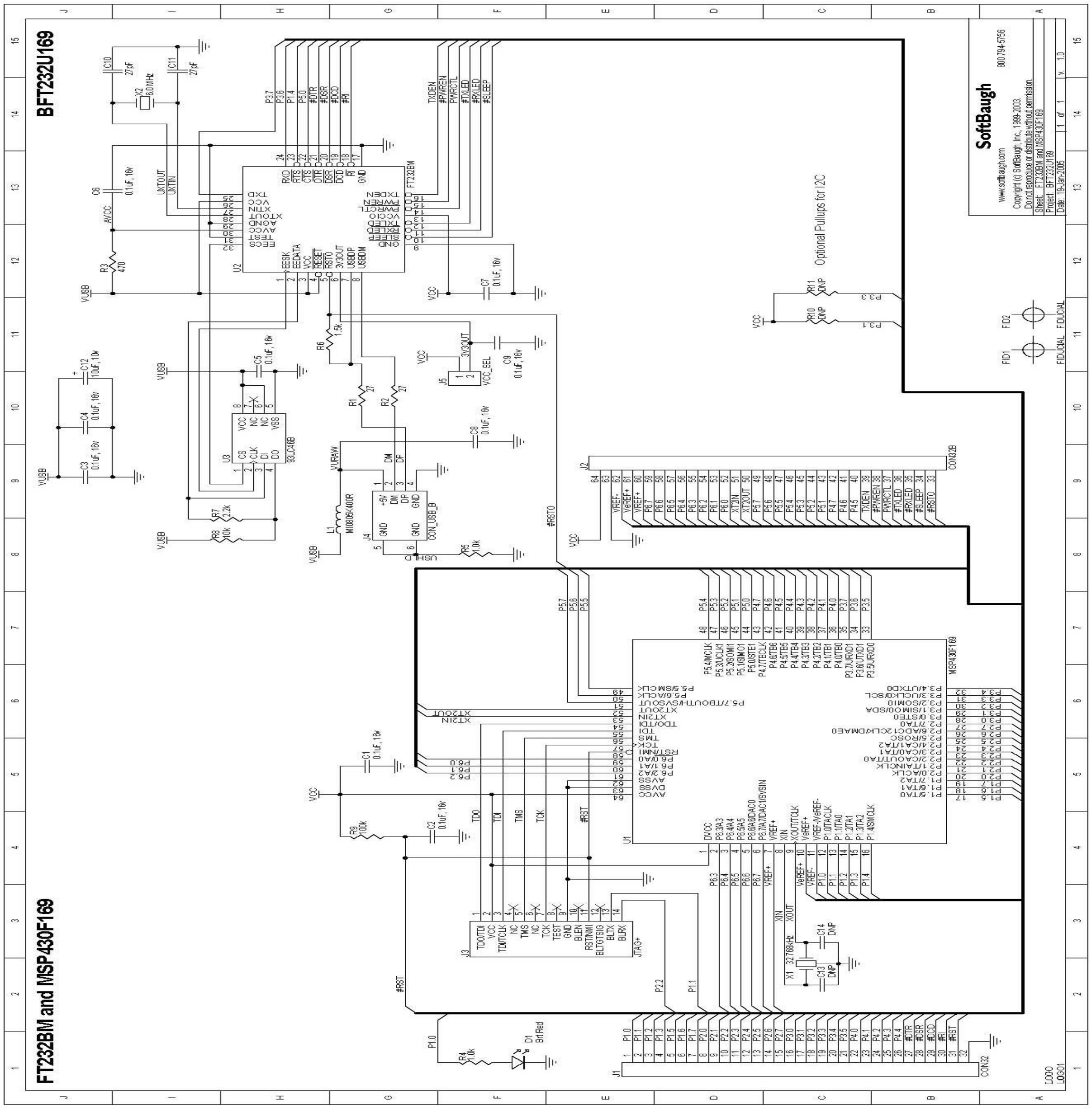
The final step is to use a small drill bit and drill press to create holes for soldering wires and other through-hole components.

After soldering the TUSB3410 chip to the custom PCB and wiring up all its supporting circuitry, detailed in the application note[9], connecting it to a PC's USB port failed to produce any actions. Many different configurations were tried, and the PC would not recognize the device under any circumstances. Due to the fact that time was a major consideration, and little to no progress was being made toward USB communication, the FT232 solution was adopted.

---

1. [9] Texas Instruments Inc. TUSB3410_UART Evaluation Board User's Guide. Sllu043.pdf

Appendix B: Softbaugh BFT232U169 Schematic



Appendix C: MSP430F169 Code

The first step in the program is to provide any include statements and initialize variables. The include statement ensures all the predefined keywords and functions designed for the MSP430x14x series will be available. The next six lines initialize the global variables for the program. Some variables to note are the array of 5 ADC results, and the 4 wheel counters. The wheel counters are all set to 32768, half-way between their minimum and maximum values to allow counting in either direction.

The main function of the code is used primarily to initialize the input and output functions of the processor. First the USART1 is configured to output on Port 3, and set up for 2400 baud using a 32kHz crystal. USART is then initialized, and the receive (RX) interrupt is enabled, which causes an interrupt in any incoming data. Next, Port 1 and Port 2 are configured to be inputs for the wheel encoders and digital buttons. Half of the inputs interrupts are enabled to allow the leading edge signal of the wheel encoders to cause in interrupt. Lastly, the analog to digital converter is initialized. It is setup to take samples from 5 channels, and store them in the appropriate registers. The reference voltages are all set to Avcc, to ensure the inputs will not hit the positive or negative limits. The ADC finish interrupt is enabled, and the conversion process is started. At this point the microprocessor is then put into lower power mode (LPM) while the conversions take place.

The next function is the interrupt vector for the analog to digital converter. This function is performed when the ADC finishes its conversion, and sets the corresponding interrupt flag. This function just takes the current results from the conversions and transfers them into global variables.

The Port 2 interrupt function is next in the program. When one of the configured inputs of Port 2 is set high, this routine is called. It goes through a series of IF statements, to determine which of the wheel encoders has tripped the interrupt. This is determined by comparing the interrupt bit all four possible inputs. Once the correct sensor is identified, the status of the 2nd signal is checked. If it is high, the wheel counter is incremented. If it is low, the counter is decremented. At the end of this routine, the interrupt flags for the wheel encoders is reset.

The last function in the program is designed to send all of the sensor data to the PC. It is called when the UART receive interrupt flag is set. If the input to the microprocessor is the ASCII code for the letter 'u', then the data is sent to the PC. This polling method is used to prevent excessive amounts of data being sent to the PC, and causing false device identification when the controller is first plugged in. When the poll character is verified, the function then checks to make sure the transmit buffer is ready to send. When it is, status of the buttons is loaded into the transmit buffer and sent. The next loop sends the contents of the wheel encoder counters. These counters are 16-bit numbers and must be split up into two separate 8-bit numbers to send over the 8-bit UART interface. This loop splits each counter into two pieces, and sends each half when the receive buffer is ready. The next loop performs the same operation of splitting and sending the data created by the analog to digital conversions. At the end of each of the interrupt functions, the microprocessor is sent back to the main loop and into lower power mode.

```
#include  <msp430x14x.h>

unsigned int i,j;
static unsigned int ADresults[5];  // These need to be global in
static unsigned int wheel_counter[4] = {32768, 32768, 32768, 32768};
static unsigned char buttons;
static unsigned char UB0;
static unsigned char LB0;

void main(void)
{
  WDTCTL = WDTPW+WDTHOLD;                      // Stop watchdog timer
  // USART Config
  P3SEL |= 0xC0;                       // P3.6,7 = USART1 option select
  ME2 |= UTXE1 + URXE1;                        // Enable USART1 TXD/RXD
  UCTL1 |= CHAR;                               // 8-bit character
  UTCTL1 |= SSEL0;                             // UCLK = ACLK
  UBR01 = 0x0D;                                // 32k/2400 - 13.65
  UBR11 = 0x00;
  UMCTL1 = 0x6B;                               // Modulation
  UCTL1 &= ~SWRST;                     // Initialize USART state machine
```

```
  IE2 |= URXIE1;                     // Enable USART1 RX interrupt
  // Digital IO
  P1SEL = 0x00;                             // All set to I/O
  P1DIR = 0xF0;                      // P1.0 - 1.3 input, rest output
  P2SEL = 0x00;                             // All set to I/O
  P2DIR = 0x00;                             // All set for input
  P2IES = 0xFF;
  P2IFG = 0x00;
  P2IE = 0x55;                       // half of signals set interrupts

  // ADC Config
  P6SEL = 0x1F;                         // Enable A/D channel inputs
  ADC12CTL0 = ADC12ON+MSC+SHT0_8;// Turn on ADC12, extend sampling time
                                      // to avoid overflow of results
  ADC12CTL1 = SHP+CONSEQ_3;    // Use sampling timer, repeated sequence
  ADC12MCTL0 = INCH_0;                      // ref+=AVcc, channel = A0
  ADC12MCTL1 = INCH_1;                      // ref+=AVcc, channel = A1
  ADC12MCTL2 = INCH_2;                      // ref+=AVcc, channel = A2
  ADC12MCTL3 = INCH_3;
  ADC12MCTL4 = INCH_4+EOS;       // ref+=AVcc, channel = A3, end seq.
  ADC12IE = 0x10;                           // Enable ADC12IFG.3
  ADC12CTL0 |= ENC;                         // Enable conversions
  ADC12CTL0 |= ADC12SC;                     // Start conversion
  _BIS_SR(LPM0_bits + GIE);          // Enter LPM0, Enable interrupts
}

#pragma vector=ADC_VECTOR
__interrupt void ADC12ISR (void)
{



  ADresults[0] = ADC12MEM0;         // Move A0 results, IFG is cleared
  ADresults[1] = ADC12MEM1;         // Move A1 results, IFG is cleared
  ADresults[2] = ADC12MEM2;         // Move A2 results, IFG is cleared
  ADresults[3] = ADC12MEM3;         // Move A3 results, IFG is cleared
  ADresults[4] = ADC12MEM4;


}
```

```
#pragma vector=PORT2_VECTOR
__interrupt void PORT2_RX (void)
{
  // wheel 1 scroll 1
  if ((P2IFG & BIT0) == BIT0) // P2.0
  {
    if ((P2IN & BIT1) == BIT1)
      wheel_counter[0]++; //CW
    else
      wheel_counter[0]--; //CCW
  }
  // wheel 2 scroll 2
  if ((P2IFG & BIT2) == BIT2) // P2.2
  {
    if ((P2IN & BIT3) == BIT3)
      wheel_counter[1]++; //CW
    else
      wheel_counter[1]--; //CCW
  }
  // wheel 3 track x
  if ((P2IFG & BIT4) == BIT4) // P2.4
  {
    if ((P2IN & BIT5) == BIT5)
      wheel_counter[2]++; //CW
    else
      wheel_counter[2]--; //CCW
  }
  // wheel 4 track y
  if ((P2IFG & BIT6) == BIT6) // P2.6
  {
    if ((P2IN & BIT7) == BIT7)
      wheel_counter[3]++; //CW
    else
      wheel_counter[3]--; //CCW
  }
  P2IFG = 0x00;   //reset interrupt

}
```

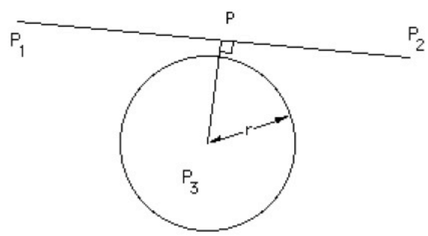
```
// UART0 RX ISR
#pragma vector=UART1RX_VECTOR
__interrupt void usart1_rx (void)
{
  if (RXBUF1 == 'u')                          // 'u' received?
  {
    buttons = P1IN;                           // get status of buttons
    while (!(IFG2 & UTXIFG1));
    TXBUF1 = buttons;
    for(j=0; j<4; j++)
    {
      LB0 = wheel_counter[j];
      UB0 = wheel_counter[j] >> 8;
      while (!(IFG2 & UTXIFG1));
      TXBUF1 = UB0;
      while (!(IFG2 & UTXIFG1));
      TXBUF1 = LB0;
    }
    for(i=0; i<5; i++)
    {
      LB0 = ADresults[i];
      UB0 = ADresults[i] >> 8;
      while (!(IFG2 & UTXIFG1));
      TXBUF1 = UB0;
      while (!(IFG2 & UTXIFG1));
      TXBUF1 = LB0;
    }
 }
```

Appendix D: Intersection Math & Code

Intersection Of A Line And A Sphere (or Circle)

Written by Paul Bourke
November 1992

C code example by author
Source code example by: Iebele Abel.
Sphere/ellipse and line intersection code for Visual Basic by Adrian DeAngelis.
LISP version for AutoCAD (and Intellicad) by Andrew Bennett intC2.lsp and intC2_app.lsp.



Points **P** (x,y) on a line defined by two points $\mathbf{P}_1$ (x_1,y_1,z_1) and $\mathbf{P}_2$ (x_2,y_2,z_2) is described by

$$\mathbf{P} = \mathbf{P}_1 + u\,(\mathbf{P}_2 - \mathbf{P}_1)$$

or in each coordinate

$$x = x_1 + u\,(x_2 - x_1)$$
$$y = y_1 + u\,(y_2 - y_1)$$
$$z = z_1 + u\,(z_2 - z_1)$$

A sphere centered at $\mathbf{P}_3$ (x_3,y_3,z_3) with radius r is described by

$$(x - x_3)^2 + (y - y_3)^2 + (z - z_3)^2 = r^2$$

Substituting the equation of the line into the sphere gives a quadratic equation of the form

$$a\,u^2 + b\,u + c = 0$$

where:

$$a = (x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2$$

$$b = 2[(x_2 - x_1) (x_1 - x_3) + (y_2 - y_1) (y_1 - y_3) + (z_2 - z_1) (z_1 - z_3)]$$

$$c = x_3^2 + y_3^2 + z_3^2 + x_1^2 + y_1^2 + z_1^2 - 2[x_3 x_1 + y_3 y_1 + z_3 z_1] - r^2$$

The solutions to this quadratic are described by

$$\frac{-b \text{ +- } \mathrm{sqrt}(b^2 - 4 a c)}{2a}$$

The exact behaviour is determined by the expression within the square root

$$b * b - 4 * a * c$$

- If this is less than 0 then the line does not intersect the sphere.
- If it equals 0 then the line is a tangent to the sphere intersecting it at one point, namely at u = -b/2a.
- If it is greater then 0 the line intersects the sphere at two points.

To apply this to two dimensions, that is, the intersection of a line and a circle simply remove the z component from the above mathematics.

Line Segment

When dealing with a line segment it may be more efficient to first determine whether the line actually intersects the sphere or circle. This is achieved by noting that the closest point on the line through $\mathbf{P}_1\mathbf{P}_2$ to the point $\mathbf{P}_3$ is along a perpendicular from $\mathbf{P}_3$ to the line. In other words if **P** is the closest point on the line then

$$(\mathbf{P}_3 - \mathbf{P}) \text{ dot } (\mathbf{P}_2 - \mathbf{P}_1) = 0$$

Substituting the equation of the line into this

$$[\mathbf{P}_3 - \mathbf{P}_1 - u(\mathbf{P}_2 - \mathbf{P}_1)] \text{ dot } (\mathbf{P}_2 - \mathbf{P}_1) = 0$$

Solving the above for u =

$$\frac{(x_3 - x_1)(x_2 - x_1) + (y_3 - y_1)(y_2 - y_1) + (z_3 - z_1)(z_2 - z_1)}{(x_2 - x_1)(x_2 - x_1) + (y_2 - y_1)(y_2 - y_1) + (z_2 - z_1)(z_2 - z_1)}$$

If u is not between 0 and 1 then the closest point is not between $\mathbf{P}_1$ and $\mathbf{P}_2$

Given u, the intersection point can be found, it must also be less than the radius r. If these two tests succeed then the earlier calculation of the actual intersection point can be applied.

```
/*
   Calculate the intersection of a ray and a sphere
   The line segment is defined from p1 to p2
   The sphere is of radius r and centered at sc
   There are potentially two points of intersection given by
   p = p1 + mu1 (p2 - p1)
   p = p1 + mu2 (p2 - p1)
   Return FALSE if the ray doesn't intersect the sphere.
*/
int RaySphere(XYZ p1,XYZ p2,XYZ sc,double r,double *mu1,double *mu2)
{
   double a,b,c;
   double bb4ac;
   XYZ dp;

   dp.x = p2.x - p1.x;
   dp.y = p2.y - p1.y;
   dp.z = p2.z - p1.z;
   a = dp.x * dp.x + dp.y * dp.y + dp.z * dp.z;
   b = 2 * (dp.x * (p1.x - sc.x) + dp.y * (p1.y - sc.y) + dp.z *
                                         (p1.z - sc.z));
   c = sc.x * sc.x + sc.y * sc.y + sc.z * sc.z;
   c += p1.x * p1.x + p1.y * p1.y + p1.z * p1.z;
   c -= 2 * (sc.x * p1.x + sc.y * p1.y + sc.z * p1.z);
   c -= r * r;
   bb4ac = b * b - 4 * a * c;
   if (ABS(a) < EPS || bb4ac < 0) {
      *mu1 = 0;
      *mu2 = 0;
      return(FALSE);
   }
```

```
   *mu1 = (-b + sqrt(bb4ac)) / (2 * a);
   *mu2 = (-b - sqrt(bb4ac)) / (2 * a);

   return(TRUE);
}
```

Appendix E: Serial Communication Code

SERIAL.H

```
// Includes
#include <windows.h>


//DEFINE for serial port settings


/////////////////Serial Functions////////////////
/*********Open******************
*return int  1 (true) 0 (false)
*This should open the serial port and
*set the settings to the constants defined above
*/
/*********Open******************************
*Function:  serialOpen
*Params:    Port Number, File Handler
*Purpose:   To open Port Number (portNum), using
*a predefined handle. It also sets the
*basic properties of the port, such as
*a 9600 baud rate, one stop bit, no parity
*8 bit Byte size etc.
*
*/
HANDLE serialOpen(int portNum, HANDLE comPrt);
/*********"READ"***********************
*Function:  serialRead
*Params:    File Handler,pointer to Byte Buffer,
*pointer to struct CONTROLDAT
*Purpose:   Reads in a set amount of data. This
*data will fall under the following
*format, with what the data represents coming in
*from the controller.
      Buttons (4): xxxxdddd
      Trackball X: dddddddd dddddddd
      Trackball Y: dddddddd dddddddd
      Joystick X: xxxxdddd dddddddd
      Joystick Y: xxxxdddd dddddddd
      Accel X: xxxxdddd dddddddd
      Accel Y: xxxxdddd dddddddd
      Accel Z: xxxxdddd dddddddd
      Scroll 1: dddddddd dddddddd
      Scroll 2: dddddddd dddddddd
Where every 'd' represents a bit that we are
interested in keeping track of. Every
'x' is data that we don't care about. It will most
likely be set to 0, but make sure
```

```
to skip over that data anyways. The data should be
recieved in 8 bit chunks.

*Output:The output should be a data structure to be
passed to the graphics program that can
then extract the data for processing. The data
structure is better explained in the serialDAT.h.
*/
int serialOut(HANDLE comPrt);
int serialRead(HANDLE File, char * buffer, int
len);
/**********Write*****************************
*Function:  serialWrite
*Params:    File Handle, pointer to struct
CONTROLDAT
*Purpose:   To write to a serial port. This is
*mostly used for testing purposes and doesn't
*really need to be output to a serial port. It
could be a
*log file.
*******WARNING******** Implementation is going to
change
*/
int serialWrite();
int serialClose(HANDLE comPrt);
```

SERIAL.CPP

```
#include <windows.h>
#include <stdlib.h>
#include <strsafe.h>

//User defined includes for serial commmunication
#include "serial.h"


HANDLE serialOpen(int portNum, HANDLE comPrt)
{
      //declare objects for class
      DCB dcb;
      COMMTIMEOUTS timeouts;
      TCHAR com[5];
      wsprintf(com,TEXT("COM%d"),portNum);

      comPrt = CreateFile(
                  com,
                  GENERIC_READ | GENERIC_WRITE,
                  0,
```

```
            NULL,
            OPEN_EXISTING,
            0,
            NULL
            );
    if (comPrt == INVALID_HANDLE_VALUE)
    {
        printf("invalid Handle value\n");
        return NULL;
    }
    else
    {
        printf("Port is now open\n");
    }
    // default: 9600,8,n,1 no flow control
    ZeroMemory(&dcb, sizeof(dcb));
    dcb.DCBlength = sizeof(dcb);
    dcb.BaudRate = CBR_2400;
    dcb.ByteSize = 8;
    dcb.Parity = NOPARITY;
    dcb.StopBits = ONESTOPBIT;

    // disable read timeouts (asynchronous mode)
    timeouts.ReadIntervalTimeout = MAXDWORD;
    timeouts.ReadTotalTimeoutMultiplier = 0;
        timeouts.ReadTotalTimeoutConstant = 0;

        //Disable write timeouts
        timeouts.WriteTotalTimeoutMultiplier = 0;
        timeouts.WriteTotalTimeoutConstant =
0;//MAXDWORD;

        // set new comm state
        SetCommState(comPrt, &dcb);
        SetCommTimeouts(comPrt, &timeouts);
        SetCommMask(comPrt, EV_TXEMPTY);


        return comPrt;
}
int serialOut(HANDLE comPrt)
{
        char * buff = "u";
        int success = 0;
        DWORD dwBytesRead, dwBytesWritten;
        dwBytesRead = 1;
        if(comPrt != INVALID_HANDLE_VALUE)
        {
              if(WriteFile(comPrt, buff, dwBytesRead,
&dwBytesWritten, NULL))
```

```
            {
                  success =1;
            }
            return success;
      }
      else
            return success;
}
int serialRead(HANDLE comPrt, char * buffer, int
len)
{
      int success;

      //check to see if the file is open
      if (comPrt != INVALID_HANDLE_VALUE)
      {
            //nread is used to keep track of the
number of chars read
            DWORD nread;
            success = 0;
            if(ReadFile(comPrt, buffer, 19, &nread,
NULL))
            {
                  //printf("Read Success\n");
                  success = 1;
            }
            else
            {
                  printf("Failed to read\n");
            }
      }
      return success;
}
int serialClose(HANDLE comPrt)
{
      int result=-1;
// close serial port
      if (comPrt != INVALID_HANDLE_VALUE)
      {
            PurgeComm(comPrt, PURGE_TXCLEAR |
PURGE_RXCLEAR);

            CloseHandle(comPrt);
            comPrt = INVALID_HANDLE_VALUE;
            result = 0;
      }

      return result;
}
```

Appendix F: Parser

PARSER.H

```
#include <windows.h>
#include "serial.h"

/***********Purpose of Parser*****
*Utilizing the "serial" class, this class parses
*the information that is on the Serial line (COM
*ports). Functions are available to grab the
*characters off the line and check for numerous
*things.
*/

class parse
{
public:
	/***********Variables**************
	*char buffer[19] -	private
	*A char array of size 13. This is the size
	*of the input from the controller. It will be
	utilized
	*	to check for the initial character of
the structure
	*	(The '$' sign). It will then store the
following 13 characters
	*
	*int portNum	-	private
	*	An integer representing the port number
that should
	*	be opened during the init function
	*/

	typedef struct CALIBRATE
	{
		int calx, caly, calz;
		int tnum,avgx,avgy, avgz;
		int xmin,ymin,zmin,xmax,ymax,zmax;
	};

	struct CONTROLDAT
	{
		char buttons;
		int track[2],joy[2],accel[3],scroll[2];
	}input;

	CALIBRATE calAccel;
```

```
/***********Functions**************
*init - Initializes the parse object. This
*opens a serial port with tbe number
*indicated by the private portNum integer.
*getCInfo - check for the first character in
*the input from the serial port. If it is a
*'$' then that indicates the beginning of the
*information coming from the controller.
*readInfo - Reads in 19 characters that are
*the information sent by the controller.
*/
parse();
int init();
int getCInfo();
int readInfo();
int closePort();
int parseBuff();

/********Setters/Getters***********/

int setHandle(HANDLE Port);
int getPort(){return portNum;};
void setPort(int x){portNum = x;};
int getPOpen(){return portOpen;};
      int poll();
      int calibrateSensors();
      void zero(CALIBRATE & x);
private:
      int portOpen,calAx,calAy,calAz;
      int portNum;
      HANDLE comPrt;
      char buffer[19];
};
```

PARSER.CPP

```
#include <windows.h>
#include "parse.h"
#include <strsafe.h>

parse::parse()
{
      printf("Parser listening at: COM2");
      portNum = 2;
      portOpen = 0;
}

int parse::setHandle(HANDLE Port)
```

```
{
      if(Port != INVALID_HANDLE_VALUE)
      {
            comPrt = Port;
            printf("COM Port handler set\n");
            return 1;
      }
      else
      {
            printf("Invalid Handle Value: COM Port
handler not set\n");
            return 0;
      }
}
int parse::init()
{
      zero(calAccel);
      if(comPrt != INVALID_HANDLE_VALUE)
      {
            if(comPrt = serialOpen(portNum,comPrt))
            {
                  printf("Serial Port %d
opened\n",portNum);
                  portOpen = 1;
                  //calibrate Accelerometer
                  printf("Calibrating
Accelerometer.");
                  for(int i = 0; i<100;i++)
                  {
                        calibrateSensors();
                        if((i%10) == 0)printf(".");
                  }
                  printf("\nFinished
Calibration\n");
                  printf("Calibration Information
for Accelerometer\n");
                  printf("-------------------------
----------------\n");
                  printf("Avg X:    %d    Min X:
      %d    Max X:      %d
\n",calAccel.avgx,calAccel.xmin,calAccel.xmax);
                  printf("Avg Y:    %d    Min Y:
      %d    Max Y:      %d
\n",calAccel.avgy,calAccel.ymin,calAccel.ymax);
                  printf("Avg Z:    %d    Min Z:
      %d    Max Z:      %d
\n",calAccel.avgz,calAccel.zmin,calAccel.zmax);
                  return 1;
            }
            else
```

```
		{
			printf("Serial Port failed to
open\n");
			return 0;
		}
	}
	else
	{
		printf("Invalid COM value\n");
		return 0;
	}
}

int parse::getCInfo()
{
	//read in characters 1 at a time
	if(serialRead(comPrt, buffer, 19))
	{
		//printf("Success reading in Controller
Information\n");
		//success in reading controller info -
return 1
		return 1;
	}
	else
	{
		printf("Error reading in Controller
Information\n");
		//fail to read controller information -
return -1
		return -1;
	}
	return 0;
}
int parse::readInfo()
{

	return 0;
}
int parse::closePort()
{
	if(serialClose(comPrt) == -1)
	{
		printf("Invalid Handler Value was
passed: Your port may not be set and/or open\n");
		return 1;
	}
	else
	{
		portOpen = 0;
```

```
			printf("Serial Port closed
successfully\n");
		}
		return 1;
}


int parse::parseBuff()
{
		if(getPOpen() == 0)
		{
			printf("Port is not open to read
from\n");
			return 0;
		}
		else if(getPOpen() == 1)
		{
			int checkValue = 0;
			//Read the information from the parser
			//being sent by the controller and
assess
			//the change to the local variables
			checkValue = getCInfo();
			//After read, check for integer
information
			//in checkValue. If 1, success and
change information
			//based on the input from the
controller
			//if -1, then dislpay error message and
break out of
			//reading and close Serial Connection.
			if(checkValue == 1)
			{
				//parse buffer into apporopriate
local variables
				input.buttons = buffer[0] & 0x0F;
				// Get Trackball info.

		input.track[0]=((((int)(buffer[1]&0xFF))<<8)|
((int)(buffer[2]&0xFF)));

		input.track[1]=((((int)(buffer[3]&0xFF))<<8)|
((int)(buffer[4]&0xFF)));
				//scroll wheel

		input.scroll[0]=((((int)(buffer[5]&0xFF))<<8)
|((int)(buffer[6]&0xFF)));
```

```
input.scroll[1]=((((int)(buffer[7]&0xFF))<<8)
|((int)(buffer[8]&0xFF)));
        // Get Accelerometer info.
            //printf("Accel %d Raw: %d
%d\n",0,(int)(buffer[9]&0xFF),((int)(buffer[1
0]&0xFF)));

input.accel[0]=((((int)(buffer[9]&0x0F))<<8)|
((int)(buffer[10]&0xF0)));

input.accel[1]=((((int)(buffer[11]&0x0F))<<8)
|((int)(buffer[12]&0xF0)));

input.accel[2]=((((int)(buffer[13]&0x0F))<<8)
|((int)(buffer[14]&0xF0)));
        // Get Joystick info.

input.joy[0]=((((int)(buffer[15]&0x0F))<<8)|(
(int)(buffer[16]&0xFF)));

input.joy[1]=((((int)(buffer[17]&0x0F))<<8)|(
(int)(buffer[18]&0xFF)));
        //update the local variables -
return 1 for success
                    printf("Numerical Input:%d
%d %d %d %d %d %d %d %d %c\n",

    input.track[0],input.track[1],

    input.joy[0],input.joy[1],

    input.accel[0],input.accel[1],input.accel[2],

    input.scroll[0],input.scroll[1],
                        input.buttons);
            return 1;

        }
        else if(checkValue == -1)
        {
            printf("Error reading from Serial
Port.\n Action(s) being taken: ");
            printf("Closing Serial Port:
%d\n",portNum);
            //close the port associated with
this read
            closePort();
            return 0;
        }
```

```
			//printf("No conditions were met for
Parsing information on Serial Port\n");
			return 0;
		}
		return 0;
}
int parse::poll()
{
		int success = 0;
		if(serialOut(comPrt))
		{
			//printf("Polling the device\n");
			success =1;
			return success;
		}
		else
		{
			printf("Failed to poll device\n");
			return success;
		}
}
int parse::calibrateSensors()
{
		int x,y,z;
		//init calibration variables
```

```
		poll();
		Sleep(100);
		getCInfo();
		x =
((((int)(buffer[9]&0x0F))<<8)|((int)(buffer[10]&0xF
0)));
		y =
((((int)(buffer[11]&0x0F))<<8)|((int)(buffer[12]&0x
F0)));
		z =
((((int)(buffer[13]&0x0F))<<8)|((int)(buffer[14]&0x
F0)));

		//printf("Accel Raw Data: %d  %d
		%d\n",x,y,z);

		calAccel.calx +=x;
		calAccel.caly +=y;
		calAccel.calz +=z;

		if(calAccel.tnum == 0)
		{
			calAccel.xmax = x;
			calAccel.xmin = x;
			calAccel.ymax = y;
```

```
            calAccel.ymin = y;
            calAccel.zmax = z;
            calAccel.zmin = z;
      }
      else
      {
            calAccel.avgx =
calAccel.calx/calAccel.tnum;
            calAccel.avgy =
calAccel.caly/calAccel.tnum;
            calAccel.avgz =
calAccel.calz/calAccel.tnum;
            if(calAccel.xmin > x)
            {
                  calAccel.xmin = x;
            }
            else if(calAccel.xmax < x)
            {
                  calAccel.xmax = x;
            }
            if(calAccel.ymin > y)
            {
                  calAccel.ymin = y;
            }
            else if(calAccel.ymax < y)
            {
                  calAccel.ymax = y;
            }
            if(calAccel.zmin > z)
            {
                  calAccel.zmin = z;
            }
            else if(calAccel.zmax < z)
            {
                  calAccel.zmax = z;
            }
      }
      calAccel.tnum += 1;

      return calAccel.tnum;
}
void parse::zero(CALIBRATE & x)
{
      x.avgx = x.avgy = x.avgz = x.calx = x.caly =
x.calz = x.tnum = x.xmax =0;
      x.xmin = x.ymax = x.ymin = x.zmax = x.zmin =
0;
}
```

Appendix G: Virtual Environment Code

Controller.h

```
#ifndef controller
#define controller
#include <windows.h>
class control
{
public:
	int lastgood,mid,thresh,minval,maxval;
	float lastMoveRate;

	void setLg(int x){lastgood = x;}
	void setMid(int x){mid = x;}
	void setThresh(int x){thresh = x;}
};

#endif
```

Controldat.h

```
#ifndef controldat_struct
#define controldat_struct
#include <windows.h>

class controldat
{
public:
	char buttons;
	int track[2],joy[2],accel[3],scroll[2];
};

#endif
```

Camera.h

```
/***********Camera Class***************
*Author: Roger Burns
*Adapted from "Computer Graphics Using OpenGL"
*/

#ifndef camera_env
#define camera_env

#include <stdio.h>
#include <stdlib.h>
#include <vector>


#include "Point3.h"

using namespace std;

class Camera{

public:
	//default constructor
	Camera();

	//similar to gluLookAt()
	void set(Point3 eye, Point3 look, Vector3
up);
	//camera movement
	void roll(float angle);
	void yaw(float angle);
	void pitch(float angle);
	void move(float delU, float delV, float
delN);
	void setShape(float vAng, float asp, float
nearD, float farD);

private:
```

```
	void setModelViewMatrix();
	Point3 eye;
	Vector3 u,v,n;
	double viewAngle, aspect, nearDost,farDist;
};
#endif
```

Camera.cpp

```
//Camera implementation
#include "camera.h"
#include "Point3.h"
#include <windows.h>
#include <gl/GL.h>		// Header File For
The OpenGL32 Library
#include <gl/glut.h>		// Header File For
The GLut Library
#include <gl/GLU.h>
#include <math.h>

#define PI 3.14159265
#define RAD 3.14159265/180

Camera::Camera()
{}
void Camera::move(float delU, float delV, float
delN)
{
	eye.x += delU * u.x + delV * v.x + delN *
n.x;
	eye.y += delU * u.y + delV * v.y + delN *
n.y;
	eye.z += delU * u.z + delV * v.z + delN *
n.z;
	setModelViewMatrix();
}
void Camera::pitch(float angle)
{
	float cs = cos(RAD * angle);
	float sn = sin(RAD * angle);
	Vector3 t = v;
	v.set(cs*v.x - sn*n.x, cs*v.y - sn*n.y,
cs*v.z - sn*n.z);
	n.set(sn*v.x + cs*n.x, sn*v.y + cs*n.y,
sn*v.z + cs*n.z);
	setModelViewMatrix();
}
void Camera::yaw(float angle)
{
	float cs = cos(RAD * angle);
	float sn = sin(RAD * angle);
	Vector3 t = u;
	u.set(cs*t.x - sn*n.x, cs*t.y - sn*n.y,
cs*t.z - sn*n.z);
	n.set(sn*t.x + cs*n.x, sn*t.y + cs*n.y,
sn*t.z + cs*n.z);
	setModelViewMatrix();
}
void Camera::roll(float angle)
{
	float cs = cos(RAD * angle);
	float sn = sin(RAD * angle);
	Vector3 t = u;
	u.set(cs*t.x - sn*v.x, cs*t.y - sn*v.y,
cs*t.z - sn*v.z);
	v.set(sn*t.x + cs*v.x, sn*t.y + cs*v.y,
sn*t.z + cs*v.z);
	setModelViewMatrix();
}
void Camera::set(Point3 Eye, Point3 look, Vector3
up)
{
	eye.set(Eye);
```

```
      n.set(eye.x - look.x, eye.y - look.y, eye.z -
look.z);
      u.set(up.cross(n));
      n.normalize();
      u.normalize();
      v.set(n.cross(u));
      setModelViewMatrix();
}
void Camera::setShape(float vAng, float asp, float
nearD, float farD)
{
      gluPerspective(vAng,asp,nearD,farD);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
}
void Camera::setModelViewMatrix()
{
      float m[16];
      Vector3 eVec(eye.x,eye.y,eye.z);
      m[0] = u.x; m[4] = u.y; m[8] = u.z; m[12] = -
eVec.dot(u);
      m[1] = v.x; m[5] = v.y; m[9] = v.z; m[13] = -
eVec.dot(v);
      m[2] = n.x; m[6] = n.y; m[10]= n.z; m[14] = -
eVec.dot(n);
      m[3] = 0;   m[7] = 0;   m[11]= 0;   m[15] =
1.0;
      glMatrixMode(GL_MODELVIEW);
      glLoadMatrixf(m);
}
```

Env.h

```
#ifndef controller_env
#define controller_env

#include <stdio.h>
#include <stdlib.h>
#include <vector>

/*Defined Classes*/
#include "object.h"
#include "camera.h"
#include "CONTROLDAT.h"
#include "CONTROLLER.h"

using namespace  std;

class env
{
      public:
      //Control Inforamation based upon input
      controldat input;
      control accelx,accely,accelz;

      //camera structure
      Camera cam;

      Point3 eye;
      Point3 look;
      Vector3 up;

      //objects
      sphere planet;
      sphere planet2;
      sphere sun;
      pyramid tut;
      pyramid tut2;
      cube romulan;
      cube romulan2;

      //selector object
      selector sel;

      /*constructor*/
      env();
```

```
	/*Environment Variables*/
	float lookrot,lastMoveX,lastMoveY,lastMoveZ;
	int init,maxrate,minrate;

	/*object funtions*/
	void updateObj(int x, int y, int z, int rotx,
int roty, int rotz, int sx, int sy, int sz);
	void updateSel();
	void envDraw();


	//object selection
	void selectObj(int i);

	//control information
	int accelxSum,accelySum,accelzSum;
	void getControlInfo(controldat info){input =
info;}
	void updateEnv();
	float movingAvg(int sample,control * cont);
	void initAccel();
	private:
};

#endif
```

Env.cpp

```
#include <windows.h>
#include <iostream>
#include <fstream>
#include <gl/GL.h>            // Header File For
The OpenGL32 Library
#include <gl/glut.h>          // Header File For
The GLut Library
#include <gl/GLU.h>
#include <math.h>
#include "object.h"
#include "env.h"
#include "camera.h"
#include "CONTROLLER.h"
/*Special Keys*/

const char BUTTON_1 = ( 0x01 );
	const char BUTTON_2 = ( 0x02 );
	const char BUTTON_3 = ( 0x04 );
	const char BUTTON_4 = ( 0x08 );

//only to be called once at startup of environment
env :: env()
{
	init = 1;
	//init the camera
	eye.x = 0;
	eye.y = 0;
	eye.z = -5.0;
	look.x =look.y =look.z = 0;
	up.x = up.z = 0;
	up.y = 1;
	cam.set(eye,look,up);
	cam.setShape(60.0, 680.0f/480.0f, 1.0,
2000.0);

	//init one of each shape
	planet.setPos(.2,10,-5);
	planet.setType(0);
	planet.setColor(0,.8,.2);
	planet.temppx = planet.temppy = 0;
	tut.setPos(1,5,0);
	tut.setType(0);
	tut.setColor(0.05,.8,.8);
	tut.setRot(0,0,0);
	tut.temprx = tut.tempry = tut.temprz = 0;
	romulan.setPos(3,3,0);
	romulan.setType(0);
```

```
	romulan.setColor(0.2,0.18,0.16);

	accelxSum = accelySum = accelzSum = 0;

	//init the various controller information
	accelx.mid = 2048;
	accelx.thresh = 30;
	accelx.lastgood = 2048;
	accelx.minval = 1024;
	accelx.maxval = 3072;
	accelx.lastMoveRate = 0.0;

	accely.mid = 2348;
	accely.thresh = 30;
	accely.lastgood = 2048;
	accely.lastMoveRate = 0.0;
	accely.minval = 2048;
	accely.maxval = 3072;

	accelz.lastgood = 2048;
	accelz.mid = 2048;
	accelz.thresh = 30;
	accelz.lastMoveRate = 0.0;
	accelz.minval = 2048;
	accelz.maxval = 3072;

	maxrate = 5;
}


void env::envDraw()
{

	//draw grid
    for( int x = -100; x < 100 ; x++ )
	  {
		for(int z = - 100; z < 100; z++)
		{
			glBegin( GL_LINES );
			  glVertex3d( 100, 0, z );
			  glVertex3d( -100, 0, z );
			glEnd( );
			}
		}


		planet.Draw();
		tut.Draw();
		romulan.Draw();

}

void env::updateEnv()
{
	//check Buttons
	if((input.buttons & BUTTON_1))
		planet.setColor(0,.2,.8);
	else
		planet.setColor(0,.8,.2);
	if((input.buttons & BUTTON_2))
		tut.setColor(1.0,1.0,1.0);
	else
		tut.setColor(.5,.8,.8);
	if((input.buttons & BUTTON_3))
		romulan.setColor(0.16,0.2,0.8);
	else
		romulan.setColor(.2,.8,.16);
	//if((input.buttons & BUTTON_4))
		//printf("Change Color of Diamond\n");
	//else
		//printf("Default Diamond Color\n");

	if(init == 0)
	{
		planet.temppx = input.scroll[0];
		planet.temppy = input.scroll[1];
```

```
            planet.temprx = input.track[0];
            planet.tempry = input.track[1];

            /*temp mid setting
            accelx.mid = input.accel[0];
            accelz.mid = input.accel[1];
            accely.mid = input.accel[2];*/
      }
      else
      {
      //check the scroll wheel
      planet.getDeltTran(input.scroll[0],input.scro
ll[1],init);
      //check the joystick
      romulan.setRot((input.joy[0]-
7),(input.joy[1]-7),0);
      //check the trackball
      tut.getDeltRot(input.track[0],input.track[1])
;
      //check the accel (if - button pressed then
rotate, else translate
      if((input.buttons & BUTTON_1))
      {

      cam.yaw(movingAvg(input.accel[0],&accelx));

      cam.pitch(movingAvg(input.accel[1],&accely));
      }
      else if((input.buttons & BUTTON_2))
      {

      cam.move(movingAvg(input.accel[0],&accelx),mo
vingAvg(input.accel[2],&accelz),(-
1)*movingAvg(input.accel[1],&accely));
      }
      }
}
float env::movingAvg(int sample,control * cont)
{
      int temp = abs(sample - cont->lastgood);
      int zero1 = ((cont->mid) - (cont->thresh)*2);
      int zero2 = ((cont->mid) + (cont->thresh)*2);
      if((sample > zero1) && (sample<zero2))
      {
            printf("Zero Zone\n");
            cont->lastMoveRate = 0.0;
            return cont->lastMoveRate;
      }
      else
      {
            if(temp >cont->thresh)
            {
                  int value = sample - cont->mid;
                  cont->lastMoveRate =
((value*maxrate*1.0)/(cont->maxval - cont-
>minval));
                  printf("temp: %d Sample: %d  Last
Good: %d Value: %d Mid: %d Move Rate:
%f\n",temp,sample,cont->lastgood,value,cont-
>mid,cont->lastMoveRate);
                  cont->lastgood = sample;
                  return cont->lastMoveRate;
            }
            else
            {
                  printf("Move Rate: %f\n",cont-
>lastMoveRate);
                  return cont->lastMoveRate;
            }
      }
}

void env::initAccel()
{
      accelxSum += input.accel[0];
      accelySum += input.accel[1];
```

```
	accelzSum += input.accel[2];

	accelx.mid = (accelxSum / (init-5));
	accely.mid = (accelySum / (init-5));
	accelz.mid = (accelzSum / (init-5));
	printf("XSum: %d  YSum: %d  ZSum:
%d\n",accelxSum,accelySum,accelzSum);
	printf("X.mid: %d  Y.mid: %d  Z.mid:
%d\n",accelx.mid,accely.mid,accelz.mid);
}
```

Object.h

```
#ifndef controller_object
#define controller_object

#include <windows.h>
#include <iostream>
#include <gl/GL.h>			// Header File For
The OpenGL32 Library
#include <gl/glut.h>		// Header File For
The GLut Library
#include <gl/GLU.h>
#include <stdio.h>
#include <stdlib.h>
#include <vector>
using namespace  std;

/*Class - object
*
*This serves as the base class for any object that
we look to make in our
*environment. It holds all the variables that we're
looking for such as size and
*color, providing setters and getters for each.
This will allow the subclasses
*that deal with specific shapes to concentrate on
the shape itself and let the
*object class worry about the specifics.
*/
class object
{
	public:
	//Properties of an object

	//Draw type of object
	// type == 1	mesh
	// type == 0	solid
	int drawType;

	//type of object
	//0 == generic object
	//1 == pyramid
	//2 == cube
	//3 == sphere
	int shapeType;

	//integers that represent the color of the
object
	float r,g,b;

	//relative size in "units" of the object.
	float sizeUnits;

	//scale of the object
	float sx, sy, sz;

	//position of the object
	float posx, posy, posz, temppx,temppy;

	//rotation of the object
	float rx,ry,rz,temprx,tempry,temprz;

	//constructor of any object
	object(){
		setScale(1.0,1.0,1.0);
```

```
			setPos(0.0,0.0,0.0);
			setRot(0.0,0.0,0.0);
			setColor(1,1,1);
			setSize(1);
			setType(1);
		}
		//setters and getters
		void setPos(float x, float y, float
z){posx=x;posy=y;posz=z;}
		void setColor(float cr, float cg, float
cb){r=cr;g=cg;b=cb;}
		void setSize(float size){sizeUnits = size;}
		void setScale(float x, float y,float
z){sx=x;sy=y;sz=z;}
		void setRot(float x, float y, float
z){rx+=x;ry+=y;rz+=z;}
		void setType(int x){drawType = x;}
		void getDeltRot(int x, int y);
		void getDeltTran(int x, int y,int init);

		//default draw
		void Draw(){printf("This is a typical
object...set it's type");}

		int getRed(){return r;}
		int getBlue(){return b;}
		int getGreen(){return g;}
		void printColor(){cout<<"\nRed :
"<<r<<"\nGreen : "<<g<<"\nBlue : "<<b;}
		float getSize(){return sizeUnits;}
};
/****************************Pyramid*************
****************************/
class pyramid:public object
{
	public:
		pyramid();
		void Draw();
};

/****************************Cube****************
****************************/
class cube: public object
{
	public:
	cube();
	void Draw();
};

/****************************Sphere**************
****************************/
class sphere:public object
{
	public:
	sphere();
	void Draw();
};

/***************************Selector*************
**************************
*The selector is a special object, but can utilize
the same variables that
*a regular object uses.
*
*posx - used to set the base of the selector.
*
*/
class selector:public object
{
	public:
		float endx,endy,endz;

	selector();
	void Draw();
};
#endif
```

Object.cpp

```
#include <windows.h>
#include <gl/GL.h>           // Header File For
The OpenGL32 Library
#include <gl/glut.h>         // Header File For
The GLut Library
#include <gl/GLU.h>
#include <vector>
#include "object.h"

void object::getDeltRot(int x, int y)
{
      //set rx to the delta between the new and
previous values
      int delt1 = (temprx - x);
      int delt2 = (tempry - y);

      rx += delt1;
      ry += delt2;

      temprx = x;
      tempry = y;

      //printf("Rotate Value (X):  %d   (Y):
%d",delt1,delt2);
}

void object::getDeltTran(int x, int y, int init)
{
      //set position of an object
      int delt1;
      if(x>temppx){
            delt1 = (x - temppx);
      }
      else
      {
            delt1 = (-1)*(temppx - x);
      }
      int delt2;
      if(y>temppy)
      {
            delt2 = (y-temppy);
      }
      else
      {
            delt2 = (-1)*(temppy - y);
      }
      //printf("Trans Value (X):  %d   (Y):
%d\n",delt1,delt2);
      if(init >3)
      {
            posx +=delt1;
            posy +=delt2;
      }
      temppx = x;
      temppy = y;

}

/****************************Pyramids!!!!!*********
****************************/
/* ---------------------------------------------------
----------------------- */
/* Function    : void pyramid()
 *
 * Description : This is the constructor for the
pyramid class. It provides
                        base size and color for the
pyramid. **NOTE** This does not
                        draw a pyramid
 *
 * Parameters  : void
 *
 * Returns     : void
 */
```

```
 pyramid :: pyramid()
 {
      shapeType = 1;
 }

/* --------------------------------------------------
----------------------- */
/* Function    : void Drawpyramid()
 *
 * Description : This function draws the pyramid to
the screen. It uses the
                        member variables from the
base object class.
 *
 * Parameters  : void
 *
 * Returns     : void
 */
void pyramid :: Draw()
{
  glPushMatrix();
        glTranslatef(posx,posy,posz);
        glScalef(sx,sy,sz);
        glRotated(rx,1,0,0);
        glRotated(ry,0,1,0);
        if(drawType == 0)
            glBegin(GL_TRIANGLES);
        else if(drawType == 1)
              glBegin(GL_POLYGON);
                              // start drawing a
pyramid
        glColor3f(r,0,0);
        glVertex3f(sizeUnits, sizeUnits,
sizeUnits);           // Top of pyramid (front)
        glVertex3f(-sizeUnits,-sizeUnits,
sizeUnits);             // left of pyramid (front)
        glVertex3f(sizeUnits,-sizeUnits,
sizeUnits);           // right of traingle (front)


        // right face of pyramid
        glColor3f(0,g,0);
        glVertex3f( sizeUnits, sizeUnits,
sizeUnits);             // Top Of pyramid (Right)
        glVertex3f( sizeUnits,-sizeUnits,
sizeUnits);             // Left Of pyramid (Right)
        glVertex3f( sizeUnits,-sizeUnits, -
sizeUnits);       // Right Of pyramid (Right)

        // back face of pyramid
        glColor3f(0,0,b);
        glVertex3f( sizeUnits, sizeUnits,
sizeUnits);             // Top Of pyramid (Back)
        glVertex3f( sizeUnits,-sizeUnits, -
sizeUnits);       // Left Of pyramid (Back)
        glVertex3f(-sizeUnits,-sizeUnits, -
sizeUnits);       // Right Of pyramid (Back)

        // left face of pyramid.
        glColor3d(r,0,0);
        glVertex3f( sizeUnits, sizeUnits,
sizeUnits);             // Top Of pyramid (Left)
        glColor3d(0,g,0);
        glVertex3f(-sizeUnits,-sizeUnits,-
sizeUnits);             // Left Of pyramid (Left)
        glColor3d(0,0,b);
        glVertex3f(-sizeUnits,-sizeUnits,
sizeUnits);             // Right Of pyramid (Left)
        glEnd();
  glPopMatrix();
}
```

```
/**************************Cubes!!!**************
**********************/
/* ---------------------------------------------------
----------------------- */
/* Function    : void cube()
 *
 * Description : This is the constructor for the
cube class. It provides
                        base size and color for the
cube. **NOTE** This does not
                        draw a cube
 *
 * Parameters  : void
 *
 * Returns     : void
 */
 cube :: cube()
 {
      shapeType = 1;
 }

/* ---------------------------------------------------
----------------------- */
/* Function    : void Drawcube()
 *
 * Description : This function draws the cube to
the screen. It uses the
                        member variables from the
base object class.
 *
 * Parameters  : void
 *
 * Returns     : void
 */
void cube :: Draw()
{
  glRotated(rx,1,0,0);
  glRotated(ry,0,1,0);
  glRotated(rz,0,0,1);
glBegin(GL_QUADS);                        // start
drawing the cube.

  glColor3d(r,g,b);
  // top of cube
  glVertex3f( sizeUnits, sizeUnits,-sizeUnits);
      // Top Right Of The Quad (Top)
  glVertex3f(-sizeUnits, sizeUnits,-sizeUnits);
      // Top Left Of The Quad (Top)
  glVertex3f(-sizeUnits, sizeUnits, sizeUnits);
      // Bottom Left Of The Quad (Top)
  glVertex3f( sizeUnits, sizeUnits, sizeUnits);
      // Bottom Right Of The Quad (Top)

  // bottom of cube
  glVertex3f( sizeUnits,-sizeUnits, sizeUnits);
      // Top Right Of The Quad (Bottom)
  glVertex3f(-sizeUnits,-sizeUnits, sizeUnits);
      // Top Left Of The Quad (Bottom)
  glVertex3f(-sizeUnits,-sizeUnits,-sizeUnits);
      // Bottom Left Of The Quad (Bottom)
  glVertex3f( sizeUnits,-sizeUnits,-sizeUnits);
      // Bottom Right Of The Quad (Bottom)

  // front of cube
  glVertex3f( sizeUnits, sizeUnits, sizeUnits);
      // Top Right Of The Quad (Front)
  glVertex3f(-sizeUnits, sizeUnits, sizeUnits);
      // Top Left Of The Quad (Front)
  glVertex3f(-sizeUnits,-sizeUnits, sizeUnits);
      // Bottom Left Of The Quad (Front)
  glVertex3f( sizeUnits,-sizeUnits, sizeUnits);
      // Bottom Right Of The Quad (Front)

  // back of cube.
  glVertex3f( sizeUnits,-sizeUnits,-sizeUnits);
      // Top Right Of The Quad (Back)
```

```
  glVertex3f(-sizeUnits,-sizeUnits,-sizeUnits);
      // Top Left Of The Quad (Back)
  glVertex3f(-sizeUnits, sizeUnits,-sizeUnits);
      // Bottom Left Of The Quad (Back)
  glVertex3f( sizeUnits, sizeUnits,-sizeUnits);
      // Bottom Right Of The Quad (Back)

  // left of cube
  glVertex3f(-sizeUnits, sizeUnits, sizeUnits);
      // Top Right Of The Quad (Left)
  glVertex3f(-sizeUnits, sizeUnits,-sizeUnits);
      // Top Left Of The Quad (Left)
  glVertex3f(-sizeUnits,-sizeUnits,-sizeUnits);
      // Bottom Left Of The Quad (Left)
  glVertex3f(-sizeUnits,-sizeUnits, sizeUnits);
      // Bottom Right Of The Quad (Left)

  // Right of cube
  glVertex3f( sizeUnits, sizeUnits,-sizeUnits);
// Top Right Of The Quad (Right)
  glVertex3f( sizeUnits, sizeUnits, sizeUnits);
      // Top Left Of The Quad (Right)
  glVertex3f( sizeUnits,-sizeUnits, sizeUnits);
      // Bottom Left Of The Quad (Right)
  glVertex3f( sizeUnits,-sizeUnits,-sizeUnits);
      // Bottom Right Of The Quad (Right)
  glEnd();                              // Done Drawing
The Cube
}
/****************************Spheres!!!!!**********
**************************/
/* --------------------------------------------------
----------------------- */
/* Function    : void sphere()
 *
 * Description : This is the constructor for the
sphere class. It provides
                        base size and color for the
sphere. **NOTE** This does not
                        draw a sphere
 *
 * Parameters  : void
 *
 * Returns     : void
 */
 sphere :: sphere()
 {
      shapeType = 1;
 }

/* --------------------------------------------------
----------------------- */
/* Function    : void Drawsphere()
 *
 * Description : This function draws the sphere to
the screen. It uses the
                        member variables from the
base object class.
 *
 * Parameters  : void
 *
 * Returns     : void
 */

void sphere :: Draw()
{
      switch(drawType)
      {
            case 0:
                  //printf("Solid Sphere drawn\n");
                  glPushMatrix();

      glColor3f((float)r,(float)g,(float)b);

      glTranslatef(posx,posy,posz);
```

```
                              glScalef(1,1,1);

      glutSolidSphere(sizeUnits,10,10);
                  glPopMatrix();
                  break;

            case 1:
                  //printf("Wire Sphere drawn\n");
                  glPushMatrix();
                  glColor3f(r,g,b);

      glTranslatef(posx,posy,posz);
                        glScalef(sx,sy,sz);

      glutWireSphere(sizeUnits,10,10);
                  glPopMatrix();
                  break;
            default:
                  cout<<"Wrong parameters passed to
Draw\n";
                  break;
      }
}



selector :: selector()
{

}
void selector :: Draw()
{
      GLUquadricObj *quadric;
      quadric = gluNewQuadric();
      glPushMatrix();
      glScalef(.25,.25,1);
      gluCylinder(quadric, 1, 0.75, 1, 15, 15);
      glPopMatrix();
}
```

Point3.h

```
/*********Point3 and Vector3*************
*
*
*
*/

#ifndef point3_env
#define point3_env

#include <stdio.h>
#include <stdlib.h>
#include <vector>

using namespace std;

class Point3{
public:
      float x,y,z;
      void set(float dx, float dy, float dz){x =
dx; y = dy; z = dz;}
      void set(Point3& p){x= p.x;y =p.y; z = p.z;}
      Point3(float xx, float yy, float zz){x = xx;
y=yy; z=zz;}
      Point3(){x=0;y=0;z=0;}
};


class Vector3{
public:
      float x,y,z;
      void set(float dx, float dy, float
dz){x=dx;y=dy;z=dz;}
      void set(Vector3& v){x=v.x;y=v.y;z=v.z;}
```

```
	void setDiff(Point3& a, Point3& b){x=a.x-
b.x;y=a.y-b.y;z=a.z-b.z;}
	void normalize();
	Vector3(float xx, float yy, float zz){x =
xx;y=yy;z=zz;}
	Vector3(Vector3& v){x=v.x;y=v.y;z=v.z;}
	Vector3(){x=y=z=0;}
	Vector3 cross(Vector3& b);
	float dot(Vector3& b);
};
#endif
```

Point3.cpp

```
#include "Point3.h"
#include <math.h>
#include <stdlib.h>
#include <windows.h>
#include <assert.h>
#include <iostream>

float Vector3::dot(Vector3& b)
{
	return(x * b.x + y * b.y + z * b.z);
}

Vector3 Vector3::cross(Vector3 &b)
{
	Vector3 c(y*b.z - z*b.y, z*b.x - x*b.z, x*b.y
- y*b.x);
	return c;
}

void Vector3::normalize()
{
	double sizeSq = x * x + y * y + z * z;
	if(sizeSq < 0.0000001)
	{
		cerr<<"\nnormalize() see
vector(0,0,0)!";
		return;
	}
	float scaleFactor = 1.0/(float)sqrt(sizeSq);
	x *= scaleFactor;
	y *= scaleFactor;
	z *= scaleFactor;
}
```

Test.cpp

```
#include "env.h"
#include "object.h"
#include "serial.h"
#include "parse.h"
#include "camera.h"

#include <windows.h>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <gl/GL.h>			// Header File For
The OpenGL32 Library
#include <gl/glut.h>			// Header File For
The GLut Library
#include <gl/GLU.h>
#include <math.h>
#include <time.h>

using namespace std;

//ASCII codes for special keys
#define ESCAPE 27
#define PAGE_UP 73
#define PAGE_DOWN 81
#define UP_ARROW 72
```

```
#define DOWN_ARROW 80
#define LEFT_ARROW 75
#define RIGHT_ARROW 77

/********************Serial Port
Variables************************
*
*/
HANDLE comPort = NULL;
char buffer[19];
parse * Parser = new parse();

/*********************Graphics
Setup******************************
*
*
*
*/
float angle = 0.0;
int dist = 0;
env ement;
float x=0,y=0,z=0;

void init()
{
	int x=2;
	//get the user input for what Port the
controller is located on
	printf("\nPlease indicate what port
(numerical value) the controller is on: ");
	cin>>x;
	//set the serial port in the Parser
	Parser->setPort(x);
	//open the serial port
	Parser->init();
	//init the graphics
	glClearColor(0.0,0.0,0.0,0.0);
	glShadeModel(GL_FLAT);
}
void display(void)
{
	//display the graphics
	glClear(GL_COLOR_BUFFER_BIT);
	glColor3f(1.0,1.0,1.0);
	glPushMatrix();
	ement.envDraw();
	glPopMatrix();
	glutSwapBuffers();
	glFlush();
}

void reshape(int w, int h)
{
	glViewport(0,0,(GLsizei) w, (GLsizei) h);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	ement.cam.setShape(60.0, (GLfloat)
w/(GLfloat) h, 1.0, 2000.0);
	//gluPerspective(60.0, (GLfloat) w/(GLfloat)
h, 1.0, 2000.0);
	//glMatrixMode(GL_MODELVIEW);
	//glLoadIdentity();
}

void serialRead()
{
	//get time for serialRead()
	clock_t t1 = clock();
	if(t1==clock_t(-1))
		{
			cerr<<"clock overflow\n";
			exit(2);
		}
	//check to see if you need to update the
variables
	if(Parser->parseBuff() == 1)
```

```
	{
		clock_t t2 = clock();
		if(t2==clock_t(-1))
		{
			cerr<<"clock overflow\n";
			exit(2);
		}
		//double d = difftime(t2,t1);
		//printf("Amount of time for
serialRead, parsebuff,varUpdate: ");
		//cout<<double(t2-t1)<<"seconds\n";
	}
}
void TimerCallback( int value )  {
 //force a poll
	if(Parser->poll())
	{
	  //read in from the serial Port
	  serialRead();
	  //pass the information to the environment
	  ement.getControlInfo(Parser->input);
	  while(ement.init < 30)
	  {
		 //read in from the serial Port
		serialRead();
		//pass the information to the
environment
		ement.getControlInfo(Parser->input);
		ement.updateEnv();
		if(ement.init > 5)
		{
		    ement.initAccel();
		}
		ement.init++;
		Sleep(100);
	  }
	  ement.updateEnv();
		ement.init++;
	    }
	// Force a redraw.
	glutPostRedisplay();
	//finish the timer - used for baud rate check

	//calculate time

	// Set it to wake us again.
	glutTimerFunc( 100, TimerCallback, 1 );
}

void keyboard(unsigned char key, int x, int y)
{
	switch(key) {
		case 'q':
			ement.cam.roll(0.5);
			break;
		case 'e':
			ement.cam.roll(-0.5);
			break;
		case 'a':
			ement.cam.yaw(1);
			break;
		case 'd':
			ement.cam.yaw(-1);
			break;
		case 'w':
			ement.cam.pitch(-1);
			break;
		case 's':
			ement.cam.pitch(1);
			break;
		case 'i':
			ement.cam.move(0,0,-1);
			break;
		case 'j':
			ement.cam.move(-1,0,0);
```

```
				break;
			case 'l':
				ement.cam.move(1,0,0);
				break;
			case 'k':
				ement.cam.move(0,0,1);
				break;
			case 'u':
				ement.cam.move(0,1,0);
				break;
			case 'o':
				ement.cam.move(0,-1,0);
				break;
			case 'v':
				ement.planet2.posx--;
				break;
			case 'V':
				ement.romulan2.posy--;
				break;
			case 'z':
				ement.accelz.thresh += 10;
				printf("Threshold(Z):
%d\n",ement.accelz.thresh);
				break;
			case 'Z':
				ement.accelz.thresh -= 10;
				printf("Threshold(Z):
%d\n",ement.accelz.thresh);
				break;
			case 'x':
				ement.accelx.thresh += 10;
				printf("Threshold(X):
%d\n",ement.accelx.thresh);
				break;
			case 'X':
				ement.accelx.thresh -= 10;
				printf("Threshold(X):
%d\n",ement.accelx.thresh);
				break;
			case 'y':
				ement.accely.thresh += 10;
				printf("Threshold(Y):
%d\n",ement.accely.thresh);
				break;
			case 'Y':
				ement.accely.thresh -= 10;
				printf("Threshold(Y):
%d\n",ement.accely.thresh);
				break;
			case ESCAPE:
				exit(-1);
	}
	glutPostRedisplay();
}

int main(int argc, char* argv[])
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
	glutInitWindowSize(500,500);
	glutInitWindowPosition(100,100);
	glutCreateWindow(argv[0]);
	init();
	glutDisplayFunc(display);
	glutReshapeFunc(reshape);
	glutKeyboardFunc(keyboard);
	//Use the idle func to run through the input
	//from the controller on the serial line
	glutTimerFunc(1,TimerCallback,1);
	glutMainLoop();
	return 0;
}
```