Motion Activated Arduino Car (MAAC)

Engineering Science 50 Final Project

Saagar Deshpande '14 and Randy Miller '13

May 1, 2012

Abstract

Our project combined multiple technologies in order to create a motion controlled car. The idea behind the project was to interface with a Microsoft Xbox 360 Kinect to control an Arduino Nano board, which in turn powered the motors of a BOE-bot vehicle. In our final project, we were able to connect these various systems together, pass information between the Kinect and the Arduino via the serial data connections, and move the car with various hand motions. We modularized our systems so that we would be able to interchange specific systems as necessary. This allows anyone to update the implementation and use different vehicles so long as the Arduino and serial connections are maintained. While our project successfully combined these systems together to create a working Motion Activated Arduino Car, we feel that there is a solid room for improvement and addition to both the hardware and software systems implemented. On the hardware side, we would like to have used more precise motors rather than the BOE-bot servo motors, add sensors for environment detection, and a wireless camera for the driver to see the vehicle's movements while using the Kinect MAAC application from a distant location. On the software, given that the hardware additions can be made, we would like to be able to implement an artificial intelligence to control the vehicle and create a tuning mode to optimize the vehicle being used as the MAAC.

1 Introduction

The Motion Activated Arduino Car (MAAC) is an Arduino powered BOE-bot car that is controlled by motion sensing via the Microsoft Kinect. The goal of our project was to be able to successfully use voice and motion commands to power and control the servos motors on the BOE-bot, which in turn would determine the speed and direction of the car's motion. The basic operating principle is to be able to use the Kinect to control the car virtually, with the effect that the car would respond as if it were a normal remote-controlled car.

Our project has three major components: the BOE-bot car, the receiver, and the transmitter. First, the car consists of a circuit that wires the Arduino Uno to the motors on the BOE-bot car. Next, the receiver involves Arduino code which captures data sent via serial input and translates this information into commands for the motors on the MAAC. Finally, the transmitter is a Kinect C# application developed in Visual Studio which translates hand positions of a user into byte packets which are sent over a serial connection to the Arduino to process into motion. This serial connection can also be emulated by a wireless X-Bee connection, in effect enabling wireless communication between the application and the MAAC.

The hardware side of the project is relatively simple, requiring simple circuitry to repurpose the BOE-bot for use with the Arduino Uno. Additional circuitry was required to interface with X-Bee controllers for wireless communication. The software side was more complex, requiring code platforms for both the Kinect and the Arduino. In the end, we were able to create this multi-system project which synchronizes correct and allows the user to virtually drive a robot car.

2 Specifications/Objectives

- MAAC can move in its full capacity.
- MAAC is fully controlled by the Arduino Uno.
- MAAC is human controlled by the Kinect using a custom C# application.
- Kinect application successfully interfaces with the Arduino via serial data connection.
- Kinect application successfully communicates to the Arduino Uno via wireless X-Bee serial data transmission.

3 Project Description

First, a demo of our final project can be found at http://youtu.be/4v98L51F9Vw.

As mentioned in the introduction, the Kinect MAAC application consists of three components: the car, the receiver, and the transmitter.

To make the car, we modified a BOE-bot car to be controlled by an Arduino Uno. To do this, we connected the servo motors on the car to two output pins (we chose 9 and 10) on the Arduino control board, using a bread board to pass signals and power between the board and the motors, as well as to provide us with some wire management. We used batteries to provide us with a consistent 5 volt power source, which powers the entire car, including the Arduino. The receiver consists of the Arduino Uno reading input from a serial connection, which is then pushed out through the output pins to the servos. We developed a small Arduino program which would allow byte packets received by the Arduino to be forwarded to the servos as motor commands such that the MAAC would move in the correct direction with the correct speed.

For the transmitter, we developed a C# application in Visual Studio to interface with the Kinect device, allowing us to capture motion and voice commands and translate them into actions within the application and to serial data to be sent to the MAAC. Serial data could be sent from the application via a direct USB connection to the Arduino Uno in the receiver or via a wireless X-Bee connection which itself constitutes a serial connection. Given the hardware that we have, we also use an Arduino Nano for forwarding application data to a X-Bee; note that this could be replaced with a X-Bee USB Explorer.

The Kinect MAAC application has four screens, or “modes”: Menu, Steering, Precision, and Pod Racing. The latter three of those modes are used to drive the MAAC. Upon start up, the Kinect MAAC application begins motion sensing and voice recognition. The user can navigate to any of the three driving modes from the main menu by allowing his/her right hand to hover over one of the three menu buttons for a few seconds. Additionally, when the MAAC is not being driven, voice commands can be used to switch to any mode. The voice commands and their functions are as follows:

- Kinect menu: switch to the main menu
- Kinect steering: switch to Steering Mode
- Kinect precision: switch to Precision Mode
- Kinect pod racing: switch to Pod Racing Mode

When the MAAC is not in motion, a voice command can be used at any time to navigate between any of the modes. Each of the three driving modes has its own control interface with mode-specific driving user-controls, read-outs of the X and Y Cartesian coordinates of the user’s left and right hands, a color stream of the user overlaid with a skeleton viewer, and a stop button.

The first of these driving modes is Steering Mode, which allows the user to simulate driving the car with a virtual steering wheel. Steering mode consists of a wheel which the user must virtually grab and turn. This mode allows the user to move the MAAC as they would drive a car. The wheel motions correspond to the power of each motor. Straight-line speed cannot be changed in this mode, and is set by default to maximum speed. This mode can be somewhat difficult to control since the servos are not very sensitive to slight changes, and it is meant to showcase the speed at which data is transferred and processed. This mode would more mirror a true virtual driving experience if the servos were more sensitive to all values.

The second mode is Precision Mode, which allows the user to control the car with fine-tune driving controls. This mode is meant to show off the precision of the Kinect and multi-system MAAC. Precision mode consists of one vertical slider bar and one horizontal slider bar. The left hand locks onto the slider on the vertical bar and controls whether the car is moving forward or backward as well as how fast the car is moving. The right hand locks onto the slider on the horizontal bar and controls whether the car is turning left or right or is moving straight ahead.

The third mode is Pod Racing Mode, which allows the user to control the car with a more intuitive sense of driving. This mode is meant to be a one-to-one mapping of user input to servos commands, as there is relatively little processing utilized to compute the servos commands. Pod Racing Mode consists of two slider bars, one for the left hand and one for the right hand. Each hand corresponds to the respective motor. At the start, both sliders are in the middle of their respective bars, which means that there is no power going to either motor. If a slider is moved upwards, the corresponding motor will move forward at the speed relative to how far the slider was moved upwards. If a slider is moved downwards, the motor will move backwards relative to how the slider was moved downwards. The two slider bar control system allows each hand to control the exact speed and direction of each motor at all times, allowing the user to get a feel for the car and the servo motors. This allows the user to execute smooth turns very easily. Readers should note that Pod Racing mode was inspired by the sport of the same name in the first episode of the epic movie, “Star Wars”.

4 Project Design

As our project involved interactions between multiple types of systems, design decisions were crucial to making sure that each piece worked independently correctly and that all the pieces interfaced correctly as well. As a result, we modularized as many components of our project as possible. This also enables us to switch out different parts in the future rather easily.

4.1 High Level Design

To implement our project, we used a modified BOE-bot, an Arduino Uno, an Arduino Nano, and X-Bee controllers. We chose to use the Arduino Uno and Nano because they allowed us to easily control outputs via the pins on the boards. This essentially means that we can send signals in the form over the pins on the Arduino Uno to control anything connected to those pins. As such, we can exchange the BOE-bot with any other type of vehicle, so long as it can be connected to the output pins on the Arduino Uno. We chose to use the Arduino Nano simply to interface with the X-Bee module. With more appropriate X-Bee/Arduino shields (eg, X-Bee USB Explorer), we would be able to forgo the Arduino Nano for a slightly simpler setup.

We used the X-Bee wireless controllers to allow the MAAC to move freely when controlled by the Kinect MAAC application . This is another choice that allowed us to maintain modularity. Because the X-Bee controllers work over serial data connections, there is no difference between using the X-Bee and the physical USB connection to the Arduino Uno from a coding standpoint. Once the Arduino Uno is loaded with the correct program, we can use either the X-Bee or the USB wired connection, and the serial communication will still take place.

The pipeline of information transfer is as follows:

1. Kinect processes input from the user (motion and/or voice commands) and translates these into byte packets.
    (a) Kinect MAAC application registers input and determines action to translate
        i. Voice commands translate to mode transitions
        ii. Motion commands translate to mode transitions, driving activation, or position translations.
    (b) Kinect MAAC application executes action or creates byte packets from the positional data.
    (c) Kinect MAAC application sends byte packets to Arduino MAAC application over serial USB connection
2. The Arduino Nano receives the byte packets over the serial USB connection and forwards them to a X-Bee controller.

3. The X-Bee controller transmits the byte packets wirelessly to another X-Bee controller.
4. The receiving X-Bee controller forwards the byte packets to the Arduino MAAC application over a serial connection.
5. The Arduino MAAC application receives and parses the byte packet.
   (a) Arduino MAAC application receives byte packet in C via open serial USB connection.
   (b) Arduino MAAC application parses positional data from byte packet into commands for the motors on the BOE-bot
6. The Arduino MAAC application sends the commands to the BOE-bot motors, sending the car in the correct speed and direction.

4.2 Code Design

Control flow is as follows:

1. MainWindow constructor is called at start up.
   (a) Initializes GUI.
   (b) Initializes different mode components.
2. The main window loads, so the Window_Loaded event handler is called.
   (a) A Kinect sensor "chooser" is assigned an event handler, enabling the application to initialize Kinect sensors.
   (b) The serial port is opened, and a stop signal is sent over the serial port.
3. A Kinect is detected, so the kinectSensorChooser1_KinectSensorChanged event handler is called.
   (a) The Kinect is initialized for video and skeleton streaming.
   (b) Voice recognition is initialized.
4. A Kinect "frame" event is fired 30 times a second, so the sensor_AllFramesReady event handler is called that often.
   (a) The application grabs the skeleton of the user.
   (b) The GUI's hands/cursors are updated to match the skeleton of the user.
   (c) A "behavior" or "compute" function is called; which behavior function is called is dependent on which mode the application is currently in.
   (d) After the behavior function has finished executing, coordinate read-out boxes are updated with the coordinates of the user's hands.
5. The menu behavior function is called when the application is in Menu mode.

   (a) The menu checks to see if the user's right hand/cursor is hovering over one of the menu buttons, and calls a "button clicked" event if the cursor has been hovering over that same button for an extended amount of time.

6. The steering behavior function is only called when the application is in Steering mode.
   (a) The internal representations of the hands are updated.
   (b) When the user has put his/her hands on the wheel, then the Arduino begins to move according to the motions of the user.
   (c) When one of the user's hands touches the stop button, the Arduino stops future movement until the hands are on the wheel again.
   (d) If the user is driving the Arduino, the wheel turns with the movements of the user's hands.
   (e) If the user is driving the Arduino, the wheel rotation is transformed to servos-friendly bytes which are then sent to the Arduino.
7. The precision behavior function is only called when the application is in Precision mode.
   (a) The internal representations of the hands are updated.
   (b) When the user has put his/her hands on the sliders, then the Arduino begins to move according to the motions of the user.
   (c) When one of the user's hands touches the stop button, the Arduino stops future movement until the hands are on the wheel again.
   (d) If the user is driving the Arduino, the sliders adjust according to the movements of the user's hands.
   (e) If the user is driving the Arduino, the slider values are transformed to servos-friendly bytes which are then sent to the Arduino.
8. The pod racing behavior function is only called when the application is in Pod Racing mode which, on a high level, is completely identical to the behavior function of Precision mode.

5 Parts List

5.1 Hardware

- Microsoft Kinect
- Car
  - Servos motors (2)
  - BOE-bot chassis and tires
  - Custom mount
  - Power source

- Communication
  - X-Bee Series 1 Wireless Transmitter (2)
  - X-Bee Arduino Shield
  - X-Bee Explorer Regulated
- Receiver
  - Arduino Uno
  - Breadboard
  - Circuitry Material (wires, pins, etc.)
- Transmitter
  - Arduino Nano
  - Micro-USB cable
  - Circuitry material(wires, pins, etc.)
- Arduino Kit
  - Arduino Nano Processing Board
  - Bread board
  - Circuitry Material (wires, pins, etc.)

5.2 Software

- Microsoft Kinect SDK [4]
- Visual Studio 2010
- Arduino Uno and Nano drivers
- Arduino Integrated Development Environment
- X-CTU (for X-Bee support) [5]
- Sandcastle for documentation [8]

6 Project Implementation

Our implementation consisted of 10 stages:

1. Demo stage: a system that was able to demonstrate primitive functionality, forwarding Kinect input data to the Arduino and then to the servos
2. Menu and Steering stage: a system that consisted of a basic menu and a mode that let the user "steer" a virtual wheel to control the BOE-bot

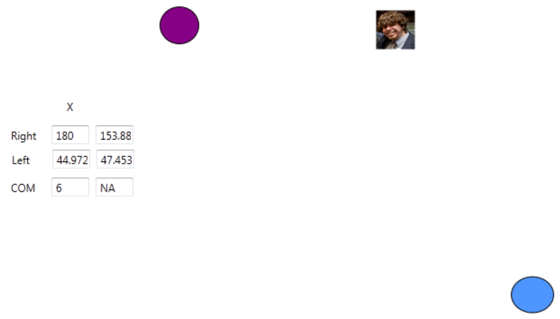
3. Precision stage: add an additional mode to the menu that let the user steer the BOE-bot using a horizontal and a vertical slider.
4. Refactoring Part I stage: split the different modes into different files
5. Pod Racing stage: add an additional mode to the menu that let the user steer the BOE-bot using 2 vertical sliders, like a Star Wars pod racer
6. Voice commands stage: add voice recognition to the application
7. Animations and Video stage: add fade in/out animations to transitions between modes
8. Refactoring Part II stage: factor out common functionality to separate functions in a new file
9. X-Bee stage: replace the USB connection with a wireless X-Bee connection
10. Polishing and Documentation stage: fix subtle bugs, further refactor code, and thoroughly comment and document code

6.1 Demo Stage

We initially set out to make a system that essentially consisted of the BOE-bot responding to any Kinect input via the Arduino and a USB serial connection. This entailed wiring up the Arduino to the BOE-bot, writing a short program for the Arduino to execute, and creating a basic C# application that forwarded Kinect input data to the Arduino. Once we figured out what the different pins on the servo motors signified, we were easily able to determine how to construct the circuit between the Arduino and the motors. We followed an online tutorial to create the Kinect application and the Arduino program[1]. We also followed the Kinect Quickstart series in order to learn more about how to use the Kinect SDK[2]. Essentially, the Kinect application did a few basic but important and useful things:

- It referenced and used KinectWPFViewers, a library that makes it easy to grab data out of the Kinect, such as skeleton data and video streams.
- It tracked the user's skeleton.
- It displayed the user's hands and head as circles and an image, respectively.
- It displayed the user's Cartesian coordinates on the window.
- It opened and successfully used serial ports for information transfer.
- It sent the user's left hand and right hand Y coordinates modulo 180 (the maximum value of the servo motors) over the serial port to the Arduino.

The Arduino program then simply wrote the given Y coordinates modulo 180 to the servos. Without even determining how the hands would affect the motors, we tested our application and were surprised to see that our system worked. We noted how the hand movement mapped to the servos, and were able to figure out what the correct input should be to the servos in order to appropriately control them (see the Piazza Posts section of the Appendix for information on how we did this). Ultimately, this stage was extremely important because it introduced us to the relevant parts of the Kinect SDK and serial port communication.



	X	
Right	180	153.88
Left	44.972	47.453
COM	6	NA

Figure 1: Demo

6.2 Menu and Steering Stage

We started a completely new C# application called ArduinoController; our focus for this stage was to create a nicer and more interactive graphical user interface (GUI) for the Kinect application. To this end, we followed an online tutorial on how to create “hover buttons” within the GUI as well as a cursor for a user’s hand[3]. We used the same style of functions to implement the first menu, which consisted of two buttons for Steering mode and Precision mode.

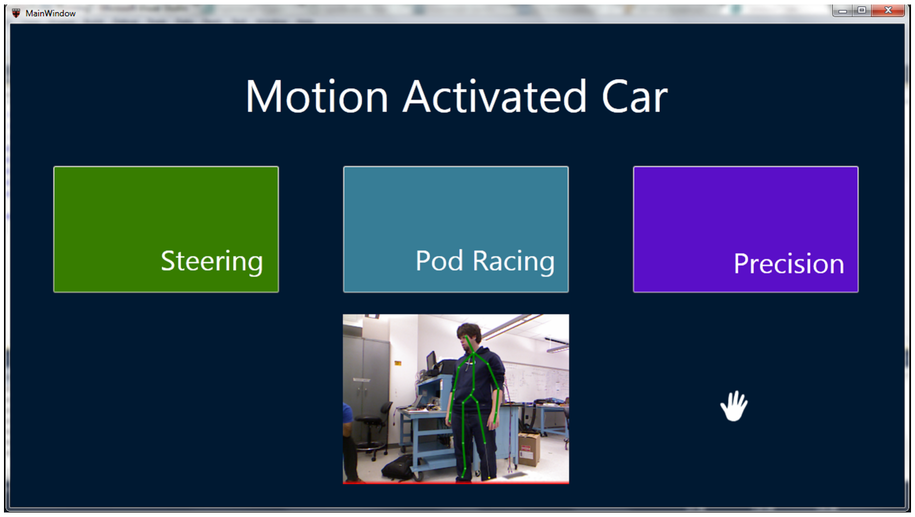
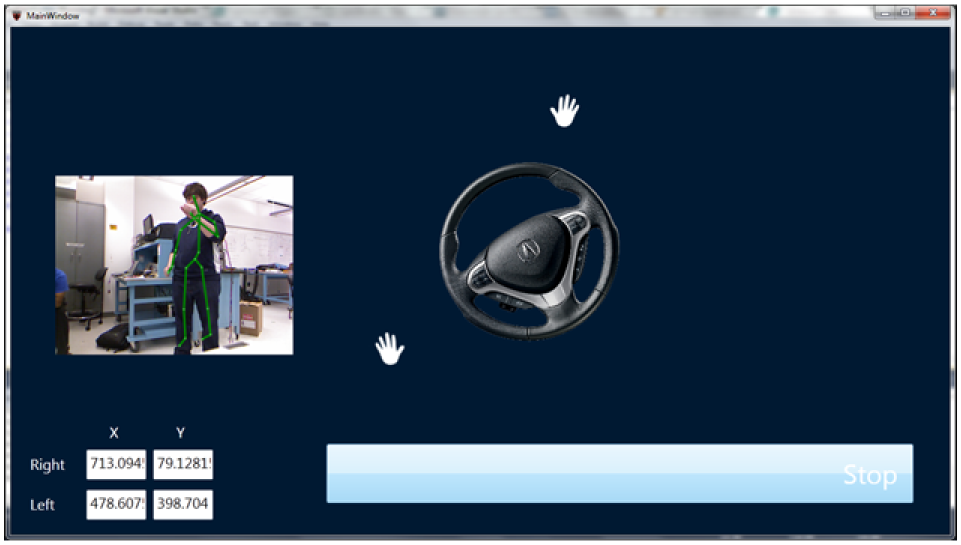


Figure 2: Menu

After implementing clicking, we then moved on to implementing Steering mode. Steering mode is actually implemented in the same class as the menu and as the main application, which in the long run was probably not a good design choice. However, at the time, implementing Steering mode in the main application class facilitated its development; in order to switch modes all we had to do was hide the menu components and make visible the Steering mode components. We were not familiar enough with the .NET platform to be able to separate the modes into different classes, so when we tried, we failed; thus, implementing the four modes in one class was not really that big of an issue.

Once we were able to successfully switch to Steering mode from the menu (ie, selecting the Steering button from the menu and hiding the menu buttons and title), we set off to implementing the functionality for Steering mode. We first added visual read-outs for the coordinates of the left and right hands, to aid in debugging. We added a wheel image for steering the car as well as a stop button that when touched, not clicked, would immediately send a stop signal to the car and would no longer send user input to the car. After much thought and many trials, we decided to "bind" the user's right hand to the wheel, such that once the user has selected the wheel with both hands, the wheel turns only when the right hand moves; moving the left hand has no effect on the wheel. Then, we implemented a function that calculated the rotation between the wheel's current position and the wheel's stationary position, which in turn rotated the wheel image on the GUI (which actually took us a while to figure out how to do). Lastly, we implemented a function that transformed that rotation angle into servos-friendly bytes that were sent to the Arduino immediately. It was at this point that we realized that the Arduino code cannot have any delays, since the Kinect sends at least 30 byte packets per second; having a delay larger than 3 milliseconds resulted in an unsynchronized system.

Ultimately, Stage 2 was important because we learned how to create GUI components and how to manipulate them, as well as created our first mode. Creating future modes then became less non-trivial.



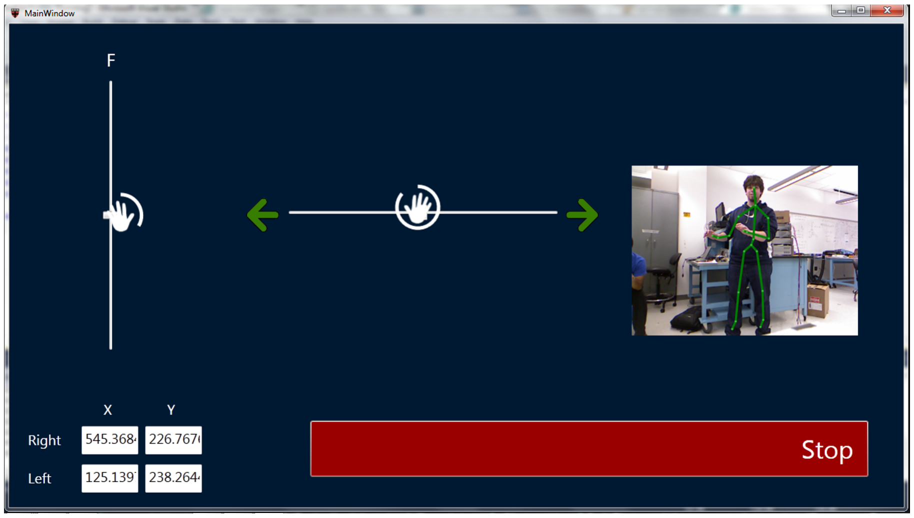
	X	Y
Right	713.094	79.1281
Left	478.607	398.704

Figure 3: Steering Mode

6.3 Precision Stage

Creating the Precision Mode was much easier once we had implemented Steering Mode. The code for transitioning from the menu to Precision mode was very similar to the transitioning code from the menu to Steering mode, and more importantly, there were no (overly) complex transformations in this stage. Once the GUI for Precision mode was set up, there were only two non-trivial parts left to implement: "binding" the hands/cursors to the sliders and transforming the slider values to servos-friendly byte packets to be sent to the Arduino.

For the binding problem, we created two invisible buttons and placed each behind one of the center of the slider bars. Thus, in order to determine if a user's hand is over the center of the slider bar or is at least near the center, we can simply ask if the user's hand is hovering over the respective invisible button.



	X	Y
Right	545.368	226.767
Left	125.139	238.264

Figure 4: Precision Mode: Binding

The transformation problem required thought because it required a means to transform speed and horizontal direction data into commands for the left and right servo motors. First, the speed and horizontal direction data were computed by finding the position of the cursor relative to the center of one of the sliders. For example, if the right hand cursor was at the $\frac{3}{4}$ mark of the horizontal slider (where the 0th mark is the very left of the slider), then the BOE-bot should turn a "half-right", or a soft right. If the cursor was at the right hand side of the horizontal slider, then the BOE-bot should turn a "full-right", or a hard right. We used the same intuition for speed: the user's relative hand position determined not only if the car was moving forwards or backwards but also how fast it was moving in that direction. With this algorithm, the BOE-bot should theoretically be able to move at varying speeds. In practice, the BOE-bot only has 2 or 3 speeds (fast, slow and stop), so this fine-grained control experience cannot be seen with our current hardware.

The speed and horizontal direction variables were then transformed to commands for the servos. In other words, to go forward, both engines should receive full "power"; to turn right, the right engine should get less "power" than the left engine, and vice versa for the BOE-bot to turn left. This intuition was captured in a short function, which would then send the resulting byte packets to the Arduino.

6.4 Refactoring Part I Stage

At this point, the code had become one gigantic file with no particular organization. We took some time to separate out functionality into different files and to reorganize control flow. Excess code from tutorials was cut from the application. At the end of this stage, the main application and the menu occupied one file, Steering mode occupied another file, and Precision mode also had its own file. Steering mode and Precision mode did not look similar at this time, even though they have extremely similar high level behavior. This was addressed later.

6.5 Pod Racing Stage

We were thinking about finishing off the project before we thought of adding a Pod Racing mode. Pod racing is a sport in the fictional world of Star Wars. It involves living beings racing against each other by driving "pods," which consist of a chassis for the person to rest in, which is then attached by steel wires to one or more engines which propel the vehicle forward. The engines and the pod stay above ground using anti-gravity technology not yet fully realized in the real world. The traditional pod racing vehicle consists of a pod and two engines.

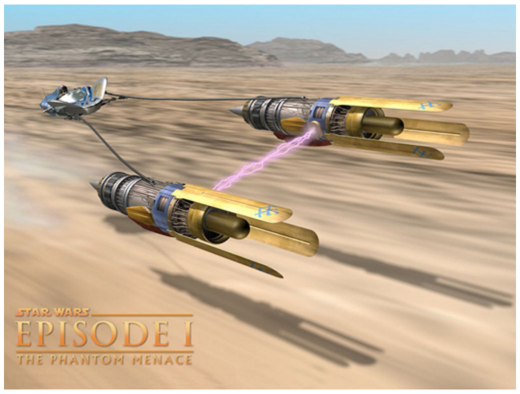


Figure 5: Traditional 2-engine Pod Racing Vehicle

The pod racing vehicle's dual engines are controlled by two levers, one for each engine. Moving a lever forward causes that respective engine to move forward at a faster speed, and moving one back causes the respective engine to slow down or even go backwards at some point.

A natural way to emulate these controls in a WPF application is to use vertical sliders: moving a slider up is equivalent to moving a pod racing lever forward, etc. Because the servos are controlled by exactly this control schema, there were no transformation functions needed to translate input data into servos-friendly byte packets. In fact, input data only needs to be scaled appropriately before it is sent to the Arduino.

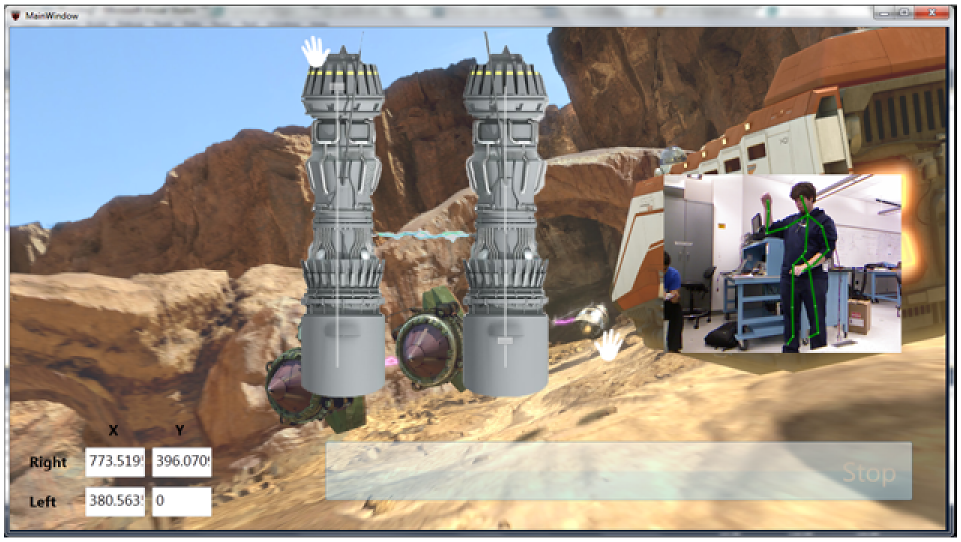


Figure 6: Pod Racing Mode

6.6 Voice Commands Stage

We did not have a natural way for users to switch modes, since we didn't want to litter the driving screens with buttons. Instead, we implemented voice recognition for mode navigation.

To do this, we followed the Voice Recognition sample very closely, which came with the Kinect SDK[4]. We copied only the necessary functions from the sample into a new class, modified them as necessary, and added a few more methods such that the main application could initialize and start the voice recognition engine. Later in the polishing stage, we added two more functions that pause the voice recognition engine while the user drives the Arduino and restarts the voice recognition engine when the user is not driving the Arduino. Thus, voice commands only work when the user is not driving the Arduino. This was done in order to improve motion tracking performance.

6.7 Animations and Video Stage

The transitions between modes at this point consisted of hiding some components and revealing others instantaneously, which from a user's perspective was rather startling. To combat this, we added fade animations. Essentially, when a the application is about to switch modes, the current mode's on screen components are disabled and fade out from the GUI, and the next mode's on screen components are enabled and fade into the GUI smoothly. We first did simple animations on one or two components for the transition from the menu to Steering mode, and then scaled that same process to all components in all modes by implementing clean helper functions.

We also added video rather easily by following the Skeletal Tracking Fundamentals tutorial as part of the Kinect for Windows Quickstart Series[6]. Adding video and a skeleton frame turned out to be as easy as adding "viewers" from the included KinectWPFViewers library onto the main window, and binding the viewers to the Kinect sensor; everything else was taken care of!

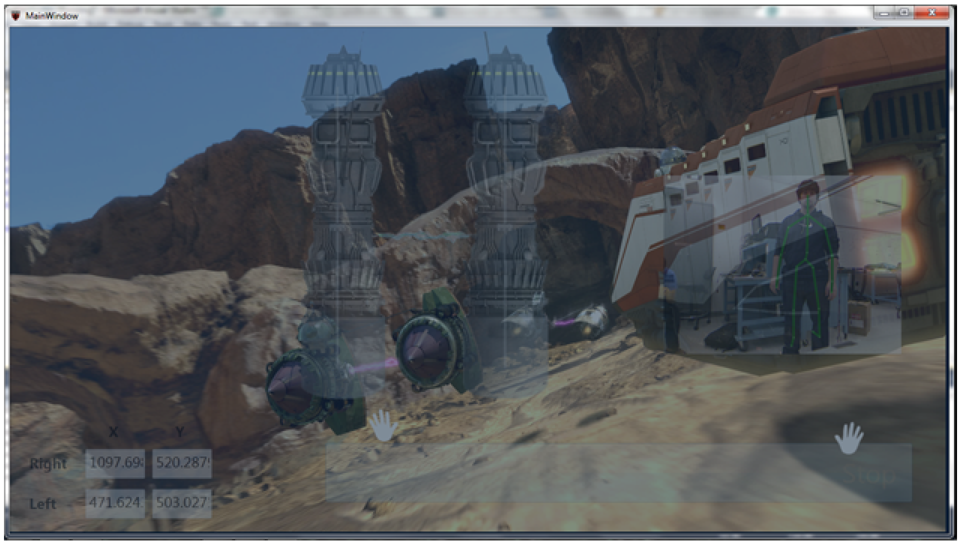


Figure 7: Fade In Animation

6.8 Refactoring Part II Stage

At this point, code was organized by mode, but not *across* modes. To address this issue, we created a new file called "Common.cs" to hold all helper functions and instance variables common to two or more modes. This greatly cut down on the size of the code, and also made clear how similar the modes are.

Similarly, the animation code was organized into clear sections: each mode now has a "Turn on/off" section/region, which consists of a uniform turn off function and and a corresponding turn on function, which fade in and out components for that particular mode.

Within each file, all the functions were organized into different sections, which can be expanded and collapsed with ease for quick navigation. We also took time to add inline clarifying comments to uncommented bits of the source code.

6.9 X-Bee Stage

Quite frankly, this was perhaps the most painful part of the project. We initially were able to successfully use X-Bee communication for our Demo application using the set-up specified by the X-Bee Piazza post (see the Appendix), after trying dozens of times to configure the X-Bees appropriately and spending a few hours to get the configurations to work. Once we realized that the X-Bees worked out of the box, we were able to use X-Bee communication with the hardware described in the X-Bee Piazza post.

However, we weren't able to borrow those hardware parts again, which resulted in lots of hours spent trying to inefficiently test a wired BOE-bot. Thanks to Professor Loncar and Kathleen

France, we were able to obtain 2 X-Bee Explorers Regulated as well as scavenge for two X-Bee S1s. Professor Loncar also provided us with an additional Arduino Nano which would act as the go-between USB connection between the PC and the X-Bee. After trying unsuccessfully to connect the system together, we soldered one of the X-Bee Explorers Regulated to 4 pins to attach to a breadboard, and by luck found an extra X-Bee Arduino shield (to interface between the Arduino and the X-Bee) lying around in the lab. The shield was an especially lucky find because we were having trouble having the X-Bee communicate with the Arduino Uno via the X-Bee Explorer Regulated.

Once we had these materials, we constructed a quick circuit between the Arduino Nano and the soldered X-Bee Explorer in order to forge a serial connection between the PC and the X-Bee. From here, we were able to construct our full circuit as show in Figure 7.

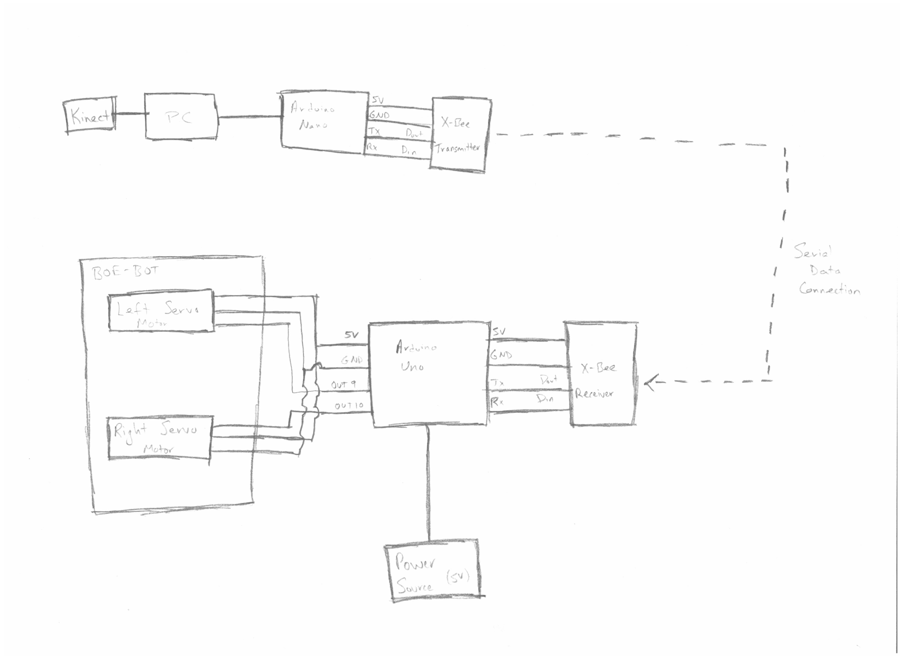


Figure 8: Full System Block Circuit

6.10 Polishing and Documentation Stage

At this point, we felt that we had completed enough features to constitute a full-fledged project. We renamed variables and methods appropriately, reorganized code, added inline comments to the code, eliminated minor bugs, polished animations, resized graphics, modified when the voice recognition engine turns on and off, and much more.

Additionally, we wrote inline XML documentation for every method and field/instance variable, regardless of its access permissions (public, private, etc). We were able to use a program called Sandcastle [8] to convert the XML documentation file into a collection of web pages organized by class and by class members (methods and fields). The resulting html web page files are a good way to get an idea of the data structures and functions used to implement our application without diving into the code itself, and should provide adequate accompanying documentation for those who choose to dive into the source code of our application.

7 Outlook and Possible Improvements

Our project was a success on both the hardware and software ends. It was relatively easy to modify the BOE-bot to work with the Arduino Uno, as it required us to connect the servo motors to two output pins, the power pin, and the ground pin. The Arduino default Servo.h library enabled us to write data read by a serial connection directly to the servos via the output pins. Using this, we were able to implement the Arduino MAAC application without hassle to work with the Arduino Uno.

We spent a majority of our time working on the Kinect MAAC application to make the user interface intuitive and on defining the transformation functions for transferring data between the Kinect and the Arduino correctly. Multiple libraries in Visual Studio allowed us to implement all the functionalities; however we did spend substantial time trying to implement the transformation functions correctly, which would send the appropriate byte packets over the serial data connection.

Finally, we had initial difficulties getting the X-Bee wireless controllers to work. Using the recommended program, X-CTU, did not actually help us to get the X-Bee receivers connected, and we recommend that future projects avoid this program unless the team knows how to fully use it or requires it to further configure the X-Bees. Documentation for the X-Bees is not well organized, and it is difficult to debug problems with the given resources. We finally realized that the default settings of the X-Bee controllers worked, and we were able to use the Arduino shields as plug and play devices to get the wireless serial data transfer working correctly. Some clear documentation on the X-Bees would go a long way in saving time for future projects.

Given additional time and resources, we believe that this project could be improved and extended in multiple ways. Both the hardware and software can be improved to add additional functionality to the program. We believe that this will allow the MAAC to have practical purposes, since it could be used with larger electronic vehicles, such as actual cars, in the real world. While we do not currently have the resources to implement this, we believe that it is of practical value to use.

7.1 Hardware Modifications

Motors - Due to the modularized structure of our project, we believe that we can update and improve the car without other systems needing to be updated. First, we would use regular motors rather than the current servo motors. The servo motors only have 2 non-zero speeds in each direction, corresponding to slow and fast. By using more precise motors, we would be able to fine tune the MAAC's turning capabilities and give the user a better distinction for the various speeds on the car.

Remote Camera - We would add a wireless camera onto the MAAC, which would potentially allow the user to see where the MAAC is going without physically watching the car moving. This allows the user to fully focus on controlling the MAAC with the Kinect MAAC application and watch the MAAC from the next room over. We would potentially need to use bluetooth or another form of wireless connection to transmit the camera's data to the computer for processing.

Sensors - Adding sensors to the MAAC would allow us to create a standardized "car safety" for the driver. We would want the MAAC to be able to detect things around it, to prevent accidental collisions and avoid damaging the hardware on the MAAC. Furthermore, we would also be able to provide the driver with more information about the environment around the MAAC, assuming that the driver can not physically see the MAAC while he is controlling it.

7.2 Software Modifications

Robustness - The Kinect MAAC application is not completely bug-free, as sometimes it will crash when the Kinect is disconnected from the PC, for instance. The application should fail more gracefully and should be able to handle more extreme use scenarios.

Design - The design of the Kinect MAAC application is not optimal. As mentioned in Section 6, the four modes are in one class; in an optimal program, the four modes should also be separate classes that inherit from an abstract base class for modes, rather than just splitting the modes up into separate files.

AI Mode - We believe that we would be able to create an AI mode where the MAAC would travel to a location specified by a voice or motion command on the Kinect Application. With the hardware changes listed above, the AI would be able to determine the best path to the target. A GPS-style sensor might be necessary to determine the relative location of the MAAC while en route to the destination.

Tuning/Optimization - In order to aid the driver, we would like to tune the car and the calculations behind the software to make the driving experience as smooth as possible. Adding more realistic math behind the different driving modes would give a more life-like experience to the driver. Furthermore, we would like to be able to add tuning/optimization modes in the software itself, which would allow the driver to customize offsets for sensors and motors, to allow for the best driving conditions possible.

8 Acknowledgments

We would like to extend our thanks to to Abishai Vase '12 and Professor Marko Loncar for their help in the lab and to Ellen Farber '13 for creating a customized mount for the MAAC.

9 Disclaimer

We allow this project to be shared in its full capacity, including code, photos, and videos. Should the results be replicated and enhanced, please cite this project and contact its authors.

References

[1] http://www.instructables.com/id/Kinect-controls-Arduino-wired-Servos-using-Visual-/

[2] http://channel9.msdn.com/Series/KinectQuickstart

[3] http://www.diaryofaninja.com/blog/2011/10/19/remaking-the-xbox-kinect-hub–an-introduction-to-new-user-interfaces

[4] http://www.microsoft.com/en-us/kinectforwindows/

[5] http://www.digi.com/support/productdetail?pid=3352

[6] http://channel9.msdn.com/Series/KinectQuickstart/Camera-Fundamentals

[7] http://channel9.msdn.com/Series/KinectQuickstart/Skeletal-Tracking-Fundamentals

[8] http://sandcastle.codeplex.com/

10 Appendix

10.1 Piazza Posts

10.1.1 Xbee and Arduino - written by Randy Miller

For people who are looking to use Xbee for point-to-point communication, the Xbee S1s in the back can do that.

To set up wireless PC/computer to Arduino communication, you can use the S1s. You will need 7 things: your PC, an Arduino, 2 Xbee S1s, Xbee Explorer (http://www.skpang.co.uk/catalog/images/wireless/08687-03-L.jpg), a micro USB cable, and an Arduino shield (http://freeduino.ru/arduino/images/XBee_Shield_Arduino_1_bi
The setup will look like PC -> micro USB -> Xbee Explorer -> Xbee S1 ——— wireless connection ——— Xbee S1 -> Xbee Shield -> Arduino (which is connected to a power source).

The Xbees are preconfigured out of the box, so there's no need to mess with them too much (although if you want to configure them yourself, use X-CTU). However, we may want to check that your computer has the appropriate drivers for the Xbees (If you have already gotten Xbees working somehow on your computer, feel free to skip this paragraph). First, download, unzip, and install the driver available here for Windows (http://ftp1.digi.com/support/driver/cdm20600.zip). I'm not sure about Linux computers and Macs, but if the Xbees don't show up correctly in your device manager, then you can install some drivers here (http://www.digi.com/support/productdetail?pid=3352). Once this is done, go ahead and connect one of your Xbees to your computer via the micro USB cable and the Xbee Explorer. On Windows, you can check that it's recognized by going to Device Manager and checking to see that there is some reference to the Xbee under the Ports group.

Before you connect anything up, you should load your code onto your Arduino. When you do this, do NOT have the Xbee shield/Xbee attached to it. If you have written code for your Arduino where the setup is just for the case when your Arduino is directly connected to your PC, you can use this same code. In other words, the Xbees don't have to be referenced in the software at all; they're just plug and play!

Once you have loaded your code onto the Arduino, you can now wire up the setup, so go ahead and connect everything together. Now you're ready to send data from your PC!

Like the code for your Arduino, the software on your PC that communicates via the Xbees to the Arduino should be the same as the case where the Arduino is directly connected to your PC, where you just send data to the serial/COM port. Again the Xbees are invisible to the software! If you don't want to write up any code, download X-CTU for windows, in which you can send data bytes to your Xbee/Arduino. For Linux and Mac, you can just use your terminal, like they do in this tutorial (http://ashleyhughesarduino.wordpress.com/2010/07/29/xbee-and-macs-the-easy-way/)

10.1.2 Arduino and BOE-bots - written by Randy Miller

Wiring up the Arduino to the BOE-BOT isn't too bad. The BOE BOT consists of a few components: the servos, the sensors, and the green board (that includes the Basic Stamp) that processes input and output. All you care about in the BOE BOT are the servos and potentially the sensors (depending on if you need them or not).

First, we want to remove the green board from the BOE BOT. To do this, you'll notice screws at each of the corners of the board. Unscrew them so that you can remove the board. Also, you'll notice 5 sets of wires that connect to the servos and the sensors. Each set consists of a black, red, and white wire. Unplug these wires from the green board. We'll talk more about these sets later. Now you're ready to use the Arduino.

Let's assume you don't want to use the sensors, so we can ignore the 3 sets of wires that are connected to the sensors for now. On a high level, the goal is to send a signal from the Arduino to the servos that will make them move (or not move). Luckily, the Arduino can also power the servos, so you won't need an external power source.

To hook up the servos to the Arduino, let's first consider the 2 sets of wires to the servos. They consist, like we said before, of a red, black, and white wire. The red powers the servos, the black is ground, and the white receives the signal on how to move the servos; we want to connect these sets of wires to the Arduino somehow. So, you should plug these 2 sets of wires into the breadboard, connect the two red wires to the Arduino's power (look at the 5.5V pin), connect the two black wires with the Arduino's ground, and connect the two white wires to two of the output pins on the Arduino (9 and 11 work).

How you do this on the bread board is of course up to you. You may want to add resistors and other wires to your circuit later on, but this is the basic idea for wiring up the servos to the Arduino for power and for output.

For the Arduino code, look into using Servo.h. There's a bunch of documentation online about how to use this, with tons of examples. Once you get some quick Arduino code up, you'll have a BOE BOT powered by Arduino!

If you need to use the sensors, they have the same set of wires as the servos, and the wires in this case mean the same thing. They'll need power as well (consider using an external power source like a 9V battery), and you can wire up their signal wire (the white one) to the input pins on the Arduino (the analog pins, below the power/ground pins). I have no idea what code is needed for that though.

10.2 Source Code

To view the source code, please see the attached zip file "SrcCode.zip". Inside are two folders: one for the Arduino application and one for the Kinect application. The Arduino application folder holds two Arduino programs, one for the Kinect demo application, and one for the current Kinect application.The "Kinect Application" folder holds the Visual Studio solution (aka, project) called

"ArduinoController" for the current Kinect application. The source code for this application can be found in the relative path "ArduinoController/ArduinoController", and each source code file has the extension ".cs". The project can be opened by double clicking on the solution file (extension ".sln") in the relative path "ArduinoController".

To build the source code, you may need to add a whole bunch of references. Be sure to check the path "Kinect Application/ArduinoController/ArduinoController/bin/Debug/" for any of the dlls that you may need.

However, we submitted an executable that should work as long as the Kinect SDK is installed and the executable is executed in the directory that it came in. The executable can be found in the path "Kinect Application/ArduinoController/ArduinoController/bin/Debug/".

10.3 Kinect Application Documentation

To view source code documentation, please see the attached zip file "SrcCodeDocumentation.zip". Inside is a folder called "Doc" that holds all of the high level source code documentation in traditional MSDN/Javadoc web format. To start browsing the documentation, navigate to the Doc folder and open "Index.html". From there, feel free to browse by exploring different classes as well as the "members" of each class, which are essentially the methods and the fields/instance variables of each class.

The documentation attached covers all methods and all fields in the application, including private members.

10.4 Pictures of Hardware

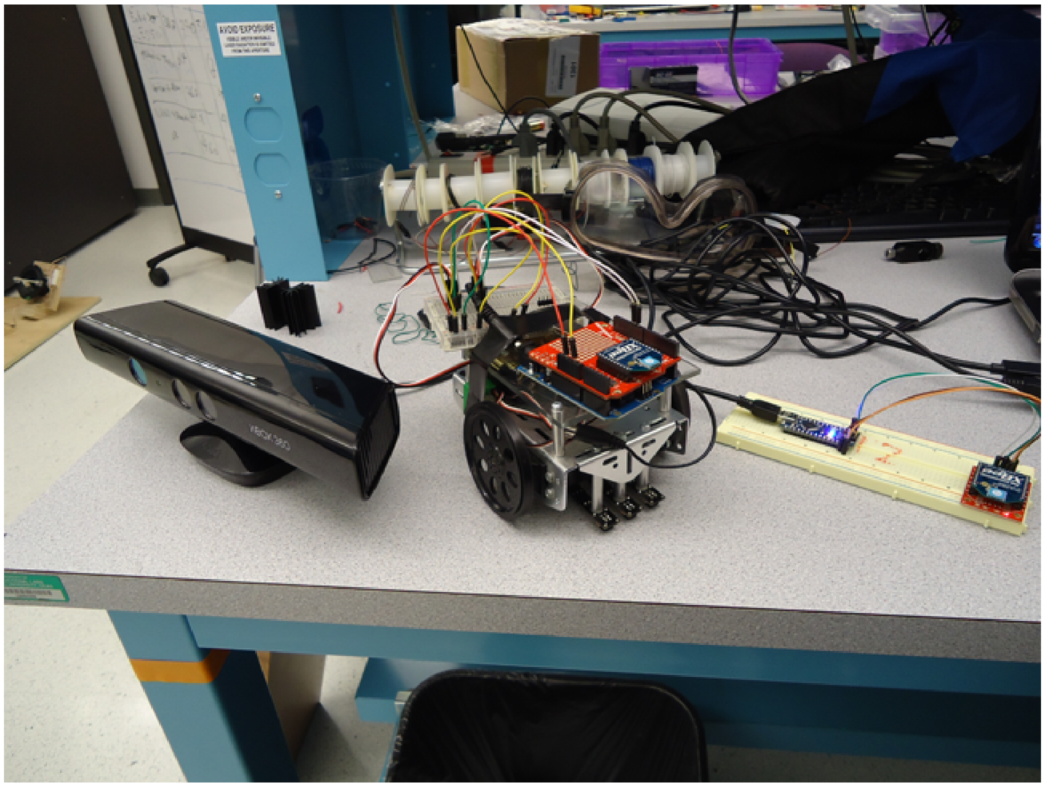


Figure 9: Kinect, Test Arduino MAAC application , and Transmitter System

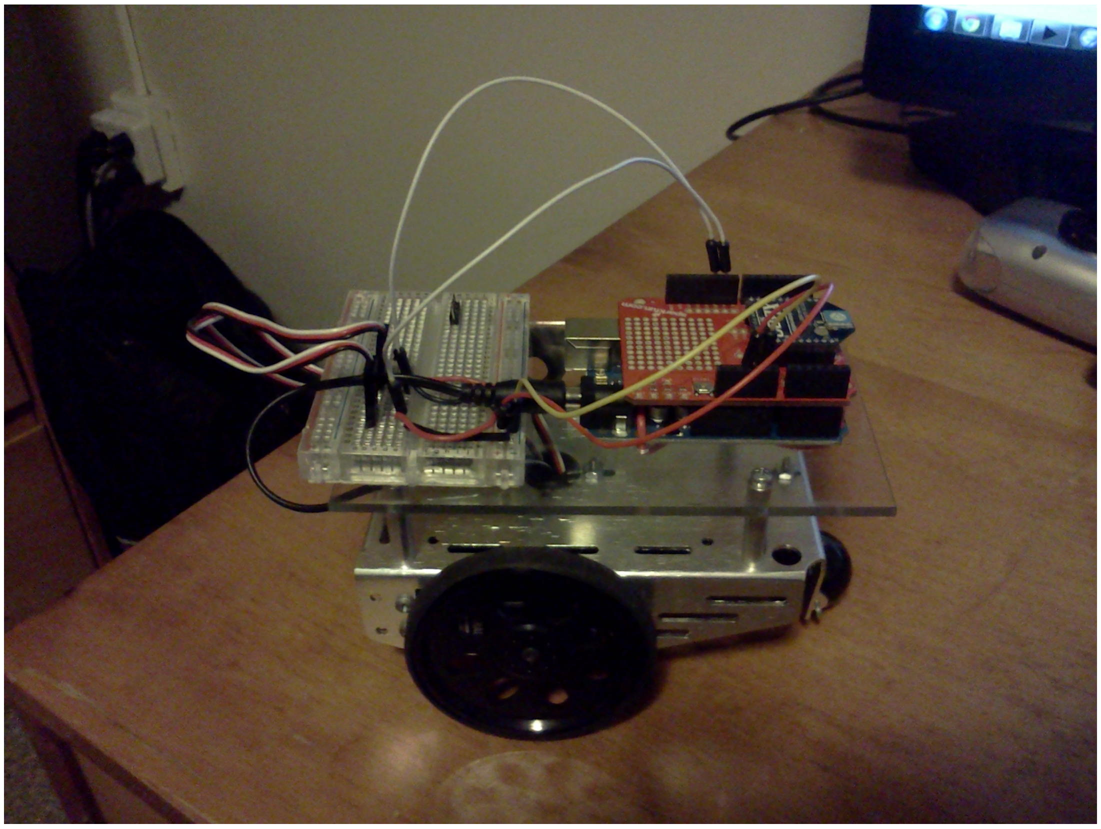


Figure 10: Final Arduino MAAC application