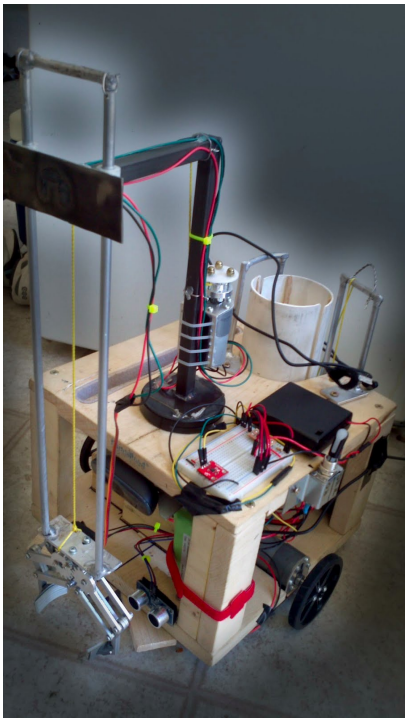


# California Polytechnic State University

**Senior Project, Winter - Spring 2012**
**Roborodentia XVII**
**Stack-E**


Alejandro Ignacio  Austin Hobbs

# Table of Contents

# Introduction

## Project Overview

The main goal for our project is to design and build a functional autonomous robot that is capable of navigating an open arena while avoiding obstacles, as well as identify other objects or cans on the field. It must also be capable of stacking and containing these cans. Deliverables will include the fully assembled robot chassis containing the essential hardware components needed to accomplish the navigation and movement, as well as capabilities like identification of objects and stacking of cans. Alongside the hardware, there will also be software developed to showcase these capabilities of the robot design, including the vision detection of colored objects as well as the intelligence for handling different situations the robot may encounter during operation. Once the robot design is fully developed, assembled, and tested to the best of our abilities, our robot will be entered into the Cal Poly Roborodentia XVII competition and compete with other student developed robots.

## Client

This project is being developed for Dr. John Seng, professor at Cal Poly San Luis Obispo in Computer Engineering for the Roborodentia XVII competition. Dr. Seng is also the faculty advisor and coordinator of the Cal Poly Roborodentia competition, which is held yearly during Cal Poly Open House. This project will benefit the school of Computer Engineering by expanding views and interest in the field of robotics for incoming Computer Engineering students. The final product will be a fully functional autonomous robot that can navigate and adapt to a changing environment, while using a vision system to collect data of objects within its surroundings.

# Design and Justification

## Process overview

The overall design of the robot was functionally decomposed which allowed for modularized divisions of functionality. This helped in specifying what each component in the system was responsible for and what those components needed to accomplish to have a working robot. Each of the decomposed components of the robot, from system to hardware, are described in some detail, specifying their inputs and outputs.

## System Architecture

At the highest level of design, we have the most basic block diagram, seen in Figure 1. A level 0 decomposition contains little detail about the specifics in the system. This however, provides an abstract outline of the overall structure of the system and how each of the two components interact.

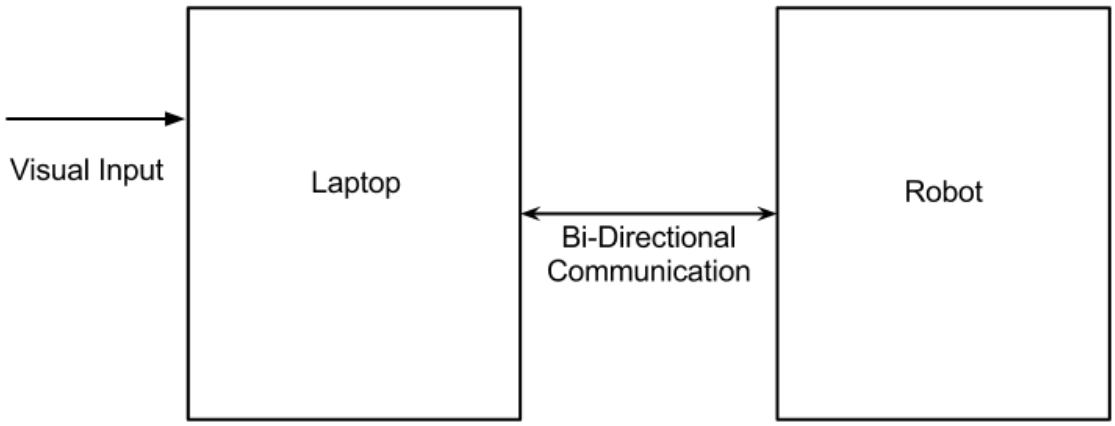



**Figure 1: Level 0 Functional Decomposition**

Figure 2 shows the next level of decomposition, a more detailed block diagram of the system containing the major components of the overall robot design. The system contains a visual component, the webcam, which communicates with the laptop. The laptop in the system interfaces

between the visual input and the microcontroller within the robot chassis. From there, the microcontroller interfaces with the other sensory components, motors, and servos.

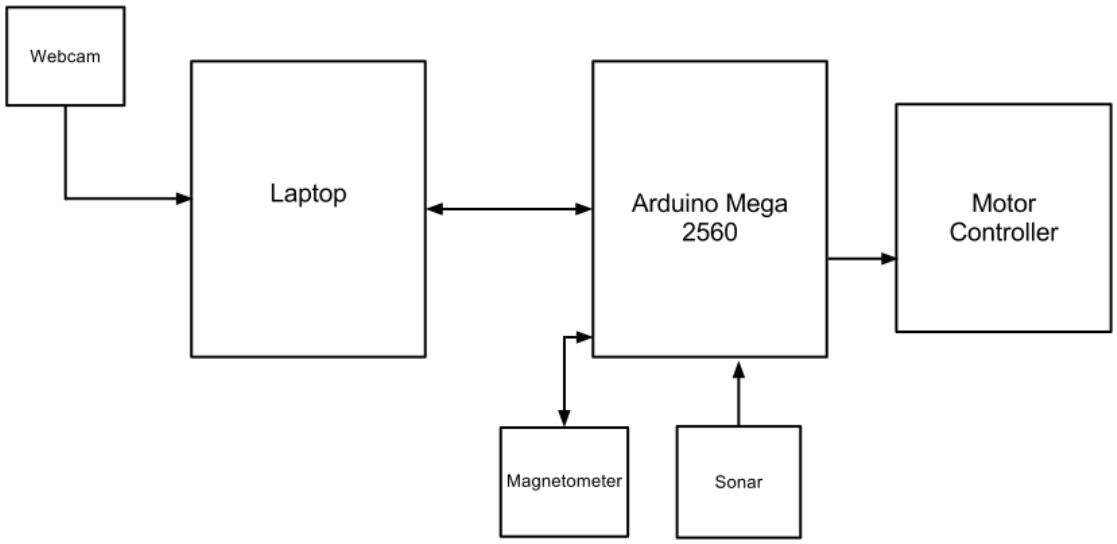



**Figure 2: Level 1 Functional Decomposition**

**Table 1: Laptop Functional Requirements**

| *Module* | Laptop |
|---|---|
| *Inputs* | • USB Data from Webcam<br>• Serial Communication |
| *Outputs* | • Serial Communication<br>• Power to Arduino Mega 2560 |
| *Functionality* | Obtain data on the current video feed from the webcam. Accumulate and process data on the laptop. Once data has been processed, send data packets over serial communication to the Arduino Mega. The laptop will also receive input from the Arduino Mega, this data is used to signal the vision system to ignore certain objects during certain states. The laptop also provides power over USB to the Arduino Mega. |

**Table 2: Arduino Functional Requirements**

| *Module* | Arduino Mega 2560 |
|---|---|
| *Inputs* | • USB Data and Power from Laptop<br>• Magnetometer Data<br>• Sonar Data |
| *Outputs* | • Serial Communication<br>• PWM Signals |
| *Functionality* | Receive vision packets from Laptop. Obtain directional headings from the magnetometer, or hardware compass. Detect distance to objects using data from Sonar module. Based on data received, control motors and servos. Send information back to laptop on current state. |

## Hardware Architecture

Figure 3 is a more in-depth look at the Arduino Mega 2560 as well as the components that are interfaced with the Arduino. These devices are placed within the robot chassis, and will be used to control the different mechanical functions of the robot, e.g., movement of the drive motors, the pulley systems and the different digital and analog sensors.

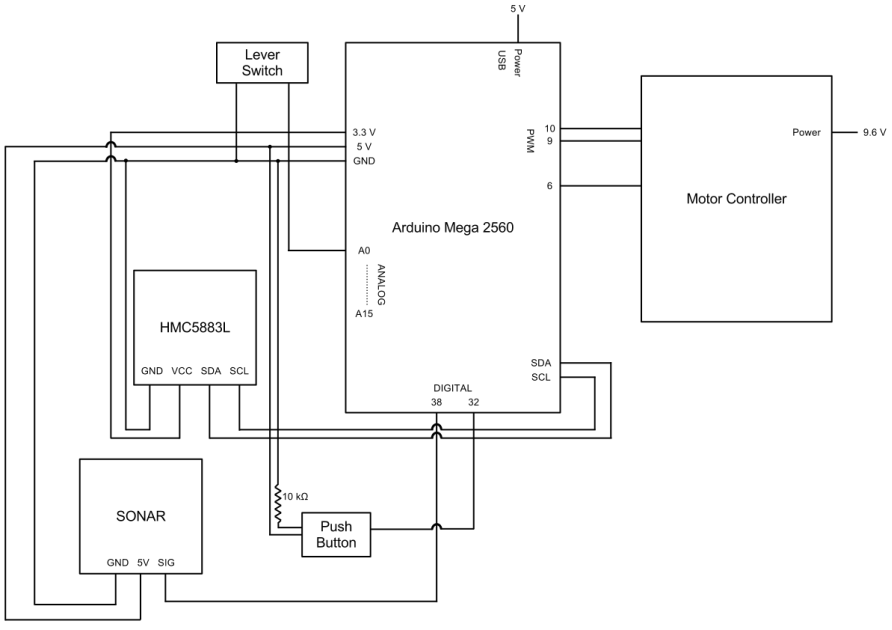



**Figure 3: Hardware Level 2 Functional Diagram**

**Table 3: HMC5883L Functional Requirements**

| *Module* | Magnetometer (HMC5883L) |
|---|---|
| *Inputs* | • 3.3V Power<br>• Data Line (SDA)<br>• Clock Line (SCL) |
| *Outputs* | • Data Line (SDA) |
| *Functionality* | Using its magneto-sensitive sensors, the compass gives a current heading in relation to magnetic North, in degrees. These values are used for determining direction on the field in relation to magnetic North. |

**Table 4: Ultrasonic Distance Sensor**

| *Module* | Parallax Distance Sonar |
|---|---|
| *Inputs* | • 5V Power<br>• Signal Line |
| *Outputs* | • Signal Line |
| *Functionality* | Obtain distance readings using ultrasonic pulses. These readings will be used to determine distances to walls or other objects that may be in front of the robot. Sonar will mostly be used in the heading home state of the robot, in which the distance is needed to determine how close to the end zone wall the robot is. |

**Table 5: Ladyada Motor Controller**

| *Module* | Motor Controller |
|---|---|
| *Inputs* | • 9.6 V Power<br>• PWM signals |
| *Outputs* | • PWM signals<br>• DC Motor Voltages |
| *Functionality* | The motor controller is used to control the three DC motors on the robot, as well as some of the servos used, e.g., the claw and table servo. The Arduino Mega does not contain any H-bridges to control any DC motors and this motor controller can support up to four DC motors, which is just enough for our design. |

## Software Architecture

The software architecture for the robot is divided into two categories, Vision and Embedded. The software architecture is also functionally decomposed to specify what each component is responsible for in the overall system design. As the name suggest, Vision is responsible for object detection and is ran off the laptop which interfaces with the microcontroller. The reason for offloading the vision software to the laptop is to use the processing power of the given CPU on the laptop and the open source vision software libraries for the C++ language. The vision system communicates with the embedded software on the microcontroller, sending specific driving and location information. The Arduino microcontroller contains the Embedded system code, which is written in C. This embedded code controls the hardware components of the robot, e.g., motors, servos, and the different analog sensors. Figure 4 shows a high level look at the flow of interactions between the vision and embedded code, as well as the overall algorithm for detecting and picking up a can.

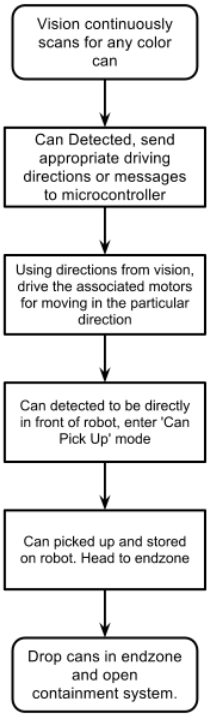



**Figure 4: Overall Algorithm Flowchart**

The vision algorithm developed and used is simple and straightforward; Figure 5 shows a more detailed flowchart of the vision code for the robot. The basic idea behind the vision code is to determine if a can is within the field of view of the webcam, determine whether it is left or right in respect to the center, and give the appropriate drive messages to the microcontroller.

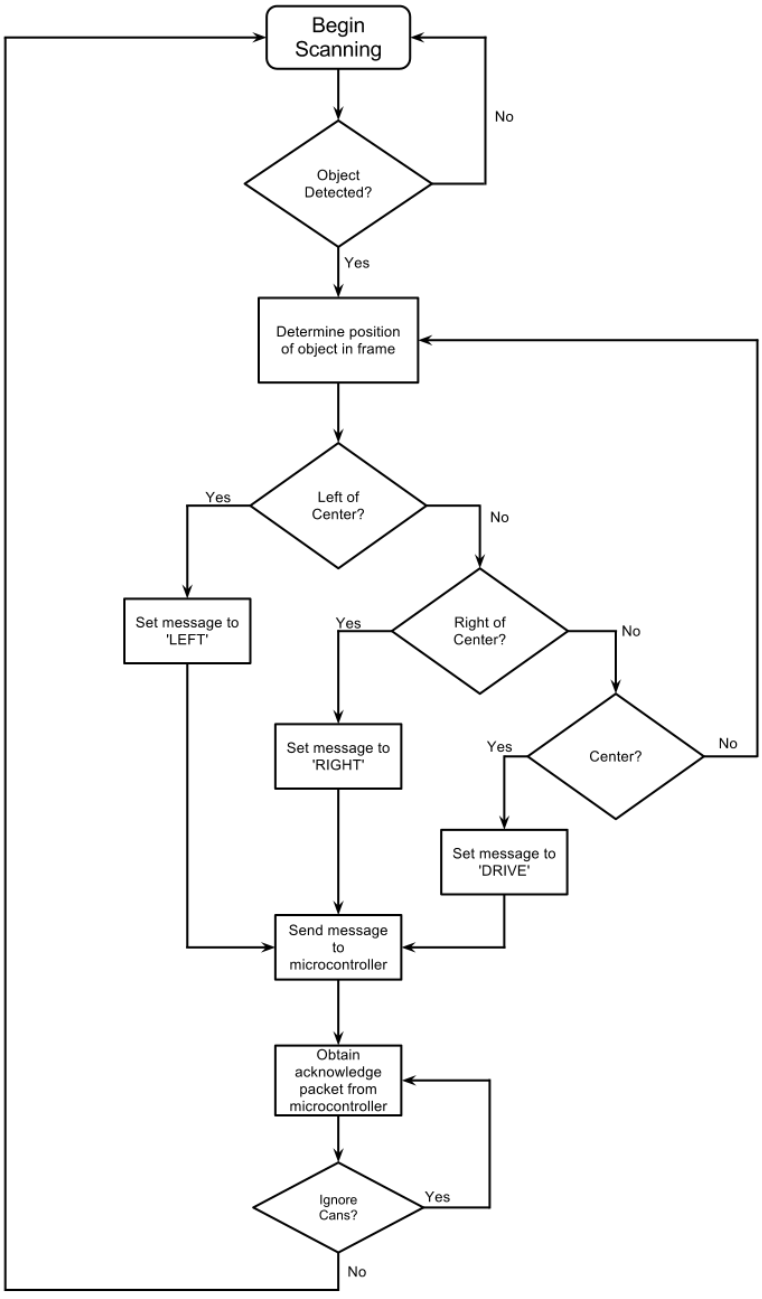



**Figure 5: Vision Flowchart**

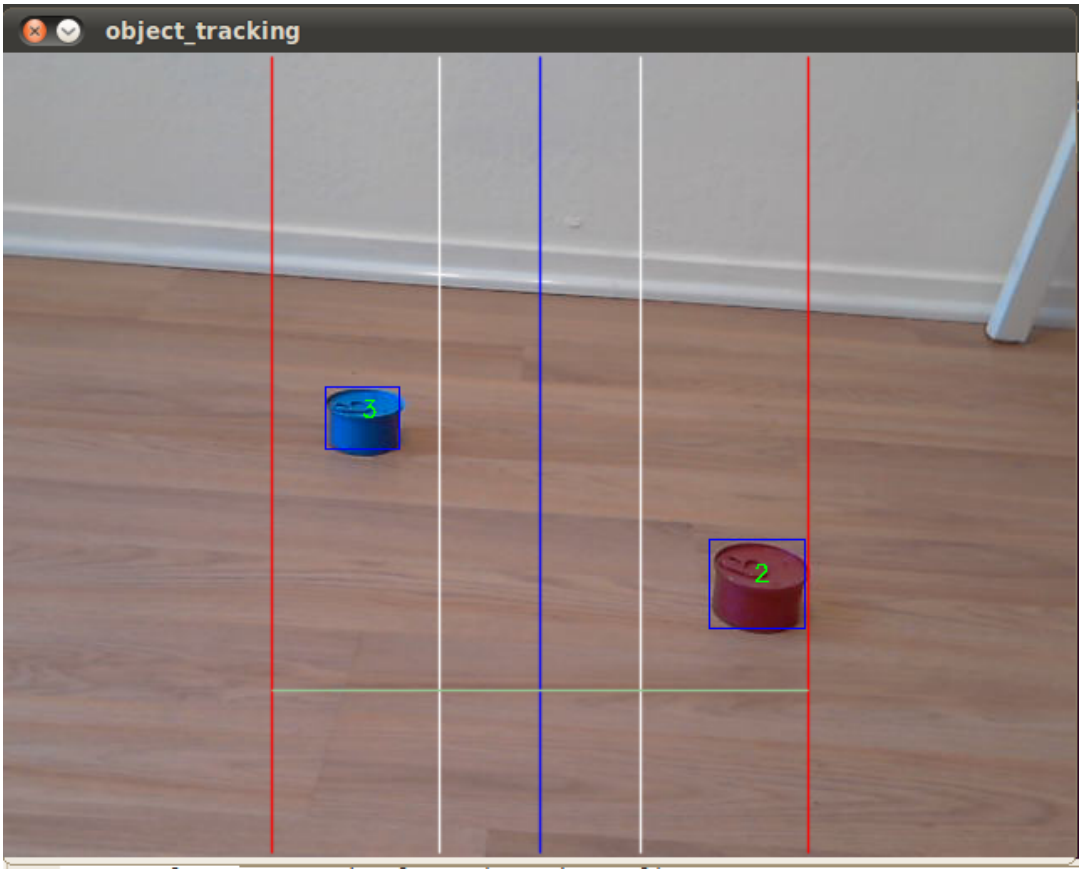



**Figure 6: Vision from webcam**

Figure 6 is the point of view for the robot vision using the webcam. As seen in the image, both red and blue cans are tracked and identified by a label. The blue bounding boxes represent the area of which the colored pixels are grouped together. The center blue line represents the center of the robot chassis, once the centroid of the colored objects, represented by their number label, crosses this blue line the vision code begins a small counter. If this counter reaches 10, then a 'Drive' signal is sent to the microcontroller through the serial communication port. This counter is used to ensure that the robot does not overturn and drive past either side of the can. The white lines represent the center threshold and this area signals to the vision to ignore all other cans and focus on the can that is centered to the robot, also these are used for adjusting the robot to the can. As described before, if the can has crossed the blue line, this would initiate the drive counter, however if the robot overturns, the can would pass both the blue and the white lines. This would signal the robot to readjust and center the can. The red vertical lines represent the left and right

thresholds and is used for identifying whether the object or can on screen is to the left or right of the robot. Within these left and right thresholds, a 'Left' or 'Right' signal is sent to the microcontroller. The green horizontal line represents the threshold for 'can pick up' and when the centroid of the colored object crosses this line, a signal is sent to the microcontroller to enter the 'Pickup Can' state. This is also the time when the vision code ignores all other cans on screen, until a 'scan for cans' signal from the microcontroller is received, this ensures that the vision does not continuously send drive or turn signals while the robot is currently picking up a can.

For the embedded code, it is broken down into sub components each representing a different state that the robot will be in, at a given time. Figure 7 shows the abstract flowchart of the Arduino Mega code, each shadowed state in the flowchart is further decomposed. Note that in Figure 7, obtaining the wall headings is done manually at the beginning of running the robot.

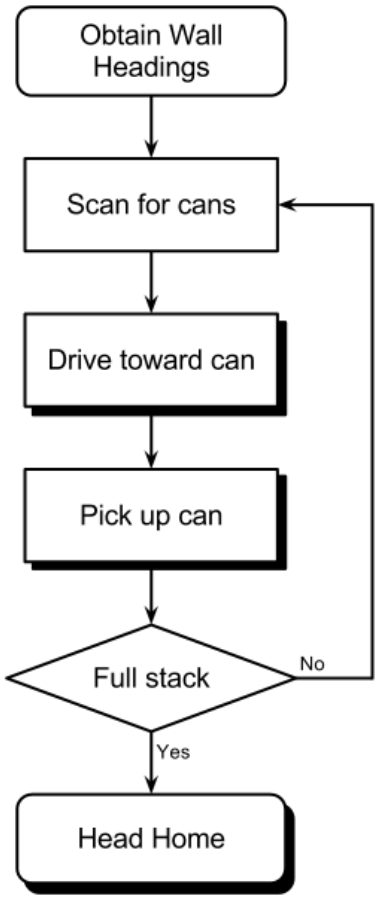



**Figure 7: Arduino Software Flowchart**

The Arduino receives driving messages or packets, in the form of an unsigned char, from the vision code through the serial communication port over USB. These packets are used by the microcontroller to determine which motors to power in order to move the robot in the desired direction toward a can. If a 'No Can Found' packet is received, the Arduino is in the 'Scan for cans' state, in which one of the drive motors is powered allowing the robot to move left or right in a circular sweeping motion. This helps the vision find cans that are no longer in view of the webcam mounted on the robot. Figure 8 shows how the packets received from the vision code are handled.

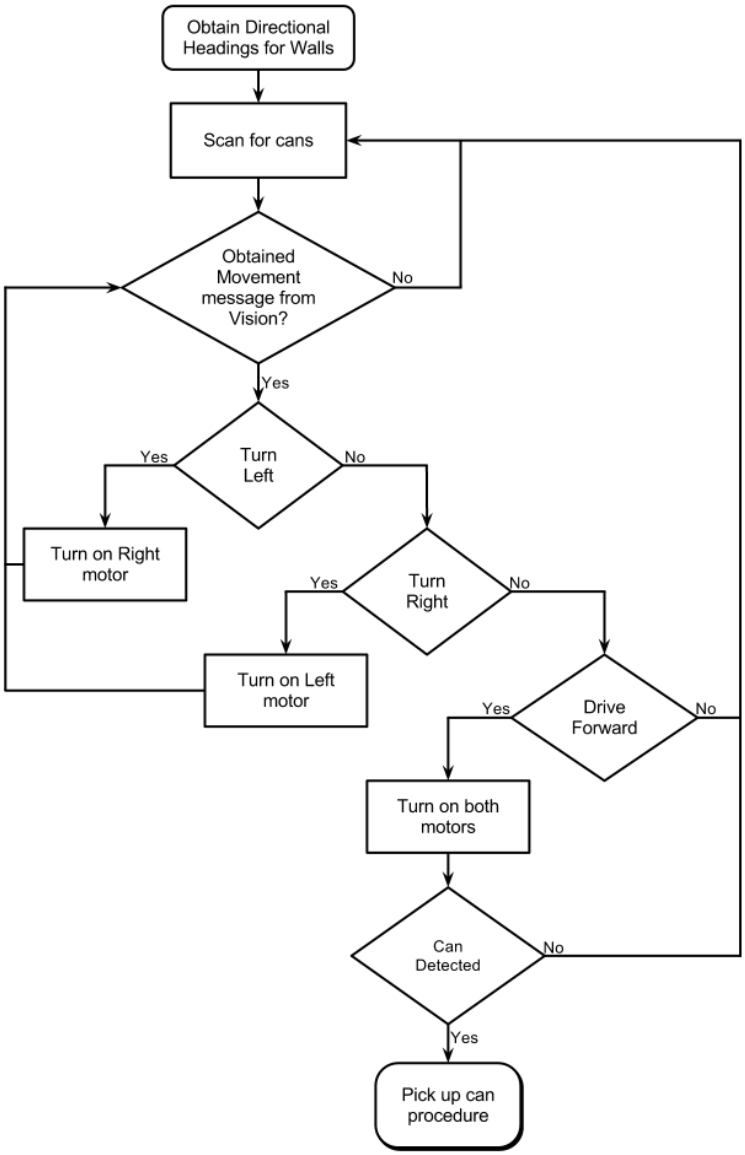
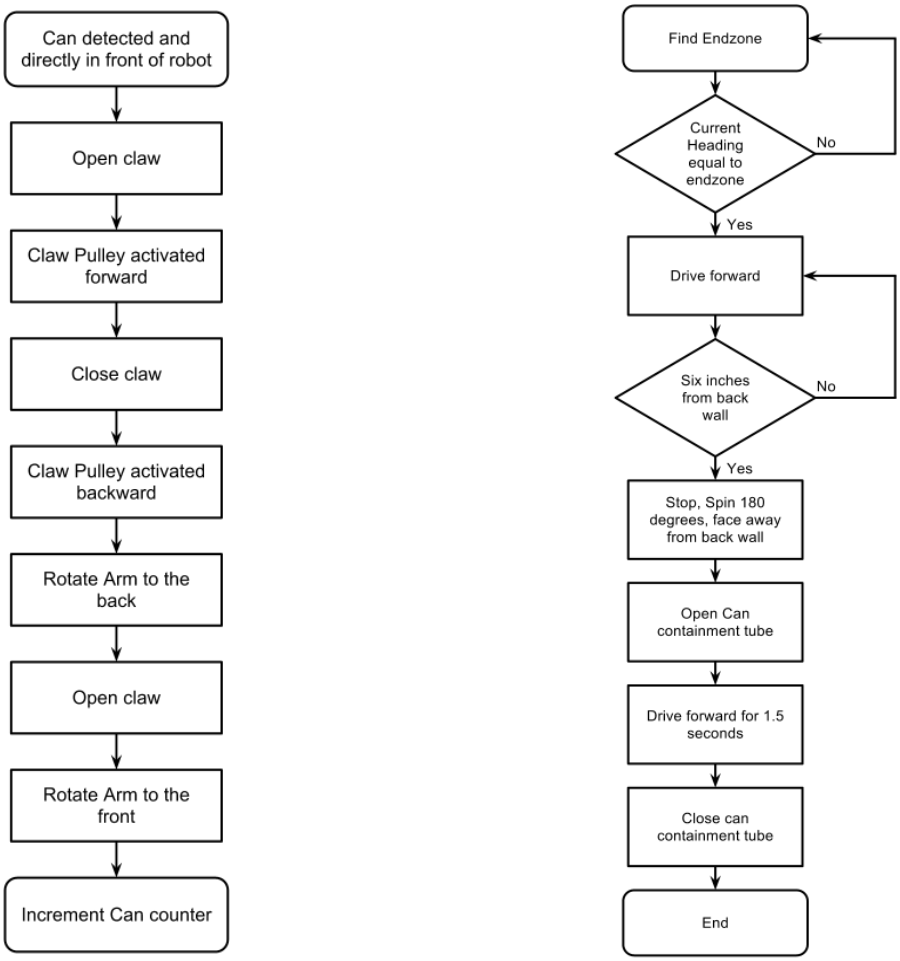



**Figure 8: Movement Flowchart**

Once serial data is obtained over the serial port, movement of the robot is handled by controlling the power of the left or right drive motors through the motor controller. A move left or right command is used to turn the robot in the said direction, to allow for the can or object to be centered with the front of the robot chassis. If a drive command is received, the left and right motors are turned on simultaneously, allowing the robot to close the distance towards the object of interest. The robot will continue to move forward until the timeout is reached or until an interrupt, in the form of an analog LOW, signal is received. This analog signal is triggered by a lever switch located on the front lower center of the chassis. This lever switch is used to indicate to the microcontroller that a can is directly in front of the robot. Once this signal is triggered, the system changes state to picking up the can. A message is sent from the Arduino to the laptop through the serial port, which is used to signal the vision code to ignore other cans while the current can is being picked up. After sending the ignore signal, the function *pickUpCan()* is then called and the embedded code has changed states.




**(a)** **(b)**

**Figure 9: a) Pickup Can and b) Head Home Flowchart**

When the Arduino enters the state to pick up a can, the process is again simple. With the robot arm rotated forward, or towards the front of the chassis, a signal is sent to the claw servo. A full 180º signal is sent to ensure that the robot claw is fully opened before collection of the can. After which, a full forward signal is given to the motor controller to power the DC motor used for the claw pulley system. The claw is now descending towards the can that is directly below the claw. After about 2.5 seconds a stop, or release signal is given to the motor controller to stop the claw pulley from moving any further. Now that the claw is above the can, a signal of 90º is given to slightly close the claw around the can, and not force a stall current on the servo when the claw cannot close fully to 0º. The DC pulley motor is set full backwards to pull up the can and claw, after

which the table servo is given a full 180º signal to rotate the robot arm directly backwards. At this point the claw and can are directly above the PVC tubing used to house the can, and a full 180º signal is given to the claw servo to release the can. After rotating the robot arm forward towards the chassis again, an internal can counter is incremented. This counter will be used to determine how many cans are stored within the robot, if this counter is equal to five, then the Arduino breaks into the next state, which is to head home.

Once the desired number of cans has been obtained, the Arduino initiates the Home state. At this point the robot does not actively scan for anymore cans and instead moves towards its designated end zone. Figure 9.b shows the process of identifying the end zone. In order to determine the direction of the end zone, the robot will use the current heading readings from the compass and the stored end zone heading, which was obtained at the beginning of operating the robot. As the Arduino continuously obtains readings from the magnetometer, this current heading is compared to the stored end zone heading. If the current readings is less than the stored end zone reading, the robot will turn right. If the value is greater than the stored end zone heading, the robot will turn left. Once the current heading is within +/- 10 degrees from the end zone heading, the robot stops turning. At this point the robot is facing the direction of the end zone and the motor controller sends a full forward signal to both drive motors. Simultaneously, the sonar placed in the front of the robot is activated and continuously reads the distance from the wall. As the robot is moving forward, the Arduino checks this distance reading from the sonar and compares this current distance to the threshold end zone distance, roughly six inches. When the distance readings read six inches or less, the motor controller sends a full stop signal to the motors and activates either the left or right motor, to turn the robot a full 180º opposite to the end zone. Next the pulley servo for the can containment tube is activated to pull up the back half of the PVC tube. Once the container is fully opened, the robot then moves forward, releasing the stacked cans into the end zone to be scored, moves forward, and then closes the container.

## Mechanical Design

Pinewood was chosen for the main frame of the chassis. This material was light, easy to work with, and provided a nice platform to be modified if modification was ever necessary. All other mechanical components on the system were made from plastic, aluminum, and steel. The robot consists of four major systems, each of which are responsible for carrying out a particular task. The systems are as follows:

- Drivetrain -- Responsible for moving the robot around the arena
    - Motors
    - Wheels
- Claw rail system -- Responsible for grabbing and picking up cans
    - Claw mechanism
    - Aluminum Rods
    - Nylon Spacers
    - Aluminum Plate
    - Steel Wire
    - Pulley
- Crane arm -- Responsible for rotating the claw rail system into place to drop the can
    - ½" Steel square stock
    - 1/16" Steel plate
    - Plastic rotating table
- Can containment system -- Responsible for keeping the cans contained, and delivery of the cans
    - Rails
    - PVC
    - Wooden dowels
    - Pulley

These systems worked together, each performing their separate tasks, to accomplish the main goal of stacking cans. Figure 10 shows the Mechanical System Overview.

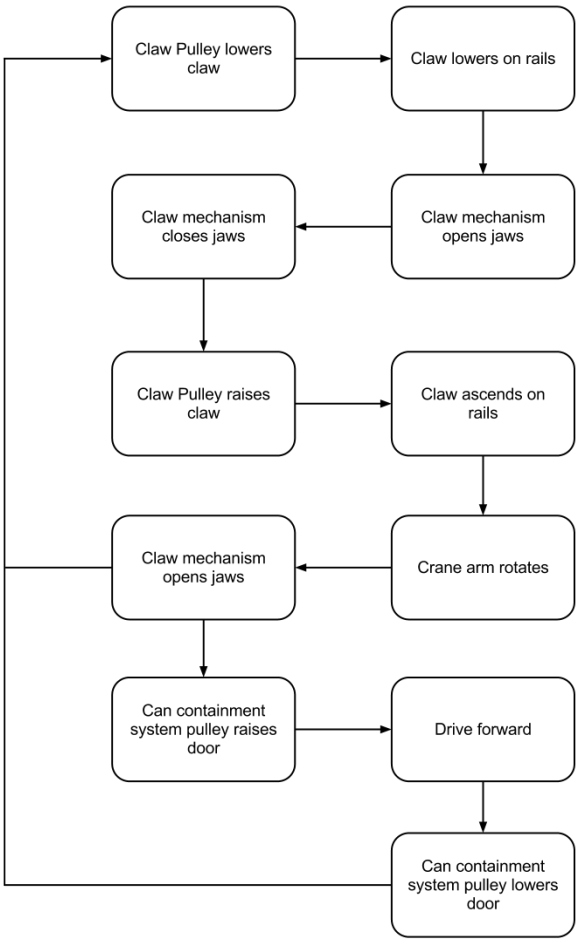



**Figure 10: Mechanical System Overview**

## Mechanical Fabrication

All of the mechanical components on the robot needed some sort of modification or fabrication. The only materials that were used that didn't require modification were fasteners (nuts, bolts, screws, zip ties), and any drive motors/servos/electronic components. The chassis frame of the robot is made out of pinewood. It was decided that the frame of the robot would be a 10" x 10" two tier system, with ~5.5" between the layers. Each of the tiers were cut out to be 10" x 10", and the gap between the tiers consists of two 5.5" tall x 1.5" wide pillars that are glued together to form a right angle. The top tier needed many modifications. The bottom side of the top tier needed to have each of the corners routed out to fit the shape of the pillars. This routing also prevented

lateral movement of the top layer when it is mounted. Other modifications for the top tier consisted of a slot that was wide enough and long enough to accommodate the Macbook Air, and a 3" diameter hole cut out (that is tangential to the backside of the robot) that will house the can containment PVC pipe. The bottom tier, besides the routing for the drivetrain, only need the same 3" diameter hole cut out for the containment PVC pipe. The pillars are screwed into the bottom tier at each corner, and fit into the top tiers routed spaces, and then screwed into the top tier.

### *Drivetrain*

The drivetrain, seen in Figure 11, consists of the base layer of the robot, two DC motors, two motor brackets, four ball casters, and sheet metal that was used for shims under the ball casters to level out the robot. The bottom layer was cut to be a 10" x 10" square first. After the bottom tier was cut out, multiple things needed to be done. Where the motors are going to be housed on the robot need to be routed out allowing for the wheels to touch the ground properly. Lastly, where the ball casters sit on the underside of the bottom layer needed to be routed out slightly to allow the robot to be level. To get the robot perfectly level while sitting on the four ball casters, and the two drive wheels shims were cut out of thin sheet metal. These shims were cut to 1" x 1" squares and stacked between the ball casters and the bottom tier of the robot until each ball caster touched the ground, the robot was level, and the drive wheels, that are pictured in Figure 12, held most of the robots weight.

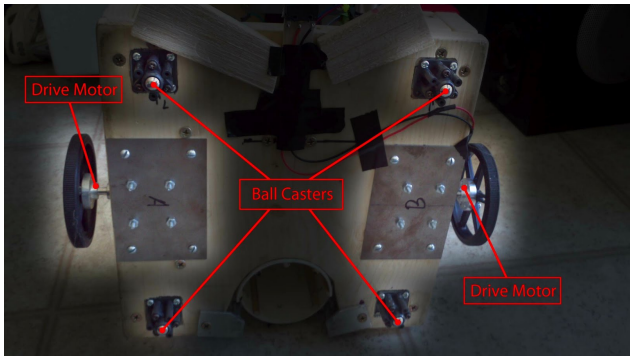



**Figure 11: Bottom of the drivetrain, showing motors and ball casters**

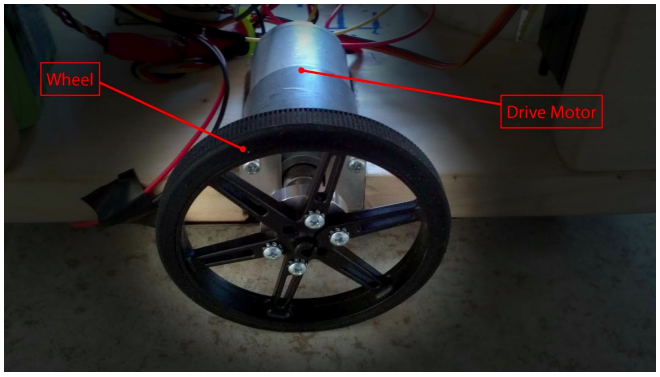
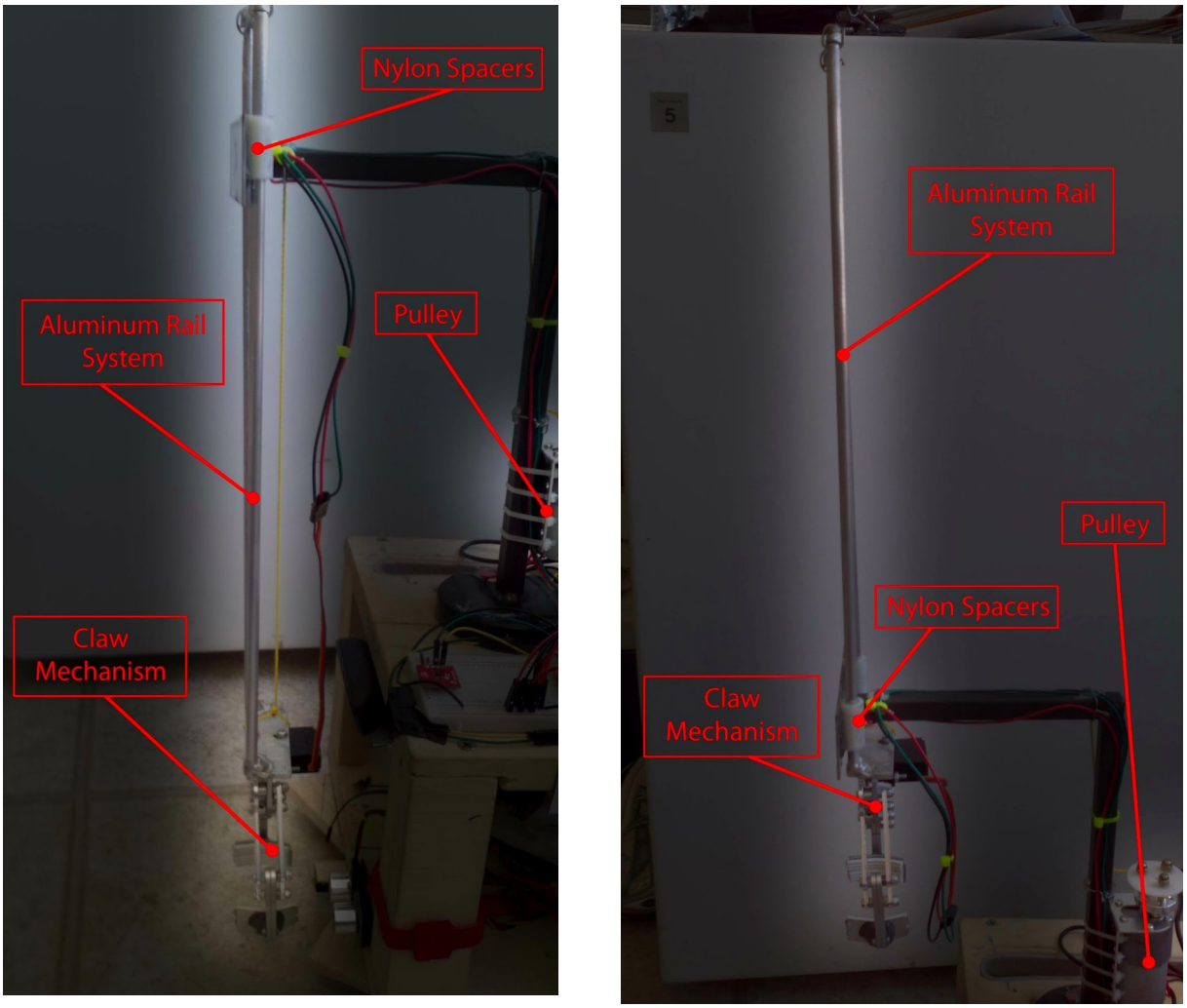



**Figure 12: Side view showing drive motor and drive wheel**

## *Claw Mechanism*

The claw mechanism consists of the claw used to pick and release cans, the rail system for moving the claw up and down as well as stabilization, and finally the pulley that moves the claw up and down on the rails. The claw itself consists of a cast aluminum claw from Sparkfun and CNC'd pieces that allowed the claw to open past the original rated jaw width. The claw is connected to an aluminum plate that is welded to the aluminum rails. The rail system run up and down through nylon spacers that are connected to the crane arm. The claw in the dow and up position is shown in Figure 13.a and 13.b respectively. The pulley that moves the claw is connected to the crane and pulls the claw up from the aluminum plate.



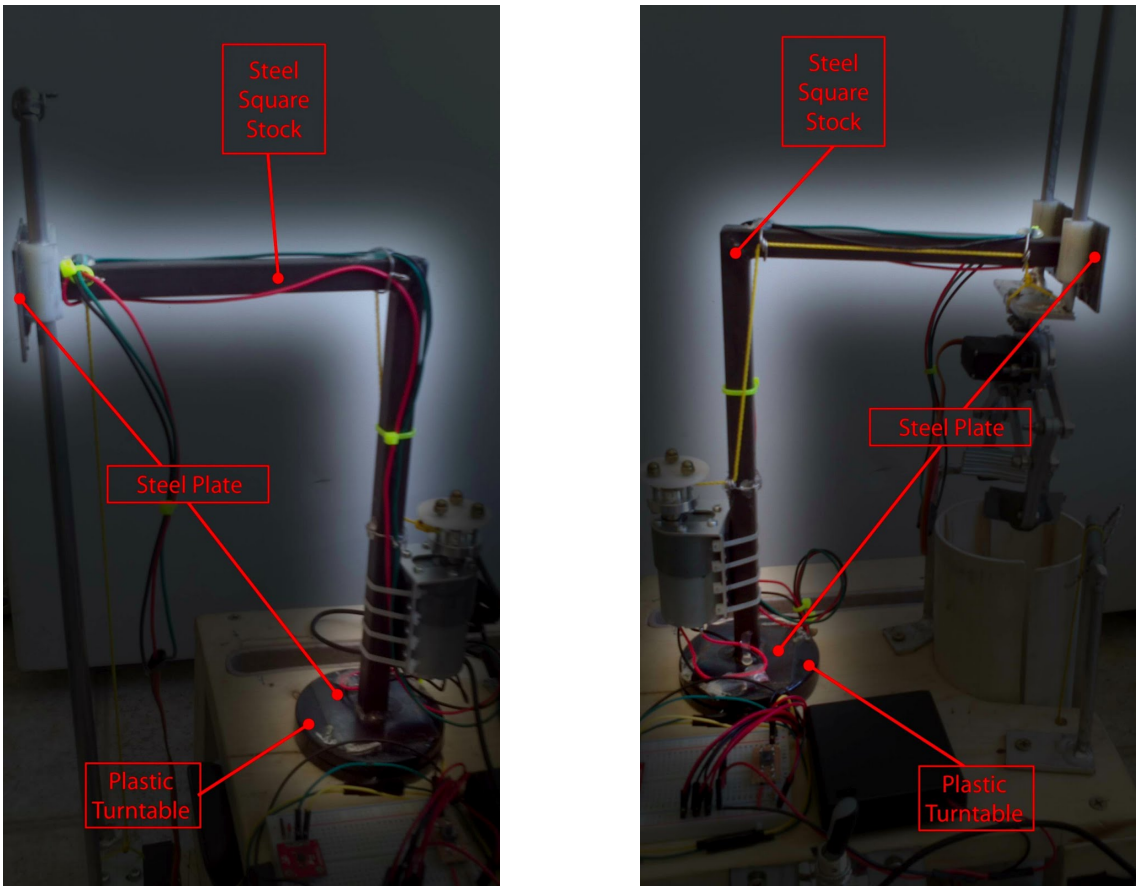

**(a)** **(b)**

**Figure 13: a) Shows the claw mechanism in the down position b) Shows the claw mechanism in the up position**

## *Crane Arm*

The crane arm was fabricated from ½" square stock steel. The horizontal portion of the arm is 6.5" long and the vertical portion of the arm is 9" long. The reason for the 6.5" long arm put the robot dimensions at just under 14" when fully extended. Having the crane arm was a design choice that provided a solution to obtaining cans that were pushed up against walls and left in corners. Other methods that required the can to be close to the robot prevented the robot from obtaining

cans against walls or in corners. The height of the crane arms was a result of how high the can needed to be lifted to clear all other components on the robot, and clear the can containment system. The height of crane arm allowed the can to be above the PCV pipe, as seen in Figure 14.b, before the can was dropped into the can containment system. The crane arm system provided a way to rotate the claw mechanism into the proper position to drop the can. This was done by welding a steel plate to the bottom of the arm, and then using epoxy to bond the plastic turntable to the crane arm. Lastly there was a steel plate welded to the crane arm that allowed for the claw mechanism to be interfaced with the crane. Nylon spacers were put in place on this steel plate using epoxy.

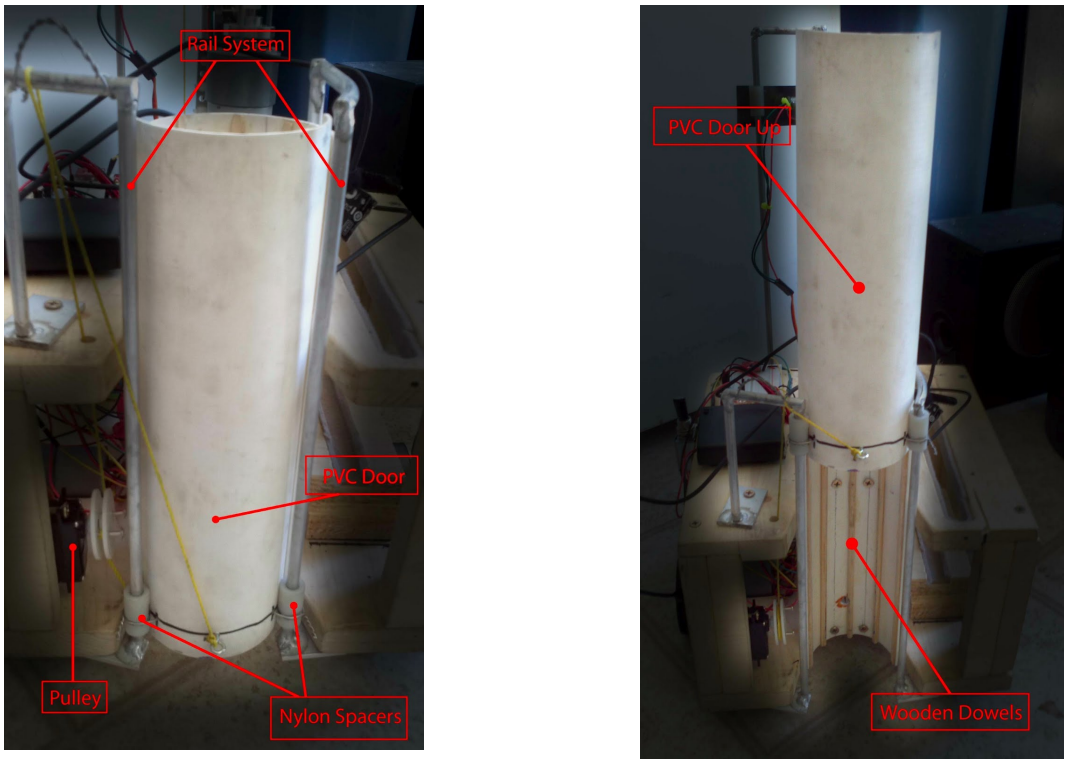



**(a)** **(b)**

**Figure 14: a) Shows the crane arm in the forward position b) Shows the crane arm in the reverse position**

*Can Containment System*

The can containment system provided a place for the cans to sit on the robot, and provided a means of keeping the cans stacked. It consists of a 3" diameter PVC pipe cut in half, wooden dowels glued to the inside to reduce the diameter of the pipe to fit the cans snug, and the aluminum rails that the PVC door can ride on. When dropping cans into the can containment system the wooden dowels prevented the can from rotating or twisting. When there was a stack of five or six can in the system, determined by the pickup counter in software, there is a pulley that will pull up the PVC door. The door has nylon spacers glued on that run along the aluminum rails for stability. The door was capable of opening high enough to allow the stack of cans to stay intact when the robot moved forward after opening the door. The pulley connected to the bottom of the PCV door and was mounted on the bottom tier of the robot.




**(a)** **(b)**

**Figure 15: a) Shows the can containment system in the down position b) Shows the can containment system in the up position**

## Mechanical Justification

Each of the mechanical systems had a very specific task that they were responsible for accomplishing. Every mechanical component had a reason for being on the robot, there was nothing that was on robot that did not have an intended purpose.

### *Drivetrain*

The drivetrain was debatably the most important mechanical system on the robot. Without the drivetrain none of the other functions of the robot would be possible as the robot would be incapable of moving away from its start location and towards the cans. The particular motors that were used had a gear ratio of 50:1 which means the motors were capable of producing more torque, and for the weight of the robot the motors needed to guarantee that they could move the robot. On the bottom tier, the spaces that were routed out to "house" the drive motors and the bracket were absolutely necessary. The motors had to be a specific distance from the ground in order to have the majority of the weight on the drive wheels and only put weight on the ball casters for stabilization. Four ball casters were used at each corner of the robot and allowed for the use of only two drive motors instead of four.

### *Claw Mechanism*

The claw mechanism allowed for the picking up of cans, and raising the can and claw to a level that could clear other components on the robot. The main claw opened its jaws large enough to grab a can, where the rails that the claw moved up and down on stabilized the claw. The main reason for stabilization was less time the robot had to wait to pick up a can. If the robot did not have solid rails that the claw rode on and instead had only a cable and a pulley system, the robot would have to wait for the claw to stop swinging when it wanted to pick up a can. The rails for the claw mechanism also made the system more robust and solid. Using the pulley to move the claw

mechanism up and down on the rails provided an easy solution to the problem, and provided a reliable method that could be done in repetition without fear of failure.

### *Crane Arm*

The crane arm provided a way to move the claw mechanism from the front of the robot to the back of the robot. The crane arm was the only way possible for the can being held in the claw mechanism to reach the can containment system to drop off the can. The crane arm is a very simple system, and serves a very particular purpose, rotation from the front to the back of the robot.

### *Can Containment System*

After picking up the cans when they have been discovered, they need to be stored and stacked somewhere. The can containment system is the system that does this. It allows for the cans to not only be stacked, due to the nature of having the cans dropped off in a tube, but also provides a way to protect the stacked can from other robots. The PVC needed to be cut in half, without this modification there would be no way for the cans to exit the PVC pipe without the robot being lifted off of the ground. The PVC door is attached to a pulley which provides the necessary functionality to get the stacked cans out/off of the robot. The can containment system also needs to have a way to stay controlled while being lifted. This is where the rail system comes into play. The rails that are in place allow the PVC door to be lifted in a controlled and stable manner.

# Conclusion

Although our robot could not perform during the competition due to technical difficulties, we were able to successfully design and build the robot, along with the necessary software libraries to have a fully functional vision and autonomous system for our robot. During the competition, the motor controller for our robot, that controlled the drive motors, as well as the pulley system for the claw had failed. We were unable to figure this out earlier, nor could we get a replacement part soon enough to compete in the competition, thus we had to forfeit the competition. Aside from the setback during competition day, we were able to develop and deliver the individual components and systems that we had originally set out to accomplish.

Developing the chassis and hardware needed for the robot was one of the most tedious portions of the design process. As stated before, the team had gone through many different revisions to develop the final design. Most of the revisions happened later in the fabrication processes when a designed component was not working as intended. Also many of the design revisions did not require the entire design of the robot change, only minor affected pieces.

One improvement to the mechanical design of the robot would be to redesign the gripping mechanism or the claw of the robot. Currently the claw in our design opens just wide enough to grip a can. The amount of space between the claspers of the claw and the sides of the can is fairly small, which leaves very little play for the can. This means that we have very little room for error and that the can must be within a certain area in order to be picked up by the claw. If the can is slightly off centered, the claw may not be able to fully grip the can and thus is unable to pick up the can. A solution to this problem would be to use a claw design that would open large enough to scoop or push cans inward as the claw closes. This increases the area of pickup. Lastly, the idea of having a crane and a claw to pick up cans is a good idea and was chosen to try and get cans that were against a wall as well as in corners. Unfortunately it complicated the design and caused more

issues than it solved. If we could redesign the crane arm and the claw mechanism there would be a much simpler solution, and although it may not be able to get the cans against the walls or in corners, it would be more successful in picking up the cans in the open. The software that was developed for the robot, both the vision and embedded architecture, was simple and straightforward. Early stages of the vision system was simple to develop and test out as some of the team had experience with the open source vision library OpenCV. The vision system was built in incremental stages, object detection for example was developed first. The team was able to get the vision to recognize certain objects that came into view, based on shape. After which, color detection was developed as an added redundancy to the object detection and identification system for the vision. As development continued, the object detection was removed, for the color detection system was sufficient enough. The logic or navigation system for the robot, embedded C code on the Arduino Mega, was also developed in a similar fashion. The embedded code for the robot was divided and developed based on the different states that the robot could be in at any given moment. The drive code was first developed and when a successful movement of the robot was achieved, the next sections of code, picking up the can, and so forth were developed in this manner. The software needed for the robot to operate fully was developed fairly quickly with minor changes throughout the design process and the software for both the vision and robot system performed quite well.

However, the software is not 100 percent complete, nor is it bug free or optimized. Some features could be removed or added to the software libraries to make a more robust software system. For the vision system, one feature that could be implemented is filtering objects out by pixel grouping or pixel area. Currently, any colored object regardless of size, is tagged and identified as an object of interest, but using a type of pixel area filter, we can ignore objects that are far too large to be a can. If other teams during the competition had colored robots, specifically red

or blue, the current vision code would recognize that as an object of interest. With the filter, we can avoid possible problems like such or even filter out smaller objects that the robot could not pick up.

Overall, aside from some minor setbacks, the overall project and design of the robot was successful. Each element in the robot, from the mechanical components to the software, was completed as specified in the original design plan. All the different components performed and functioned together correctly, the Vision system was able to identify colored objects and send the necessary information to the microcontroller through the serial communication port. From there the microcontroller was able to control the different sensory components and the different motors and servos. The mechanical components, like the crane arm, claw mechanism, andthe can containment system all performed to specifications. All the different systems from hardware to software worked well together to create an autonomous robot.

# Appendices

## Budget and Justification

For this project, the budget will be around $300. The most expensive part in this project will be the microcontroller. We will need a microcontroller that can interface with a laptop through serial communication, as well as interface easily with other components, like sensors and motor controller, on the robot. The two choices for a microcontroller are the PolyBot board and the Arduino Mega 2560 R3, both of which are quite capable in terms of processing power. However, the Arduino Mega 2560 has up to four UARTs and can support up to 48 servos, and with the added ability to attach the many different Arduino shields, the Arduino itself becomes a more capable microcontroller. Another reason for choosing the Arduino was for its relatively small size, for the robot has certain design and size restrictions, we are limited to the space on the robot for components like the microcontroller and the other sensors. Along with the microcontroller, there

will also be a need for the servos to control the robotic arm, claw, and pulley systems. The DC motors, up to three motors, will be needed in the design of the robot. Finally for the chassis of the robot, lightweight Pine wood will be used in designing the frame of the robot, while steel and aluminum will be needed for use in the robotic arm and rail systems for the pulley and claw.

## Bill of Materials

| **Item** | **Cost** | **Justification** |
|---|---|---|
| Microcontroller | $58.95 | Microcontroller is necessary to handle all of the lower level IO. All sensor output was fed into the microcontroller to read and interpret. It was responsible for outputting signals to the motor controller and servos. |
| Motor controller | $19.50 | The microcontroller does not have an h-bridge on it, so a motor controller is needed. The pulse width modulation (PWM) signals are sent to the motor controller from the microcontroller and the motor controller is able to use the PWM signal to determine if the DC motor(s) should be forward/reverse biased, and how fast they should be moving. |
| Webcam | $0 (Owned) | This was what the whole algorithm of the robot depended on. Without the webcam we would have no vision and would not be capable of using opencv/cvblobs for object detection on the field. |
| Macbook | $0 (Owned) | The microcontroller is very fast but not quite fast enough to handle all of the images that were being read in from the webcam. Something more powerful is needed for image processing. Also the size and weight of the Macbook Air allowed for the laptop to be on the robot and stay within the size constraints and not adding much weight. |
| Compass | $14.95 | The compass was vital for the robot to find its way back to the endzone for scoring. It enabled us to have a more dynamic algorithm instead of having a preprogrammed route with timing to find its way back to the end zone. |
| Sonar | $29.99 | The sonar was a coarse grain object detector that would let us know that we were heading in the right direction when the cans were detected by opencv. The second sonar was for detecting how far away a wall was. Using this information combined with the compass information we could determine if on our way back to the endzone if we were going to hit the center wall. |
| RC Battery | $31.49 | This provided the necessary power for the DC motors to operate quickly and with enough power. |
| DC Motor | $24.95 | These particular motors were chosen for their power and their gear ratio. We wanted motors that could push the weight of the robot and up to six cans on board. We ended up choosing a motor with a 75:1 gear ratio. |
| Servo | $12.95 | Servos were used to rotate the crane arm, open and close the claw, and for a pulley system. Servos were chosen for their strength and the fact that they do |

| | | |
|---|---|---|
| | | not need a motor controller to use with the microcontroller. |
| Rotating Table | $3.98 | The rotating table provided a means for the crane arm to rotate 180 degrees. These particular ones were used because they were cheap, fulfilled our requirements for rotating, and would allow us to rotate it with a servo. |
| Wheels | $9.25 | These wheels were small enough that they did not overwhelm the robot, but were large enough to provide enough surface area on the ground to move the weight of our robot. |
| Claw | $9.95 | The claw we bought was already made to work with a particular servo, it was small enough to work for our application, and compared to other claws we looked at was quite cheap. We only had to modify a few pieces to allow it to accommodate the size of the can. |
| Pine Wood | $8.99 | Pine would allow us to create a very lightweight chassis, that can be screwed into, routed, cut, glued, etc. A very versatile material for the chassis. |
| Aluminum Rod | $3.99 | The aluminum rods were used for the claw system to run up and down the crane arm, and for the can containment system(PVC) to run up and down on. The aluminum was chosen over steel for its weight. |
| Steel Square Stock | $0 (Donated) | Steel was used for the crane arm making it slightly stronger than aluminum and easier to weld. The square stock was used because it gives us flat surfaces to work with, and if we needed we could run wires within the square stock. |
| Steel Flat Stock | $0 (Donated) | The flat stock was used for the "faceplate" of the crane arm. It provided a plane for the claw rails to run up and down on. |
| PVC | $0.50 | The PVC was chosen for the can containment system because it was already round, it was light, and was easily fabricated into what we needed it to be. |

# References


1. Bradski, Gary and Kaehler, Adrian. Learning OpenCV. 2008

   Reference book for the open source vision software library OpenCV. Contains information about certain functions and techniques used for vision based problems.

2. Ladyada. Adafruit Motor Shield Library. [Software]

   Software library used with the Arduino IDE to use the associated functions necessary to interface with the motor controller.

   https://github.com/adafruit/Adafruit-Motor-Shield-library

3. Liñán, Cristóbal. cvBlobs: Blob library for OpenCV Software (Version 0.10.4) [Software].

   Vision software library used in conjunction with the OpenCV library for C++.

   Library used to identify colored objects and track them on screen.

   http://code.google.com/p/cvblob/

4. Love Electronics. HMC5883L Compass Tutorial. [Software].

   Software library used for the HMC5883L compass.

   https://www.loveelectronics.co.uk/Tutorials/8/hmc5883l-tutorial-and-arduino-library

5. Sinha, Utkarsh. (February, 2012). Tracking colored objects in OpenCV.

   Reference tutorial on tracking colored objects with OpenCV.

   http://www.aishack.in/2010/07/tracking-colored-objects-in-opencv/

## Software Source Code

**Figure 16: Vision Source Code**

```
/**
 * ColorDetect.cpp
 * ----------------------------------------
 * Vision code for Robot Senior Project
 * This code detects colored objects, either RED or BLUE
 * Draws a box aroud detected object and tracks
 * Said object.
 *
 * Author: Cristóbal Carnero Liñán
 * http://code.google.com/p/cvblob/
 * Modified: AJ Ignacio
 * Date Modified: May 27, 2012
 **/

#include <iostream>
#include <iomanip>
#include <opencv/cv.h>
#include <opencv/highgui.h>
#include <cvblob.h>
#include <cvaux.h>
#include <unistd.h>
#include <stdio.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define BOTTOM 100
#define SIDE 100
#define SBOUND 60
#define VWIDTH 640
#define VHEIGHT 480

using namespace cvb;
using namespace std;

int rlThresh1 = 158, rlThresh2 = 96, rlThresh3 = 0,
    rlThresh4 = 255, rlThresh5 = 255, rlThresh6 = 255;

int rhThresh1 = 120, rhThresh2 = 140, rhThresh3 = 70,
    rhThresh4 = 255, rhThresh5 = 255, rhThresh6 = 255;

int bThresh1 = 100, bThresh2 = 100, bThresh3 = 0,
    bThresh4 = 130, bThresh5 = 255, bThresh6 = 255;

unsigned char message = 158;
unsigned char stop = 's';
unsigned char val;

bool doIgnore = true;
bool dontSend = true;
bool blobFound = false;
```

```
bool driving = false;

enum { IGNORE, LEFT, SLEFT, RIGHT, SRIGHT, DRIVE, CAN, START };

int main()
{

    int p1, p2, point, pointY, oCount = 0;
    int innerP1, innerP2, bot, stopCount = 0;
    int centerLine = 0;
    int prevPoint = 0;
    int goCount = 0;
    int ignoreCount = 0;
    int kCount = 0;
    bool quit = false;
    bool notCenter = true;
    bool fromLeft = false;
    bool fromRight = false;
    bool crossedCenter = false;
    int currState = START;
    int prevState = IGNORE;
    int label = 0;

    CvTracks tracks;

    #ifdef __APPLE__
        cout << "On an APPLE computer\n";
        int fd = open ("/dev/tty.usbmodemfa131", O_RDWR | O_NOCTTY |
O_NONBLOCK);
    #endif

    #ifdef __linux__
        cout << "On a LINUX computer\n";
        int fd = open ("/dev/ttyACM0", O_RDWR | O_NOCTTY | O_NONBLOCK);

        if (fd == -1) {
            fd = open ("/dev/ttyACM1", O_RDWR | O_NOCTTY | O_NONBLOCK);
        }
    #endif

    if (fd == -1) {
        std::cout << "Could not open the port\n";
        return -1;
    }

    cvNamedWindow("object_tracking", CV_WINDOW_AUTOSIZE);

    #ifdef DEBUG
        cvNamedWindow("imgRed", CV_WINDOW_AUTOSIZE);
        cvNamedWindow("imgRedHigh", CV_WINDOW_AUTOSIZE);
        cvNamedWindow("imgBlue", CV_WINDOW_AUTOSIZE);
        cvCreateTrackbar("Thresh1", "imgRed", &rlThresh1, 256, 0);
        cvCreateTrackbar("Thresh2", "imgRed", &rlThresh2, 256, 0);
        cvCreateTrackbar("Thresh3", "imgRed", &rlThresh3, 256, 0);
        cvCreateTrackbar("Thresh4", "imgRed", &rlThresh4, 256, 0);
        cvCreateTrackbar("Thresh5", "imgRed", &rlThresh5, 256, 0);
```

```
        cvCreateTrackbar("Thresh6", "imgRed", &rlThresh6, 256, 0);

        cvCreateTrackbar("Thresh1", "imgRedHigh", &rhThresh1, 256, 0);
        cvCreateTrackbar("Thresh2", "imgRedHigh", &rhThresh2, 256, 0);
        cvCreateTrackbar("Thresh3", "imgRedHigh", &rhThresh3, 256, 0);
        cvCreateTrackbar("Thresh4", "imgRedHigh", &rhThresh4, 256, 0);
        cvCreateTrackbar("Thresh5", "imgRedHigh", &rhThresh5, 256, 0);
        cvCreateTrackbar("Thresh6", "imgRedHigh", &rhThresh6, 256, 0);

        cvCreateTrackbar("Thresh1", "imgBlue", &bThresh1, 256, 0);
        cvCreateTrackbar("Thresh2", "imgBlue", &bThresh2, 256, 0);
        cvCreateTrackbar("Thresh3", "imgBlue", &bThresh3, 256, 0);
        cvCreateTrackbar("Thresh4", "imgBlue", &bThresh4, 256, 0);
        cvCreateTrackbar("Thresh5", "imgBlue", &bThresh5, 256, 0);
        cvCreateTrackbar("Thresh6", "imgBlue", &bThresh6, 256, 0);
    #endif

    CvCapture *capture = cvCaptureFromCAM(0);

    cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_WIDTH, VWIDTH);
    cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_HEIGHT, VHEIGHT);

    cvGrabFrame(capture);

    if(!capture) {
        printf("Could not initialize capturing...\n");
        return -1;
    }

    IplImage *img = cvRetrieveFrame(capture);

    CvSize imgSize = cvGetSize(img);

    IplImage *frame = cvCreateImage(imgSize, img->depth, img->nChannels);

    IplConvKernel* morphKernel = cvCreateStructuringElementEx(5, 5, 1, 1,
CV_SHAPE_RECT, NULL);

    ////// Set Boundry points
    p1 = (imgSize.width / 4);
    p2 = (imgSize.width * 0.75);

    innerP1 = p1 + SIDE;
    innerP2 = p2 - SIDE;

    centerLine = innerP2 - SBOUND;

    bot = imgSize.height - BOTTOM;
    //////

    while (!quit&&cvGrabFrame(capture))
    {
        IplImage *img = cvRetrieveFrame(capture);

        cvConvertScale(img, frame, 1, 0);
        IplImage *imgHSV = cvCreateImage(imgSize, 8, 3);
```

```
        cvCvtColor(img, imgHSV, CV_BGR2HSV);

        // Create the Image frames for the different colors and thresholds
        IplImage* imgRed = cvCreateImage(cvGetSize(img), 8, 1);
        IplImage* imgRedHigh = cvCreateImage(cvGetSize(img), 8, 1);
        IplImage* imgRedComb = cvCreateImage(cvGetSize(img), 8, 1);
        IplImage* imgBlue = cvCreateImage(cvGetSize(img), 8, 1);
        IplImage* imgThreshed = cvCreateImage(cvGetSize(img), 8, 1);

        // Establish the ranges for the color Red
        cvInRangeS(imgHSV, cvScalar(rlThresh1, rlThresh2, rlThresh3),
cvScalar(rlThresh4, rlThresh5, rlThresh6), imgRed);
        cvInRangeS(imgHSV, cvScalar(rhThresh1, rhThresh2, rhThresh3),
cvScalar(rhThresh4, rhThresh5, rhThresh6), imgRedHigh);

        cvDilate(imgRed, imgRed);
        cvDilate(imgRedHigh, imgRedHigh);

        // Combine the two low and high ranges for Red color
        cvOr(imgRed, imgRedHigh, imgRedComb);

        #ifdef DEBUG
            cvShowImage("imgRedCombined", imgRedComb);
            cvShowImage("imgRed", imgRed);
            cvShowImage("imgRedHigh", imgRedHigh);
        #endif

        cvInRangeS(imgHSV, cvScalar(bThresh1, bThresh2, bThresh3),
cvScalar(bThresh4, bThresh5, bThresh6), imgBlue);

        #ifdef DEBUG
            cvShowImage("imgBlue", imgBlue);
        #endif

        //Add the color ranges
        cvOr(imgRedComb, imgBlue, imgThreshed);

        //cvShowImage("Thresh", imgThreshed);
        cvMorphologyEx(imgThreshed, imgThreshed, NULL, morphKernel,
CV_MOP_OPEN, 1);

        IplImage *labelImg = cvCreateImage(cvGetSize(frame), IPL_DEPTH_LABEL,
1);

        CvBlobs blobs;
        cvLabel(imgThreshed, labelImg, blobs);
        cvFilterByArea(blobs, 500, 1000000);
        cvRenderBlobs(labelImg, blobs, frame, frame,
CV_BLOB_RENDER_BOUNDING_BOX);
        cvUpdateTracks(blobs, tracks, 200, 5);
        cvRenderTracks(tracks, frame, frame,
CV_TRACK_RENDER_ID|CV_TRACK_RENDER_BOUNDING_BOX);

        read(fd, &val, 1);
        if (val == 'G' && goCount == 0) {
            cout << "Got signal to start" << endl;
```

```
            doIgnore = dontSend = blobFound = driving = fromLeft = fromRight
= false;
            crossedCenter = false;
            notCenter = true;
            prevPoint = 0;
            stopCount = 0;
            oCount = 0;
            ignoreCount = 0;
            currState = START;
            message = 'A';
            write(fd, &message, 1);
            kCount = 0;
            goCount++;
        } else if (val == 'H' && ignoreCount == 0) {
            cout << "Ignoring\n";
            doIgnore = true;
            message = 'A';
            write(fd, &message, 1);
            goCount = 0;
            ignoreCount++;
            currState = prevState = IGNORE;
        }

        label = 0;
        if (doIgnore == false) {

            for (CvBlobs::const_iterator it=blobs.begin(); it!=blobs.end();
++it)
            {
                label = it->second->label;  //Count of number of blobs on
screen

                blobFound = true;
                point = it->second->centroid.x;
                pointY = it->second->centroid.y;

                if (point > p1 && point < p2) {

                    if (point > p1 && point < innerP1 && driving == false) {
                        cout << "Approaching from left\n";
                        currState = LEFT;
                        message = 'l';
                        fromLeft = true;
                        kCount = 0;
                    }

                    if (point > innerP1 && point < centerLine && driving ==
false) {
                        cout << "Still coming from left\n";driving = false;
                        currState = SLEFT;
                        message = 'l';
                        fromLeft = true;
                        kCount = 0;
                    }

                    if (point > innerP2 && point < p2 && driving == false) {
```

```
                        cout << "Approaching from right\n";
                        currState = RIGHT;
                        message = 'r';
                        fromRight = true;
                        kCount = 0;
                    }

                    if (point < innerP2 && point > centerLine && driving ==
false) {
                        cout << "Still coming from right\n";
                        currState = SRIGHT;
                        message = 'r';
                        fromRight = true;
                        kCount = 0;
                    }

                    if (point > centerLine && fromLeft == true) {
                        cout << "Crossed center line, from left, drive\n";
                        message = 'k';
                        kCount++;
                        driving = true;
                        crossedCenter = true;
                        prevPoint = pointY;
                    } else if ( point < centerLine && fromRight == true) {
                        cout << "Crossed center line, from right, drive\n";
                        message = 'k';
                        kCount++;
                        driving = true;
                        crossedCenter = true;
                        prevPoint = pointY;
                    }

                    // Can may over shoot the center threshold
                    // If this happens, re adjust to center the can
                    if (crossedCenter == true && (fromRight == true ||
fromLeft == true)) {

                        if (point < innerP1) {
                            cout << "Passed the center and coming from
left\n";
                            fromRight = false;
                            fromLeft = true;
                            currState = LEFT;
                            message = 'l';
                            driving = false;
                            crossedCenter = false;
                            kCount = 0;
                        } else if (point > innerP2) {
                            cout << "Passed the center and coming from
right\n";
                            fromRight = true;
                            fromLeft = false;
                            currState = RIGHT;
                            message = 'r';
                            driving = false;
                            crossedCenter = false;
```

```
                            kCount = 0;
                        }
                    }

                    // Simple delay
                    // Robot may turn too much or too fast and can may pass
                    // Through center threshold, but may not be there anymore
                    // When Can is centered, kCount will increase
                    // This ensures the can is still in front of the robot.
                    if (kCount > 10) {
                        cout << "Can is sitting centered, go drive now\n";
                        currState = DRIVE;
                        message = 'D';
                    }

                    // The can has passed the lower threshold on screen.
                    // Stop and Ignore other cans
                    // Let Arduino take over, to drive towards and pick up
can
                    if (prevPoint > bot) {
                        cout << "Can in front\n";
                        currState = CAN;
                        message = 'c';
                        doIgnore = true;
                        driving = false;
                        prevPoint = 0;
                        kCount = 0;
                        break;
                    }
                }

                if (point < p1 && notCenter == true) {
                    cout << "Turn left\n";
                    message = 'l';
                    currState = LEFT;
                    kCount = 0;
                } else if (point > p2 && notCenter == true) {
                    cout << "Turn right\n";
                    message = 'r';
                    currState = RIGHT;
                    kCount = 0;
                }
            }

            if (label == 0) {
                //Send scanning signal to microcontroller
                cout << "No blobs\n";
                prevPoint = 0;

                notCenter = true;
                fromLeft = fromRight = driving = crossedCenter =false;
                stopCount = 0;
                oCount = 0;
            }

            blobFound = false;
```

```
        }

        if (currState != prevState) {
            cout << "Sending\n";
            write(fd, &stop, 1);
            write(fd, &message, 1);
            prevState = currState;
        }

        //Draw on screen the threshold boundries
        cvLine(frame, cvPoint(p1, 0), cvPoint(p1, imgSize.height),
cvScalar(0, 0, 255), 1, CV_AA, 0);
        cvLine(frame, cvPoint(p2, 0), cvPoint(p2, imgSize.height),
cvScalar(0, 0, 255), 1, CV_AA, 0);
        cvLine(frame, cvPoint(innerP1, 0), cvPoint(innerP1, imgSize.height),
cvScalar(255,255,255), 1, CV_AA, 0);
        cvLine(frame, cvPoint(innerP2, 0), cvPoint(innerP2, imgSize.height),
cvScalar(255,255,255), 1, CV_AA, 0);
        cvLine(frame, cvPoint(centerLine, 0), cvPoint(centerLine,
imgSize.height), cvScalar(255,0,0), 1, CV_AA, 0);
        cvLine(frame, cvPoint(p1, bot), cvPoint(p2, bot),
cvScalar(150,200,150), 1, CV_AA, 0);

        cvShowImage("object_tracking", frame);

        cvReleaseImage(&labelImg);
        cvReleaseImage(&imgBlue);
        cvReleaseImage(&imgHSV);
        cvReleaseImage(&imgRed);
        cvReleaseImage(&imgRedHigh);
        cvReleaseImage(&imgRedComb);
        cvReleaseImage(&imgThreshed);

        char k = cvWaitKey(10)&0xff;
        switch (k)
        {
            case 27:
            case 'q':
            case 'Q':
                quit = TRUE;
                break;
        }

        cvReleaseBlobs(blobs);
    }

    cvReleaseStructuringElement(&morphKernel);
    cvReleaseImage(&frame);
    close(fd);
    cvDestroyWindow("object_tracking");

    return 0;
}
```

**Figure 17: Arduino Embedded Source Code**

```
#include <Wire.h>
#include <Servo.h>
#include <HMC5883L.h>
#include <AFMotor.h>

HMC5883L compass;

AF_DCMotor motorA(2, MOTOR12_64KHZ); // create motor 64KHz pwm
AF_DCMotor motorB(1, MOTOR12_64KHZ);
AF_DCMotor pullyFront(4, MOTOR12_64KHZ);

Servo claw;
Servo table;
Servo pullyBack;

unsigned char value;
unsigned long time = 0;

boolean start = true;
boolean isFacingHome = false;
boolean hasGoneHome = false;
boolean sabotageMode = false;
boolean hasCan = false;
boolean endTime = false;

int margin = 20;
int check = 0;
int canCount = 0;
int val;
int buttonState;
int pressCount = 0;
int totalCount = 0;

float endzoneHeading = 0.0;
float rightWall = 0.0;
float leftWall = 0.0;

const int wallSonar = 40;
const int switchPin = 32;
const int midMin = 48;
const int midMax = 55;
const int WALLS = 1;
const int SLOW = 50;
const int MED = 75;
const int CAN_NUM = 1;

void setup()
{
    Wire.begin();
    Serial.begin(9600);
    Serial.println("Serial Started.");

    motorB.setSpeed(100);
```

```
    motorA.setSpeed(125);
    pullyFront.setSpeed(175);

    compass = HMC5883L();
    pinMode(switchPin, INPUT);
    buttonState = digitalRead(switchPin);

    check = compass.EnsureConnected();
    //check = 1;
    if (check == 0) {
        Serial.println("Compass Not Connected");
    } else {
        int error = compass.SetScale(1.3);

        if (error != 0) {
            Serial.println(compass.GetErrorText(error));
        }

        error = compass.SetMeasurementMode(Measurement_Continuous);
        if (error != 0) {
            Serial.println(compass.GetErrorText(error));
        }
    }
}

void loop()
{
    long  currDistance = 0;
    float currHeading = 0.0;

    if (start == true) {
        val = digitalRead(switchPin);
        if (val != buttonState) {
           if (val == LOW) {
               if (pressCount == 0) {
                   endzoneHeading = getHeading();
                   Serial.print("got a home heading \t");
                   Serial.println(endzoneHeading);
                   pressCount++;
               } else if (pressCount == 1) {
                   rightWall = getHeading();
                   Serial.print("found right wall \t");
                   Serial.println(rightWall);
                   pressCount++;
               } else if (pressCount == 2) {
                   leftWall = getHeading();
                   Serial.print("found left wall \t");
                   Serial.println(leftWall);
                   pressCount++;
               } else if (pressCount >= 3) {
                   start = false;
                    //Turn table 180 degrees to the front
                   //delay(1000);
                   Serial.println("Turn table front");
                   tableTurn(1);
                   talkToVision('G');
```

```
                time = millis();
            }
        }
    }
    buttonState = val;
}

// Timeout, when time is 2:30, stop all and drive back home
if (time == 135000) {
    endTime = true;
    gotoMid();
    gotoHomeBase();
}

if (start == false && endTime == false) {
  // when characters arrive over the serial port...
    time = millis();
    currHeading = getHeading();

    if (Serial.available()) {
        // wait a bit for the entire message to arrive
        delay(100);
        // read all the available characters
        while (Serial.available() > 0) {
            value = Serial.read();
            //Display the read value to the debug
            Serial.write(value);

            if (value == 's') {       //This is dirty, must fix.
                Serial.println("Stop");
                motorB.run(RELEASE);  //Stop Motors, then delay for
                motorA.run(RELEASE);  //Half a second, then drive forward
                delay(200);
            } else if (value == 'D') {
                Serial.println("Drive");
                delay(1000);
                motorB.run(FORWARD);
                motorA.run(FORWARD);
            } else if (value == 'r') {
                Serial.println("Motor B: Forward");
                motorB.run(FORWARD);
            } else if (value == 'l') {
                Serial.println("Motor A: Forward");
                motorA.run(FORWARD);
            } else if (value == 'c') {
                getCan();
            }
        }
    }

    if (sabotageMode == true && hasCan == true) {
        Serial.println("Entering Sabotage Mode");
        findEdge();
    }

    if (canCount == CAN_NUM && sabotageMode == false) {
```

```
            talkToVision('H');
            hasGoneHome = true;
            //gotoMid();
            gotoHomeBase();
            canCount = 0;
            Serial.println("Count is Zero");

            ///Send signal to vision code to find cans
            talkToVision('G');

        }
        hasCan = false;
    } else if (check != 1) {
        Serial.println("Compass Not Connected");
    }
}

// When vision code sends 'Can' signal
// Call this function to drive forward towards can
// Assuming can is some what centered to the robot.
void getCan()
{
    int sensorValue;

    while(1) {
        sensorValue = analogRead(A0);
        if (sensorValue == 0) {
            Serial.println("CAN FOUND");
            motorB.run(RELEASE);
            motorA.run(RELEASE);

            ///Send signal to vision code to ignore cans
            talkToVision('H');

            //delay(100);
            pickUpCan();
            hasCan = true;
            Serial.print("Count::\t");
            Serial.println(canCount);
            if (sabotageMode == false) {
                if (canCount == CAN_NUM) {
                    if (hasGoneHome == true) {
                        sabotageMode = true;
                    }
                    talkToVision('G');
                    //break;
                }
            } else {
                talkToVision('G');
                //break;
            }

            ///Send signal to vision code to find cans
            talkToVision('G');
            break;
        } else {
```

```
            Serial.println("driving");
            motorA.run(FORWARD);
            motorB.run(FORWARD);
        }
    }
}
void talkToVision(unsigned char val)
{
    while (1) {
        Serial.println("Talking");
        if (Serial.read() == 'A') {
          break;
        }
        Serial.write(val);
    }
}

//This function is called when in sabotage mode
//This will find the areana walls and drive towards them
void findEdge()
{
    float currHeading = 0.0;;
    long inches = 0;
    boolean isFacingWall = false;
    boolean moved = false;
    boolean facingHome = false;

    while(1) {
        inches = getDistance(WALLS);
        if (isFacingWall == true) {
            if (facingHome == true) {
                if (inches >= 6 && inches <= 8) {
                    Serial.println("In End Zone");
                    motorA.run(RELEASE);
                    motorB.run(RELEASE);
                    letCanGo();
                    break;
                }
            } else {
                if (inches >= 3 && inches <= 5) {
                    Serial.println("Found wall");
                    motorA.run(RELEASE);
                    motorB.run(RELEASE);
                    letCanGo();
                    break;
                }
            }
        }

        currHeading = getHeading();
        if (currHeading > (endzoneHeading - margin) && currHeading <
(endzoneHeading + margin)) {
            isFacingWall = true;
            facingHome = true;
        } else if (currHeading > (rightWall - margin) && currHeading <
(rightWall + margin)) {
```

```
            isFacingWall = true;
        } else if (currHeading > (leftWall - margin) && currHeading <
(leftWall + margin)) {
            isFacingWall = true;
        } else {
            isFacingWall = false;
            moved = true;
        }

        if (isFacingWall == true) {
            if (moved == true) {
                Serial.println("Derp Derp");
                motorA.run(RELEASE);
                motorB.run(RELEASE);
                moved = false;
                delay(1000);
             }
             Serial.println("Facing wall");
             motorA.run(FORWARD);
             motorB.run(FORWARD);
        } else {
             Serial.println("looking for wall");
             motorB.run(FORWARD);
             moved = true;
        }
    }
}

void letCanGo()
{
    Serial.println("Open claw");
    //Open claw
    claw.attach(10);
    claw.write(130);
    delay(1000);
    claw.write(130);
    claw.detach();
}

// Get the headings from the compass
float getHeading()
{
    // Retrive the raw values from the compass (not scaled).
    MagnetometerRaw raw = compass.ReadRawAxis();

    // Calculate heading when the magnetometer is level, then correct for
signs of axis.
    float heading = atan2(raw.YAxis, raw.XAxis);

    // Your mrad result / 1000.00 (to turn it into radians).
    float declinationAngle = 231.3 / 1000.0;
    // If you have an EAST declination, use += declinationAngle, if you have
a WEST declination, use -= declinationAngle
    heading += declinationAngle;

    // Correct for when signs are reversed.
```

```
    if(heading < 0)
        heading += 2*PI;

    // Check for wrap due to addition of declination.
    if(heading > 2*PI)
        heading -= 2*PI;

    // Convert radians to degrees for readability.
    float headingDegrees = heading * 180/M_PI;
    //Output(headingDegrees);

    return headingDegrees;
}

//Make sure we're at the middle of the areana, to avoid 1.5' center walls
void gotoMid()
{
    float currHeading;
    long inches;
    boolean moved = false;
    boolean facingWall = false;

    motorA.run(RELEASE);
    motorB.run(RELEASE);
    delay(1000);

    while (1) {

        currHeading = getHeading();
        if (currHeading > (rightWall - margin) && currHeading < (rightWall +
margin)) {
            facingWall = true;
        } else if (currHeading > (leftWall - margin) && currHeading <
(leftWall + margin)) {
            facingWall = true;
        } else {
            facingWall = false;
            moved = true;
        }

        if (facingWall == true) {
            if (moved == true) {
                Serial.println("Derp Derp");
                motorA.run(RELEASE);
                motorB.run(RELEASE);
                moved = false;
                delay(1000);
             }
             Serial.println("Facing wall");
             break;
        } else {
             Serial.println("looking for wall");
             motorB.run(FORWARD);
             moved = true;
        }
    }
```

```
    motorA.run(BACKWARD);
    motorB.run(BACKWARD);
    delay(1000);
    motorA.run(RELEASE);
    motorB.run(RELEASE);
    delay(1000);

    while (1) {
        inches = getDistance(WALLS);

        Serial.println(inches);
        if (inches < midMin) {
            Serial.println("Backing up");
            motorA.run(BACKWARD);
            motorB.run(BACKWARD);
        } else if (inches > midMax) {
            Serial.println("Moving forward");
            motorA.run(FORWARD);
            motorB.run(FORWARD);
        } else {
            Serial.println("Mid point");
            motorA.run(RELEASE);
            motorB.run(RELEASE);
            delay(1000);
            break;
        }
    }
}

//Drive home when we have 5 cans
void gotoHomeBase()
{
    float currHeading;
    long inches;
    boolean moved = false;
    isFacingHome = false;

    Serial.println("Stop motors, get ready to go home");
    motorA.run(RELEASE);
    motorB.run(RELEASE);
    delay(1000);
    while (1) {
        inches = getDistance(WALLS);

        if (isFacingHome == true && (inches >= 6 && inches < 12)) {
            Serial.println("In End Zone");
            motorA.run(RELEASE);
            motorB.run(RELEASE);
            dropCans();
            break;
        }

        currHeading = getHeading();
        if (currHeading > (endzoneHeading - margin) && currHeading <
(endzoneHeading + margin)) {
```

```
            if (moved == true) {
                Serial.println("Derp Derp");
                motorA.run(RELEASE);
                motorB.run(RELEASE);
                moved = false;
                delay(1000);
            }
            Serial.println("Facing the direction to Home");
            isFacingHome = true;
            Serial.println("running home");
            motorA.run(FORWARD);
            motorB.run(FORWARD);
        } else {
            if (moved == false) {
                Serial.println("moved == false");
                motorA.run(RELEASE);
                motorB.run(RELEASE);
                delay(1000);
            }
            Serial.println("Turning");
            if (currHeading > endzoneHeading) { /// Turn left to face endzone
                motorA.run(FORWARD);
            } else if (currHeading < endzoneHeading) {  /// Turn right to
face endzone
                motorB.run(FORWARD);
            }
            moved = true;
            isFacingHome = false;
            //Output(currHeading);
        }
    }
}

//Get the distance readings from the sonar
long getDistance(int pickSonar)
{

    long duration, value;
    int sonar = wallSonar;

    if (pickSonar == 1) {
        sonar = wallSonar;
    }

    // The PING))) is triggered by a HIGH pulse of 2 or more microseconds.
    // Give a short LOW pulse beforehand to ensure a clean HIGH pulse:
    pinMode(sonar, OUTPUT);
    digitalWrite(sonar, LOW);
    delayMicroseconds(2);
    digitalWrite(sonar, HIGH);
    delayMicroseconds(5);
    digitalWrite(sonar, LOW);

    // The same pin is used to read the signal from the PING))): a HIGH
    // pulse whose duration is the time (in microseconds) from the sending
    // of the ping to the reception of its echo off of an object.
```

```
    pinMode(sonar, INPUT);
    duration = pulseIn(sonar, HIGH);

    // convert the time into a distance
    value = microsecondsToInches(duration);
    //delay(100);
    return value;
}

// Control arm pulley motor and claw servo
// Function that is called when a can is in front
// Of the robot and is ready for pick up.
void pickUpCan()
{

    Serial.println("Open claw");
    //Open claw
    claw.attach(10);
    claw.write(130);
    delay(1000);
    claw.write(130);
    claw.detach();

    //Drop Claw
    delay(1000);
    Serial.println("Drop claw");
    pullyFront.run(BACKWARD);
    delay(2500);
    pullyFront.run(RELEASE);

    delay(1000);
    Serial.println("Close Claw");
    //Close claw
    claw.attach(10);
    claw.write(170);
    delay(1000);
    claw.detach();

    //Pull up claw
    delay(1000);
    Serial.println("Pull up claw");
    pullyFront.run(FORWARD);
    delay(4200);
    pullyFront.run(RELEASE);

    if (sabotageMode == false) {
        //Turn table 180 degrees to the back
        delay(1000);
        Serial.println("Turn Table back");
        tableTurn(0);

        //Open claw
        delay(1000);
        claw.attach(10);
        Serial.println("Open Claw");
        claw.write(130);
```

```
        delay(1000);
        claw.detach();

        //Turn table 180 degrees to the front
        delay(1000);
        Serial.println("Turn table front");
        tableTurn(1);

        canCount += 1;
    }
}


// Control the table servo
void tableTurn(int side)
{

    table.attach(9);

    if (side == 0) {
        /// Turn table back
        table.write(56);
        delay(1000);
    } else if (side == 1) {
        ///  Turn table front
        table.write(129);
        delay(1000);
    }

    table.detach();
}

// Function that controls the motors and servos in the pulley for the
// Back PVC container.
void dropCans()
{
    //Turn 180 degrees, face opposite of endzone wall
    //Prepare for dropping cans.
    motorA.run(FORWARD);
    Serial.println("Turn 180");
    delay(2000);
    Serial.println("Stop");
    motorA.run(RELEASE);

    //Pull up PVC container wall
    Serial.println("Pull up PVC");
    pullyBack.attach(6);
    pullyBack.write(180);
    delay(15000);
    pullyBack.detach();

    //Drive forward
    Serial.println("Dropped cans, driving away");
    motorA.run(FORWARD);
    motorB.run(FORWARD);
    delay(2000);
```

```
    Serial.println("Stopped");
    motorA.run(RELEASE);
    motorB.run(RELEASE);

    //Drop PVC wall
    Serial.println("Closing can containment");
    pullyBack.attach(6);
    pullyBack.write(0);
    delay(9000);
    pullyBack.detach();
}

// Output the data down the serial port.
void Output(float compassHeading)
{
    Serial.print("\theading:\t");
    Serial.print(compassHeading);
    Serial.println(" Degrees   \t");
}

long microsecondsToInches(long microseconds)
{
  // According to Parallax's datasheet for the PING))), there are
  // 73.746 microseconds per inch (i.e. sound travels at 1130 feet per
  // second).  This gives the distance travelled by the ping, outbound
  // and return, so we divide by 2 to get the distance of the obstacle.
  // See: http://www.parallax.com/dl/docs/prod/acc/28015-PING-v1.3.pdf
  return microseconds / 74 / 2;
}
```