University of North Carolina Asheville
Journal of Undergraduate Research
Asheville, North Carolina
May 2019

Autonomous Unification of Computer Vision and Mechanical Sorting Systems

Andre Rucker, Jeffery Hay, Kathryn Ysteboe, Kyle Wertheim, Luke Martin, Justin Turner, Andrew James, Tate Folds, Joshua Mashewske, Alex Mancini, and John Sauvigne

Mechatronic Engineering
The University of North Carolina at Asheville
1 University Heights
Asheville, North Carolina 28804 USA

Faculty Advisor: Dr. Rebecca Bruce

Abstract

A competition held by the Institute of Electrical and Electronics Engineers (IEEE) invites STEM students in the Southeast Conference to compete at a collegiate level with academic peers. A group of Mechatronic students annually represent the University of North Carolina at Asheville at this conference. The 2019 competition was NASA themed, and held at the NASA Marshall Space Flight Center (NASA MSFC) in Huntsville, Alabama. Following the theme of the hosting location, the teams must build an autonomous robot that can maneuver a large square board, avoiding obstacles (i.e. satellites) while collecting spherical and cube shaped blocks (i.e. space debris), sorting the blocks based on color, and then depositing the blocks at their respective locations, all under three minutes. These varying tasks require the knowledge of computer, electrical, mechanical, and control systems and a means unify these systems to achieve the maximum amount of points. The combination of computer vision and a mechanical shelf sorting machine is the most reasonable configuration to complete the requirements of this competition.

1. Introduction

1.1 Robot Requirements

In order to qualify for and compete in the hardware competition, the representing robot must fall inside the dimensions of a nine inch long, nine inch wide, and eleven inch high rectangular prism. This is the required starting size of the robot, but once the round has begun, the robot may have a feature extend three inches in width and length as long as the robot remains motionless. This rule applies to any position on the playing board and at any time in the round. These extensions must be physically attached to the robot, meaning this must be a singular modular robot, no detachable or remote-controlled extensions. Pyrotechnics, compressed gas, hydrocarbons, toxic or corrosive materials are deemed unsafe and dangerous and is means for disqualification. The robot must have at least a one inch high bumper located at 1.5 inches above the ground. This bumper is required to surround at least 80% of the robot's perimeter in a continuous manner. The remainder of the perimeter is permitted to be used as an opening. The bumper is included in the original dimensions. Modifications are allowed in between rounds for programming and mechanical features, but any physical changes will need to be re-inspected for safety and dimensional guidelines.

1.2 Game Board

The theme of this challenge is space related, with each robot needing to clear as much "space junk" as possible in the time allotted while sorting the debris into the corners that share a common color. The debris comes in in the form of spheres and cubes with four possible colors: red, yellow, green, and blue. The spheres are made of a hollow plastic whereas the cubes are made of wood and painted the appropriate colors with acrylic paint. At the center of the game board is a rectangular tower, representing a space station, painted with each of the four colors and oriented so that each side points to the corner that matches their respective colors. Surrounding this tower is a white circle with "satellites", each made with flashing LED lights placed on top of a block half the size of one of the cubes painted black, spaced at 90 degrees around the tower. If contact is made with the any of the "satellites" or the "space station", the team responsible will receive a penalty to their score for the round.

ATTACHMENT A

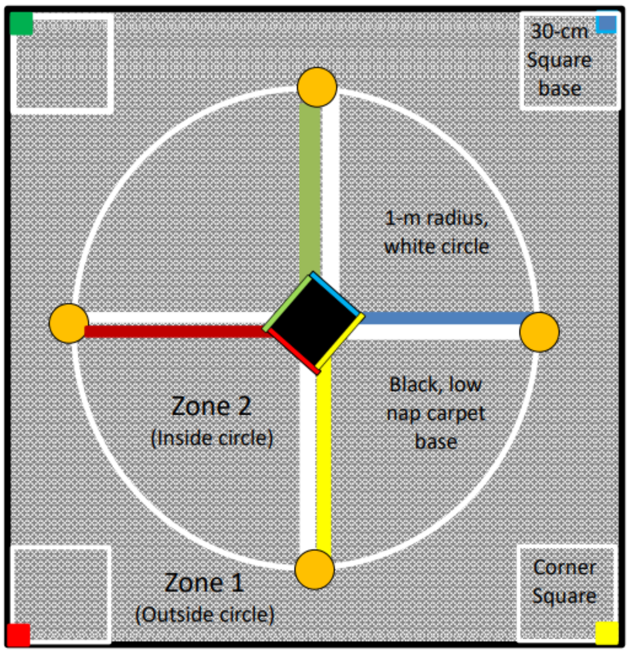


Figure 1. Playing board layout from IEEE hardware competition rules

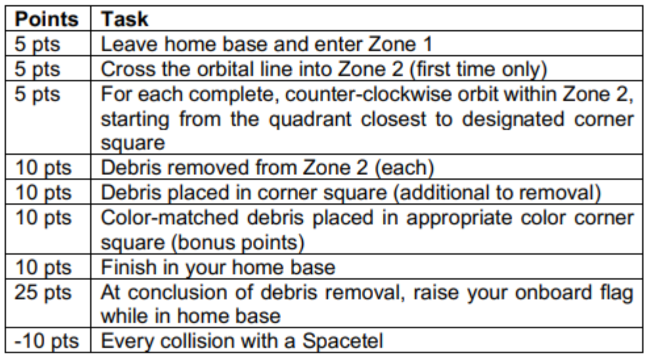
1.3 Competition Rules

Every team is required to build a completely autonomous robot to be eligible for the competition. Once placed into the starting position and turned on by the user, a round is begun when a button is pressed on the robot. Once the button is pressed, the three-minute timer is started and the robot is allowed to move from the starting position and begin completing tasks on the board. In addition, the robot must not act in any destructive manner; if the playing board is damaged by a team's robot, that team will be penalized.

There are two rounds of competition, a qualifier round and finals. The qualifier round consists of the robot running the course by itself and having the entire game board for scoring. There are two rounds of qualifiers and scores from each round are summed together to determine which teams goes to finals. The top eight team with the highest score continue to the finals where a bracketed system is used to determine the winner. In the final rounds two teams compete head-to-head on the same game board. Each team starts in opposing corners. The only change to the scoring in the final rounds is teams only score point on their half of the game board. Their half of the game board is determined by their starting square and the corner counter-clockwise to them.

1.4 Points and Scoring

The goal of this competition is to score as many points as possible within the three-minute time limit. There are multiple ways to score points, as shown in the following image:



Points	Task
5 pts	Leave home base and enter Zone 1
5 pts	Cross the orbital line into Zone 2 (first time only)
5 pts	For each complete, counter-clockwise orbit within Zone 2, starting from the quadrant closest to designated corner square
10 pts	Debris removed from Zone 2 (each)
10 pts	Debris placed in corner square (additional to removal)
10 pts	Color-matched debris placed in appropriate color corner square (bonus points)
10 pts	Finish in your home base
25 pts	At conclusion of debris removal, raise your onboard flag while in home base
-10 pts	Every collision with a Spacetel

Figure 2. Score Table from IEEE hardware competition rules

2. Methodology

2.1 Mechanical System

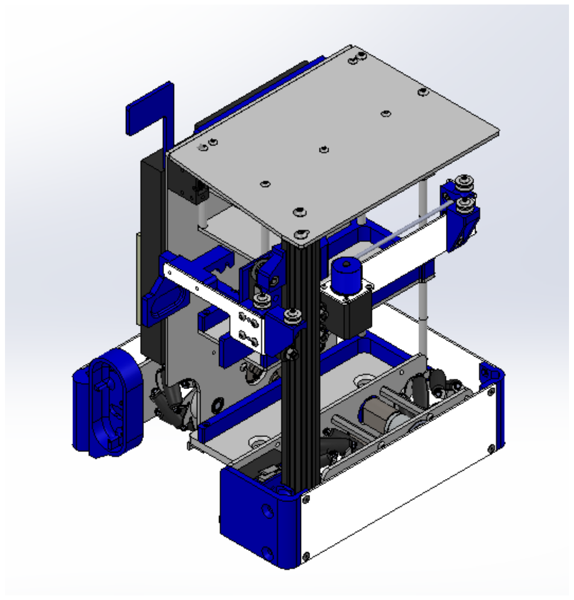


Figure 3. Full Assembly

The mechanical systems were divided into two main components: the wheel assemblies and the sorting mechanism. In order to be a competitive design, the robot needed to be able to pick up, store, and sort objects by color under a time restriction. After going through several ideas on how to carry a large number of objects in order to make efficient use of the time allotted, it was determined that having three vertical rows would be the best option. The robot has a maximum size that will be scrutinized before the robot is allowed to participate in the competition. If the robot did not take advantage of the space it can occupy, it would be at a disadvantage in having to make more trips to the corners of the board to deposit objects. A more efficient design would have a better opportunity to collect a larger number of objects and, as a result, be at a significant advantage. Initially, the design utilized three rows, the largest amount of space that could be allotted to collect the objects on the board. These three rows would enable the robot to take a maximum of nine objects off of the game board in the best case scenario, preventing the opposing team from being able to collect more of the objects than us.

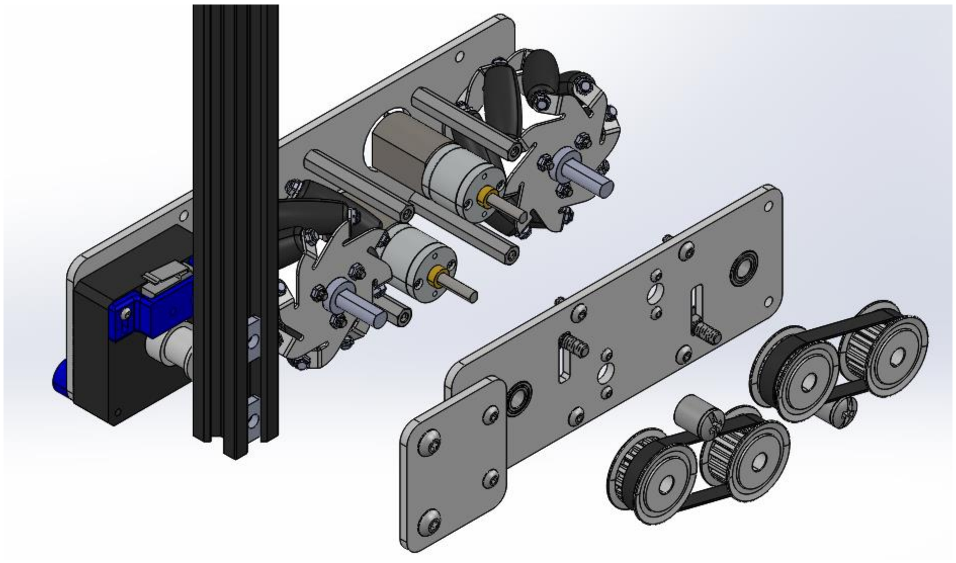


Figure 4. Wheel Assembly

The wheel assembly used in this design was borrowed from an older project the hardware team had worked on. One of the major changes made from the previous iteration of the wheel assembly was to drive the wheels with belts. Having the motors connected directly to the wheels took up a significant amount of space that could have otherwise been utilized, so it was determined that having the wheels be belt-driven would allow the robot to have the lowest row closer to the ground and allow for more room to place electrical components of the regions above the wheels.

Mecanum wheels were chosen for this design because they offer the most versatility for volume consumption and usability on the carpeted game board. Omni wheels were considered but they underutilized the amount of space that they required and their ability to gain traction on the game board. To determine the motors necessary, calculations were done based on the coefficient of static friction of carpet being 0.7, the radius of our wheels being 3cm, and an assumption of a maximum weight of 30 pounds (133.4N):

$$F_f = 133.4N * .7 = 93.4N \quad F_m = 4 * (Torque/Length) \text{ and } F_m >= F_f$$
$$\therefore Torque >= (93.4N/4) * .03m \Rightarrow Torque >= .7N * m$$

Once the minimum required torque was calculated, the smallest motors that had enough torque were chosen.

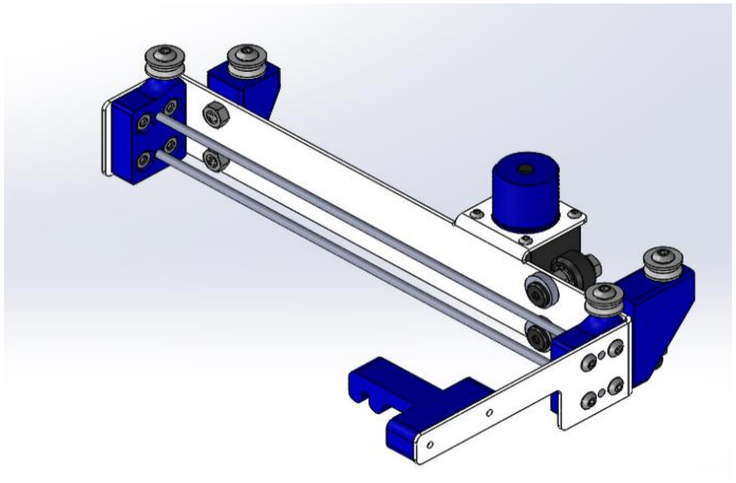


Figure 5. Arm Assembly

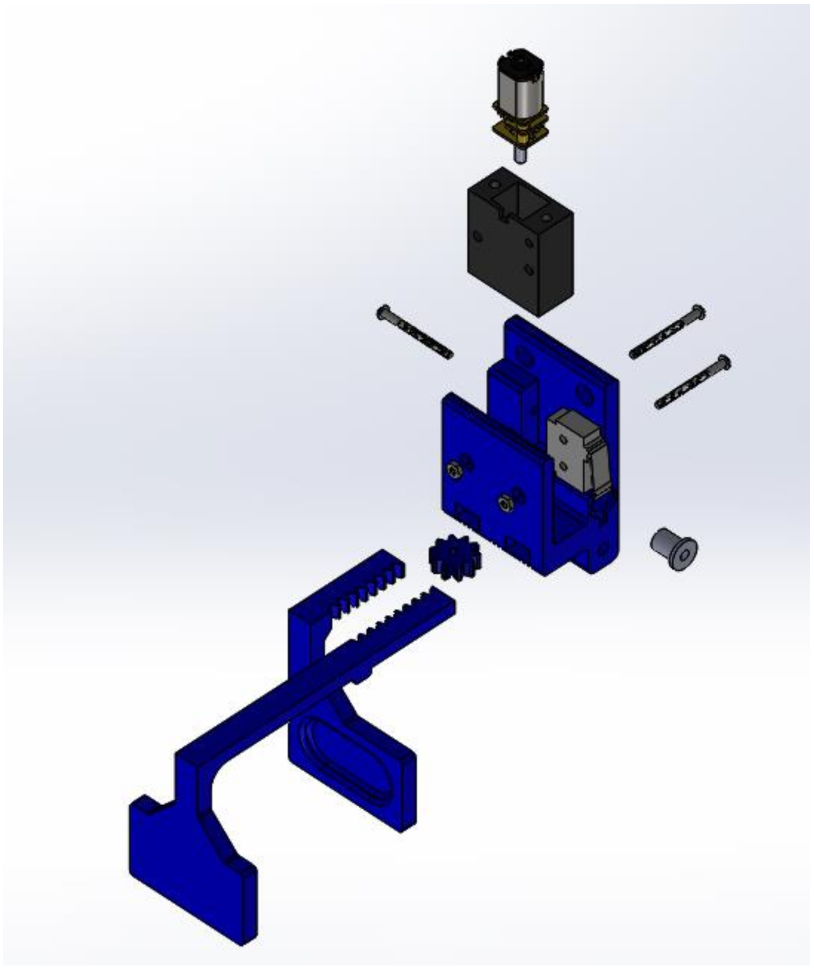


Figure 6. Gripper Assembly

The use of a gripper mechanism (as seen in figure 6) was deemed to be the best method of picking up either spheres or cubes while also allowing the robot to move collected objects into or out of a row. This was accomplished with an altered design of a 3-D printer axis system. V-rail was used for the vertical movement because of the rigidity and accuracy it offered. A string drive system was implemented for the forward/backward movements because of how compact of a design it is. After settling on the design, it was obvious that stepper motors were necessary so that the gripper could accurately move to a particular spot quickly. Calculations for the strength of the stepper motors were then needed:

2.1.1 vertical stepper motor:

$$\text{Assuming weight W=3lbs=13.35N and gear diameter D=1 in.=0.0254m}$$
$$F_v >= W \text{ and } T = F_v * (D/2) \;\therefore\; T >= 0.17N*m \Rightarrow T >= 1.735kg*cm$$

2.1.2 horizontal stepper motor:

$$\text{Assuming weight W=3 lbs=13.35N, gear diameter D=0.5in=0.0127m and the coefficient of friction } \mu\text{=.1}$$
$$2F_f = W * \mu = 1.335N \text{ and } T = F_f * (D/2) \;\therefore\; T >= 0.00424N*m \Rightarrow T >= 0.0435kg*cm$$

The goal of using three rows is for the robot to be able to collect objects and sort them by color into separate rows. Utilizing the vertical height for storing blocks enables the robot to spend more time in the field collecting objects, to maximize the number of points it could score. Each row has enough space for two cubes and one sphere, a full color set. When the robot is ready to deposit the contents of the row, it grips the object furthest back and pushes the row out. The gripper assembly was designed to be cantilevered to the chassis to minimize the amount of space such a system would require. The gripper has two directions of motion: it can move vertically between rows via a belt attached to a stepper motor and it can also move along two rods that run parallel to the rows through the use of a pulley system connected to a second stepper motor. The assembly also makes use of limit switches to tell the robot when the gripper has reached certain locations such as the furthest forward or lowest positions it is allowed to have.

2.2 Electrical System

The electrical system of the robot is powered by a 12 volt, 4200 milliamp-hour, 50.4 Watt-hour battery pack mounted on the top shelf. The voltage is regulated down to 10 volts and 5 volts separately. The 12 volts run to the wheels and the wheel motor drivers, the 10 volts is for the gripper and stepper motors and the 5 volts is for the Arduino Teensy++ 2.0, the limit switches, and powering the stepper motor drivers. The stepper motor drivers are powered with 5 volts and use 10 volts through them to power the stepper motors. The Nvidia board is powered by 9 volts. The voltage regulators were installed on one of two custom-made circuit boards: the first board, dedicated to power, holds all of the voltage regulators as well as the common ground while the other board holds the Teensy and the motor drivers.

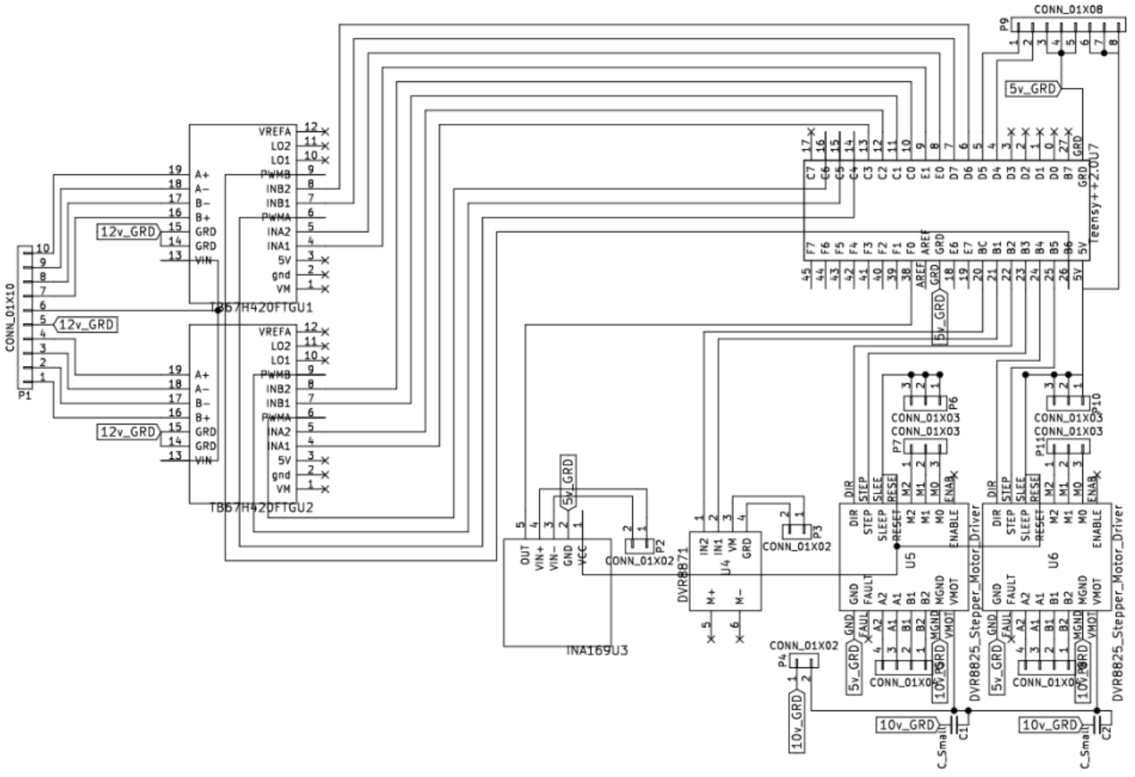


Figure 7. Electrical Schematic

There are 3 different motor drivers used. Two TB67H420FTG higher power chips for the wheel motors were used for its ability to drive two DC motors at 1.7A (max) per channel in small footprint. Two DRV8825 chips for the stepper motors were picked for it ability to drive stepper motors up to 1.5A per phase without the need for additional cooling and ability to drive bipolar stepper motors. One DRV8871 chip to control the gripper motor was used for it compactness and ability to drive DC motors up to 3.6A. The TB67H420FTG and DRV8871 operate similar in the sense that it takes two signals per motor from the Teensy to determine which direction to spin. The difference in these two drivers is that the gripper motor could only operate at full speed and run one motor, whereas the wheel driver could operate the motors at a speed determined by our controller, the PWM. The stepper motor drivers have two inputs from the Teensy, one for direction and one for rotating. These drivers operate differently though because when it gets the signal to turn, it only moves 1 step, where the default number of steps to complete 1 full rotation is 200. To make the motor continually turn, the signal has to be pulsed on and off. This driver also gives the option of making the step size as small as 1/32 of a step, which means we can move more accurately to an exact position.

The circuit boards were designed and wired to fit within a specific area on top of the Nvidia board that is above the wheels and below the heatsink.

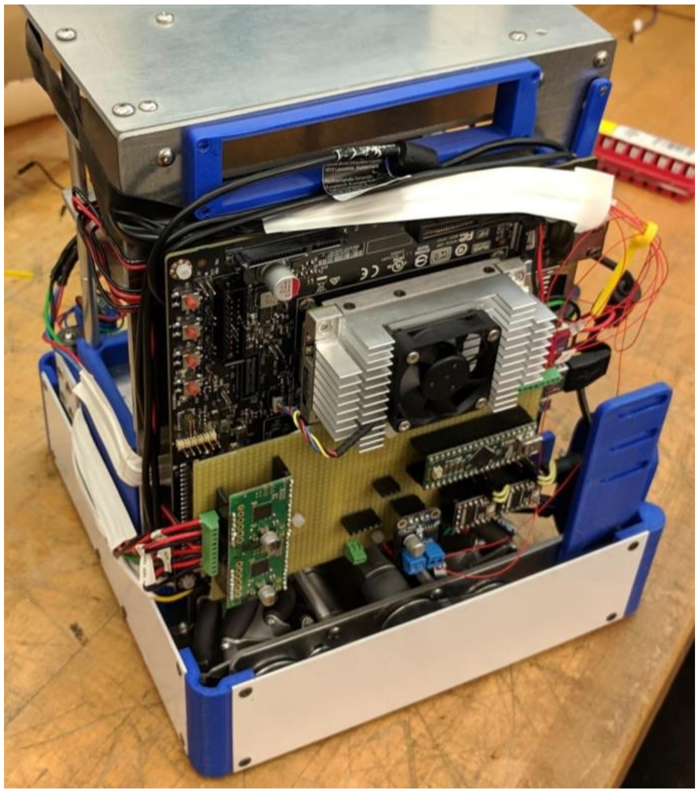


Figure 8. Full Electrical and Mechanical Assembly

Three holes were punched through the circuit board to line up with three mounting holes on the Nvidia. The wire paths were organized based on where the mounting holes needed to be and the components were organized to allow for connections to be on the sides they were closest to without being blocked by the Nvidia board as well as to avoid crossing any wires.

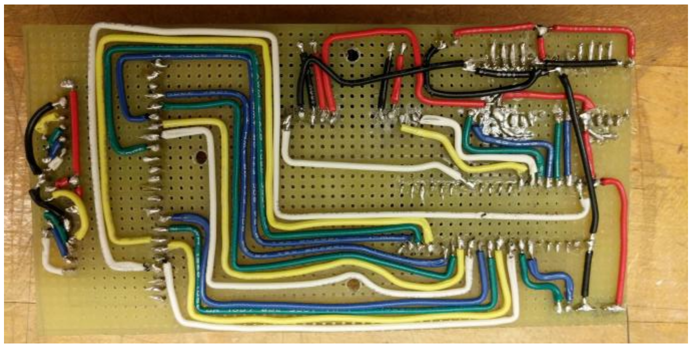


Figure 9. Teensy and Motor Drivers Configuration

2.3 Programming

The programming portion of the project was broken into four main components; the vision code, the decision code, the communication code, and the navigation code. All of the vision code and decision code was written on the Nvidia Jetson TX2 due to its processing power. Communication code between the Nvidia Jetson TX2 and the Arduino Teensy is on both of them. The functions used for driving were put onto the Arduino Teensy. In order to drive effectively, the wheels were run off of Pulse Width Modulation (PWM), which is a feature present on the Arduino Teensy and not the Nvidia Jetson TX2.

The vision code was written in the C++ programming language utilizing the OpenCV function library. OpenCV is an open source computer library that focuses on functions built for real-time computer vision. It allowed the robot to read information from a webcam or a similar sensor and perform operations to find and classify objects. The camera used in this project was a Logitech C310 HD webcam. The webcam was chosen based on two factors: its low cost and that it also provided high quality images within a 60° field of vision.

The vision code performed two main functions: image processing and object detection. When an image was brought in by the camera, operations needed to be performed in order to obtain data from it. Each image consisted of 480x640 pixels defined by the camera. The first step was changing the image from color to grayscale. In this project, it was more necessary to have clearly defined lines than color. This was done also to increase the speed of the code. The grayscale reduced the number of arrays the code had to run through by converting an NxMx3 BGR array to an NxMx1 array. The next step was to blur the image. It seemed counterproductive to blur an image but the vision system saw pictures differently than human eyes[4]. If a signal contained a lot of noise, a lot of false positives appeared when searching for lines or edges[4]. A Gaussian Blurring smoothed out the image and made the transition from one color to the next much smoother[4].

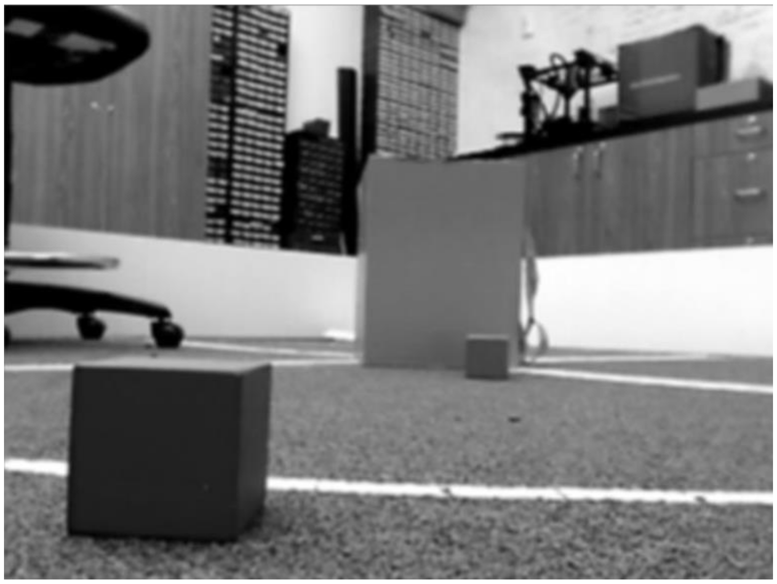


Figure 10: Blurred Image

The BGR image was converted to CIELAB color space and then masked for the different colors the robot was looking for on the board. The masks were overlaid with the grayscale image so that specific contours in the objects could still be identified. CIELAB was used so that the L vector could be ignored as dynamic lighting was an issue. The benefit of overlapping the masks and the gray scale is that it reduced the workable image size so that later functions could run more quickly.

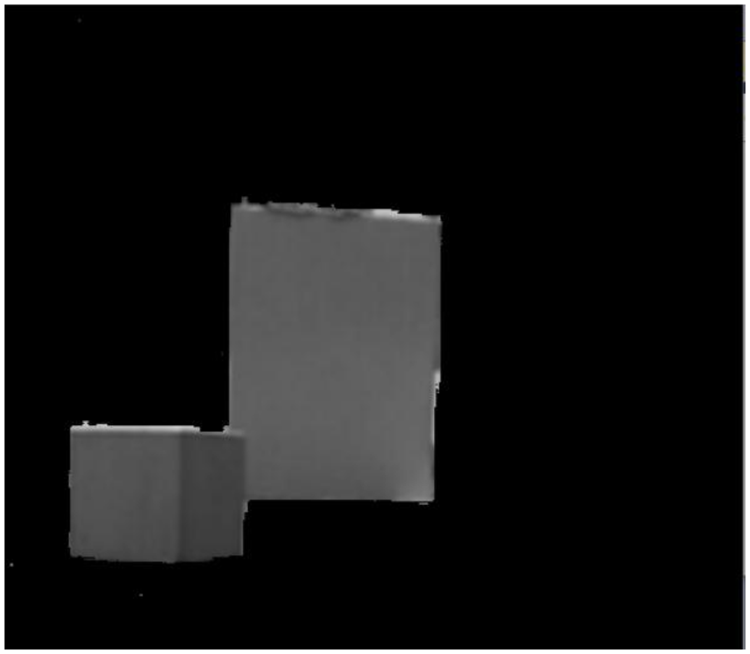


Figure 11: Color Filtered Image

Once the grayscale and blur were applied to an image, the next step was to locate any lines present. The Hough Line transform is a function implemented in OpenCV and is used to find lines in the images. The Hough Line transform worked by expressing lines in the Polar system[1]. For each pixel in the image, a family of lines are defined that pass through it[1]. When plotting the family of lines which pass through the point, the result was a sinusoidal wave[1]. This operation was repeated for all the points in an image and at each intersection, these points were considered to share the same line[1].

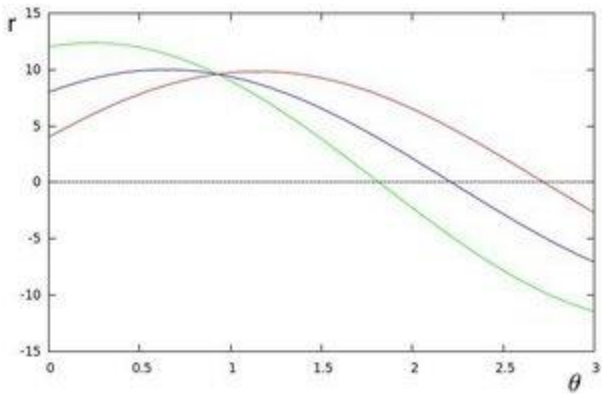


Figure 12. Hough Line Transform[1]

In order to use this function, it required a defined threshold value to represent the minimum number of intersections needed to detect a line[1]. The Hough Line Transform kept track of the intersections between curves and if the number of intersections was greater than the threshold, it defined the line with the Polar coordinates[1].

In the OpenCV library, a function was built for finding circles in an image called the Hough Circle Transform. The Hough Circle Transform worked nearly the same way as the Hough Line Transform[2]. A circle however could not be represented by the Polar coordinates as we needed three parameters to define a circle[2]. The three parameters were: the x-location of the center, the y-location of the center and the radius. The math behind locating the circles was more complex and used the Hough Gradient method in order to find them. This Hough Gradient method has been used in other research as well. This method began by using the equation:

$$(x-a)^2 + (y-b)^2 = r^2$$

(a, b) were set as the coordinate of the circle center and then r represented the radius of that circle[5]. (x, y) was transformed into a right circular cone in the (a, b, r) space[5]. The gradient method helped to speed up the overall process[5]. Its main purpose was to decompose the circle finding into two parts[5]. First, it had to find candidates for centers and then find a radius that satisfied[5]. The center candidates were checked against a threshold value and it kept the centers above that set threshold[5]. Necessary thresholding values were needed for the code to run faster as well[5]. A maximum and minimum radius were necessary because otherwise the function would run through looking for possible circles at every possible radius within the image. This could cause very small or large circles to be detected falsely as well as slow down the code significantly. In addition to this thresholding value, we defined the minimum distance between detected centers. This allowed us to also remove many of the overlapping circles we found in the image.

All of the information regarding the circles was stored within a 3-bit vector after the Hough circle transform was completed. This vector defined the location of the center of the circle, the radius of the circle. Using this information, a drawing function was called draw the circle where it located it on the image and then displayed it to the user's screen. Drawing the circles on the screen did not add any additional benefit within the code itself; this was used as a method of debugging so that the user could see what the webcam saw and whether all of the circles were positive hits.

Finding squares and cubes used different functions compared to the circles. The first step is to define the edges on the image. The function we used was the Canny edge detection. This function worked through multiple stages and was best explained by the OpenCV documentation. The first step was to reduce noise within the image[3]. For this, a 5x5 Gaussian filter was used[3]. This smoothed the image similar to a blur[3]. The next step was to find the intensity gradient within the smoothed image[3]. The image was filtered with a Sobel kernel in the horizontal and vertical directions and this gave us the first derivative in each direction[3]. Using this, the edge gradient and direction for each pixel were found using the formula:

$$Edge\ Gradient:\ G = \sqrt{G_x^2 + G_y^2}\quad Angle\ \theta = tan^{-1}\left(\frac{Gx}{G_y}\right)$$ [3]

The Gradient direction was always perpendicular to edges[3]. After the gradient magnitude and angle were found, a full scan was completed to remove any unwanted pixels[3]. Each pixel was checked to see if it had a local maximum in roughly the same direction as the gradient[3]. A minimum and maximum value needed to be chosen to test the gradient intensity[3]. Any edges that had an intensity gradient greater than the maximum value is considered to be a definite edge while any edge below the minimum was considered to be a non-edge[3]. The edges that lie between the two values were determined by their connectivity[3]. If it was between the two values and connected to a definite edge pixel, it was considered to be an edge as well[3]. This allowed a lot of the small pixel noises to be removed[3]. The Canny Edge function took the input image, which was the blurred, color filtered image, and returned a binary representation of the image[3]. The binary image was made up of either black or white pixels and this was used in finding the contours on the image[3]. This method worked well for this project but it also had its shortcomings. These were explored in research done by Jun Li and Sheng Ding. One problem corresponds to the use of the Gaussian function. The main purpose of this was to eliminate noise, however, when images were smoothed by the filter, the edges were treated as high frequency components and can be smoothed out[8]. Another issue that pertained to this project regarded the high and low threshold values[8]. These values were fixed, which reduced the adaptability for this code to separate images[8].

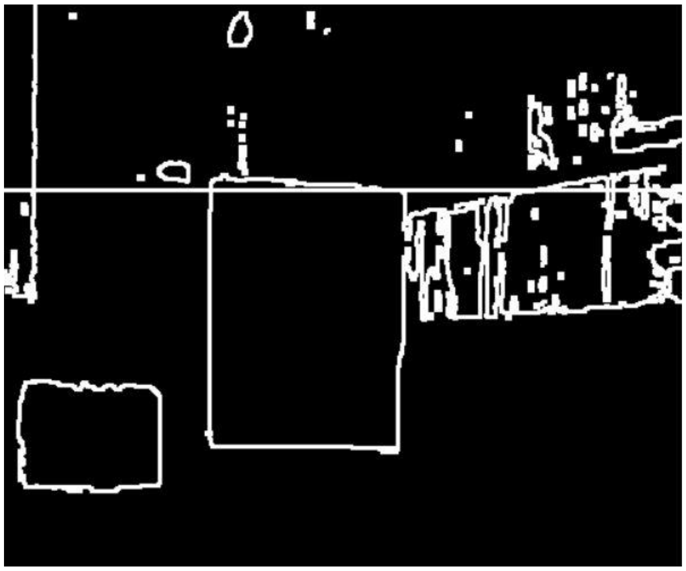


Figure 13: Edges on Image

The next step in image processing was to find the contours of the image. Information regarding how contours were found was included in research from Michael Maire. The contours can be described as a curve that joined all the continuous points along a boundary that shared the same color or intensity[6]. This returned an image that outlined the edges and this was used for finding the contours of the image[6]. Each contour was saved as a vector of points using the simple approximation method[7]. The simple approximation method compresses the line segments into only their end points[7]. A contour size of four was used to decide if a shape was a quadrilateral. An approximation was performed to create a more precise contour[7]. The function used for approximation was called ApproxPolyDP, which approximated a polygon with another polygon to reduce the distance between them[7]. With each cube and rectangle saved, thresholds were added to create a maximum and minimum area. In order to determine the difference between a square and rectangle, the aspect ratio of the object was used. If the object's aspect ratio was close to one, the object was considered a square. The shapes were then drawn on the screen in a similar way the circles were. The top left corner location was passed into the drawing function and then the square was drawn based on the height and width that was found.

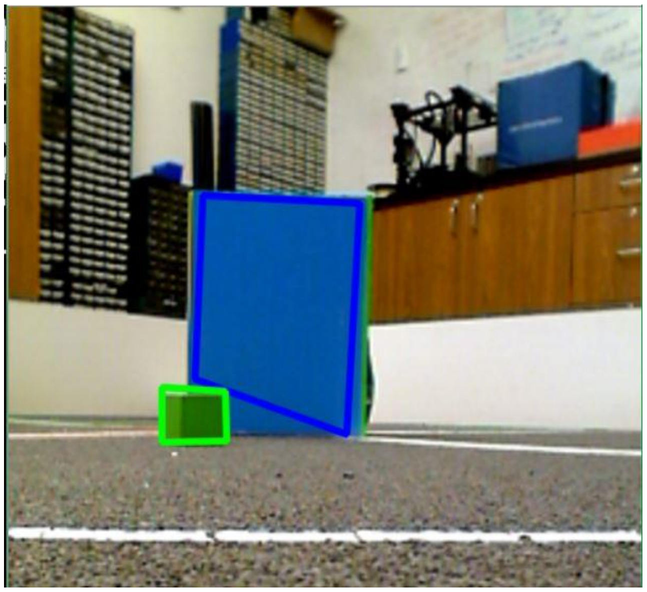


Figure 14: Final Image

The main function of the vision code began by opening a frame of video from the webcam. This frame was used as the image being passed into the image processing function and subsequent circle and square locating functions. Upon completion of these, the image was displayed to the user with the circles and squares drawn on the image itself. A copy of the original frame was also displayed to check if the proper items were being found by the image processing. Afterwards, the code grabs a new frame and repeats the process. This is why the Nvidia Jetson TX2 was chosen for this project. Its powerful operating system allows for the code to run faster than many other processors. Adjustments were made across the span of the project in order to speed up the operation of the code.

Once all of the data was collected from the vision code, a decision code needed to be used to decide where to go with the information provided. The data from the vision code was saved in an array that stores the location, distance, and color of the object it sees. The decision code processed what information needed to be sent to the Teensy based on the state sent by the Teensy. When objects were found, the Nvidia checked what state the Teensy was currently running. In the first state (000), no information was being sent to the Teensy, this was used as a reset or waiting state. The next state (001) asked for information regarding any object. Once an object was seen, it returned the closest object to the robot. It continued to send information regarding the same object until the Teensy signified that it was done looking for that specific object and returned to searching for any objects. The third state (010) allowed the Nvidia board to send information about the tower. This was used to determine the position of the robot on the board. The fourth state (100) was used to detect and navigate to the corners of the board. If in this state, the Nvidia would send information about the corners of the playing board and the robot would drive directly to the corner. Once close enough to a corner or object, the robot would enter the fifth state (011), which was the pick-up/drop-off state. Depending on what the previous state was, the robot would either pick up an object or deposit a certain row of objects that it had collected.

The Nvidia board used the Teensy's state machine to communicate information and was all controlled by the communication code. The communication system was composed of the Nvidia board sending a ten-character message to the Teensy with encrypted data describing what the vision system is seeing. The first and last character was used as a transmission character, we used a '$' as the start of the transmission and a '%' as the end. The purpose of these characters is to indicate to the Teensy that this is the start and end of a valid data line. The eight characters in between those two characters held information on the object's shape, color, angle from camera, and distance from the robot. The second character in the string can be one of four letters: 'S' which symbolizes squares, 'C' which symbolizes circles, 'T' which symbolizes the tower, and 'R' which symbolizes corners. Based on the state the Teensy is sending to the Nvidia board, the Nvidia board will send it the object that best corresponds. The third character in the string represents the color of the object it is looking at, sending a 'R' for red, 'B' for blue, 'Y' for yellow, and 'G' for green. The fourth, fifth, and sixth characters are used to tell the Teensy a three-digit value that is the angle of object from the center of its field of vision. The camera can see 480 pixels from the left to the right, therefore the center of its field of vision is 240 pixels from either the left or right, so the value being transmitted can be between zero and 480. The seventh, eighth, and ninth characters represent a three-digit value for the distance the object is from the robot. This units used for distance is centimeters, which we calculated by converting from pixels to centimeters using the following conversion equation:

$$\textit{Distance in Centimeters} = 27000 \,/\, (\textit{number of pixels for the object's radius or width})$$

In the Teensy communication code, it opened the line for communication and saved only the correct string of data from the Nvidia, parsing out anything else that might have come through or been stuck in the buffer. If the Teensy got a complete string and it corresponded to the correct object the Teensy was looking for, then the Teensy would process that data. The Teensy turned the three characters apiece it got for distance and x-coordinate into a single integer. This integer was fed into a PID specific to whether the object was a corner, tower, or an object to pick up. After this the Teensy would move into its movement code.

The movement code on the Teensy determined whether the robot would move laterally or vertically, how fast it would move, and how long it would move. Each object had its own distance and x-coordinate thresholds based on how accurately the robot needed to be away from it. If the robot was outside of the distance threshold, it would move vertically unless the object was too far to the left or the right, then the robot would move laterally. If the object was too far to the left or to the right and it drove forward, it could easily lose sight of the object. If the robot was within the distance threshold but outside of the x-coordinate threshold, then the robot would move laterally. Each PID controller for objects was capped at certain speeds. The output for the PIDs were the PWMs for the motors. The motors need a minimum amount of 55 PWMs to turn, so the PIDs were capped at a minimum of 55. They were also capped at a maximum to prevent the robot from moving too quickly and losing sight of the object. The maximum was different for each object, and if the object was a certain distance away and driving forwards, the maximum speed allowed was greater. If the object was very close to the distance and x-coordinate thresholds, then the robot no longer continuously drove, but instead was only allowed to drive a short distance. This was to help speed up the process of reaching the desired thresholds.

The navigation code, which was implemented on the Arduino Teensy, used a state machine (as seen in figure 11) and the communication previously discussed to determine the motion of which is required.

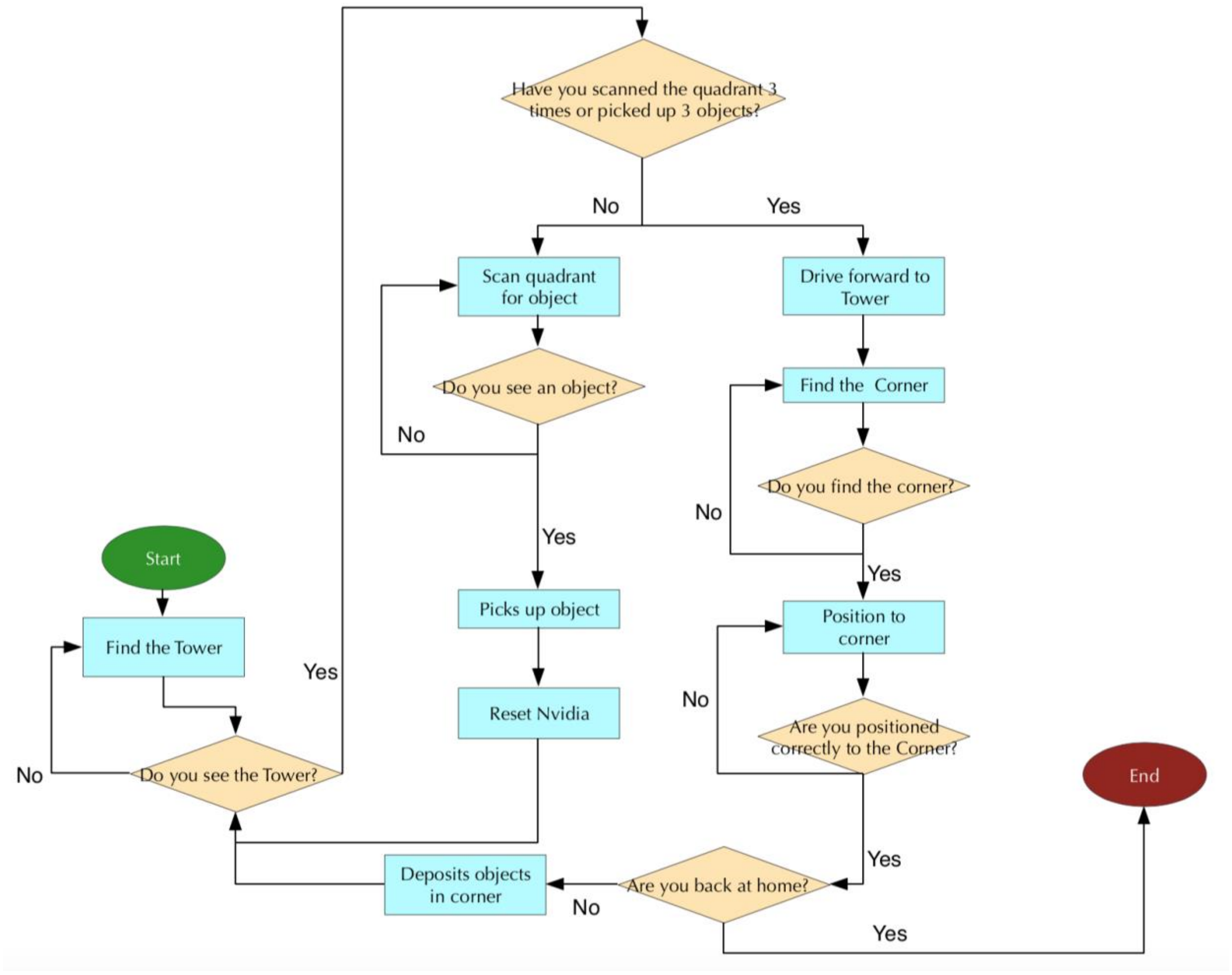


Figure 15. State Machine Diagram for Teensy

3. Application

3.1 Application of Mechanical System

The final design of the robot's chassis only made use of two rows as opposed to three: during testing, it was realized that having all three rows would make it difficult to find space for both of the circuit boards, used to power the robot and communicate with the Nvidia board. While this reduced the robot's projected ability to outcompete other robots by holding onto more objects, it was determined that there was no alternative for this issue and that it needed to be ensured that the robot adhered to the size constraints placed on the competition.

Throughout the design, there were several ideas on how to hold the objects within the chassis to ensure that the robot would not run into the issue of having an object leave one of the shelves or, in the worst-case scenario, cause harm to one of the mechanical or electrical systems onboard the robot. Initially, the plan was to use 3D-printed barriers on each shelf to ensure this goal. On the back of each shelf, the opening between the shelves were completely blocked except for a small opening. While this solution was workable, it was determined that the wiring connecting the gripper mechanism to the rest of the electrical system, which ran across the rear of the robot, would be sufficient for this end. In addition, a small plate was placed at the front of the object to prevent any of the objects from leaving the opening when the gripper is not actively trying to remove object(s). The gripper was originally planned to deposit all of the objects at once into a corner, but it was eventually decided that it needed to deposit an object one at a time in order for the robot to receive the best results. Another issue was that the limit switch that the gripper was supposed to hit as it was lowered was not being activated because of it slowly shifting each time it was activated, resulting in the code becoming stuck in a loop. One of the issues that occurred late into the testing phase was with the robot being inconsistent driving on the board. This problem was found to be the result of the hex nuts used on the wheel assemblies being tightened too much, resulting in two of the wheels being unable to move at the proper speed due to the hex nuts catching on the aluminum plates.

3.2 Application of Electrical System

The majority of the electrical system did not significantly deviate from the original designs. The greatest issues that occurred required troubleshooting when sections of the robot stopped functioning. Most of the testing that occurred with the robot was done with the Nvidia board being powered separately from the rest of the robot. It was connected to a monitor to allow for easier diagnostics of the vision system. When beginning to test the robot as a whole, the electrical needed to be adjusted minorly to include powering the Nvidia board to have the robot running on the battery power alone. After this was completed, a power switch and a start button needed to be included. Once this was completed, it was found that the start button was not dissipating power, backfeeding the circuit and powering the Teensy on. It was found that we needed a resistor to ground the start button, once this was added, it worked as expected. After experiencing this issue, it was deemed that the circuits should have been designed with diodes to have avoided similar issues in the first place.

3.3 Application of Programming

The final programming portion did not significantly change from the original code. The main portion that did change was the values used for thresholds and colors. Depending on location, lighting and the movement of the camera, the performance of the vision code would change. This was primarily an issue when it came to searching for the center tower or corners. To account for these changes, a line was drawn on our image to cut off any false positives above a specified camera height. The last change that was added was code to rotate the camera's image in the code. The webcam was mounted vertically rather than horizontally which cause the image in the code to be rotated by 90°. This was fixed in code by rotation so that the values for the location of the x and y axis were consistent with what the camera was actually seeing. While at the Von Braun Center, most of the time the team had was devoted to adjusting the code of the robot to work on the provided practice boards. This primarily involved making sure the robot could approach an object, align itself with the tower, and drive to a corner. After significant progress was made on the robot, the USB port on the Nvidia board stopped receiving signals. While a consideration of replacing the USB port was considered, the main priority became recovering all of the code stored on the Nvidia board's internal hard drive. This information was recovered by using Linux and ethernet to download the code to a flash drive. After diagnosing the Nvidia board, the command console output "port 1 over-current detected" and "tegra18x_phy_xusb_handle_overcurrent: clear port 1 pin 1 OC". Eventually it was determined that the only options would be to risk damaging the board by trying to ignore the overcurrent warning or to reflash the Nvidia board. The issue found with flashing the board was that it normally takes 4-6 hours to complete the process. Given the number of people using the Wi-Fi at the convention center, it was estimated to take up to 6-8 hours in the best possible case of having consistent Wi-Fi. After an effort was made to try to have the robot score a minimum amount of points without the Nvidia board, but it was determined that the robot was not going to be able to compete.

4. Data

The results of the vision system were quite positive. The distance calculations performed were close to perfect once they had been tuned. A distance of 100 cm would be reported as around 95 - 105 cm. These results were accurate enough for the robot to approach and successfully pick up objects with a high success rate.

The results of the cube finding function were also fairly good. The vision system could see the object from roughly six inches to about five feet, reliably. The object could be seen from closer to the robot as well as further from the robot but the results were not as accurate or reliable.

The ball detection code did not work as well as hoped. The color masking used for the balls did not perform to the same standard as the cube finding filter. The reflectiveness of the plastic that the balls were made of was the primary reason for this issue. Large portions of the ball would be seen as white by the vision code due to the light reflecting off of the balls. Because of this, the robot could only reliably see the blue and red balls. The yellow ball was unreliable but would be seen as red on occasion. The green ball would not be picked up under any situation and could not be seen overall.

5. Conclusion

Using computer vision combined with a mechanical gripper mechanism, the robot was able to autonomously pick up, sort, and store spheres and cubes of different colors. While the robot was ultimately unable to compete due to a hardware failure, it did meet all the goals set by the IEEE competition specs. The robot was able to start based on a button, leave the starting area, enter zone 2, pick up objects, navigate the board without colliding with any obstacles, deposit the collected objects, complete a lap around the board, return home and raise the team flag.

6. Acknowledgements

We would like to acknowledge our gratitude towards Dr. Rebecca Bruce and Professor Jeremy Brown for being our research supervisors and helping with scheduling, accountability, research, and travels. Their constructive suggestions and willingness to provide their time were vital to the completion and success of this project.

We would also like to thank Dr. Marietta Cameron for her assistance and knowledge in our computer vision system. Her knowledge in computer vision systems gave us a basic understanding and a starting place for our robot.

Our greatest appreciation to the University of North Carolina at Asheville, North Carolina State University, and the Western North Carolina Chapter of IEEE for giving us this opportunity.

7. References

1. Hough Line Transform - OpenCV 2.4.13.7 Documentation, “Hough Line Transform,” 12 July 2018, https://docs.opencv.org/2.4.13.7/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html.

2. Hough Circle Transform - OpenCV 2.4.13.7 Documentation, “Hough Circle Transform,” 12 July 2018, https://docs.opencv.org/2.4.13.7/doc/tutorials/imgproc/imgtrans/hough_circle/hough_circle.html.

3. Canny Edge Detection - OpenCV, “Canny Edge Detection,”12 July 2018, https.//docs.opencv.org/3.1.0/da/d22/tutorial_py_canny.html.

4. Cameron, M. (2018). Edge Detection. Presented during CSCI 412 - Computer Vision course.

5. Hong Liu, Yueliang Quian and Shouxun Lin, *Detecting Persons Using Hough Circle Transform in Surveillance Video* Chinese Academy of Sciences.

6. Michael Maire, *Contour Detection and Image Segmentation*, University of California, Berkeley, 2009.

7. Structural Analysis and Shape Descriptors - OpenCV 2.4.13.7 Documentation , “FindContours,” 12 July 2018, https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html

8. Jun Li and Sheng Ding, *A Reasearch on Improved Canny Edge Detection Algorithm,* Wuhan University of Science and Technology, 2011.