Spring 5-2-2023

# Programming an Autonomous Robot

Maxwell Brueggeman

Murray State University Honors College

HONORS THESIS

Certificate of Approval

Programming an Autonomous Robot

Maxwell Brueggeman

05/2023

Approved to fulfill the
requirements of HON 437

_____

Dr. Michael Siebold, Professor and Advisor
School of Engineering

Approved to fulfill the
Honors Thesis requirement
of the Murray State Honors
Diploma

_____

Dr. Warren Edminster, Executive Director
Honors College

Examination Approval Page

Author: Maxwell Brueggeman

Project Title: Programming an Autonomous Robot

Department: School of Engineering

Date of Defense: May 2, 2023

Approval by Examining Committee:

_____ _____

(Dr. Michael Siebold, Advisor) (Date)

_____ _____

(Dr. Gheorghe Bunget, Committee Member) (Date)

_____ _____

(Dr. Aleck W. Leedy, Committee Member) (Date)

# Programming an Autonomous Robot

Submitted in partial fulfillment
of the requirements
for the Murray State University Honors Diploma

Maxwell Brueggeman
05/2023

## Abstract

Ravaged by hurricanes, Florida needed help restoring its natural beauty and returning its wildlife to their homes. This was the task for the IEEE SoutheastCon 2023 Hardware Competition. Florida's restoration was simulated by returning various ducks and pillars that lay strewn across a game board to their proper places. Ducks needed to return to their pond, pillars needed to be stacked to create statues, and food needed to be placed in the manatee and alligator aquariums. Competing teams were challenged to create an autonomous robot capable of performing these tasks. During the first semester, sensor selection was tackled. Research was done on the appropriate sensors for each application, comparing their costs, abilities, and online resources. Eventually, four different sensors were selected. These were the Vl6180X time of flight sensor, two TCS34725 color sensors, a VL53L5CX time of flight sensor, and two Pixy2.1 cameras. Emphasis was then put on implementing those sensors at both the hardware and software levels within the team's autonomous robot during the second semester. A Raspberry Pi was used to program each of the sensors mentioned previously as well as the robot's servos, motors, and other electronics. The logic was then created and implemented for each gameplay function. The robot was built to start automatically, deliver the manatee and alligator food, intake and sort pillars based on color, and stack those pillars in the correct color order. At the competition, the robot could reliably auto start and deliver food to both the manatee and alligator aquariums. These processes were exceptionally consistent through three preliminary rounds, leading to the robot qualifying for the single elimination tournament. The team eventually placed seventh in the IEEE SoutheastCon 2023 Hardware Competition.

# Table of Contents

# Figures/Equations/Tables

# 1 Hardware Selection and Placement

## 1.1 Sensor Selection

The robot built for the IEEE Hardware Competition was fully autonomous. For a robot to perform such a complicated task with no external intervention, a wide variety of sensors was required. From vision to sorting to object avoidance, sensors gave the robot the information it needed to make the best choices.

Programming any autonomous robot is a complex and time-intensive task. This complexity made the sensor and camera selection process essential to the success of the final robot. Properly selected sensors also simplify the programming, so being able to select the sensors before programming them enabled ideas to formulate for how certain processes might be coded later on.

Various constraints required the IEEE robot's sensors to be cost-effective, small, consistent, and versatile. The robot's physical size was limited to one cubic foot, so any unnecessary electronics would consume valuable space while increasing the robot's cost. Several environmental factors, e.g. differences in ambient light and humidity levels, also impacted the robot's performance, so accurate and robust electronics were necessary. Most importantly, a rigorous sensor selection process would ensure that integration with the robot's hardware was relatively simple and predictable.

### 1.1.1 Procedure

The robot required color sensors, distance sensors, and cameras. The costs, abilities, resources, and limitations for each candidate were compared to find the optimal sensor for each application. In addition, increased emphasis was put on desirable characteristics that changed

based on the unique requirements for each robot subsystem. The following sections will show this review process for each sensor type, introduce the sensor requirements for each application, and detail the best sensor for each application.

### 1.1.2 Color Sensors

The sorting system needed color sensing to know what color pillar the robot had just consumed. Robots earned more points if they stacked pillars with white on bottom, green in the middle, and red on top [1]. To achieve this, a color sensor needed to face the sorting mechanism entrance and identify the red, green, and white pillars. This sensor needed to be consistent and accurate to avoid assigning the wrong color to a pillar.

The game startup mechanism also needed color sensing. According to the IEEE hardware competition rules, robots earn extra points by automatically starting when a red LED on the game board is illuminated [1]. A color sensor placed on the outside of the robot should detect this LED amidst varying ambient lighting conditions.

The TCS3200 is shown in Figure 1. This color sensor cost $7.90. It had several online tutorials and resources, but it lacked an IR filter on the camera and needed to be calibrated to sense color [2].


**Figure 1:** TCS3200 Color Sensor [3]

The ISL29125 is shown in Figure 2. This color sensor cost $8.50. It measured RGB intensity, not pulse-width like the TCS3200. A built-in IR blocker also provided more accurate color sensing results in varied lighting conditions. Lastly, it functioned over I2C connection, one of the most reliable sensor communication methods. The only drawback was that it needed an additional 2-channel logic converter to work [4].


**Figure 2:** ISL29125 Color Sensor [5]

The TCS34725 is shown in Figure 3. This color sensor cost $7.95. It could sense RGB values without calibration, had a built-in IR filter, used I2C connection, and had several helpful libraries and tutorials online. Additionally, individuals online chose this sensor over the ISL29125 for its high accuracy [6].

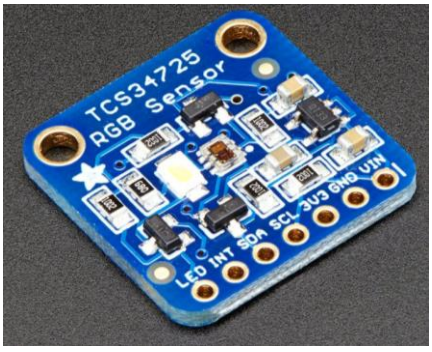


**Figure 3:** TCS34725 Color Sensor [6]

The results were tabulated in a decision matrix shown in Table 1. Each sensor's accuracy, programmability (online resources), and cost was ranked on a scale from zero to five, five being the best possible score. The total of these values was then normalized based on a perfect score of 30. Looking at the results, the TCS34725 was the cheapest by a small margin, but this difference was not enough to factor into the decision. The worst performing of all three was the TCS3200 which needed calibration to sense color, had the least accurate results of the three sensors, and did not have libraries for easy coding. The ISL29125 was not much better, needing an additional logic converter and getting less accurate results than the TCS34725.

The TCS34725 had the most functionality with the ability to get color values without calibration, and it was the most accurate when compared to all three sensors. The built-in IR filter and LED would also be very helpful in getting consistent measurements. Overall, the TCS34725 was better in every way and was the clear choice for both the sorting mechanism and the starting mechanism.

**Table 1:** Color Sensor Decision Matrix

| Weights: | x1 | x3 | x2 | **Normalized** |
|---|---|---|---|---|
| | **Cost** | **Accuracy** | **Programmability** | **Total** |
| **TCS3200** | 5 | 2 | 1 | $= \mathbf{12 \rightarrow 0.4}$ |
| **ISL29125** | 5 | 3 | 3 | $= \mathbf{20 \rightarrow 0.67}$ |
| **TCS34725** | 5 | 4 | 4 | $= \mathbf{25 \rightarrow 0.83}$ |

### 1.1.3 Distance Sensors

Two of the robot's systems required distance sensors. Sensing a pillar's entry into the sorting mechanism was the first. Upon sensing this entry, the robot needed to rotate the circular sorting mechanism to an open position so that it was ready to receive another pillar. A distance sensor for this application needed to work well at close range and have a resolution small enough to distinguish a change in distance of approximately 50 mm (the diameter of a pillar) [1]. These parameters, therefore, were weighted more heavily when choosing this sensor.

Sensing obstacles, such as walls and stacked pillars, also motivated the distance sensor selection. Without a sensor to detect and avoid the pillar stacks, the robot would run into them and ruin its progress. Recognizing walls would enable the robot to avoid them during gameplay and seek them when searching for and actuating the firework switch at the end of the run, a task to earn extra points. This sensor needed to have an 8 ft range to see across the entire gameboard and the capability of modeling a 3D space.

When looking into the different types of distance sensors, four main types were found that could work: time of flight (ToF) distance sensors, ultrasonic distance sensors, LIDAR, and IR sensors. Eventually, this was narrowed down to time of flight sensors and ultrasonic sensors, as IR sensors were too dependent on material shape and LIDAR was far too expensive and over-engineered for the given applications [7-8].

The VL53L0CX is shown in Figure 4. This time of flight sensor cost $7.25. It used I2C communication and had a range of 3 cm – 100 cm with a resolution of 1 mm in optimal conditions. Several online resources existed for this sensor as well, making it easy to program [9].

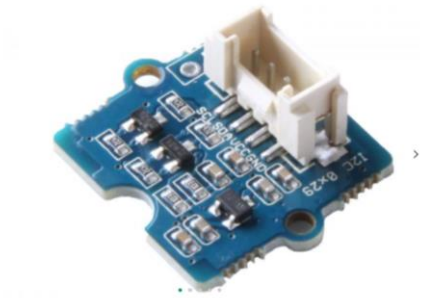

**Figure 4:** VL53L0CX Time of Flight Sensor [9]

The VL6180X is shown in Figure 5. This time of flight sensor cost $13.95. It was an I2C sensor with a 5 mm – 200 mm range and a 1 mm resolution in most conditions. It also had a multitude of online resources and libraries for easy programming [10].

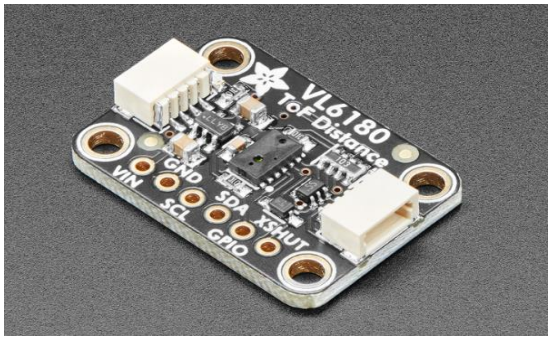


**Figure 5:** VL6180X Time of Flight Sensor [10]

The VL53L5CX is shown in Figure 6. This time of flight sensor cost $24.95. It was an I2C sensor that could create a 4x4 or 8x8 grid of distance readings for the viewing area. The grid would function like a distance heat map of the surrounding area, returning 16 or 64 distances at a rate of 60 Hz (i.e., 60 4x4 or 8x8 grids every second). Lastly, it boasted a maximum range of 4 m range and a 63° diagonal square field-of-view (45° x 45°) [11].

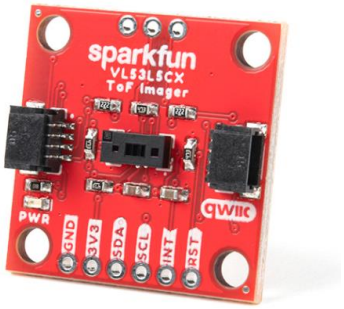


**Figure 6:** VL53L5CX Time of Flight Sensor [11]

The VL53L7CX is shown in Figure 7. This time of flight sensor cost $28.64. It was an I2C sensor, and it functioned exactly like the previous VL53L5CX time of flight sensor but had a 90° diagonal square field-of-view (63.6° x 63.6°) [12].


**Figure 7:** VL53L7CX Time of Flight Sensor [13]

The Grove Ultrasonic Distance Sensor is shown in Figure 8. This sensor cost $3.95. It had a 3 cm – 350 cm range with a resolution of 1 cm. It also had several online resources and libraries for simple programming [14].

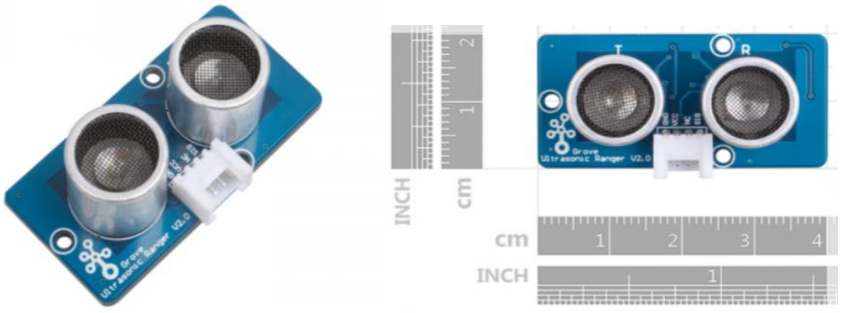


**Figure 8:** Grove - Ultrasonic Distance Sensor [14]

The widely used HC-SR04 ultrasonic distance sensor was not included in this report because the Grove ultrasonic distance sensor was better in nearly every capacity when compared to the HC-SR04. This comparison can be seen in Table 2 where the Grove's broader voltage compatibility, reduced pin connections, and plug-and-play connection pushed it past the HC-CR04. The HC-SR04 boasted a slightly broader measurement range and smaller resolution of 0.3 cm (not included in the Table 2), but these were not enough to warrant choosing it over the Grove sensor [7].

**Table 2:** Grove vs HC-SR04 [7]

| Sensor | Grove – Ultrasonic Distance Sensor | HC-SR04 |
|---|---|---|
| **Working Voltage** | 3.3V / 5V compatible<br>Wide voltage level: 3.2V – 5.2V | 5V |
| **Measurement Range** | 3cm – 350cm | 2cm – 400cm |
| **I/O Pins needed** | 3 | 4 |
| **Operating Current** | 8mA | 15mA |
| **Dimensions** | 50mm x 25mm x 16mm | 45mm x 20mm x 15mm |
| **Ease of pairing with Raspberry Pi** | Easy, direct connection | Voltage Conversion Circuit Required |

Short-ranged, consistently accurate readings were most important for the sorting mechanism's distance sensor. Table 3 compares these features and shows that the VL6180X performed best in both. Neither the VL53L0X nor the Grove ultrasonic sensor had a smaller resolution than the VL6180X, and they both had minimum ranges too low for reliable sensing.

Pillars entering the system would be closer than 30 mm to the sensor and could cause unpredictable readings from the two longer ranged distance sensors.

**Table 3:** Comparison of Distance Sensors for Use in Sorting Mechanism

| Sensor | VL53L0X ToF Sensor | VL6180X ToF Sensor | Ultrasonic Distance Sensor (Grove) |
|---|---|---|---|
| **Range (mm)** | 30 - 2000 | 5 - 100 (200) | 30 - 3500 |
| **Resolution (mm)** | 1 (in optimal conditions) | 1 (consistent) | 10 |

The VL53L5CX and VL53L7CX were the optimal candidates for obstacle detection. Both were inexpensive for their functionality, had long distance ranges, and were able to model their field-of-view in a 4x4 or 8x8 grid of distance readings. Their 3-D modeling capabilities would give the robot vital information on which direction was safe to travel. Additional programming could then distinguish between stacked pillars, nearby walls, and far walls. The deciding factor, therefore, was the field-of-view (FoV). Because the output from either sensor was an 8x8 grid, the VL53L7CX's increased FoV resulted in objects farther away becoming less detailed. The VL53L5CX time of flight sensor, with its lower FoV, was the best option for obstacle detection.

### 1.1.4 Cameras

Two systems in the robot required cameras. Tracking game pieces was the first. By being able to sense unique game pieces on the board, such as differently colored pillars and ducks, the robot could travel toward the pieces it needed to intake. Cameras for this application needed to be

able to distinguish between identically shaped pieces based on color and recognize each type of game piece.

Identifying different zones on the game board required a second camera. Placing food in the appropriate aquariums, stacking pillars inside designated areas, and moving ducks into the recycling areas all required accurate identification of zones on the game board. This zone identification was most easily achieved via a second camera system. Candidate camera models for this system needed to recognize each of the various colored areas on the gameboard.

The Pixy2 is shown in Figure 9. This camera cost $65.95. It was a 60 Hz camera that contained several unique features, the most promising being its color-connected components algorithm, which enabled the Pixy2 to learn the color of an object and identify it later. It could store up to seven different color signatures and match those signatures to objects in its view. In each frame, the camera could track up to one hundred objects matching those seven color signatures. It also had an onboard image processor to handle all the visual calculations for tracking objects and assigning color signatures [15].

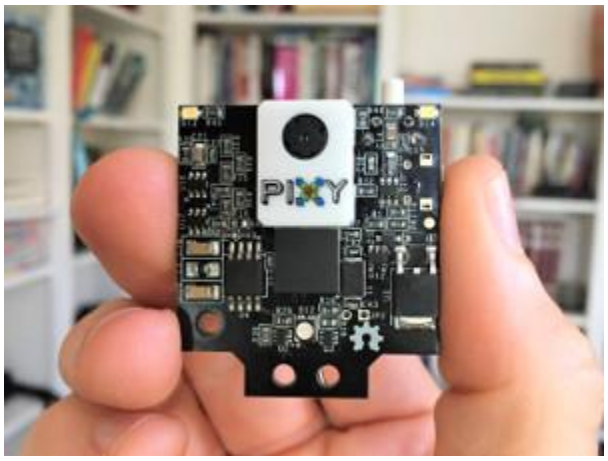


**Figure 9:** Pixy2 Camera [15]

The Pixy2.1 is shown in Figure 10. This camera also cost $65.95. It was a direct upgrade to the Pixy2 camera with less distortion and noise, and it boasted a wider field-of-view at 80° compared to the Pixy2's 60° horizontal and 40° vertical field-of-view [16].

**Figure 10:** Pixy2.1 Camera [16]

The HuskyLens is shown in Figure 11. This camera cost $54.90. It could learn one unique object and track that object no matter the color. Additionally, it could learn one unique color and track that color as well. It used I2C communication and came with a small user-interface screen to make programming the camera easier [17].

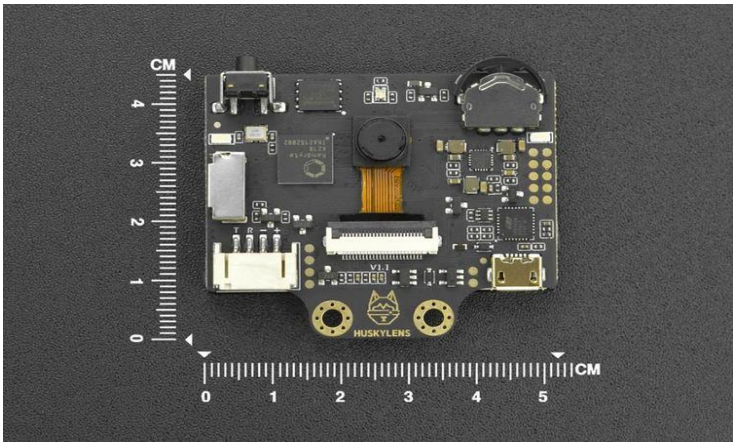


**Figure 11:** HuskyLens Camera [17]

Several other cameras used by hobbyists like the ESP32 and NVIDIA Jetson Nano were researched, but they either did not have the object recognition capabilities necessary or were too complicated for this project. The only cameras that were both affordable and made object detection simple were the Pixy2, Pixy2.1, and HuskyLens cameras.

Comparing the Pixy2 and Pixy2.1, the Pixy2.1 was a direct upgrade to the Pixy2. It gained several quality-of-life improvements without sacrificing any key features. Pixy2, therefore, was not considered in the sensor selection.

Identifying identically shaped objects by their colors was the first game piece detection requirement. Looking at videos of the Pixy2.1, it excelled at detecting and discriminating between different colored objects, being able to track dozens of colored balls falling through the air [15]. HuskyLens could also track different colored objects but at a much lower level, admitting that "color recognition is greatly affected by ambient light" and that it "may misidentify similar colors" [18]. The Pixy2.1's color-connected components algorithm gave it the capability to detect multiple objects of several different colors at the same time. In fact, it could recognize an object by color, categorize it, and then assign it a unique tracking index so the robot could focus only on that one object during its tracking.

Identifying the type of individual game pieces was the second requirement informing the camera selection. Pixy2.1 fulfilled this requirement through its color-connected components algorithm, as it could remember seven different color signatures and track all instances of them in every frame. The IEEE hardware competition featured exactly seven different colored game pieces. HuskyLens, however, could only track one type of object at a single time. It might have been able to see every pillar on the gameboard, but it would not be able to find the pond, aquariums,

statue areas, ducks, or recycling areas. Additionally, HuskyLens could not be operated in both color detection and object detection mode simultaneously.

The decision matrix for the three camera options can be seen in Table 4 where each category is ranked on a scale from zero to five, five being the best possible score. Weighted highest is the sensor's ability to identify game pieces followed by its overall consistency, programming simplicity, and cost. The total of these values is then normalized based on a perfect score of 50. The result shows that the Pixy2.1 was the best option for game piece detection.

**Table 4:** Camera Decision Matrix

| Weights: | x4 | x3 | x2 | x1 | **Normalized** |
|---|---|---|---|---|---|
| | **Ability** | **Consistency** | **Programmability** | **Cost** | **Total** |
| **Pixy2.1** | 5 | 4 | 3 | 3 | $= \mathbf{41} \rightarrow \mathbf{0.82}$ |
| **HuskyLens (Object Recognition Mode)** | 1 | 4 | 3 | 4 | $= \mathbf{26} \rightarrow \mathbf{0.52}$ |
| **HuskyLens (Color Recognition Mode)** | 4 | 2 | 3 | 4 | $= \mathbf{32} \rightarrow \mathbf{0.64}$ |

The second camera identified different colored sections of the game board. Shown in Table 4, the Pixy2.1 outperformed the HuskyLens color recognition mode. Because the gameboard had less than seven different colored areas, an additional Pixy2.1 was the best choice for this application.

### 1.1.5 Additional Purchasing

Three additional electronic components were purchased for the robot. These were necessary for robot functionality and programming, and they interfaced well with the chosen sensors.

The TCA9548A is shown in Figure 12. This eight-channel I2C multiplexer cost $6.95 [19]. Each channel was its own I2C bus, so it enabled the use of multiple sensors with the same I2C address. No two sensors can have the same I2C address on the same I2C bus. The VL6180X time of flight sensor and both TCS34725 color sensors had I2C addresses of 0x29, so the multiplexer was necessary. It was also produced by Adafruit, so it interfaced well with other Adafruit sensors such as the VL6180X and TCS34725. Programming the TCA9548A required setting its I2C connection to the main I2C bus and then initializing all other sensors with an I2C connection corresponding to their specific TCA channel.

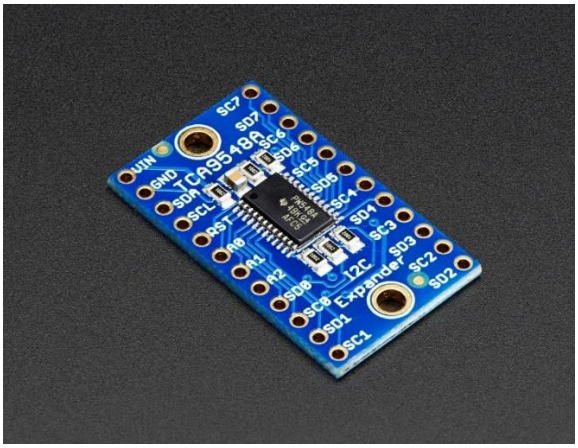


**Figure 12:** TCA9548A 8-Channel Multiplexer [19]

The PCA9685 is shown in Figure 13. This sixteen-channel servo driver cost $14.95 [20]. It connected to each of the eight robot servos and supplied them with PWM voltage. This was essential for proper servo functionality. It was also an Adafruit electronic, so the programming

was intuitive. After loading in the Adafruit Servokit library and initializing the servo driver, all servos could be initialized with the PWM connection of their respective PCA9685 channels.

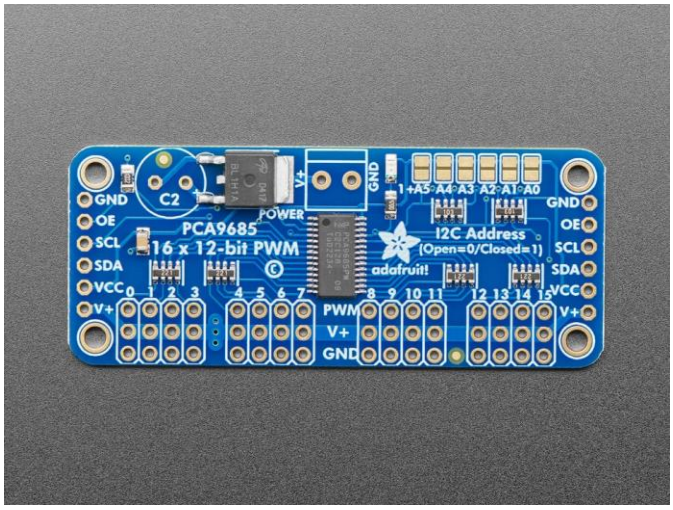


**Figure 13:** PCA9685 16-Channel Servo Driver [20]

The MCP3008 is shown in Figure 14. This analog-to-digital converter cost $4.50. It took analog signals from the sorting cylinder encoder and converted them to digital signals able to be read by the Raspberry Pi. It was also an Adafruit electronic, so it had a multitude of online resources [21].

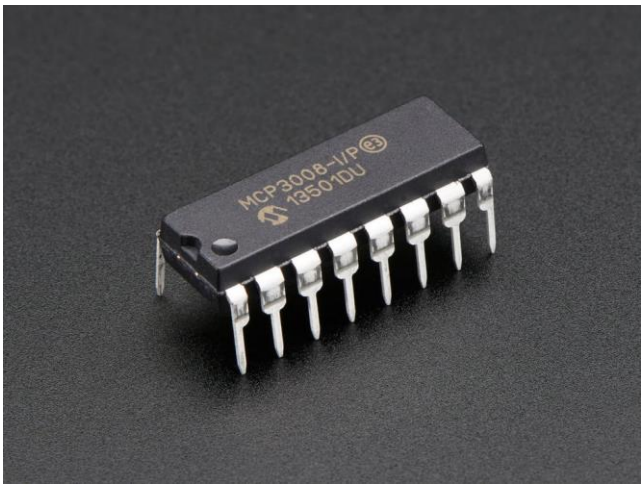


**Figure 14:** MCP3008 Analog to Digital Converter [21]

## 1.2 Sensor Visualization and Simulation

After selecting the robot's sensors, simulating their placement on the robot enabled the team to determine which sensor configurations yielded the best performance. This visualization was difficult to perform without the sensors in hand, so an accurate simulation was vital to the success of the robot. The following section will dive into the mathematics and coding used to create a simulation for the VL53L5CX time of flight distance sensor.

Using Godot, an open-source 3D game engine, the game board and game pieces were modeled, and a robot was created which the player could control. Attached to this robot was a camera and raycast which would change its position to 64 evenly spaced locations within the VL53L5CX's 63.6° diagonal square field-of-view (FoV). The raycast was an invisible line that extended outward from the player and provided information about any object it contacted. When this raycast collided with a surface, it would output the collision location which could then be converted into a distance traveled. These distances were then plotted on an 8x8 grid to show what a VL53L5CX would display in real life. A simplified version of this interaction is shown in Figure 15.

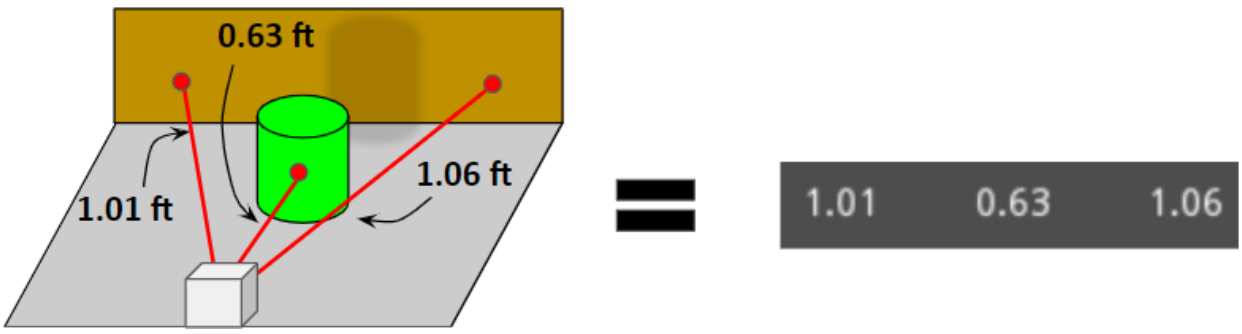


**Figure 15:** Simplified Distance Mapping with Raycasts

The first step in this process was creating a variable for a maximum ray distance equal to the VL53L5CX's maximum range (13.1234 ft or 4 m) and creating another variable for the horizontal/vertical FoV of 45°. The following equation was used to find the maximum offset:

$$offset_{max} = ray\_distance_{max} \cdot \tan\left(\frac{FoV}{2} \cdot \frac{180}{\pi}\right) \ . \tag{1}$$

The maximum offset was the distance from the center of a plane to its closest edge. Perpendicular to the ground and positioned at a distance equal to the maximum ray distance, the plane in question represented the maximum reach of the VL53L5CX's rectangular FoV as seen in Figure 16 and would be used as a location for sending raycasts. Its center would also be the x and y origin relative to the camera's center.

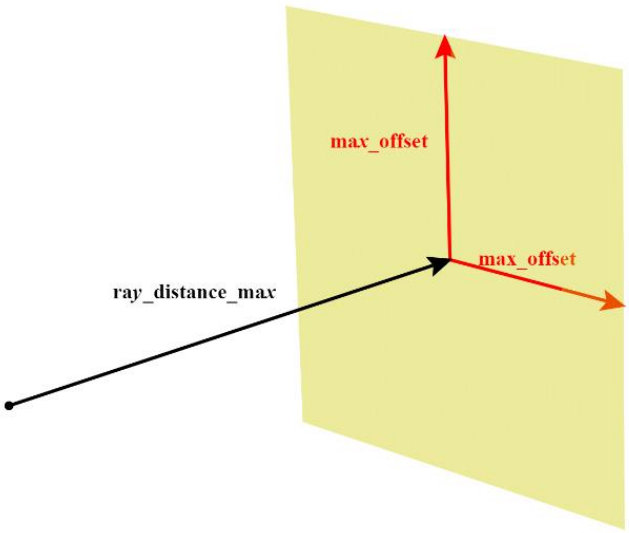


**Figure 16:** Field-of-View Projection Plane

To create eight rows and eight columns of evenly spaced points over this plane, the plane would have to be divided into eight sections. The x and y positions can be found for any point in the $i^{th}$ row and $j^{th}$ column by plugging the previously calculated maximum offset into the following equations:

$$x_{position} = \left(-1 + (j-1) \cdot \frac{2}{7}\right) \cdot offset_{max} \text{ , and} \quad (2)$$

$$y_{position} = \left(1 - (i-1) \cdot \frac{2}{7}\right) \cdot offset_{max} . \quad (3)$$

The raycast location could then be set to a vector where the x coordinate and y coordinates matched the x and y positions solved for previously and the z coordinate matched the maximum ray distance. Then by looping over eight rows and eight columns, checking for the collision point of each raycast, and storing these in a list, it was possible to display an 8x8 grid of distances like the VL53L5CX would in real life.

The final simulation can be seen in Figure 17 and Figure 18 where the red dots represent the collisions of each raycast and the numbers in the bottom right represent the distances in feet associated with these collisions. Being able to simulate possible gameplay experiences and use that information to modify the VL53L5CX's location was an invaluable tool. The game board and game pieces being to scale would also allow for a smoother migration to hardware.

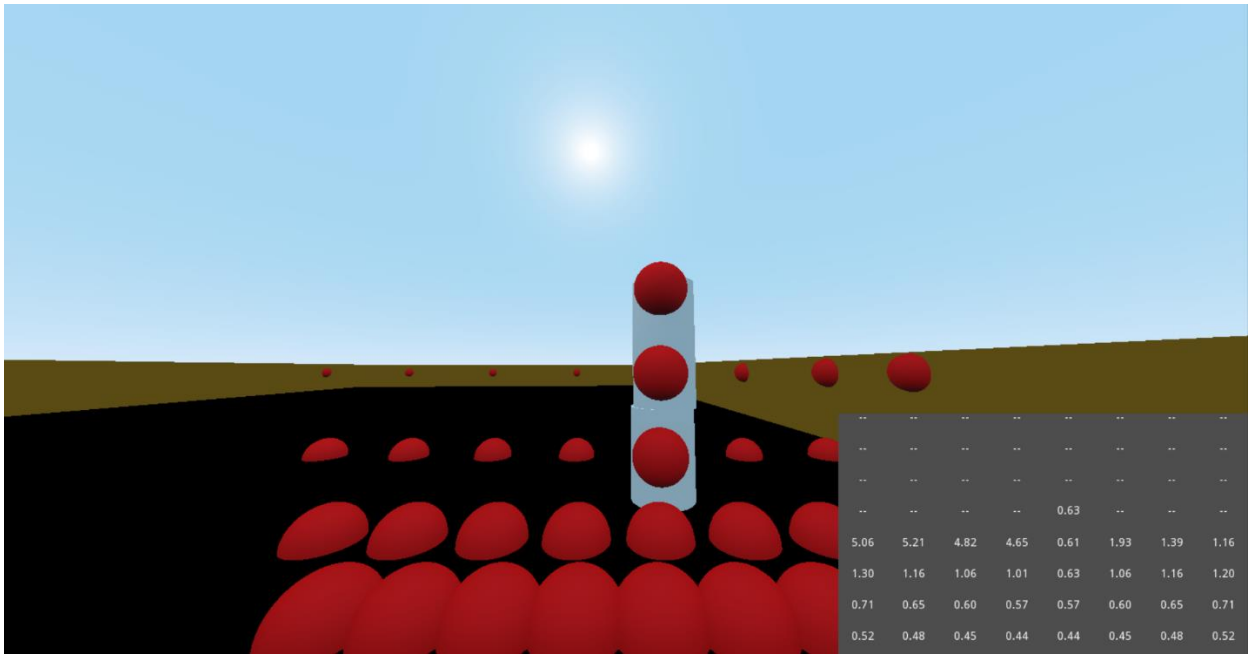


**Figure 17:** Godot Simulation for VL53L5CX

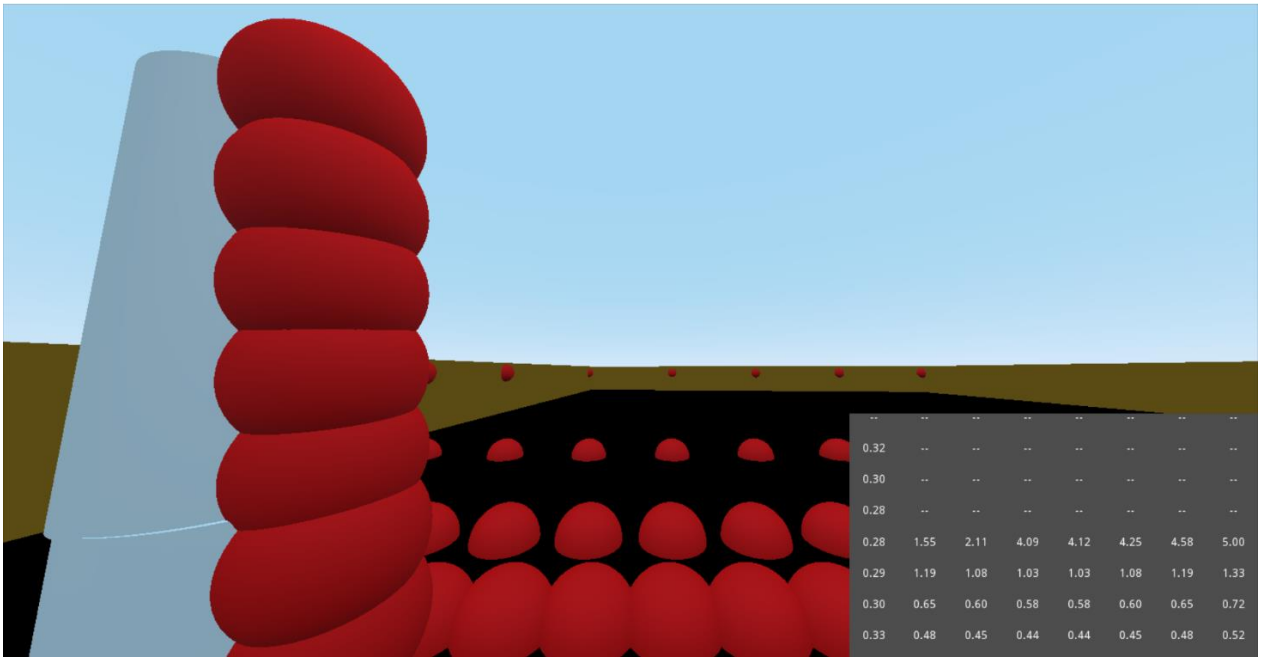


**Figure 18:** Godot Simulation for VL53L5CX (Cont.)

# 2 Software Development and Implementation

## 2.1 Controlling Individual Systems

Mentioned previously, programming any autonomous robot is a complex and time-intensive task. This process started with learning to interface with each of the individual electronics. Among these were the Raspberry Pi, the chosen sensors from Section 1, motors, servos, and other electronics. The Raspberry Pi was the most important of these, as it facilitated the addition of autonomy to the robot.

### 2.1.1 Raspberry Pi

The idea to use a Raspberry Pi as the microcontroller was teammate Nolan Hays'. Inexperienced with microcontroller selection, the team defaulted to his positive experiences with the Pi in the previous year's IEEE SoutheastCon competition. Four additional items were also purchased for use with the Raspberry Pi. These were an ethernet cable, microSD card, approved power bank, and heat sinks.

Ethernet Cable: This connected the Raspberry Pi to a laptop or computer. It allowed the user to send scripts, access the terminal, and communicate with the Pi at any location. It also enabled the Pi to access the internet through the computer's Wi-Fi connection so it could update and download necessary libraries.

The Sandisk A1 32GB Extreme Pro is shown in Figure 19. This microSD card enabled the Raspberry Pi to read and write information. It was rated as one of the best microSD cards for Raspberry Pi's, boasting 100MB per second read and 90MB per second write speeds [22].


**Figure 19:** Sandisk A1 32 GB Extreme Pro MicroSD Card [22]

Approved Power Bank: This powered the Raspberry Pi with 5V 3A. It eliminated the possibility for over voltage and low voltage issues.

Heat Sinks: These cooled the Raspberry Pi. The Pi did not come with an internal cooling mechanism, so these heat sinks were necessary for its safety. Overheated Raspberry Pi's could become damaged or corrupted.

Two communication methods existed for the Raspberry Pi. The first was Secure Shell Communication (SSH), and the second was Virtual Network Computing (VNC). Both were exceptionally useful.

SSH was the preferred method for connecting remotely to a Raspberry Pi. It allowed the user to access the Pi's terminal from a personal computer and made headless implementation, or running the machine without a monitor, possible. PuTTY was used to establish SSH connection with the Raspberry Pi because it stored the connection address as a saved session for future logins.

VNC was also used to connect remotely to the Raspberry Pi. Unlike SSH connection, VNC connection gave access to the Pi's graphics user interface (GUI). This was extremely helpful when searching for file locations on the Pi and troubleshooting network connection issues. To establish VNC connection, VNC Viewer was used which also saved previous connections and made it simple to immediately connect to the Pi's GUI.

Writing and executing code with the Raspberry Pi required safety procedures. This was the advice from author Danny Staple in his book *Learn Robotics Programming: Second Edition* where he warned against writing programs on the Raspberry Pi [23]. If the Raspberry Pi ever lost power unexpectedly or if the microSD card became damaged, the microSD card and all code on it could become corrupted. He advised having a separate location where the code could be stored and modified before sending it to the Pi.

The solution to this problem was the Secure File Transfer Protocol (SFTP) [23]. It was possible to create and modify all Python programs on a laptop and then use FileZilla, an SFTP tool, to send those scripts to the Raspberry Pi via SSH connection. This way, if the Pi crashed or became corrupted, the code would be backed up. The only downside to SFTP was that it added upwards of ten seconds to every debug process.

Three proper operating procedures were also employed when using the Raspberry Pi. Proper shutdown procedure was the first. Before unplugging the Pi, the command 'sudo poweroff' needed to be executed to safely shut it down. During this process, the Pi wrote all necessary information to the microSD card and prepared it for power loss. This process was signified by a flashing green LED at the base of the Raspberry Pi. If the Pi became unplugged while this green light was flashing, the microSD could become corrupted.

Employing the shutdown procedure before connecting or disconnecting any wires was the second proper operating procedure. Working with live wires increased risk for GPIO pin shorting and overvoltage.

Preventing overvoltage was the last proper operating procedure. The Raspberry Pi GPIO pins were only rated for 3.3V, so 5V logic damaged these pins. Double and triple checking all wire connections before powering the Raspberry Pi ensured this would not be an issue.

### 2.1.2 Preliminary Plan

The preliminary plan began with interfacing with each of the sensors individually and writing basic scripts to show their functionality. Programs controlling all sensors simultaneously could then be written. This process would be repeated for the servos, motors, and other electronic components. A visual gameplay flowchart for the robot's performance was also essential, providing the basic logic for each gameplay function. Once these two things were completed, scripts could be written for each of the robot's major functions: Startup, Emergency Stop, Food Delivery, Pillar Tracking, Intake, Sorting, Stacking, Statue Area Tracking, Pond Tracking, and Firework Light Switch Tracking. These scripts could then be combined into a final coding framework for the robot. In addition, multi-threading could be used to send information between simultaneously running scripts which would inform gameplay decisions.

### 2.1.3 Sensors

Testing and programming each sensor individually was the first step. The VL6180X time of flight sensor and TCS34725 color sensors were the easiest to program because they were ordered from Adafruit. Not only did Adafruit electronics have the most online resources and libraries of any sensors on the market, but they all had dedicated CircuitPython libraries. CircuitPython was a user-friendly programming language employed in several commercial microcontrollers, and Adafruit had a special library called Adafruit-Blinka to allow their

CircuitPython libraries to work with basic Python. Since the Raspberry Pi uses Python, this Adafruit-Blinka library made interacting with Adafruit sensors much simpler.

Programming the VL6180X time of flight sensor was simple, and its output was accurate. After connecting the sensor to an I2C bus, the output range could be called from a single function. Measuring the true distance with a ruler and comparing it to the output range, the two were nearly identical for all ranges less than 70 mm. No testing was done at farther ranges because it would never need to sense anything that far away when installed in the robot.

The TCS34725 color sensors were equally simple to code. A single function gave the color output as a list where the red, green, and blue values were items in the list. This sensor also gave accurate color readings for objects during testing. When putting the red side of a Rubik's cube in front of the sensor, the red output would spike while the green and blue values would fall. Faced with a white object, all values would even out.

Unfortunately, the VL53L5CX time of flight sensor was exceptionally difficult to program. It was not an Adafruit sensor, so it used a different library called the VL53L5CX CTypes Python Wrapper. This library used the smbus2 library to connect via I2C, and it refused to coexist with the Adafruit-Blinka library which used a different base library for I2C connection. The solution was an initialization parameter called i2c_dev in the VL53L5CX CTypes Python Wrapper library. By default, this parameter was set to I2C BUS1 (smbus2.SMBus(1)), but it could be changed to I2C BUS0 (smbus2.SMBus(0)) when initializing the VL53L5CX object. By changing this value from BUS1 to BUS0 and physically connecting the sensor to I2C BUS0 on the Raspberry Pi, the VL53L5CX could function without fault on its own, separate I2C bus. Using I2C BUS0 for sensor connection is usually frowned upon since the pull-up resistors on this bus are disabled. This, however, was not an issue because the internal pull-up resistors on the VL53L5CX board were

enough to promote safe function. After some modifications to the VL53L5CX library's example code, the sensor's 8x8 distance grid was set to the correct orientation, and the distances were found to be accurate to within a few millimeters.

The Pixy2.1 cameras worked very well in the example code. Using PixyMon, a program that showed what the Pixy2.1 was seeing in real time, the Pixy2.1 was able to be taught the green, blue, and red colors from a Rubik's cube. It could then detect and track those colors in the Pixy2.1 library's example code.

During attempts to create custom Python code for the Pixy2.1 cameras, however, the library's functions were unreachable. The solution was to delete the main Pixy2.1 library and download a different Pixy2.1 library with Simplified Wrapper and Interface Generator (SWIG) compatibility. The SWIG library was used for interfacing with libraries created in C or C++, both of which the main Pixy2.1 library was written in. After making this switch and then copying the new library's two important initialization scripts to the same directory as the custom Python program, the pixy was able to be controlled from a custom script. Coding for the pixy was unique and simple. A single line of code would refresh the Pixy2.1 whenever written, and all information for each of the blocks sensed could be found within the Pixy2.1's list of sensed blocks.

### 2.1.4 Servos

The team purchased eight servos for various robot functions: three HSR-2645CRH continuous rotation servos, three HS-485HB standard servos, and two M0090 micro servos. The continuous rotation servos rotated the left and right intake cylinders as well as the sorting cylinder. Then the standard servos pushed pillars into the stacking mechanism, opened the stacking

mechanism, and deployed the light switch arm. Lastly, the two micro servos dropped the red and green food pellets into their respective aquariums.

The Adafruit-Blinka Servokit library was used to control the servos instead of the standard GPIO library, RPi.GPIO, which caused severe servo jittering. Programming with the Servokit library, the continuous rotation servos were driven with a 'throttle' parameter ranging from -1 to 1. Likewise, an 'angle' parameter could be set from 0 to 180 for the standard servos. These angles and throttles did not reflect the actual angle or throttle measurements but rather the maximum and minimum values for each servo. To find the real maximum angles, a simple script was written to move each standard servo from 0 to 180, and the actual angle difference was measured with a protractor. These values varied widely from 115° to 150°. No servo needed a full 180°, so this discrepancy did not affect the final robot performance. Tests were also done on each of the continuous rotation servos to see what throttle value would put the servo at rest. A throttle of 0.1, not 0, worked to stop each servo.

### 2.1.5 Motors

The team purchased two EMG-30 motors for the robot drivetrain. To control these motors, the TB6612FNG Dual H-Bridge motor driver was purchased. This driver could control both motors simultaneously and had several online resources. Unfortunately, the MD25 motor driver, a board specifically designed for use with the EMG-30 motors, was overlooked. This motor driver had quick-connect ports designed for the EMG-30, and it offered much more functionality immediately out of the box, namely a simple method for interacting with the EMG-30's onboard encoders. The TB6612FNG library, however, still provided useful functions for driving, braking, and reversing.

Eight functions were programmed for the robot's drivetrain movement. The robot had two driving wheels in the front and caster balls in the back, so drivetrain functions were modeled with those movement possibilities in mind. For example, the function for rotating to the right while pivoting forward was to drive with the left motor and brake with the right motor. These eight functions for rotating and turning enabled the robot to complete several complicated movement patterns.

### 2.1.6 Pushbutton

The last, most important, electronic component was the pushbutton. It was a latching pushbutton, meaning it was toggleable, and it had an internal red LED. This button was both the start and stop button. An emergency stop button was a requirement for the competition, so this button needed to stop all robot functions no matter where it was in the code.

This was accomplished through an event detect function. After initializing the button as a GPIO object using the RPi.GPIO library, an event detect function was created that triggered when the voltage fell across the pushbutton (when it was toggled from on to off) and a callback function was created which the program would run immediately afterwards. This callback function executed an 'os.kill' function which raised a keyboard interrupt exception and safely stopped the program wherever it was at that point in time. Lastly, a 'finally' block at the end of the program set all electronics to default conditions and reset all Raspberry Pi pins.

## 2.2 Gameplay Sequences

### 2.2.1 Revised Code Structure

The revised code structure was one large script. Seen in Figure 20 below and Figure 28 in Appendix A was the original code structure plan. Each colored section was a unique gameplay sequence that would be programmed as separate scripts. Each script would call another script as it ended, and scripts that needed to send information between one another would do so through multi-threading. This did not go as planned because multi-threading was exceptionally difficult to learn and implement, so every process was combined into one large script.

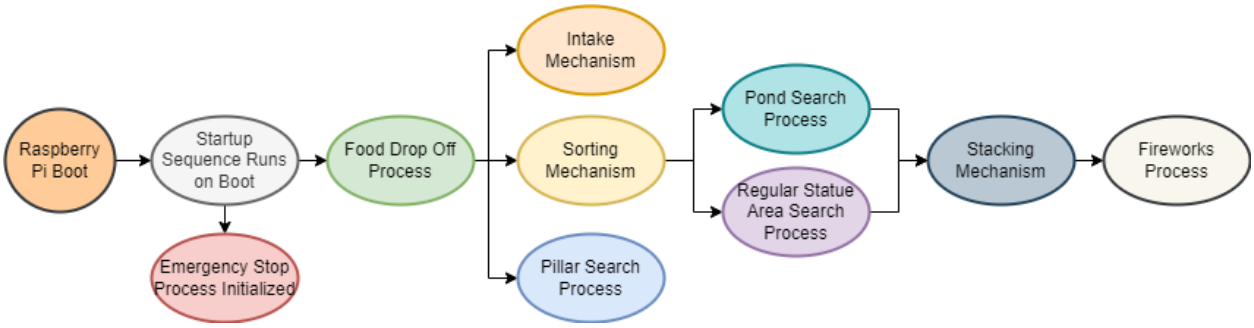


**Figure 20:** Initial Gameplay Sequence Flowchart (Simplified)

The next sections will describe the programming involved in each gameplay function and any changes that needed to be made when combining these scripts into one large game loop. Increased emphasis is placed on the startup and food delivery sequences because they were the only gameplay functions used at the competition.

### 2.2.2 Startup Sequence

Reference the flowchart shown in Figure 21 for a visual aid of the startup sequence. Once the start button was pressed, the program began taking input from the TCS34725 color sensor on the outside of the robot. This color sensor was positioned to face a red LED because the robot

earned more points if it auto started when this LED was turned on. During testing, it was found that the LED's green and blue values would sit around 80 or 90 when the LED was off but would plummet to 0 when the LED was on. The program would wait for the green value to dip below 20 (to be safe), before enabling the food delivery sequence.

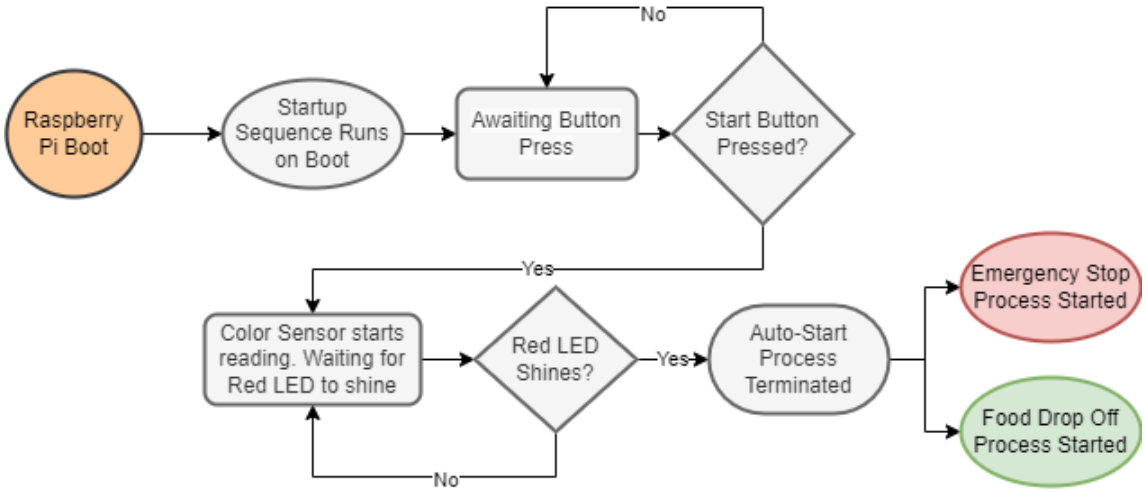


**Figure 21:** Startup Sequence Flowchart

### 2.2.3 Food Delivery Sequence

Reference the flowchart shown in Figure 22 for a visual aid of the food delivery sequence. The competition rules stated that the manatee (green) and alligator (red) aquariums would swap positions on different boards at the event. To combat this, the delivery system was programmed to be robust. The Pixy2.1 camera mounted on the front of the robot would find and track each aquarium. Functions were created for finding the closest aquarium based on horizontal distance from the center of the robot, centering the robot on the aquarium, and moving towards the aquarium while also tracking it and maintaining a constant heading.

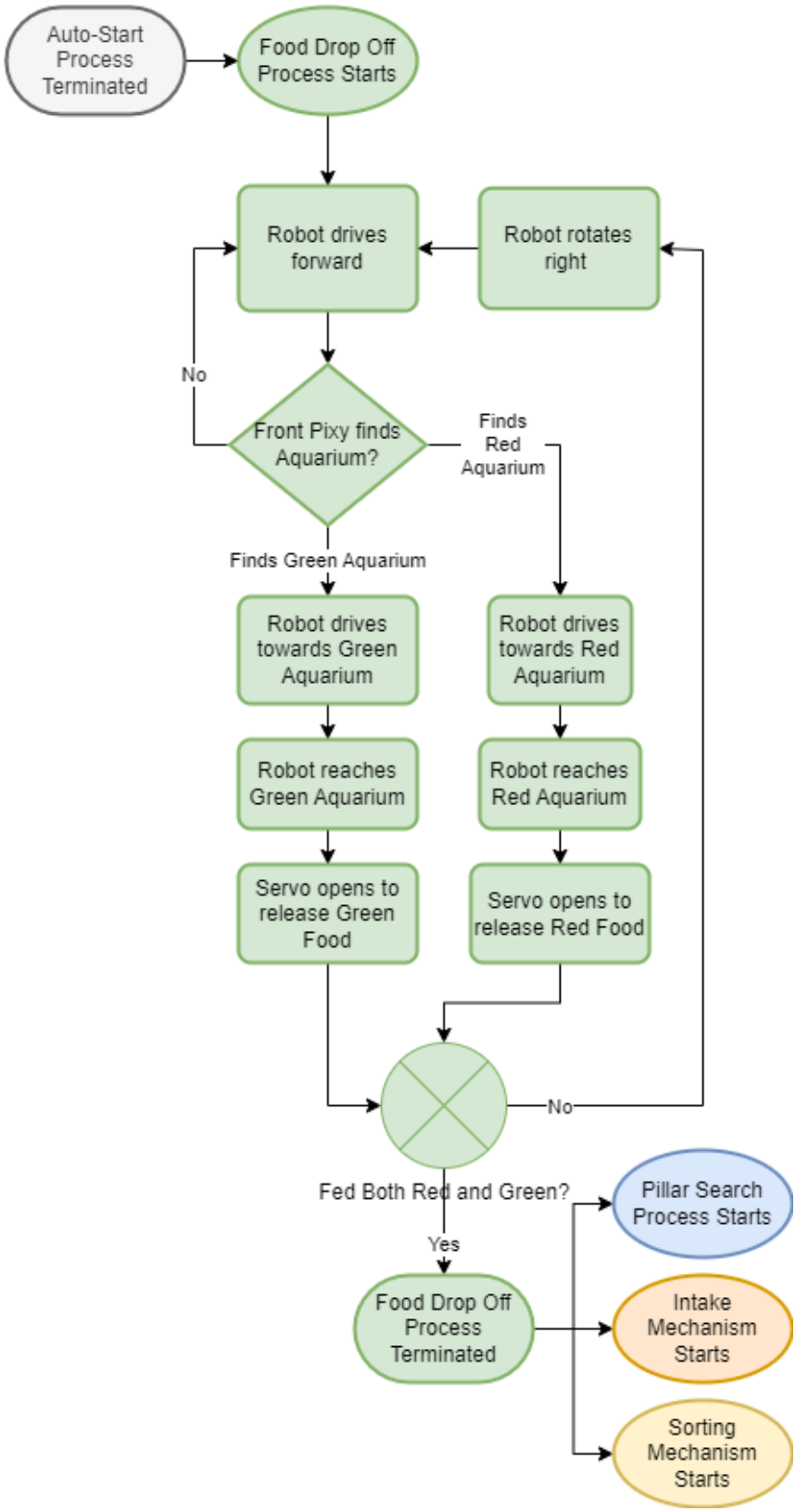


**Figure 22:** Food Delivery Flowchart

### 2.2.4 Intake Mechanism

Reference the flowchart shown in Figure 23 for a visual aid of the intake mechanism programming. The intake mechanism started by spinning the intake rollers. It then waited until the VL6180X time of flight sensor positioned in front of the sorting mechanism sensed an object by reading a range less than the calibrated limit. Meanwhile, the other TCS34725 color sensor, which was sitting opposite the time of flight sensor and pointed towards the same area was constantly storing color values. Once the VL6180X had sensed an object, the color sensor would sense the color of this object and then compare its red and green values to those it had stored previously. A calculation then ran that looked to see if the red or green values had increased by a certain percentage. If the red spiked and the green did not, then it was a red pillar. If the green spiked and the red did not, then it was a green pillar. If both spiked, it was a white pillar. This information would then be passed to the sorting cylinder and pillar search process.

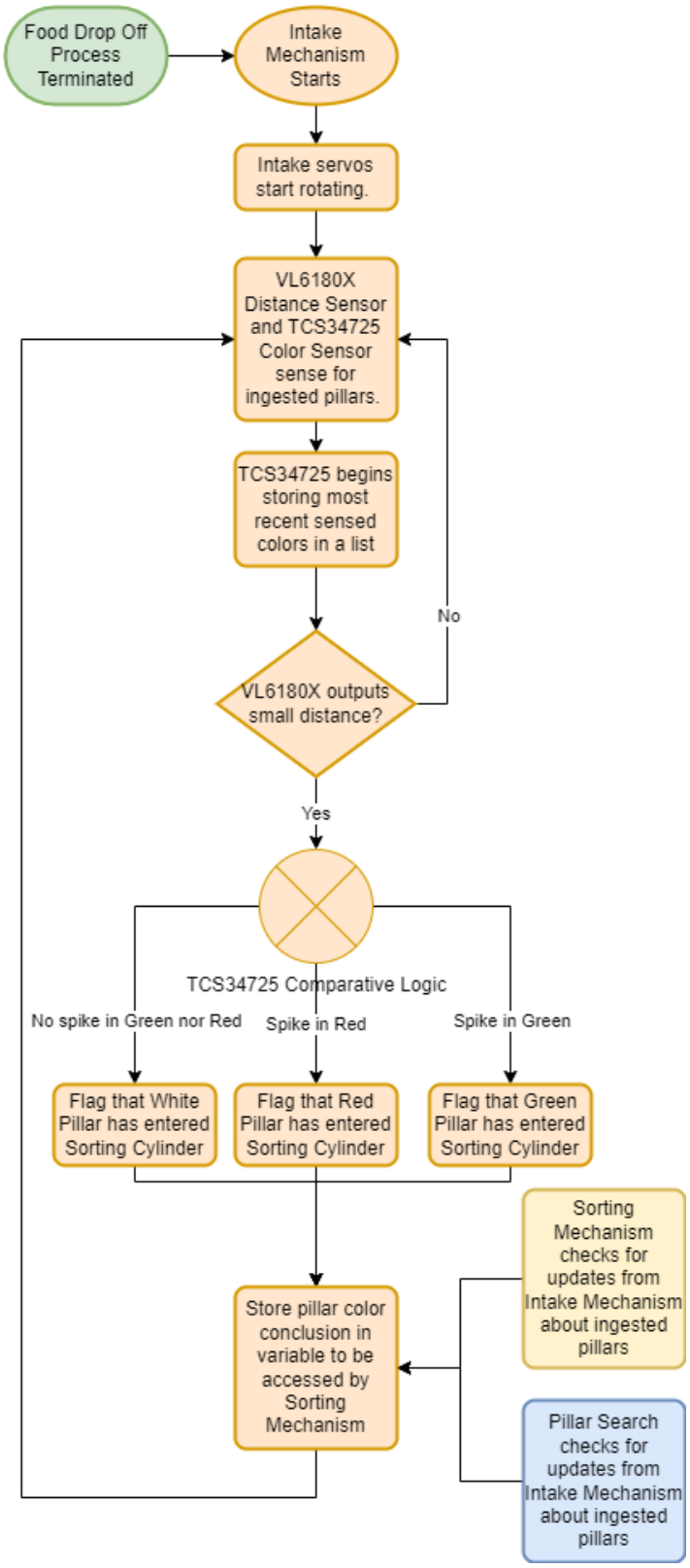


**Figure 23:** Intake Mechanism Flowchart

### 2.2.5 Sorting Mechanism

Reference the flowchart shown in Figure 24 for a visual aid of the sorting mechanism programming. The sorting cylinder functioned like a revolver with five equidistant holes that could be chambered at the intake, rotated to the top, and ejected into the stacking cylinder. To program this, each of the five holes was initialized as an object of the Hole class. This class had parameters for the top and bottom positions of the hole, if it was filled, what color pillar it had consumed, and if it was ready for intake. The top and bottom positions were hard coded values that could be passed to the hole object on initialization, and they represented the encoder position when the hole was aligned with the top exit hole or with the bottom entrance hole. By default, each hole was set to be empty and have a color of 'none'.

Several functions were also created for basic movement and logic in the sorting cylinder. Among these were functions for aligning a hole to its top and bottom positions, assigning a hole a new color, checking for empty holes, checking for holes ready for intake, and checking the sorting state to see if it was time to start stacking. Additionally, this process was compressed into a single function that could be called after each successful pillar intake.

After initializing the hole objects, Hole 1 was programmed to rotate to its bottom position and flag itself as 'ready for intake.' This ensured that the first pillar consumed would have a spot in the sorting mechanism. After a pillar was consumed, that pillar's color would be assigned to the hole with the 'ready for intake' flag. A check for empty holes would then commence, and an empty hole would rotate to its bottom position to be ready to intake another pillar.

Initially, the sorting cylinder checked if it could make a successful stack after every successful pillar intake, and it prioritized being able to make a three-pillar statue: white on bottom,

green in the middle, and red on top. It would continue searching for these pieces until it either found them or filled all five holes without consuming a red pillar in which case it would settle for a two-pillar stack. This became unnecessary when the sorting cylinder mechanism script was combined with all other gameplay function scripts because the robot could track the red, green, and white pillars and then immediately start the sorting procedure without needing to use this code to check.

The stacking process checked for holes filled with white pillars. It then aligned one of those holes with the top position and activated a servo at the top of the sorting cylinder to push the pillar out of the mechanism, down a ramp, and into the stacking cylinder. It repeated this process for a green pillar and then once again for a red pillar if it was making a three-pillar stack.

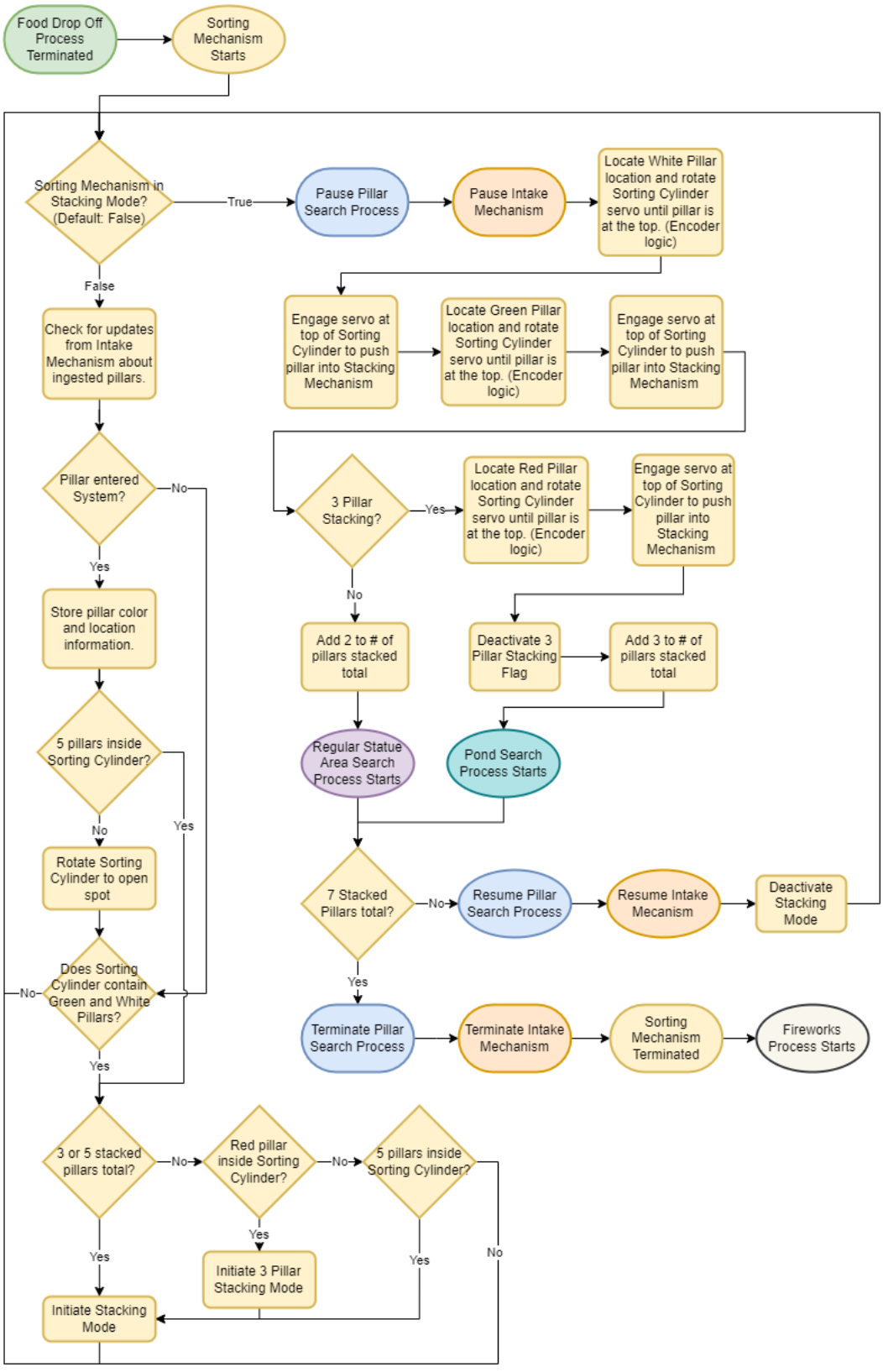


**Figure 24:** Sorting Mechanism Flowchart

### 2.2.6 Pillar Search Algorithm

Reference the flowchart shown in Figure 25 for a visual aid of the pillar search algorithm. This process relied entirely on the Pixy2.1 camera. To simplify the code, several functions were created for basic Pixy2.1 processes, such as checking for instances of a block color or checking to see if the Pixy2.1 was still tracking the object. Larger, more complicated logic and movement functions were also created for intractability later. These included calculating a block's horizontal distance from the Pixy2.1's center, finding the closest block to the Pixy2.1's center, tracking the farthest away object, centering the robot on a block, and tracking and moving towards a desired block. Each of these functions were common during the aquarium, pillar, and pond search processes.

For the pillar search process specifically, the robot would start by rotating until it saw a red pillar. It would then center itself on that pillar and begin moving towards it. The robot was programmed to maintain heading with the pillar during this process, adjusting itself whenever necessary. It would do this until it lost sight of the pillar or until the VL6180X time of flight sensor in the intake sensed the pillar entering the system. After sorting the red pillar, the robot would execute same tracking procedure for the remaining green and white pillars.

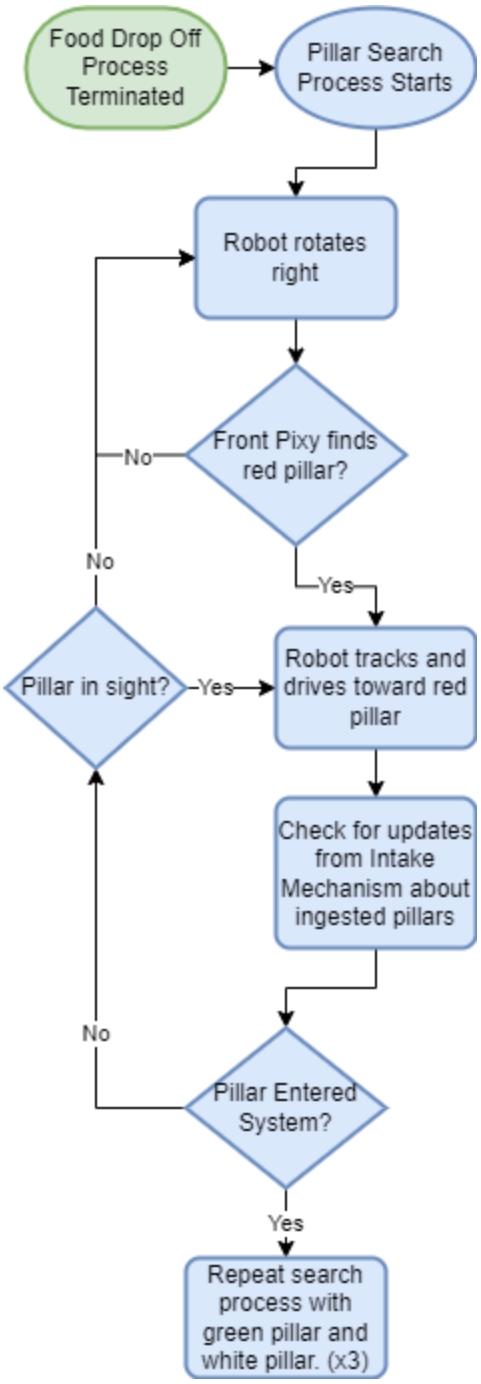


**Figure 25:** Pillar Search Algorithm Flowchart

### 2.2.7 Pond Search Algorithm

Reference the flowchart shown in Figure 26 for a visual aid of the pond search algorithm. This process was nearly identical to the pillar search process. The Pixy2.1 camera positioned at

the front of the robot would search for the blue pond while the robot rotated. Then the robot would center itself on the pond and drive towards it, maintaining constant heading. Once the robot lost sight of the blue pond, it would stop and initiate the stacking mechanism.

The original plan had a different stop condition for a more accurate stack placement. Once the Pixy2.1 camera positioned near the stacking mechanism at the back of the robot saw two blue objects, the robot would stop moving and initiate the stacking procedure. These two blue objects would be two edges of the pond on either side of the statue area, and it meant the stacking mechanism was positioned directly over the statue area in the middle of the pond. Difficulties with operating two Pixy2.1 cameras at the same time resulted in this process not being programmed.

The Pixy2.1 was also unable to recognize the firework switch and the regular statue areas, so these processes were not programmed. Instead, the team decided it was best to make a three-pillar stack and travel to the pond to deliver that stack.

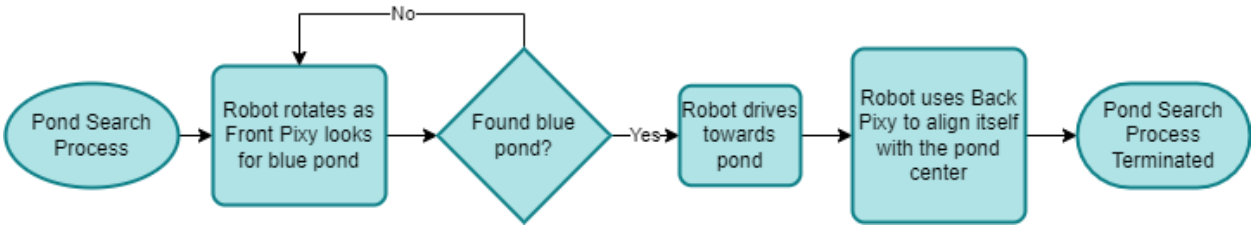


**Figure 26:** Pond Search Algorithm Flowchart

### 2.2.8 Pillar Stacking Mechanism

Reference the flowchart shown in Figure 27 for a visual aid of the pillar stacking mechanism. This mechanism would actuate a servo to open the stacking cylinder like a door, drive forward, and then close the stacking cylinder. All pillars would have already been loaded into the stacking mechanism by the sorting mechanism, and the stack would be left in place when the robot drove away.

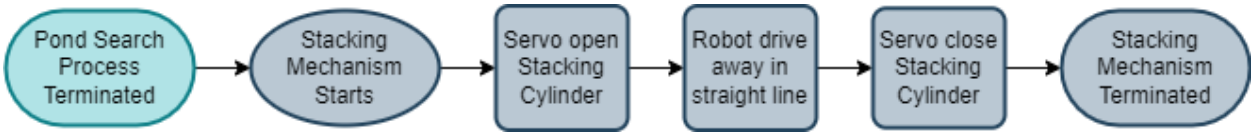


**Figure 27:** Pillar Stacking Mechanism Flowchart

# 3 Competition

## 3.1 Final Code Structure

The team encountered several issues in Florida. Because the robot was not assembled until three days before the competition, mechanical and electrical issues plagued it. The caster wheels interfered with the drivetrain, the intake mechanism was set too low to intake pillars, the improvised ramp from the intake to the sorting cylinder rubbed against the floor and further interfered with the drivetrain, the sorting cylinder was improperly fastened and too difficult to spin, the first servo driver was destroyed, and the encoder was returning unrealistic values dissimilar to those on the Arduino in prior testing. The team eventually decided to pour all its efforts into food delivery.

Immediately, simplifications were made to the food delivery system. Several teammates remembered that teams could pre-load food chips into their robots while setting up for runs. This meant the robot could be programmed to open a certain servo for each aquarium, and the user could load that servo with the correct food corresponding to the aquarium positions. It also meant it was possible to program a set path to each aquarium instead of relying on the Pixy2.1 camera for directions. At this point, the team was experiencing multiple difficulties with the Pixy2.1. It was inconsistent at detecting red objects, such as the alligator aquarium and red pillars, and it often lost track of objects when the robot cast a shadow over them. A solution to this issue would have been to add a large light on the front of the robot to give consistent readings for any room lighting. Already at the competition, the decision was made to refrain from using the Pixy2.1. Future teams using the Pixy2.1 camera should account for this lighting issue in the design.

Programming a set path for food delivery was simple with the drivetrain movement functions that were already programmed. Correct directions and timings were found from trial and error on the gameboard. One day before the competition, however, the robot began turning left instead of going straight. The omnidirectional steel ball transfers used for the caster wheels were not functioning as intended. When moved in certain directions, the casters locked up and provided considerable resistance to the drivetrain. Its effects were consistent for a short time, but they grew more inconsistent as testing continued throughout the day.

As a temporary fix for this issue, the code was modified to include a new drive function that would keep the robot moving straight. This function took a new parameter for a decreased right motor speed that could be changed in the setup based on how far the robot veered to the left in testing.

Once the food delivery sequence was working reliably, the robot was programmed to drive across the gameboard towards the recycling area. If it managed to push any game pieces into the recycling area, it would earn more points than other teams who focused solely on delivering food. The team refrained from additional testing to avoid creating further variation in the caster wheels' performance. Lastly, the Raspberry Pi was programmed to run the gameplay loop on boot, which was a competition requirement.

The final gameplay script can be found in Appendix B. This code includes functions for all gameplay sequences, but it only executes the startup sequence, food delivery sequence, and path to the recycling area, as the robot performed only those three in competition. Code and functions for untested gameplay sequences could contain syntax errors or redundancies.

## 3.2 Performance and Results

On competition day, only the startup sequence, food delivery sequence, and recycling sequence described previously were utilized. The competition consisted of three preliminary rounds followed by a single elimination tournament. Scores for each round were totaled, and the top eight teams advanced to the tournament.

The robot scored 52 points in the first preliminary round. It successfully auto started, delivered all six food pellets, and got stuck on a duck on its way to the recycling area. In the second preliminary round, it received 51 points. It successfully auto started, delivered five food pellets, and pushed three game pieces into the recycling area. In the third preliminary round, it received 45 points. It successfully auto started, delivered five food pellets, and got stuck on a duck on its way to the recycling area.

The team qualified for the single elimination tournament as the seventh seed with a total of 148 points after the first three qualifying rounds. Eventually, the team placed seventh at the competition after losing to the second seed robot from University of Kentucky in the first round of the tournament.

## Conclusion

The team was successful in designing, assembling, and programming a fully autonomous robot. Although far from perfect, the robot's consistency enabled it to receive seventh place at the IEEE SoutheastCon 2023 Hardware Competition.

This success required an in-depth understanding of each system and how they all needed to work together for the final product. From researching and selecting the best sensors for each application to programming each robot function, it required involvement in every step of the process. It came from learning useful career skills, such as programming a Raspberry Pi, interfacing with foreign electronics, and experiencing the unforgiving yet rewarding final integration process. Any individuals or teams interested in this project should compete in a local robotics competition. The skills learned from creating a robot are invaluable and best experienced firsthand.

## Works Cited


[1] Hopkins, Stephen, *IEEE SoutheastCon 2023 Hardware Competition: Hurricane Alley*, IEEE, 2023. https://drive.google.com/file/d/1bEue_yfS0Bxg47CCK_rJmApdyFGn3quW/view?usp=share_link.

[2] *TCS3200, TCS3210 PROGRAMMABLE COLOR LIGHT-TO-FREQUENCY CONVERTER*, Texas Advanced Optoelectronic Solutions Inc., 2009. https://www.mouser.com/catalog/specsheets/tcs3200-e11.pdf.

[3] "DFRobot Color Sensor TCS3200," *RobotShop*, 2022. https://www.robotshop.com/products/dfrobot-color-sensor-tcs3200.

[4] *Digital Red, Green and Blue Color Light Sensor with IR Blocking Filter: ISL29125*, Intersil, 2014. https://cdn.sparkfun.com/datasheets/Sensors/LightImaging/isl29125.pdf.

[5] "SparkFun RGB Light Sensor - ISL29125," *SparkFun*, 2022. https://www.sparkfun.com/products/12829.

[6] "RGB Color Sensor with IR filter and White LED - TCS34725," *Adafruit*, 2022. https://www.adafruit.com/product/1334.

[7] Shawn, "Types of Distance Sensors and How to Select One?," *SeeedStudio*, 2019. https://www.seeedstudio.com/blog/2019/12/23/distance-sensors-types-and-selection-guide/.

[8] Shawn, "All about Proximity Sensors: Which type to use?," *SeeedStudio*, 2019. https://www.seeedstudio.com/blog/2019/12/19/all-about-proximity-sensors-which-type-to-use/.

[9] “Grove - Time of Flight Distance Sensor(VL53L0X),” SeeedStudio, 2022. https://www.seeedstudio.com/Grove-Time-of-Flight-Distance-Sensor-VL53L0X.html.

[10] “Adafruit VL6180X Time of Flight Distance Ranging Sensor (VL6180) - STEMMA QT,” *Adafruit,* 2022. https://www.adafruit.com/product/3316.

[11] “SparkFun Qwiic ToF Imager - VL53L5CX,” *SparkFun,* 2022. https://www.sparkfun.com/products/18642.

[12] “VL53L7CX: Time-of-Flight 8x8 multizone ranging sensor with 90 degrees FoV,” *STMicroelectronics,* 2022. https://www.st.com/en/imaging-and-photonics-solutions/vl53l7cx.html.

[13] “SATEL-VL53L7CX,” *Mouser,* 2022. https://www.mouser.com/ProductDetail/STMicroelectronics/SATEL-VL53L7CX?qs=sGAEpiMZZMu3sxpa5v1qrg00HnlQ5dqx9%2FGW28WAHjA%3D.

[14] “Grove - Ultrasonic Distance Sensor,” *SeeedStudio,* 2022. https://www.seeedstudio.com/Grove-Ultrasonic-Distance-Sensor.html?utm_source=blog&utm_medium=blog.

[15] “Pixy2 Overview,” *PixyCam,* 2018. https://docs.pixycam.com/wiki/doku.php?id=wiki:v2:overview.

[16] LeGrand, Rich, “Introducing Pixy 2.1,” *Charmed Labs,* 2021. https://charmedlabs.com/default/introducing-pixy-2-1/.

[17] “Gravity: Huskylens - An Easy-to-use AI Camera | Vision Sensor,” *DFRobot,* 2022. https://www.dfrobot.com/product-1922.html.

[18] “SEN0305 HUSKYLENS AI Machine Vision Sensor,” *DFRobot,* https://wiki.dfrobot.com/HUSKYLENS_V1.0_SKU_SEN0305_SEN0336#target_42.

[19] "Adafruit TCA9548A 1-to-8 I2C Multiplexer Breakout," *Adafruit*, 2015. https://learn.adafruit.com/adafruit-tca9548a-1-to-8-i2c-multiplexer-breakout/overview.

[20] "Adafruit PCA9685 16-Channel Servo Driver," *Adafruit*, 2012. https://learn.adafruit.com/16-channel-pwm-servo-driver.

[21] "Analog Inputs for Raspberry Pi Using the MCP3008," *Adafruit*, 2012. https://learn.adafruit.com/reading-a-analog-in-and-controlling-audio-volume-with-the-raspberry-pi.

[22] "Best microSD Cards for Raspberry Pi 2023," *Tom's Hardware*, 2023. https://www.tomshardware.com/best-picks/raspberry-pi-microsd-cards.

[23] Staple, Danny. *Learn Robotics Programming: Second Edition*. Birmingham, Packt, 2021.

# Appendix A

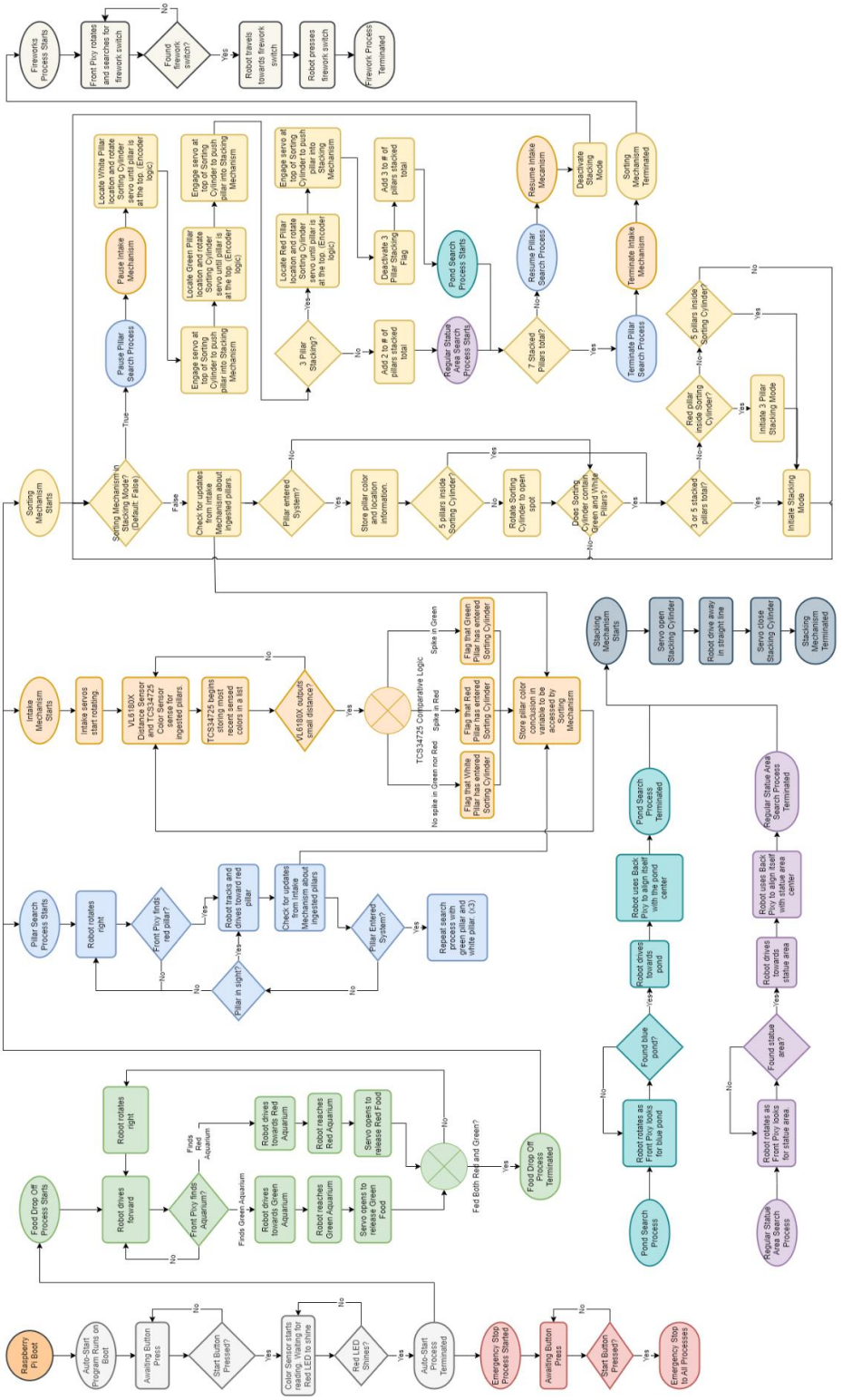


**Figure 28:** Initial Gameplay Sequence Flowchart

## Appendix B

Shown below is the final gameplay script. This code includes functions for all gameplay sequences, but it only executes the startup sequence, food delivery sequence, and path to the recycling area, as the robot performed only those three in competition. Code and functions for untested gameplay sequences could contain syntax errors or redundancies.

```
# Final Code Used for Competition
# Just Startup, Food Delivery, and Hail Mary to Recycling Area

#---Importing---#
import time
import board
import adafruit_tcs34725
import adafruit_tca9548a
import RPi.GPIO as GPIO
from digitalio import DigitalInOut, Direction, Pull
import os
#import threading
import signal
import adafruit_pca9685
import adafruit_servokit
import pixy
import ctypes
import adafruit_vl6180x
import adafruit_mcp3xxx.mcp3008 as MCP
from adafruit_mcp3xxx.analog_in import AnalogIn
import busio

# define a callback function for when the button press event happens
def button_callback(channel):
    print('Button pressed!')
    # Terminate the threads gracefully
    os.kill(os.getpid(), signal.SIGINT) # Sends SIGINT signal to the main
process


#---Classes---#
# Define Motor class
class Motor:
    in1 = ""
    in2 = ""
    pwm = ""
    standbyPin = ""

    #Defaults
    #hertz = 1000
```

```
    hertz = 20000 # This is the average that was recommended online for most
motors. The TB6612FNG has an upper limit of 100kHz
    reverse = False #Reverse flips the direction of the motor

    #Constructor
    def __init__(self, in1, in2, pwm, standbyPin, reverse):
        self.in1 = in1
        self.in2 = in2
        self.pwm = pwm
        self.standbyPin = standbyPin
        self.reverse = reverse

        GPIO.setup(in1,GPIO.OUT)
        GPIO.setup(in2,GPIO.OUT)
        GPIO.setup(pwm,GPIO.OUT)
        GPIO.setup(standbyPin,GPIO.OUT)
        GPIO.output(standbyPin,GPIO.HIGH) # Setting standby to high so the
motors will work
        self.p = GPIO.PWM(pwm, self.hertz)
        self.p.start(0)

    #Speed from -100 to 100
    def drive(self, speed):
        #Negative speed for reverse, positive for forward
        #If necessary use reverse parameter in constructor
        dutyCycle = speed
        if(speed < 0):
            dutyCycle = dutyCycle * -1

        if(self.reverse):
            speed = speed * -1

        if(speed > 0):
            GPIO.output(self.in1,GPIO.HIGH)
            GPIO.output(self.in2,GPIO.LOW)
        else:
            GPIO.output(self.in1,GPIO.LOW)
            GPIO.output(self.in2,GPIO.HIGH)
        self.p.ChangeDutyCycle(dutyCycle)

    def brake(self):
        self.p.ChangeDutyCycle(0)
        GPIO.output(self.in1,GPIO.HIGH)
        GPIO.output(self.in2,GPIO.HIGH)

    def standby(self, value):
        self.p.ChangeDutyCycle(0)
        GPIO.output(self.standbyPin,value)

    def __del__(self):
        GPIO.cleanup()

class Blocks (ctypes.Structure):
```

```
    _fields_ = [ ("m_signature", ctypes.c_uint),
    ("m_x", ctypes.c_uint),
    ("m_y", ctypes.c_uint),
    ("m_width", ctypes.c_uint),
    ("m_height", ctypes.c_uint),
    ("m_angle", ctypes.c_uint),
    ("m_index", ctypes.c_uint),
    ("m_age", ctypes.c_uint) ]

class Hole():
    def __init__(self, top, bottom, filled = False, color = 'none'):
        self.__top = top #This is the encoder value when this hole is
centered at the top
        self.__bottom = bottom #This is the encoder value when this hole is
centered at the bottom
        self.__filled = filled
        self.__color = color
        self.__ready_for_intake = False

    def get_state(self):
        return self.__filled
    def get_color(self):
        return self.__color
    def get_top(self):
        return self.__top
    def get_bottom(self):
        return self.__bottom
    def get_ready_for_intake(self):
        return self.__ready_for_intake
    def add_pillar(self, new_color):
        self.__filled = True
        self.__color = new_color
        self.__ready_for_intake = False
    def remove_pillar(self):
        self.__filled = False
        self.__color = 'none'
    def set_ready_for_intake(self, new_state):
        # Either True or False
        self.__ready_for_intake = new_state
    def set_color(self, new_color):
        self.__color = new_color
    def set_state(self, new_state):
        self.__state = new_state


#---Basic Functions for Pixy---#
def check_for_instances(blocks, desired_color_signature):
    # Gets all the instances of a certain color that the robot currently
sees.
    block_instances = []
    for block in blocks:
        if block.m_signature == desired_color_signature:
```

```
            # Adding all similar colored blocks to an instance list for later
use
            block_instances.append(block)
    return block_instances

def tracking_check(blocks, block_index):
    # Outputs True if the pixy still sees the block we're tracking. Outputs
False if it lost track of it
    for block in blocks:
        if block.m_index == block_index:
            return True
    return False

def calc_distance_from_center(blocks, block_index):
    # Return block's x_distance from the center of the pixy2 vision
    for block in blocks:
        if block.m_index == block_index:
            distance_from_center = 157 - block.m_x
            return distance_from_center

def closest_to_center(block_list):
    # Outputs the block index of the block closest to the center of the
screen.
    # Useful for future centering on an aquarium vs a random pillar that we
don't care about.
    # METHOD 1
    '''
    distances_from_center = []
    for block in block_list:
        # Getting the distances from center for each block and saving them
        distance_from_center = abs(157-block.m_x) # The x range of the Pixy2
grid is x=0 to x=315
        distances_from_center.append(distance_from_center)
    # Getting the distance value for the closest to the center
    closest_distance = min(distances_from_center)
    # Getting the list index of that distance. This will be the same list
index of that block in the block_list
    index_of_list = distances_from_center.index(closest_distance)
    # Getting the Pixy2 index of the block with the closest distance to the
center
    block_index = block_list[index_of_list].m_index
    actual_distance = 157-block_list[index_of_list].m_x
    '''
    # METHOD 2
    #'''
    closest_block_to_center = block_list[0]
    closest_distance = abs(157-block_list[0].m_x) # setting limit at the
first block's distance from center
    if len(block_list) > 1: # if we found more than one green
        for block in block_list[1:]: # Only looking at the blocks that aren't
the first block (cuz i already did the stuff for that)
            distance_from_center = abs(157-block.m_x) # computing the
distance from the center of the block to the center of the FOV
```

```
            if distance_from_center <= closest_distance: # if the center to
center distance is smaller than the previous smallest
                closest_distance = distance_from_center # set new smallest
center-to-center distance
                closest_block_to_center = block # save the block info cuz we
want to go towards this block
    block_index = closest_block_to_center.m_index # save index of the block,
because that's how we'll identify it in future frames
    actual_distance = 157 - closest_block_to_center.m_x
    #'''
    # Outputs the Pixy2 index of the closest block to the center and the
distance from the center
    return block_index, actual_distance

def track_farthest(blocks, block_instances, rmotor, lmotor, rotation_speed,
turn_speed, acceptable_offset, optimal_offset, vl6, vl6_detection_limit):
    # Find out what block is farthest away
    farthest_index = block_instances[0].m_index
    farthest_distance = block_instances[0].m_y
    for block in block_instances:
        if block.m_y > farthest_distance:
            farthest_distance = block.m_y
            farthest_index = block.m_index
    return farthest_index


#---Driving Functions---#
def keep_forward(rmotor, lmotor, forward_speed, keep_forward_speed):
    # For robots that turn left when going straight
    # keep_forward_speed will be less than forward_speed
    rmotor.drive(keep_forward_speed)
    lmotor.drive(forward_speed)

def turn_left_forward(rmotor, lmotor, forward_speed, turn_speed):
    # Turning left while moving forward. Turn speed will be slightly lower
than forward speed
    rmotor.drive(forward_speed)
    lmotor.drive(turn_speed)

def turn_right_forward(rmotor, lmotor, forward_speed, turn_speed):
    # Turning left while moving forward. Turn speed will be slightly lower
than forward speed
    rmotor.drive(turn_speed)
    lmotor.drive(forward_speed)

def rotate_left_forward(rmotor, lmotor, rotation_speed):
    # This is for turning left by running the right motor forward and doing
nothing with the left motor.
    # (i.e. we turn left by going forward)
    rmotor.drive(rotation_speed)
    lmotor.brake()

def rotate_left_backward(rmotor, lmotor, rotation_speed):
```

```
    # This is for turning left by running the left motor in reverse and doing
nothing with the right motor.
    # (i.e. we turn left by going backwards)
    rmotor.brake()
    lmotor.drive(-rotation_speed)

def rotate_left_even(rmotor, lmotor, rotation_speed):
    # This is for turning left by running the left motor in reverse and
running the right motor forward.
    # (i.e. we turn left by running each motor evenly but opposite. Hopefully
stay in the same spot)
    # I think it's best to run each at half of the speed, but we'll stick
with this for now.
    rmotor.drive(rotation_speed)
    lmotor.drive(-rotation_speed)

def rotate_right_forward(rmotor, lmotor, rotation_speed):
    # This is for turning right by running the left motor forwards and doing
nothing with the right motor.
    # (i.e. we turn right by going forwards)
    rmotor.brake()
    lmotor.drive(rotation_speed)

def rotate_right_backward(rmotor, lmotor, rotation_speed):
    # This is for turning right by running the right motor in reverse and
doing nothing with the left motor.
    # (i.e. we turn right by going backwards)
    rmotor.drive(-rotation_speed)
    lmotor.brake()

def rotate_right_even(rmotor, lmotor, rotation_speed):
    # This is for turning right by running the left motor forward and running
the right motor in reverse.
    # (i.e. we turn right by running each motor evenly but opposite.
Hopefully stay in the same spot)
    rmotor.drive(-rotation_speed)
    lmotor.drive(rotation_speed)


#---Robot Tracking Features---#
def forward_tracking(blocks, block_index, rmotor, lmotor, forward_speed,
turn_speed, acceptable_offset, optimal_offset, keep_forward_speed):
    # Track block index and head straight towards it.
    # Stops when pixy can no longer see the block with the block_index you're
looking for
    keep_forward(rmotor, lmotor, forward_speed, keep_forward_speed)
    #rmotor.drive(forward_speed)
    #lmotor.drive(forward_speed)
    distance_from_center = calc_distance_from_center(blocks, block_index)
    #while we haven't reached our goal
    approaching = True
    while approaching:
        # Check to see if the robot is too far offset from the center
```

```
        if abs(distance_from_center) > acceptable_offset: # if we're too far
off-center
            while abs(distance_from_center) > optimal_offset: # We want to
reset to a closer, more optimal offset
                if distance_from_center < 0: # if we need to turn left
                    turn_left_forward(rmotor, lmotor, speed, turn_speed)
                else: # if we need to turn right
                    turn_right_forward(rmotor, lmotor, speed, turn_speed)
                # Update the Pixy2 vision
                count = pixy.ccc_get_blocks (100, blocks)
                # Update the distance_from_center
                for block in blocks:
                    if block.m_index == block_index:
                        distance_from_center = 157 - block.m_x
                # Check to see if we lost track of the object
                if not tracking_check(blocks, block_index): # If pixy lost
track of it
                    # Stop moving when we lose track of the block. We might
be right over it
                    rmotor.brake()
                    lmotor.brake()
                    approaching = False
                    break

        # Check to see if we lost track of the object
        if not tracking_check(blocks, block_index): # If pixy lost track of
it
            # Stop moving when we lose track of the block. We might be right
over it
            rmotor.brake()
            lmotor.brake()
            approaching = False

        else:
            # Start driving forward once we are on track
            keep_forward(rmotor, lmotor, forward_speed, keep_forward_speed)
            #rmotor.drive(forward_speed)
            #lmotor.drive(forward_speed)

            # Update the Pixy2 vision
            count = pixy.ccc_get_blocks (100, blocks)

            # Update the distance_from_center
            for block in blocks:
                if block.m_index == block_index:
                    distance_from_center = 157 - block.m_x

def forward_tracking_pillar_searching(blocks, block_index, rmotor, lmotor,
forward_speed, turn_speed, acceptable_offset, optimal_offset, vl6,
vl6_detection_limit, keep_forward_speed):
    # Like forward tracking but for pillars
    # Track block index and head straight towards it.
```

```
    # Stops when pixy can no longer see the block or when the vl6180x senses
it has entered the sorting system
    keep_forward(rmotor, lmotor, forward_speed, keep_forward_speed)
    #rmotor.drive(forward_speed)
    #lmotor.drive(forward_speed)
    distance_from_center = calc_distance_from_center(blocks, block_index)
    #while we haven't reached our goal
    approaching = True
    while approaching:
        # Check to see if the robot is too far offset from the center
        if abs(distance_from_center) > acceptable_offset: # if we're too far
off-center
            while abs(distance_from_center) > optimal_offset: # We want to
reset to a closer, more optimal offset
                if distance_from_center < 0: # if we need to turn left
                    turn_left_forward(rmotor, lmotor, speed, turn_speed)
                else: # if we need to turn right
                    turn_right_forward(rmotor, lmotor, speed, turn_speed)
                # Update the Pixy2 vision
                count = pixy.ccc_get_blocks (100, blocks)
                # Update the distance_from_center
                for block in blocks:
                    if block.m_index == block_index:
                        distance_from_center = 157 - block.m_x
                # Check to see if we lost track of the object
                if (not tracking_check(blocks, block_index)) or (vl6.range <
vl6_detection_limit): # If pixy lost track of it or the block has been
consumed
                    # Stop moving when we lose track of the block. We might
be right over it
                    rmotor.brake()
                    lmotor.brake()
                    approaching = False
                    break

        # Check to see if we lost track of the object
        if (not tracking_check(blocks, block_index)) or (vl6.range <
vl6_detection_limit): # If pixy lost track of it or the block has been
consumed
            # Stop moving when we lose track of the block. We might be right
over it
            rmotor.brake()
            lmotor.brake()
            approaching = False
        else:
            # Start driving forward once we are on track
            keep_forward(rmotor, lmotor, forward_speed, keep_forward_speed)
            #rmotor.drive(forward_speed)
            #lmotor.drive(forward_speed)

            # Update the Pixy2 vision
            count = pixy.ccc_get_blocks (100, blocks)
```

```
            # Update the distance_from_center
            for block in blocks:
                if block.m_index == block_index:
                    distance_from_center = 157 - block.m_x

def center_robot(blocks, block_index, rmotor, lmotor, rotation_speed):
    centering = True
    while centering:
        # Get distance from center for the desired block
        distance_from_center = calc_distance_from_center(blocks, block_index)
        # Remember the pixy is upside down, so if distance from center is
negative, then the object is to the left of the center of the pixy
        if distance_from_center < 0: # need to turn left
            rotate_left_even(rmotor, lmotor, rotation_speed)
        elif distance_from_center > 0: # need to turn right
            rotate_right_even(rmotor, lmotor, rotation_speed)
        else:
            rmotor.brake()
            lmotor.brake()
            centering = False

        # Update the Pixy
        count = pixy.ccc_get_blocks (100, blocks)
        if count == 0:
            centering = False
            print('Error. Block Lost. Function aborted')


#---Functions for Food Mechanism---#
def second_aquarium_tracking(blocks, desired_color_signature, rmotor, lmotor,
rotation_speed, forward_speed, turn_speed, acceptable_offset, optimal_offset,
keep_forward_speed):

    ###### INPUT THE APPROPRIATE VALUE FOR THE SLEEP TIMER #########
    sleep_time = 1

    # Rotate right by putting the right motor in reverse
    rotate_right_backward(rmotor, lmotor, rotation_speed)
    time.sleep(sleep_time)
    # Start rotating in place. Eventually, we'll be searching for the
aquarium while still rotating
    rotate_right_even(rmotor, lmotor, rotation_speed)
    # Looking for the desired aquarium
    looking = True
    while looking:
        # Update the Pixy2 vision
        count = pixy.ccc_get_blocks (100, blocks)

        # Store all sighted blocks matching the desired color_signature
        block_instances = check_for_instances(blocks,
desired_color_signature)

        # If we see the color we're looking for
```

```
        if len(block_instances) > 0:
            # Get the block with the closest distance to center
            block_index, distance_from_center =
closest_to_center(block_instances)
            looking = False

    # Center on the aquarium once we've found it
    center_robot(blocks, block_index, rmotor, lmotor, rotation_speed)

    # Start moving towards the aquarium and keeping on track
    forward_tracking(blocks, block_index, rmotor, lmotor, forward_speed,
turn_speed, acceptable_offset, optimal_offset, keep_forward_speed)
    # We are at the aquarium by this point


#---Sorting Cylinder Functions---#
def align_bottom(encoder, hole, cylinder_servo, appropriate_throttle,
zero_throttle):
    # Align specified hole with the bottom position
    hole_bottom = hole.get_bottom()
    while int(encoder.value/100) != hole.get_bottom():
        #keeping it simple by only rotating in one direction for now
        cylinder_servo.throttle = appropriate_throttle
    #stop servo movement
    cylinder_servo.throttle = zero_throttle
    hole.set_ready_for_intake(True)

def rotate_to_empty_hole(holes, encoder, cylinder_servo,
appropriate_throttle, zero_throttle):
    # Rotate the sorting cylinder to a new, empty hole on bottom
    looking_for_empty_hole = True
    for hole in holes:
        #once we find the hole and align it, we don't want to keep aligning
others.
        if looking_for_empty_hole == True:
            #if hole is empty
            if hole.get_state() == False:
                #initiate hole movement towards bottom alignment.
                align_bottom(encoder, hole, cylinder_servo,
appropriate_throttle, zero_throttle)
                looking_for_empty_hole = False

def align_top(encoder, hole, cylinder_servo, appropriate_throttle,
zero_throttle):
    # Align specified hole with the top position
    hole_top = hole.get_top()
    while int(encoder.value/100) != hole.get_top():
        #keeping it simple by only rotating in one direction for now
        cylinder_servo.throttle = appropriate_throttle
    #stop servo movement
    cylinder_servo.throttle = zero_throttle
```

```
def rotate_to_top_hole(holes, desired_color, encoder, cylinder_servo,
appropriate_throttle, zero_throttle):
    # Rotate hole with desired color to the top
    checking_holes = True
    for hole in holes:
        if checking_holes: # If you're still looking for the hole with the
desired color
            if hole.get_color() == desired_color:
                # once you find the hole with the desired color, align it
with the top hole.
                align_top(encoder, hole, cylinder_servo,
appropriate_throttle, zero_throttle)
                # Calling this makes 'none' color and 'False' state. This is
in preparation for the next step which is pushing the cylinder out into the
stacking mechanism
                hole.remove_pillar()
                checking_holes = False

def assign_intake_color(holes, new_color):
    # Assign color to hole that just got filled with a pillar
    for hole in holes:
        # If hole is aligned with the bottom
        if hole.get_ready_for_intake() == True:
            # Assigning that hole the color. Also marking it as filled and
not ready for intake.
            hole.add_pillar(new_color)

def empty_holes_check(holes):
    # True if empty holes exist. False if every hole is filled.
    for hole in holes:
        # If a hole has a false state (not filled)
        if not hole.get_state():
            #break here if we found a hole that was empty
            return True
    # returns false if all holes are filled
    return False

def ready_for_intake_check(holes):
    # Checks all holes to see if any hole has the ready for intake flag. True
if so. False if not.
    for hole in holes:
        # if a hole is already set to be ready for intake
        if hole.get_ready_for_intake():
            return True
    return False

def sort_pillar(holes, pillars, stacked, encoder, cylinder_servo,
appropriate_throttle, zero_throttle):
    # Sorts new pillar. Should be called after a pillar has entered the
system.
    #---START OF SORTING CYLINDER STUFF---#
    #---Checking for (and Tracking) Consumed Pillars---#
```

```
    # Go through the normal loop of taking in pillars and tracking their
position.

    # Making sure that a hole is setup in the bottom position (in the case
where 5 were filled and none were moved into place before the sorting
sequence took place.
    if not ready_for_intake_check(holes): # If no hole is ready for intake
        rotate_to_empty_hole(holes, encoder, cylinder_servo,
appropriate_throttle, zero_throttle) # rotate to an empty hole

    # Once a pillar has been consumed
    # Assigning the pillar to the cylinder aligned with the bottom hole.
    assign_intake_color(holes, pillars[-1])

    #---Rotate to Empty Hole---#
    #finished taking in the pillar and now needs to rotate to next available
open space (if there is an empty hole)
    if empty_holes_check(holes):
        # If there is an empty hole, rotate to an empty hole
        rotate_to_empty_hole(holes, encoder, cylinder_servo,
appropriate_throttle, zero_throttle)

def check_sorting_state(holes, stacked):
    # Runs all of the logic to see if it's time to start stacking
    # Initially, the sorting cylinder will not be in stacking mode.
    two_pillar_stacking_mode = False
    three_pillar_stacking_mode = False
    #---Check to see if we need to start stacking---#
    # Creating a list of the current consumed colors for logic calculations.
    current_colors = []
    for hole in holes:
        current_colors.append(hole.get_color())

    # Checking to see if we have consumed a green and white pillar.
    if all(x in current_colors for x in ['green','white']): #if we have
consumed one green and one white
        # Run through other logic
        if stacked in [3,5]: # If we have stacked 3 or 5 pillars already
            two_pillar_stacking_mode = True
        else:
            if 'red' in current_colors: # if we have consumed a red pillar
                # white, green, and red means we're ready to make a 3 stack
                three_pillar_stacking_mode = True
            else:
                if not empty_holes_check(holes): # If all holes are filled
(i.e. if there are five pillars in the sorting cylinder)
                    # We don't have time to wait for a red. We're full, and
we need to make a 2 stack.
                    two_pillar_stacking_mode = True
    return two_pillar_stacking_mode, three_pillar_stacking_mode

def check_sorting_state_simple(holes):
    # Function for sorting cylinder demo
```

```
    # Checks to see if it's time to stack SIMPLE
    # SIMPLE: If white, green, and red then three stack. If all five are
filled and no red then two stack.
    two_pillar_stacking_mode = False
    three_pillar_stacking_mode = False
    #---Check to see if we need to start stacking---#
    # Creating a list of the current consumed colors for logic calculations.
    current_colors = []
    for hole in holes:
        current_colors.append(hole.get_color())

    # Checking to see if we have consumed a green and white pillar.
    if all(x in current_colors for x in ['green','white']): #if we have
consumed one green and one white
        # Run through other logic
        if 'red' in current_colors: # if we have consumed a red pillar
            three_pillar_stacking_mode = True
        else:
            if not empty_holes_check(holes): # If all holes are filled (i.e.
if there are five pillars in the sorting cylinder)
                # We don't have time to wait for a red. We're full, and we
need to make a 2 stack.
                two_pillar_stacking_mode = True

    return two_pillar_stacking_mode, three_pillar_stacking_mode

def sort_check_stack_pillar(holes, pillars, encoder, cylinder_servo,
appropriate_throttle, zero_throttle, pushing_servo, resting_angle,
pushing_angle, pushing_time):
    # Function for sorting cylinder demo
    # Same as sort_pillar, but we add in check_sorting_state and stacking (if
necessary) before we rotate to empty hole
    #---START OF SORTING CYLINDER STUFF---#
    #---Checking for (and Tracking) Consumed Pillars---#
    # Go through the normal loop of taking in pillars and tracking their
position.

    # Making sure that a hole is setup in the bottom position (in the case
where 5 were filled and none were moved into place before the sorting
sequence took place.
    if not ready_for_intake_check(holes): # If no hole is ready for intake
        rotate_to_empty_hole(holes, encoder, cylinder_servo,
appropriate_throttle, zero_throttle) # rotate to an empty hole

    # Once a pillar has been consumed
    # Assigning the pillar to the cylinder aligned with the bottom hole.
    assign_intake_color(holes, pillars[-1])

    # Checking to see if we need to make a stack
    two_pillar_stacking_mode, three_pillar_stacking_mode =
check_sorting_state_simple(holes)
    if two_pillar_stacking_mode: # If we need to make a two pillar stack
        print('Making a two pillar stack.')
```

```
        two_pillar_stacking(holes, encoder, cylinder_servo,
appropriate_throttle, zero_throttle, pushing_servo, resting_angle,
pushing_angle, pushing_time)
    if three_pillar_stacking_mode: # If we need to make a three pillar stack
        print('Making a three pillar stack.')
        three_pillar_stacking(holes, encoder, cylinder_servo,
appropriate_throttle, zero_throttle, pushing_servo, resting_angle,
pushing_angle, pushing_time)

    #---Rotate to Empty Hole---#
    #finished taking in the pillar and now needs to rotate to next available
open space (if there is an empty hole)
    if empty_holes_check(holes):
        # If there is an empty hole, rotate to an empty hole
        rotate_to_empty_hole(holes, encoder, cylinder_servo,
appropriate_throttle, zero_throttle)
    print('Ready for next pillar.')

#---Stacking Mechanism Functions---#
def push_pillar(pushing_servo, resting_angle, pushing_angle, pushing_time):
    # Used to push the pillar from the top hole into the stacking mechanism
    pushing_servo.angle = pushing_angle # Push
    time.sleep(pushing_time) # Wait
    pushing_servo.angle = resting_angle # Reset to default position

def two_pillar_stacking(holes, encoder, cylinder_servo, appropriate_throttle,
zero_throttle, pushing_servo, resting_angle, pushing_angle, pushing_time):
    #---Stacking the Consumed Pillars---#
    # Move white pillar to top
    print('Rotating white to top.')
    rotate_to_top_hole(holes, 'white', encoder, cylinder_servo,
appropriate_throttle, zero_throttle)

    # Engage pushing servo to push cylinder into the stacking mechanism
    print('Ejecting white.')
    push_pillar(pushing_servo, resting_angle, pushing_angle, pushing_time)

    # Move green pillar to top
    print('Rotating green to top.')
    rotate_to_top_hole(holes, 'green', encoder, cylinder_servo,
appropriate_throttle, zero_throttle)

    # Engage pushing servo to push cylinder into the stacking mechanism
    print('Ejecting green.')
    push_pillar(pushing_servo, resting_angle, pushing_angle, pushing_time)

def three_pillar_stacking(holes, encoder, cylinder_servo,
appropriate_throttle, zero_throttle, pushing_servo, resting_angle,
pushing_angle, pushing_time):
    #---Stacking the Consumed Pillars---#
    # Move white pillar to top
    print('Rotating white to top.')
```

```
    rotate_to_top_hole(holes, 'white', encoder, cylinder_servo,
appropriate_throttle, zero_throttle)

    # Engage pushing servo to push cylinder into the stacking mechanism
    print('Ejecting white.')
    push_pillar(pushing_servo, resting_angle, pushing_angle, pushing_time)

    # Move green pillar to top
    print('Rotating green to top.')
    rotate_to_top_hole(holes, 'green', encoder, cylinder_servo,
appropriate_throttle, zero_throttle)

    # Engage pushing servo to push cylinder into the stacking mechanism
    print('Ejecting green.')
    push_pillar(pushing_servo, resting_angle, pushing_angle, pushing_time)

    # Move red pillar to top
    print('Rotating red to top.')
    rotate_to_top_hole(holes, 'red', encoder, cylinder_servo,
appropriate_throttle, zero_throttle)

    # Engage pushing servo to push cylinder into the stacking mechanism
    print('Ejecting red.')
    push_pillar(pushing_servo, resting_angle, pushing_angle, pushing_time)


#---Pillar Search Functions---#
def search_for_red(blocks, desired_color_signature, rmotor, lmotor,
rotation_speed, turn_speed, acceptable_offset, optimal_offset, vl6,
vl6_detection_limit, keep_forward_speed):
    searching_for_red = True
    while searching_for_red:
        count = pixy.ccc_get_blocks(100, blocks)
        block_instances = check_for_instances(blocks,
desired_color_signature)
        # We want to wait until we see the first red block on the screen.
        while len(block_instances) == 0:
            count = pixy.ccc_get_blocks(100, blocks)
            block_instances = check_for_instances(blocks,
desired_color_signature)
            # Rotate right backwards
            rotate_right_backward(rmotor, lmotor, rotation_speed)
        # Stop moving
        rmotor.brake()
        lmotor.brake()
        block_index = block_instances[0].m_index
        # Center Robot on the first red block it sees
        center_robot(blocks, block_index, rmotor, lmotor, rotation_speed)
        # Travel towards the farthest block
        forward_tracking_pillar_searching(blocks, block_index, rmotor,
lmotor, forward_speed, turn_speed, acceptable_offset, optimal_offset, vl6,
vl6_detection_limit, keep_forward_speed)
        searching_for_red = False
```

```
def main():
    #---Channel Values---#
    # Channels for TCA Multiplexer
    pca_channel = 7
    tcs_startup_channel = 2
    v16_channel = 0
    tcs_channel = 3

    # Channels for PCA Servo Driver
    rservo_channel = 1
    lservo_channel = 2
    cylinder_servo_channel = 0
    pushing_servo_channel = 4
    stacking_servo_channel = 3
    back_food_servo_channel = 6
    front_food_servo_channel = 7

    # Pins for Button
    button_pin = 20 # GPIO 20
    button_pin_digital = board.D20 # GPIO 20
    button_led_pin = 25 # GPIO 25
    button_led_pin_digital = board.D25 # GPIO 25

    # Pin for Startup TCS34725 LED
    tcs_startup_led_pin_digital = board.D22 # GPIO 22

    # Pins for Intake TCS34725 LED
    tcs_led_pin_digital = board.D23 # GPIO 23

    # Channels for MCP3008 Analog to Digital Converter (ADC)
    encoder_channel = MCP.P0 # Channel 0 of the MCP
    cs_pin = board.D17 #GPIO 17


    #---Calibrated Color Signatures---#
    red_signature = 2 # set these based on what CCC signature we did for each
color
    green_signature = 1
    white_signature = 4
    blue_signature = 5
    green_pillar_signature = 3



    #---Initializing---#
    # Initialize I2C for the board
    i2c = board.I2C()

    # Initialize the TCA9548A Multiplexer
    tca = adafruit_tca9548a.TCA9548A(i2c)
```

```
    # Stuff for MCP3008
    # Create the SPI bus
    spi = busio.SPI(clock = board.SCK, MISO = board.MISO, MOSI = board.MOSI)
    # Create the cs (chip select)
    cs = DigitalInOut(cs_pin)
    # Create the mcp object
    mcp = MCP.MCP3008(spi, cs)
    # Create analog input channel on pin 0
    encoder = AnalogIn(mcp, encoder_channel)

    # Initialize the VL6180X. Connected to a channel on TCA Multiplexer
    vl6 = adafruit_vl6180x.VL6180X(tca[vl6_channel])

    # Initialize the TCS34725 Startup Color Sensor and other TCS34725
    tcs_startup = adafruit_tcs34725.TCS34725(tca[tcs_startup_channel])
    tcs = adafruit_tcs34725.TCS34725(tca[tcs_channel])

    # Initialize the PCA9685. Connected to a channel on TCA Multiplexer
    pca = adafruit_pca9685.PCA9685(tca[pca_channel])
    pca.frequency = 50 # set PWM frequency to 50Hz

    # Set PCA frequency and Initialize ServoKit library. Automatically
connects to PCA. Redirected to TCA channel that PCA is on
    kit = adafruit_servokit.ServoKit(channels = 16, i2c = tca[pca_channel])

    # Initializing servos. Connected to channels on PCA
    rservo = kit.continuous_servo[rservo_channel] # Initializing right and
left continuous rotation servos. Connected to channels on PCA
    lservo = kit.continuous_servo[lservo_channel] # R and L as if you were
playing as the robot
    cylinder_servo = kit.continuous_servo[cylinder_servo_channel] # for
sorting cylinder
    pushing_servo = kit.servo[pushing_servo_channel] # for servo that pushes
pillars into stacking mechanism
    stacking_servo = kit.servo[stacking_servo_channel] # for servo that opens
stacking mechanism
    back_food_servo = kit.servo[back_food_servo_channel] # for red food
    front_food_servo = kit.servo[front_food_servo_channel] # for green food

    # GPIO setup
    GPIO.setmode(GPIO.BCM)
    '''
    # Initializing the Pixy2
    pixy.init()
    pixy.change_prog("color_connected_components")
    '''
    # Startup TCS34725 initialization parameters
    tcs_startup_integration_time = 150
    tcs_startup_gain = 4
    tcs_startup.integration_time = tcs_startup_integration_time
    tcs_startup.gain = tcs_startup_gain

    # Intake TCS34725 initialization parameters
```

```
    tcs_integration_time = 150 # Optimal for fast sensing
    tcs_gain = 4
    tcs.integration_time = tcs_integration_time
    tcs.gain = tcs_gain

    # Initialize the button for Startup
    button = DigitalInOut(button_pin_digital)
    button.switch_to_input(pull=Pull.DOWN) # Button is set to be False when
not pressed

    # Initializing the led for the button
    button_led = DigitalInOut(button_led_pin_digital)
    button_led.direction = Direction.OUTPUT

    # Turning off the Startup TCS34725 LED because it allows the bright red
LED to be detected better
    tcs_startup_led = DigitalInOut(tcs_startup_led_pin_digital)
    tcs_startup_led.direction = Direction.OUTPUT
    tcs_startup_led.value = False

    # Turning off the Intake TCS34725 LED
    tcs_led = DigitalInOut(tcs_led_pin_digital)
    tcs_led.direction = Direction.OUTPUT
    tcs_led.value = False


    #---Sensor Calibration Limits---#
    # Set minimum distance value for pillar inside the system (mm)
    vl6_detection_limit = 40

    # Set multiple for color to spike (compared to previous) to be of
interest
    red_multiple = 1.5
    green_multiple = 1.5


    #---Servo Limits---#
    # Limits for left and right intake servos
    rservo_ccw_limit = 0.9
    #rservo_cw_limit = -0.9
    rservo_stop_limit = 0.1
    #lservo_ccw_limit = 0.9
    lservo_cw_limit = -0.9
    lservo_stop_limit = 0.1

    # Limits for sorting cylinder servo
    appropriate_throttle = 0.2 #set appropriate value for throttle for
cylinder servo
    zero_throttle = 0.1 # set appropriate value for zero throttle for
cylinder servo

    # Limits for pushing_servo
    resting_angle = 140 # set appropriate value for non-pushing angle
```

```
    pushing_angle = 180 # set appropriate value for pushing angle
    pushing_time = 2 # set appropriate value for waiting period between push
and return to resting position

    # Limits for stacking dropoff servo
    stacking_servo_closed_angle = 5 # set appropriate value for closed
position
    stacking_servo_open_angle = 90 # set appropriate value for open position

    # Limits for food servos
    back_food_servo_closed_angle = 50 # set appropriate value for initial
angle for red food servo
    back_food_servo_open_angle = 180 # set appropriate value for activated
angle for red food servo
    front_food_servo_closed_angle = 30 # set appropriate value for initial
angle for green food servo
    front_food_servo_open_angle = 180 # set appropriate value for activated
angle for green food servo


    #---Motor Limits---#
    # set appropriate values for motor speeds
    forward_speed = 80
    turn_speed = 60
    rotation_speed = 80
    aquarium_forward_speed = 80
    #keep_forward_speed = 65
    keep_forward_ratio = 0.75
    keep_forward_speed = int(keep_forward_ratio * forward_speed)
    print(keep_forward_speed)


    #---Motor Setup---#
    # Right and left as if you were the robot and seeing as it saw
    # Set pins for motors
    AIN1 = 5
    AIN2 = 6
    PWMA = 12
    STBY = 16
    BIN1 = 19
    BIN2 = 26
    PWMB = 13
    lmotor = Motor(AIN1, AIN2, PWMA, STBY, False) # right motor # WE WILL
NEED TO SET ONE OF THEM TO TRUE (for reverse)
    rmotor = Motor(BIN1, BIN2, PWMB, STBY, True) # left motor


    #---Timing Limits---#
    empty_sorting_entrance_time = 1 # How long to make the sensing pause
after a pillar has entered (sec)
    #pause_time = 0.05 # Sleep time in between each pillar check (sec)
    drive_away_time = 2 # time for driving away from stacked and placed
pillar
```

```
    '''
    #---Pixy Centering Limits---#
    acceptable_offset =  # the amt of x coordinates offset from center before
we need to recenter on the object we're tracking
    optimal_offset =  # the amt of x coords offset from center that we want
to recenter to when trying to recenter
    #if we keep recentering to the acceptable_offset, then we might just end
up centering and then going right back out cuz we're already on the edge of
the acceptable limit
    aquarium_offset =


    #---Measured values for the encoder positions---#
    hole1_top_value =
    hole1_bottom_value =
    hole2_top_value =
    hole2_bottom_value =
    hole3_top_value =
    hole3_bottom_value =
    hole4_top_value =
    hole4_bottom_value =
    hole5_top_value =
    hole5_bottom_value =

    #--------END OF MANUAL INPUT-----------#

    #---Creating hole objects---#
    hole1 = Hole(hole1_top_value, hole1_bottom_value)
    hole2 = Hole(hole2_top_value, hole2_bottom_value)
    hole3 = Hole(hole3_top_value, hole3_bottom_value)
    hole4 = Hole(hole4_top_value, hole4_bottom_value)
    hole5 = Hole(hole5_top_value, hole5_bottom_value)
    holes = [hole1,hole2,hole3,hole4,hole5]
    '''

    #---Setting General Variables and Lists for Later Use---#
    # For sorting mechanism logic
    stacked = 0

    # Creating lists for later use (Intake Mechanism)
    #recent_colors = []
    pillars = []
    '''
    # Creating Pixy2 List
    blocks = pixy.BlockArray(100)
    '''

    #---SET DEFAULT CONDITIONS---#
    # Set Motors to be still
    rmotor.brake()
    lmotor.brake()
```

```
    # Set servos to be at their resting/default positions
    rservo.throttle = rservo_stop_limit
    lservo.throttle = lservo_stop_limit
    cylinder_servo.throttle = zero_throttle
    pushing_servo.angle = resting_angle
    stacking_servo.angle = stacking_servo_closed_angle
    back_food_servo.angle = back_food_servo_closed_angle
    front_food_servo.angle = front_food_servo_closed_angle

    ## Rotate sorting cylinder so that hole1 is aligned with the bottom.
    #align_bottom(encoder, hole1, cylinder_servo, appropriate_throttle,
zero_throttle)


    #---START STARTUP CODE---#
    try:
        # Wait for start button to be pressed
        waiting = True
        while waiting:
            if button.value == True: # When the button is being pressed
                print('Button Pressed')
                waiting = False
                # Set up button as a GPIO button
                GPIO.setup(button_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
                time.sleep(1) # one second was a good sleep time during
testing
                # Add event handler for button press
                GPIO.add_event_detect(button_pin, GPIO.FALLING, callback =
button_callback, bouncetime = 300)
                button_led.value = True

        #---Starting Waiting for LED---#
        #red_list = [] # Making a list for adding color sensor values
        #difference_multiple = 1.5 # Red needs to be x times bigger than the
previous red
        waiting_for_led = True
        while waiting_for_led:
            color1_rgb = tcs_startup.color_rgb_bytes
            if color1_rgb[1] < 20: # if the red led is turned on
                print('LED ON')
                waiting_for_led = False

        #---Start food dropoff hardcoded in---#
        time_to_first_aquarium = 5
        time_to_back_up = 1.5
        time_to_rotate = 1.6
        time_to_second_aquarium = 3
        time_to_back_up2 = 1
        time_to_rotate2 = 1.5
        time_to_recycling = 9

        print('Driving towards aquarium 1')
        keep_forward(rmotor, lmotor, forward_speed, keep_forward_speed)
```

```
    time.sleep(time_to_first_aquarium)

    print('Braking')
    rmotor.brake()
    lmotor.brake()

    front_food_servo.angle = front_food_servo_open_angle

    print('Backing Up')
    keep_forward(rmotor, lmotor, -forward_speed, -keep_forward_speed)
    time.sleep(time_to_back_up)

    print('Rotating Right Back')
    rotate_right_forward(rmotor, lmotor, rotation_speed)
    time.sleep(time_to_rotate)

    print('Driving straight towards aquarium 2')
    keep_forward(rmotor, lmotor, forward_speed, keep_forward_speed)

    time.sleep(time_to_second_aquarium)

    print('Braking')
    rmotor.brake()
    lmotor.brake()

    back_food_servo.angle = back_food_servo_open_angle
    time.sleep(1)
    print('TA DA!')

    print('Rotating')
    keep_forward(rmotor, lmotor, -forward_speed, -keep_forward_speed)
    time.sleep(time_to_back_up2)

    rotate_right_forward(rmotor, lmotor, rotation_speed)
    time.sleep(time_to_rotate2)

    keep_forward(rmotor, lmotor, forward_speed, keep_forward_speed)
    time.sleep(time_to_recycling)

    rmotor.brake()
    lmotor.brake()


finally:
    #---RESET TO DEFAULT CONDITIONS---#
    # Set Motors to be still
    rmotor.brake()
    lmotor.brake()

    # Set servos to be at their resting/default positions
    rservo.throttle = rservo_stop_limit
    lservo.throttle = lservo_stop_limit
```

```
        cylinder_servo.throttle = zero_throttle
        pushing_servo.angle = resting_angle
        stacking_servo.angle = stacking_servo_closed_angle
        back_food_servo.angle = back_food_servo_closed_angle
        front_food_servo.angle = front_food_servo_closed_angle

        GPIO.cleanup()

main()
```