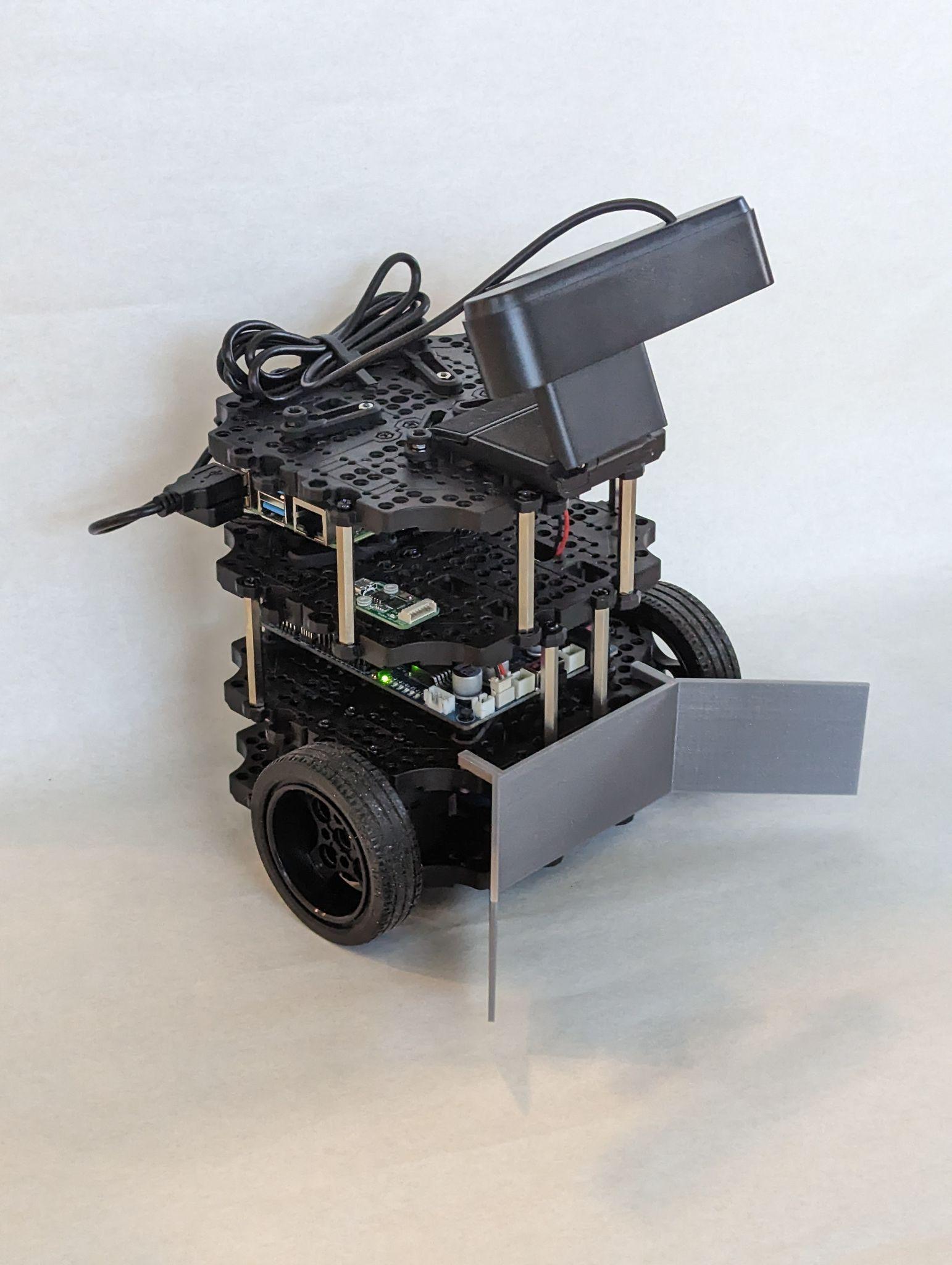
# NibblesBot: Final Report

By Ethen Carrell, Charles Moertle, and Calab Reeder



## Introduction

A common problem in robotics is locating objects and arranging them in some manner. In most cases, this is accomplished using an overhead camera that provides a “God’s eye” view, i.e., a view of an entire area. In this project, we are attempting a variation of this problem by removing the overhead camera and relying solely on a camera attached to the mobile robot that only provides a partial view of the area. In order to test our approach, our team’s goal is to use a TurtleBot3 Burger robot to rearrange several blocks into a specified order. To accomplish this task, we will assemble the mobile robot and modify it to suit the problem. The overall approach has the robot searching a designated area for blocks in a back and forth search pattern. As each block is found, the block will be moved to a staging area for further processing. After scanning the entirety of the area, the robot will identify each block in the staging area, then move the blocks to the desired order.

## Technology

### TurtleBot3 Burger

The TurtleBot3 Burger was chosen for our project due to our familiarity with TurtleBots from our work in The Constructsim. Our team gained experience with programming and troubleshooting the basics with a simulation using TurtleBot2models as we learned ROS from our previous robotics course. Also, the TurtleBot3 Burger was very simple in design and easy to put together. This allowed for us to quickly create the base model for our project so that we could begin to understand what modifications needed to be made before moving forward.

### ROS Noetic

ROS Noetic was the ROS distribution that our group was the most familiar with at the time of starting the project. This distribution of ROS was familiar to us so it was easier to learn new concepts and uses as we progressed in our project.

### Ubuntu 20.04 LTS

Ubuntu 20.04 LTS was our choice because it was required by ROS Noetic. The lifetime support provided by this Ubuntu distribution would also mean that the project would not need to change to a different Ubuntu distribution for later improvements and implementations of this project.

### Camera

The camera model that we used for the final design of NibblesBot was an Argmao 1080p resolution web camera with a standard field of view. We had originally used a 1080p Logitech camera but found that it was not able to use our computer vision algorithms well enough for our purposes due to issues with focusing a frame. This change was necessary due to how pivotal our computer vision algorithms were to our project. Any camera could be used for NibblesBot so long as it is able to take in enough details to properly use both Canny Edge Detection and ORB (Orientated FAST, Rotated BRIEF).

### Oracle Virtual Box versus Dual Boot

Originally, we planned to use Oracle Virtual Box, a virtual machine, to run the required Ubuntu 20.04 distro on our remote laptops. However, we encountered problems when attempting to send commands to the TurtleBot3 Burger robot. To send commands, we needed to update the ./bashrc file with IP addresses of the Raspberry Pi 4 and the remote desktop which we would be using to communicate with the robot. Unfortunately, we could not do this through the virtual machine because of unknown errors that prevented the bridged connection from working properly. After spending a few days attempting to debug the issue, we decided to switch to dual booting Ubuntu alongside our current operating system as an alternative to using a virtual machine. Thankfully, we managed to get the dual boot operational after some small issues, and were able to successfully send commands to the robot.

Although the virtual machine and dual boot were not difficult to learn, each technology presented its own set of problems. The dual boot installation had unique issues for each team member, some of us having conflicts between our graphics cards and the dual boot, and others where the BIOS seemed to resist all attempts at dual booting. Despite these challenges, we believe it was the correct decision switching over to using a dual boot, as it was faster and easier than continuing to debug the Oracle Virtual Box bridged connection issue. Additionally, by this point in time we had already spent several weeks building and installing the necessary software to run the TurtleBot3. We could not afford to waste more time debugging the software when we knew of another viable solution, even if it meant having to reinstall Ubuntu and ROS all over again.

## Design

### Hardware Design

The NibblesBot is a small mobile robot built on the open source TurtleBot3 Burger platform by ROBOTIS. The robot's dimensions are 138 x 178 x 192 (L x W x H mm) and is equipped with a Raspberry Pi 4B, an OpenCR board, an Argmao 1080p camera, a 3D printed cow catcher, two PID drive motors, and an 11.1V Li-Po battery giving an estimated 2 hours of operating time. In addition to the equipped sensors we are attaching a small usb camera for vision recognition. The Raspberry Pi will be installed with Ubuntu 20.04.05 LTS, ROS Noetic, TurtleBot3 ROS packages, and OpenCR board. The NibblesBot will have a linear top speed of 0.22 m/s and a max rotation speed of 2.8 rad/s.

The primary limitation of the Raspberry Pi is the processing power of the SoC built into the board. This was addressed by a computer, also running Ubuntu 20.04.05 LTS and ROS Noetic, that handles the logical operations of the NibblesBot remotely and transmits instructions back to the mobile frame. This is done using the computer as a ROS master node and the rostopics as a go between for data transmission. In this configuration the NibblesBot will only need to listen to the master node for movement instructions and transmit image data to the master node for image processing and logic processing. Figure 1 shows how the data is communicating between the pieces of hardware.

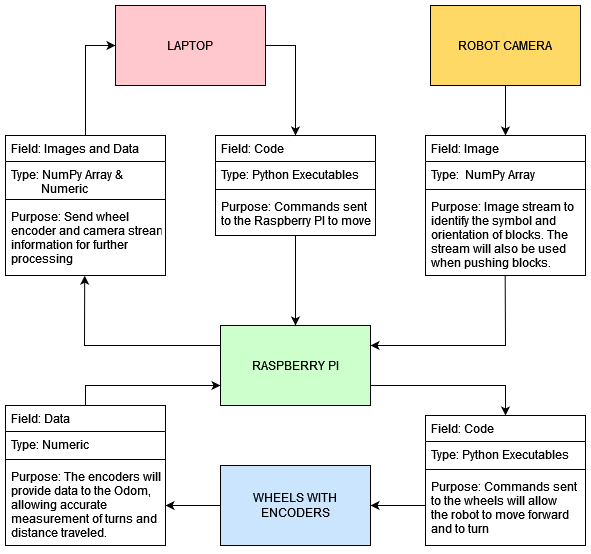


Figure 1: Message Diagram

Instead of using the base structure of the TurtleBot3 to push the blocks, a pusher was 3D printed for manipulating the blocks. The design of the pusher took into consideration the size of the waffle plates and the height of the plate stack for the core of the pusher. Further design considerations were made to accommodate easy attachment, block retention, protecting the robots drive wheels, and ease of printing. The front flat face of the pusher was designed to reach to the bottom of the base waffle plate from just above the 2nd waffle plate but not reach the ground. The width of the pusher’s front is made to be the same as the width of the waffle plate. The method for attaching the pusher is to print hooks that interface with the existing holes on the waffle plate while keeping mostly flush with the front of the robot. Given the flat, smooth surface of the pusher angled sections were placed on each side to help keep the blocks centered during turns. These angled sides were extended to shroud the wheels to keep from running over a block and causing slip that would disrupt odometry. The thickness of the pusher was made so that light impact and multiple blocks would not be enough to exceed the yield strength of the plastic, which in worst case scenarios is around the 17 MPa mark. Finally, the overall design was made so that printing would not need support structures making the printing process faster with less material used.

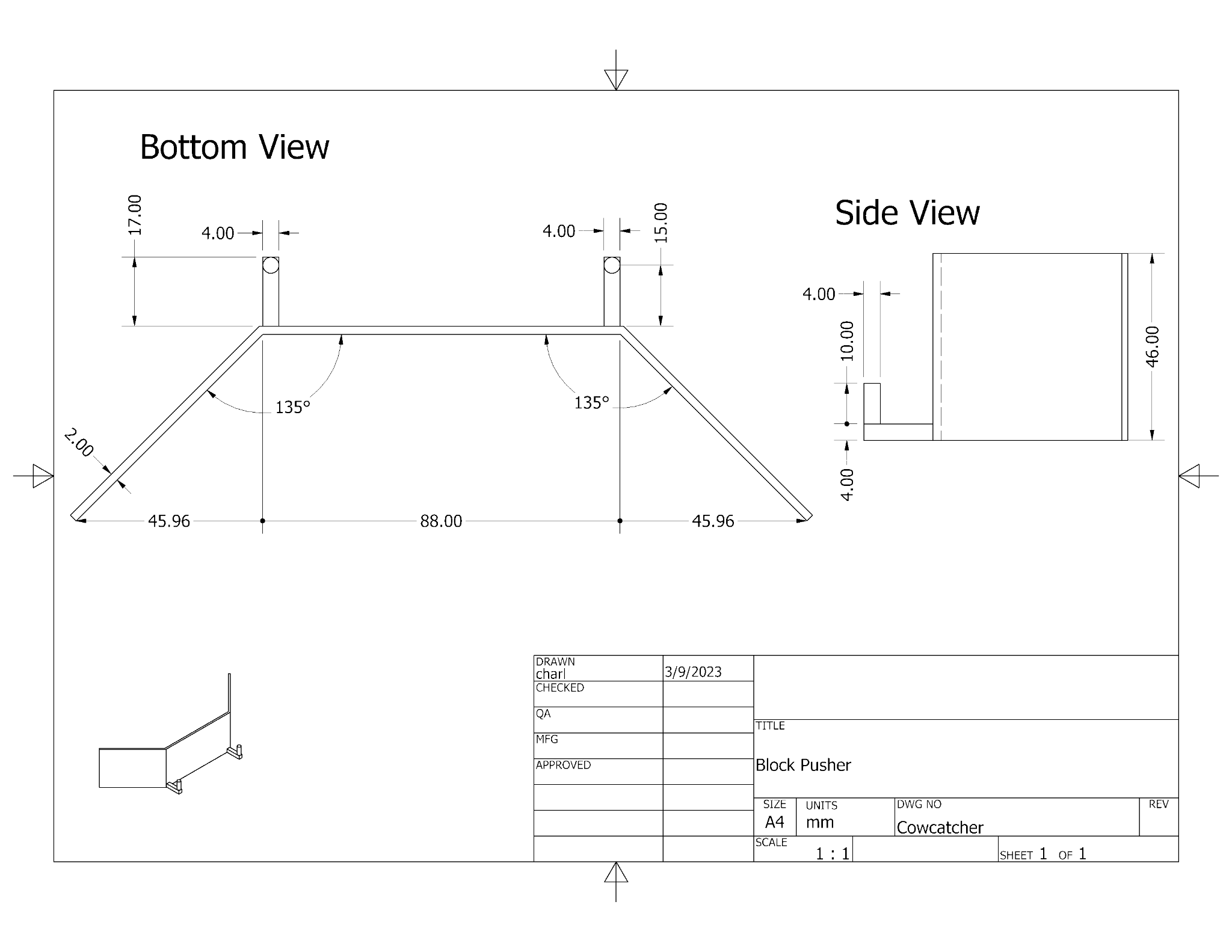


Figure 2: Cow Catcher Schematics

### Environment Design

To conduct the experiment, we prepared the environment by covering a hard and smooth plank of wood with a white sheet of paper. The purpose of the white sheet of paper was to enhance the contrast of the colored blocks, making them easier to detect with our various computer vision software. Additional considerations were made in an effort to prevent multiple blocks from being visible to the camera at the same time. To achieve this, we spaced the blocks at least 28 cm apart, which corresponds to the approximate width of the camera’s top edge. Finally, other than the blocks, no other obstacles were allowed to be in the environment to keep the problem strictly to the identification and retrieval of blocks.

### Algorithm Design

#### Search and Retrieval Algorithms

In order to begin the search for blocks in the environment, the starting position of the NibblesBot will be set as a “zero” point and will be used as a reference for the staging area that the blocks should be delivered to. The staging area is the first lane that the robot will travel with staging points set at 15 cm intervals. The intervals give enough space for the robot to retrieve the block from the staging area in the assembly step. After recording the start, the robot will then proceed to start searching in a back and forth pattern. The robot will move forward until it reaches the boundary of the environment, known through a series of image masks and by counting the number of black colored pixels in an image, before beginning to execute a series of turns. The first turn is performed such that the robot faces to the right of its previously set origin point, moving forward slightly, and then turning to face away from the wall. Once the robot has fully executed the turn in front of the wall, it will continue to move forward until it either finds a block, or reaches another wall.

When a block is found, the robot will cease its search orders, store the current location of the robot, and begin the process of grabbing the block. To grab the block, the robot uses the centroid of the block provided by the Canny algorithm. The robot will know which direction to turn in order to fully face the block by comparing the centroid x coordinate pixel to the value of the half-width of the frame. Using this information, the robot can adjust its heading and angular velocity according to the location of the centroid. By adjusting the angular velocity based on the difference between the centroid coordinate and the frame’s center, larger corrections can be made faster while smaller adjustments can be made with more control. As the robot moves forward to the block, the centroid y-coordinate is then compared to a threshold in the lower portion of the frame to indicate the collection of a block.

Once the block is confirmed to have been captured, NibblesBot will then begin returning to the staging area using its origin as reference. To take the block to the staging area, the robot needs to calculate the angle to the staging point by using the coordinate data from odometry and adjusting for the difference of the x coordinate of the target point. Further corrections need to be made based in the event that the block is found at an x coordinate smaller than the staging point or if the values are equal. These conditions determine if the angle given by the atan function needs to be reversed or not since trigonometry functions only return angles residing in two quadrants. Once the robot knows the target angle it begins to turn to face the staging point. During the turn the robot must maintain a forward motion preventing a turn in place to keep the block in the cow catcher. This means that the angle to the target area is constantly changing during the turn and needs to be recalculated throughout. Once the target angle and the current facing converge, the robot stops turning and drives to the staging point.

Once the robot has arrived at the staging area, the block will be positioned at the staging point and the robot will back away until the block is clear of the cow catcher. If there are more blocks to find and the robot has not searched the entire area, the robot will use the same atan function with the coordinates of the last position stored when the block was found to calculate the direction it needs to face. In this instance the robot can turn in place and does not need to recalculate the direction. Once the robot has face the proper direction the robot will return to the point with detection turned off. Lastly, when the robot reaches the last stored coordinate, the robot will face the last direction it was traveling while searching, enable detection, and resume the searching path. If all of the area has been searched, or if the correct number of blocks has been found, the robot can then begin sorting the blocks in the staging area.

#### Computer Vision Algorithms

One of the integral computer vision algorithms of our project is the ORB (Oriented FAST and Rotated BRIEF) feature detection algorithm. Additionally, the FAST in ORB stands for “Features from Accelerated Segment Test”, and BRIEF stands for “Binary Robust Independent Elementary Features”. ORB feature detection is an algorithm that is often used in object detection, using FAST as a means to extract feature points from some image and then using BRIEF to combine these feature points so they can locate an object. The features from FAST are calculated using pixel brightness, if a certain number of pixels surrounding some selected pixel (keypoint) are brighter or darker than the intensity threshold, then that pixel keypoint and location will be taken to be a ‘feature’ of the image. FAST in this case derives its name from the fact that, while comparing the pixel brightness surrounding some selected pixel, the algorithm first checks a pixel in each of the cardinal directions and the rest of the pixels will only be checked if at least three of the four cardinal pixels satisfy the intensity threshold we are looking for. After scanning the entire image for pixels that qualify as features, BRIEF will begin computing descriptors for each detected feature. Each descriptor will act as an ID, allowing the software to record any interesting information about a feature’s surrounding pixels. The gathered descriptor information is then converted into a binary feature vector in order to represent an object. At this point, the algorithm is done scanning and compiling information about what to detect in an image. The next step is to begin analyzing a second picture for the object we are looking for. In order to achieve more accurate results, the second image will need to be blurred as the BRIEF algorithm is very noise sensitive. The results will vary depending on the level of blur applied to the secondary image, so independent testing is needed to achieve an appropriate level of success to blurred image ratio. After blurring the image, the algorithm creates a similar binary feature vector as was created from the initial image using features gathered from the new image. The final step is to use a matching algorithm to compare the features and descriptors of both images in order to find the object from the first image in the second image. The matching algorithm we plan to use in our version of ORB is a brute force matcher that will compare each pair of descriptors for the object we are looking for, before drawing a square around the desired object.

The second integral computer vision algorithm is Canny edge detection. Canny works through several stages, the first of which involves reducing the noise in an image. In order to do this, Canny applies a gaussian filter to the image which performs a weighted average of the surrounding pixels to subsets of the larger image and thus blurring the image. Next, it will find the intensity gradient in order to begin a process called “non-maximum” suppression which removes unwanted pixels that may not constitute an edge. It does this by checking every pixel to see if it is a local maximum in regards to the direction of the gradient. The final step to using Canny involves hysteresis thresholding, which is where we determine the true edges in an image. As in its name, thresholding, this process involves a minimum and maximum threshold which will need to be adjusted until the appropriate edges are detected. An example of Canny can be seen below in figure 3.

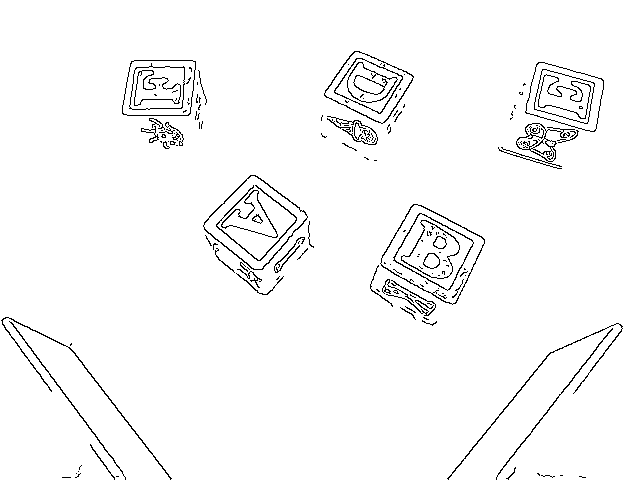


Figure 3: Canny edge detection

In our project, we use ORB, Canny Edge Detection, and color masking in tandem in order to find and identify the blocks in the environment. We first use Canny when actively searching for blocks, a process that involves constantly blurring an image, making it grayscale, and then passing it through Canny to get the edges in the image. After that, we also erode the image to further filter out noise, and then dilate the edges detected through Canny. The final step is to form contours from the edges and determine if the area inside the contours is large enough to contain a block. If the area is indeed large enough to contain a block, we draw the contours onto the main image frame and also draw in the centroid of the contours using the shape’s corners. The robot can then use the location of the centroid inside of the Canny detected contours to locate and grab a block in the area.

Before the contours can be used in ORB however, several training sets must be made to compare to the testing images (images taken directly from the robot’s camera). In total, three training sets will need to be created, each pertaining to a particular color that we will mask a testing image for (red, green, and blue). We do this in order to increase the accuracy of ORB, as there will be fewer images to compare the testing image to and so there is less of a chance of misidentifying the letter, and to increase the efficiency of the algorithm. As it stands, our current implementation of ORB will compare every single training image in a dataset to the testing image, and the training image that shares the most features is determined to be the letter in the testing image. While it works, a major drawback of this method is that it is incredibly resource intensive, and can take several seconds to fully process on a slower system. In order to partially alleviate this issue, we can mask for colors in an image, and split the workload into thirds. For example, now each red colored block will only be compared to other red colored blocks instead of the entire alphabet.

Once the datasets are created, we can use the contours discovered using Canny to dynamically crop images from the main frame and run them through the color masking and ORB algorithms. Dynamic cropping in this instance refers to cropping each successive image received from the robot’s camera so that the block we are focused on is always the only one that appears in the robot’s view. Using the cropped image, we can then use color masking to determine the appropriate dataset to use, and then use that dataset along with the cropped image to identify the correct letter on the block using ORB. This information is then stored inside a dictionary containing the letter of the block as a key and its value being the centroid of the contour area that was analyzed. NibblesBot can then proceed to use this information to sort the blocks in the staging area in their specified order. An example of ORB and Canny with contours is shown below in figure #.

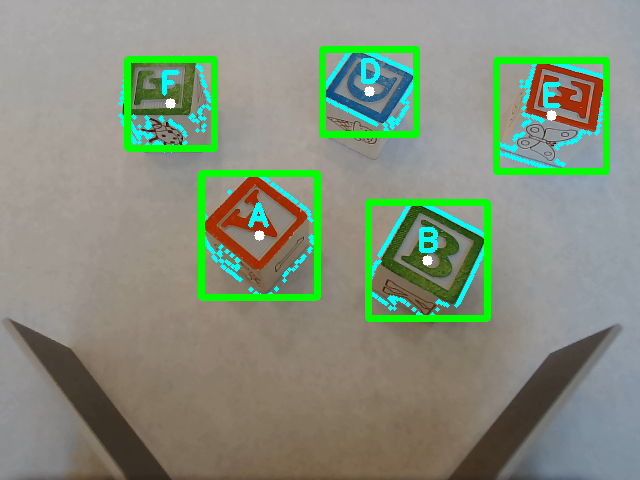


Figure 4: ORB working with Canny

## Deploying Software

### PC Setup

After making sure the computer meets the required specifications:

1. Download the Ubuntu 20.04 LTS desktop image
2. Create installation media (USB or CD)
3. Enter the systems BIOS and set the installation media type for the boot device
4. Restart the machine
5. Follow steps to install the OS
6. Restart the machine

Note: Driver issues may occur with certain hardware and may require further troubleshooting.

### Raspberry Pi Setup

1. Using an internet connected computer and a web browser of your choice go to <https://www.raspberrypi.org/software/> and download the Raspberry Pi Imager
2. Connect the microSD card to the computer and run the Imager software
3. In the Operating System selection choose “Other general-purpose OS” > Ubuntu > “Ubuntu Server 20.04.5 LTS (64-bit)”
4. In the Storage selection choose the microSD card
5. Start the write process and connect the Raspberry Pi to the internet by using the onboard Ethernet port
6. After the Imager is complete eject the microSD from the computer, insert it into the Raspberry Pi, and start the Raspberry Pi
7. When the system has fully started create a login and password
8. Log into the system and run the command:

$ sudo apt update && sudo apt upgrade

1. Install SLiM using:

$ sudo apt install slim

1. Install the Ubuntu Desktop using:

$ sudo apt install ubuntu-desktop

1. When prompted to choose a desktop manager the options will include “GDM3” and SLiM. Choose SLiM
2. Reboot the system

After the system reboots it can be disconnected from the Ethernet and connected to WiFi

### OpenCR Setup

1. On the Raspberry Pi open a terminal
2. Run the following commands:

$ sudo dpkg --add-architecture armhf

$ sudo apt-get update

$ sudo apt-get install libc6:armhf

1. Open ~/.bashrc in a text editor of your choice and add the following lines then save, exit, and source the file:

export OPENCR\_PORT=/dev/ttyACM0

export OPENCR\_MODEL=burger\_noetic

1. Download the firmware and loader using:

$ wget https://github.com/ROBOTIS-GIT/OpenCR-Binaries/raw/master/turtlebot3/ROS1/latest/opencr\_update.tar.bz2

$ tar -xvf opencr\_update.tar.bz2

1. Run the update using:

$ cd ./opencr\_update

$ ./update.sh $OPENCR\_PORT $OPENCR\_MODEL.opencr

### Installing ROS Noetic

#### On computer and Raspberry Pi:

1. In the Software & Updates application under the Ubuntu Software tab enable the “universe”, “restricted”, and “multiverse” settings.
2. Open a terminal and run the following command:

$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb\_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

1. Update the Debian package index using sudo apt update

#### On the computer:

1. Install the desktop version of ROS using:

$ sudo apt install ros-noetic-desktop

1. Set the ROS environment source script to run every time a terminal opens:

$ echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc

$ source ~/.bashrc

1. Install rosdep:

$ sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator python3-wstool build-essential

$ sudo rosdep init

$ rosdep update

1. Open the ~/.bashrc file in a text editor of your choice and append the following:

export ROS\_MASTER\_URI=http://{ip\_of\_pc}:11311

export ROS\_HOSTNAME={ip\_of\_pc}

1. Source the ~/.bashrc file

#### On the Raspberry Pi:

4. Install the base version of ROS using:

$ sudo apt install ros-noetic-ros-base

5. Set the ROS environment source script to run every time a terminal opens:

$ echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc

$ source ~/.bashrc

6. Install rosdep:

$ sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator python3-wstool build-essential

$ sudo rosdep init

$ rosdep update

7. Install dependent ROS packages:

$ sudo apt-get install ros-noetic-joy ros-noetic-teleop-twist-joy \

ros-noetic-teleop-twist-keyboard ros-noetic-laser-proc \

ros-noetic-rgbd-launch ros-noetic-rosserial-arduino \

ros-noetic-rosserial-python ros-noetic-rosserial-client \

ros-noetic-rosserial-msgs ros-noetic-amcl ros-noetic-map-server \

ros-noetic-move-base ros-noetic-urdf ros-noetic-xacro \

ros-noetic-compressed-image-transport ros-noetic-rqt\* ros-noetic-rviz \

ros-noetic-gmapping ros-noetic-navigation ros-noetic-interactive-markers

8. Install the TurtleBot3:

$ sudo apt install ros-noetic-dynamixel-sdk

$ sudo apt install ros-noetic-turtlebot3-msgs

$ sudo apt install ros-noetic-turtlebot3

1. Install the LiDAR driver using:

$ sudo apt update

$ sudo apt install libudev-dev

$ cd ~/catkin\_ws/src

$ git clone -b develop https://github.com/ROBOTIS-GIT/ld08\_driver.git

$ cd ~/catkin\_ws/src/turtlebot3 && git pull

$ rm -r turtlebot3\_description/ turtlebot3\_teleop/ turtlebot3\_navigation/ turtlebot3\_slam/ turtlebot3\_example/

$ cd ~/catkin\_ws && catkin\_make

1. Open the ~/.bashrc file in a text editor and add the following lines at the end:

export LDS\_MODEL=LDS-02

export TURTLEBOT3\_MODEL=${burger}

export ROS\_MASTER\_URI=http://{IP\_ADDRESS\_OF\_REMOTE\_PC}:11311

export ROS\_HOSTNAME={IP\_ADDRESS\_OF\_RASPBERRY\_PI\_3}

1. Source the ~/.bashrc file

#### Running the System

Running the system will require a stable internet connection.

1. Connect the battery to the TurtleBot
2. Connect the Raspberry Pi to a monitor, mouse, and keyboard
3. Power on the OpenCR board using the switch above the cow catcher
4. Log into the Raspberry Pi and open a terminal
5. Run the command: $ sudo ifconfig wlan0
6. Record the ipv4 address
7. Open the ~/.bashrc file in a text editor
8. Add the recorded ipv4 address to the ROS\_HOSTNAME line shown in step 10 of the Raspberry Pi installation instructions
9. On the host computer repeat steps 4-8 while also changing the the ROS\_MASTER\_URI line ip address to match the computers listed ipv4
10. On the Raspberry Pi add the computer’s ipv4 address to the ROS\_MASTER\_URI line
11. Save both files and source them
12. Disconnect the TurtleBot from the peripherals and place blocks in the area to be searched
13. On the host computer ssh to the turtlebot using:

$ ssh {turtlebot\_user\_name}@{turtlebot\_ipv4\_address}

1. On the computer’s terminal run roscore
2. On the ssh terminal run the command:

$ roslaunch turtlebot3\_bringup turtlebot3\_robot.launch

1. Open another ssh terminal and run the command:

$ rosrun nibbles\_test robot\_007\_camera\_stream\_test.py

1. Once in loop is printing on the host computer open a new tab and run:

$ rosrun computer laptop\_003\_object\_coord\_test.py

1. Once “Area” is printing on screen run:

$ rosrun computer laptop\_009\_retriever\_v2.py

To edit the search area or number of blocks open a text editor and edit the blocks\_total, area\_size\_x, and area\_size\_y parameters.

## Known Bugs

The first major issue we have faced is the non-uniform lighting conditions of our testing environment. These conditions have greatly affected the accuracy of our computer vision algorithms, causing misidentification of objects and even failing to detect them under extreme changes in lighting. To address this issue, we would need to establish an isolated testing area that is free from outside sources of light such as sunlight. This will allow us to have complete control over the amount and direction of lighting in the experiment through the use of spotlights and other forms of illumination. By doing so, we hope to improve the accuracy of our current computer vision system, and also allow for more effective identification of the blocks for when we train machine learning models in place of our current computer vision, this will be discussed in the following paragraphs.

Another issue we have had regarding our computer vision algorithms was the tendency of Canny Edge Detection to identify objects outside of the targeted blocks. Some unwanted objects that have been identified include the cow catcher mounted on the front of the robot, dirt and other particles that fall onto the testing area, and finally the black tape border is also detected by Canny. A potential solution to this issue would be to train a machine learning model to more accurately locate the blocks. A model that fits this problem is called Haar cascade, an object detection model that would allow us to solely detect the blocks in a frame and completely ignore the cow catcher and any other objects that are inside the test area. This would be possible by supplying the model with a multitude of positive and negative images, positive images meaning images that pertain to the objects we wish to detect, and negative images being things we do not wish to detect. So in our case we would pass in a large selection of images of the blocks for positive images, and for negative images we could supply the model with pictures of the cow catcher, the white background, and the black tape border.

The final issue that needs to be accounted for is the inaccuracy and lack of efficiency surrounding the use of ORB and its constant masking of the blocks. While these algorithms can identify the correct block more often than not, there is still a high degree of uncertainty and probability that the wrong block may be identified. This along with the high cost of running the algorithms even once lead to a need for a better solution. Similar to with Canny edge detection, a better solution in this instance would require the training of a machine learning model. The model in question would be PyTesseract, a python wrapper for the Tesseract OCR which focuses on character recognition in images. Training a PyTesseract font for the blocks would remove the need for our mask and ORB combination which we currently use, and would be far more accurate as we could supply the model training process with blocks under different lighting conditions. Once we have trained a new font for PyTesseract based on the blocks we have, we could then combine our two machine learning algorithms by using Haar cascade to find and crop objects out of the main frame, and then run that cropped image through PyTesseract to identify what letter it is.

## Future Work

There are several features that still need to be implemented to the current robot design. The first feature is the use of the camera frame positioned inside the cow catcher. This frame is to be used to determine whether a block has been captured or not, to place the block more accurately, and finally determine and manipulate the orientation of the block. The next unimplemented feature is the use of image masking to keep the robot within the map boundary. While some parts for this feature have already been constructed, the final combination still requires logical statements to determine what degree of detected pixels are appropriate to determine when we are at a wall, and the logic to handle when and where to turn in order to avoid crossing the boundary at particular sections of the wall. In terms of bug fixes, we will need to obtain an isolated room to set up uniform lighting conditions, and the spotlights that will provide even illumination in all areas of the robot’s environment. Also, complex machine learning algorithms have been discussed that would allow for the robot to better adjust to the light level in an area. These algorithms would allow for NibblesBot to adjust the color parameters in its functions to better represent its current environment. Finally, the majority of our time before the next release will be spent training the machine learning models for Haar cascade and PyTesseract. These models require hundreds or even thousands of hand-selected images before we can begin training, and we may need to perform parameter tuning on the final resulting model, so the required time to train these models can easily escalate into multiple months.