

## Final Report

Autonomous Subteam  
Huey

Spring 2025  
Cornell University

# Contents

1. [Introduction](#)
  - a. [Subteam Purpose](#)
  - b. [Subteam Members](#)
2. [Fall 2024 Recap](#)
3. [Spring 2025 Overview](#)
4. [Spring 2025 Timeline](#)
5. [Code Structure](#)
6. [Subsystems](#)
  - a. [Robot Kit](#)
  - b. [Camera](#)
  - c. [Object Detection](#)
  - d. [Corner Detection](#)
  - e. [Algorithm](#)
  - f. [Transmission](#)
  - g. [Integration](#)
  - h. [Unity](#)
7. [Competition](#)
8. [Review and Reflection of the Year](#)
9. [Fall 2026 Next Steps](#)

# Introduction

## Subteam Purpose

Within the Autonomous Subteam, we specialize in the construction of 3lb autonomous combat robots. Our mission is to build, wire, and code robots that can autonomously compete with conventional human-controlled robots. Autonomous aims to push boundaries in the autonomous robotics field, challenging ourselves to see how well our algorithms can perform in the arena, and continuously enhancing them over time.

Our approach to robot design revolves around 3 critical subsystems that all members participate in

- **Machine Learning/Computer Vision:** Researches and builds the machine learning model to detect our robot from a camera outside of the box through object detection, as well as using computer vision algorithms to find the robot's position in the arena. This year, this section consisted on building the ML model using Roboflow and integrating it into the main code file and designing Corner Detection which took care of finding which robot in the arena is ours and using colored corners to find orientation.
- **Algorithm Design:** Designs an algorithm that takes the positions of robots in the arena and other state variables as input to decide what our robot should do. This year, we plan to develop situational strategies through experimenting with simulated battles using Unity and PyGame. This section also includes the task of integrating all the different subsystems together (Camera, Machine Learning, Corner Detection, Algorithm, and Transmission) in a main.py file.
- **Robot Construction (Mechanics & Embedded Systems):** Researches robot types and components to purchase, as well as ensuring consistent communication between our sensors, computer, and motors. In the Spring semester, we will build and test our new robot.

The culmination of these efforts results in consistent and autonomous robots that break the mold of standard remote-controlled combat robotics competitions.

## Subteam Members

Aaron Harnish	Computer Science '27	Autonomous Lead
Chris Adonizio	Mathematics '27	Autonomous Lead
Ethan Zhang	Computer Science '26	Autonomous Engineer/Ex-Autonomous Lead

Grace Lim	Computer Science '26	Autonomous Engineer /Ex-Autonomous Lead
Shao Stassen	Computer Science '26	Autonomous Engineer/Ex-Autonomous Lead
Alyssa Hsu	Computer Science '27	Autonomous Engineer/Trainer
Ananya Jajodia	Computer Science '26	Autonomous Engineer/Trainer
Camille Yap	Computer Science '27	Autonomous Engineer/Trainer
Jenny Wu	Electrical and Computer Engineering '27	Autonomous Engineer
Ethan DeCamp	Computer Science '28	Autonomous Engineer
Riddhi Lamba	Computer Science '28	Autonomous Engineer
Rohin Phukan	Computer Science '28	Autonomous Engineer
Sophie Cheng	Computer Science '28	Autonomous Engineer
Tyler Lovejoy	Computer Science '28	Autonomous Engineer

## Fall 2024 Recap

The bulk of the fall semester was spent carefully implementing each subsystem on their own. We had 6 different subsystems: Robot Kit, Camera, Object Detection, Corner Detection, Algo, and Transmission.

### Robot Kit

Last semester, we chose our robot for the year, [Jolt](#), from Absolute Chaos Robotics. Nothing else was done in the Fall other than picking the robot kit.

### Camera

Last semester, we finished implementing the warp function using CV2. We first click the 4 corners of the arena to generate a “homography matrix” that describes a transformation from the current image to a supposed “bird’s eye view.” The warp function uses the homography matrix to flatten the plane, given any input image, and make it seem like the camera has a bird’s eye view, which is important for the ML, Corner and Algo subsystems. We tested with an iPhone camera using the “Camo Studio” app, which had the free 0.5x feature.

## Object Detection

Object Detection identifies and classifies robots and house bots from a bird's-eye view of the arena using machine learning models. It employs two main model classes—OurModel (our own custom-built PyTorch model) and YoloModel (a pre-trained YOLOv8 model via Roboflow)—which both return predictions of where robots and house bots are in the arena in a standardized format. While OurModel allows for internal customization and skill development, it has fewer parameters and slower accuracy improvements compared to YOLOv8, which is more accurate but slower to run. Training data is collected by manually warping and annotating frames from overhead NHRL fight footage.

## Corner Detection

The Corner Detection subsystem determines Huey's orientation as an angle relative to the positive x-axis, which is essential for navigation and algorithms like Ram Ram. It works by analyzing cropped images of potential robots, identifying Huey based on the most dominant color (e.g., bright pink), and then locating the colored front (red) and back (blue) corners on Huey using HSV filtering and contour detection. Spatial moments are used to find the centroids of these corners, and if one corner is missing, a helper function estimates its location. Orientation is calculated using trigonometry based on the front corner positions. The system consists of two classes: “ColorPicker”, which allows manual color selection of Huey and his front and back colors, and “RobotCornerDetection”, which automates the detection of corners and angle calculation. A testing script enables batch processing of images using consistent color parameters for streamlined analysis.

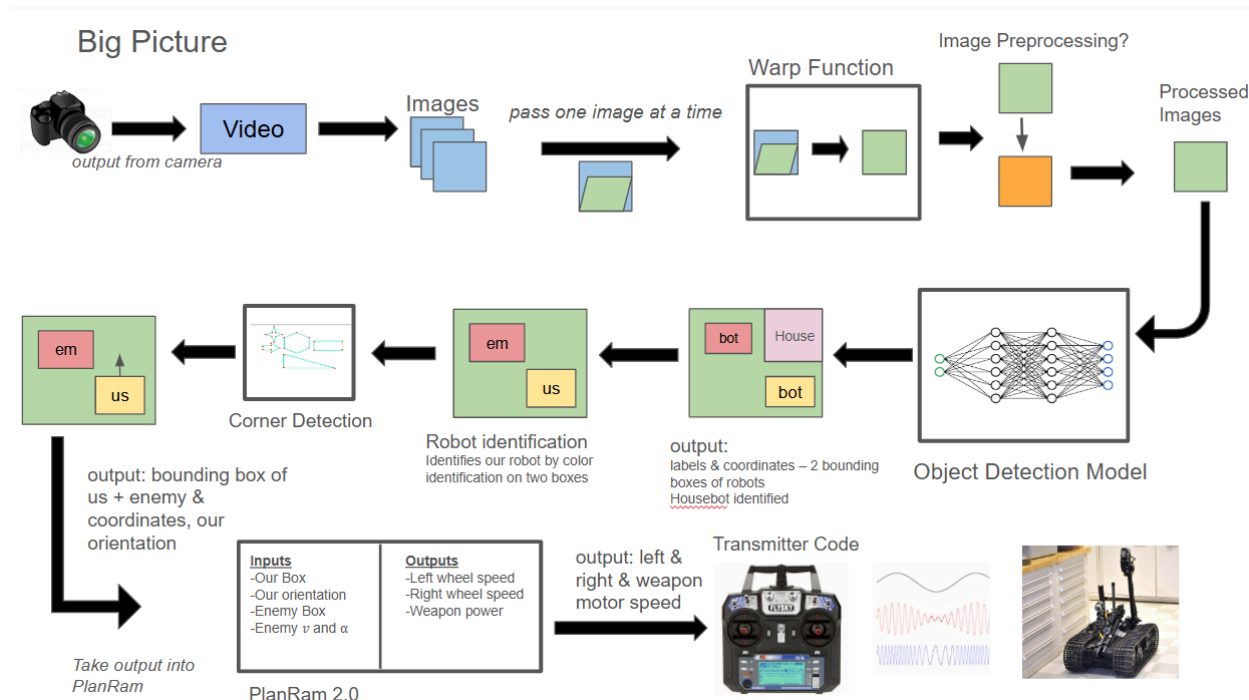
## Algo

The RAM RAM algorithm determines where Huey should go by combining data from Object Detection and Corner Detection to calculate motor speeds that guide him toward the enemy like a homing missile. Based on Huey's and the enemy's current and past positions, the algorithm predicts the enemy's next location and computes the desired orientation angle for Huey. Using these predictions, it calculates speed and turn values. In the Fall, we used pygame and Tkinter to simulate robot behavior and collect data in CSV files for debugging and potential reinforcement learning applications. The math behind orientation prediction involves vector calculations using dot products, trigonometric functions, and normalization to determine turning angles and motor control logic.

## Transmission

The transmission subsystem was fully implemented. This is a procedure that remotely controls the robot from a computer outside the arena by sending motor commands via a serial connection to an Arduino, which converts the input into a PPM signal for a FlySky controller. The output of our Ram Ram algorithm, which is the desired turn and speed values, is sent to an Arduino, which converts those values into PPM for the transmitter, which then is sent remotely to the robot in the cage.

Below is an image of the entire pipeline from last semester:



## Spring 2025 Overview

During this second semester, we picked up from where we left off during the previous [Fall 2024](#) season and worked towards making our robot, Huey, competition ready.

## Robot Kit

After purchasing one full kit with additional files, we decided to order and print the parts separately to save money and time. From the Jolt guide, we compiled a list of the necessary parts and how to get them. In testing the two copies there were physical issues that varied between the versions and adjustments were made. Before comp, we added titanium cleats to the front two wheels by 3D printing holders similar to Kinetic's. We also used silicone lube and sometimes vaseline on the axles to reduce friction. Other than those adjustments, Jolt remained the same.

## Camera

In the first half of the semester, we switched to using the “EpocCam” for initial testing for Integration. This is because “EpocCam” has a faster camera response time then the previous app we used. We paid \$8 for this app since the premium version had the 0.5x feature, and we created an apple ID for the crefirmware account so everyone on the subteam could download it. We are testing with both an external camera and iPhone so that we can have multiple people testing with a camera at the same time. Through rigorous testing, the iPhone appeared to have varying delays that led us to look into other options.

In the second half of the semester, we bought an [external camera](#) that connects directly to the laptop because it would lead to lower latency and better video quality.

## Object Detection

After analyzing the performance of our custom object detection model against a pretrained model, we decided to pivot towards implementing a pretrained model in the interest of time, the team's priorities, and performance benchmarks.

We optimized our pretrained model to also track robots using unique bounding boxes so that each bounding box will consistently detect the same robot throughout a match.

## Corner Detection

Several key improvements have been made to enhance the robot vision and control interface. A new 4-color viewer has been added alongside the color picker, allowing users to easily select and preview the robot's main color, front corner color, back corner color, and ventral color.

We utilized the detected corners to determine the orientation of our robot, which resulted in a stable and accurate estimate of Huey's orientation.

## Algorithm

Most of the basic mechanics of the algorithm can be found in the Fall 2024 report. However, this report includes additional features that were added such as

- Unit Tests
- Test Videos
- Invertibility
- Backing up
- UnFisheye

## Unity

A way to test the algorithm and create more data for object detection without physically setting everything up was to create a physics simulation. To do this, a Unity application was created to represent a random NHRL competition with the robots in a 3lb arena. Within Unity, two Game Objects represent the enemy robot and the autonomous robot, Huey. The enemy robot is controlled by arrow keys while Huey is controlled by the algorithm's outputs.

## Transmission

## Integration

The `main.py` file was created to unify and simplify the robot vision and control pipeline. It defines key global variables and manages the main workflow—starting from the image warping process, moving through the color picking interface, and ultimately running the full match execution with object detection, corner detection, the RAM RAM algorithm, and transmission. The integration process has been streamlined by removing unnecessary clicks and clarifying on-screen instructions, making setup quicker and more intuitive. To improve robustness, try-catch blocks and error handling have been added throughout `main.py`, ensuring smoother runtime performance and easier debugging. Additionally, visual arrows now display the robot's current orientation based on detected corners, as well as a predicted future orientation derived from the RAM RAM algorithm, offering clearer real-time feedback and aiding in both control and development.

Overall, these updates significantly improve system integration by unifying the workflow in `main.py`, reducing manual setup steps, and enhancing real-time feedback—making the entire process of setting up, running, and monitoring the robot more seamless and intuitive.

## Spring 2025 Timeline

Jan Feb	Robot Assembly; Write tests for algorithm; Create a pygame simulation for the algorithm; Pivot to pretrain models for object detection
Week 1	Test integrated system with videos that identify a “fake Huey” moving around
Week 2	Started designing a unity simulation for algorithm; Began manufacturing our own Hueys (not full kit)
Week 3	Integrate transmission into <code>main.py</code> file
Week 4	Ordered numerous parts; Improved UI for <code>main.py</code>
Week 5	February Break
Week 6	Improved decision making in <code>main.py</code> based on confidence score; Specced Battery; Trained new Model and tuned ML parameters
Week 7	Ran the whole integrated system for the first time
Week 8	Investigated the merits of using a predicted future position vs. current position to steer Huey; Decided on current position
Week 9	Got Unity to communicate with Huey's algorithm; Resolved bugs in integration
Week 10	Showcase; Ran algorithm for many hours



Week 11	Spring Break
Week 12	Start the Subteam Report; Implement invertibility for Corner Detection; Start building the second Huey
Week 13	Addressed some drive issues with Huey
Week 14	Implemented code for inverted Huey
Week 15	Huey Back-up code implemented and attempted to undo our lens distortion; Finished up unity with weapon
Week 16	Prepare for Competition; Compete in competition!

## Code Structure

On the main branch:

1. Algorithm/: *For development and testing the RamRam algorithm*
  - a. RamRamTest/: *For storage of RamRam test csvs*
2. camera\_test/: *For collecting camera frames and conducting tests with warping and framerates*
3. corner\_detection/: *For identifying Huey, the four corners, and his orientation in the cage*
  - a. bot\_images/, color\_combos/, helper\_files/, non\_bot\_images/, warped\_images/:  
*Various folders holding test data for the corner\_detection algorithm*
4. machine/: *Contains code for object detection. The file we use in main.py is predict.py. All other files was used for creating the model from scratch from from the fall semester*
5. main\_files/test\_videos/: *Stores the custom videos we made to test the algorithm and integration*
6. raspberry\_pi/: *Incredibly outdated; initial attempts at transmission from the Pi to a computer when we attempted pi-receiver experiments. We do not use this in main.py*
7. scripts/: *Base scripts for setup of development environments*
8. transmission/: *For connection to the arduino and transmission of motor speeds (arduino code)*
  - a. test/: *Test scripts for ensuring the transmission worked*
9. vid\_and\_img\_processing/: *For turning video files into usable training data*
10. main.py/: *Arguably the most important file in the repo, integrates all the different components of the project*

# Subsystems

## Robot Kit

The jolt kit was assembled and we had purchased necessary material for the two additional iterations from other suppliers. With the flexibility we experimented with our brushed drive ESC and brushless ESC to test other models to maximize weight and size. The kit's ESC didn't function as expected, given one of the three signals lacked power and the inability to "tank" drive, and later ESCs also failed; one reverted to tank drive, after building our code base around speed-turn, without prompting. The issue was resolved by purchasing the original ESC from the Jolt supplier.

The physical component changes were done throughout the course of assembling and testing the kit. We separately ordered the metal components from SendCutSend, which arrived promptly and to the desired specifications. Our own 3D prints allowed for custom colors and some modifications, like the wheel hubs later on. We asked the original supplier many questions through Discord regarding the weapon. He told us to buy a bolt that appeared to be too long, resulting in an attempt to machine down the bolt but due to the shape, no possible changes were able to be made. In the interest of time and efficiency, our solution was to purchase specific components from the jolt kit website. We later found out that the bolt was the right size, we had just been screwing it in without the weapon for testing. See competition section for information about titanium cleats.

Altogether, while purchasing individual components for the second and third bot separately did likely save us money, it cost us time and effort to find links to, and ensure we had, every part, as well as staggered order times. To that end, overlooked components and last-minute additions likely added overheads and additional stress. For ease of mind and without putting more emphasis on the mechanical side of our robot, ordering the kit was likely an easier solution and should be considered moving forward.

## Camera

In the spring, we transitioned back to using the "EpocCam" app as our iPhone-to-computer streaming solution after discovering that it offered a noticeably faster transmission rate compared to the "Camo Studio" app. While Camo Studio had previously been selected for its low latency and access to the 0.5x lens without a paywall, further testing revealed that EpocCam provided a more responsive feed in our runtime environment—an essential improvement for real-time robot operation. This change allowed for more efficient data flow from the iPhone to the laptop, minimizing potential frame lag. Importantly, this adjustment only affected the streaming layer; the downstream processing, including image warping and frame transformation, remained unchanged. The warp initialization and runtime transformation pipeline continues to use the same homography-based CV2 methods as previously described.

In the second half of the semester, we upgraded our setup by purchasing an external camera that connects directly to the laptop via USB. This decision was motivated by the need to reduce latency and improve video quality, both of which are critical for real-time computer vision tasks. Unlike our previous setup, which relied on a smartphone camera and wireless streaming through Camo Studio, the external camera

provided a faster and more stable connection. The direct USB interface eliminated network delays and compression artifacts, resulting in a clearer video feed with minimal lag. This improvement significantly enhanced the performance of our vision-based algorithms, particularly in tasks like corner detection and real-time tracking, where timing and image clarity are essential.

In the days leading up to comp, we found a few drawbacks with the camera. One, it was capped at 50 FPS, and if you tried to pull frames any faster it would buffer for ~30 ms before feeding you the next one. Two, it is a low light camera, which either didn't help or severely affected our ability to detect colors consistently. Three, the FOV wasn't wide enough, solved by purchasing and swapping on a wide angle lens.

## Object Detection

At the start of the semester, we continued optimizing OurModel, while also exploring the possibility of switching to a pretrained model.

### Optimizing OurModel

Our optimization efforts focused on runtime, training time, and accuracy. Most of our testing was done on average-performing laptops like the M2 MacBook Air. During the first week, we experimented with hardware upgrades—including an M3 chip, gaming rigs, and high-performance GPUs. Notably, Aaron's computer achieved a 9 ms runtime in the native terminal and 10 ms in the VSCode terminal. By leveraging the Apple Silicon GPU, we reduced runtime even further to 2 ms. While this 8–9 ms improvement might seem minor, Huey's performance depends on real-time feedback, so any runtime reduction is valuable.

As our dataset grew, training times naturally increased. A larger dataset helps the model generalize better and improves testing accuracy, so we chose to prioritize accuracy over speed since training is done before competition day. However, we did look into training on Google Collab to speed up the training process which required minimal code changes—mainly in import statements and file handling.

We also attempted several model architecture adjustments—such as adding dropout layers, implementing early stopping, and tweaking hyperparameters. Most of these steps were outlined in the Fall Subteam Report. By late January, however, we had exhausted most debugging options and began to question the long-term viability of OurModel.



The image shown earlier tracks the loss across several OurModel training sessions, each represented by a different color. Both the house bot and robot classes had unstable loss curves with little convergence. Although not shown, the mean Average Precision (mAP)—our measure of accuracy—remained consistently low, often near 0, which confirmed OurModel was underperforming.

### New Member Model

As part of a training exercise and a new approach to making our own model, a few of the new members (Rohin and Tyler) attempted to build their own CNN model called Tohin. The main idea going into building Tohin was to work from the team's `ah_model` while making a few key differences backed by industry standard practices and some specific research. The goal was to get our model architecture accurate enough at a much smaller size and run time than models like YOLO, which have bulky architecture. Some of the key changes we were attempting were: max pooling every set of layers but with kernel of three and stride of two, a higher number of features, and one less layer to compensate for a higher number of features. Outside of the model architecture we attempted to look into the optimizer and loss weights. In the end the pivot to pretrained models happened before we could perfect our model, however we found promising results with fractional max pooling and additional features.

### The Pivot to Pretrained Models

By late January, we decided to pause development on OurModel and pivot to a pretrained model. This decision was made for two main reasons:

1. Faster integration and debugging: As we began integrating all subsystems, we needed a reliable object detection model to isolate issues elsewhere in the pipeline. A better-performing model allowed us to test the entire system more effectively.
2. Higher accuracy: Despite extensive optimization efforts, OurModel consistently underperformed. The pretrained models offered superior accuracy and solid runtimes with minimal setup.




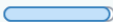
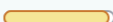








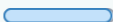
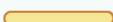






### Pretrained Model Performance

Our first pretrained model from last semester was a Roboflow YOLOv8 model with a runtime of 30 ms. Early this semester, we trained a YOLOv9 model, which cut the runtime in half to 15 ms while maintaining the same accuracy. We later moved to a YOLOv11 model, which we used extensively until Roboflow released YOLOv12, which we began training and testing in late February.

The table below compares the different YOLO versions and explains how their evolving architectures contributed to improved accuracy and runtime:

YOLO Version	Key Architecture Changes	Accuracy Gains	Runtime Improvements
YOLOv8	Standard CNN with PANet	Solid baseline accuracy	Decent speed and efficiency
YOLOv9	Better anchors & feature fusion	Improved object localization	Lighter layers, faster inference
YOLOv11	Added attention & skip connections	Better for small/occluded objects	Optimized memory usage
YOLOv12	Hybrid CNN-Transformer backbone	Best accuracy with refined loss	Fastest due to efficient ops

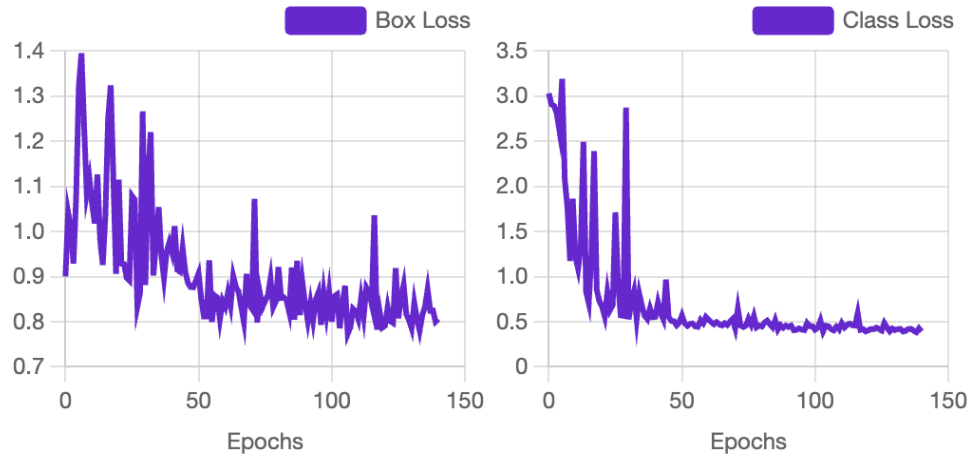
The image below shows our performance in mAP, precision, and recall across different YOLO versions as well as the date we trained the model.

<b>NHRL Robots 15</b> ID: nhrl-robots/15 	 3/20/25 8:39 PM	mAP@50 96.3%  Precision 95.7%  Recall 94.8% 	YOLOv12 Object Detection (Fast)
<b>NHRL Robots 14</b> ID: nhrl-robots/14 	 2/25/25 10:28 PM	mAP@50 95.7%  Precision 94.6%  Recall 93.5% 	YOLOv12 Object Detection (Fast)
<b>NHRL Robots 13</b> ID: nhrl-robots/13 	 2/20/25 9:16 PM	mAP@50 98.7%  Precision 97.6%  Recall 98.9% 	YOLOv12 Object Detection (Fast)
<b>NHRL Robots 10</b> ID: nhrl-robots/10   Early Stop	 1/17/25 11:34 AM	mAP@50 98.2%  Precision 97.0%  Recall 98.5% 	YOLOv11 Object Detection (Fast)

The noticeable drop in accuracy in our last two YOLOv12 models was due to a data splitting issue. Our dataset (2,200+ images from multiple NHRL 3lb matches) was originally split using a standard 80/10/10 train/val/test ratio. However, some robots appeared in multiple splits, meaning the model could have already seen a robot during training before encountering it in validation or testing—artificially inflating accuracy.

### Additional Exploration

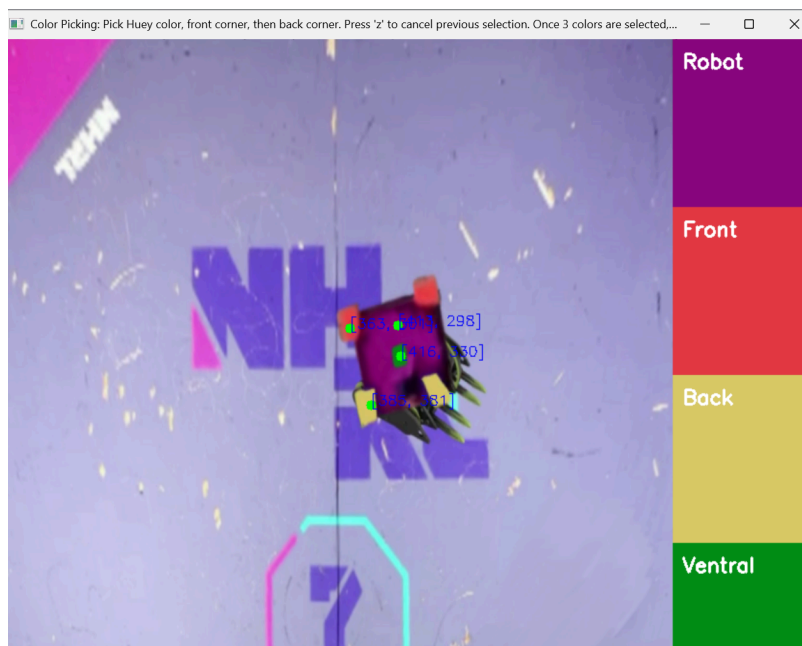
We also explored running multiple object detection instances concurrently on subimages to reduce runtime. However, due to Python's concurrency limitations and added complexity, this approach wasn't feasible for our current setup.



Overall, our most recent YOLOv12 model demonstrates strong convergence and low loss, a clear contrast to OurModel's unstable training curves. This semester's shift toward pretrained models has resulted in better accuracy, faster runtimes, and a more reliable system as we continue building Huey.

## Corner Detection

To improve usability and accuracy in robot vision setup, we significantly enhanced the color picking interface by integrating a 4-color visual preview panel alongside the existing manual color selection process. Previously, users could click on an image to select HSV colors for the robot and its front/back corners, but there was no immediate, persistent visual feedback beyond terminal print statements. The updated interface now provides a live, side-by-side color viewer that dynamically displays the selected colors in labeled blocks—"Robot", "Front", "Back", and "Ventral"—as users click points on the image.



This was achieved by modifying the ColorPicker class in the camera\_test directory. Now, during selection, every click triggers a redraw of the image window, overlaying the selected points and displaying a separate vertical panel filled with the currently selected colors in BGR format (for human interpretability). This panel helps users verify that their selections are visually distinct and appropriate for downstream color segmentation tasks.

The system enforces the order of selection—robot color, front corner, back corner, and ventral color—and allows undoing the last point using the 'z' key, providing greater flexibility during the setup process. Once all four colors are picked, they are saved in HSV format to a text file and visualized again in a horizontal color block layout using the display\_colors method.

An important technical refinement was also made to ensure all selected colors are properly displayed during the color picking process. Initially, the interface was designed to accept four color selections—robot body, front corner, back corner, and ventral color. However, due to how the OpenCV event loop works, the display panel wouldn't render the fourth (final) color unless the user made an additional input after selecting the last point. To solve this, the logic now waits for 5 total clicks, using the last one as a trigger rather than an actual color selection. This ensures that all four intended colors are visually displayed on the color panel without requiring confusing or unnecessary extra user input. After the fourth click, the code discards the extra color and finalizes the first four as the selected HSV values. While the change may seem minor, it significantly improves the clarity and reliability of the interface, making the setup more intuitive for users.

Importantly, this UI enhancement does not alter the downstream logic of the robot's corner detection or color segmentation. It simply improves the interface at the setup stage, helping reduce errors and making the process more intuitive for operators.

### Orientation Determination

Initially, Huey's orientation was determined by taking the perpendicular direction to the line along which the two front corner points lay. However, we realized that this method was liable to large amounts of error because if the two front corners were detected improperly, the error would propagate and significantly affect the orientation error and can cause the orientation direction to fluctuate greatly with discrepancies in object and corner detection. Furthermore, smaller changes in the front corner position/orientations would be given greater weight in the overall orientation calculation. We noticed this especially when we attempted to drive Huey in a straight line and observed that Huey's fluctuating orientation values (even when its actual orientation had not changed significantly) made it difficult for Huey to travel in a straight path. To mitigate this issue, we began using both the front and back corners to determine the orientation. To do this, we take the median point between the front two corners and the median point between the back two corners. We take the orientation to be the direction of the vector that goes from the median of the back two corners to the median of the front two corners. By taking into account more points for the determination of orientation, we are able to get a more stable and accurate estimate of Huey's orientations that allowed us to drive much straighter and improved our accuracy in movement.

# Algorithm

During January Fabrication, we worked to thoroughly test the algorithm we wrote in the fall. We did this by creating unit tests for each helper function in the algorithm. This process was grueling but incredibly helpful as we found many bugs. One thing we learned is that numpy types are passed by reference (not by value). This means if we updated a numpy parameter in a function, it would modify the real variable. To simulate expected inputs to the algorithm, we created a pygame. Users could control our bot and the enemy bot in game and observe some of the algorithm output on the top of the screen. We also had a csv file that would save the values of inputs and outputs to the algorithm on each iteration. We later developed a version of the pygame that had our bot controlled by the algorithm and the enemy controlled by the user. These were crucial for high level testing before we had a physical robot.

At the start of February, we reviewed the fall semester's algorithm code and what's left to work on. First we looked into ways at optimizing the algorithm such as

Late February, we began implementing changes to help us select a predictor. Originally, there was 3 options:

1. Position loss: predicts the enemy to be at the last recorded enemy position
2. Velocity loss: predicts the enemy to be at some calculated position based on last recorded enemy velocity
3. Acceleration loss: same as (2) but based on last recorded enemy acceleration

Briefly after some simple tests, we realized acceleration loss is out because its loss was significantly higher than prediction and velocity loss every time. Loss is calculated as the Euclidean distance between the enemy's actual position and predicted position. We hypothesized it was overshooting every time since its complex to properly predict position based on acceleration and time elapsed. Thus, our next task was choosing between position and velocity loss or finding a perfect balance between the two.

## Test Videos

To help test both main and which loss function to use for the algorithm we made test videos using a few different methods. We started by taking a png of huey and a different 3lb bot and adding them over an image of an empty NHRL cage, and using Canva, made custom paths for them simulating a battle. Next we tried downloading 3lb fights and using the real paths of the bots by overlaying an image of Huey over one. This method struggled as making Huey's path match the bot's was difficult. The final and best way of making test videos was using CapCut motion detection, which would animate Huey's image over a bot with remarkable accuracy.

On the top corner of the pop up video of the match, we display the loss which updates every frame. There is also a blue arrow indicating Huey's orientation and a red arrow indicating Huey's desired angle (facing the enemy). In addition to displaying the loss values on the video, we also write it to new txt files with a timestamp for further inspection.



## Invertibility

In early April we realized that there was no way to know whether Huey was inverted. The agreed upon solution was to add a fourth color to the bottom of Huey and add a swatch of that color to the top for the first frame of main so that we could select it. To facilitate the solution we modified the color picker document to take four colors and then modified the algorithm. The method we settled upon was to check whether we could find a big enough contour of the top color, and if not we would check for the bottom color.

## Back-Up

In late April, we further improved the algorithm by adding considerations for when Huey has been in the same (or similar) position and orientation for a long period of time (presumably if Huey is stuck against a wall or is pinning/being pinned by the enemy robot). We consider orientation as well as just position in case Huey is in the same position for a long time but is changing its orientation to turn to face the enemy. In order to address this, we began recording Huey's position every 5 frames in an array that stores the 20 most recent frames. Before running the main algorithm code, we check if Huey has been in similar positions and orientations (within a tolerance of 10 pixels for position and 5 degrees for orientation) for 8 or more of these past 20 frames. If so, we do not run the regular algorithm and instead send Huey instructions to back up (by setting Huey's speed to -1 and turn to 0). For the next 0.5 seconds, Huey is instructed to continue backing up in this way instead of running the regular algorithm to ensure that Huey is given sufficient time to back up. After this, the algorithm resumes as normal.

We initially also added considerations that check if Huey is against the wall and its orientation has its back towards the wall. In these cases, we did not instruct Huey to back up because backing up would only drive Huey deeper against the wall and force it to stay in that position. However, we ended up removing these considerations for competition because we assumed that if Huey was stuck, going with the normal algorithm was already failing to help Huey out of his position and so backing up would always be a better option to allow Huey to find a different angle to escape.

Overall, this backup code in the algorithm has been very useful because it has allowed Huey to readjust its position and escape from situations when it is pinned or against the wall. Previously, Huey had no defense in these cases and would simply be unable to make any useful movement until it was removed from where it was stuck by an external factor. Now, Huey can fix itself by backing up out of the place it is stuck, readjust, and keep going.

# Unity

## Summary

The idea of a full physics based simulation first came up during JanFab at the beginning of the semester when ideating possible ways of testing our algorithms without having to deal with the physical limits of batteries and motors (setting up a real life test run takes time and confounding variables make it difficult to determine what error is due to algorithms vs mechanical and physical variables). In the short run, we decided to make pygame, which was a 2D model of what the algorithm aimed to accomplish.

We realized though that we would eventually need something more robust to get a realistic sense of what our algorithm would look like in competition (and possibly underpin reinforcement learning models in the future). We quickly settled on Unity, as most physics are handled by the model, meaning we could focus more on getting our simulated Huey to match our algorithms.

None of the group members working on the simulation had learned Unity previously, so the learning curve was at first steep. Learning Unity involved watching tutorials and helpful tips from Luke Murphy. We first worked making cars drive, then imported a CAD of Huey, which we then allowed to drive. The next step was simulating our autonomous Huey.

In the simulation, two GameObjects with the Huey prefabs that represent Huey and the enemy robot are positioned on opposite corners of the arena, representing the NHRL 3lb arena setup. For visual purposes, the walls are transparent. The enemy robot can be controlled either with the arrow keys or a PS4 Controller depending on which files are selected. Meanwhile, Huey is controlled by the algorithm. Both bots can be controlled by PS4 controllers as well.

### In Depth:

Autonomous Huey involves two parts: Python and C#. The Python side runs the algorithm through ramram, and the C# side controls Unity. Two two scripts communicate with each other through .txt files (one is read by Python and written by C# and the other is read by C# and written by Python). C# communicates positions and orientations, and Python communicates back right wheel speed, left wheel speed, turn and forward speed.

Realistic physics was key for making Unity worth the time. For this, we added delay on both the C# side and the Python side. The delay on the Python side (50ms) represents processing delay associated with object detection, and the delay on the C# side (35ms) represents camera delay.

There are three Python files (unity\_test.py, unity\_reads.txt, and unity\_write.txt) and multiple files on Unity (Auto, Auto\_Control, Auto\_Weapon, Car, PS4\_Controller\_Enemy, Restart, and Weapon). The Python file, unity\_test.py, is where all the communication on the algorithm side is. It will write the right wheel speed, left wheel speed, turn and forward speed from the algorithm into the text file unity\_reads.txt and read the positions of both bots and the orientation of Huey from the text file unity\_write.txt.

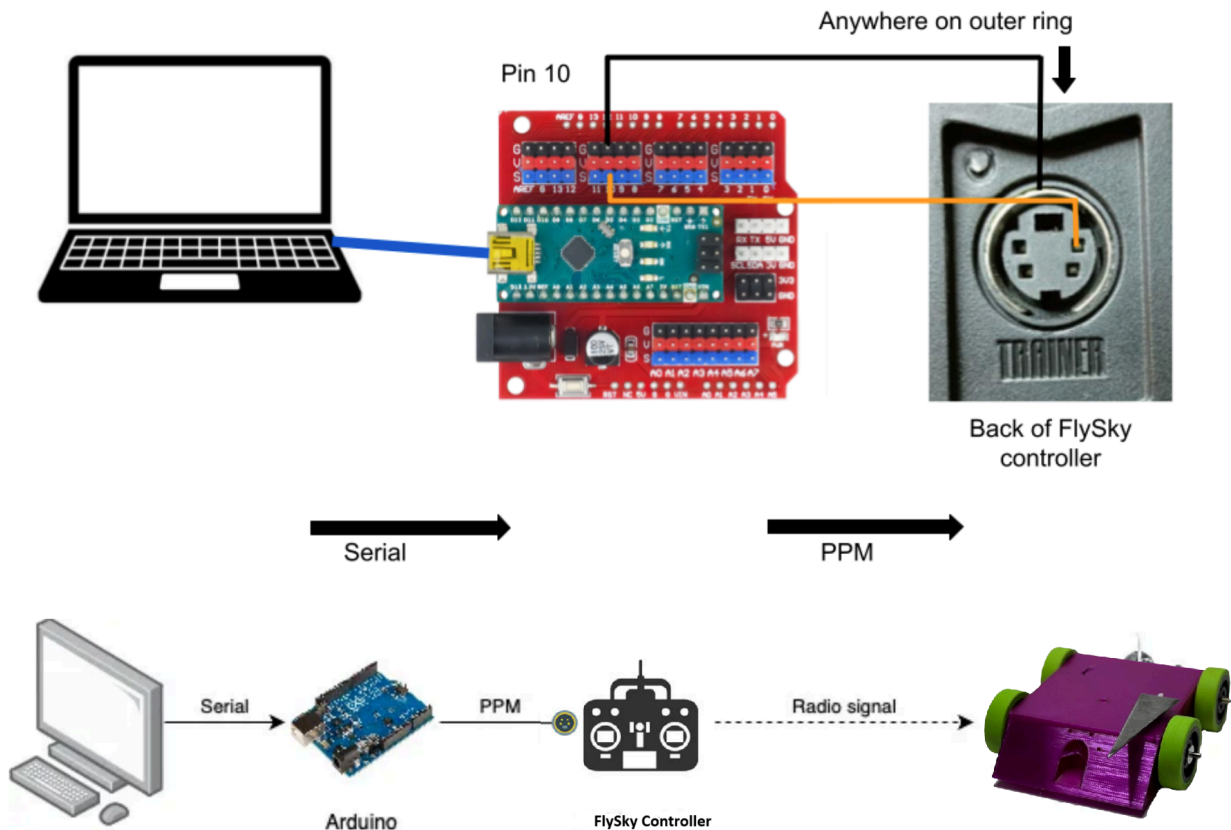
The Auto, Auto\_Control, and Auto\_Weapon files are all attributed to the Huey GameObject. Auto is the file to select for autonomous control as it will write information about the bots in unity\_write.txt, while Auto\_Control is the file to select for manual control on Huey. Auto\_Weapon and Weapon both control the weapons on the respective bot. Car and PS4\_Controller\_Enemy both control the enemy bot; Car is used for arrow key control while PS4\_Controller\_Enemy is used for PS4 Controller control. To make testing efficient, the Restart file allows the user to restart the simulation by pressing the Spacebar.

### Future Development:

As of now, the simulation fully follows the algorithm as we expect the bots to. However, many improvements could be made with respect to the physics aspect. Collisions are not accurate as the weapons on both bots slide under the other bot instead of directly interacting and hitting the bot.

## Transmission

### Summary



Transmission was used to transmit desired control signals to the robot. Like using the flysky controller manually, the motor class allows you to change a specific channel in the controller. This channel is linked to a motor on the robot (since we used a dual esc with mixing, we had one channel linked to 'speed' and one channel linked to 'turn'). Motor updates can be done by creating a serial object and passing it into the motor class with the channel for the motor. From there, use the move function to update the speed of the motor (from -1 [full speed backwards] to 1 [full speed forward]). See the sample code below for an example of usage. See the Transmission README for all setup information and the Fall 2024 report for more detail.

### Changes for comp

When timing our code, we observed that adjacent motor signal updates would result in the second taking much longer than the first. This is most likely since both motor updates need use of the serial

communication with the Arduino. Since the Arduino needs time to process the first request, the second gets delayed. A change was made to the motor class so you could optionally update 2 motors at the same time. The code still works for updating an individual channel as all second channel parameters are optional.

## Integration

The `main.py` file is where all the different subsystems interact with each other. This is the file we run during competition by executing `python main.py`. If we want to profile the performance of each function in the code to identify bottlenecks or optimize runtime behavior, we run `python -m kernprof -lvr --unit 1e-3 main.py`. This command uses the `line_profiler` module to measure the execution time of functions line by line, reporting results in milliseconds for precise performance analysis.

To see a more detailed description of how to exactly run the `main.py` file, please refer to the [Autonomous 2025 Competition Guide](#).

The robot's main control loop orchestrates several vision and control subsystems in sequence, each contributing to real-time autonomous decision-making. At the start, we run an initial cycle to initialize all components—camera connection, warping setup, and model loading—ensuring the system is fully prepared before entering the live loop. First, video frames are captured from a camera source (either an iPhone or the external webcam). We can choose whether to process every single frame or skip frames to improve performance, depending on the system's current load and the desired responsiveness. These raw images are then passed to the warp function, which performs a perspective transformation to simulate a top-down view. This transformation is essential in ensuring consistent spatial reasoning, enabling subsequent components like object detection and corner detection to operate in a normalized coordinate space.

Next, we pass the warped bird's eye view image to a YOLO (You Only Look Once) model to the warped frame to identify entities of interest, such as robots and housebots. The YOLO model will return the robot and housebot's coordinates in the image as well as their class labels.

Following passing images into object detection, Integration will invoke the Corner Detection module, which analyzes the warped frame to locate specific colored markers (e.g., red and green) on the robot. We use these markers to both tell which robot of the two in the arena is ours as well as to find the front and back of the robot, allowing the system to calculate its orientation. This orientation is crucial for enabling strategic movement and interaction with the environment.

The detected corners and object positions are then passed into the autonomous control logic, known as the RAM RAM algorithm. This algorithm uses enemy positions, velocities, and predicted future locations to determine the most effective motor commands, whether for chasing, avoiding, or maneuvering toward goals.

Finally, if the `IS_TRANSMITTING` boolean is true, these motor commands are transmitted to the robot via a serial connection using the Transmission module. Once the robot receives these commands, it executes the corresponding movement in real time. This entire loop—spanning vision, detection, reasoning, and actuation—runs continuously, allowing the robot to respond dynamically to its environment in the arena.

The `main.py` configuration uses Boolean flags to control runtime behavior:

**1. Environment & Hardware**

- `MATT_LAPTOP`: Use Matt's laptop-specific settings.
- `JANK_CONTROLLER`: The autonomous controller labeled “Old” has an issue where the weapon's true zero position is at -1 instead of 0. So if we are transmitting and using the old controller, we set `JANK_CONTROLLER = True` to apply a correction offset, ensuring the weapon behaves as expected during autonomous operation.

**2. Performance & Competition**

- `COMP_SETTINGS`: Enables competition mode—disables visuals, debug prints, and sets `MATT_LAPTOP = True`.
- `IS_ORIGINAL_FPS`: If `True`, processes every frame; otherwise, skips frames to improve speed.

**3. Preprocessing & Initialization**

- `WARP_AND_COLOR_PICKING`: Runs setup for perspective warp and color picking of Huey's front/back corners.
- `UNFISHEYE`: Applies fish-eye lens correction to the camera input.

**4. Execution Mode**

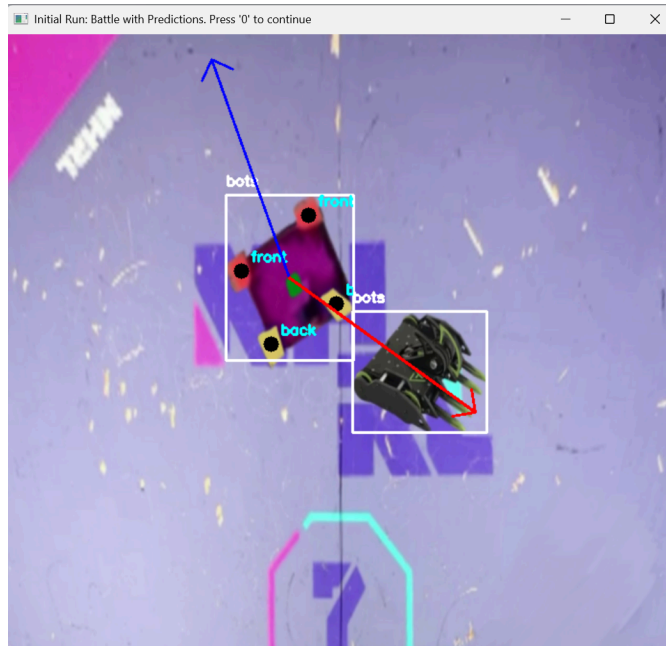
- `IS_TRANSMITTING`: Enables live communication with Huey via serial.

**5. Debugging & Visualization**

- `PRINT`: Enables debug print statements.
- `SHOW_FRAME`: Displays processed camera frames.
- `DISPLAY_ANGLES`: Shows current and future orientation angles; only works if `SHOW_FRAME` is `True`.
- `TIMING`: Logs processing times for performance profiling.

Additionally, if `SHOW_FRAME` is `False`, `DISPLAY_ANGLES` is automatically disabled.

Over the course of the semester, we added multiple features to our display during the match. First, we display the bounding box that shows which areas the object detection model detects bots in. We label housebot as red and the other robots as white. For the bot that is detected as Huey, we display the points that are being detected as Huey's corners, label the front and back corners, draw a blue arrow for Huey's orientation, and a red arrow for the direction in which the algorithm is instructing Huey to drive.



## Competition

In the days leading up to comp, a few of us stayed up late and made some last minute changes to Huey that deviated from our plans up to that point.

### Pre-comp changes

#### Titanium Cleats

For most of the semester, we had been testing on carpeted floors in Upson with our foam wheels. Upon testing on wood in the loading dock, we realized three things. One, the foam wheels did not generate enough traction for controlled driving. Two, the hubs were not connected well to the foam wheels, and with a small amount of force the wheels would spin independently of the hubs. Three, the bolt in the middle holding the weapon would drag on the ground when we drove, increasing our steering problems. Adding tape on the wheels helped these, but not sufficiently. To solve all of these, we had James CAD 3D printed titanium cleat holders similar to the ones on Benny that weighed an ounce more than our normal wheels. We had 5 cleats left over from when Sportsman ordered the wrong ones that we attached and used during comp. The cleats greatly improved traction and made our robot drive much much nicer. We used two on the front, but could have used two on the back also if we made the holders less heavy and had more cleats.

#### PID

We added a PD control system for turning to solve an “orbiting” pattern in our movement. Our algorithm originally used a purely proportional system, where the turn value was directly proportional to our angle away from the enemy. This meant that we had extra rotational momentum when we finally faced the

enemy and turned past, starting a cycle. The derivative factor was much smaller, but acted in the opposite direction to our rotational velocity. For example, if we were 90 degrees away from the enemy, and moving at 45 degrees towards the enemy per iteration, our turn would be  $(0.5 * 1) + (0.25 * 0.1 * -1)$ . This meant we would slow down more when quickly turning towards an enemy. In testing, we found this to be very helpful and saw great improvements in the bot's driving ability, but didn't have enough time to fully tune the parameters for a good drive. We also linearly scaled the speed/turn values after applying PID to lower our top speed and try to retain more control.

## Corner Detection

The changes made here were more sporadic, but we played around a lot with the boundaries for HSV in both the bot detection and corner detection to get more consistent detections. It ultimately did not work great, but one good change was measuring the lower threshold for a detection as a percentage of that detection's area instead of a hard pixel count.

## Camera Frame Limiting

When timing debugging info was added, we saw a consistent cycle of large spikes (nearly double) in image capture time every 2 or 3 frames. Upon research and trial and error, we found that limiting the framerate to 50 FPS got rid of these spikes, as they were caused by the program trying to pull another frame from the camera when it wasn't ready yet, so would wait another cycle. Later at comp, we found out that the camera was advertised as limited to 50 FPS.

## Competition Performance

At comp, we did not perform very well in the traditional sense of combat robotics, but we did learn a LOT. First, here is a quick summary of what happened at safety and in our matches. The videos can be found in the box drive.

### Safety (night before)

After speaking with NHRL officials, we were pleased to learn that we did not have to use our autonomous system during the safety check as everything ran through the transmitter, and we could just drive it manually. Safety went very well. After setting up the camera near the upstairs test box, we saw that the object detection model was performing poorly and barely detecting our bot. This led us to try again on one of the real boxes downstairs, where the object detection model worked perfectly, but the corner detection was flaky at best. We took a video downstairs, which was used for some debugging later at the hotel for corner detection. Our weapon was not spinning up, so we had to change some code to set the weapon speed to 1 at the start of each match, which worked. We briefly crashed out at the hotel, buying spray paint and nearly coating the entire bot to get more consistent robot detections, but decided against it as the bot detections were not the issue, the corners were. We debugged corner detection in the lobby to try different methods to make the camera work, but ultimately did not even pull the few changes made. We also realized that our camera was A. limited to 50 FPS and B. designed to be low light, so our color issues were partially because the image was oversaturated i.e. there was too much light.

## Pre First Match

We realized that we needed to be charging for Matt's laptop to run at faster speeds, so we asked NHRL for a small generator to carry next to the cage. We trusted that object detection worked fine on the test video, so it would be fine in match even if it wasn't working in the upstairs cage. However, we had to take an extra 15 minutes because we were unprepared; all the wires were unstable and flaky, and as such we were unsure whether the system would run successfully; for instance, the camera was showing all red or blue tinted sometimes.

## First Match

On the walk down, the two wires into the arduino came unplugged, and we had to scramble to get them back in the right spot as we walked. Walking into the arena, we set up the camera without looking at the laptop screen, leaving it at a slightly higher angle than it should have been. We had to take extra time to drive Huey back into the center so an image could be taken for corner detection with Huey there. Didn't have a swatch for the bottom color. Was able to run weapons and drive manually. Turned competition settings on for faster runtime at the last minute, so we didn't have visuals of what was happening. Upon match start and flipping the transmitter switch (i think?), the weapon did not spin up and we didn't move. Could be because arduino was plugged in poorly (they were in the right positions tho) or somehow didn't flip the transmitter or the weapon code didn't work. From the prints, we were consistently detecting the enemy robot and identifying it as not ours, but never detected our own robot. This is because our robot was out of frame in the bottom left corner, and when not detected it continued doing what it was doing, which was nothing at initialization. The poor camera angling was expounded by the defisheye function, which crops out some more of the corners by design. Ultimately, we did not move and were pummeled mildly.

# Review and Reflection of the Year

## Summary

This year was the subteam's 2nd year of making a robot as a proof of concept. This time we wanted to show the viability of making an autonomous robot that handled all of the computing/sensors outside of the actual combat robot that we made. With this end in mind, despite an unlucky first match at competition, we largely proved that this was a viable goal and created a robot that made decisions with some intelligent aim in mind.

## Improvements discussed post-Competition

The subteam after competition had a meeting to discuss what went well and could be improved in terms of our team structure. In this section we will be briefly summarizing some of the different viewpoints shared during this meeting.

We found that a lot of improvements could be made in the way we went about **testing**. Specifically, we decided that throughout the semester we should try to do more unit testing, like we successfully did for



our algorithm code. Additionally, something new that was suggested is Failure Mode and Effects Analysis (FMEA) which is a structured method to brainstorm any small thing that could cause an issue in a large engineering project. Lastly, we thought that more frequent code reviews and more professionalism in our code writing could help resolve many issues we ran into during the later half of the semester.

There were a variety of recommendations to make sure that knowledge from the team was better distributed among subteam members. Some of these recommendations include working on a smaller sprint style environment that would allow people to switch around more freely and also keeping groups closer to 2-3 people.

## Fall 2026 Next Steps

Due to the late competition date this year, we will not have sufficient time to have a thorough discussion for our next steps. It's uncertain whether we want to build off of and continue optimizing Huey or start anew the upcoming semester. For this reason, our next steps will be finalized at the start of next semester as opposed to being in this report.

Some key ideas from our discussions:

- Build a more robust system
  - Design choices (don't rely on noisy inputs like color)
  - FEMA
- Sprints of some form
  - More effective version of timelining