ENPH 353 Final Report

2025 Clue Detective Competition

16/12/2025

Taichi Kamei, Bowen Yuan

2025-2026

ENPH 353

Table of Contents

I - Introduction

I.I - Competition Requirements and Goals

In this project, we designed a fully autonomous "Fizz Detective" robot that can navigate a racecourse in a simulated ROS Gazebo environment while identifying clues to a crime on boards located around the track. There were two main challenges in this competition:

- Drive around the track without violating any traffic rules
- Detect the prescence of clue boards and read the clues

Points were awarded for succesfully identifying the clues to solve the crime. However, points were deducted for traffic rule violations like driving off the road or hitting other vehicles and pedestrians. There were four minutes of simulation time to try and achieve the maximum 57 points with ties being broken by time.

Our goal for the competition was to make it possible for our robot to achieve maximum points. Specifically, this meant we wanted to be able to drive through the entire course without using respawning our robot as that would incur point deductions and restrict our maximum point potential.

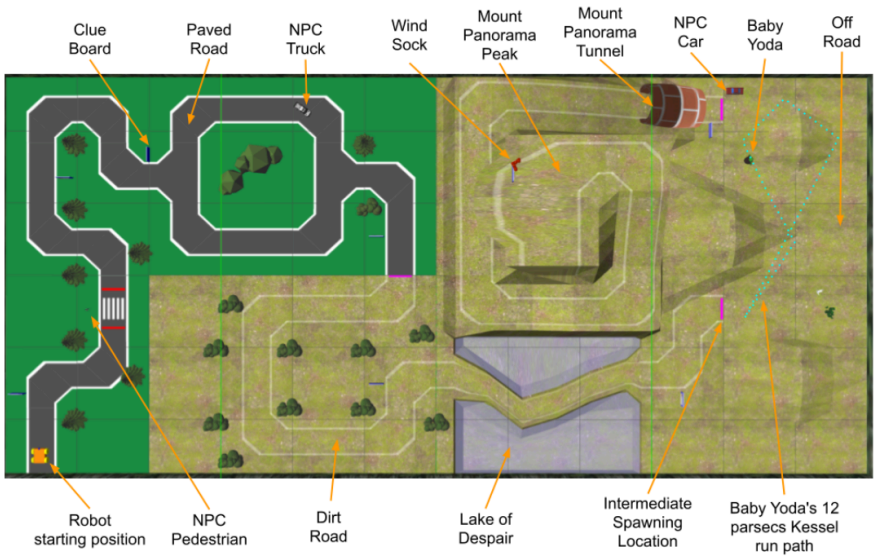

Figure 1 — Map of ROS Gazebo Competition Enviroment

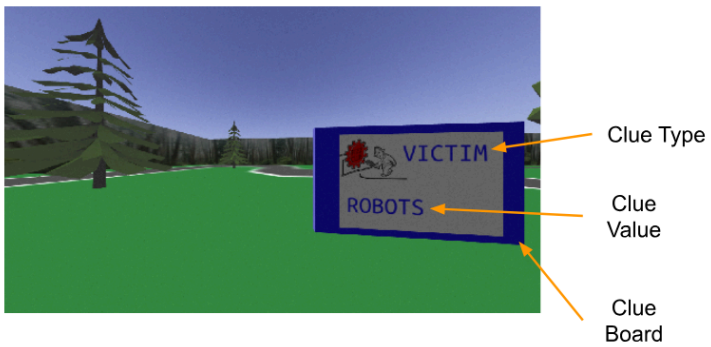

Figure 2 — Simulated Camera Feed of Clue Board

I.II - Contribution Split

We brainstormed the overall strategy we would follow together. We decided on using PID navigation as it would be simpler to implement than an imitation learning or reinforcement learning system. Taichi developed the robot's PID navigation algorithm and state machine. For detection of clue boards and optical character recognition (OCR), we decided to use a simpler system based on masking the specific blue colour of the board and a small convolutional neural network as opposed to using a YOLO based system. Bowen developed the clue board detection and the convolutional neural network (CNN) for OCR.

I.III - ROS Architecture

Shown below is the structure of our ROS nodes and topics. The ROS nodes are in the black box, and the topics are highlighted in grey. Bold arrows indicate the ROS node interactions through topics, while dashed arrows represents local access relationships.

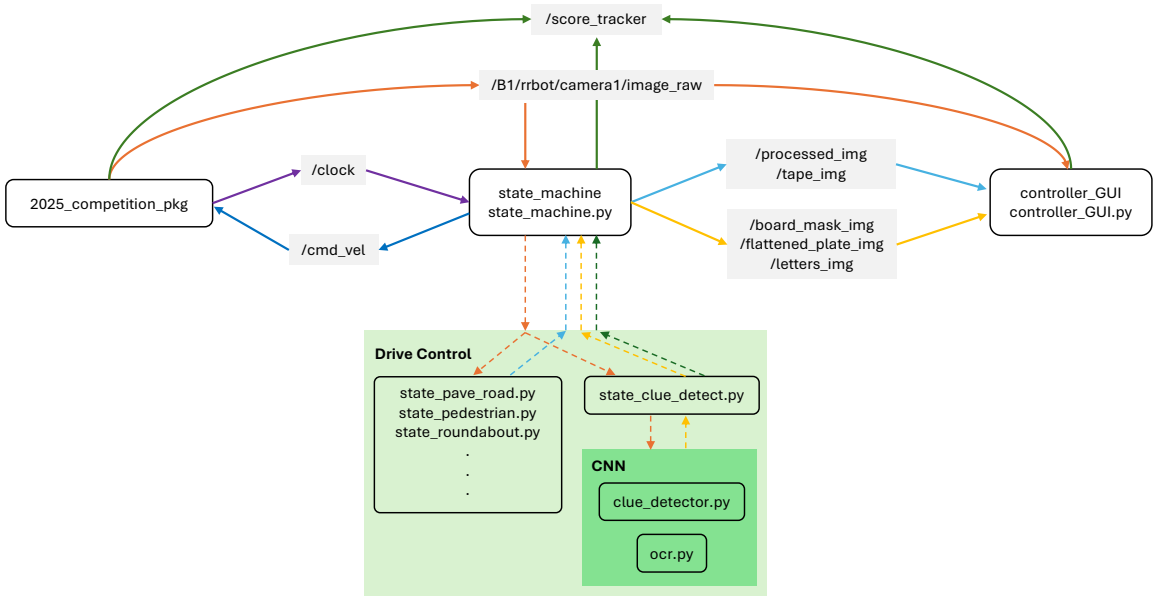

Figure 3 — ROS Node and Topic Diagram

Our robot has 3 main ROS nodes, competition, state machine, and controller GUI nodes (see Controller GUI). Connecting those nodes are the topics, and we made 5 new topics for debugging purposes. `/processed_img` and `/tape_img` are used for driving, and `/board_mask_img`, `/flattened_plate_img`, and `/letters_img` are used for clue detection. All of these images are processed inside each state and are internally referenced to the main `state_machine` node, which then gets published as Image topics.

Our CNN is integrated in a `clue_detect` state instead of running independently. This is done so we can transition to the state once the robot faces to the clue board, avoiding unexpected PID control behavior from sudden clue detection.

I.IV - Finite State Machine Architecture

A Finite State Machine (FSM) was implemented for our robot to manage driving in various surface conditions, detecting obstacles and clue boards. The FSM consists of 16 states, and below is the diagram illustrating the transitions between these states based on sensor inputs.

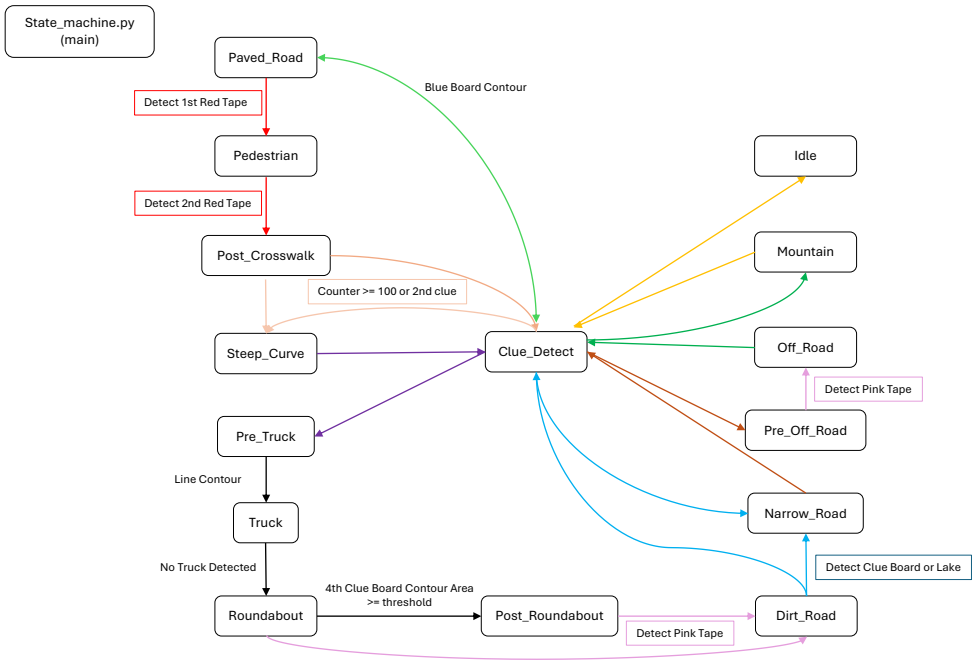

Figure 4 — Finite State Machine Diagram

There are multiple state transition based on clue boards, and those are done by using the clue type detected by our OCR CNN model. In some states, there is a chance of robot missing the clue board, and failing to transition to the desired state. To avoid this, we decided to have backup transition condition if possible. For example, the transition from `Dirt_Road` to `Narrow_Road` can happen either by detecting the clue board or by detecting the contour of the Lake of Despair.

II - Driving System

II.I - Filtering valid contours

For the PID driving, extracting the side lines and filtering out any other noises are crucial. We realized that the ground to sky ratio in the frame was always constant on a flat surface, so we first crop the raw image and only keep the ground portion. Then, we gray-scale and binarize the cropped image, and find the contour using `cv2.findContours(img_bin, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)`.

With high enough binarized threshold, we can filter out most of the small contour in dirt road section and other similar surface condition, but we could not eliminate all of them, so we tried filtering out by contour area.

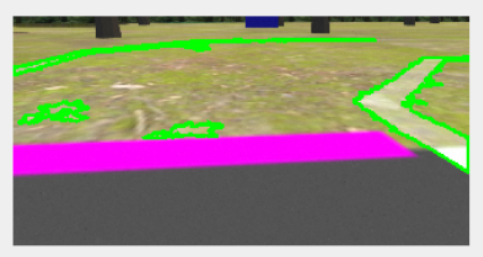

Figure 5 — Contours in dirt road section after filtering by area

As it can be seen, undesired contours still remained after the area filtering. Due to the nature of camera FOV, the contour area gets stretched at the bottom. Therefore, some noises can suddenly pop up at the bottom of the frame even if those were initially filtered out. To resolve this issue, we added another filtering layer using `cv2. boundingRect()`, and only keep the bounding rectangles which touch the sides and is not at the middle bottom of the frame.

Figure 6 — Valid contours

Through multiple layers of filtering, we finally got valid contours like in the image above.

II.II - General PID Algorithm

We generally use the contour's area moment, calculate the coordinates of the center of the lane, and find the error between center coordinate and the center of the frame. However, unlike line-following a single line, we were driving in the middle of two white lines, so the proportional control required an adjustment to the center lane adjustment algorithm for a smoother drive.

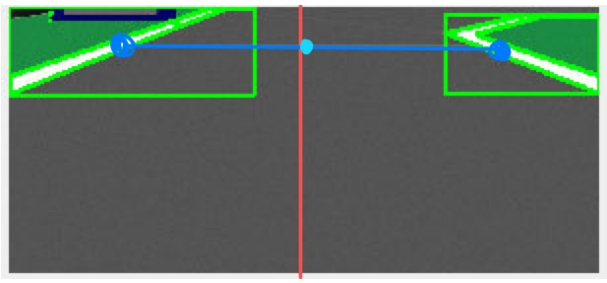

Figure 7 — 2 lanes drive with traditional P-control

In this case, we find the area moment of two lines and average the Cx value to get the lane center which is shown as a light blue point. We calculate the error from the frame center which is shown as a red line, and control the yaw. This works for straight roads with two lines, but fails at a steep curve where there is only one line.

```
error = (frame_width / 2.0) - cx

self.state_machine.move.linear.x  = self.linear_speed
self.state_machine.move.angular.z = self.kp * error
```

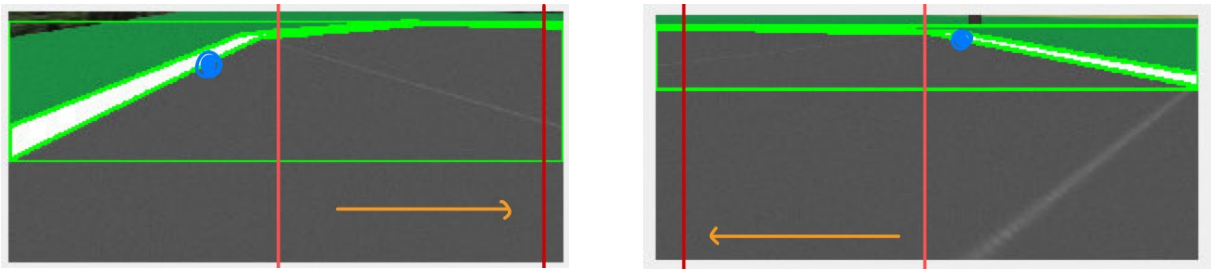

Figure 8 — Steep curve with single line

The light blue dot is the estimated Cx, and it is very close to the frame center (light red). With the same error calculation as above, the error would be way smaller than the required error to fully turn. This is due to the frame center being significantly off from the lane center. To solve this issue, we thought of shifting the frame center more to the side like shown as the dark red line on the images. The closer the contour is to the bottom of the frame, the more shift that's required, so we used the Cy value of the area moment, and tested a proportional relationship between Cy and the magnitude of the shift. The following is the adjusted error calculation code:

```
slope = 2.5

if cx ⩽ frame_width * 0.5:
  slope = -1 * slope

center_shift = slope * cy
error = center_shift + (frame_width / 2.0) - cx
```

The slope value was adjusted in each state through trial and error, which ranged from 2 to 4.

It turned out that a single line driving is much more stable than dual-line driving, so we ended up following a single line with P-control for most of the time. We made the robot move straight while two lines were detected (and the error was within certain range). However, There were some challenges that arose when trying to see clue boards while driving along this path. For details on our attempt to solve these issues see Intentional swerving.

II.III - Roundabout

Another difficulty we encountered were the left turns at the entrance and exit of the roundabout.

For the entrance, we made the robot turn left while exiting the `Truck` state so the robot can start driving clock-wise when it enters the `Roundabout` state.

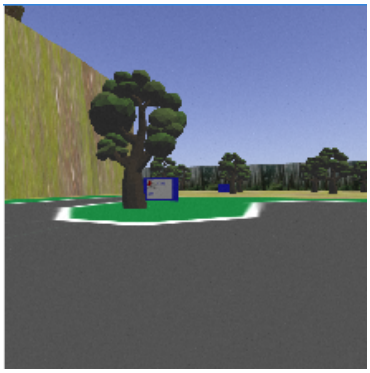

Figure 9 — Using 4th clue board contour area as transition condition

for exiting roundabout, the robot would turn left most of the time because we made the P-control favor left turning in the `Roundabout` state. However, there was a decent chance of robot failing to do so, so we created a `Post_Roundabout` state which rotates left for a short time after entering this state. The transition condition from `Roundabout` state to this state is the contour area of the 4th clue board exceeding a certain threshold area.

II.IV - Off-Road

The strategy for this section was:

1. Position the robot perpendicular to the pink tape using the edge

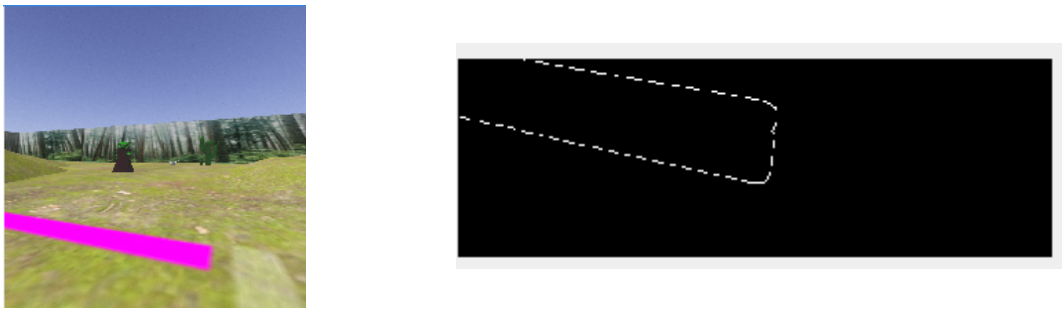

Figure 10 — Positioning robot perpendicular to the pink tape

2. Move straight for some period of time
3. Turn 90 degrees left
4. Move straight and go over the hill
5. Home toward the pink tape next to the tunnel with PID

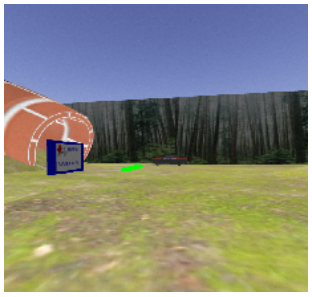

Figure 11 — Homing at the pink tape

6. Move straight until the pink contour area is above a threshold
7. Detect the clue
8. Position the robot perpendicular to the 2nd pink tape.

For details on navigating the mountain which ended up being unused, see Mountain.

III - Obstacle Detection

III.I - Pedestrian & Truck

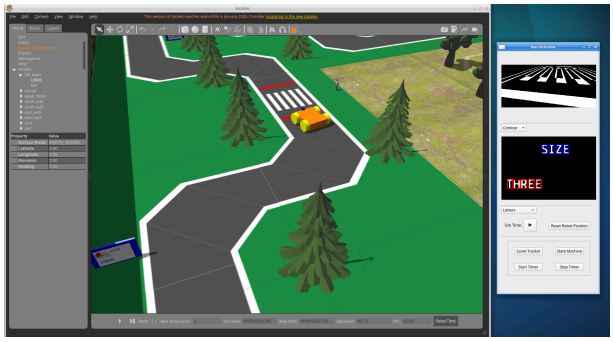

Figure 12 — Pedestrian detection using `cv2.absdiff()`

Our strategy for pedestrian and truck detection was to stop the robot when entering the state, using `cv2.absdiff()` to find the difference in two consecutive frames. If there were no difference for 2 consecutive frames, we would stop driving.

The stop condition is the same during both `Pedestrian` and `Truck` states, but detecting the transition to each state is different.
For the pedestrian detection, we detect the red tape using a red mask, enter `Pedestrian` state, stop the robot, and detect moving objects. We cropped the top part of the frame so that the truck in the background wouldn't be considered as a moving pedestrian.

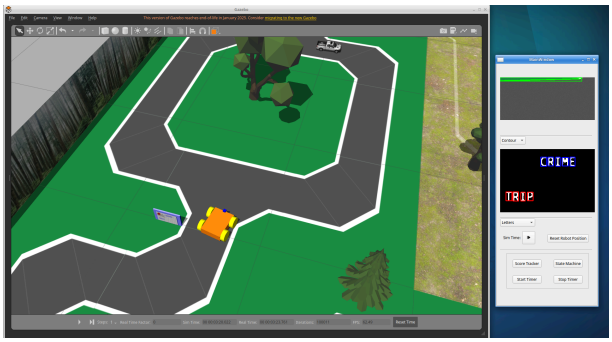

Figure 13 — Flat and wide bounding rectangle at the top of the frame

For the `Truck` state, we made a `Pre_Truck` state which transitions to the `Truck` state when there is a wide and flat bounding rectangle at the top as it is shown in the top right side of this figure. Exiting this state, the robot will turn slightly to the right so the camera can capture the truck better.

III.II - Clue Detection transition Algorithm

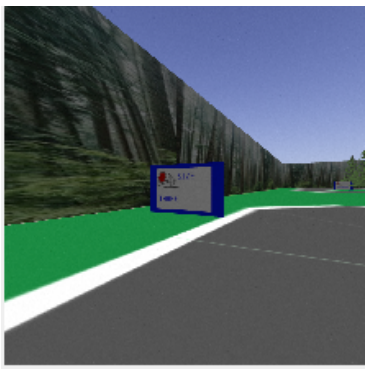

Figure 14 — Homing at the board

Our clue detection CNN runs inside the `Clue_Detect` state only when the robot is facing the board and is close enough. By doing so, we can ensure the entire clue board is in frame and avoid unexpected behavior. In each driving state, we have a clue board detection function which returns true when contours are above a certain minimum area threshold. When that becomes true, the robot transitions to the `Clue_Detect` state. We use PID homing to face towards the board, run the clue detection script, and move the robot closer to the board until the OCR function returns a valid set of letters. Once the letters are detected, the robot sends it to the score tracker, turns away from the clue board, and transitions to the next state depending on the clue type that was detected. We implemented downtime after clue detection because the robot sometimes catches the board again and gets stuck in an endless loop in the `Clue Detect` state.

IV - Clue Detection

IV.I - Plate Extraction

Once we have a clue board in the raw camera feed, we first `create_blue_mask` out of the raw image using `cv2.inRange` (we found the HSV range of [80, 125, 0] to [160, 255, 255] to work the best). We then `extract_board` by taking the largest contour (and bigger than a `min_sign_area`) and then use `cv2.approxPolyDP` to find the four corners of the plate. Using `cv2.findHomography` and `cv2.warpPerspective` we can then perspective transform it to make a flat projection of the clue board. We repeat this process again except masking for white/gray pixels in order to `extract_plate`.

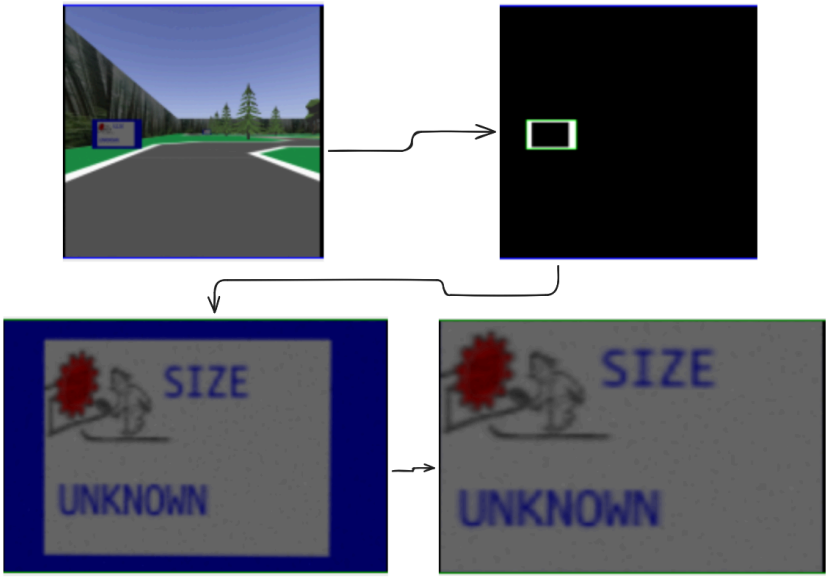

Figure 15 — Plate Extraction Image Processing Pipeline

IV.II - Letter Extraction

With our plate image, we mask for the blue letters and find the bounding boxes of all the contours using `cv2.boundingRect`. The plates always have the two words different rows so we `group_contours_into_rows` by identifying whether a contour is in the upper or lower cluster. As well, the letters are sometimes too close together horizontally and get placed into one bounding box so we `process_single_row` and split up bounding boxes with anomalous widths. The letters are then resized to our OCR CNN model's input dimensions (100, 150) and the input tensor is passed through to identify the clue category and clue value strings. Since the clue categories are always the same, we use `rapidfuzz.process.extractOne` to do fuzzy matching based on Levenshtein Distance in case the OCR makes a mistake. For more details on our CNN model, see CNN Summary.

Figure 16 — Letter Extraction Image Processing Pipeline

IV.III - Optical Character Recognition Convolutional Neural Network

In order to train our CNN, we chose to start our training dataset generation from the plates instead of generating a dataset straight from clean images of the `UbuntuMono-R.tff` font because then artifacts of different letters neighbouring each other would be preserved in the dataset.

We generated clean plates with random two pairs of five character alphanumeric strings. To ensure a uniform dataset, we made sure each character appeared on a plate 10 times for a total of 36 clean plates. Then each clean plate was distorted in 20 different ways with a Keras ImageDataGenerator applying random rotations, zooms, brightnesses, and shears, as well as our custom preprocessing function, `mangle`, which added a random noise map, random gaussian blur, and random small perspective warping. These distorted versions were similar to what the plate looked like after extraction from the Gazebo simulation. In total we had 200 training images per character for a total of 7200 training images. For more details on the training data generation, see CNN Dataset Generation.

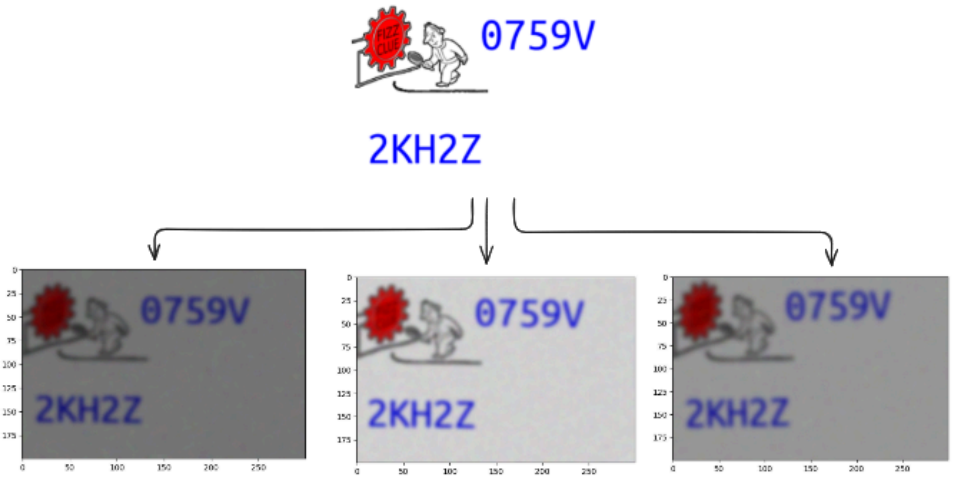

Figure 17 — ImageDataGenerator and `mangle` examples

For training, we used a 30% validation split. We used an initial learning rate of 1×10^{-4} for 100 epochs. However, we also had an `EarlyStopping` callback monitoring validation loss with a patience of 10 epochs and a `ReduceLROnPlateau` callback which would reduce the learning right by a factor of 0.99 when validation loss plateaued for 3 epochs. The CNN was trained for 87/100 epochs and restored the best weights from epoch 77. The final categorical cross entropy loss 0.1740 and the validation loss was 0.1847. For more details on our CNN model, see CNN Summary.

V - Conclusion

V.I - Competition Result

In the competition, our robot achieved an official score of 18 in 240 simulation seconds. We tied for 11th place out of 17 total teams. Unfortunately, our robot was not able to reliably see the clue boards in frame while driving (see Intentional swerving) which caused unexpected bugs to occur. Running our robot again for demonstration purposes after our run, we were able to reach the tunnel and achieve a score of 38 points in 187 simulation seconds. In principle, our robot could to drive to the tunnel without respawning and read every clue along the way and

get 49 points. Unfortunately, the clue boards would not appear in frame reliably enough for this to happen in an actual run.

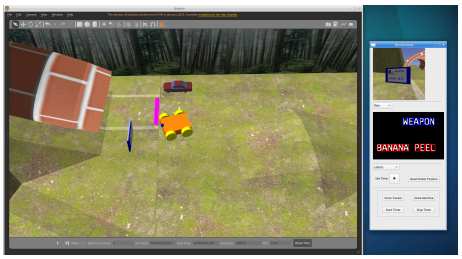

Figure 18 — 7th Clue Detected

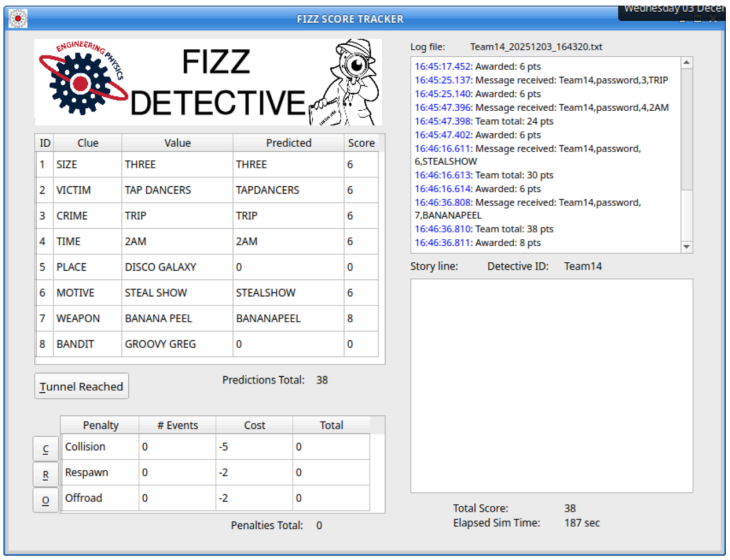

Figure 19 — Unofficial Demonstration Run During Competition

As a team that developed a PID control system, we think it is very hard to implement. Although it is simpler to start with, there are too many edge cases to efficiently design around. This is mainly due to the code complexity required and unavoidable uncertainty at the off-road section arising from 2nd pink tape homing, 2nd pink tape alignment. All the teams that went beyond the tunnel were either using imitation learning or created a drone.

V.II - Unused Ideas

One last minute idea we had in order to try and resolve the issue of clue boards not showing up on our driving line was to change the horizontal FOV of the robot's camera. This would have allowed the robot to see the clue boards more easily. However, we abandonned this idea because it meant we would have to debug our entire driving system through trial and error.

V.III - Future Improvements

If we were to develop this robot further, a key area of improvement would be the driving system. The PID control system required too much hardcoding and manual trial and error to design. If we had started designing with a larger FOV camera, it may have been possible. Most of the teams that could drive around reliabliy used imitation learning. Imitation learning would allow us to essentially hardcode a path that we want the robot to follow ("imitate") that would never have ensured we saw every sign. Although, this might cause issues when integrating as it seems quite a few other teams struggled when integrating their IL driving with other neural networks because of the extra computational load.

VI - Appendix

VI.I - Controller GUI

We developed a controller GUI for monitoring different view types, controlling simulation environment, and launching scripts.

The primary reason why we created this GUI was to increase productivity by centralizing all control on one window instead of having multiple terminal tabs and going on Gazebo to stop or reset robot. This allowed us to easily test changes on VSCode without getting frustrated from moving cursor all over the screen.

GUI Functionalities:

- View different driving camera feeds (Raw, Contour, Tape)
- View different clue detection image processing stages
  (Board Mask, Flattened Plate, Letters)
- Pause/Resume Gazebo simulation
- Reset robot position in Gazebo
- Launch State Machine script
- Launch Score Tracker script
- Start and stop score tracker timer (Debugging purposes)

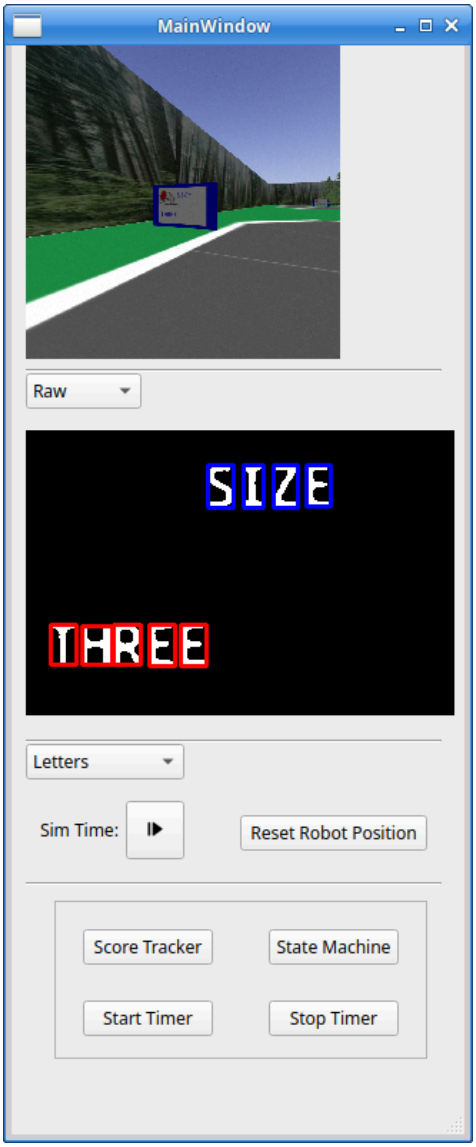

Figure 20 — Controller GUI

VI.II - Intentional swerving

Our PID algorithm followed the outer curver of the road too well, and missed clue boards right after the steep curve because the board was never in the camera frame. We realized this problem too late to be able to simply change the FOV of our camera as that would mean reworking the entire driving system. To solve this problem, we implemented intentional swerving in which the robot would swerve left and right periodically for a specified period. This allowed the robot to cover a wider area and increased the chances of detecting clue boards.

This swerving was implemented in a rather simple way by using a counter that gets incremented every time the state's run function gets called, and changing the "slope" value periodically for mod2 = 0.

```
slope = 1.4

if self.count ⩽ 95 and self.count ⩾ 85 and self.count % 2 == 0:
  slope = 3.7

if cx ⩽ frame_width * 0.5:
  slope = -1 * slope

center_shift = slope * cy
error = center_shift + (frame_width / 2.0) - cx

self.state_machine.move.linear.x  = self.linear_speed
self.state_machine.move.angular.z = self.kp * error
```

This technique was used in “Post_Crosswalk” and “Post_Roundabout” states.

VI.III - Mountain

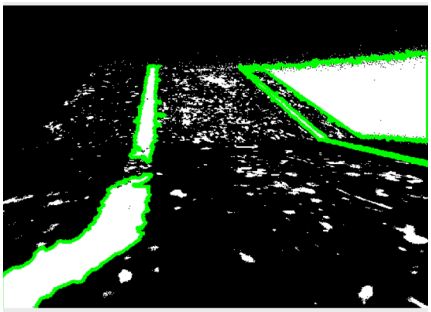

Figure 21 — Sky showing up as a huge contour

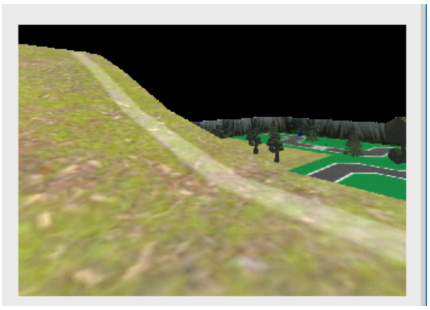

Figure 22 — Sky masked out

The problem with driving up the mountain was the sky showing up as a huge contour. Because it is always at side and will have big contour area, we can’t filter it out using the same method as before. We masked out the pale blue sky by turning the raw image to HSV, create blue mask, and use `cv2.bitwise_not()` to only remove the blue.

VI.IV - CNN Summary

Here is the final CNN design we used for the competition and some details about it’s performance.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	148 × 98 × 32	320
max_pooling2d (MaxPooling2D)	74 × 49 × 32	0
dropout (Dropout)	74 × 49 × 32	0
conv2d_1 (Conv2D)	72 × 47 × 64	18,496
max_pooling2d_1 (MaxPooling2D)	36 × 23 × 64	0
dropout_1 (Dropout)	36 × 23 × 64	0
conv2d_2 (Conv2D)	34 × 21 × 128	73,856
max_pooling2d_2 (MaxPooling2D)	17 × 10 × 128	0
dropout_2 (Dropout)	17 × 10 × 128	0
flatten (Flatten)	21760	0
dense (Dense)	256	5,570,816
dropout_3 (Dropout)	256	0
dense_1 (Dense)	36	9,252

Total parameters: 5,672,740 (21.64 MB)
Trainable parameters: 5,672,740 (21.64 MB)
Non-trainable parameters: 0 (0.00 B)

Best Epoch:

```
Epoch 77/100
      [1m315/315[0m [32m━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m5s[0m 11ms/step - acc: 0.9673
- loss: 0.1740 - val_acc: 0.9671 - val_loss: 0.1847 - learning_rate: 9.2274e-05
```

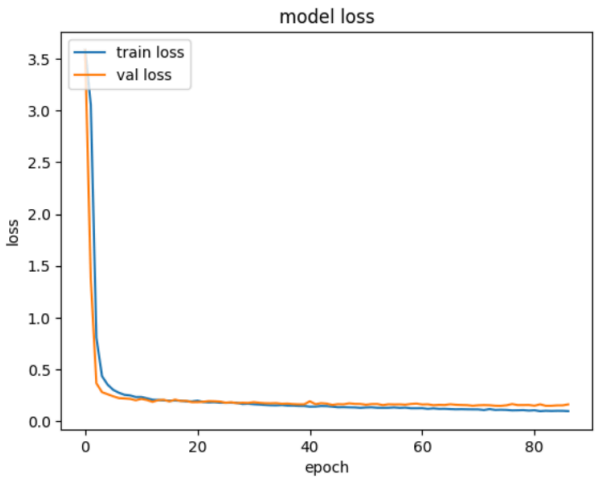

Figure 23 — CNN Model Categorical Cross Entropy Loss

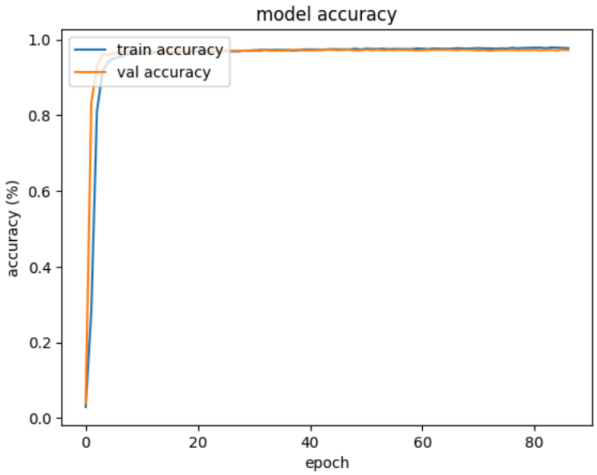

Figure 24 — CNN Model Accuracy

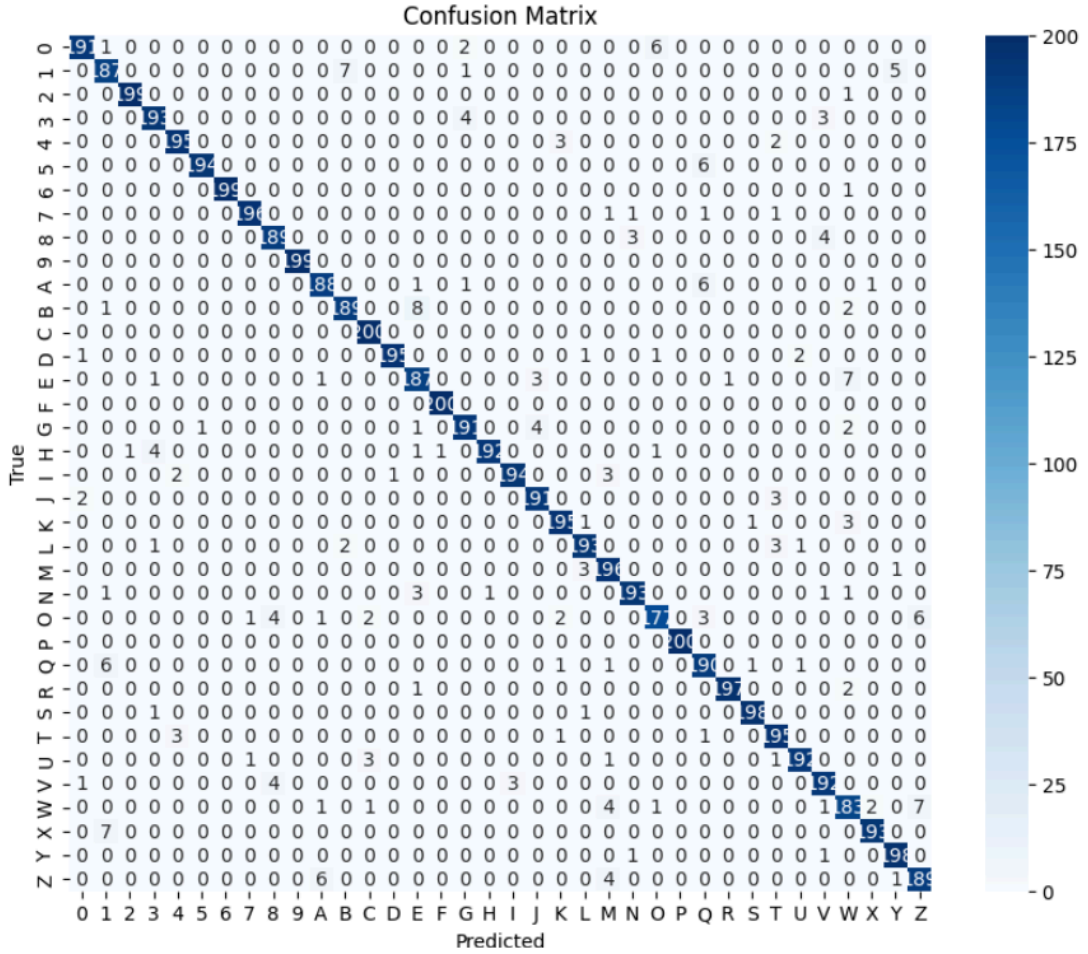

Figure 25 — Confusion Matrix

VI.V - CNN Dataset Generation

To ensure each alphanumeric character was generated 10 times, we used the following code:

```
alphanumeric = ""
for i in range(0, 10):
  alphanumeric += (string.digits + string.ascii_uppercase)

l = list(alphanumeric)
random.shuffle(l)
alphanumeric = ''.join(l)

for i in range(0, NUMBER_OF_PLATES):

    # Pick first word
    key = alphanumeric[:WORD_LEN]
    alphanumeric = alphanumeric[WORD_LEN:]

    # Pick second word
    value = alphanumeric[:WORD_LEN]
    alphanumeric = alphanumeric[WORD_LEN:]

    blank_plate_pil = Image.fromarray(banner_canvas)
    # Get a drawing context
    draw = ImageDraw.Draw(blank_plate_pil)
    font_size = 90
    monospace = ImageFont.truetype("/usr/share/fonts/truetype/ubuntu/UbuntuMono-
R.ttf",
                                   font_size)
    font_color = (255,0,0)
    draw.text((250, 30), key, font_color, font=monospace)
    draw.text((30, 250), value, font_color, font=monospace)
    # Convert back to OpenCV image and save
    populated_banner = np.array(blank_plate_pil)

    # Save image
    cv2.imwrite(os.path.join(SCRIPT_PATH+"unlabelled/",
                                "plate_" + key + value + ".png"), populated_banner)
```

This shuffles a string with 10 of every alphanumeric character and then draws 5 character words onto 36 different plates.

We created 20 different distortions of each clean plate with the following data generator. Random rotations, zooms, brightnesses, shears, noise maps, blurring, perspective warping are applied to the clean plates to make them more realistic.

```
ImageDataGenerator(rotation_range=2, zoom_range=0.05,
                            brightness_range=[0.4, 1.0], shear_range=2,
preprocessing_function=mangle)

def mangle(img,
               noise_std_range=(0.025, 0.05),
               blur_ksize_range=(5, 15),
               perspective_prob=0.25,
               perspective_scale_range=(0.025, 0.05)):
    """
    Randomly add Gaussian noise, Gaussian blur, and perspective transform to a color
```

```
image.

    Parameters:
    - img: input image, assumed uint8 BGR or RGB (0-255)
    - noise_std_range: tuple, min and max standard deviation for Gaussian noise
    - blur_ksize_range: tuple, min and max kernel size (odd integers) for Gaussian
blur
    - perspective_prob: probability of applying perspective transform (0-1)
    - perspective_scale_range: tuple, min and max scale for perspective distortion

    Returns:
    - Augmented image, uint8
    """
    img = img.astype(np.float32) / 255.0  # normalize to 0-1

    # Random Gaussian noise
    std = np.random.uniform(noise_std_range[0], noise_std_range[1])
    noise = np.random.normal(0, std, img.shape)
    img_noisy = img + noise
    img_noisy = np.clip(img_noisy, 0, 1)

    # Random Gaussian blur
    ksize = np.random.randint(blur_ksize_range[0], blur_ksize_range[1] + 1)
    # Make sure kernel size is odd
    if ksize % 2 == 0:
        ksize += 1
    img_blur = cv2.GaussianBlur(img_noisy, (ksize, ksize), 0)

    # Random perspective transform (applied with probability)
    if np.random.random() < perspective_prob:
        h, w = img_blur.shape[:2]

        # Define original corners (clockwise from top-left)
        src_corners = np.array([
            [0, 0],      # top-left
            [w-1, 0],    # top-right
            [w-1, h-1],  # bottom-right
            [0, h-1]     # bottom-left
        ], dtype=np.float32)

        # Randomly perturb corners for perspective effect
        scale = np.random.uniform(perspective_scale_range[0],
perspective_scale_range[1])

        # Generate random offsets for each corner
        dst_corners = src_corners.copy()

        # Top-left corner: shift right and down
        dst_corners[0, 0] += np.random.uniform(0, scale * w)  # x: right
        dst_corners[0, 1] += np.random.uniform(0, scale * h)  # y: down

        # Top-right corner: shift left and down
        dst_corners[1, 0] -= np.random.uniform(0, scale * w)  # x: left
        dst_corners[1, 1] += np.random.uniform(0, scale * h)  # y: down

        # Bottom-right corner: shift left and up
```

```
        dst_corners[2, 0] -= np.random.uniform(0, scale * w)  # x: left
        dst_corners[2, 1] -= np.random.uniform(0, scale * h)  # y: up

        # Bottom-left corner: shift right and up
        dst_corners[3, 0] += np.random.uniform(0, scale * w)  # x: right
        dst_corners[3, 1] -= np.random.uniform(0, scale * h)  # y: up

        # Calculate perspective transform matrix
        M = cv2.getPerspectiveTransform(src_corners, dst_corners)

        # Apply perspective transform
        img_perspective = cv2.warpPerspective(img_blur, M, (w, h),
                                              borderMode=cv2.BORDER_REFLECT)
        img_blur = img_perspective

    return (img_blur * 255).astype(np.uint8)
```