Engineering Physics 353
Machine Learning Project Report

Ella Majkic
Avi Guha

"Fizz Detective"
ENPH 353 Project Report
The University of British Columbia
December 3, 2025

Contents

1 Introduction

This report illustrates our approach in developing the software and machine learning algorithms needed to control an autonomous robot capable of navigating a course while reading clue boards to solve a puzzle for the ENPH 353 course competition.

1.1 Background

The competition task was to develop an autonomous agent that drives through a course composed of various terrains with random NPC obstacles in a ROS Gazebo simulation (Fig. 1.1). The agent's goal was to read random messages ('clues') composed of numbers and letters inscribed in the eight clue boards (e.g. Fig. 1.2) placed throughout the course. For every clue board read correctly and reported to the ROS `/score_tracker` topic, the team would score points, with a maximum possible score of 57 points for a perfect run (all clues reported accurately, no NPC collisions, perfect driving).

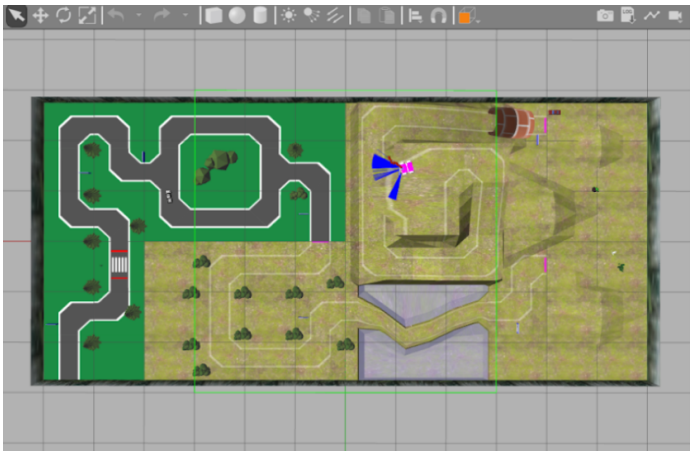

Figure 1.1: Course layout

Figure 1.2: Example clue board

1.2 Contribution split

We split the task into two main categories: 1) line following and navigation; 2) clue board reading and detection. Ella led the development of all clue board modules, while Avi implemented NPC avoidance and navigation through the course. We organized the project by working in separate branches of the same GitHub repository [Click] and merging modules as needed. Branches of interest include:

- `main`: Contains all finalized integrated logic, saved models, and competition code.

- `clue-board-reading`: Includes data generation scripts, data preprocessing, CNN model training, YOLOv8 data, saved YOLOv8 model, and ROS Python classes/nodes implementing clue board logic.
- `line-following`: Line following model development, training scripts, data, and inference nodes.
- `ella-debug, avi-debug`: Separate efforts to increase code efficiency and save compute during model integration.

1.3 Software Architecture

Our overall approach for the competition was to choose a simple strategy that we could implement robustly. Machine learning modules developed to achieve this goal included an imitation learning model for line following, a convolutional neural network (CNN) for reading clue boards, and a YOLOv8 model to detect clue boards within the environment. These models were composed within a ROS environment to execute the competition task. Figure 1.3 shows a high-level overview of the software call chain from initial entry to course completion.

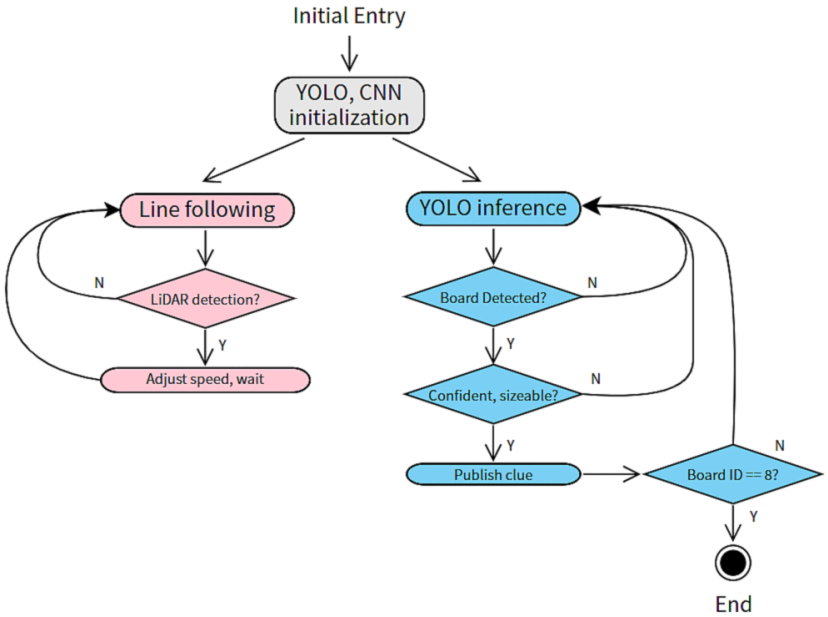

Figure 1.3: High-level software interactions

We will later examine what criteria were established to ensure that the best board images were taken in for reading (such as "confident" and "sizeable" as written in Fig. 1.3).

We did not employ a formal state machine architecture in controlling the logic of the robot agent. We had only one line following imitation learning (IL) model, so there was no need to

register external clues to switch between driving models. Further, the line following module handled all NPC avoidance internally. The `BoardDetector` class (`board_detector.py`) handled all logic for detecting and processing clue boards. Here we implemented a sense of 'state' in a Python dictionary named `board_map`. In `board_map` (Listing 1.1), keys corresponded to pre-set IDs of each clue board. Associated values held a list with the corresponding clue board label and a boolean tracking if this board had been previously reported to `/score_tracker`. This worked because the clue board IDs always corresponded to the same board label, even though the clue text below the label would be random.

```
self.board_map = {
    0: [True,  "START"],      # Placeholder 'board 0' to start timer
    1: [False, "SIZE"],       # Init all course boards as False (unread)
    2: [False, "VICTIM"],
    3: [False, "CRIME"],
    4: [False, "TIME"],
    5: [False, "PLACE"],
    6: [False, "MOTIVE"],
    7: [False, "WEAPON"],
    8: [False, "BANDIT"],
}
```

Listing 1.1: Board Map Dictionary

Fig. 1.4 illustrates the high-level ROS node-topic connections within our software control. ROS nodes (Python scripts/topic subscribers) are boxed, while ROS topics are denoted with a leading '/' character.

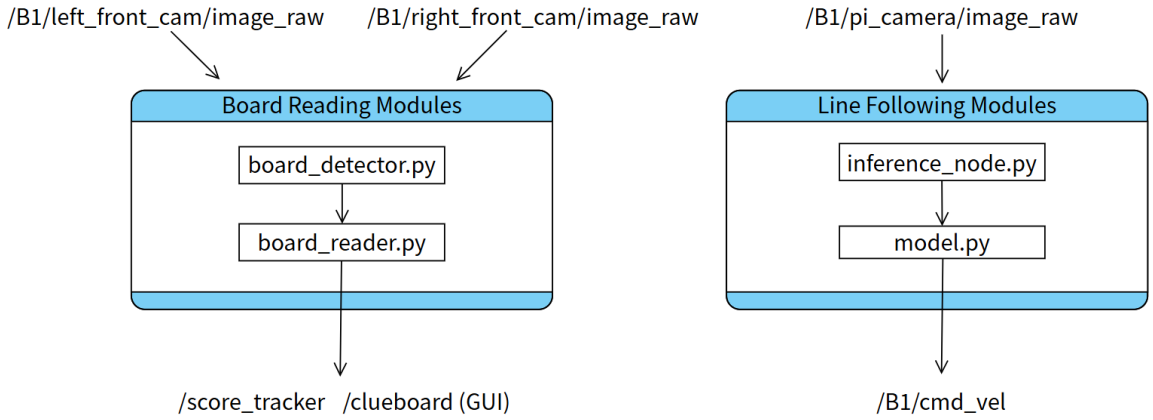

Figure 1.4: ROS node/topic connections

Section 2 decomposes our software into its three constituent modules (line following, board reading, board detection) and outlines the process we followed in developing them.

2 Discussion

The three main pillars of our robot's functionality were a CNN model for board reading, a YOLOv8 model for clue board detection, and an imitation learning model for line following. Here we examine the development and training of each module in depth.

2.1 Clue Board Synthesis

To enable the robot agent to read text written on the course clue boards, we developed and trained a convolutional neural network from scratch using Keras (TensorFlow). Our CNN was trained to recognize 37 classes (see A.1): alphanumeric characters (A-Z, 0-9) and spaces (Fig. 2.1). The goal was to develop a robust CNN that would be able to read text off of images subject to imperfections and distortions within the dynamic course.

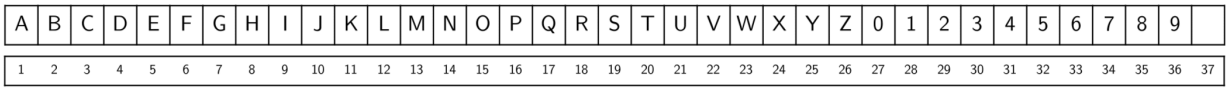

Figure 2.1: One-hot encoding of 37 classes

Note that the `BoardReader` class (`board_reader.py`) is where the CNN model was loaded and board reading was performed. Namely, when a board image was captured and deemed ready for reading, it was sent to `BoardReader`, which is where the logic discussed in this section was implemented.

All work regarding CNN development was done in Google Colab to make use of the T4 GPU available for training. The environment and notebook were linked to Git and can be found within our GitHub repository here (see `353_Clueboard_Reader_NN.ipynb`).

2.1.1 Data Collection

The first step in preparing data for the CNN was to assemble a large quantity of images of the 37 classes of interest. In order to achieve this, we generated 500 sets of 'crimes', namely a .csv file containing pairs of words with the first being one of the possible clue board labels [Size, Time, Place, ...] and the second being a randomly generated word (alphanumeric). Then, perfect clue boards were generated for each pair using the `clue_generator.py` script, mimicking the exact appearance of the clue boards throughout the course.

We then developed three functions which were later instantiated in the `BoardReader` class: `extract_words()`, `characterize_word()`, and `pad_imgs()`. OpenCV was utilized to morphologically dilate foreground (characters) and find bounding boxes using `cv2.findContours()`

within `extract_words()` and `characterize_word()`. These functions were used to extract individual characters from a clue board; their composition is illustrated in Fig. 2.2.

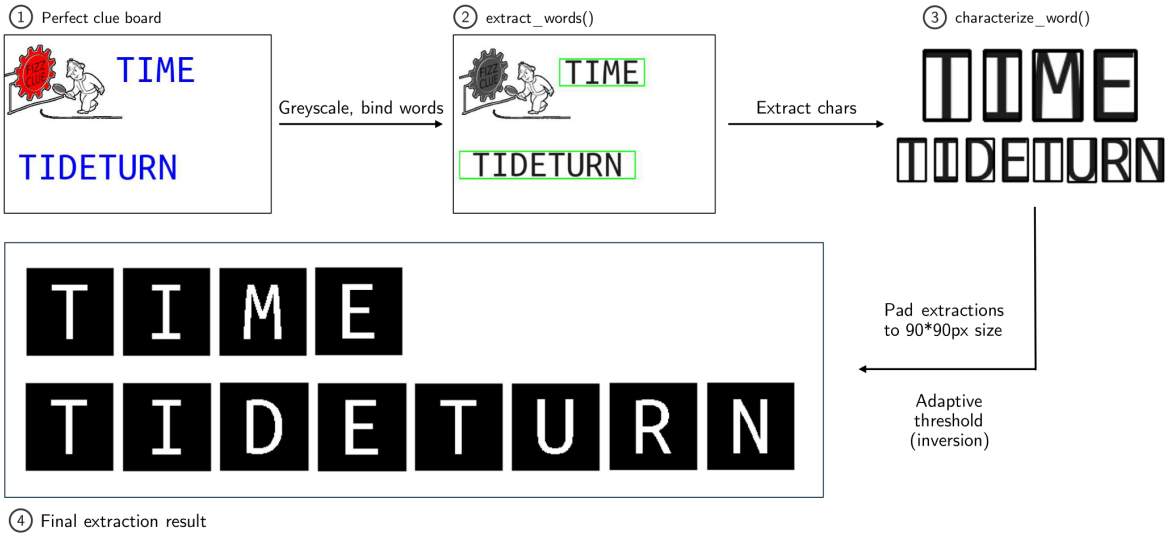

Figure 2.2: Board character extraction pipeline

2.1.2 Data Preprocessing

It was crucial to preprocess the data before assembling all characters for CNN training as real clue board images taken while driving through the course introduced noise, distortion, and lighting changes. We set up a TensorFlow ImageDataGenerator (Listing 2.1) to replicate reasonable distortions on perfect clue board images to add to the training data.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=5,
    zoom_range=0.10,
    brightness_range=[0.5, 1.3],
    shear_range=0.04,
    fill_mode=nearest
)
```

Listing 2.1: DataGen Parameters

For each perfect clue board image in the training data, a distorted image was created by the `ImageDataGenerator`. Examples of images produced by the data generator can be found in the appendix (A.2).

2.1.3 CNN Architecture

After extracting all characters from the perfect and distorted clue boards and assembling them for training (excerpt: A.7), the class distribution split was as shown in Figure A.1. Note that certain classes appeared more frequently because the constant clue board labels [Size, Place, ...] appeared with every randomized word.

Figures A.3, A.4 show a detailed model summary of our finalized CNN architecture. Greyscale 90x90 pixel inputs passed through 6 ReLU-activated Conv2D layers. We added 3 BatchNormalization layers and a 50% Dropout layer to the model to reduce overfitting. Our final Dense layer implemented softmax to deliver a probability result in each class. Listing 2.2 summarizes the training parameters and describes how our model was compiled for the multi-class classification task.

```
epochs = 32 # lean towards less to avoid overfitting
batch_size = 64
learning_rate = 0.001
val_split=0.2 # % of data to be used as validation

model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Listing 2.2: Training parameters and model compilation

2.1.4 Validation Performance/Analysis

After 32 epochs of training, the model converged to 99.1% validation accuracy. Confusion matrices were plotted using `sklearn.metrics` and can be found in Figures A.5 and A.6. The second confusion matrix is for additional training that was done upon adding more numbers to the dataset.

2.2 Clueboard Detection

The next challenge was to determine how to detect clue boards while driving within the environment. This was achieved by training a YOLOv8 model on one class ('clueboard'). Highly confident YOLO inference results were taken in for CNN reading if and only if certain criteria were met. A class named `BoardDetector` was implemented for handling YOLO inference and board detection logic. Once an image was ready for reading, `BoardDetector` would send the board image into `BoardReader`, and publish the CNN's prediction to `/score_tracker`.

2.2.1 Data Collection

Training our YOLO model involved driving through the course and taking many images of the clue boards from various angles, perspectives, and lighting conditions. We took approximately 50 base images. RoboFlow was used to annotate all images of clue boards and prepare a dataset for YOLO training. As was done for the character CNN, we created a data generated image for each perfect image, resulting in a dataset of around 100 images. Data generation was done on RoboFlow and included brightness, shear, and zoom distortions.

2.2.2 YOLOv8 Model

Our YOLO model was trained locally by running the following command:

```
yolo train model=yolov8n.pt data=./roboflow_data/data.yaml epochs=100
    imgsz=640 batch=16 name=clueboards_exp1
```

Listing 2.3: YOLOv8 training parameters

Namely, we trained for 100 epochs with a batch size of 16.

2.2.3 Model Performance

After 100 epochs of training, the YOLO model had a bounding box loss of around 0.1. Figure A.9 displays the model training plots, and validation bounding box results can be seen in Figure A.8.

2.2.4 Dynamic Board Processing

As mentioned, `BoardDetector` handled all YOLO inference while the agent drove through the course. When YOLO would return a bounding box indicating a board was found, three criteria would be checked: `confident`, `aspectratio`, and `sizeable`. We mandated that the YOLO confidence be above 65%, the image have an aspect ratio of at least 1.2, and that the image width be at least 235 px. These parameters were tuned to optimize the quality of images being read without over constraining the system.

When a bounded image of a board passing this criteria was found, it went through one last function before being sent into `BoardReader` for reading. This was `process_board()`, which extracted the inner segment of the clue board for ease of reading. This was achieved by applying a blue mask to the image to filter out the largest connected blue section (outer border of board). Edge contours of inner section were used to fit a polygon to its perimeter `cv2.approxPolyDP`, and this polygon was transformed to a rectangle using homography. Example results are seen in Figure 2.3.

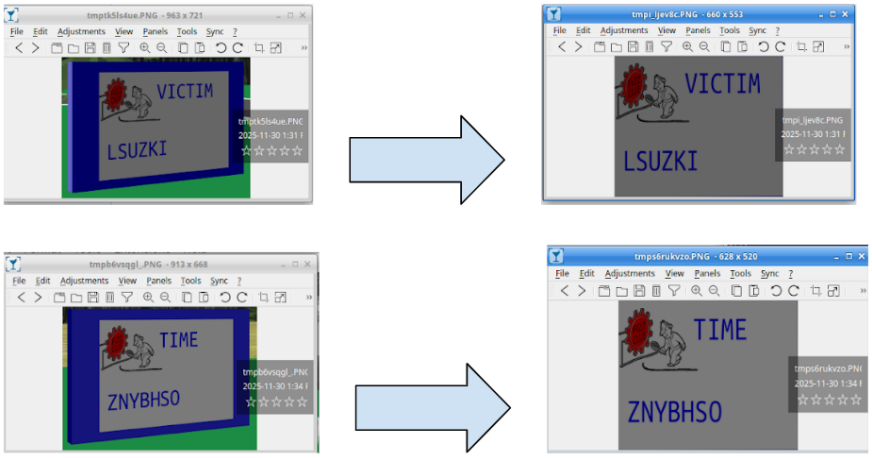

Figure 2.3: Board processing

We also developed a simple GUI (Figure A.10) using PyQt6 to display while the robot was driving to show images being taken in for reading, the words found within the image, and the characterization of the word.

2.3 Driving control

The robot's navigation relies on a behavioral cloning system mapping camera inputs directly to steering commands. This module evolved iteratively from simple imitation to a robust policy handling course correction and obstacles.

2.3.1 Data Collection

High-quality data is essential for effective modeling. We replaced keyboard teleoperation with a PS4 controller, utilizing analog joysticks to record smooth, continuous velocity values necessary for training a regression model. A custom node, `data_collector.py`, synchronized camera frames (`/camera/image_raw`) (see A.11) with velocity commands (`/cmd_vel`) (see A.12). To maintain dataset purity, we used the controller's "triangle" button as a trigger, logging only optimal driving maneuvers and pausing during resets or collisions.

2.3.2 Model Design

We implemented a Convolutional Neural Network (CNN) based on NVIDIA's PilotNet for end-to-end driving. Built in PyTorch, the model comprises two blocks shown in Fig. 2.4:

1. **Feature Extraction**: Input images are normalized to $[0, 1]$. Three convolutional layers (5×5 kernels, 2×2 stride) capture broad geometries, followed by two non-strided layers (3×3 kernels) for fine-grained features.

2. **Control Head**: Flattened features pass through fully connected layers (`nn.Linear`), outputting a single scalar for predicted angular velocity (`angular.z`).

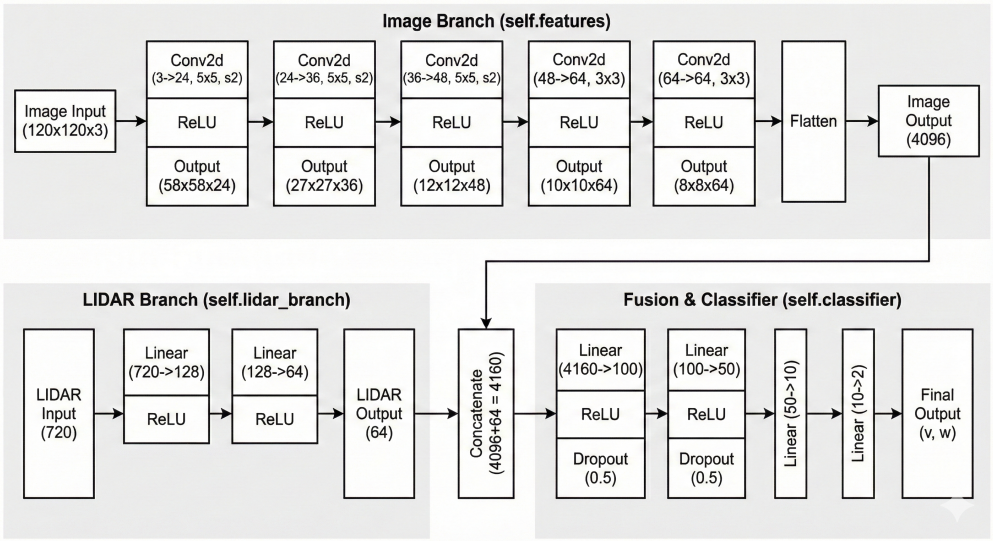

Figure 2.4: Self-Driving Model Design

2.3.3 Fine Tuning

Initial training on "perfect" centerline driving failed to generalize; the model could not recover from minor drifts. We corrected this by collecting a dataset of **recovery behaviors**—driving toward edges and recording sharp corrective steering.

Telemetry analysis showed 70% of frames involved straight driving. We balanced this by discarding 50% of low-steering samples, forcing the network to prioritize cornering. We also removed all stationary data ($\texttt{angular.z} \wedge \texttt{linear.x} \approx 0$) to ensure consistency.

2.3.4 Model Training

We implemented a PyTorch training loop utilizing a `ReduceLROnPlateau` scheduler to halve the learning rate upon validation loss stagnation, facilitating fine-grained convergence. To prevent overfitting, early stopping terminated the process after 20 consecutive epochs of no improvement, while a checkpointing mechanism ensured only the model state corresponding to the lowest validation loss was saved.

2.3.5 Model Performance

The inference node bridges the network and actuators. Predicting both linear and angular velocity proved inaccurate on sharp turns, so we decoupled control. The network predicts **steering** (`angular.z`), while **linear velocity** is fixed at 1.6 m/s. We scaled angular predictions by 1.45× for faster correction, eliminating trajectory stutter.
Lacking visual depth perception, we added a safety layer using 2D LIDAR. If obstacles appear within 20 cm in the forward 180-degree sector, the system halts. Peripheral detections bias steering away from obstacles, adding reactive avoidance to the predictive model.

3 Conclusion

3.1 Competition Performance

Our efforts culminated in a perfect score of 57/57 during competition day, securing second place. Our agent navigated the course flawlessly, adhering to lane boundaries and avoiding all NPC collisions while correctly identifying and publishing every clue board message. We placed second solely on time, trailing the first-place team by approximately 10 seconds.

3.2 Discarded Designs

We initially explored YOLO for dynamic obstacle detection. However, running a secondary neural network induced unacceptable inference latency, compromising control loop stability. We pivoted to LIDAR-based avoidance, enabling robust distance measurement with negligible computational cost.

We also considered Reinforcement Learning (RL) for the driving policy but found the necessary image preprocessing and training convergence times prohibitive. We opted for behavioral cloning to leverage expert demonstrations for rapid, reliable model development.

3.3 Future Improvements

Though our robot performed exceptionally, improvements can always be made! Future development would prioritize consolidating the launch process into a single, unified command for streamlined deployment. Additionally, physically adjusting the robot's center of mass would improve stability, enabling higher velocities without compromising control.

4 References

Various sources were referenced/used to guide the development of our software. Notable such items are listed below.

- https://wiki.ros.org/ [used as a ROS reference sheet]
- https://roboflow.com/ [used to annotate images for YOLO model]
- https://www.kaggle.com/code/bryanb/cnn-for-handwritten-letters-classification [used as a guide for TensorFlow CNN development and model performance visualization]
- https://docs.ultralytics.com/integrations/roboflow/ [documentation regarding RoboFlow YOLO integration]
- https://docs.ultralytics.com/models/yolov8/ [documenation regarding YOLOv8]
- https://github.com/lhzlhz/PilotNet [basis of autonomous driving model]
- ChatGPT was used as a debugging tool throughout the project

A Appendix

A.1 Supplementary Images

See below for additional figures referenced within the text.

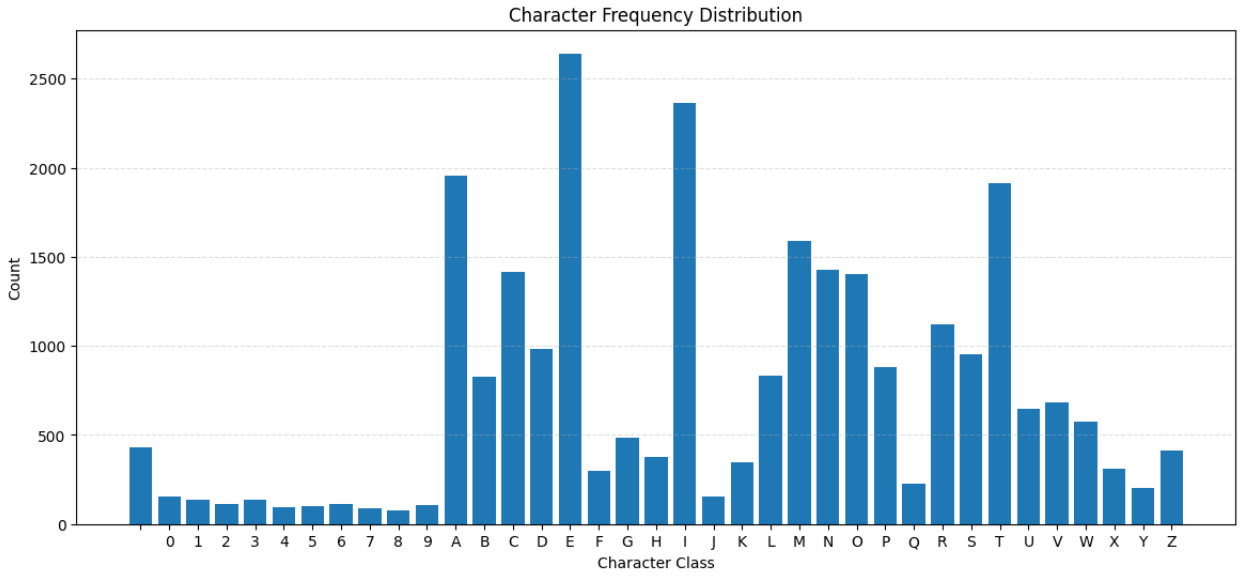

Figure A.1: Class (character) split in training data

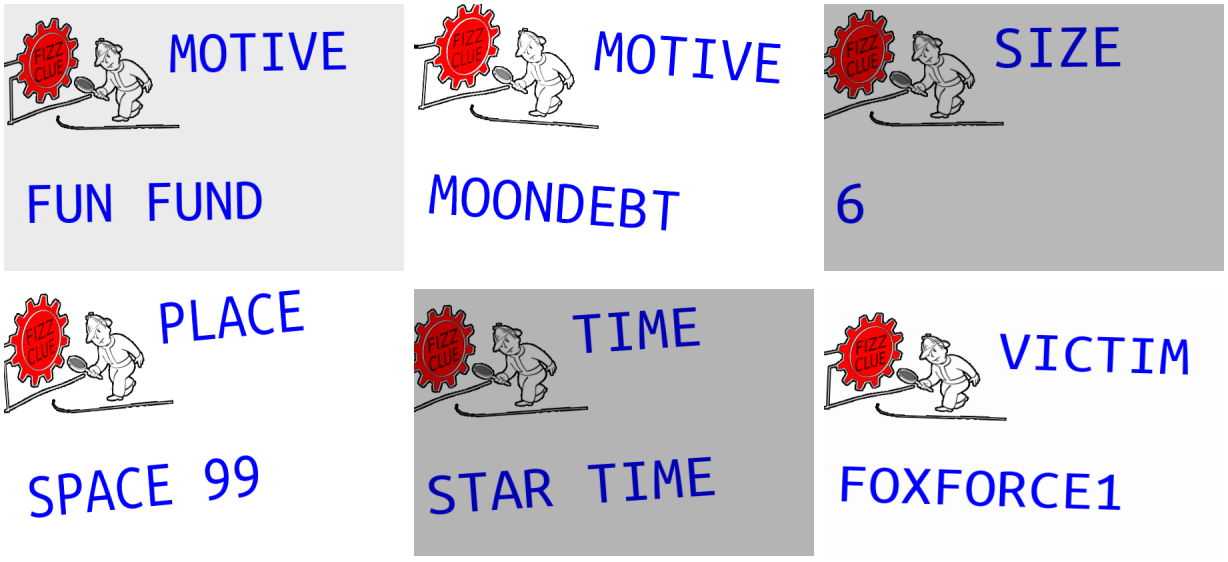

Figure A.2: Examples of boards produced by data generator

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 90, 90, 32)	320
batch_normalization (BatchNormalization)	(None, 90, 90, 32)	128
conv2d_1 (Conv2D)	(None, 90, 90, 32)	9,248
max_pooling2d (MaxPooling2D)	(None, 45, 45, 32)	0
conv2d_2 (Conv2D)	(None, 45, 45, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 45, 45, 64)	256
conv2d_3 (Conv2D)	(None, 45, 45, 64)	36,928
max_pooling2d_1 (MaxPooling2D)	(None, 22, 22, 64)	0
conv2d_4 (Conv2D)	(None, 22, 22, 64)	36,928
batch_normalization_2 (BatchNormalization)	(None, 22, 22, 64)	256
conv2d_5 (Conv2D)	(None, 22, 22, 64)	36,928
max_pooling2d_2 (MaxPooling2D)	(None, 11, 11, 64)	0
flatten (Flatten)	(None, 7744)	0
dense (Dense)	(None, 256)	1,982,720
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 37)	9,509

Total params: 2,131,717 (8.13 MB)
Trainable params: 2,131,397 (8.13 MB)
Non-trainable params: 320 (1.25 KB)

Figure A.3: Character Recognition CNN: Model Summary

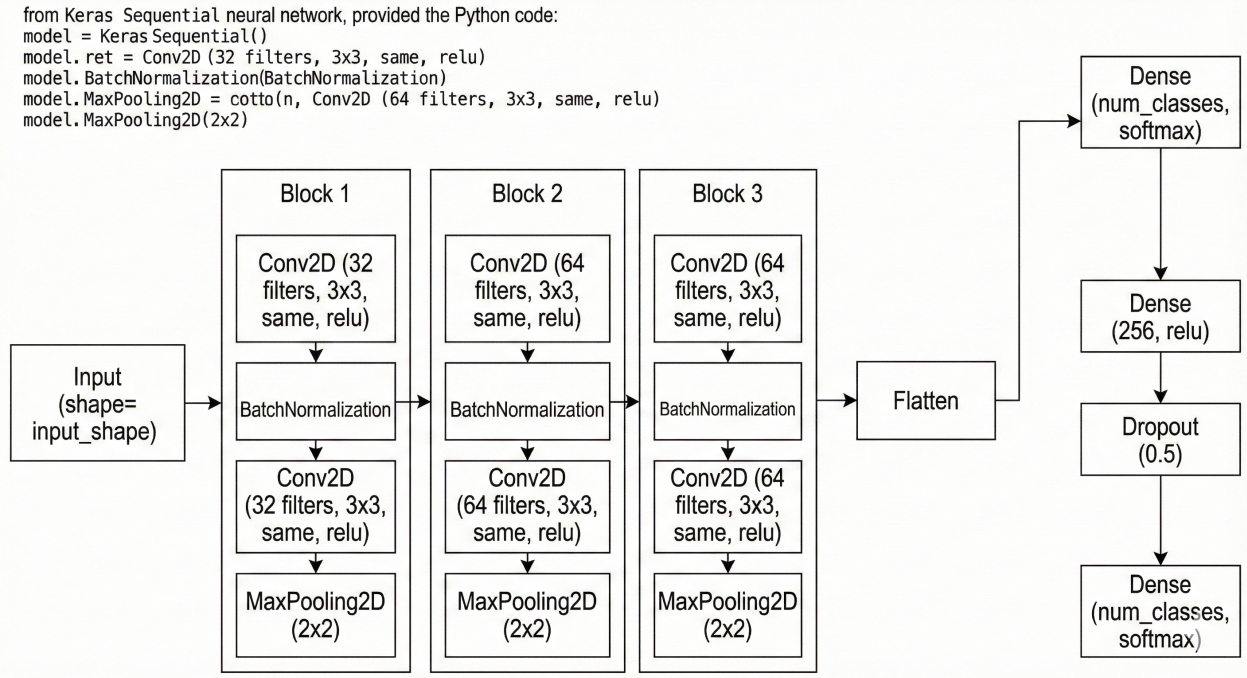

Figure A.4: Character Recognition CNN: Model Summary

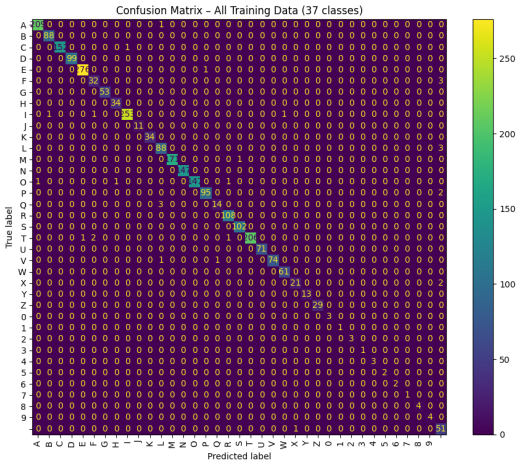

Figure A.5: Confusion matrix

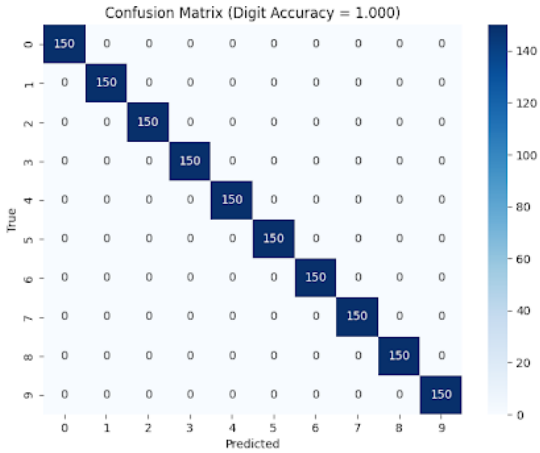

Figure A.6: Additional confusion matrix

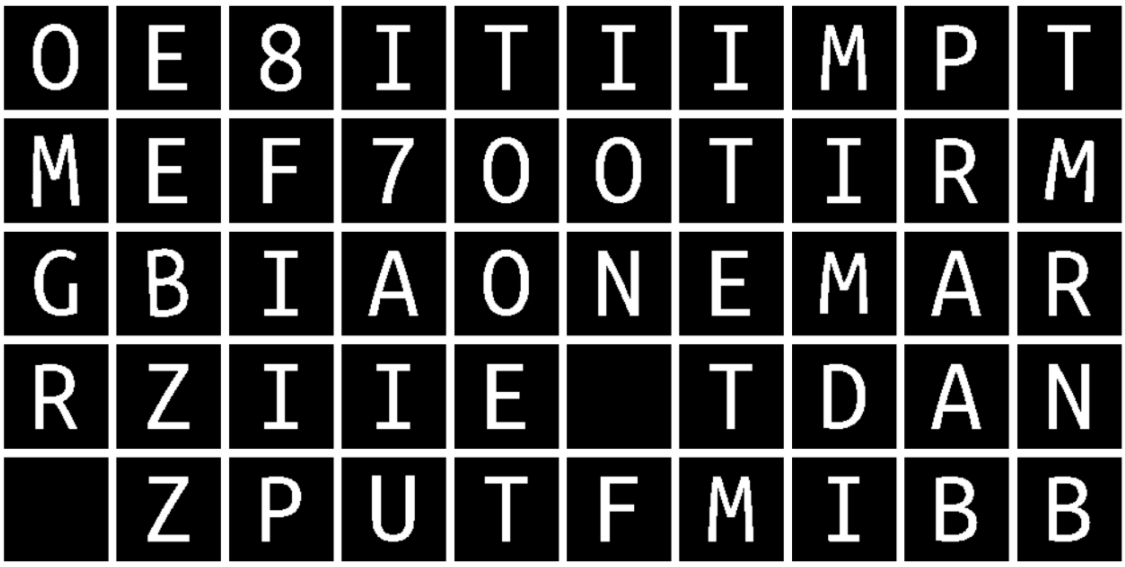

Figure A.7: Sample of training characters

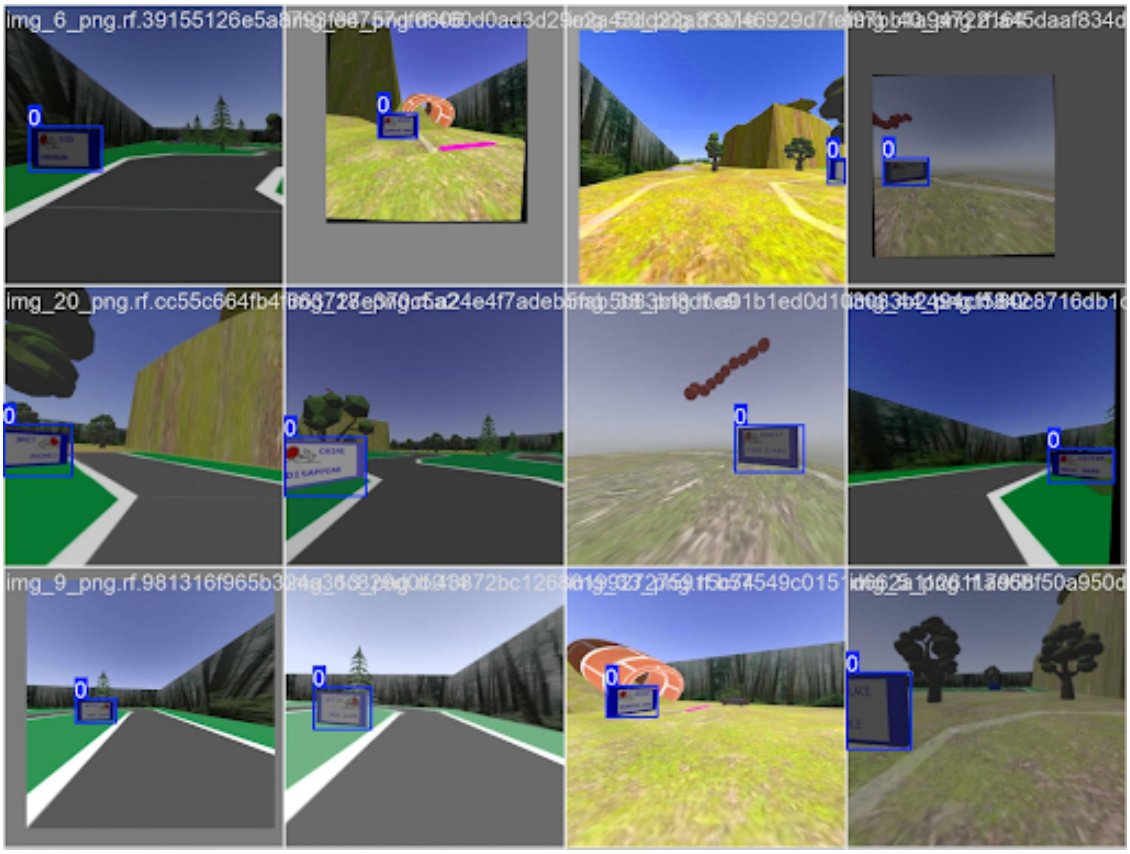

Figure A.8: YOLO bounding boxes

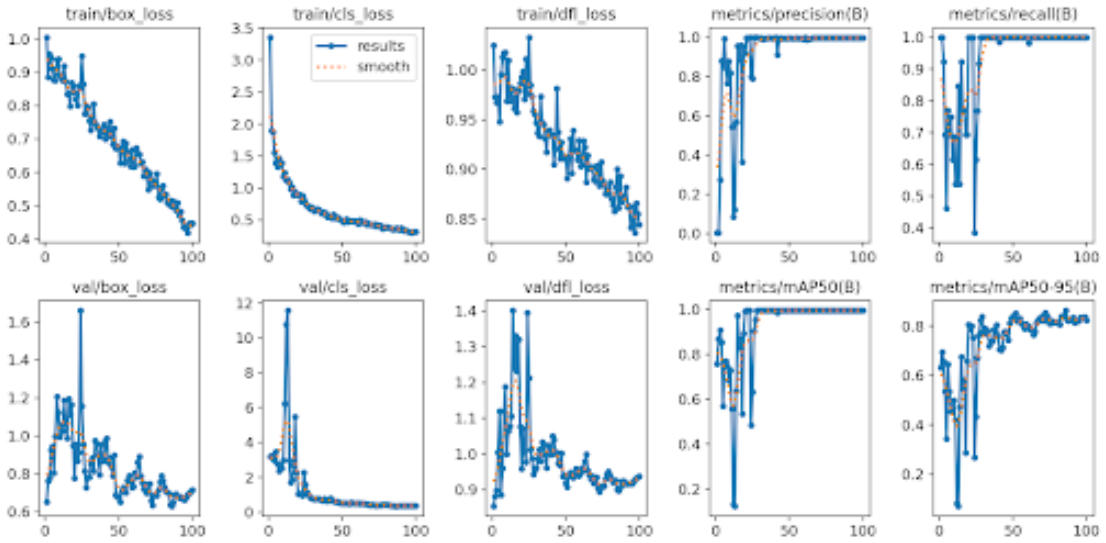

Figure A.9: YOLO training statistics

Figure A.10: Debugging GUI

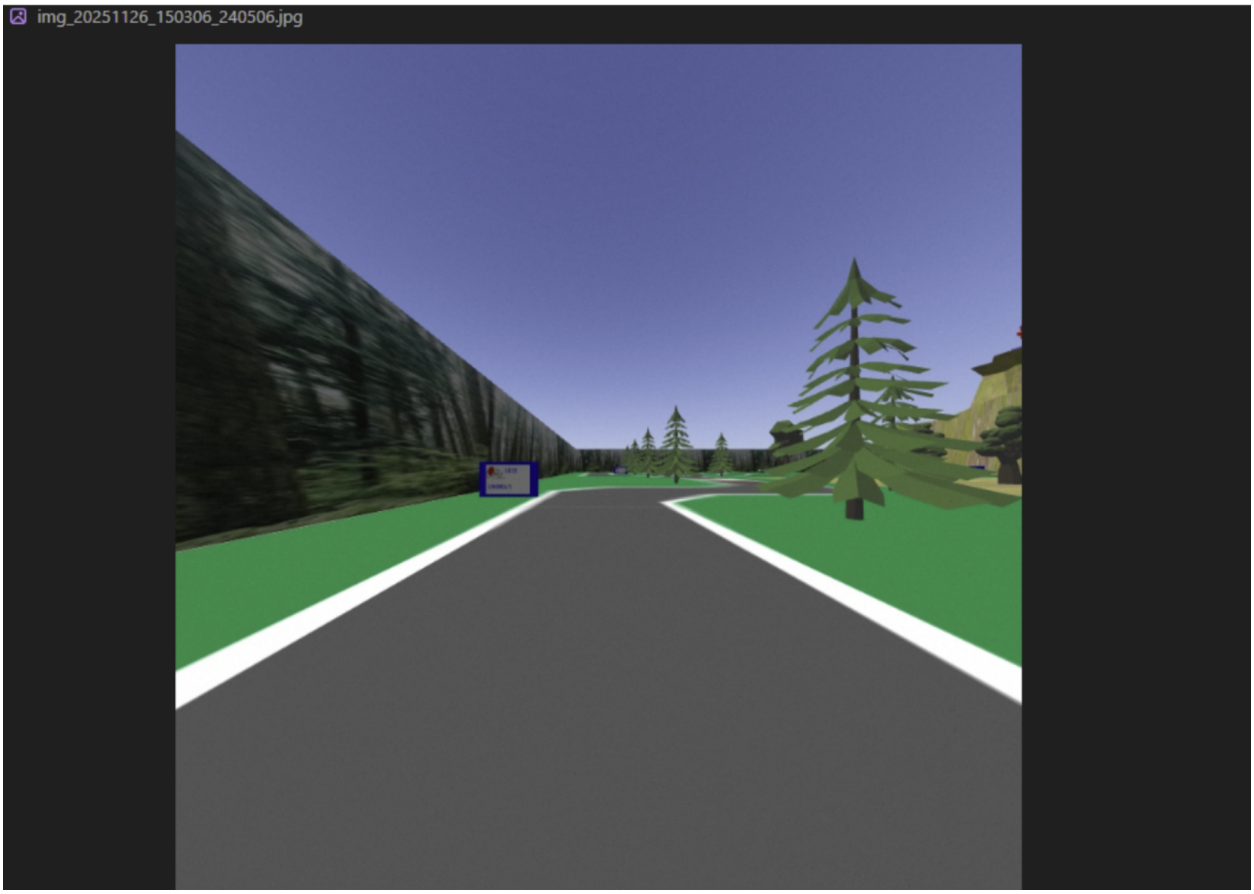

Figure A.11: Representative camera frame captured during data collection

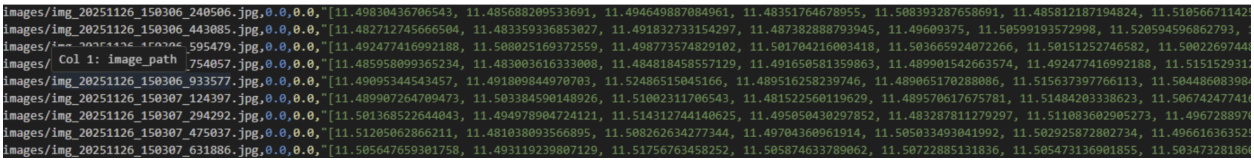

Figure A.12: Sample of Data Saved During Run (`img,v,w,lidar[]`)

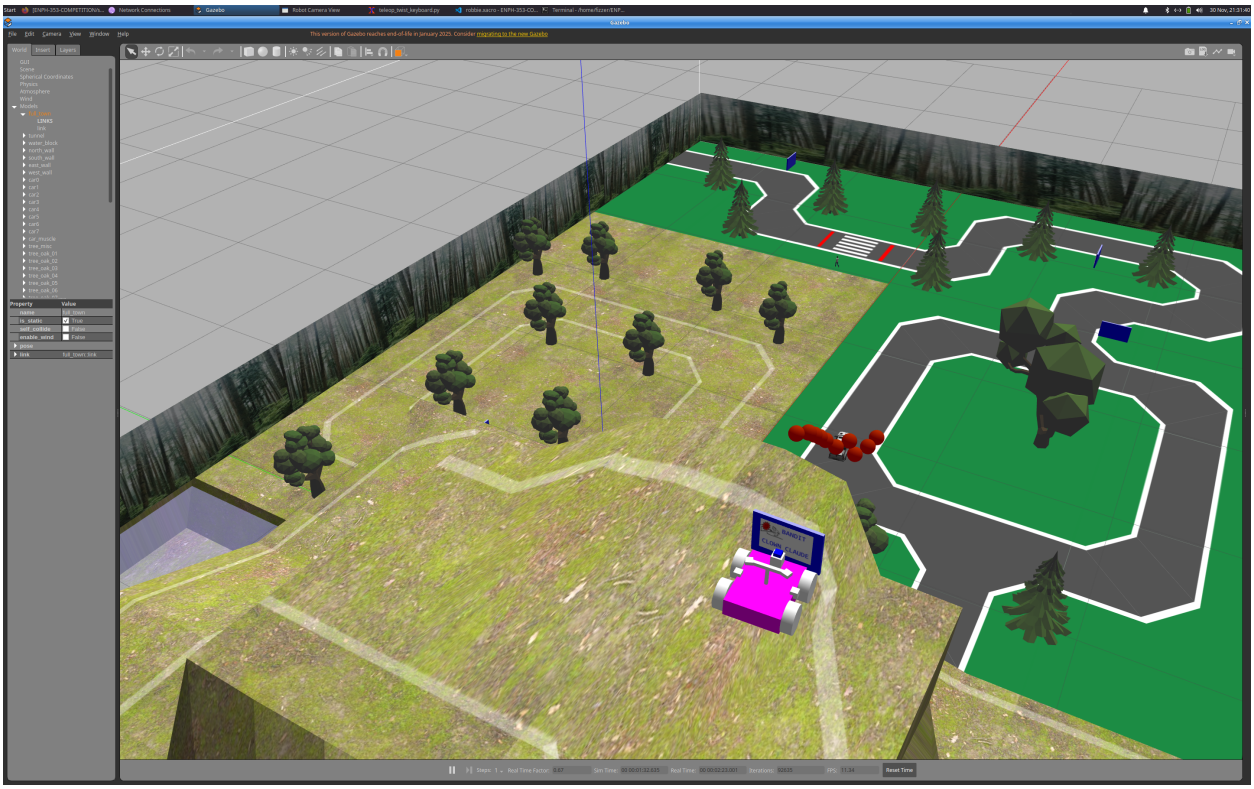

Figure A.13: Scenic image of our robot enjoying the Gazebo view