ENPH 353 Team 1 Final Report

Levi Varsanyi, Lucas Henderson

December 17, 2023

Contents

1 Introduction

1.1 Background

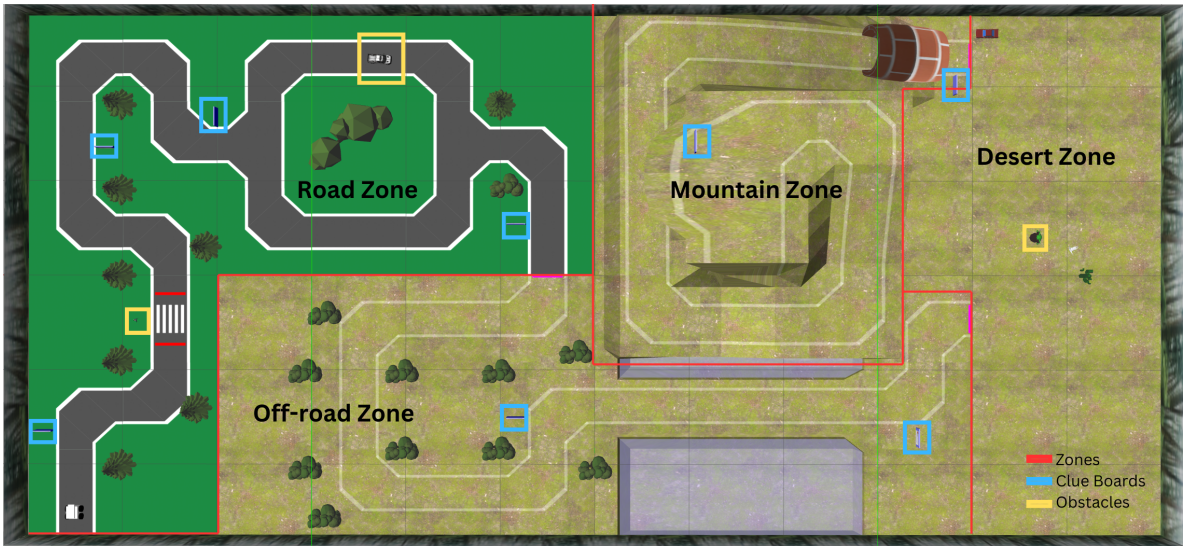

Figure 1: Overhead view of the map

Our goal was to design and test a robot in simulation using Gazebo that could complete one lap of the map shown in Figure 1. It would start at the indicated position, and follow the white lines to the top of the mountain. As it passed the clue boards, it would use a convolutional neural network to recognize the letters on the board and submit the results to the score-tracking system. The robot would not hit any of the obstacles indicated on the map. These were the pedestrian, the truck and Baby Yoda. The robot would also stay within the lines at all times. Failure to stay within the lines, or to avoid the obstacles, would result in point deductions. Once these requirements were met, we would start optimizing the controller to drive faster and more consistently for a shorter lap time.

1.2 Contribution Split

Levi I implemented the clue system, including detecting the clue, getting the letters into the format that could be given to a network, as well as training the network. I also worked on improving the off-road driving, to improve reliability. This involved creating a new algorithm to work with the new line input we had switched to instead of contours.

Lucas I implemented the road driving controller and created the mask and first logic for off-road driving. I also made the desert controller. I worked mainly with classical image processing techniques, including thresholding, masking, edge detection and contour detection to drive the robot.

1.3 Overarching Software Architecture

We used a state machine to tell the robot what action it should take based on several factors. These included location, pedestrian detection, and certain colour thresholds. Then the action would be chosen from a priority queue given the state. The state itself was stored using bitwise OR, which allowed it to have multiple stares stored as a 1 or 0 in a binary string. The action and location together would then be used to decide which function to call. Each different zone had its own driving function. The high-level logic for our robot can be seen in Figure 2.

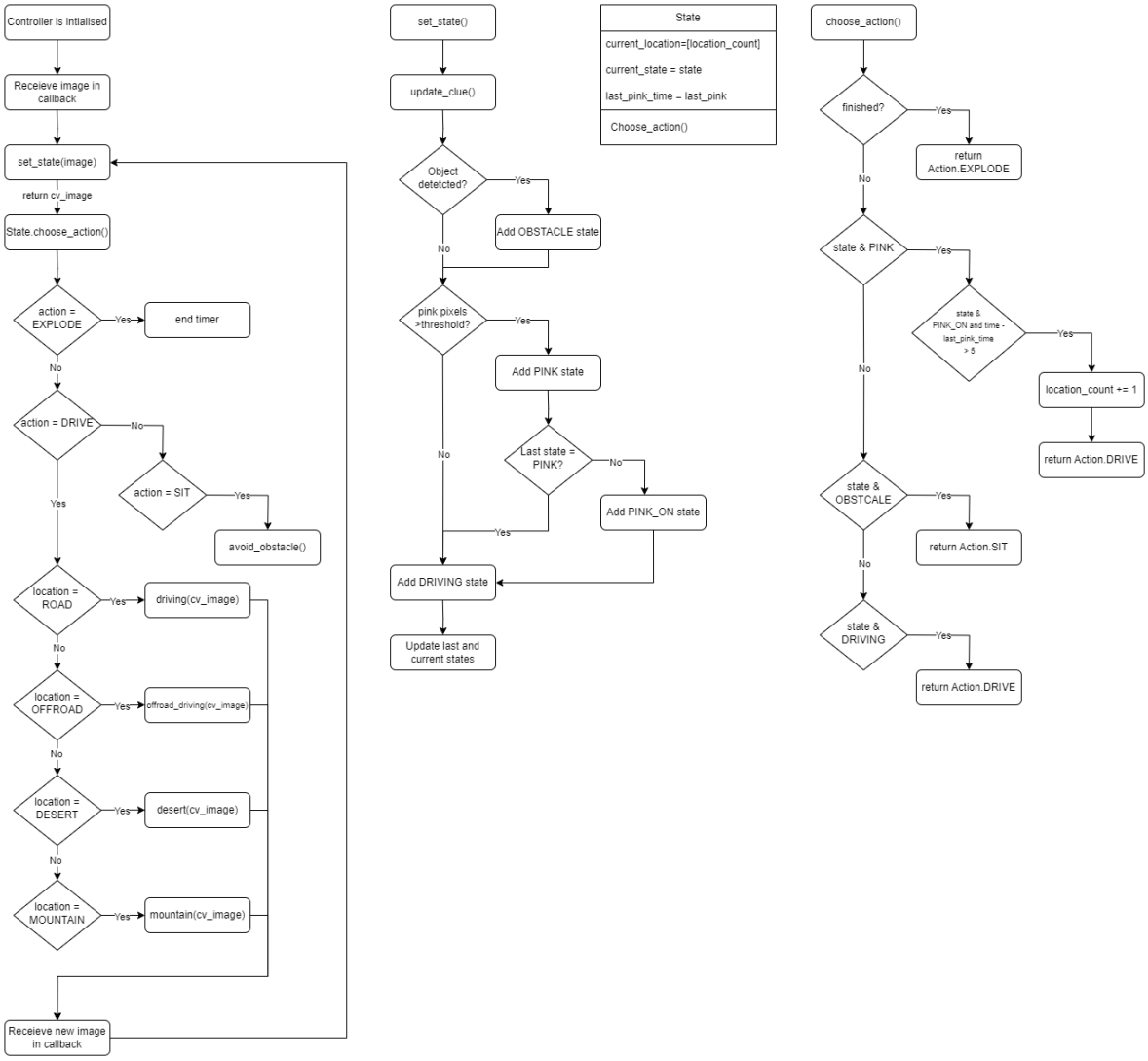

Figure 2: High-level architecture used for our controller

2 Discussion

2.1 Robot Control

2.1.1 Driving Control

Our Driving Controller is split into four different regions of the map, using three different algorithms. We use RGB threshold and contour detection for the road zone, HSV threshold and edge detection for the off-road and mountain zones, and RGB and contour detection mixed with hard coding for the desert zone.

2.1.1.1 Road Zone

We first apply a threshold mask to the raw image we receive from the robot's camera. This uses RGB thresholding with values chosen so that every pixel inside the white lines would be white, and every other pixel would be black. The values for the upper and lower threshold were chosen to allow the red pedestrian lines and the thin grey road lines to be passed though, as these were causing issues during testing such as causing the robot to lose the road and drive into the grass.

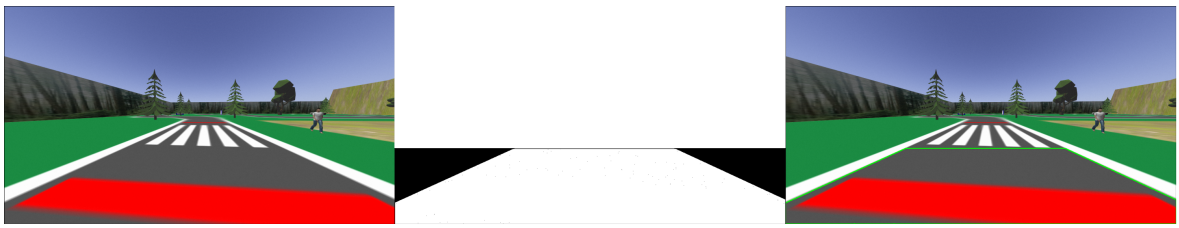

Figure 3: Raw camera feed, the mask we apply, and the outputted contour (in green)

Our algorithm is quite simple following this mask. It finds the largest external contour in the image using `cv2.findcontours` and then checks whether the smallest x and y values are equal to zero. If they are, it assigns the flag `lost_left`. It then calculates the x centroid of the contour using `cv2.moments`. It then calculates the error by taking the difference between half the image width and the centroid. However, if the `lost_left` flag is true, the error is hard coded to be `-LOSS_FACTOR`, a constant chosen through trial and error. This will cause the car to turn left at a set angular velocity. This occurs when it has lost the road on the left side, and is used to bias it to turn left at the roundabout. Lastly, angular velocity is calculated using proportional `KP` and derivative `KD` constants, which were chosen based on trial and error. The linear velocity is calculated by subtracting the difference between the constant `MAX_SPEED` and the error multiplied by `SPEED_DROP`. This is done to limit how fast the robot goes when it is turning. A list of constants can be found in Appendix D.

2.1.1.2 Off-road Zone

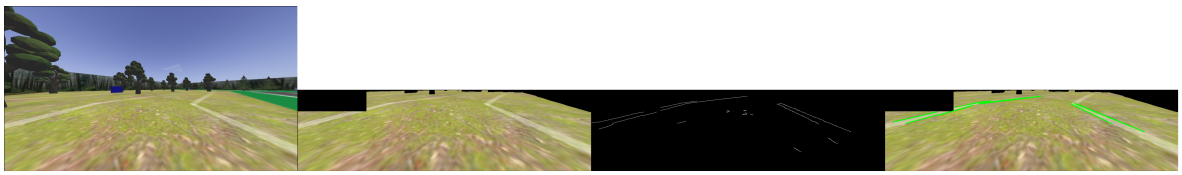

Figure 4: The masks applied to the raw image, the blurred image, Canny edges and the final set of lines (in green).

Image processing We begin by processing the image, which is done in two ways. The first is to create a mask. We start by using HSV thresholding to set everything that is not the dirt road to black, and everything else to white. Then using `cv2.bitwise_and`, we apply the mask to the image, so that all pixels under the mask remain unchanged, and all pixels under black pixels are set to black. This step is repeated, but this time done for pixels that fall within the green colour of the grass. Throughout both these steps, dilation and erosion are used to clean up the masks. Lastly, we crop the image where the highest point in the image is the highest non-black pixel.

An additional mask is added during this process which sets all pixels within a small rectangle on the side of the image to black. This was done to stop the robot from seeing old road lines as this would cause it to miss a turn and is only done in the off-road zone. (see Appendix A)

The second part of the processing is then started. We first apply two rounds of `cv2.bilateralfilter`, with specifically chosen values, obtained through trial and error. This blurs the image but maintains edges, much better than a Gaussian blur.

`cv2.canny` is then run twice, once on the blurred image, and once on the previously generated complete mask. The canny mask is then dilated and overlaid on the canny image, to remove any edges that are touching black. This is so only edges in the centre of the mask are brought forward.

Lastly, `cv2.HoughLinesP` is run twice, one after the other. (see Appendix B) This is done to have more selection with what lines are created. The first pass selects shorter lines, so no road lines are missed. The second pass only selects longer lines, with the hope that after the first pass, all the shorter

lines that made up the road were combined, leaving only long lines. The exact value for the constants used in each pass of `cv2.HoughLinesP` were chosen based on trial and error, as well as intuition for which lines would be most important.

After all the processing steps, the result is a set of lines which ideally match the white lines of the road. These lines are then used in our algorithm to calculate the error.

Algorithm Once the lines have been found, we classified them as left road, right road, or neither.

Classification

To classify them, We would first choose the longest line that fits the required conditions from each side:

- Fully contained in its respective side
- Angled towards the other side

Of these, classification expansion would begin with whichever line was lowest on the image (the lower lines are less likely to be the other side of the road having curved onto that side of the image).

During classification expansion, any unclassified line close enough to any other line in the set of that side is added to the set. This is repeated until no more lines are added.

If the second longest line of that side is classified, then we move on to the second side and repeat the process. Otherwise, before finding the second side's set, the classification expansion process is repeated with the second longest line from the first side. The set with the most number of lines in it is kept as the first side's set. This is done to prevent outliers from being treated as a road, as they were less likely to have as many lines near them. The same is done with the second side if its second-longest line is unclassified.

Error Calculation

Once we had a left road set and a right road set, we could perform the error calculation. The methodology used for this was developed to have either side be enough for PID control. The error for each side was based on the lateral position of the lines and the angle of the lines.

To find the lateral error, an equivalent line was used that went from the bottom-most point of the set to the top-most point. The center of that line was used and compared exponentially to the target position (which was not the center of the screen but offset for each side), which gave us the lateral error. The exponential used here was small (less than two), but it helped the weighting to be more accurate.

The angle error was based on the angle of the set, which was calculated using the average angle of all lines if there weren't very many lines, or as the angle of the equivalent line from above if there were enough lines. This angle was compared (also exponentially) to the target angle for that side, which was slanted inward due to the perspective view of the camera. Furthermore, the angle error was exponentially larger the lower down the lines were, as that meant that a turn was closer, thus turning is more important.

The lateral and angle errors from both sides were added together to give a total error, and this was used in our PID.

This error calculation used lateral error primarily for smaller turns, as it would make small corrections to keep itself in the middle of the road, whereas the angle error would kick in most during sharp turns. This let us go much faster than we had been able to go before, and it would take turns very optimally. Additionally, since the errors of both sides were based on a target position, the car could drive reasonably well with only one line, which helped with improving consistency if the line detection was suboptimal.

2.1.1.3 Desert Zone

The desert zone is much more hard coded in order to make it quicker and avoid obstacles. The general algorithm is therefore fairly straightforward.

Once the robot reaches the second pink line and sees enough pink pixels to go over its threshold, it begins to turn left, until it sees the blue pixels of the clue board. Then it drives toward the pixels, using PID to keep the centroid of the blue in the centre of its frame. Then, once it sees pink again, it changes direction and drives toward the centroid of the pink mask for a certain amount of time (Figure 5).

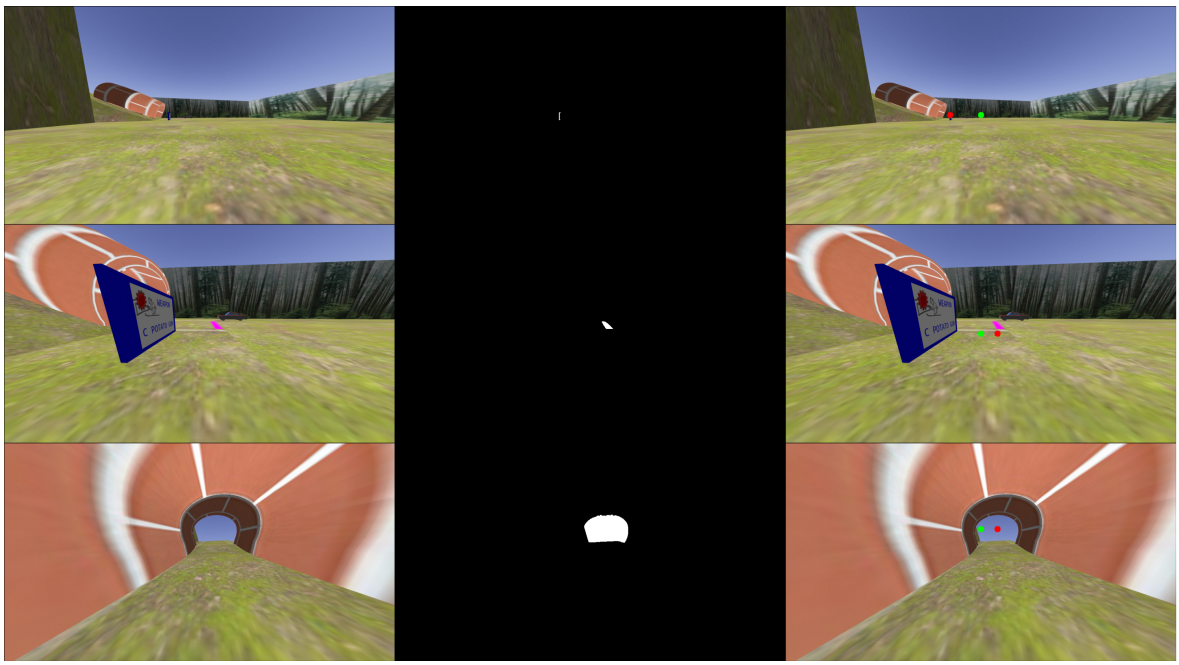

Figure 5: The three primary masks used in the desert zone and the beginning of the mountain zone. The circles show the centroids of the masks (red) and the target positioning for the robot (green).

Then it stops and begins turning toward the middle of the tunnel. Lastly, it drives forward, through the tunnel, using the sky behind the tunnel to guide it.

The timing and thresholds for all the different steps were found through trial and error. Switching between what it was following, based on certain conditions, required many local variables to be made and checked. This made the implementation somewhat clunky but was still very reliable and allowed us to not have to detect Baby Yoda or implement a more complicated driving system.

2.1.1.4 Mountain Zone

The driving controller for the mountain zone was similar to the off-road zone, with some additional states at the beginning to make it through the tunnel.

This zone caused us the most problems during trial runs before the competition, but this was due in part to a lack of testing. If we had more time to tune the specific edge cases for the mountain, it could have performed as well as the off-road zone. It worked perfectly during the competition, but we had to reduce the speed compared to the off-road section.

The mountain zone started by following a mask of the sky that could be seen through the tunnel (Figure 5). This was done while in the tunnel, which was specified by a variable `self.tunnel_state`. The robot was hard-coded to drive straight for a small amount of time to avoid seeing any sky that was outside the tunnel. Then it calculated its error based on the centroid of the sky mask, in the same way as for the road zone. This error was then biased to the left by a small amount, to account for the sky being slightly to the right of the tunnel opening. This biasing was done by adding a small value to the error.

Once the robot was through the tunnel, which was known once the sky mask reached a certain size, it would exit the tunnel state, and the mountain driving function would be called. The function would first check whether the floor mask was less than a given height, as this would mean the robot had reached the top of the mountain. This method could be used because when the robot went up the last incline, the edge of the mountain was gone, so the sky started lower down (Appendix C, Figure 14). When this became true, the robot was pre-programmed to drive straight and then turn left, to avoid clipping itself at the uppermost point of the mountain, which had been a recurring issue (Appendix C, Figure 15).

If it was not at the top of the mountain, the same image processing function was used as in the off-road zone to generate a set of lines defining the road. Then, the same algorithm was used, just with different parameters. The lines were classified and then used to calculate lateral and angle errors. The main difference was that the angle error was not exponential, but linear. The reasoning behind this is that at a slower speed, it did not have to ramp up as drastically, as it had more time to turn. The errors of the left and right lines were then added together and used to set the angular and linear speed, just as was done in the off-road zone. The only other difference between the two functions was the constants that were used. These are outlined in Appendix D.

2.1.2 Obstacle Detection

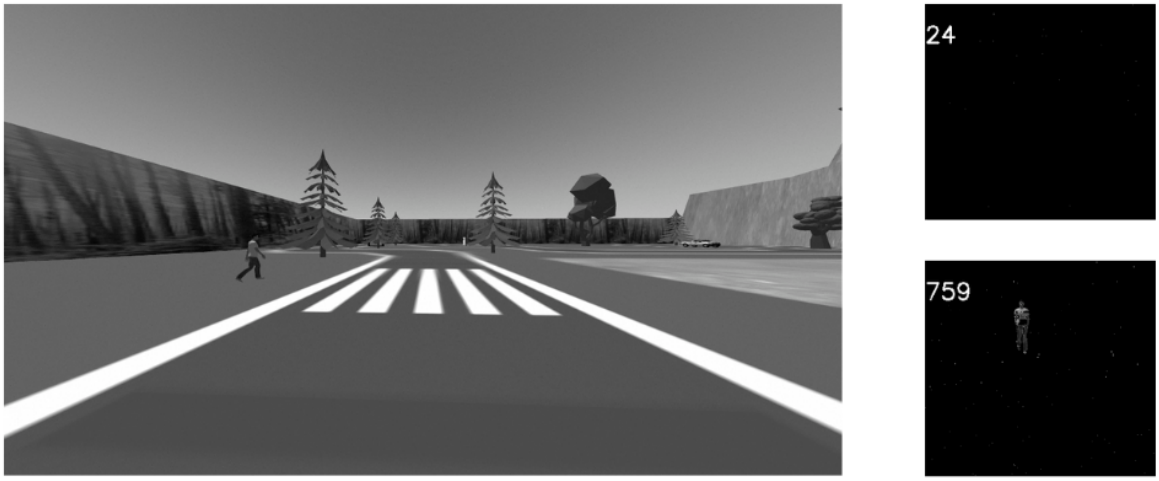

Figure 6: The greyscale image, and then the images after processing. The numbers indicate the non-black pixel counts in their respective images

Due to how we implemented our driving in the desert, we did not have to worry about detecting Baby Yoda. This allowed us to simplify our obstacle detection to solely check it while in the road zone.

Pedestrian and truck detection utilizes the same logic and masks. Doing it this way made it simpler, and did not require keeping track of additional states. A background subtractor was made on initialization and ran continuously through the road section. A mask was made from this, so that road pixels would be set to black. Then if the pixel count ever exceeded a certain threshold, that meant either the pedestrian or truck was in front which would trigger the state to change and the robot would stop and do nothing, until the pixel count once again decreased below the threshold (Figure 6).

The first step was masking all the road lines, including the white, red and grey, to create a continuous smooth grey. This mask was then eroded to remove any additional artifacts that would trigger false positives. Then, the mask was utilized in the background subtractor.

The background subtractor utilized `cv2.backgroundSubtractorMOG2`. This would separate any foreground objects from background objects. When this was run on the mask, it allowed only the pedestrian and truck to display non-zero pixels. Lastly, we crop the image to the bottom middle third, to remove any chance of false positives. The pixel count threshold was chosen to be high enough to not count the artifacts left over from the mask, but low enough to trigger when the pedestrian and truck first entered the frame.

If the function returned true, the state machine would add the obstacle state, which would cause it to return the action `SIT`. This forced the robot to stop moving until the pixel count had once again fallen below the threshold.

2.1.3 Robot Performance

The controller for the road zone was always very consistent. The only issue was when an obstacle appeared. The desert zone was also very consistent once tuned properly. The off-road and mountain zones however were much less consistent.

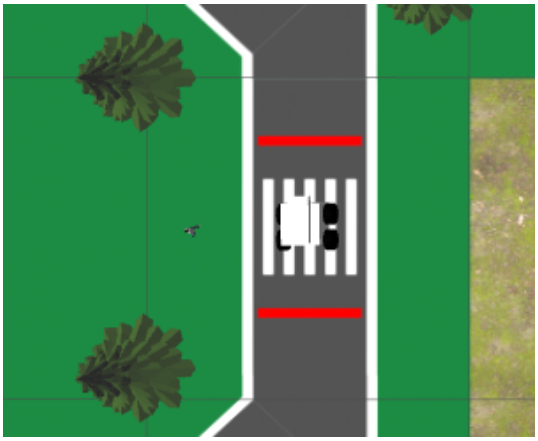

Figure 7: Issue detecting the pedestrian when it approaches laterally

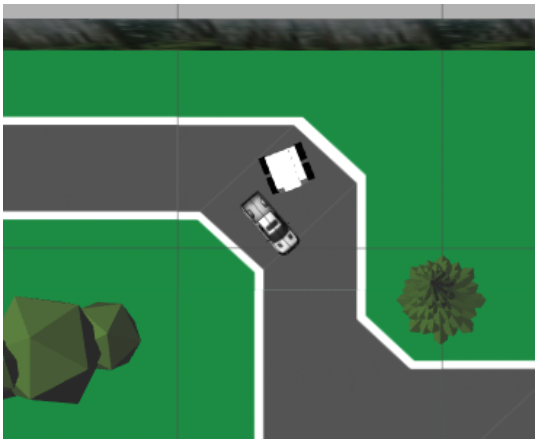

Figure 8: The robot misses the truck and clips it

One major issue with pedestrian detection is due in part to the cropped field of view. If the pedestrian approaches the robot from the side, the robot will not respond and will be hit by the pedestrian (Figure 7). At the roundabout section, if the robot happens to arrive right as the truck passes the entrance, it will follow behind it, stopping and starting as the truck moves in and out of frame. This does not pose a great issue on the straight parts, but when the truck turns, it can briefly leave the robot's field of view, in just the right way, causing the robot to hit the edge of the truck (Figure 8).

As mentioned, once the controller for the desert zone was tuned, it was able to go from the second pink to the end of the tunnel consistently. However, if the robot's max speed is changed, it will not be able to correctly time itself to set up for the tunnel. Additionally, it needs to see the second pink line to know to switch states, which requires the off-road driving to not miss the last turn.

The off-road and mountain sections are the most inconsistent. During our testing, we spent a lot of time tuning the relevant parameters and adjusting the algorithm to stay within the lines. Unfortunately, creating a perfect mask for the dirt was very difficult. Because of this, many artifacts remain which can cause the robot to turn away from the road lines.

Another edge case we dealt with was detecting the 5^{th} clue board (Figure 9). If we just allowed the robot to follow its algorithm, it would always miss the clue board, because it was never fully in frame. To overcome this, whenever the robot sees enough of the clue board, indicating that it is right beside it, it turns left until it sees the entire clue board, and then continues forward, where the algorithm continues driving. This is not a perfect solution, as the robot could see the clue board too early if it had not been driving well before, which would cause it to turn into the middle of the track and be lost.

At the top of the mountain, the robot often got lost because of the steep drop-off from the slope. This was discussed in the mountain zone section. The solution helped at the very top, but we had similar issues in other locations on the mountain. These earlier issues caused the robot to turn into the mountain and get stuck.

Overall, these issues combined with the already more difficult-to-mask terrain made the mountain the worst-performing zone, and the site of the most frequent crashes.

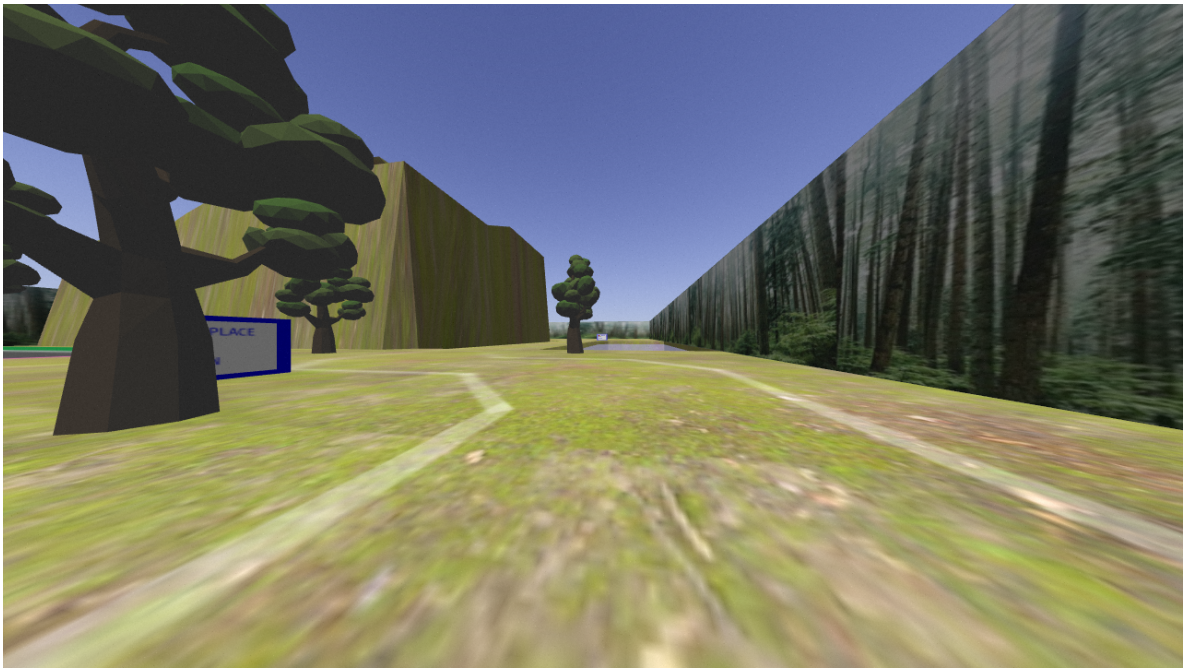

Figure 9: This image shows the 5th clue board, which was obscured by a tree and hard to read

2.2 Clue Plate Recognition

2.2.1 Data Engine

We acquired our data by using the same code as in the plate generator. That code was run with random letters instead of actual clues.

2.2.2 Data Pre-Processing

Each plate was augmented randomly in various ways:

- It was pixelated some amount by decreasing the resolution and then re-increasing it
- The perspective of it was warped to make it slightly diagonal/shifted
- It had a random amount of blur and noise added to it.
- It was darkened by a random amount

These augmentations created a fairly robust dataset that would match the results from the simulation closely.

For simulation images, the first step was to get the clue into a form that resembled one of these plates, as unwarped as possible. This was achieved by finding the contours of a blue mask and choosing the largest contour that had a parent. If the contour area was large enough, we knew this contour was the inside of the clue's blue border. The contour was then approximated to have 4 points, which would be the corners of the clue. `WarpPerspective` was used on this to the same resolution as the images were generated in, giving us the same form as that of our generated data. This was done only on the clue that had the largest area, which was usually the most clear.

After this (and for our generated images), the letters were found using blue contours again. The bounding box of each letter was warped to the same resolution every time, and this was the input into the network.

Sometimes if the image was not clear enough, the contours of multiple letters would combine. To fix this, if any of the bounding boxes were too wide, they would be split so that they would have the correct width.

2.2.3 Neural Network Architecture

Layer (type)	Output Shape	Param #
Conv2D	(None, 128, 78, 16)	160
MaxPooling2D	(None, 64, 39, 16)	0
Conv2D	(None, 62, 37, 64)	9280
MaxPooling2D	(None, 31, 18, 64)	0
Conv2D	(None, 29, 16, 128)	73856
MaxPooling2D	(None, 14, 8, 128)	0
Conv2D	(None, 12, 6, 128)	147584
MaxPooling2D	(None, 6, 3, 128)	0
Flatten	(None, 2304)	0
Dropout	(None, 2304)	0
Dense	(None, 512)	1180160
Dense	(None, 36)	18468
Total params: 1429508 (5.45 MB) **Trainable params: 1429508 (5.45 MB)** **Non-trainable params: 0 (0.00 Byte)**		

Table 1: Model Summary

2.2.4 Training Parameters

Learning Rate	0.0004
Epochs	80
Dataset Size	2000 clues (21000 letters)
Batch size	16
Validation split	20%

Table 2: Training parameters

After training, we quantized the network to a `tflite` model to decrease the size of the model and to improve the speed of the predictions. This was done by giving a representative dataset.

2.2.5 Training and Validation Test Performance

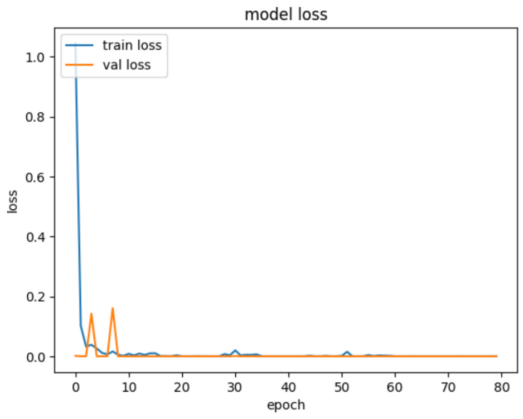

Figure 10: Loss of the model against the epoch

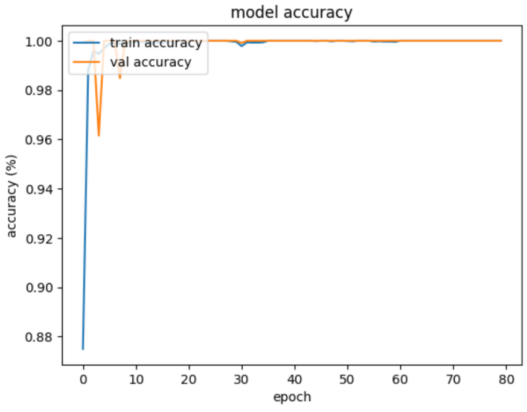

Figure 11: Accuracy of the model against the epoch

The network performed very well after training. Most validation was done in Gazebo, however.

2.2.6 Error Analysis

To test the network, we ran it in the simulation and inputted images from the simulation. We had already developed a script to generate images from within the simulation, as we originally planned to train it with real images.

We also continued to have the network running while testing other systems (driving, obstacle detection, etc.), to ensure that it continued to work. It seemed to always perform well, even at odd angles such as for how we approached the 7^{th} clue board, so we deemed it sufficient. However, we saw that it was not sufficient as it made two mistakes in identifying letters during the competition. We had never seen it make a mistake before, so this was surprising. If we had known this before, we would have worked to make it more robust.

Additionally, the model may have lost too much information during quantization. We did not test the quantized model sufficiently, as that was done sometime after we had completed the network and done our testing. The quantized model had seemed good during passive testing, but some similar-looking letters may have been more difficult for it than we had anticipated. The main reason we had quantized the model was to speed it up since it ran relatively slowly on one of our laptops, but if we had just kept the original model which was fast enough on the better laptop, it may have gotten all predictions correct.

3 Conclusion

3.1 Summary of Robot Performance

Overall, the robot did well in the competition. The robot was able to complete the full course with no major errors in any of the driving zones, and the robot stayed within the lines and avoided all obstacles. This was a good demonstration of the driving controller and resulted in the fastest time to complete the course, without teleporting. Unfortunately, the robot incorrectly guessed the last two clues, mistaking a "U" for an "O", and an "A" for an "R". This resulted in missing 16 points in total. As discussed in the clue recognition section, there were things we could have done (or not done) that may have helped prevent this. If we had gotten those 2 letters correct, we would have won the competition since we were the fastest.

3.2 Methods We Abandoned

We originally tried to use reinforcement learning for the desert zone. This was mostly a fun challenge, to see whether it would be able to do it in time. The robot was spawned at the beginning, near the pink line, with some randomness to account for the uncertainty of the off-road zone. To train it, a reward was given based on the distance to a target path that went from the beginning pink line to the second pink line, as well as its upward speed. We had to subscribe to additional nodes to get this information for training, but once it was trained we would no longer need that node, so it would work for competition.

The model was given a downscaled image that had line detection applied to it. The hope was that the smaller resolution and lines would give it enough information to differentiate different positions at a faster learning rate so that it could converge faster.

This was abandoned after training for some hours, as the model was not converging fast enough. It looked like it may have had promise, but not in time for competition. We were not very surprised, but it was fun to try anyway.

3.3 What We Would Have Done Differently

We would have developed the mountain zone further, to try to eliminate the edge cases with a change in slope. We would have also tried to smooth out the algorithm for both the off-road and mountain zones by more finely tuning the constants.

We would have either tested our quantized model more thoroughly or tried the full model before quantization, as it was fast enough on our better laptop. Lastly, we would have taken more than one prediction of each clue to improve robustness, and if required (and allowed), we would have also compared our prediction to an English dictionary to check for simple mistakes.

Appendix

A Off-road Masks

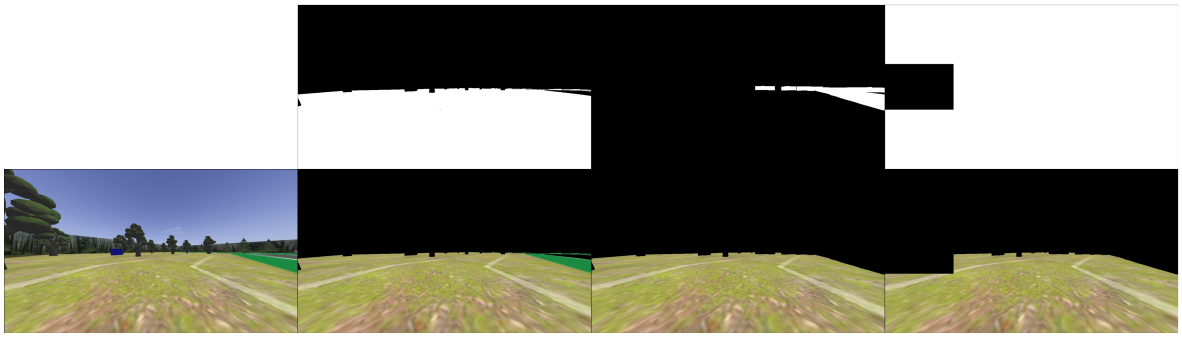

Figure 12: The progression of the masks applied to the original raw image. The generated masks are shown above, and the image image directly below the mask is the previous image with the new mask applied

B Off-road Lines

Figure 13: Lines generated from the masked image in Figure 12. The top shows the edges applied to the blurred raw image. The bottom shows the edges that come from the mask. These are then overlaid to remove the lines that are the same. The last two images show the first and second rounds of line generation.

C Mountain Issues

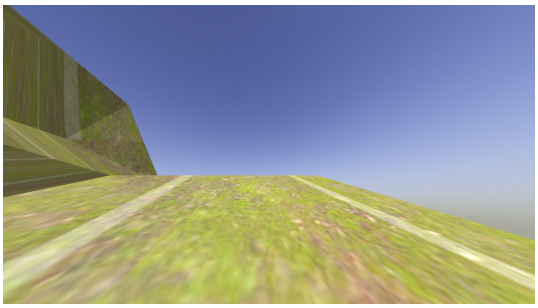

Figure 14: The steep slope on the mountain, and how it could be used to know when the robot was near the top

Figure 15: The problematic spot at the top of the mountain

D Constants

Constant	Function uses	Scope	Purpose
KP	driving() offroad_driving() desert()	GLOBAL	Used to control how much the proportional error affects the angular speed
KD	driving() offroad_driving() desert()	GLOBAL	Used to control how much the derivative error affects the angular speed
OKP	offroad_driving()	GLOBAL	Used to adjust KP for driving in the off-road zone
OKD	offroad_driving()	GLOBAL	Used to adjust KD for driving in the off-road zone
OKX	offroad_driving()	GLOBAL	Used to adjust the lateral error calculated from the lines when driving in the off-road zone
OKY	offroad_driving()	GLOBAL	Used to adjust the angular error calculated from the lines when driving in the off-road zone
MAX_SPEED	driving() offroad_driving() mountain_driving()	GLOBAL	Used to control the maximum possible linear speed, assuming the error is equal to zero
SPEED_DROP	driving() mountain()	GLOBAL	Used to limit the linear speed, similar to KP, when the robot is moving to fast it subtracts an amount proportional to the error
SPEED_DROP_OFFROAD	offroad_driving() mountain_driving()	GLOBAL	Same purpose as SPEED_DROP but used in the off-road zone
SPEED_DROP_MOUNTAIN	mountain()	GLOBAL	Same purpose as SPEED_DROP but used in the mountain zone
Y_MULT_CUTOFF	offroad_driving() mountain_driving()	GLOBAL	Calculated as a certain amount of the image height, it's used in the algorithm for the line classification to weight lines higher up on the screen
Y_MULT_CUTOFF_MOUNTAIN	mountain()	GLOBAL	Same as Y_MULT_CUTOFF but used when in the mountain zone

Table 3: Important constants used

Constant	Function uses	Scope	Purpose
`neutral_angle`	`offroad_driving()`	`offroad_driving`	The is used when calculating the angular error of the lines. This is what is considered the neutral or ideal position for the line.
`neutral_angle`	`mountian_driving()`	`mountain_driving`	This has the same function but has a different value when used in `mountain_driving()`
`neutral_x`	`offroad_driving()` `mountian_driving()`	`offroad_driving()` `mountian_driving()`	This is considered the neutral or ideal lateral location for the line centroid and is used when calculating the error
`angle_exp`	`offroad_driving()`	`offroad_driving()`	This is the exponential applied to the angular error and is used to affect how much the angular error contributes to the total error
`y_mult_intercept`	`offroad_driving()`	`offroad_driving()`	This is used in conjunction with `y_mult_exp` to make the angular error bigger the lower the line is and acts as a constant added to the error
`y_mult_exp`	`offroad_driving()`	`offroad_driving()`	Used to affect how much bigger the angle is the lower the line is

Table 4: Important constants used cont.