Final Challenge Report

Team 13: Five Guys and a Car

Allen “Bowser“ Baranov
James “Toad” Morin
Jian Ming “Yoshi” Chen (Editor)
Remeyn “Princess Peach” Mechura
Ryan “Diddy Kong” Xiao

6.4200/16.405, Robotics: Science and Systems
May 13, 2024

1 Introduction

[Remeyn Mechura]
This report represents the culmination of 5 Guys’ throughout the the semester. In the interest of brevity, it is assumed the reader has already examined all of our previous reports detailing previous accomplishments, and providing a foundation for all work to be examined in this report. There are two parts to the final challenge. The first, named Mario’s Circuit, is a 200 meter race around Johnson Indoor Track at MIT. The objectives are as follows:

1. Drive 200 meters as quickly as possible. Car speed is capped at 4 meters per second, minimum time 50 seconds.
2. Stay within our assigned lane.
3. Do not crash into anyone else’s car.
4. Do not crash our car.

The second, Luigi's Mansion, is a cityscape recreated in the basement of the Stata building. The objectives are as follows:

1. Drive to 3 points to be selected by a TA to recieve a shell.
2. Stop at each point for no less than 5 seconds.
3. Stop at all stop signs and red lights.
4. (Bonus) Return to the starting position with all three shells.

2 Part A: Mario's Circuit

[Jian Ming Chen]

In this section, we will go over our approach to controlling the car so that it stays within a single lane while it traverses around a track. The main challenge posed with this part is first, how can we determine where the lane lines are from the robot's camera view, and second, from the lane lines, determine a point in which our pure pursuit controller can steer towards.

2.1 Technical Approach

[Ryan Xiao]

The first part of our approach deals with how the robot can determine the position of the track lanes that it needs to stay between, given an image from its forward facing camera. As seen in Fig. 1, the image from the camera may contain many different lines, all of which, assuming no filtering is done, could be possible candidates for the track lane. In order filter out the wrong lines to choose the right lanes, we can first make the observation that when the robot is in the middle of a lane, the lane lines are guaranteed to be positioned within a small window, as seen in Fig. 2. Using this observation, we can immediately rule out parts of the image that are not within this image window. Therefore, we only consider points that are not blocked by the black and green triangles.

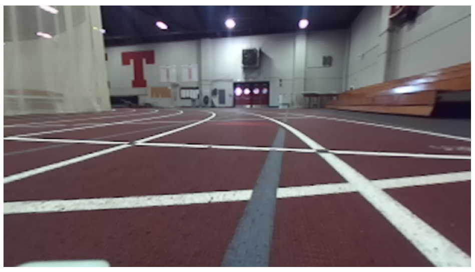


Fig. 1. The robot receives an image from its camera. As seen in the raw image, there are multiple lines that could be confused by the robot as lane lines.

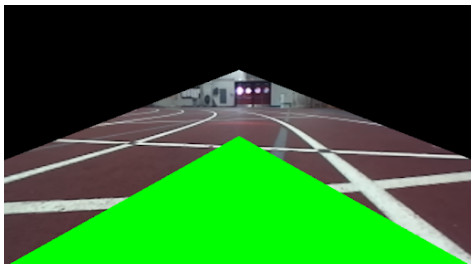


Fig. 2. The lane lines are always visible in the window between the black and green sections. Therefore, only pixels within this window will be considered.

Another important observation is that pixels that are part of the lane lines are distinctively colored white compared to other pixels in the image. To take advantage of this, we first convert the image from rgb to grayscale so that the color of each pixel corresponds to a single number. Then, we can set a threshold value where only pixels with a color value higher than the threshold value will be shown in the outputted image. This results in an image similar to the one shown in Fig. 3. As seen in the images, the track lanes are many pixels thick, so we reduce the number of considered pixels by running the image through edge detection, which preserves the shape of the lines in the image and reduces the number of pixels. This helps both to reduce noise and reduce the number of pixels that need to be computed. An example of an output resulting from the edge detection is given by Fig. 4.

With an image containing only edges, we can now use a useful computer vision technique called a Hough Transform, which works to detect shapes, like lines and circles, in an image. For our purposes, we use it for line detection. In the OpenCV python library, there is a function, `HoughLinesP`, that is able to apply the Hough Transform on our inputted image. This function outputs lines represented by two points on the image coordinate plane, as seen in Fig. 5. It is important to note that on the image coordinate plane, the top left corner is the origin, and traveling right and down corresponds to increasing the x and y coordinates, respectively. From two points, the slope of the line can be calculated. Using trial and error, we determine a range of slopes in which the right and left lane lines could be. Our range is robust where it will function no matter what direction the robot traverses the track. If the slope

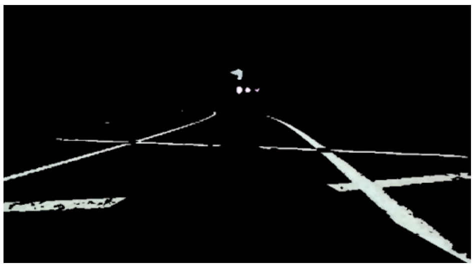


Fig. 3. Lane lines are white, so we only consider pixels that meet a threshold value. This effectively checks how ”white” pixels are.

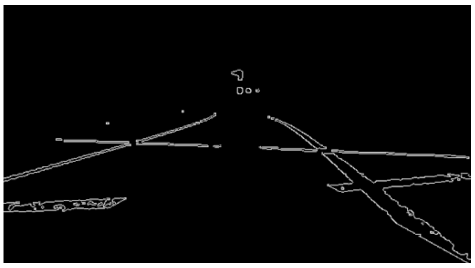


Fig. 4. Lanes are many pixels thick. So, to reduce the number of considered pixels, we run an edge detection, reducing noise and saving computation.

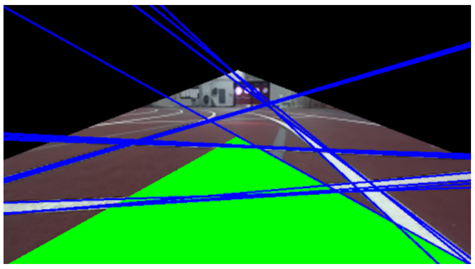


Fig. 5. The robot receives an image from its camera. As seen in the raw image, there are multiple lines that could be confused by the robot as lane lines.

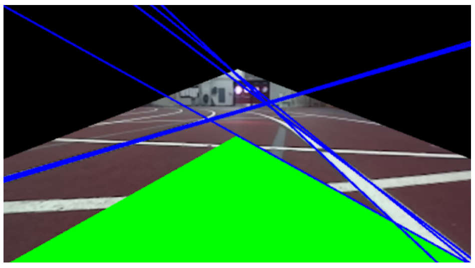


Fig. 6. The lane lines are always visible in the window between the black and green sections. Therefore, only pixels within this window will be considered.

of a line is outside of this range, it can be eliminated from consideration. Further, if the line intersects with the green triangle at the bottom, it will also be removed from consideration. An example of the resulting lines can be seen in Fig. 6. In order to determine positions of the lane lines from the given subset of lines, we find the lines that are closest to the midpoint of the image. We can do this because of the perspective of the camera. As the lanes are further from the camera, they tend to converge around a point near the middle of the image. Therefore, it is a safe bet for us to say that the lane lines are the lines, on the right and left, closest to the the midpoint of the screen, as seen in Fig. 7.

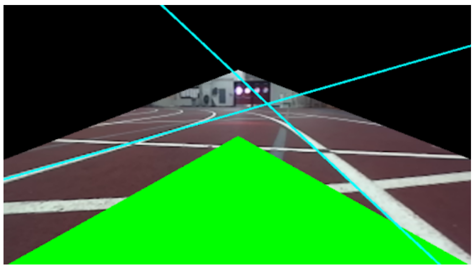


Fig. 7. Lane lines are white, so we only consider pixels that meet a threshold value. This effectively checks how "white" pixels are.

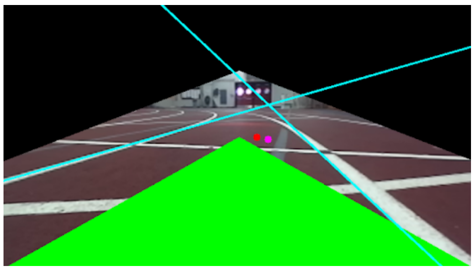


Fig. 8. Lanes are many pixels thick. So, to reduce the number of considered pixels, we run an edge detection, reducing noise and saving computation.

We can now move on to the second part of our approach: how to generate the point that our pure pursuit controller steers toward. To generate this point, we can use our generated lane lines. At the intersection of the generated lane lines, we find the line that bisects the intersection vertically. Then, we can choose a point on the bisecting line at a pre-determined distance from the intersection of the lane lines. As seen in Fig. 8, picking a point on the bisecting line yields a point that is more central in the lane than simply shifting the intersection point downwards. This leads to better lane following performance as the car stays more central within the lane and does not hug either lane. Once a point has been determined in the image coordinate frame, a homography transform can be applied to translate the point from an image coordinate to a coordinate on the map, in the local frame of the robot. This map coordinate becomes our pursuit point, which can be fed into our Lab 6 implementation of the pure pursuit controller.

2.2 Experimental Evaluation

[Jian Ming Chen]

Our experimental evaluation consisted of a visual confirmation over many test runs of the robot, driving around the track at 4 m/s. Most importantly, our car did not leave the lane, no matter which lane it was placed or in what direction. Further, our car did not oscillate much within the lane, meaning we tuned the look-ahead distance of the pure pursuit controller to be just right. This optimized the speed at which we traveled around the track as there were fewer excess movements and our path wasted no time traveling a longer distance than absolutely necessary. These factors combined to allow the robot to achieve a time of 49 seconds around the 200 meter track.

3 Part B: Luigi's Manor

3.1 Technical Approach

[Allen Baranov]

We broke the city driving challenge down into three distinct problems: path planning for the shells, following road laws, and following traffic laws. On competition day, we did not successfully implement the third goal of following traffic laws. However, to tackle the tasks of collecting multiple shells and adhering to road rules together, we refined our path planning approach from lab 6. We began by designing a new map that incorporated the original Stata basement layout with added lanes and further enhancements. Additionally, we upgraded the path planner to handle multiple goals efficiently by organizing them into a queue. This modification allowed for seamless path planning across various objectives. Lastly, we improved the path follower script to pause at each goal, signalling to the path planner that a goal was reached, whereby the planner would generate the next segment of the path. Using our previously existing path planner code made it easier to implement a robust solution to the final challenge.

3.1.1 Multi-Goal Path Planning

We implemented a first in-first out (FIFO) goal queue to keep track of goals. In a FIFO system, each goal is addressed in the order it enters the queue. This strategy was essential for our car to collect three shells in a specific sequence, as it streamlined the transition from one goal to the next without manual intervention. The FIFO system offered a clear and organized method for handling multiple goals, making it easier to predict the car's path and timing. This ensured that all goals were met in the correct order.

Additionally, we implemented logic to manage the process of reaching each goal, ensuring a smooth transition to the next one. We defined "reaching a goal" as the car being within 1 meter of a designated point. Upon reaching a goal, the follower script uses a ROS2 node to send information to the planner script so that it can generate a new trajectory. If the car has not remained at a goal for three seconds, or if a new trajectory has not yet been generated, the follower will issue a drive command of zero, effectively pausing the car's movement. This ensures that the car remains frozen long enough to accurately register the completion of a goal before proceeding.

3.1.2 Right-Sided Lane Following

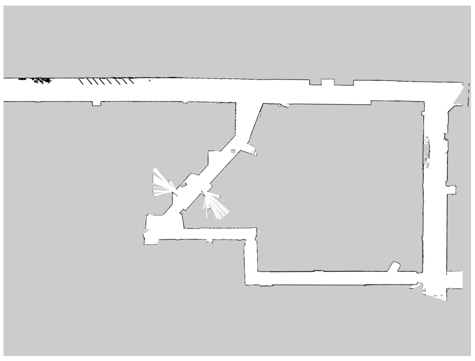


Fig. 9. The original image of the Stata basement lidar scan is the base image we use to generate our new map for city driving.

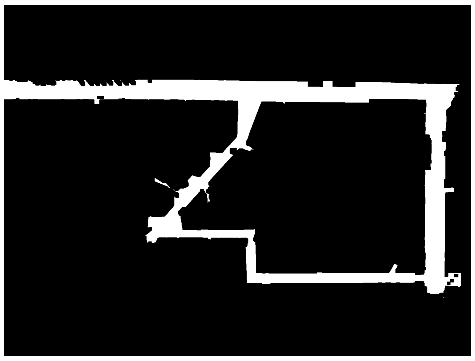


Fig. 10. We perform preprocessing steps as seen previously in the path planning lab, converting to black and white and thickening walls.

By applying image transforms to the Stata basement lidar scan, we generated a new map for path planning that adheres to lane traffic laws. Starting with a copy of the original Stata basement image (Fig. 9), we applied a wall-thickening transform, similar to the method used in the path-planning lab, to prevent the car from clipping walls. We selected a smaller kernel size to avoid creating choke points where an entire hallway cross-section would be marked as an obstacle, which would disrupt our path-planning algorithm. The resulting image, now in black and white (Fig. 10), served as the foundation for our lane-drawing process.

Using staff-provided trajectories, we mapped the lanes onto the image as ob-

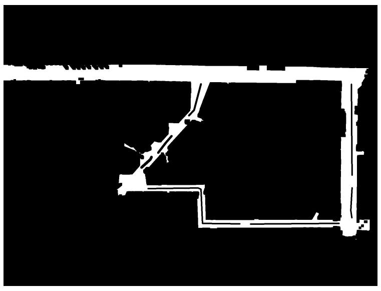


Fig. 11. The added lanes are not entirely indicative of where real-life lanes are, so for the final image, we had to manually draw additional obstacles based on trial and error.

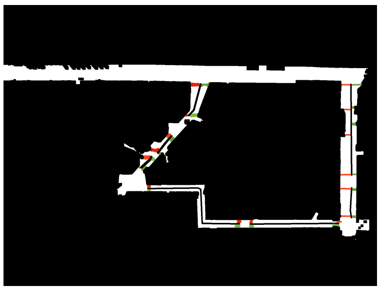


Fig. 12. Only two colors were required to designate lane directions since each color blocks movement in two directions.

stacles. This prevented the car from creating paths that cross over the lanes. Each lane segment was further colored at its entrance and exit to indicate permissible directions: green regions prohibited downward or leftward movement, while red regions restricted upward or rightward movement. Lastly, additional obstacles were drawn manually to assist the car in maneuvering through the lanes efficiently. The result was a map that could be used to effectively path plan and path follow (See Fig. 11).

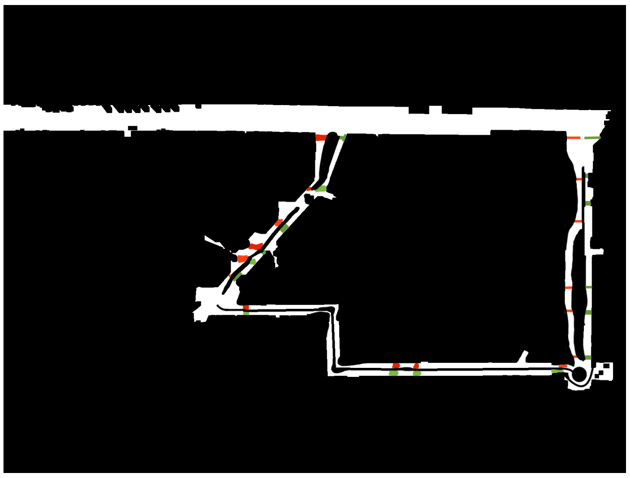


Fig. 13. The final map has

We generated an occupancy grid from the final image seen in Fig. 13 by converting the data between various formats. First, we converted pixels to integers: white to 0, black to 100, red to 33, and green to 67. After converting each pixel, we saved the result in a NumPy array. This array conforms to the OccupancyGrid data structure, as every element is an integer from 0 to 100. Then, we uploaded the array directly to the car. We modified the path planner code to read directly from this array upon initialization, instead of taking an occupancy grid of Stata basement. This grid is used to create paths where the car follows the right side of the lanes.

3.2 Traffic Signals

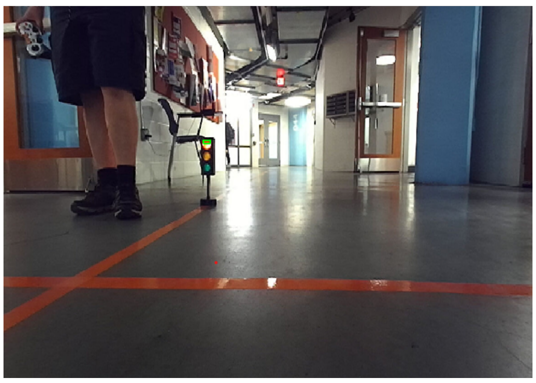
[James Morin]

Our modified path finder and pure pursuit path follower did an excellent job in keeping us driving on the correct side of the road, and combined with a safety stop, we kept from causing any collisions. Next we wanted to see how best we could obey the rest of the traffic laws: following the signals indicated by stoplights and stop signs.

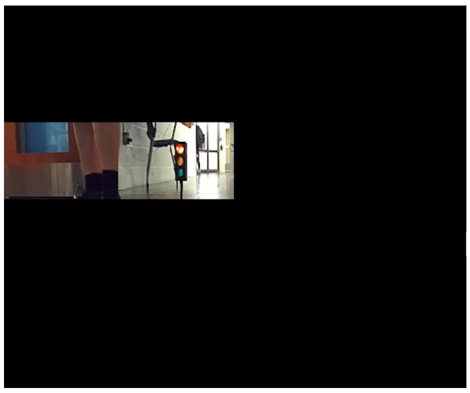
3.2.1 Stop Lights

To obey stop lights, we needed to determine the state of any stop light that we approached. We chose to use color segmentation with images captured by the racecar's camera. Several test images were taken with the car in order to tune parameters in our computer vision. Our goal is to determine the state of any stoplight that we encounter. The first step is to isolate the parts of the image that would contain any stoplights. We determined that since our lane following was so successful, any stop lights would only appear on the left half of the captured image. Additionally, the stoplights were at the same elevation as our camera, so no matter how close or far we were to the stoplight, it would always appear around the horizontal center line of the image. We drew black boxes to isolate the remaining part of the image where stoplights were determined to ever be, so that other lights or noise in the rest of the image would not throw off our detection. See Figure 14 for images describing this process.

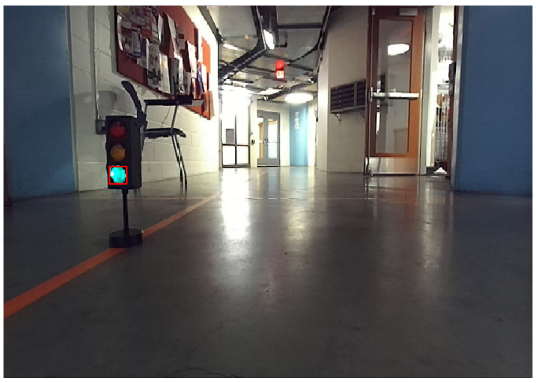
Using color segmentation, we independently observed whether there was a lit green light and whether there was a lit red light on the stoplights. If we



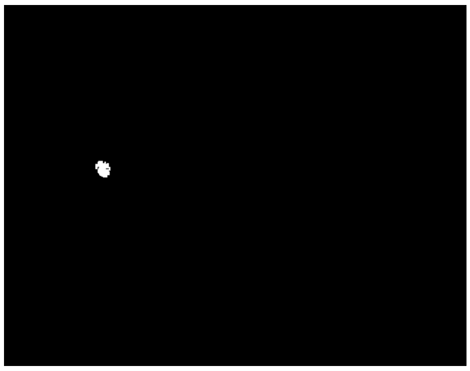
((a)) Image of stoplight at mid-range distance. A small green box is drawn around the red light to signify red light detection.



((b)) Image (a) with only a section visible, obstructing any regions where stoplights would not be.



((c)) Image of stoplight at close-range distance. A box is present around the lit green light.



((d)) Mask of image (c) after color segmentation for detecting the green light.

Fig. 14. Color segmentation for stop light state detection.

saw a red light and no green light, we determined that we ought to stop for this stoplight. In out test images, our results were accurate about 90% of the time.

When we used this stoplight detection on the car, we did find that we always stopped when we needed to. However, we also had some recurring behavior where the vehicle would stop when it didn't need to. These false positives for red lights were prevalent enough and cost us enough time that we decided it would be best for us to remove the stoplight detection altogether so we could just complete the course at a quick pace and not lose too many points on excessive time taken.

3.2.2 Stop Signs

We also attempted stop sign detection, but this also ended up being scrapped around the time we decided that stop light detection was not fully reliable enough to keep on the car. We programmed the functions in the staff-provided framework with a solution that theoretically would've worked just fine, but we had significant issues getting the detection to run on the robot. We were also getting word from some of the other teams working on this at the time that for them, the stop light detection had a rate of about 1 image per second, which is a very low rate. Facing significant import errors and with doubt that a successful import of the stop sign detector in our racecar code would work in a useful manner that didn't hinder the computation of racecar's more important operations, we sentenced our racecar to an existence of numerous traffic violations. This approach seemed generally to be the case among the teams in this competition, but we were still able to distinguish ourselves as top contenders with the quality of the work that we did include in the final version of our racecar's code.

3.3 Experimental Evaluation

3.3.1 Simulator Performance

[Allen Baranov]

The modified path planning and path following algorithms showed excellent results in simulation, as seen in the paths shown in Fig. 15 and Fig. 16, which stick to the road rules. In Fig. 15, the path keeps close to the right wall, showing that we are following the rule to drive on the right side of the road as desired. Even more impressive, in Fig. 16, the robot's path avoids making an immediate U-turn. Instead, it follows the lanes correctly, waiting to make the U-turn at an allowed intersection rather than crossing lanes directly, which would break traffic rules.

Also, the ability to manage multiple goals is clearly shown by how the paths are linked between Fig. 15 and Fig. 16. The endpoint in the first figure is the starting point in the second figure. This means that the path planner automatically moved to the next goal after reaching the first one. This smooth transition between goals shows how well our goal queue system works.

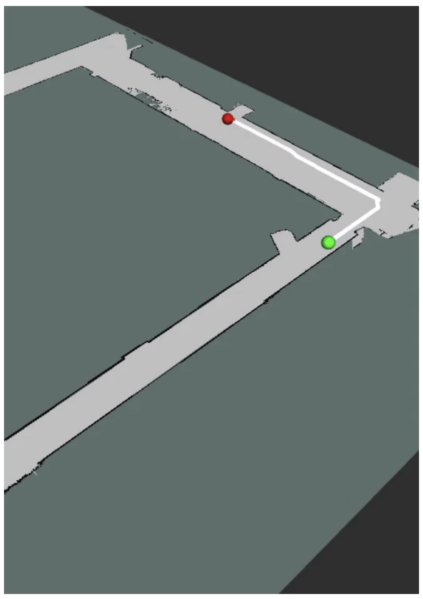


Fig. 15. The car begins by finding a path from the green point (start) to the red point (first goal).

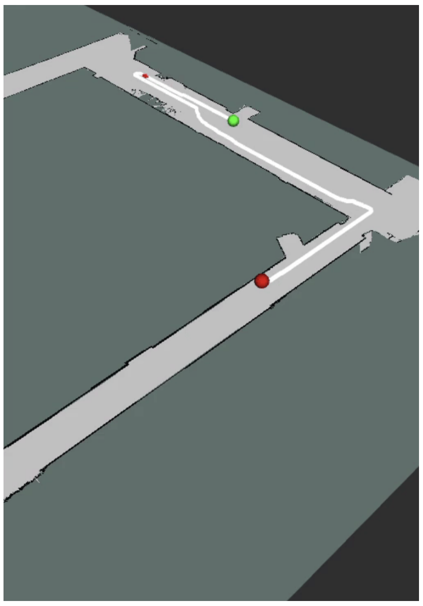


Fig. 16. Once the car has reached the goal, a period of 3 seconds elapses and then a path is calculated from the new green point (first goal) to the red point (second goal).

We conducted many rounds of testing with various starting points, numbers of goals, and goal locations to ensure the consistency of our system. We saw that the path planner performed well under any of our different scenarios. This adaptability was crucial for the challenge, since the goals were not predefined and the car would have to be able to path-plan and path-follow anywhere in Stata basement. The success in these tests made us confident that the system would perform just as well on the actual car and handle real driving challenges similarly to how it did in simulation. However, this did not turn out to be the case due to hurdles in the real world.

3.3.2 Robot Implementation

[James Morin]

In Luigi's mansion, one of the greatest challenges that we faced when porting our solution to the physical racecar was the reliability of the car's localization. Ever since the localization lab, we had pretty good localization in the section of the hallway outside of the lab room that had many features we could orient the car with. But long, uniform halls without much detail were difficult. The racecar was liable to lose track of its position, meaning as it drove forward, the location where the robot claimed to be was left behind relative to the real position. We refined our implementation of localization, conferring with other teams about the parameters they used, and referencing the staff solution. This meant that our localization was much more reliable and we were able to complete the collection of all shells.

While in simulation we can get close to and even pass through walls, in reality these were behaviors to avoid. We made sure to iteratively edit our occupancy map so that for any region where our pure pursuit following ran too close to or into a wall, we updated the map in ways that led the path planning to correct these issues. See the bottom right of Figure 11 for an example of a large roundabout that we imposed on the path finding map so that a u-turn in this section would steer wide enough to avoid wall collisions. Many similar alterations were proactively and reactively added to the map as we refined it.

In the competition, we proudly collected 3 shells without any manual assistance and with only a couple of traffic infractions related to the stop lights and stop signs. The only reason that we did not also make it back to the starting point was that our final goal pose was set inside of a wall of the edited occupancy grid which meant that no path was generated from the 3rd shell back to the start. One of our first improvements if we were to work on this further would be to update the path finder so that selected points close to the permissible path areas but inside of a wall would automatically choose the nearest available spot to navigate to, rather than ignoring the destination.

4 Conclusion

[Remeyn Mechura]

Over the course of this final challenge, we have absolutely demolished Mario's

Circuit using a Hough transform to first filter for line candidates, and then we used several other novel techniques to increase accuracy, computational efficiency, and smoothness. Our blistering 49 second run stands as proof of our success. In the Stata basement, our code conquered Luigi's domain. In a highly realistic scenario, we decided to forego the stop lights and stop signs, in true Bostonian fashion. Using Allen and James' brilliant alteration of the occupancy grid and further refinement to constrain directional motion, making sure we drove on the right, our car took no prisoners. Our car successfully stopped at each of the points, collected all of the shells, and also required no manual intervention. Ours was one of the best performances of the night, and potentially in the history of RSS.

5 Lessons Learned

5.1 Allen Baranov

From this project, which was huge in scope and very open-ended, I learned that I shouldn't ever focus on one strategy for a problem. I felt that in my part of the project, I dove far too deeply into solutions that ended up being unsuitable. This is a natural part of any difficult project, but in my case, I begin implementing solutions while ignoring red flags that arise about the efficacy of the solutions. This leads me to sink far more time into a dead-end than I needed to.

5.2 James Morin

This final project taught me how effective a good team with a healthy dynamic is, regardless of the obstacles faced. Everyone on our team respects each other's effort and skill and we trust each other to collaborate without pride or ego. We always enjoyed getting things done together and making progress on the challenges. Even though we did not achieve every single thing we wanted to get done, we delivered a very capable robot that scored very high in the competition.

5.3 Jian Ming Chen

The final project showed me how important it is to start on tasks early. I was able to start on Mario's circuit relatively early and as a result finished the algorithm for it ahead of schedule. This allowed me more bandwidth to help my team with city driving as it is a more complicated problem. This project had many moving parts and really tested our team's ability to delegate tasks and work asynchronously. I think we did relatively well on that, and I learned again how important communication and trust is in a group project.

5.4 Remeyn Mechura

Sometimes you just get lucky.

5.5 Ryan Xiao

Through this final project, I learned about the beauty of trial and error. I was mainly focused on getting the robot to complete Mario's Circuit. Although our initial implementations seemed to work, rigorous testing on the car exposed the many special cases that we did not in fact think of. One such instance was a very tricky line in one section of the track that crossed lanes 3-6. On our image, this line appeared to have a very similar slope to the true lane line, causing the robot to miscalculate the lane line position. This messed up the pursuit point calculation and made our robot sharply veer off course. It was important for us to test components one at a time in order to pinpoint specific issues in the car and allow us to complete the challenge.