Vision Based Dynamic Obstacle Interactions using Inverse Perspective Mapping

Archit Hardikar
University of Pennsylvania
Philadelphia, USA
architnh@seas.upenn.edu

Nicholas Marion
University of Pennsylvania
Philadelphia, USA
npmarion@seas.upenn.edu

Lenning Davis
University of Pennsylvania
Philadelphia, USA
ldavisiv@seas.upenn.edu

I. Introduction

This paper seeks to describe the methods team 4 utilized in order to compete in race III, in addition to detailing our work for the final course project. The overall setup for our car on race III involved breaking the code down into three distinct nodes. These nodes included the raceline and velocity planning node, Pure Pursuit node, the RRT* node. Additionally, a SLAM node was utilized for initial racecourse map building, and a particle filter node for state estimation. The general code setup and data flow for this project is as follows. First, our team would create a map using the SLAM package. Next, we would utilize the raceline and velocity planning node to create a desired raceline and velocity profile. From there, this information would be uploaded into our Pure Pursuit node prior to racing. With that data uploaded, we then utilized Pure Pursuit for controlling the vehicle around the course, and the RRT* node for predicting collisions and planing a safe route as necessary.

After going through the fine details of the modules of race III, we will then describe in meticulous detail our development efforts for the final project. The goal of our final project was to utilize cameras to track opponents during a race, and modify the RRT* plan based on their current pose and velocity by introducing a 4th "Inverse Perspective Mapping (IPM) dynamic obstacle" node. In order to accomplish this task, we setup a four camera system that allows our vehicle to see its full surroundings. We created a neural network for identifying opponents, and then used inverse perspective mapping to convert the opponent location in the image space into coordinates relative to the ego vehicle. With an estimate of opponent pose and velocity we modified our planning algorithm to reduce false positive collision detection, while still accounting for collisions with static obstacles. Results from tests on the real F1Tenth vehicle establish the feasibility of the "IPM dynamic obstacle" node and identify areas of development towards high-speed implementation

II. Related Work

Using inverse perspective mapping is not a new development in the field of autonomous driving. In fact, there are many papers describing both its usefulness as well as its limitations, in addition to the many other modules our team implemented such as pure pursuit and RRT.

In one paper titled Automatic Real-Time Road Marking Recognition Using a Feature Driven Approach [1], the authors use inverse perspective mapping in order to detect road signs embedded in the pavement. In this application, they use color thresholding and shape extraction for object recognition. Additionally, they tested the results of utilizing a neural network for detecting the road text and features. Where their testing differs from our own, is when they begin comparing output results and training their data-set on outputs in varying weather and driving conditions. Our project was simpler than theirs in the sense that our driving conditions were indoors and were controlled, which limited the uncertainty in our detections. Another paper titled Motion Planning for Urban Driving Using RRT [2], used RRT to to develop an autonomous car capable of driving with other vehicles. The authors present a novel implementation of RRT that takes into account not only the surroundings of the vehicle, but also the vehicle dynamics, in order to plan feasible paths. Additionally, they explore several heuristics to bias RRT into choosing a path that adheres to requirements such as passing the shortest path condition. They then actually tested their algorithm on a real car in a simulated city environment, in order to test its effectiveness. Interestingly, they mention that one of the issues in their implementation was that they were unable to detect slow moving vehicles, and would often crash into them as they were considered static obstacles. This parallels our project objective of detecting dynamic obstacles and treating them uniquely as compared with statics ones, thus ensuring a safe planed path in these scenarios.

III. Race III

A. Mapping, Global Raceline and Velocity Planning

Unlike the previous races, race III was going to be on a map that could not be mapped long beforehand. Thus, we created a pipeline to capture a map, generate a raceline, and build a velocity profile for an arbitrary racetrack. Our occupancy map for the race track was captured on the day of the race using the ROS SLAM Toolbox [3]. The occupancy map was converted into an image and thresholded to detect the inside and outside boundaries. The inside and outside boundaries were used to construct a centerline and boundary map (.csv format). The boundary map was fed into the TUM global trajectory planner to construct the optimal raceline [4]. This raceline was adjusted

manually to account for differences in the map on race day (changes to the course caused by wind).

In order to control the velocity of the car, we determined that it would be best to pre-plan the vehicle velocity based on the curvature of the racecourse. To perform this in the second race, our team calculated the second derivative of our race spline. A second derivative represents curvature, and thus we utilized this information to scale the values into velocities within our planned range. In an effort to make our velocity transitions smoother, we also used convolutions to apply Gaussian blur to the derivative plot. By tweaking the scaling of the derivative results, as well as adjusting the width of our convolutional filter, we were able to create a relatively smooth velocity profile that worked well on the Levine hall track.

Because race three involved fewer straight aways, and had additional turns, we wanted to improve our velocity profiling setup used in race 2. To do this, we utilized a more advanced method for pre-planning our velocities that took into account the lateral and longitudinal acceleration limits of our vehicle. This allowed us in theory to be able to create the fastest possible profile, while still maintaining grip. To perform this task, we utilized the forward backward velocity solver [5]. This solver utilized the curvature of the track, in a similar way as what we did in race 2, but also takes into account the physical limitations of the car in order to calculate a drive-able velocity profile using two distinct steps. The general flow for this algorithm is as follows. First, the curvature is calculated at each point of the spline. Next, the maximum possible lateral acceleration is calculated at each point. From there, the forward step is performed. In this step the algorithm creates a plot of all of the positive accelerations. Next, the backward step is applied, which accounts for all of the deceleration's along the course. In order to determine the maximum longitudinal and lateral accelerations of our car, we tested several different maximum acceleration values. We continued testing until we were able to determine values that caused our vehicle to be on the edge of extreme skidding, while still maintaining control. The following shows an example velocity profile we created. In this example, red indicates slower regions while green indicates locations where higher velocities were applied on the spline.

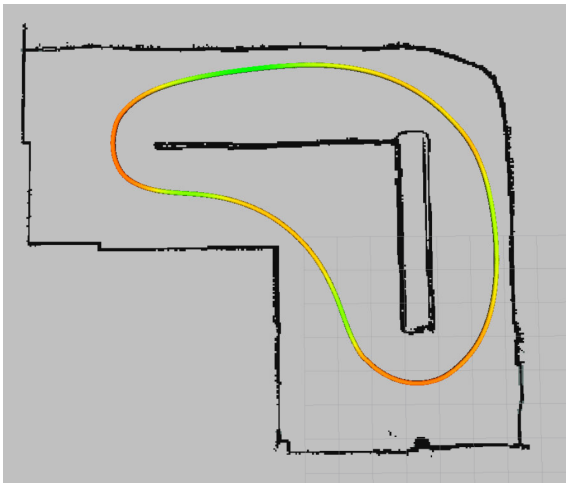


Fig. 1: Velocity Profile for Race 3 Track

Algorithm 6: RRT*

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $X_{\text{near}} \leftarrow \texttt{Near}(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}^*}(\log(\text{card}\,(V))/\,\text{card}\,(V))^{1/d}, \eta\})$;
    $V \leftarrow V \cup \{x_{\text{new}}\}$;
    $x_{\min} \leftarrow x_{\text{nearest}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{nearest}}) + c(\texttt{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Connect along a minimum-cost path
      if $\texttt{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \wedge \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ **then**
        $x_{\min} \leftarrow x_{\text{near}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$
    $E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Rewire the tree
      if $\texttt{CollisionFree}(x_{\text{new}}, x_{\text{near}}) \wedge \texttt{Cost}(x_{\text{new}}) + c(\texttt{Line}(x_{\text{new}}, x_{\text{near}})) < \texttt{Cost}(x_{\text{near}})$ **then** $x_{\text{parent}} \leftarrow \texttt{Parent}(x_{\text{near}})$;
      $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
return $G = (V, E)$;

Fig. 2: RRT* pseudo-code

B. Local Planning and Obstacle Avoidance

1) Pure Pursuit and Modified Spline based RRT :* For local path planning, we decided to perform spline-based tracking. The car had a dynamic look-ahead system put in place. It followed an optimal race-line using pure pursuit. The pure pursuit look-ahead was used to follow the pure pursuit trajectory. The second look-ahead checked for potential collisions along the optimal pure pursuit race-line up until the pure-pursuit look-ahead. On encountering a possible obstacle in the path, it would turn on the RRT* algorithm for obstacle avoidance.

Another look-ahead was used to plot a point on the path which would provide an instantaneous path using spline-based RRT* and steer the ego-car away from the obstacles. If RRT* was unable to return a path, the look ahead would reduce by a small amount and the same steps would be followed. This method was repeated thrice to guarantee a path around the obstacle. This helped us reduce the resolution of the RRT* grid, and reduce the number of nodes. RRT can be quite heavy for calculations especially when the LaserScan update is as fast as 15-40 ms based on manufacturer. For the Hokuyu 2D LIDAR, refresh rate of scans was 25 milliseconds. In order to synchronize the RRT* based spline computation with the LaserScan update, a performance study was essential.

For head-to-head racing, it is essential, to have an optimized RRT* planning with a fast refresh rate. The faster it publishes a path, the faster the split-second decision, and the better the obstacle avoidance. By changing the occupancy grid size, number of nodes, and the resolution of the occupancy grid, we were able to speed up the operation of RRT* significantly. Because our problem statement involved planning a small path for local obstacle avoidance, synchronous path planning updates were necessary. It was essential to limit the activation of RRT only when necessary (e.g for obstacle avoidance and overtaking). We performed a performance study of our RRT* algorithm as shown in Table I in order to find the optimal grid size that favored execution time, but left a fine enough scale

resolution to still be useful during the race.

TABLE I: Performance Studies

# Nodes	Grid Size (pix)	Resolution (m/pix)	Execution Time(s)
100	46*46	0.14	0.007
100	100*100	0.14	0.050
300	100*100	0.14	0.061
300	100*100	0.04	0.108
1000	150*150	0.04	0.151

The execution times listed in the table are based on 15 consecutive readings recorded while the car calculated paths around the same obstacle during each experiment. With 1000 nodes, a 150 pixel *150 pixel occupancy grid with 0.04 m/pixel resolution, the maximum execution time clocked 0.2907 seconds. As against, for 100 nodes with 46*46 grid size and 0.14 resolution, the grid is considerably smaller, nodes are placed more sparsely, and the reach is also lower. But this helps bring down the execution time. The minimum execution time clocked for this case was 0.0009 seconds. The spline better approximates the path and accounts for the sparse nature of the RRT* algorithm. The average execution time of the fifth case is 20 times slower than the first case. Thus, by doing performance studies, we made our RRT* based spline generation much faster. This helped us a lot during the race for head-to-head high speed racing. We noticed that our obstacle avoidance was significantly better than our competitors.

2) Collision Detection: If we assume that the fastest lap time will come from pure pursuit, RRT* should only activate when there is an obstacle preventing the car from following the optimal raceline. Thus, we created a node that determines what driving algorithm to use and published a boolean for the other nodes. We accomplished this by creating a second "collision check" occupancy grid that connects a line that's the width of the car from the car's local position to a reference look ahead point on the optimal raceline. This collision check occupancy grid is occupied for points between the current car position and goal point. When a scan message is published, the occupancy grid and the collision check occupancy grid are compared. If any two matching cells on both grids are occupied, an obstacle is in the way of pure pursuit so RRT* is activated. Once the occupancy grid and the collision check occupancy grid no longer intersect, RRT* is turned off and the car returns to pure pursuit. A visualization of the two occupancy grids is shown in Fig. 3.

The line between the current car position and the reference point was drawn on the occupancy grid using Bresenham's line algorithm [6]. However, this caused issues, as shown in Fig. 4a, where the collision detection node would return true around turns even when there is no obstacle in the path of the vehicle. This was resolved by calculating a path in multiple segments with Bresenham's line algorithm along the raceline to determine if the path is blocked, shown in Fig. 4.

C. Race III Discussion

We noticed some advantages and some disadvantages to our approach towards Race III. While our static obstacle avoidance was quite robust, we experienced quite a few scenarios when the car would drift at higher speeds, which would make it deviate from the optimal Pure Pursuit raceline. This would lead to higher computational costs. Our Pure Pursuit had a built in controller to account for drifting and help steer. It started becoming increasingly cogent, that we needed a way to distinguish static obstacles from dynamic obstacles and limit the use of RRT*. With lower computational costs, and less number of hindrances, we could potentially reduce lap time.

Additionally, when testing on a static course with no opponents, we found that RRT* could guarantee that we would not hit anything on the track. This was even successful when placing static obstacles along tight turns, or when moving extremely fast along straightaways. This was not the case however once we began racing against dynamic opponents. The collision detection algorithm could not distinguish static obstacles from our opponents. As a result, our vehicle would

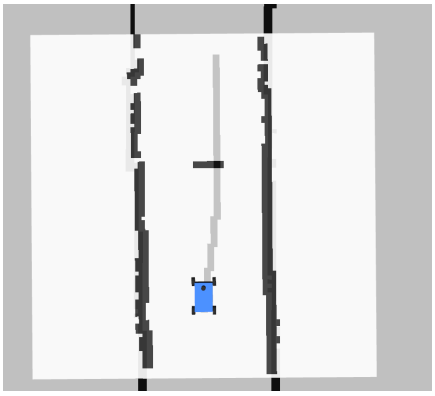
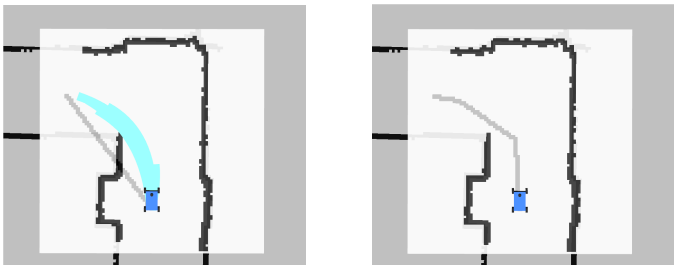


Fig. 3: Occupancy grid shown in black and "collision check" occupancy grid shown in gray overlaid in simulation. The intersection of these spaces signifies a collision and activates RRT*.



(a) False positive collision detected by the intersection of the occupancy grid and the "collision check" occupancy grid. RRT* is activated and the resulting local path is shown in blue.

(b) Collision detection using the segmented path fix.

Fig. 4: The false positive collision error around turns is resolved by applying Bresenham's Line algorithm in segments along the pure pursuit global trajectory.

veer violently when approaching an opponent vehicle.

IV. Final Project

A. Motivation and Overview

Based on the results from race three, we had several takeaways about weaknesses in our current implementation and insights into how some of our competitors were able to leverage that to their advantage. One of the most evident drawbacks to our implementation was that despite having a very quick pure pursuit raceline, we were never able to run a lap with our max expected performance speed, because our obstacle avoidance algorithm would oftentimes turn-on around corners or when we neared our opponents, forcing the vehicle to slow down. Contrasting our overly conservative method, to that of our opponents, many of whom chose to bypass implementing any type of obstacle avoidance into their system, it was clear that our chosen method which favors prudent handling of collisions with adversaries, was no match for a more brute force approach, such as one solely using an extremely optimized version of pure pursuit for high speed circumnavigation of the racecourse, which disregarded the challenge of avoiding adversaries.

Additionally, we found that in situations where we were being passed by an opponent vehicle, our vehicle would oftentimes perform a violent turning maneuver to avoid the vehicle passing our car, due to its proximity to ours as it passed directly in front of our vehicle. This issue caused our vehicle to crash and forced our team to consider alternative approaches to path planning and sensing, that would prevent this type of situation from occurring. Our initial thoughts were to utilize the 2D LIDAR point cloud to detect our opponent and track their velocity. This proved to be quite challenging as it is difficult to detect the opponents, which show up as blobs in the range data. After further consideration, our team deduced that utilizing cameras would be an ideal method for determining our opponent's state.

Upon further discussion the team decided that the best way to track an opponent from all angles using cameras, would be to position them on all sides of the vehicle. To do this, we created a CAD model for a four camera mounting bracket and lasercut the bracket into ABS material. An image of the entire setup with the Realsense cameras that we utilized and a screenshot of the CAD model are shown in the Fig 5 and Fig. 6.

In order to determine the position of our adversaries in space, we needed to first identify them, and then track their location. To do this, we decided to use a convolutional neural network to identify bounding boxes around the opponent vehicle, which would provide us information about not only the position of the vehicle in the frame, but also about the size of the vehicle. From there, we determined that we could perform inverse perspective mapping on the images to project the sensor data from the image plane, onto a flat ground plane. We then could derive the position in the world of the opponent car and also track its velocity and acceleration as it maneuvers along the racecourse.

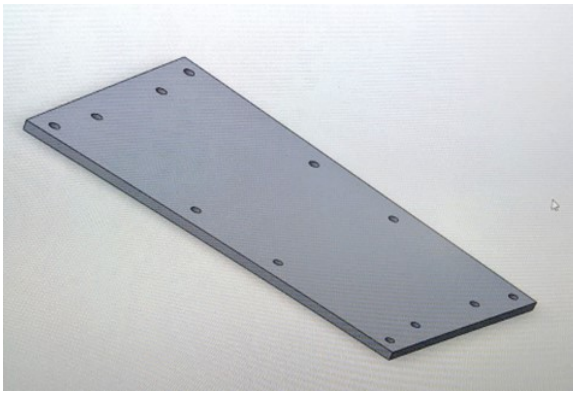


Fig. 5: CAD model of the camera mounting bracket that we created.

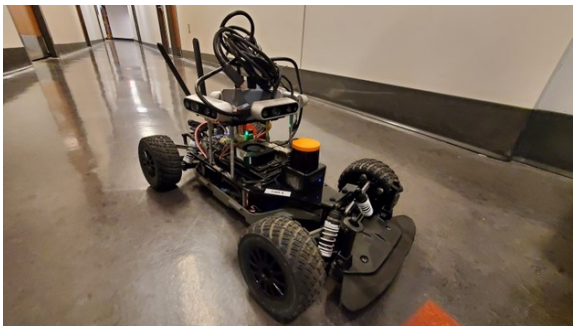


Fig. 6: Image of the four camera system attached to our laser-cut mounting bracket on the car.

Once we obtained the position and velocity data of our opponents, we can then begin to modify our motion plan in order to leverage the newfound data to our advantage. To do this, our team derived an algorithm that masks the opponent's position in an occupancy grid fed into the RRT* algorithm for planning, when the opponent is moving at or above the current speed of our vehicle. This remedies the issue we experienced in the track, in which our car would try and avoid faster moving vehicles, while it also increases the speed and reliability of our vehicle as it reduces the number of times RRT* needs to takeover the steering commands from the pure pursuit node. An example comparing the standard RRT* reaction vs our new algorithm's reaction are shown in Fig. 7. Note that on the left image using the original implementation, our vehicle has to take a longer path around a vehicle that it would never catch anyways, than in the image on the right, in which our car doesn't stray from the white pure pursuit raceline as it follows an opponent moving at the same speed.

B. Methods: Opponent Detection

Our initial planned opponent detection pipeline involved using a convolutional neural network to create bounding boxes on images captured in real time with our Realsense cameras. The images would then be fed into our inverse perspective mapping algorithm in order to determine the position of those opponents in 3d space.

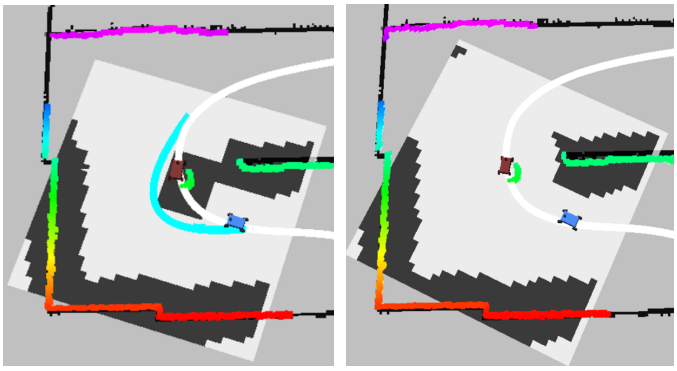


Fig. 7: **Left:** RRT* planned route around vehicle traveling at the same speed. **Right:** Filtered occupancy grid using IPM, opponent vehicle is no longer an obstacle

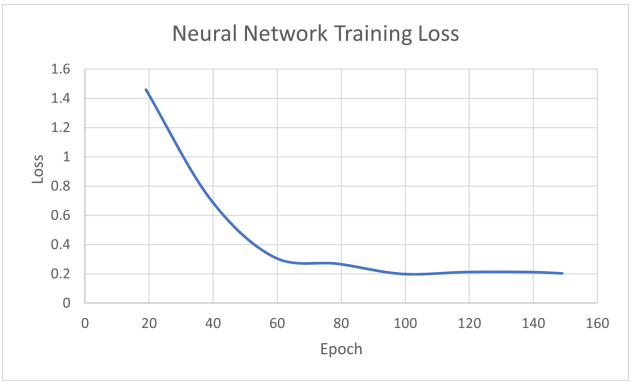
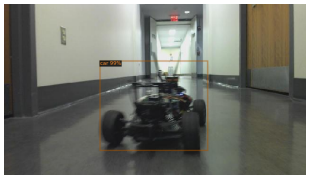


Fig. 8: Network training loss over 150 epochs

In order to accomplish the detection task, we utilized the machine learning libraries Pytorch and Detectron2. Pytorch is a standardized software tool that works with Python for creating and training neural networks, while Detectron2 is a python library that utilizes Pytorch and simplifies some of the process of setting up and training neural networks. After reviewing multiple detection frameworks available with Detectron2, our team decided on utilizing Faster R-CNN with a Feature Pyramid Network and 50 backbone layers as our detection framework. There were two main reasons for this decision. First, this framework trains relatively quickly, which saved us time as we didn't need to spend hours waiting for the network to learn. Most importantly though, this network had a very small inference time. This was vital to our system because in a racing scenario opponents move extremely fast, so being able to rapidly detect their position is vital in order to react to their motions. Additionally, because we are running our code on the Jetson board, we already have limited computing power when compared with a standard laptop computer. One drawback we had to accept with this network was that the average precision is lower than for example a faster R-CNN network with 100 backbone layers. After performing testing with our Realsense cameras though, we found this to not be a major issue as the system seemed to have no issue distinguishing the opponent car from its surroundings. The image shown in Fig. 8 illustrates the training loss for the network.

The final network we utilized trained for 75 epochs. We chose this value based on the plot shown above. Because the loss plot slows down to an almost flat line after 90 epochs, we knew that it wouldn't improve the overall accuracy of the model as the loss was no longer decreasing. Additionally, using more training epochs could actually cause worse classification accuracy because the system would be over-fitting to the training data, thus making it less able to generalize for images that appear different from the training data. Our intention with this network was to leverage the bounding box outputs to track the position of an opponent, and using inverse perspective mapping, actually track its velocity. The network satisfied all of our expectations as it was reliable and relatively quick to update on a streamed video. Example detection output images are shown Fig. 9.

(a) Outdoor environment



(b) Indoor environment

Fig. 9: Detection outputs from Faster RCNN Neural Network.

After running the opponent detection pipeline on the car, we next moved on to attempting to run the detection software inside the docker. This was required because ROS2 is installed inside of the docker, and we planned on using ROS to broadcast the coordinates of our detections in order to perform velocity estimation using another module in our system. While we were able to get Pytorch working using solely the CPU for detections, we were unable to install Pytorch with the GPU enabled. Without a GPU, the detections were extremely slow and were unusable. After attemping several install methods, including multiple methods for building the code from source, our team made the decision to abandon the object detection code and utilize Apriltags for opponent detection. This decision was mainly made in an effort to continue progressing towards our end goal, as we didn't want to waste too much time trying to get the detections to work inside of the docker.

Because we determined that using Apriltags were the best choice of action, we made the assumption that from this point on, the opponent car will always have an Apriltag attached to the back of it. In order to detect these Apriltags, we utilized the Apriltag python library for detection. An example of a detection is shown in Fig. 10. While the Apriltag package can provide tag orientation and position data given a known tag size, we are solely interested in obtaining the x,y position of the tag within the image frame. This position will be used in the inverse perspective mapping for calculating the position of the tag in world coordinates.

Fig. 10: Example substitute Apriltag opponent detection

C. Methods: Inverse Perspective Mapping

Once an opponent race car has been detected we must estimate its relative position and velocity before doing anything useful. The LIDAR provides very accurate range estimation, however it is difficult to identify and track the opponent race car from the range readings. Our solution was to leverage the cameras themselves to estimate the relative position and velocity of the opponent racecar to inform the decision whether to overtake with RRT* or continue with pure pursuit to close the gap.

Estimating the position of a point in 3D space from a camera image requires an understanding of the pinhole camera model. The pinhole camera model is a method to describe the relationship between points relative to a world frame P^w, points relative to the camera frame P^c and points in the image frame P^i using camera intrinsics K and camera extrinsics RT. Where:

$$P^w = \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix}, P^c = \begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix}, P^i = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \tag{1}$$

The camera intrinsics describe how a point is mapped from the camera frame to the image frame P^c to P^i, where lamba is a scaling factor used to reduce the 3rd element of P^i to 1.

$$\lambda \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} \tag{2}$$

The camera extrinsics are a 4x4 homogeneous transformation that describe how a point is mapped from the world frame to the camera frame P^w to P^c, where R denotes the 3x3 rotation matrix between the coordinate frames, and T denotes the 3x1 translation vector.

$$\begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} = \begin{bmatrix} R & T \end{bmatrix} \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix} \tag{3}$$

The camera intrinsics are calculated through calibration. We leveraged the OpenCV calibration functions and a calibration grid of known size to return estimates for the intrinsics across all four cameras at 1920x1080 and 640x480 resolution.

The camera extrinsics were more difficult to estimate. We elected to set the "world" frame relative to the car; the x axis is aligned with the relative longitudinal direction, the y axis is aligned with the relative lateral direction, and the z axis is pointing upwards. Because the origin of this world frame is not visible to all four cameras at the same time, we elected to estimate the camera extrinsics manually through measurements. The translation is measured as the offset of the camera lens from the base (world) frame of the car. Next, the rotation matrix of the four cameras relative to the world frame was estimated. This matrix was constructed using the Euler angles of the cameras; the three cameras not facing forward had a yaw rotation. We attempted to mount the cameras with zero roll and pitch, but the motion of the suspension and the camera mounting tower both experienced motion during testing. Thus, we included small roll and pitch angles to account for these motions and better align our images. Finally, the axis must be swapped (R_{swap}) to align the camera z axis as pointing directly away from the camera. The construction of the camera extrinsic matrix is summarized as follows:

$$RT = R_{swap} R_w^c T_w^c \tag{4}$$

Thus, a point in the 3D world space can be projected into the image using the entire pinhole camera model:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \frac{1}{\lambda} \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R & T \end{bmatrix} \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix} \tag{5}$$

The pinhole camera model can be used for Inverse Perspective Mapping (IPM), or a bird's eye view around the vehicle from the images captured on the cameras. All pixels in the camera images are assumed to have a Z (height) value of 0, and thus lie on the ground. With perfect calibration, lines that converge to the vanishing point will be rectified back into parallel lines. The inverse perspective map image is defined in coordinates relative to the car, thus it is useful for planning and can be used as an alternative method of distance estimation. For construction, a blank image is defined with a size and resolution (meters/pixel), along with a meshgrid representing every pixel of the blank image in *xyz* coordinates. Every point in the meshgrid P^w is then projected into the image frame P^i using the camera intrinsics and extrinsics. If the point lies within the camera frame height and width, the value of the pixel it projects to is copied into the blank image. Example images and their resulting inverse perspective map are shown in Fig. 11. Note that the orange tape is rectified to a straight line in the world frame. Additionally, errors in camera calibration are visible as distortions in the reconstructed IPM image.

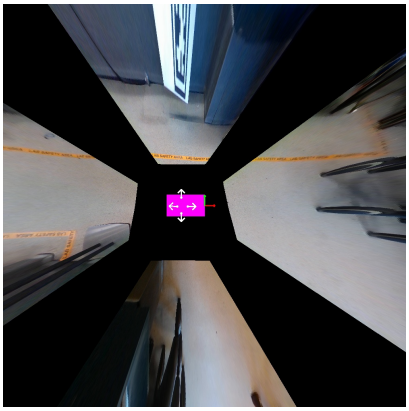
As mentioned previously, the inverse perspective map and the pinhole camera model can be used to estimate the real world position of an image pixel. This involves applying the inverse of the camera projection outlined in Eq. 5. However, we run into the issue of rectifying depth for the pixel. In the pinhole camera model, all world points along a line extending

(a) Font view

(b) Back view

(c) Left view

(d) Right view



(e) Complete 5x5 meter IPM with the car shown in purple, cameras shown with white arrows, world x axis shown in red and world y coordinates green.

Fig. 11: Complete IPM image construction.

from the camera lens project onto the same point in the image. In other terms, simply knowing where a pixel lies in image coordinates *(u,v)* does not rectify depth, and instead returns a line of possible points in world coordinates. This ambiguity can be resolved with methods such as a stereo camera set-up or depth readings. While the RealSense camera provides depth, the ideal range is limited to a maximum of 3 meters.

The inverse perspective map resolves depth by assuming all pixels have a height of $Z^w = 0$ and solving for the unknown scaling parameter λ. The scaling factor λ and the subsequent world coordinates of the point X_w and Y_w is found when u, v are known as follows:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \frac{1}{\lambda} \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R & T \end{bmatrix} \begin{pmatrix} X_w \\ Y_w \\ 0 \\ 1 \end{pmatrix}$$

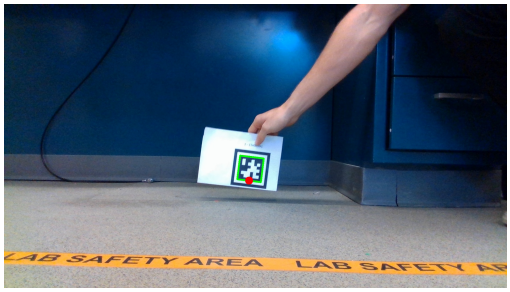
$$\begin{bmatrix} R & T \end{bmatrix} \begin{pmatrix} X_w \\ Y_w \\ 0 \\ 1 \end{pmatrix} = \lambda \begin{pmatrix} \frac{(u-u_0)}{f_x} \\ \frac{(v-v_0)}{f_y} \\ 1 \end{pmatrix} \tag{6}$$

$$\begin{pmatrix} X_w \\ Y_w \\ 0 \\ 1 \end{pmatrix} = \lambda \begin{bmatrix} R & T \end{bmatrix}^{-1} \begin{pmatrix} \frac{(u-u_0)}{f_x} \\ \frac{(v-v_0)}{f_y} \\ 1 \end{pmatrix}$$

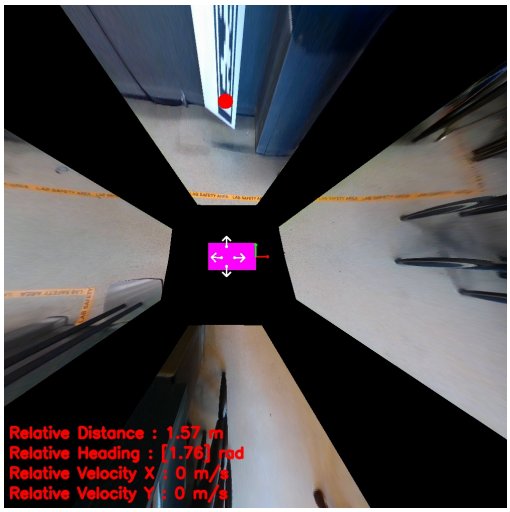
With this inverse projection model, the distance of any point in an image can be estimated given that it lies on the ground ($Z_w = 0$). Thus, we use the inverse projection model to estimate the distance to opponent vehicles. The opponent detection pipeline discussed in Section IV-B returns a bounding box on the opponent vehicle in an image. I would like to clarify that we are not assuming any a priori knowledge concerning the size of the opponent vehicle or AprilTag; the detection pipline simply returns the bounding box that defines the vehicle in the image. The midpoint of the bottom line of the bounding box is selected as the point to estimate distance as it will be the closest to the ground and at the visible center of the opponent vehicle. Assuming the point lies on the ground, the point is transformed into world coordinates using the camera intrinsic and extrinsic parameters. An AprilTag detected in an image and its subsequent distance estimation is shown in Fig. 12.

The IPM distance estimation approach is validated when overlaying the IPM image on top of the distance scans returned from the LIDAR. As an aside, we created a node that publishes the IPM image into RVIZ with the global position and orientation of the ego vehicle. This enables us to visualize the inverse perspective map of the world alongside the range readings. As illustrated in Fig. 13, the LIDAR scans and the IPM images align well. As discussed later in Section V, the information contained in the IPM image could be combined wtih other information to provide enhanced navigation capabilities. Disparities in certain parts of the image are visible, and can be attributed to errors in camera calibration (since we are not accounting for distortion) and errors in our camera extrinsics (as the car suspension and camera mounts move).

As discussed, Inverse Perspective Mapping returns the position of the opponent vehicle in real world coordinates given a bounding box on the opponent car in one of the camera images. Using multiple cameras facing in different directions ensures that an opponent vehicle can be tracked at all times, not just when they are directly in front of the ego vehicle. Additionally, sequential estimates of opponent vehicle position at a fixed interval yields the opponent's relative velocity to our vehicle. The position and relative velocity of the opponent vehicle are sent and used to better inform our local planning.



(a) Detected opponent vehicle/apriltag. Bounding box is shown in green and reference point for estimation shown in red.



(b) Estimated position of opponent reference point shown in world coordinates.

Fig. 12: Visualization of the inverse projection pipeline. The point identified in the first image is mapped into world coordinates using the inverse projection model described in Eq.6.

D. Methods: Dynamic Obstacle Algorithm

Given the position and velocity of the opponent vehicle, we next needed to modify our RRT* algorithm in order to consider our opponent as a dynamic obstacle, as opposed to a static one. After reading several papers that utilize RRT and RRT* for path planing [2], we found that most approaches assume that all obstacles are static, and would simply increase the update rate for RRT, in order to account for moving objects. This approach works well when dynamic obstacles are slow moving, but doesn't translate well for racing, especially when the opponent vehicles make quick heading changes and can easily completely block the path ahead of our vehicle.

One method our team devised in order to remedy this issue, was to remove the opponent vehicle from the RRT occupancy grid when it poses little to no risk to our vehicle. This situation occurs when the opponent vehicle is in front of our vehicle and driving faster than our own, meaning that it is escaping from us. It also occurs when the opponent vehicle is driving at the same speed as our vehicle. In these cases, its better to continue

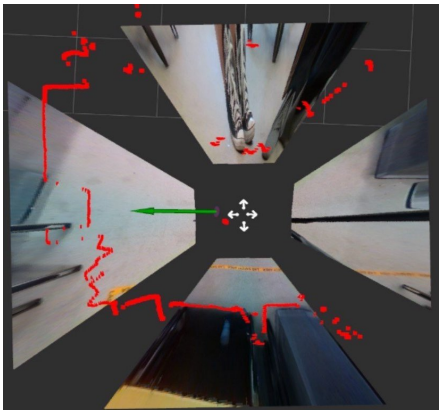
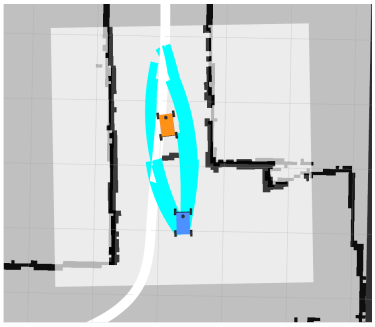


Fig. 13: Example IPM/LIDAR Image

on course, instead of trying to plan around an adversary who we wouldn't be able to pass anyways. This approach not only allows our vehicle to maintain speed as adversaries pass by, but also prevents our car from swerving out of the way of a vehicle as it passes us. This exact situation occurred in race three and is one of the reasons we chose to address this in our project. In that situation, once an opponent vehicle performs an overtake, it ends up directly in front of our vehicle with little to no space between the two cars. In this scenario our car would violently swerve away from the opponent car, which RRT* would consider an extremely close static obstacle, that must be avoided.

The basic outline of the algorithm is as follows. First the RRT node generates the occupancy grid following its standard protocol of using the 2D LIDAR pointcloud to fill in the grid cells. Next, the velocity and pose estimates of the opponent vehicle from the inverse perspective mapping node (discussed in Section IV-C) are used to determine if the opponent vehicle has an equal or greater relative velocity than our vehicle. If that was the case, the pose estimate of the opponent vehicle is used to remove the cells corresponding to the car from the occupancy grid. The size of the region eased was based on the expected size of the opponent vehicle. Once this step was complete, the modified occupancy grid would then be fed directly into the collision check and RRT* algorithms. The collision check algorithm discussed in Section III-B2 determines if the desired path is blocked and if so, creates a new spline around any obstacles, otherwise the car would continue to follow the path specified by Pure Pursuit. A visualization of this process is shown in Fig. 14. Fig. 14a shows a situation where the ego vehicle is gaining on the opponent, the opponent is present in the occupancy grid and RRT* is activated for an overtake (blue lines). Fig. 14b shows a situation where the ego vehicle is not gaining on the opponent, thus the opponent is removed from the occupancy grid using the IPM position estimate and pure pursuit remains activated.

Initially when implementing this system, we encountered several issues due to our velocity estimate. The first being noise in the velocity measurement. When the car is driving, the vibrations and suspension movements cause the cameras



(a) Opponent vehicle has negative relative velocity (estimated distance is decreasing). RRT* activated.

(b) Opponent vehicle has zero or positive relative velocity (estimated distance is constant or increasing). Pure Pursuit activated.

Fig. 14: The dynamic obstacle algorithm facilitates faster lap times by only activating RRT* when we are closing the distance to an opponent vehicle.

to move, inducing errors is the camera extrinsics and consequently the distance/velocity estimates. To account for this, we filtered the data and rounded off small readings to zero. Additionally, during initial testing we also had issues where the detected point using the IPM system wouldn't occur in the same location as the LIDAR points scanning the same objects. What we found was that our camera calibration values were not entirely correct and were causing errors in the pose estimate. After re-calibrating the cameras, we were instantly able to obtain data that matched very well to the LIDAR scans, indicating to us that our calibration was correct.

E. Real Car Performance and Discussion

The methods described in the previous sections define our approach to improving race performance using vision. With the structure of our full stack complete, we sought to validate the performance of the software on the real car; while it is worth while to present how the algorithms work in simulation, we understand that the goal of this project is to definitively improve race performance on a real vehicle.

Tests on the real vehicle were conducted in the Levine 2nd floor loop on our F1Tenth vehicle with the retrofitted camera module. We sought to demonstrate our pure-pursuit/RRT* planner with the IPM dynamic obstacle algorithm. In comparison to our race III approach, we expected to see the ego vehicle continue with pure pursuit while an opponent vehicle is blocking the path, so long as the relative velocity of the opponent vehicle is greater than or equal to zero. In other terms, RRT* should only activate when a static obstacle is blocking the ego vehicle path or the ego vehicle is gaining on the opponent vehicle (overtake maneuver). To test this behavior,we pulled a box with an attached AprilTag in front of the ego vehicle to simulate an opponent. The desired behavior is to maximize the time under pure pursuit control by removing/including the opponent vehicle in the occupancy grid depending on its relative velocity. An example image captured during the Levine tests is shown in Fig. 15.

Fig. 15: Image from front facing camera during Levine testing.

The results of the tests establish feasibility of the IPM dynamic obstacle approach and provide areas of improvement for future work. Demonstrated in the videos provided in Section VII, the dynamic obstacle algorithm successfully removes the opponent vehicle during runs down the hallway at 1 m/s and reduces the instances in which RRT* obstacle avoidance is active. While the dynamic obstacle node successfully removes the opponent vehicle, it does so intermittently causing the car to activate RRT* at certain instances when not necessary.

The cause of the intermittent RRT* activation issue in testing was identified to be with opponent vehicle detection; the vehicle detection method (AprilTags) was operating a low frequency and not identifying the opponent vehicle as a result of motion blur. The operating frequency issue was identified by timing the detection and inverse projection mapping algorithms. We originally ran tests with all four cameras active which caused issues with all USB devices (VESC and Joystick). After reducing to a single camera the node was able to run successfully, but at a frequency of several Hz. By reducing the image resolution to 640x480, the vision pipeline ran at 6-8Hz. This frequency is not ideal for high speed racing; in hindsight, we recognize that building our vision pipeline in C++ over Python would have enabled higher frequency operation. The other issue we encountered was motion blur. The camera module is mounted high on the vehicle and consequently experiences more shaking when the ego vehicle moves. Combined with the motion of the simulated opponent, the Apriltag was detected approximately every other frame. This issue could be solved in the future with the proposed

neural network approach to opponent detection; the detection neural network should be robust to motion blur with the proper training data.

Despite the intermittent detection issues, the IPM dynamic obstacle node noticeably reduces the amount of time that RRT* is activated unnecessarily in comparison to our race III software stack. This outcome aligns with our project intention to amend the obstacle avoidance issues we experienced during race III. As demonstrated with our real-car testing, with faster hardware and quicker vision execution the methods outlined in this project can drastically improve race performance in a head-to-head format. Videos of real car testing are available in the presentation provided in Section VII.

V. Future Work

Discussed in Section IV-E, we understand the next steps to improve the IPM dynamic obstacle node. A C++ implementation of our inverse projection software would reduce runtime and improve opponent detection. Further, reducing the camera exposure and implementing the neural network for opponent detection would resolve the motion blur issues. To note, the neural network trained for detection worked well offline, and it is worth spending time to implement the network inside the docker container to investigate the performance on the NVIDIA Jetson. With these solutions, we will test the performance of the IPM dynamic obstacle node at race speeds. Further, we will investigate returning multiple cameras to the camera module to provide vision to the sides of the ego vehicle.

We believe that there is more worth investigating with the IPM image constructed from the four camera module - potentially other research projects. For example, overlaying the LIDAR scans with the IPM image (shown in Fig. 13) establishes the feasibility of substituting calibrated cameras and IPM for range sensors. Can we complete a race with IPM as a stand-in for LIDAR? Furthermore, we understand that the layout for our camera module was not optimal; future work could include overlapping the cameras' fields of view and improving the IPM image using feature matching and image stitching.

VI. Acknowledgements

We thank professor Rahul Mangharam, and the ESE615 teaching staff from University of Pennsylvania, Philadelphia for their technical guidance and expertise that made this project possible. We would also like to thank the entire F1TENTH community for providing the open-source software used throughout the semester.

VII. Media

Since presenting in class, we have captured more videos of our software stack running on the real car. We have uploaded the videos to our presentation.

- Link to presentation video: LINK.
- Link to presentation slides: LINK.

References

[1] T. P. B. Alireza Hkeyrollahi, "Automatic real-time road marking recognition using a feature driven approach," *Machine Vision and Applications*, pp. 123–133, 2012.

[2] J. T. E. F. J. P. H. e. A. Yoshiaki Kuwata, Gaston A. Fiore, "Motion planning for urban driving using rrt," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2008.

[3] S. Macenski and I. Jambrecic, "Slam toolbox: Slam for the dynamic world," *Journal of Open Source Software*, vol. 6, no. 61, p. 2783, 2021. [Online]. Available: https://doi.org/10.21105/joss.02783

[4] A. Heilmeier, A. Wischnewski, L. Hermansdorfer, J. Betz, M. Lienkamp, and B. Lohmann, "Minimum curvature trajectory planning and control for an autonomous race car," *Vehicle System Dynamics*, 2019.

[5] J. S. Kapania, Nitin R. and J. C. Gerdes, "A sequential two-step algorithm for fast generation of vehicle racing trajectories," *Journal of Dynamic Systems, Measurement, and Control*, 2016. [Online]. Available: https://arxiv.org/pdf/1902.00606

[6] J. E. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems journal*, vol. 4, no. 1, pp. 25–30, 1965.