Path Planning and Obstacle Avoidance

DSC180A A02 Group 3
Yuxi Luo, Seokmin Hong, Jia Shi

Abstract

Path planning and obstacle avoidance are important aspects of autonomous driving with regards to efficient routing and passenger safety. In order to travel from one point to another, GPS, IMU, and other sensors/applications have to work in unison in order to achieve this goal. Along the distance traveled, it is highly probable that obstacles will occur, deterring the original path intended for the vehicle to best travel. A legitimate example of this can involve animals that cross the road abruptly. In these instances, algorithms and techniques implemented to address this issue will have to be robust and quick enough to respond, especially if vehicles are moving at high velocities. In this report, different algorithms such as ROS navigation stacks and behavior cloning/computer vision/reinforcement learning methods are implemented to create path planning and obstacle avoidance. Metrics and criteria are set up to evaluate the performance of each respective algorithm that is implemented. Using a controlled map with obstacles, generated through a gazebo simulation, each method of path planning and obstacle avoidance is evaluated on track completion speed, number of collisions, and efficient path navigation. Also, several path navigation methods were compared and evaluated on Donkey car simulator.

I. Introduction

An important aspect of path planning and obstacle avoidance with regards to autonomous driving is efficient pathing. In order to create the most efficient path, data must be fed as an input in order to derive the best possible output. Ideally the most desirable path is to maneuver in a straight line from point A to point B, however, obstacles along this path will often deter this possibility. This dilemma creates a need for obstacle avoidance and path planning as a result. This concept can be applied to any moving robot as it will have to avoid obstacles in order to arrive at the desired destination. In the following section, the methodology for avoiding obstacles and path planning will be described in further detail. Following the methods section, this report will describe the results of each respective navigation algorithm that was implemented. Metrics for evaluating each algorithm will also be discussed and each respective algorithm will be evaluated on its performance based on these metrics. Following the results section, there will be a discussion regarding the current state of obstacle avoidance and path planning algorithms.

II. Methods

The main components of this methodology will utilize the Intel RealSense D435 camera, a Linux operating system (OS), Donkey AI simulator, OpenCV library, Gazebo Simulator, RViz, and many ROS packages. The Intel RealSense camera ROS configurations available on the Intel website [1] is updated primarily for Linux usage, and as a result, Linux OS is the desired operating system due to improved compatibility. A frequent bottleneck that may arise is the lack of compatibility between some Ubuntu Linux versions with some ROS launch files provided by the Intel framework. It must be noted that Ubuntu version 16.04 and version 18.04 would work better with version ROS Kinetic and ROS Melodic respectively. If a different version of ROS were to be installed with another version of Ubuntu, errors are likely to arise frequently. With this information in mind, it should be noted that the methodology presented will be developed in Ubuntu version 18.04 and ROS Melodic.

The primary step taken was to ensure that each hardware and software components were working properly. Data collection and processing was completed using the TurtleBot robot [2], a personal robot kit with open-source software that supports ROS and more importantly, ROS Navigation packages. However, instead of physically using these TurtleBot robots in real-life, they were virtually spawned using the Gazebo simulator [3], which created a workspace to test different algorithms on the TurtleBot robots in different environments and maps. The TurtleBot uses the Intel Realsense R200 camera and an LDS-0 360 Laser Distance Sensor as its default hardware sensors. Because the robot being built for real-life autonomous navigation uses the Realsense D455 camera and a SICK Lidar, the configurations for the TurtleBot's default camera was changed to the Realsense D435 camera and the default lidar to the Hokuyo Lidar. The objective in these changes is to help decrease the hardware discrepancies between the Gazebo simulations and our real-life robot.

Once the TurtleBot was properly configured into the Gazebo simulator, our group was able to perform different navigation algorithms for autonomous navigation. We believe that the most important sensors for path planning and obstacle avoidance was processing RGB-Depth images from the Intel Realsense camera and incorporating LaserScan data from the Hokuyo Lidar. Our group wanted to see which sensors would be useful for different kinds of situations, such as correctly mapping the robot's surrounding areas, or allowing the robot to navigate around an obstacle from point

A to point B. To compare the capabilities of these two sensors, our group decided to integrate G-Mapping and RTAB-Mapping as the algorithms to test which sensor was superior for a certain situation. Identifying the advantages and disadvantages of the two sensors are extremely crucial, because our domain will be testing our real-life autonomous navigation robots in tracks with different environments. For example, the F1 Tenth race track [4] utilizes walls and barricades to create its race tracks, while the Thunderhill race track [5] consists of only the racetrack and the surrounding grass around the tracks. This is important, because LaserScan data generated from the SICK Lidar may not be useful in the Thunderhill racetrack because there would be no walls to scan data, whereas the RGB-D Images generated from the Realsense cameras in the F1 Tenth track might cause confusion for autonomous navigation using RGB images due to the similar looking walls and barricades creating the racetrack.

G-Mapping [6] uses ROS Navigation and ROS Perception packages to provide a laser-based algorithm that creates a 2-D occupancy grid map from laser and position data collected by the robot. To create the grid map using G-Mapping on the Gazebo simulator, our group launched multiple ROS nodes using "roslaunch." To test how accurately the Hokuyo lidar scans its surrounding objects, we used the default TurtleBot map which consists of different obstacles such as a trash bin, cube, bookshelf, and steel plates. After launching into the default TurtleBot map using Gazebo, we used a keyboard to navigate through the map while the lidar saves its LaserScan data into rostopics along with positional data. After scanning through the map, our group was able to save this scanned map into a ".yaml" file which consists of the map's meta-data, depicted in **Figure 1.** After saving the map file, we launched another ROS node that launches RViz, [7] a robot visualizer ROS package. This node will load the ".yaml" file created from the Gazebo simulator into RViz. Our group integrated path planning into RViz, allowing the user to input a "goal" for the vehicle to autonomously navigate using the 2D Nav Goal button, depicted in **Figure 2**.

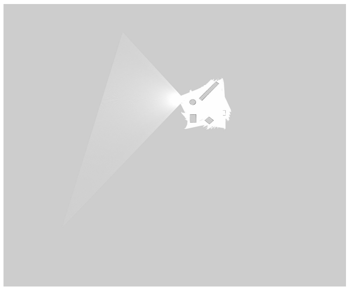
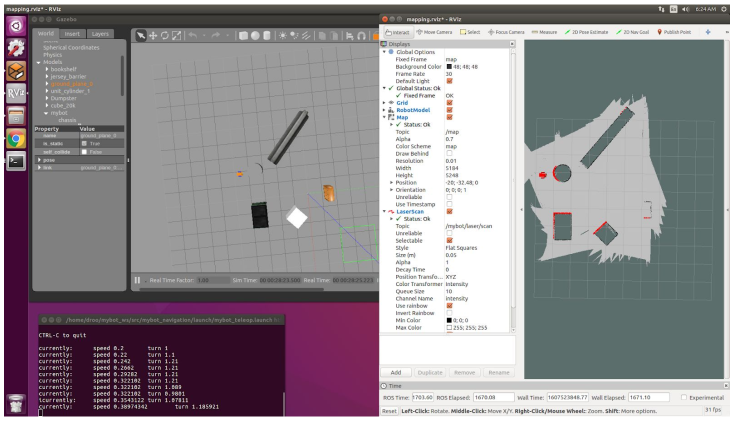


Figure 1: Visualization of ".yaml" file map meta-data



Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
Global Status: Ok	
Fixed Frame	OK
Grid	✓
RobotModel	✓
Map	✓
Status: Ok	
Topic	/map
Alpha	0.7
Color Scheme	map
Draw Behind	☐
Resolution	0.01
Width	5184
Height	5248
Position	-20; -32.48; 0
Orientation	0; 0; 0; 1
Unreliable	☐
Use Timestamp	☐
LaserScan	✓
Status: Ok	
Topic	/mybot/laser/scan
Unreliable	☐
Selectable	✓
Style	Flat Squares
Size (m)	0.05
Alpha	1
Decay Time	0
Position Transfo...	XYZ
Color Transformer	Intensity
Queue Size	10
Channel Name	intensity
Use rainbow	✓
Invert Rainbow	☐
Min Color	■ 0; 0; 0
Max Color	☐ 255; 255; 255

ROS Time: 1703.60 ROS Elapsed: 1670.08 Wall Time: 1607523848.77 Wall Elapsed: 1671.10

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel: Zoom. Shift: More options.

Real Time Factor: 1.00 Sim Time: 00 00:28:23.500

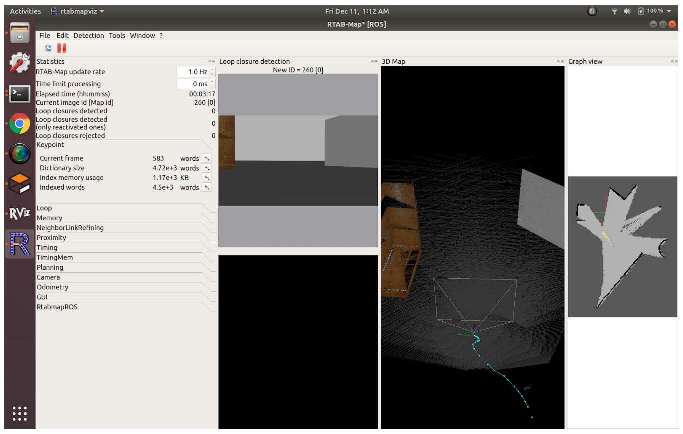
```
/home/droo/mybot_ws/src/mybot_navigation/launch/mybot_teleop.launch

CTRL-C to quit

currently:      speed 0.2       turn 1
currently:      speed 0.22      turn 1.1
currently:      speed 0.242     turn 1.21
currently:      speed 0.2662    turn 1.21
currently:      speed 0.29282   turn 1.21
currently:      speed 0.322102  turn 1.21
currently:      speed 0.322102  turn 1.089
currently:      speed 0.322102  turn 0.9801
icurrently:     speed 0.3543122 turn 1.07811
currently:      speed 0.38974342        turn 1.185921
```

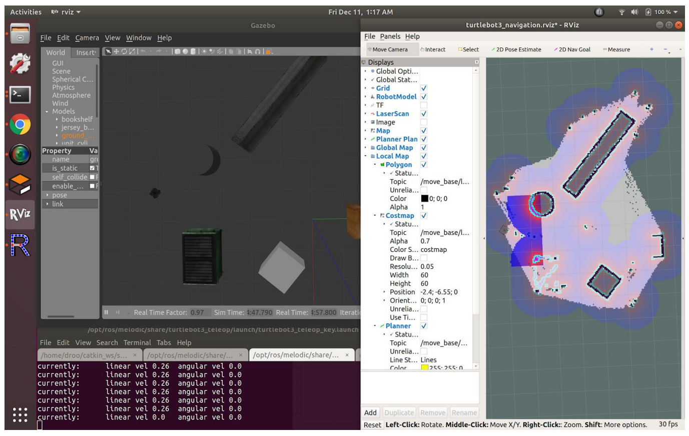
Figure 2: RViz scanned map loaded for G-Mapping

RTAB-Mapping [8] integrates a loop closure detector that uses a bag-of-words approach to determine how likely a new image comes from a previous location, or a new location. RTAB depends on the RGB-D images that are produced by the Realsense depth camera and converts the depth images into LaserScan data using the DepthImage to LaserScan package provided by ROS Navigation. This laser data along with position data allows the vehicle to generate a 2-D grid map for autonomous navigation. Similar to G-Mapping, our group launched multiple ROS nodes using "roslaunch" and also tested the algorithm's robustness in detecting its surrounding objects. This process was completed using the Gazebo simulator and TurtleBot's default map. After launching into the default map, we used a keyboard to navigate through the map while the depth camera collects raw depth images and DepthImage to LaserScan converts these images to a 2D laser scan. While navigating through the map, the RTAB package provides a visualization of the current image the camera is detecting, the 3-D Map of the car's past navigation, and a 2-D grid map that will be used for autonomous navigation, depicted in **Figure 3**. Then, this data is saved into a database, or ".db" file which stores the 3D and 2D map data generated from driving around the map. After this process, RViz can be used to allow the user to input a "goal" for the vehicle to autonomously navigate using the 2D Nav Goal button using the 2D map that was created earlier using DepthScan to LaserScan. This depiction of Rviz is shown in **Figure 4**.



```
Statistics
RTAB-Map update rate                 1.0 Hz
Time limit processing                  0 ms
Elapsed time (hh:mm:ss)            00:03:17
Current image id [Map id]            260 [0]
Loop closures detected                    0
Loop closures detected
(only reactivated ones)                   0
Loop closures rejected                    0
Keypoint
  Current frame         583      words
  Dictionary size       4.72e+3  words
  Index memory usage    1.17e+3  KB
  Indexed words         4.5e+3   words
Loop
Memory
NeighborLinkRefining
Proximity
Timing
TimingMem
Planning
Camera
Odometry
GUI
RtabmapROS
```

Figure 3: Visualization of RTAB-Map Visualizer



```
Real Time Factor: 0.97   Sim Time: 4:47.790   Real Time: 4:57.800
/opt/ros/melodic/share/turtlebot3_teleop/launch/turtlebot3_teleop_key.launch
currently:      linear vel 0.26  angular vel 0.0
currently:      linear vel 0.26  angular vel 0.0
currently:      linear vel 0.26  angular vel 0.0
currently:      linear vel 0.26  angular vel 0.0
currently:      linear vel 0.26  angular vel 0.0
currently:      linear vel 0.26  angular vel 0.0
currently:      linear vel 0.0   angular vel 0.0
```

Figure 4: RViz 2D grid map visualizing RTAB-Mapping

After deciding that G-Mapping and RTAB-Mapping are the best algorithms for autonomous navigation using lidar and depth camera sensors, our group focused on finding the best evaluation metrics to determine the advantages and disadvantages of each algorithm. To test the robustness of the algorithms, our group decided to record and compute the time needed to get from point A to point B, the number of collisions the simulated robot caused during autonomous navigation, and how efficiently the algorithm navigated its paths towards the final destination given by the user.

Once the path planning and obstacle avoidance simulations were completed, the data was logged in rostopic messages that could be retrieved. The preferred data structure to store the generated Gazebo data was a rosbag [9]. To visualize the data that was generated as well as perform exploratory data analysis (EDA), selected rostopics and messages were pulled into the rosbag before further data manipulation was performed. As mentioned prior, G-Mapping and RTAB-Mapping utilizes a lidar and depth image approach respectively to avoid obstacles and create a path. As a result of this setup, the respective rosbags for each simulation will contain data that is meaningful to what was generated. The significance of utilizing these rosbags is that the data contained can be extrapolated to evaluate the performance and results of each respective navigation.

Aside from Gazebo simulations, our group tested different navigation methods using Donkey AI [10]. Donkey AI is a small open source self driving car platform for small scale cars. In our experiment, we used the Donkey AI simulator, a unity base framework to conduct a comparison study between the performance of three common navigation methods, which are behavioral cloning, computer vision based techniques, and reinforcement learning.

Behavioral cloning method is one of the most common and robust methods to training self driving robots, which allows the robot to learn a mapping between the camera input and the output steering angle and throttle value through supervised learning. Our group first used an Xbox game controller to control the robot to collect enough training data for the training. The robot will collect images along the way. Then, we put those training data into a convolutional neural network to learn for the mapping between input and output. This method is highly dependent on the training image from the user.

Computer vision based technique is another common method for Donkey car training. The main idea is to use the camera image with some image processing techniques to extract feature information real-time from the track. This is often based on manual feature design and depends on good heuristic. One of the common computer vision techniques is to line following. This allows the robot to track either the middle line or outer track and use a PID (proportional integral derivative) controller to allow the robot to follow the tracted line. In our case, we had chosen to track the middle line for its distinguishable color of yellow. We had adopted the algorithm from [11], which first converted the RGB image to HSV color space, and then used a map to capture the range of color parameters corresponding to yellow in our image. At the end, the movement of the robot will be updated depending on the relative position of the yellow line in the camera image. This method is less robust than the behavior cloning and highly dependent on algorithm heuristic. However, it doesn’t need to have the user to spend time on the data collection which is time consuming.

Reinforcement learning (RL) is a very promising approach for action learning. It’s an artificial intelligence technique which allows the agent (the robot) to interact with the environment and receive a reward as feedback. This feedback can either be positive or negative. And the goal is to maximize its reward (faster speed, less out of track, higher stability, etc.) during its training. In order to maximize the reward, the agent must adapt its own action to the surrounding. The end to end RL learning framework is made up of a feature extract, to learn information from the observation, and the second part of RL is the regression to output the action by learning control policy. Theoretically it's an ultimate method of robotic self-learning, however, it may not converge all the time and also depend on the training policy defined in the framework. In the Donkeycar case, the agent is the car, the input is the image from the camera, and the action is the throttle and steering output. We had adopted the open source training policy from [12], which uses a reward policy of +1 for each time step stya on the track, and -10 for each time it runs off the track. Also, a penalty policy will be adopted if the car gets off the road too fast. Beside, a positive reward will be given in ratio to its throttle at each time frame. Thus, the robot will learn to stay on the track while maximizing its throttle speed.

III. Results

To evaluate both the G-Mapping and RTAB-Map approach, simulations of both models were run several times. Example simulations for both model can be found in these respective hyperlinks:

G-Mapping
RTAB-Mapping

As it can be seen in the videos, each simulation tries to move Turtlebot from the right end of the cylinder spire to the left end. The average completion speed, number of collisions and efficient pathing are the metrics used to evaluate each model or simulation. Before delving into the results, these metrics are standard metrics in the autonomous navigation space and were inspired [13] from this paper. For the purposes of this report, each of the metrics can be defined as:

1. completion speed
    - the amount of time (seconds) that it takes to traverse the distance between point A and point B
2. number of collisions
    - the count of instances where Turtlebot made contact with an obstacle
3. efficient pathing
    - the number of instances in which the local path changes due to obstacles, better pathways, or overfitting

	G-Mapping	RTAB-Mapping
completion speed (s)	63	108
number of collisions	0	0
efficient pathing	6	29

Figure 5: Evaluations of G-Mapping and RTAB-Map through defined metrics

Before evaluating the two algorithms with the metrics stated above, we must provide better information of how autonomous navigation was achieved using RViz and its 2D Nav Goal function. For G-Mapping, after the user inputs a "goal" for the vehicle to follow autonomously, a global path is shown as a blue line, which indicates the final destination that the vehicle must reach. After a global path is established, the vehicle tries to follow the global path, and the current path that the vehicle is taking to reach the final destination is called the local path, which is shown as a green line. During autonomous navigation, the global path does not change, because it is the final endpoint that the user set for the vehicle. However, the local path may change when the algorithm finds a more efficient route, or when the vehicle detects there is an obstacle in its pathway, changing directions to avoid that obstacle. This logic is identical for RTAB-Mapping, where the global path is represented by a red line, and the local path is shown as a white line.

The metrics used to evaluate both G-Mapping and RTAB-Mapping show a clear picture. Both of the algorithms were run using the Turtlebot framework and on the same simulated environment. Given these controlled variables, it can be concluded with reasonable confidence that G-Mapping performed better than RTAB-Mapping in our simulations. The average completion speed for G-Mapping was almost twice as fast and the efficient pathing was almost five times better compared to its counterpart. Number of collisions seen for both models across multiple simulations was zero, which means that obstacle avoidance for both models are satisfactory. For both algorithms, there were no collisions with the vehicle and obstacles, because when the algorithms detected a foreign object in the vehicle's pathway, the local paths were changed to avoid the obstacle. With regards to path planning, the better completion speed and efficient planning metrics define that G-Mapping is better than RTAB-Mapping given these controlled settings.

Although both algorithms showed satisfactory results for obstacle avoidance in a controlled environment, the efficiency of path planning for the algorithms showed different results. Before talking about the performance of path planning, we want to highlight that our G-Mapping algorithm was tuned to incorporate an Adaptive Monte Carlo Localization (AMCL) probabilistic system [14]. AMCL is very helpful for autonomous navigation, because it uses particle filters to track the position of the robot against a known map, allowing the vehicle to understand its current location inside the map. We believe that through this implementation, the vehicle was able to detect exactly its position against the different obstacles, allowing it to path more efficiently without changing its local path compared to the global path given by the user. Being able to identify its current location inside the map allows the vehicle to navigate with faster speeds and efficient pathing.

On the other hand, RTAB-Mapping did not receive any fine-tuning of the AMCL probabilistic system because we believed that the RGB and depth images along with LaserScan data by converting these images would be sufficient to create a high quality 2D grid map for the vehicle to autonomous navigate on. However, our group noticed that without the AMCL implementation, the algorithm had difficulties identifying the vehicle's current location on the map, the vehicle changed its local path multiple times to accommodate for its uncertainty against its current position and the location of obstacles surrounding it. Due to this result, the completion speed for RTAB-Mapping was almost two times slower than G-Mapping. Also, our group noticed that the RTAB-Mapping algorithm seems to overestimate the distance of the vehicle relative to the obstacles in its pathways. For example, there were instances where the vehicle was far away from an obstacle, but the algorithm performed local path changes, because it overestimated that the vehicle would collide with the obstacle. This example is depicted in this hyperlink: RTAB Over-Estimation.

For the evaluation of three methods with Donkey AI, we first built up visualization for monitoring the steering and throttle in real time, depicted in **Figure 7**. It also supports showing the degree of confidence for model prediction of each level of steering and throttle direction. Beside, we had adopted some similar evaluation metric as Gazebo simulation being:

1. Simulation completion speed
    - the average time (seconds) that it takes to traverse the finish five rounds in generated track
2. Number of run-offs
    - the count of instances where the robot had pass the cone obstacle during the five laps
3. Stable pathing
    - manual evaluation of the stability of its moving path (whether it's very zigzag or smooth)

	Behavior Cloning (linear)	Behavior Cloning (Categorical)	Computer Vision	Reinforcement Learning
completion speed (s)	26	28	37	N/A
number of run-offs	3	6	7	N/A
Relative Stable Pathing	Medium	Low	Low	High

Figure 6: Evaluations Donkey AI Navigation Methods

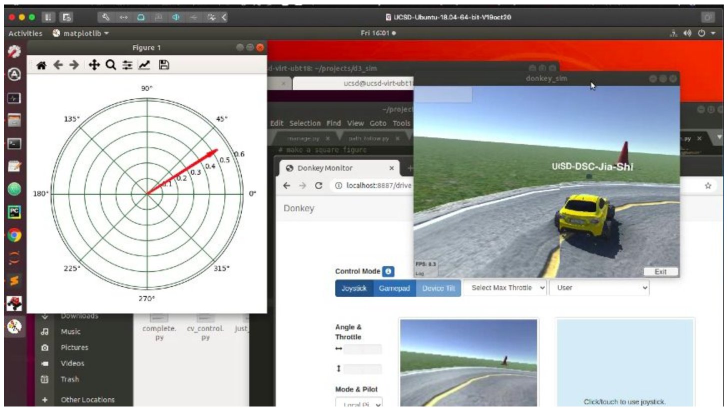


Figure 7: Real-time monitoring of Steering/Throttle

For our experiment, we used the code for behavioral cloning, and had referred to some open source codes for the implementation of computer vision base technique and reinforcement learning.

For the behavior closing, we adopted two ways of training, the first one is linear model, whose convolution network had output two header of steer and throttle, and the second one is categorical model, which outputs the probability for the model to predict each direction of steering and throttle. That is 16 classes for steering and 20 classes for throttling. Compared to the linear model, it performs worse because it has a harder training test thus will be harder to converge and less stable than the linear model.

For our computer vision model, we chose to follow the middle yellow line. In general, this is not as stable as other methods. Also, it's the slowest in terms of completion time since we chose to reduce the throttle speed in order to make it more stable. However, there's still some corner case where the vehicle will make some unpredicted movement and run out of the track directly. As I mentioned above, this is highly dependent on 'messiness' of the background itself since it is based on RGB color image input, and it also highly depends on a good heuristic, since it's not a learning base method. Thus it's expected not to perform as good as other navigation methods.

We adopted the open source code directly in our experience for the RL. By now, we are still configuring the framework of reinforcement learning and it's expected to be finished soon. Theoretically and based on observations, reinforcement learning should perform the best among all the methods in terms of stability and efficiency, because it optimizes the learning objectives by high level policy rather than specific design features(like computer vision). Also, rather than learn from humans directly, like the behavior of cloning), it would not inherit from the mistakes and errors of humans, which is a more optimal solution from a statistical perspective.

IV. Discussion

Recently, our group was able to collaborate with group one, where Siddhartha created the Thunderhill race track [15] on the Gazebo Simulator. Although it is a very simple map with no obstacles, it allowed our group to test RTAB-Mapping on the race track. Using the RTAB visualizer application, we were able to create a database file containing the depth camera images, 2D and 3D maps for autonomous navigation, depicted in **Figure 8.** Through this success, our group is planning to test out G-Mapping on this race track, use RViz to allow users to input "goals" for the vehicle to autonomously navigate, and add obstacles such as construction cones to test efficient obstacle avoidance in the race track. We believe that it is essential to find Gazebo simulation maps that are similar to the F1 Tenth and Thunderhill race tracks to test our algorithms and integrate them into our real-life robots for adequate autonomous path planning and obstacle avoidance.

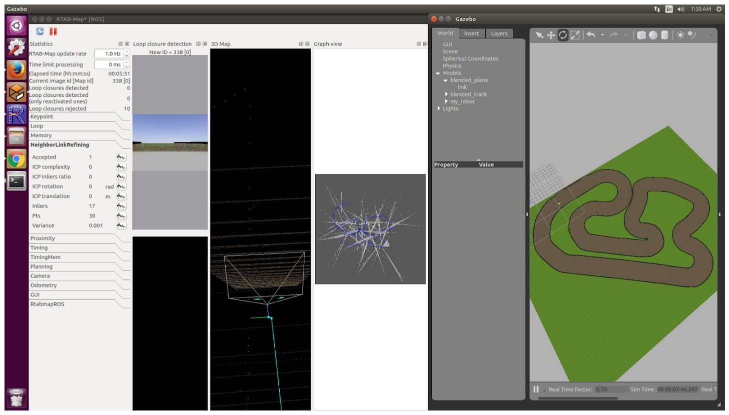


Figure 8: RTAB Visualizer depicting Thunderhill Track

In the field of obstacle avoidance, path planning, and autonomous navigation, the technology used is constantly evolving. GPS systems, lidar technology, camera depth and image capturing capabilities are all constantly improving. Over the course of the last few decades, it can be noted that computational power and efficiency has improved. This improvement in computational power has greatly helped to advance developments in both software and hardware in relation to autonomous navigation. It is probable that with the development in artificial intelligence and supercomputers that obstacle avoidance and path planning will improve in the coming years [16]. If artificial intelligence and supercomputers continue to advance, these technologies can greatly help to improve both deep learning/behavioral cloning and traditional programming models in numerous ways.

V. Conclusion

In this report, our main objective was to research and implement different autonomous navigation algorithms into simulators such as Gazebo and Donkey AI to test their efficiency and performance for real-life integration. Our group successfully integrated the Intel Realsense D435 camera into Gazebo, and was able to test G-Mapping and RTAB-Mapping using RViz and other ROS Packages such as DepthImage to LaserScan and AMCL. We were also able to test the efficiency of three different algorithms using Donkey AI, which will be extremely useful for our real-life vehicles. Different metrics were created for both Gazebo and Donkey AI simulated algorithms, which allowed our group to determine which algorithm would be more useful for different tracks such as Thunderhill and F1 Tenth. By assessing these different navigation techniques using simulators, our group was able to find the advantages and disadvantages of each algorithm in advance, allowing us to smoothly transition into our real-life robot's path planning and obstacle avoidance.

VI. References

[1] Intel Realsense Depth Camera
https://www.intelrealsense.com/get-started-depth-camera

[2] TurtleBot Robots wiki
http://wiki.ros.org/Robots/TurtleBot

[3] Gazebo Simulator
http://gazebosim.org/tutorials?tut=ros_overview

[4] F1 Tenth Racing
https://f1tenth.dev/

[5] Thunderhill Track Racing
http://selfracingcars.com/
[6] G-Mapping wiki
http://wiki.ros.org/gmapping

[7] RViz wiki
http://wiki.ros.org/rviz

[8] RTAB-Mapping wiki
http://wiki.ros.org/rtabmap_ros

[9] Rosbag wiki
http://wiki.ros.org/rosbag

[10] Donkey AI
http://donkeycar.com/

[11] Computer Vision Line Follower (2020), Github Repo.
https://github.com/autorope/donkeycar/blob/4580ec748
00db2a9e4288d49c36c41b1c2077c42/donkeycar/templ
ates/cv_control.py

[12] Learning To Drive In 5 Minutes (2020), Github Repo.
https://github.com/araffin/learning-to-drive-in-5-minutes

[13] Calisi, Daniele and Nardi, Daniele. "Performance evaluation of pure-motion tasks for mobile robots with respect to world models." 2009,
https://www.researchgate.net/publication/200744624_P
erformance_evaluation_of_pure-motion_tasks_for_mob
ile_robots_with_respect_to_world_models

[14] Adaptive Monte Carlo Localization (AMCL) wiki
http://wiki.ros.org/amcl

[15] DSC180A Slam Simulator (2020), Github Repo.
https://github.com/sisaha9/dsc180a-slam-simulator/tree/
main?fbclid=IwAR3b-jkkB0VUGdEm4pWDdQxwG3
R6gI8_n23VGCdmtp7Ll94vEog-EgCivEU

[16] A. A. Zhilenkov and I. R. Epifantsev. "Problems of a trajectory planning in autonomous navigation systems based on technical vision and AI." 2018,
https://ieeexplore.ieee.org/abstract/document/8317265