TASK 1:

Code regarding this task is located in **transformation.cpp**, **marker.cpp, marker.hpp**. To see transform of map frame called **/real_robot_pose** you can open **frames.pdf**. To see true robot location and orientation and to see robot pose obtained from odometry open **rviz** and add **/real_and_odom_poses** topic. This topic has two different namespaces, one for real pose and one odom pose.

TASK 2:

Code regarding this part is located in **path_planner.cpp, AStar.cpp, AStar.hpp.**

For path planning I chose A* algorithm, because it's one of the best algorithms for path finding, regarding performance and memory usage. I also wanted to have pixel perfect path so instead of slicing map into robot size grids I left its normal representation. Also, because wanted my A* to find the shortest path possible, with minimum number of turns I choose to use A* which would try to expand path in all 8 directions. For A* heuristics I used Euclidean distance.

After first implementation, I realized that with things I want to achieve (**pixel perfect** path and expansion in **8 directions**) simple A* implementation won't work, even though I'm using C++. It took around **10 seconds** to find path in ~**1-meter range**. After that I found my biggest bottleneck: for each neighbor (8 of them) I had to check if neighbor was in closed list (which means I have to skip it) in my naïve implementation I stored all the closed neighbors in list and then did search on that list, which could be (map width * map height – 8) size in worst case and searching through it would be **O(n+m).** So instead of storing closed nodes on list I created basic cache type array of map size. Now to find if neighbor node was closed was **O(1),** because it was only simple lookup operation on 2d array. So, I had to sacrifice application memory for pretty big array (800 000 elements) for better speed performance which I think perfectly suits my situation as I am more constrained on performance than memory usage.

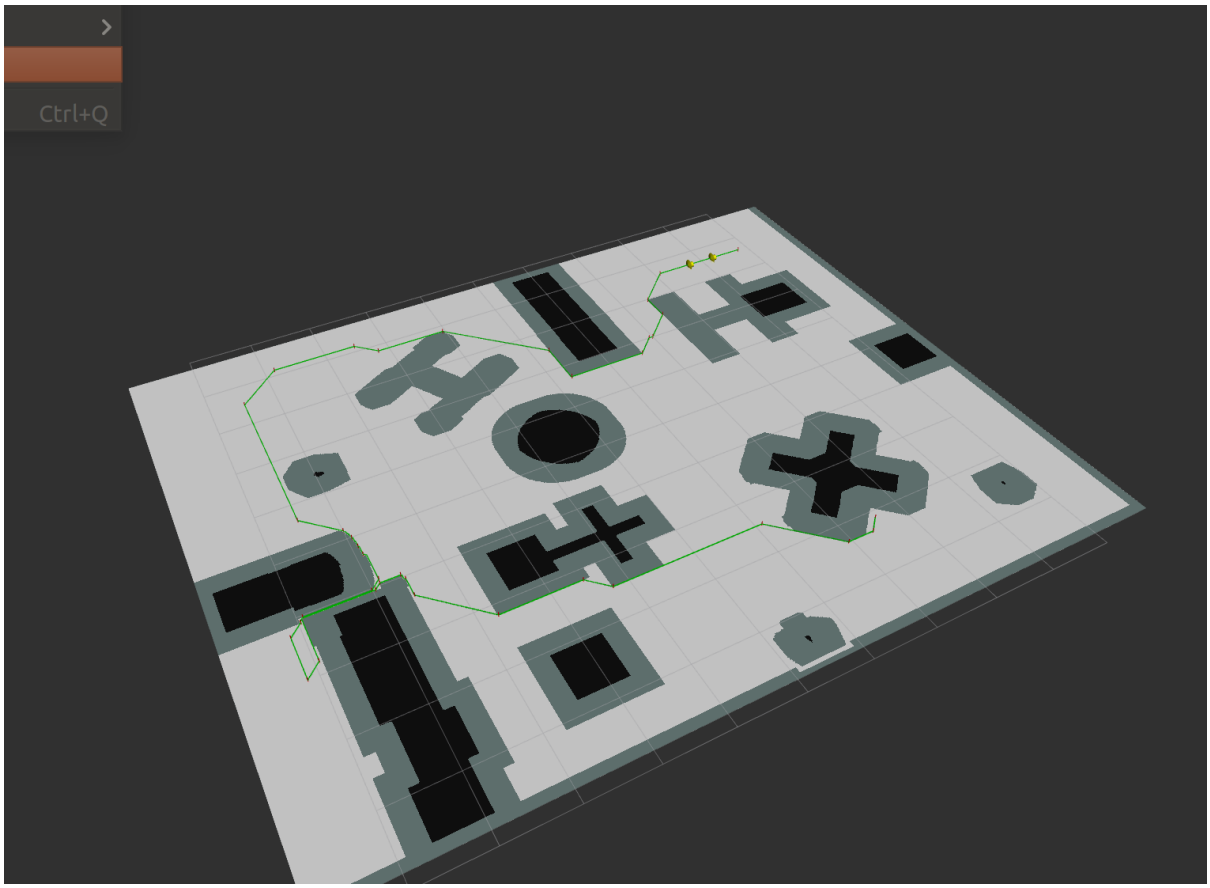
A* performance increased drastically it would take around **2 seconds** to find path in **~1meter range**. However, to find longer paths algorithm was still not as fast as it could be. So, I inspected algorithm again to find another place which gets called a lot and could be improved. That's when I found that my open nodes list implementation was too naïve as well. I was just storing all open nodes in array. And then to find node with lowest cost, I would search through that list, which again in worst case could be **O(n*m).** Because I was extracting lowest value more often than inserting it, I choose to use priority queue implemented with min-heap instead of plain list. So now it took **O(1)** to find node with lowest cost. This improvement made algorithm around **~2 faster**, which made it pretty good for finding distance between two points.

However, to find path between all **6 points** it took more than **1 minute** which is not ideal. The problem was that long and hard path between two points would block search for other points which are relative close and finding path between them would be relatively fast. My solution to that was first use Euclidean heuristics to find points that are close to each other. Then spawn thread for each pair of points and run A* between those points on separate thread and finally merge all points into

one array. By doing that I decreased path planning for all points from over **1 minute** to **26 seconds**, which looked reasonable to me.

For future improvements, I was thinking that it would be possible to do bidirectional A* that would start two threads and search for path from start and end at the same time. One thread would store its current lowest cost node x and y index on shared memory and when other thread has lowest cost node at the same index paths, you would join threads and merge paths.

Another part of path planning was to make path that would make sure that robot won't hit obstacles. To achieve that I inflated border walls by robot size. However, this had one drawback: if there was space that was smaller than robot size * 2 and it had walls from both sides, walls would be expanded from both sides and robot would think that there is no path, even though robot could fit. To fix that I changed my expansion algorithm to scan map for small spaces which are smaller than (robot size * 2), but bigger than robot size. After finding those spaces I added them to exception list and then when I was expanding obstacles, if obstacle was in exception list I would expand it by robot size / 2.



Map in rviz after obstacle expansion with planned path

TASK 3:

Code regarding this part is located in **driver.cpp, geometry.cpp, geometry.hpp**

Driver node receives path from **path_planner**, which before sending path extracts only important waypoints which indicate change of direction. Then normal driver starts driving, while driving laser scanner gets information from environment, depending how close and in what degree obstacle is, driver might change its driving mode. There are 3 different modes: normal, bug, narrow.

Normal driver just follows given path, from one waypoint to another. **Bug driver** tries to avoid obstacle by following obstacle to right and after obstacle is behind changes mode back to driver. **Narrow driver** is used when there are 2 close obstacles from both sides. It tries to correct the angle and drive past obstacles, after that normal driver takes responsibility. By combining both open loop (planned path) and closed loop (laser scan information adoption) robot is able to drive through the map.

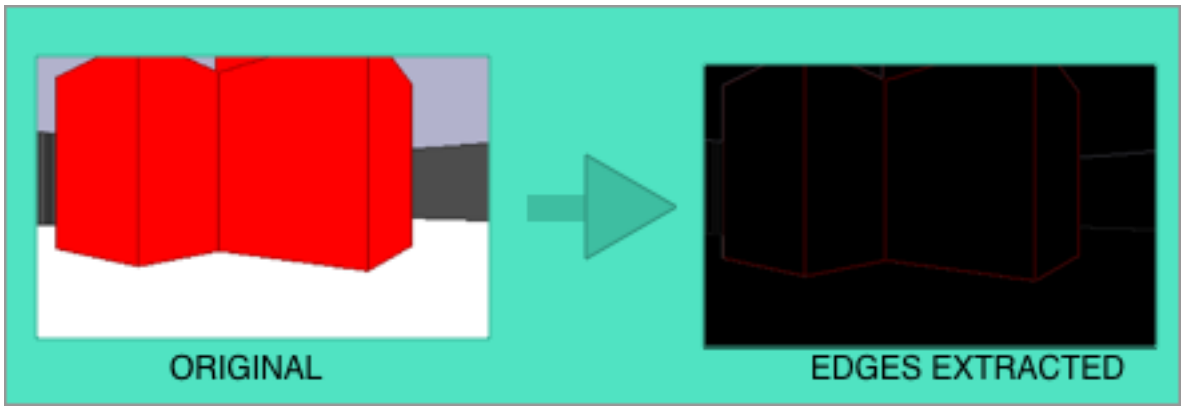
TASK 4:

I used already existing ros package called AMCL, which uses Monte Carlo localization approach, with likelihood field model. However, localization is not super accurate especially when dynamic obstacles are encountered. I think it's mainly because how odom tends to drift really far away after accumulated errors.

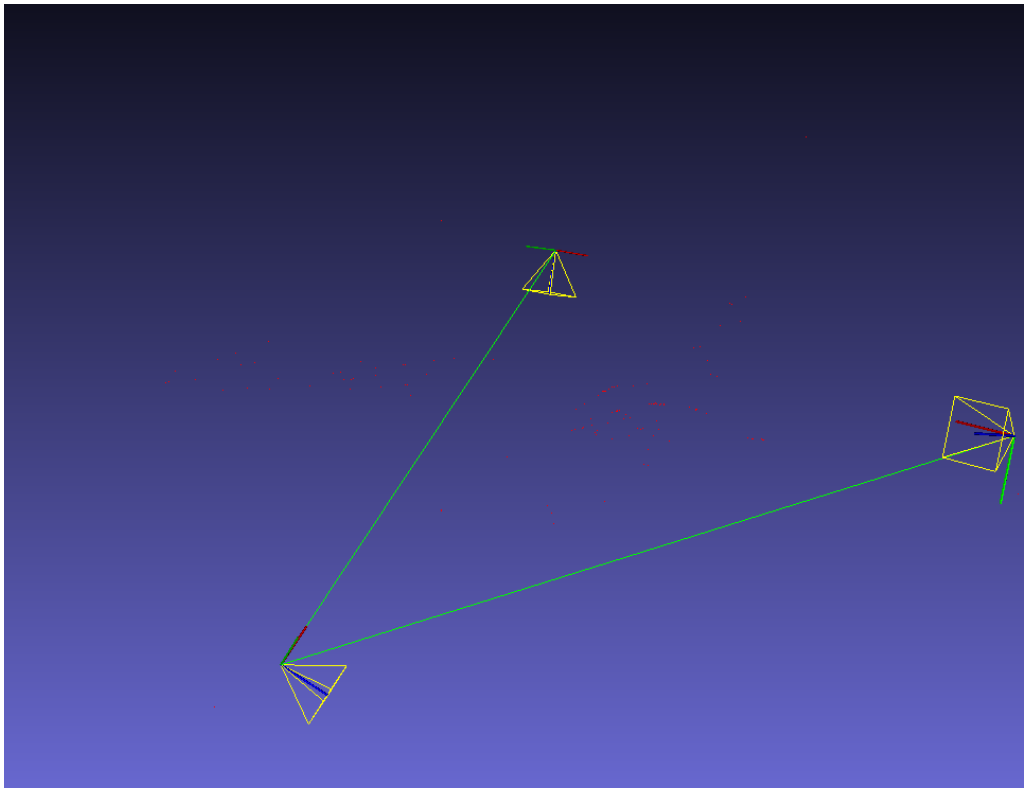
TASK 5:

For this task, I found a really cool module in OpenCV called **sfm**. However, because it is extra module it wasn't include with ROS OpenCV library. I tried to add it to ROS OpenCV library, but it didn't work. Then I downloaded OpenCV and built it locally on my computer. However, after trying to link it with ROS I started to get linking errors with Eigen library. After spending 3 days trying to make it work with ROS, I chose to try to use it locally without ROS. So, I went to stage simulator took few pictures of red obstacle in the map. Then I took readings from **/camera_info** topic to get intrinsic constraints.

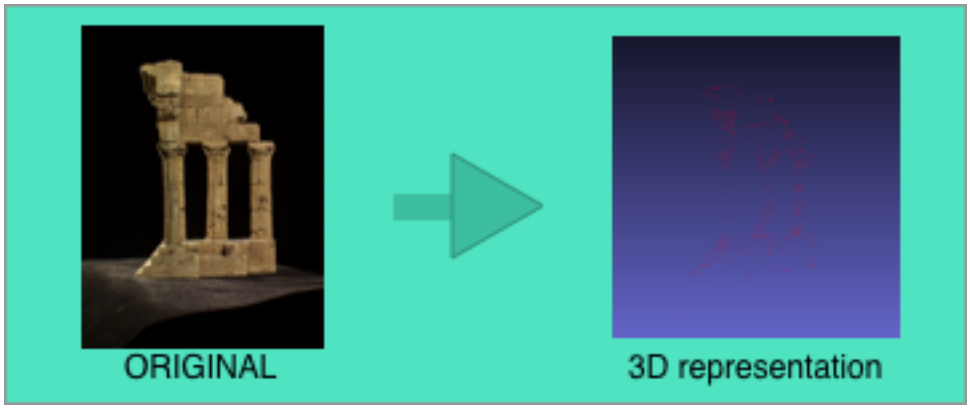
Then I tried to reduce noise with blur filter and use Canny edge detection to extract edges.



And after that I was trying to use it with **sfm** module, however results were really bad. I think it was, because my images were misaligned, as you can obviously see from bottom image.



Just to make sure that code works I tried it with images downloaded from middleburry.edu (http://vision.middlebury.edu/mview/data/). As you can see it was way more accurate.



To see code related to scene reconstruction in 3D go to image_processing and open file **main.cpp** to run please install and link OpenCV with sfm module and Ceres Solver.