Introduction to ROS – Autonomous Drones

Group 3: Haoxuan Li, Yinghan Huang, Jingkun Feng, Tao Zhou, Xuhui Li

1. Introduction

With this project we aim to perform cooperative unmanned aerial vehicle exploration in an unknown environment in ROS. The environment is generated and simulated in Unity. Our implementation can control two drones to explore this environment and generate a 100m x 100m 3D voxel map with the voxel resolution of 1m x 1m x 1m.

The subsequent part of this documentation is structured as follows:
In session 2, we introduce an overall structure of our program and indicate the cooperation between different components. The modules for different subtasks are introduced in Session 3. In the final session, we sum up our projection by providing some interesting results and discussing the limitations of our program.

2. Overall Structure

Our implementation contains 6 main modules:

- The simulation module reads Sensor data from unity and sends commands to it.
- **Mapping** reads camera frame and depth image and publishes a global octomap and a projected 2d map. (Session 3.1)
- **Planning** takes the map as input and listens to commands from exploration to perform path planning. (Session 3.2)
- The **state machine** monitors drone state and user input and triggers control and exploration according to the current state. (Session 3.3)
- **Controller** reads the drone state from simulation and sends commands back to make sure that the drone follows the path or commands from the state machine. (Session 3.4)
- **Exploration** takes the current generated map and drone state to decide where the next goals are for drones to explore. (Session 3.5)

Figure 1 shows roughly how different modules communicate with each other while Figure 2 is a ROS node graph of the project, which provides detailed information including nodes and topics.

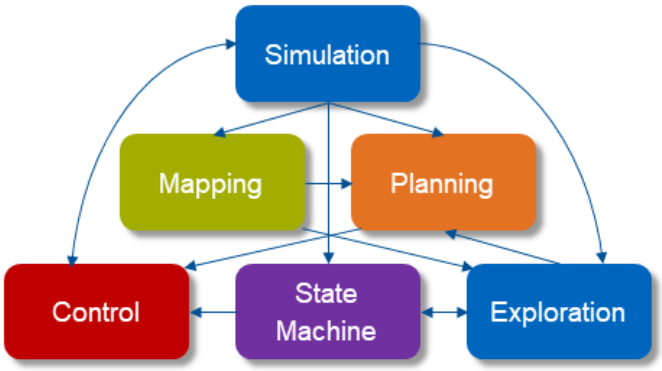

Figure 1. Communication between modules

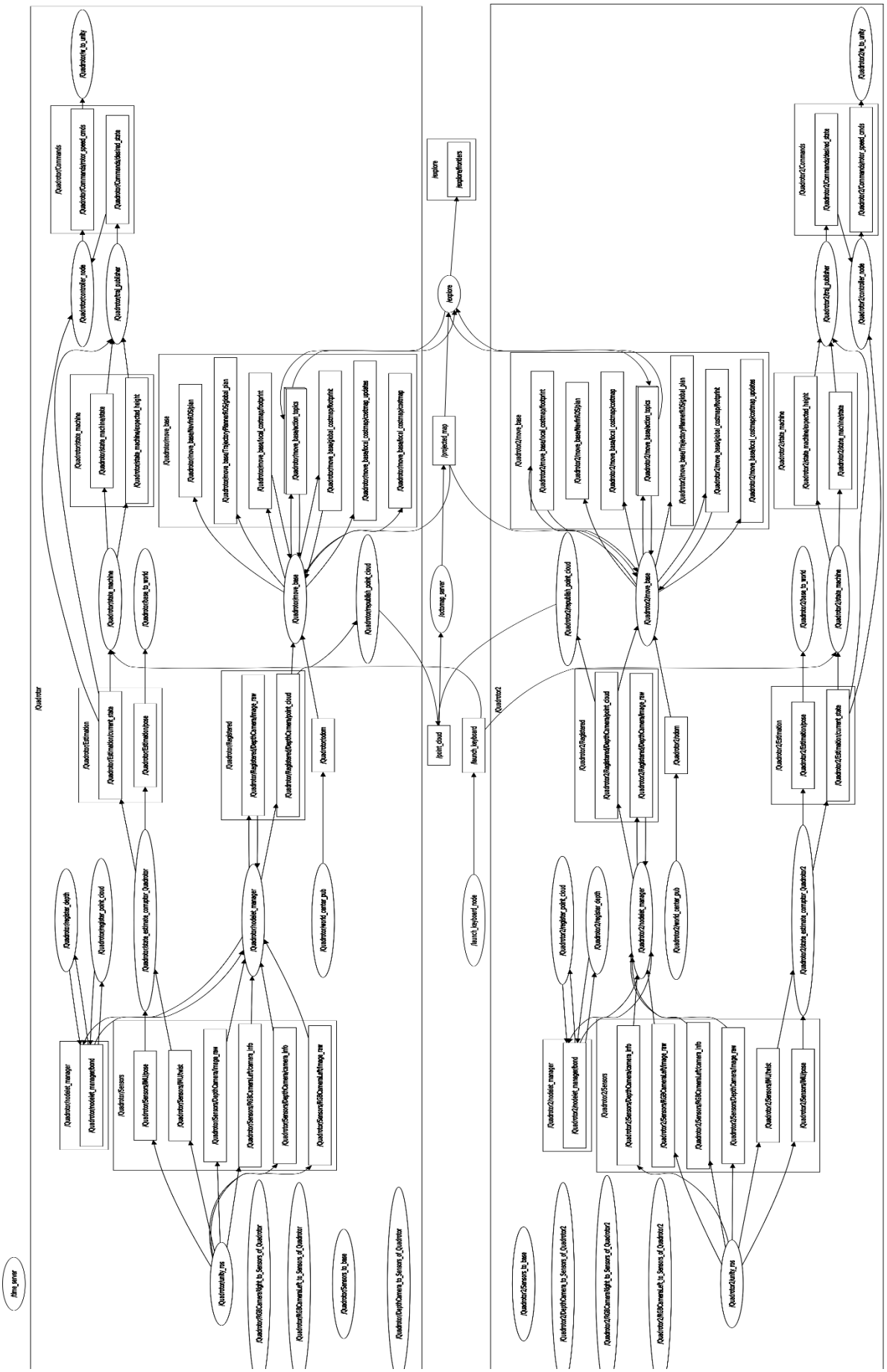

Figure 2. ROS node graph generated when two drones are running.

3. Main Components

In this session we introduce the main components of our program providing basic concepts and methods we use as well as naming its main contributor.

3.1. Mapping

We want an octomap with color to show a real world and we also want to project a 2D map through the generated 3d map (octomap with voxel grid) to prepare for the later exploration.

First, we subscribe to depth camera and RGB camera to publish a point cloud topic(/point_cloud). The associated code located in *"point_cloud.launch"* in map.

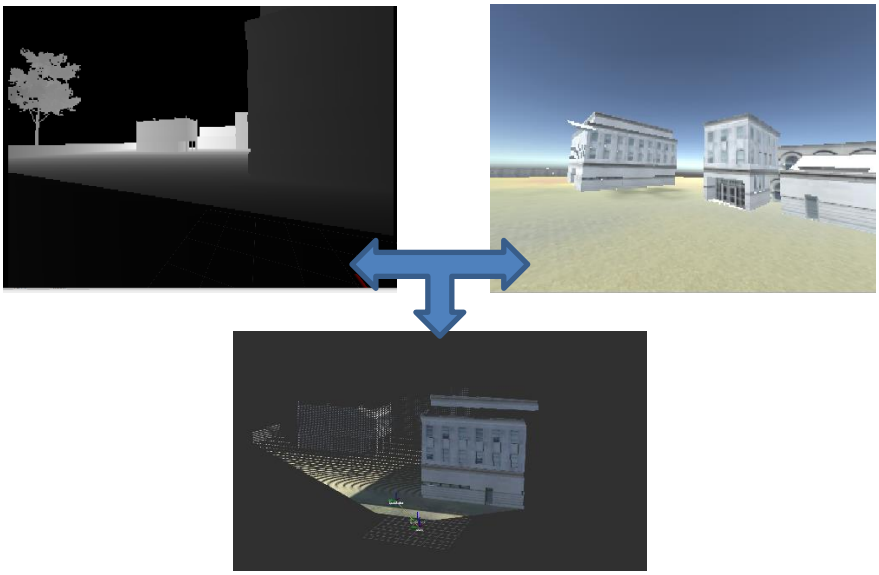

Figure 3. Combine the depth camera and RGB camera to generate the point cloud with RGB

We generate the octomap by subscribing to the topic of the point cloud. The associated code and parameters located in *"map_service_drone.launch"* and *"map_service.launch"* in map Generally, we need to filter the ground to generate a suitable 2D Projection map. Through practice, it is found that due to the limitation of the performance of our computer, we need to limit the scanning range of the point cloud in the next x, y, and z when generating a three-dimensional octomap. At the same time, resolution is also very important. If its value is too small, the speed of generating the map will be very slow. Through practice, we set it to 1. By setting these parameters, we can reduce the amount of calculation for generating octomap. The 2D Projection map and octomap will be generated more quickly to prevent hitting walls [1], [2]. At the same time, to simulate the appearance of a real world, we give the voxel grid color through parameter settings. The associated code and parameters located in "*octomap.launch"* in map. The generated octomap will publish a topic "*project_map*". This topic will be used to complete the corresponding task when exploring later. We can complete the construction of the 2D Projection map and octomap in about 40s.

3.2. Planning

The planning node is developed by Xuhui Li. It receives the projected map that is generated previously, meanwhile subscribes to a topic that publishes the goal point in global world frame. Then it publishes the generated global plan for further exploration.

There are totally 6 different kinds of yaml-files that would be called when running the launch file. They contain different parameters about either information of our drone and sensors, or

information of the global/local costmap. Some of these files are responsible for both drones, while some are not. These files that are not responsible for both drones are used twice with different parameters for different drones.

"base_local_planner_params.yaml" contains parameters and limitations that would be considered during pathing generating.

"costmap_common_params.yaml" contains parameters that should be used when sensing the unknown environment and generating costmap.

"global_costmap_params_drone1.yaml" and "global_costmap_params_drone2.yaml" contain parameters about global costmaps, e.g. their coordinate frame, update & publish frequency.

"global_planner_params.yaml" contains parameter that is needed when global planning, most of them stay in default value.

"local_costmap_params_drone1.yaml" and "local_costmap_params_drone2.yaml" contain parameters that are similar to global costmap parameters.

"move_base_params_drone1.yaml" and "move_base_params_drone2.yaml" contain move_base node parameters.

All these files are from internet, and we tuned the parameters inside so that they work well with our task.

3.3. State machine

The state machine node is mainly accomplished by Yinghan Huang. Subscribing the message provided by the TCPStream, state machine cooperates with the trajectory publisher (section 3.4) and generates desired pose and velocity in different states.

The keyboard control node supervises the keyboard input in the terminal where the simulation is launched, user can press the space key to launch drones to individual expected height (which user can specify in"state_machine_params.yaml"). Then the state machine will call the "/explore/switch_explore" service in the "explore_lite pkg", where the start function in explore lite will be triggered and begin exploration and map construction. Besides, two exploration modes are provided – one is explored with self-rotation of drones, and another is explored while the poses change according to the trajectory direction. These 2 modes do not make much difference in the exploration time. However, more details about the environment can be constructed in the final octomap with the rotation of drones. And changing with trajectory direction is a more practical motion, so we leave this for users as an option.

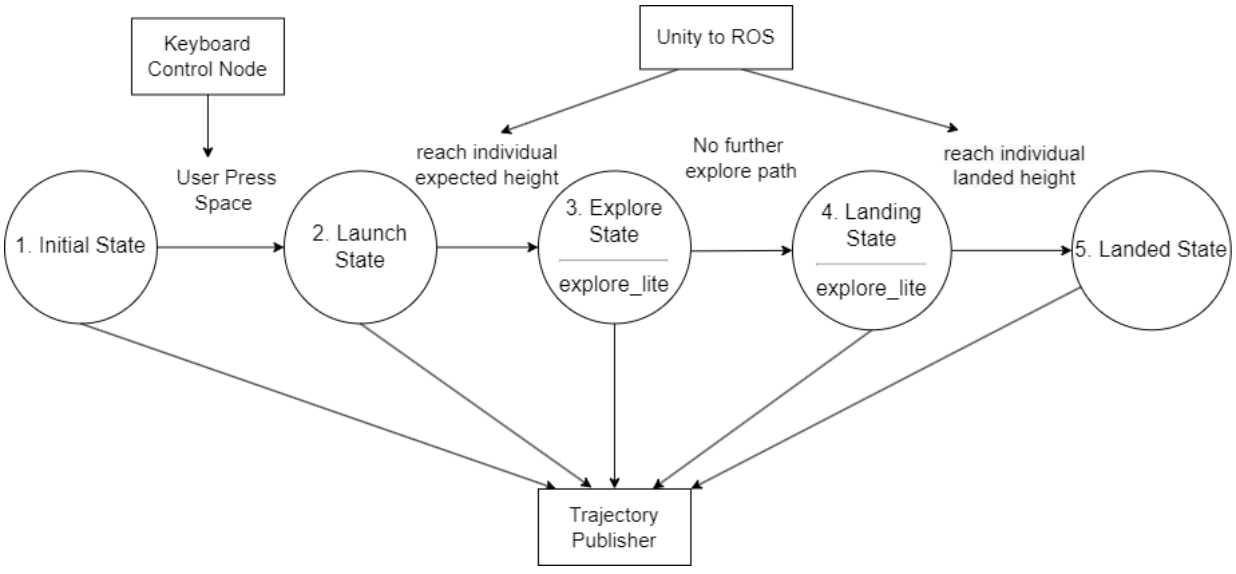

Figure 4. State machine

When the explore_lite cannot find further paths, explore_lite would call the “stop service” in the state machine and translate into the landing state, and the drone will land in the place where it stops exploring.

3.4. Controller

The controller node is based on the work of Prof. Markus Ryll and further developed by Jingkun Feng. It takes charge of controlling the drones such that they can follow the generated path and fly smoothly. It utilizes the concept of a cascade PID controller und consists of two main components.

The first component takes the path generated by the Planning node and the current states of the drone including the pose and velocity and computes the desired poses (x-, y-, z-coordinates in world frame and yaw) and the desired velocities. The associated code located in “*traj_planner.h”* as well as “*traj_planner.cpp”* in the “*controller_pkg”*. User can change the P-, I-, D-gains correspond to individual control values by modifying the “*pid_param.yaml”* file.

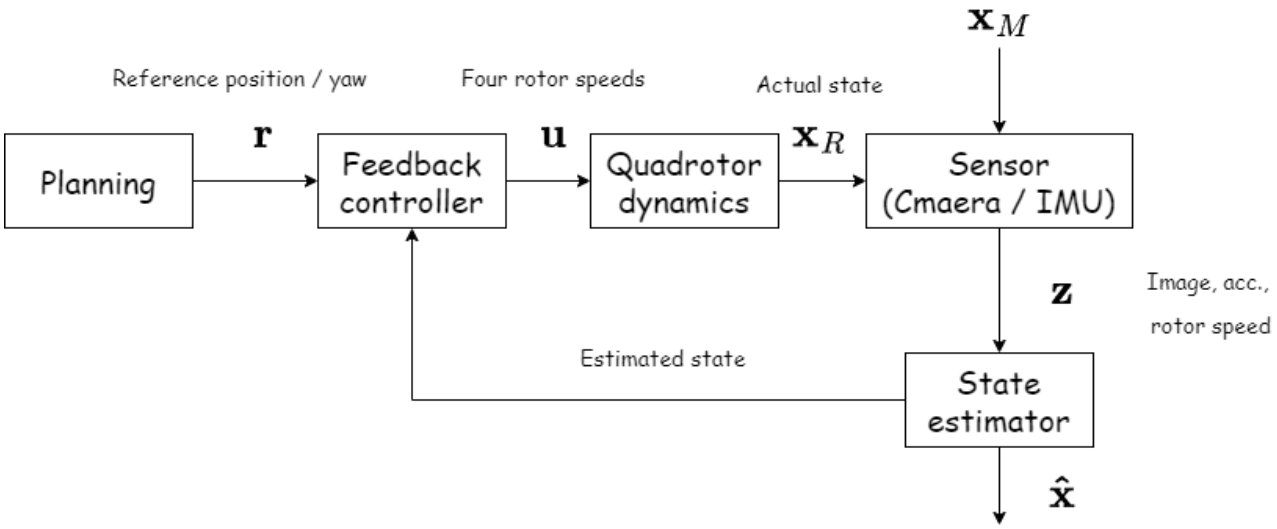

Figure 5. PID drone control

In addition to the basic functionality of a controller, this component provides two motion modes of the drones. One is pure path following by changing the drone orientation according to the path direction. The other is rotating in yaw while following path. With the first mode, the drone flies more stable. However, it always passes by the buildings very quickly and not enough observations are obtained. Therefore the mapping is mostly bad. Because of that the second mode is developed. This mode rotates the drone continuously in yaw while following the generated path. It enables the drone to gain more and better observation of its surroundings and thus able to produce better mapping. The variable to change the mode is in “rotation.yaml”.

The second component is the propeller speed controller. It makes use of the output of the first component and generates the velocity of the four propellers. This component is developed by Prof. Markus Ryll based on [3].

To tune tha parameters, we refers to the rule of thumb described in [4]. However, the parameter tuning is cumbersome and complex, the version we provided is the best version we tuned but it is far away from perfect.

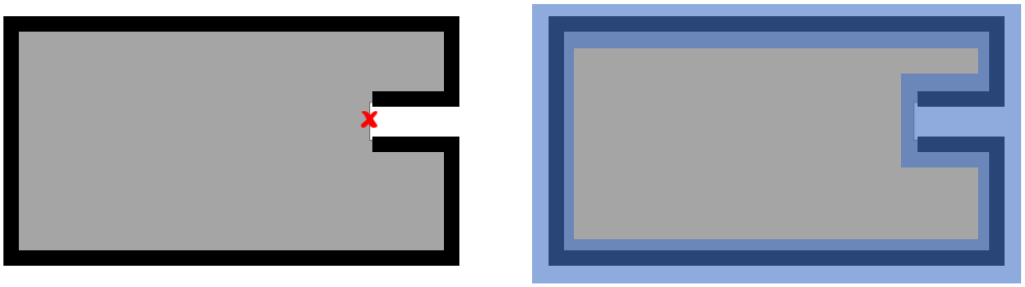

Figure 6. Usage of inflected map in explore. The unknow area is marked with gray and known obstacles are marked in black. On the left we have a projected map with no inflection, so the explore goal could be set to the narrow area, which is marked by a red cross. It is likely that the drone crashes into obstacles around it when exploring this narrow area. On the right we have inflected map, the near range of the obstacle is marked in blue, in this case, none of the frontier in the blue area is considered as a goal.

3.5. Exploration

The exploration part is implemented by Haoxuan Li based on explore_lite [5], [6] and costmap_2d packages. We use the idea of inflected map and target filtering to optimize for our task.

The inflected map is generated with the projected map provided by the mapping module. As shown in Figure 6, the near range around obstacles will not be considered as a target, which helps to prevent the drones from crashing into the walls.

The second key idea is target filtering. To optimize the performance of explore_lite, and to make it compatible with multi robots, we add filters to both frontier and goal search algorithm. We add parameters to set the limit of the explore range, so that all potential frontier points outside the interested region are ignored. For the goal position, we not only add positional filters, but also filter out the goals that are already assigned to other drones. This is done by maintaining a list which stores the current target of all drones, as shown in Figure 8. With these two filters, the explore_lite is capable of scheduling multiple drones to explore limited areas effectively.

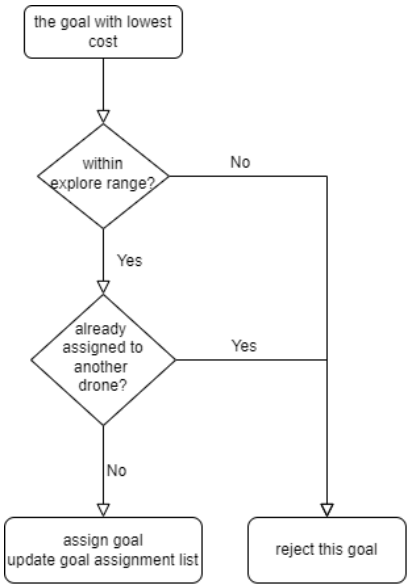

Figure 7. Filter the goals in explore_lite

Besides the adaptation of the algorithm, we also implemented a service server and a service client in explore_lite, so that it can accept orders from state machine to start and stop the exploration. After the exploration is complete, it is also able to call service of state machine to trigger the landing phase.

We also implemented another explore strategy of exploring. As shown in Figure 7. The original explore_lite uses the geometric center of frontiers as goal positions. In the new strategy, the goal position is set to the nearest point on the frontier. This strategy has several advantages. First, it is set on the frontier, so the goal position is either an obstacle or reachable point. When using the original strategy, the goal position could be set somewhere in the middle of the unknow area, which might be impossible to reach, which increases the risk of crashing. Second, if we have multiple drones, a frontier can be explored by them simultaneously, because the nearest points to different drones can be different. The original strategy on the other hand, has only one possible goal location, which means a single frontier cannot be explored by multiple drones at the same time. However, the drawback of this approach is also obvious. The goal is very likely to be set on an obstacle, as shown in Figure 8. For our specific task, the trajectory planner and the map server are not fast enough to react, which could cause crashing. Considering the success rate of exploration, we keep using the original explore strategy.

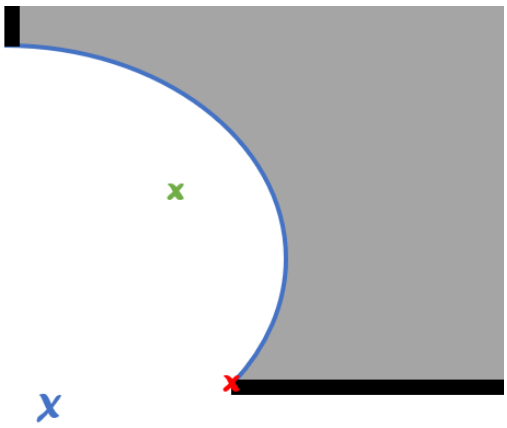

Figure 8. Different explore strategy. The blue cross represents the drone, the black range represent obstacles, the grey range represent unknown area, while the blue curve represents the frontier. The green cross, which is the geometry center of the frontier, is set as goal in original explore strategy. The red cross, which represents the nearest point to the drone on frontier, is set as goal in the new strategy.

4. Conclusion

In this project we implement a program in ROS which controls two drones to explore an unknow Unity environment and create a voxel map with the grid resolution of 1m x 1m x 1m. Perception and mapping is completed by the mapping node, while planning and explore node are responsible to generate goal and plan a path to it. The dynamics of the drone is controlled with the controller node so that it can follow the generate path stably. Meanwhile, the state machine node control and monitor the states of a drone managing the takeoff, path following and landing. A demonstration video is posted on YouTube. Some screenshots of the generate voxel map and costmap are shown in Figure 9 and Figure 10.

Although our implementation can complete its task mostly, there are still some problems users may observe. One common problem is that sometimes the drone will still hit the obstacles because our implementation is not able to update and generate the costmap quick enough. Lack of map information it is not possible for the planning and explore node to export a safe path

without passing through or getting close to obstacles. In addition to accelerate the map generation process, we can also include an obstacle avoidance mechanism to avoid collision. Another problem users may observe is that the landing performance differs on different device. Although we have an PID controller which performs good path tracking. The drone position jitters in the z-axis. Four of our team members observed small jitter and the drones perform landing without problems while one team member observed large jitter and the drone was not able to land on his device. We proposed a solution by adding an offset in the z-coordinate of the desired position. This solves the landing problem but may threaten the maneuver.

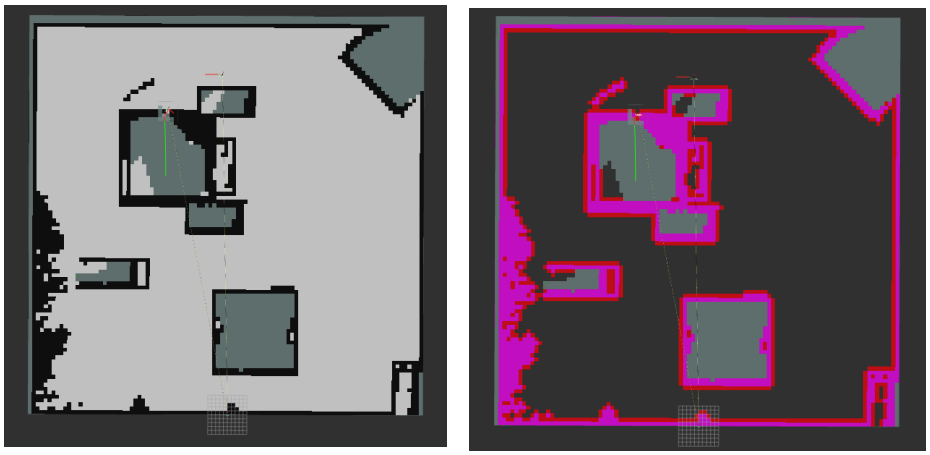

Figure 10. Generated projected map (left) and costmap (right).

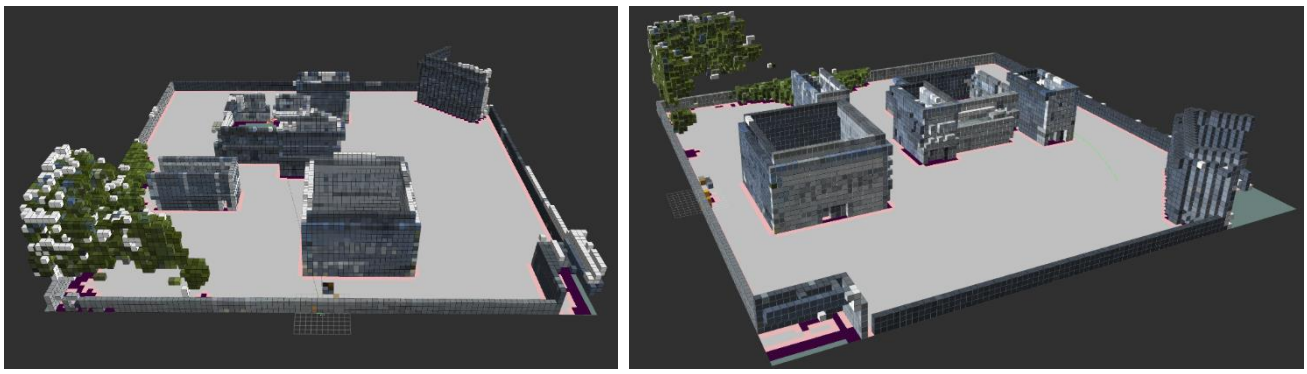

Figure 9. Generated voxel grip representation of the environment.

Bibliography

[1] “gmapping - ROS Wiki.” http://wiki.ros.org/gmapping (accessed Aug. 02, 2022).

[2] “octomap_mapping - ROS Wiki.” http://wiki.ros.org/octomap_mapping (accessed Aug. 02, 2022).

[3] T. Lee, M. Leok, and N. H. McClamroch, “Geometric tracking control of a quadrotor UAV on SE(3),” in *49th IEEE Conference on Decision and Control (CDC)*, Atlanta, GA, Dec. 2010, pp. 5420–5425. doi: 10.1109/CDC.2010.5717652.

[4] “Setting the PID controller of a drone properly.” https://www.technik-consulting.eu/en/optimizing/drone_PID-optimizing.html (accessed Aug. 02, 2022).

[5] “explore_lite - ROS Wiki.” http://wiki.ros.org/explore_lite (accessed Aug. 02, 2022).

[6] J. Hörner, “Map-merging for multi-robot system,” p. 49.