Autonomous Navigation and Mapping with a UAV*

*Note: Group project for the lecture Autonomous Systems offered by Prof. Markus Ryll (TUM)

Shervin Koushan Weihang Li Junpeng Hu Luca Obwegs Xuan-Pu Autumn Hong

***Abstract*—This paper presents our approach to enabling a UAV (unmanned aerial vehicle) equipped with a depth camera to autonomously navigate, explore and map an unknown environment. The depth image from the UAV is converted into point clouds, which is then used to create a map represented by Octomap. We perform global planning on the continuously updated occupancy grid map to generate a set of collision-free waypoints the drone will follow.**

I. Introduction

Our goal is to develop an autonomous UAV system that can take off, fly among buildings without collision, explore the unknown environment and build a map simultaneously. The drone has depth image as perception input and localization information is available. To tackle the above mentioned tasks, we formulate the workflow in fig. 1. The quadrotor is controlled by the rotor speed. Our system is built with the Robot Operating System (ROS) and the simulation is achieved with Unity3D. The navigation will be performed in 2 dimensions, as we assume the height of the drone to be fixed. The final map is in 3D, and can be stored as a binary file for later use.

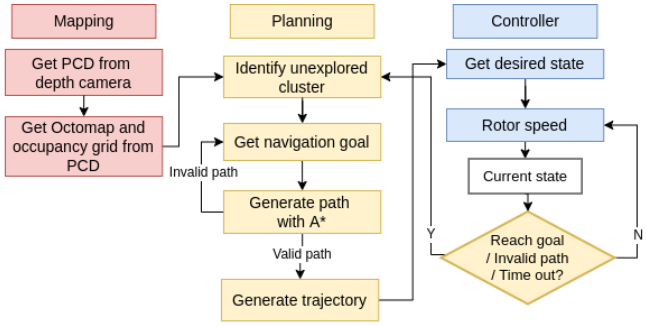

Fig. 1. System architecture

II. Simulation

Unity3D is a physics and rendering engine which offers a wide range of assets. Some of these assets were used to build the 200 by 200 outdoor environment displayed in fig. 2. The environment mainly consists of prefabs from a free asset package called Sun Temple [1]. The whole city is enclosed by high walls, which prevent the drone from getting lost outside the intended space. The drone was additionally equipped with a third person camera and an overhead camera to easily find mistakes in the mapping process.

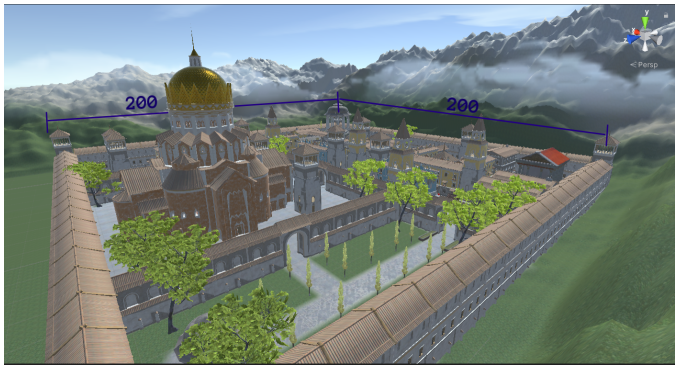

Fig. 2. Unity environment

A. Quadrotor Sensors

The drone is equipped with 4 sensors which can be seen in fig. 3. The two RGB cameras are displaced from the center by 50 mm. The depth camera has a maximum range of 20 m and a field of view of 60 degrees. The inertial measurement unit measures the current position and rotation, as well as linear and angular velocity. The data from the sensors is passed on to ROS.

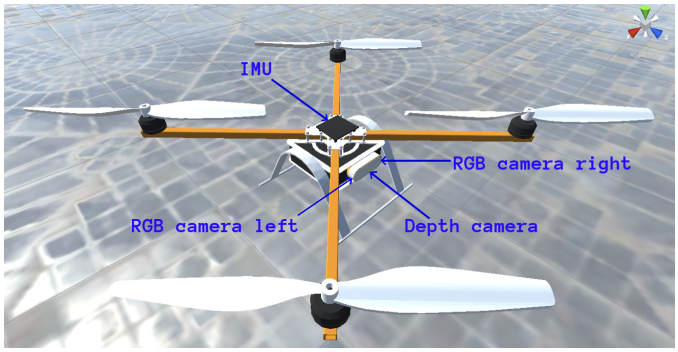

Fig. 3. The sensors of the drone

B. Communication between Unity and ROS

The messages are passed via a local TCP server, where different messages are published on different ports. The messages include all the data from the sensors as well as the rotor speed commands from the controller. The different messages are then adapted and assigned to the corresponding ROS topic by the respective parser.

III. Perception

A. Point Cloud from Depth Image

The sensor used for perception is a depth camera. Its output is depth image, where the value of each pixel represents the distance between the corresponding point in real world and the plane of the camera. This relationship is described by the pinhole model of the camera as shown in fig. 4. From the similar triangle approach we can obtain that:

$$x = \frac{(u - c_x)z}{f_x}$$
$$y = \frac{(v - c_y)z}{f_y}$$

The focal length f_x and f_y might be different because of non-rectangle image sensor or lens distortion. Writing this in matrix form and taking the transformation between world frame and camera frame into consideration, we get eq. (1), where K, the intrinsic matrix of the depth camera, contains information about the focal length and the center (c_x, c_y) of the image plane.

$$\begin{bmatrix} u \\ v \\ 1 \\ 1/z \end{bmatrix} = \frac{1}{z} \underbrace{\begin{bmatrix} \boldsymbol{K} & \boldsymbol{0} \\ \boldsymbol{0} & 1 \end{bmatrix}}_{4\times4} \underbrace{\begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{bmatrix}}_{4\times4} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{1}$$

$$\mathbf{K} = \begin{vmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{vmatrix} \tag{2}$$

From this formula, we can reconstruct the position (x, y, z) in the original 3D world frame from the coordinate (u, v) on the image plane. In our project, this part is achieved by the package depth_image_proc [3].

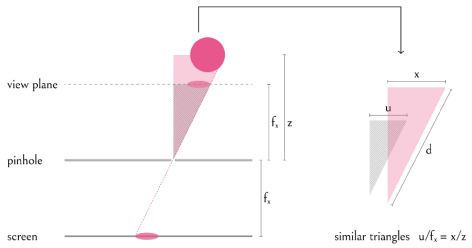

Fig. 4. Pinhole camera model [6]

B. Voxelization

The point cloud data is usually noisy and might contain outliers. It is therefore filtered with a a voxel grid filter from PCL [5]. Points are split into voxels (i.e. tiny 3D boxes) and points inside the same voxel will be approximated by the centroid. This process cleans the data and reduces its size at the same time, making the computations in the mapping part less demanding.

IV. Mapping

Point cloud has the advantages of no discretization and unlimited mapping area. Nevertheless, it uses a lot of memory and has no direct state indication of free, occupied or unknown. The Octomap framework [2] is used to efficiently generate probabilistic maps in 2 and 3 dimensions.

A. Octomap framework

Octomap is a tree-based (octree) map representation that models a three-dimensional space as a cube. The octree continuously divides the volume (a voxel) into eight cubes of equal size until the highest accuracy is achieved, as shown in fig. 5. Each voxel is in one of the states: free, occupied and unknown. If all the children of a node are in the same state, the information will not be stored, which saves a lot of memory.

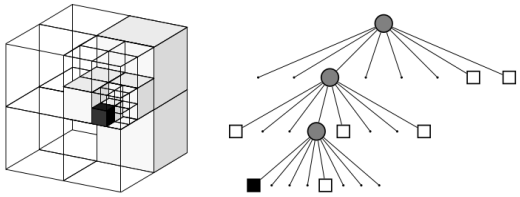

Fig. 5. A volumetric model and its tree representation [2]

B. Probabilistic map updating

Octomap integrates sensor model with occupancy model as the recursive binary bayes filter introduced in [7]. The probability of a node is:

$$P\left(n \mid z_{1:t}\right) = \left[1 + \frac{1 - P\left(n \mid z_t\right)}{P\left(n \mid z_t\right)} \frac{1 - P\left(n \mid z_{1:t-1}\right)}{P\left(n \mid z_{1:t-1}\right)} \frac{P(n)}{1 - P(n)}\right]^{-1} \tag{3}$$

where n is a node in the map and $z_{1:t}$ denotes a set of sensor measurements to time t. $p\left(n \mid z_t\right)$ is the probability given current measurement z_t, and $p\left(n\right)$ is the prior probability and equals to 0.5.

The above probabilistic form may express the occupation probability outside the interval of [0, 1]. In practice, we set the prior probability to 0.5, and then use log-odds to represent the occupation probability which is denoted by:

$$l(x) = \ln\left(\frac{x}{1 - x}\right) \tag{4}$$

Then the probability of a node given sensor measurements $z_{1:t}$ can be represented as:

$$L\left(n \mid z_{1:t}\right) = L\left(n \mid z_{1:t-1}\right) + L\left(n \mid z_t\right) \tag{5}$$

Since log-odds are convertible to probabilities, each node no longer stores probabilities, but log-odds values. Notice that log-odds values are infinitely stackable, so when a node is observed to be occupied k times, it would take at least k non-occupied observations to change the occupation status of

that node as well. This property is fine when the environment objects are static, but when the environment is highly dynamic, it causes the map to update too slowly. To ensure dynamic adaptation to the environment, it is common to give the log-odds function an artificial upper and lower limit on the value (Clamp) [9], as shown in the following equation.

$$L\left(n \mid z_{1:t}\right) = \max\left(\min\left(L\left(n \mid z_{1:t-1}\right) + L\left(n \mid z_t\right), l_{\max}\right), l_{\min}\right), \tag{6}$$

where $l_{\min}$ and $l_{\max}$ denote the lower and upper bound on the log-odds value. In this way, Octomap can be updated effectively dynamically.

C. Mapping in 2 and 3 dimensions

Two octomap server [10] nodes are run in parallel. The first one only considers point clouds around the same height as the drone. The 2-dimensional occupancy grid generated by this node is used for collision checking in the navigation part. It therefore has a relatively low resolution, to limit the computational load. The second node considers point clouds from the ground up, and it has a higher resolution to generate a more detailed map. The projected occupancy grid generated by the 2D-node is shown in fig. 7, and the final 3D-map is shown in fig. 9.

V. Navigation

Autonomous exploration of unknown space can be separated into two tasks: Finding a suitable goal, and generating a collision-free path towards it. Algorithm 1 describes this process, and it will be further explained in the following.

Data: Location of the UAV, image sensors
Result: A complete mapping of the environment
while *true* **do**
  Get current state;
  Get occupancy grid;
  while *path is invalid* **do**
    Select new goal;
    Generate path with A*;
  end
  Generate trajectory;
  while *Not (goal reached or invalid path or timeout)* **do**
    Follow trajectory;
  end
end

Algorithm 1: Autonomous exploration of unknown space

The algorithm does not have a termination criteria. This is fine, since the drone will simply stay in place if no collision-free path is found.

A. Finding a suitable goal

The objective is to generate a map. Unknown space must therefore be identified. At first, the goal is set to be the location where there are the most unknowns. We find this by converting the occupancy grid into a binary matrix, which has the value of 1 if the cell is unknown, and 0 otherwise. The goal is then found as the center of the largest rectangle formed by the set of 1's. Figure 6 demonstrates this procedure. It may seem inefficient to perform this transformation, but a conversion is needed in any case, since the occupancy grid is published as a 1-dimensional array.

$$\begin{bmatrix} 0 & 80 & 20 & 50 \\ -1 & -1 & -1 & 70 \\ -1 & -1 & -1 & 0 \\ -1 & -1 & -1 & 10 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & \mathbf{1} & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Fig. 6. Occupancy grid to binary matrix. The probability of occupancy is unknown if the cell has the value -1. The location in bold on the rightmost matrix is the next goal, as it is the center of the rectangle with the most unknowns.

B. Collision checking

A* is used to generate a collision-free path from the current location of the drone to the goal. Obstacles are defined as the points in the occupancy grid that have a high probability of being occupied. A* will find the shortest path if it exists. Therefore care must be taken so that the path is not too close to corners, as can be observed in fig. 7. We do this by also considering the neighbourhood of an obstacle as an obstacle. The path is generated in the 2D-plane defined by the occupancy grid. It is converted to the world frame by reversing eq. (7).

$$grid_x = \frac{world_x - map.origin.position.x}{map.resolution} \tag{7}$$

1) Iteration: If A* does not find a collision-free path, a new goal must be chosen. This time it is chosen as the first position which is defined as unknown (value of -1 in the occupancy grid). This process is re-iterated until a collision free path is found.

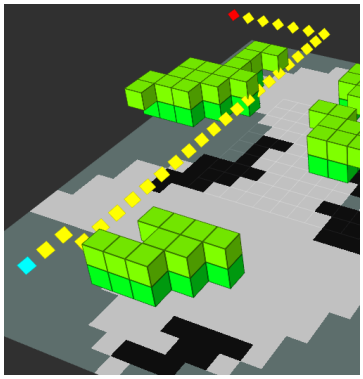

Fig. 7. An example path from start (red) to goal (blue)

C. Trajectory generation

Once a path has been found, the mav_trajectory_generation package [4] from ETH Zurich is used to generate a minimum snap trajectory. The waypoints are used as positional

constraints in the optimization problem. We chose to use all waypoints as constraints, for two reasons:

a) The trajectory will be slower due to this. This sounds like a disadvantage, but the mapping will be worse if the drone accelerates or decelerates too fast, as the camera sensors will not be able to register the environment correctly.
b) If too few constraints are set, the trajectory might not be collision free.

Once the trajectory has been generated, it is sampled to the controller at a given rate.

D. Generating new waypoints

There are three scenarios which lead to the generation of new waypoints:

a) The goal has been reached.
b) New information (updated map) shows that the path is not collision-free.
c) The drone has been stationary for too long, which indicates that it is stuck.

VI. Controller

To make the UAV move along the trajectory, the geometric control model proposed by Taeyoung Lee [8] is used.

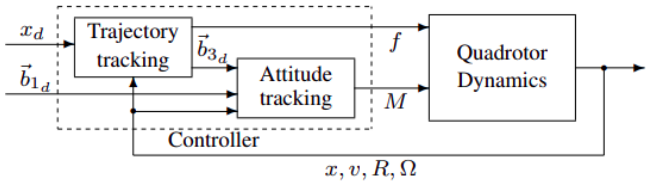

Fig. 8. Controller structure

The controller follows a prescribed trajectory of the location of the center of mass, $x_d(t)$, and the desired direction of the first body-fixed axis, $\vec{b}_{1_d}(t)$. The tracking errors are calculated based on the pose and velocity of the drone.

VII. Results

The map generated after an autonomous flight can be seen in fig. 9. It must be noted that the drone was not able to achieve such a good mapping in each run. It sometimes crashed into an obstacle at an early stage. The performance seemed to be affected by the hosting PC, as the controller was more unstable when we ran several processes in parallel.

VIII. Future work

A natural step would be to improve the navigation system so that it is able to map the entire environment. The main weakness at the moment is the conversion between the 2-dimensional occupancy grid and the world frame. As of now, certain configurations result in non-sensible paths. This can be explained by eq. (7), which may lead to negative values for the indices of the occupancy grid, which should not be possible. The navigation algorithm can also be updated to select a more sensible goal when the location of the most unknowns is not reachable. It could sample a position closer to the current location of the drone. It would also be interesting to run an online collision avoidance system in parallel.

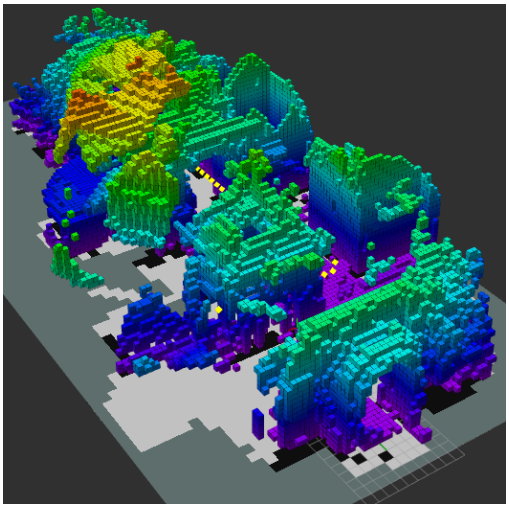

Fig. 9. The 3D map generated by the UAV after an autonomous flight

IX. Summary

In this project we have created a system for autonomous exploration of an unknown environment. We assumed the height of the drone to be static, therefore the path planning was performed in 2D. The planning algorithm seeks to find the cluster of most unknown cells in the occupancy grid, and navigate to this location using waypoints generated by A*. A large part of the environment was mapped with this approach, but further work is needed to make the drone crash less frequently.

References

[1] Sandro. T (2019) Sun Temple [Source code]. https://assetstore.unity.com/packages/3d/environments/sun-temple-115417
[2] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," Autonomous Robots, 2013.
[3] Willow Garage (2008) image_pipeline [Source code]. https://github.com/ros-perception/image_pipeline/tree/melodic.
[4] Rik Bähnemann mav_trajectory_generation [Source code]. https://github.com/ethz-asl/mav_trajectory_generation.
[5] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," 2011 IEEE International Conference on Robotics and Automation, 2011, pp. 1-4, doi: 10.1109/ICRA.2011.5980567.
[6] yodayoda. "From depth map to point cloud", Medium. https://medium.com/yodayoda/from-depth-map-to-point-cloud-7473721d3f(accessed Mar.15, 2022)
[7] H. Moravec and A. Elfes, "High resolution maps from wide angle sonar," Proceedings. 1985 IEEE International Conference on Robotics and Automation, 1985.
[8] Lee, Taeyoung, Melvin Leok, and N. Harris McClamroch. "Geometric tracking control of a quadrotor UAV on SE (3)." 49th IEEE conference on decision and control (CDC). IEEE, 2010.
[9] Yguel M, Aycard O, Laugier C (2007a) Update policy of dense maps: Efficient algorithms and sparse representation. In: Field and Service Robotics, Results of the Int. Conf., FSR 2007, vol 42, pp 23–33.
[10] A. Hornung. (2010) octomap_mapping [Source code]. Available: https://github.com/OctoMap/octomap_mapping