3-DIMENSIONAL DRONE TOPOGRAPHY (SIMULATION IN ROS)

Project Report

Eklavya Mentorship Programme

At

SOCIETY OF ROBOTICS AND AUTOMATION, VEERMATA JIJABAI TECHNOLOGICAL INSTITUTE, MUMBAI

AUGUST-SEPTEMBER 2022

Acknowledgement

We're extremely grateful to our mentors Jash Shah and Sarrah Bastawala for their support and guidance throughout the duration of the project. We would also like to thank all the members of SRA VJTI for their timely support as well as for organizing Eklavya and giving us a chance to work on this project.

Soham Mulye

mulyesoham@gmail.com

Unmani Shinde

mailto.unmani@gmail.com

Table of Contents

Project Overview

- The traditional problem addressed by surface reconstruction is to recover the digital representation of a physical shape that has been scanned, where the scanned data contains a wide variety of defects. At its core, therefore, surface reconstruction is the process by which a 3D object is inferred, or "reconstructed", from a collection of discrete points that sample the shape, which is, in our case, obtained from LiDAR Sensors.

- The idea is to have a Drone fly over some terrain in the **Robot Operating System (ROS)** with a GPS and a Depth sensor, then get the point cloud data from the drone and create a 3D map of the topography of the terrain.

- The project was started with the use of ROS to obtain the Point Cloud data of a terrain with the use of LiDAR sensors. The use of **Point Cloud Library (PCL)** followed to create 3D Polygonal Meshes and understanding its usage and experimenting with the various algorithms used in PCL for reconstructing meshes from the point clouds.

- Throughout the course of this project, we learnt about several reconstruction techniques that included such as subsampling, up-sampling, estimation of surface normals and algorithm for mesh creation from these calculated normals.

- PCL makes use of many algorithms like Marching Cubes, Grid Projection, Greedy Projection, etc. After experimenting, the project focused on creating the mesh using the **Greedy Projection Triangulation** algorithm. The data for GPT has been created from the Point Cloud using **Moving Least Squares** via **Maximum Likelihood Estimation.**

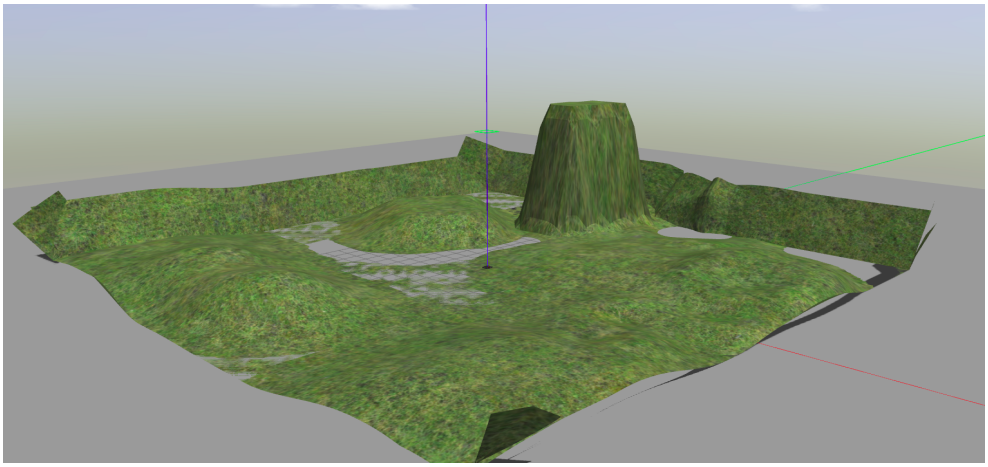
1. INTRODUCTION

1.1 DESIGNING A DRONE

Designing any drone comes with the constraints employed by various factors such as the physical conditions it is supposed to operate in, the kind of data it is supposed to obtain and process, amongst several other factors. For our project, we have decided to use the quadrotor, which as suggested by its name, utilizes four rotors.

1.2 NEED FOR SIMULATION

Post the design, testing experimental software on real world hardware consists of several risk factors. Having a good simulation environment is a valuable tool in robotics, as testing on real hardware can often be expensive and time consuming. The actions of some robots can be hazardous, and deploying code (especially in early development) carries risks. Working with real hardware can also introduce issues that impede core algorithm development. For this reason, time spent building an accurate simulation environment is usually well worth it.



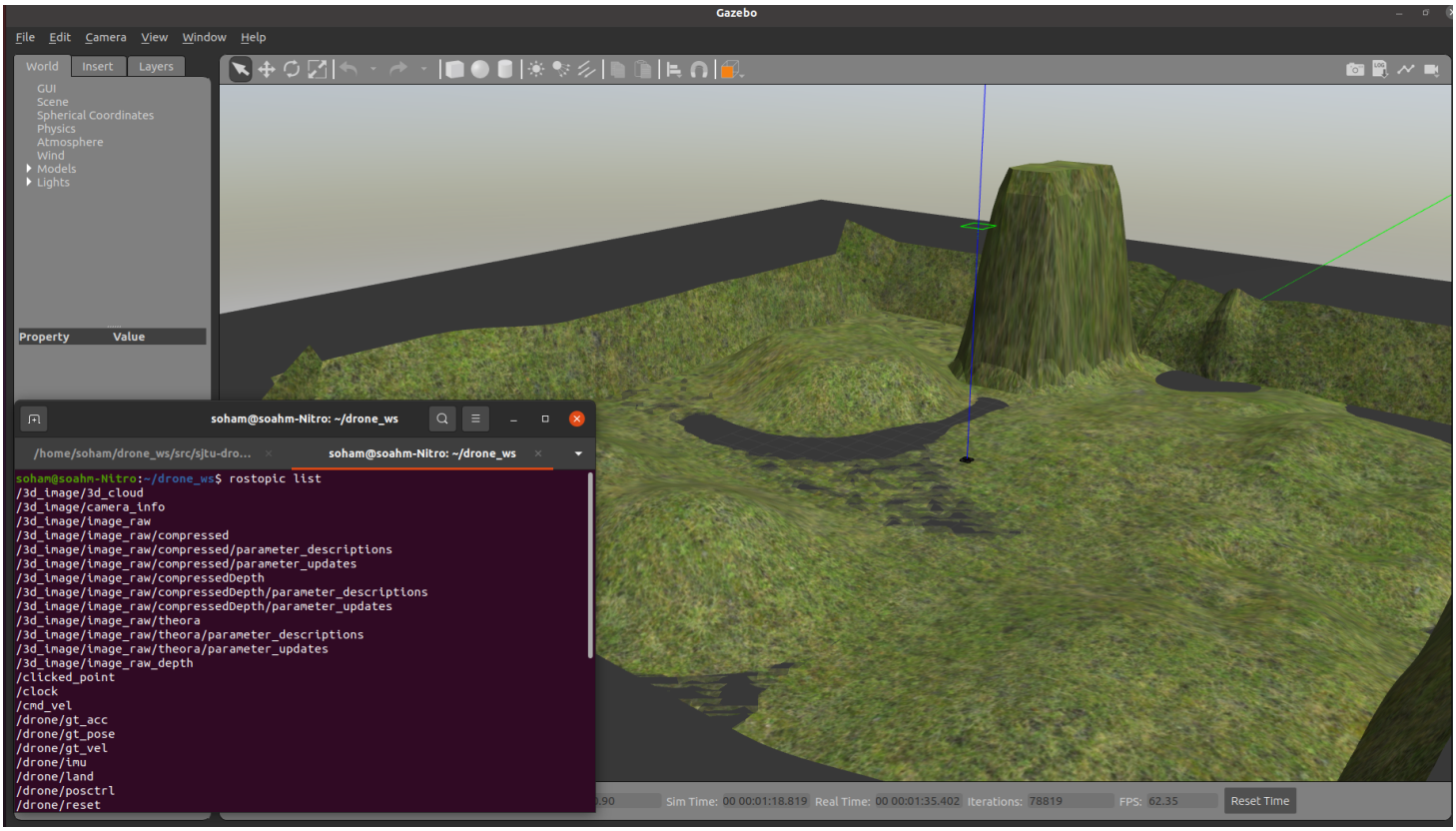
1.3 USE OF THE POINT CLOUD LIBRARY

The Point Cloud Library (PCL) is a standalone, large scale, open project for 2D/3D image and point cloud processing. To simplify both usage and development, PCL is split into a series of modular libraries. The set of released modules in PCL that have been heavily utilized in our project have been tabulated as follows:

Sr. No. :	Library/Module:	Description:	Mechanism(s) Used:
1.	pcl_io	contains classes and functions for reading and writing point cloud data (PCD) files, as well as capturing point clouds from a variety of sensing devices.	---
2.	pcl_filters	contains outlier and noise removal mechanisms for 3D point cloud data filtering applications.	Voxel Grid Filter
3.	pcl_features	contains data structures and mechanisms for 3D feature estimation from point cloud data, like estimated curvature and normal at a query point p	Principal Component Analysis, Moving Least Squares
4.	pcl_kdtree	provides the kd-tree data-structure, using FLANN, that allows for fast nearest neighbor searches.	Kd-Tree
5.	pcl_surface	deals with reconstructing the original surfaces from 3D scans, usually a mesh representation or a smoothed/resampled surface with normals.	Greedy Projection, Marching Cubes with RBF

1.4 USE OF ROS & GAZEBO

- Although ROS stands for Robot Operating System, it is not actually an OS, but an open-source robotics middleware suite, i.e., a set of common libraries and tools on which to build more complex robotic systems. ROS was designed to be as distributive and modular as possible. ROS packages are easy to make and distribute, which makes it an ideal choice for testing robots.
- Gazebo is a free, open-source robot simulation environment. The project is run by Open Robotics, the same group looking after ROS, however the projects are managed separately and Gazebo is not a "part of" ROS. With Gazebo, we can create a virtual "world", and load simulated versions of our robots into it. Simulated sensors can detect the environment, and publish the data to the same ROS topics that real sensors would, allowing easy testing of our surface reconstruction algorithms



2. IN-DEPTH ANALYSIS

2.1 OBTAINING POINT CLOUD DATA FROM ROS

2.1.1 ROS Terminologies

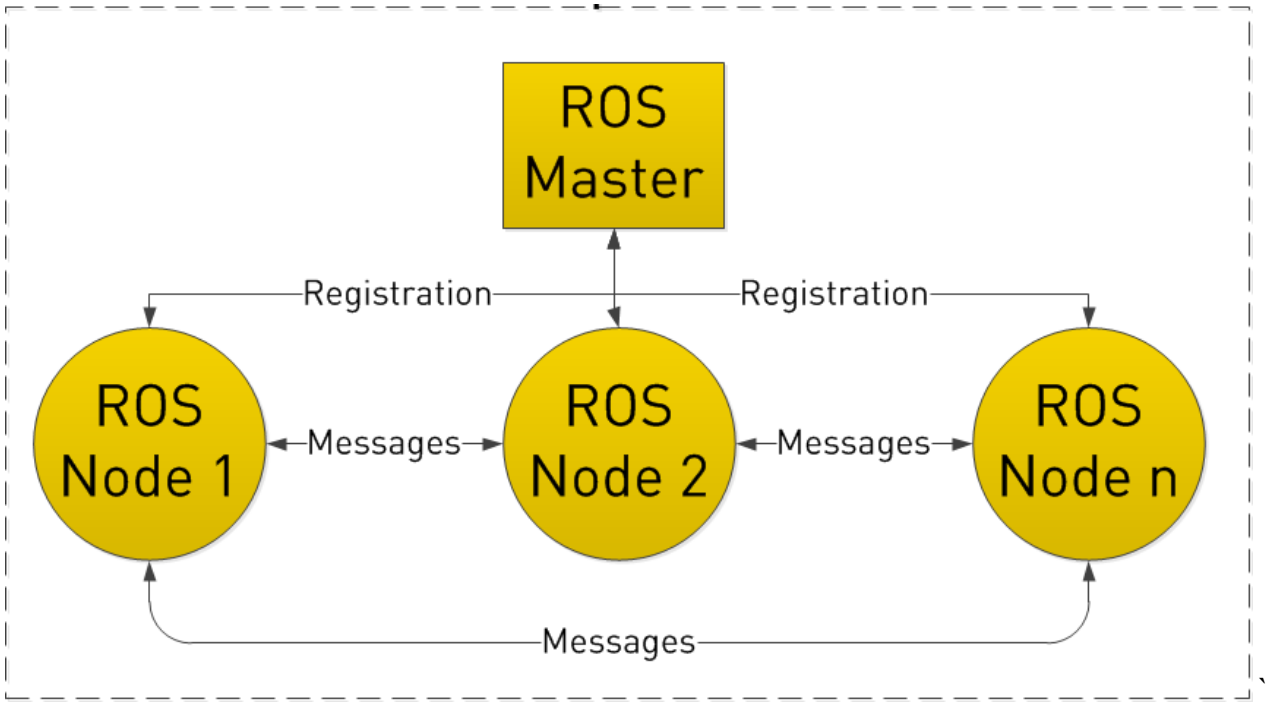
- **Catkin Workspace**: A catkin workspace is a folder where you modify, build, and install catkin packages. A trivial workspace may look like this:

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for package_n
      package.xml        -- Package manifest for package_n
```

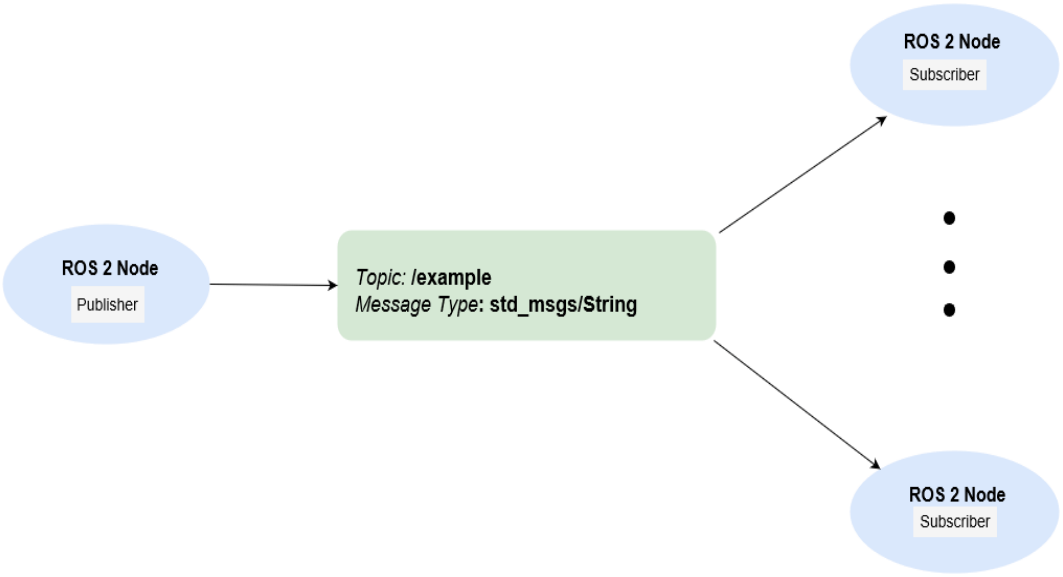
- **ROS Package**: Software in ROS is organized in packages. A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module.
- **ROS Node, Master, Topics, Services and Messages:**

  A *node* really isn't much more than an executable file within a ROS package. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a Topic, or provide or use a Service.

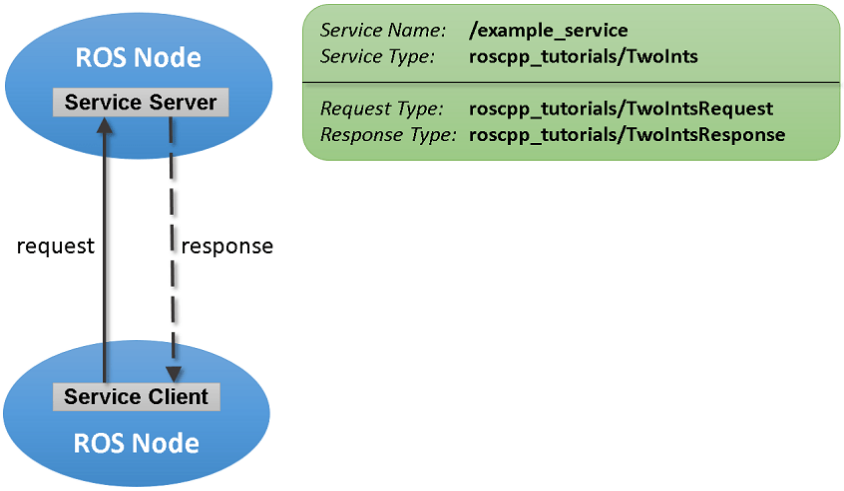
  The *ROS Master* provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer.



Topics are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. Topics are intended for unidirectional, streaming communication.

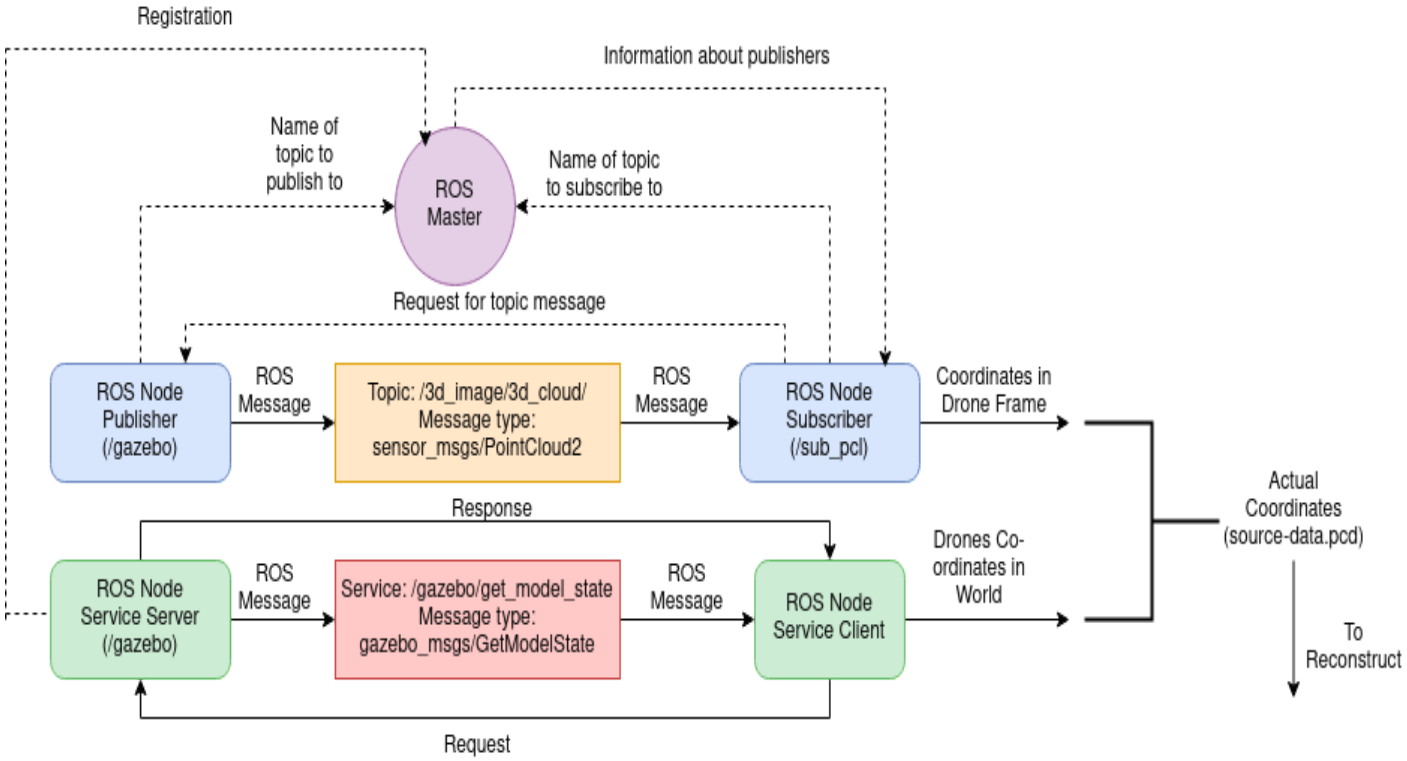


Unlike the many-to-many one-way transport model of Topic communication, request / reply is implemented via a *Service,* which is defined by a pair of messages: one for the request and one for the reply.



2.1.1 General Workflow

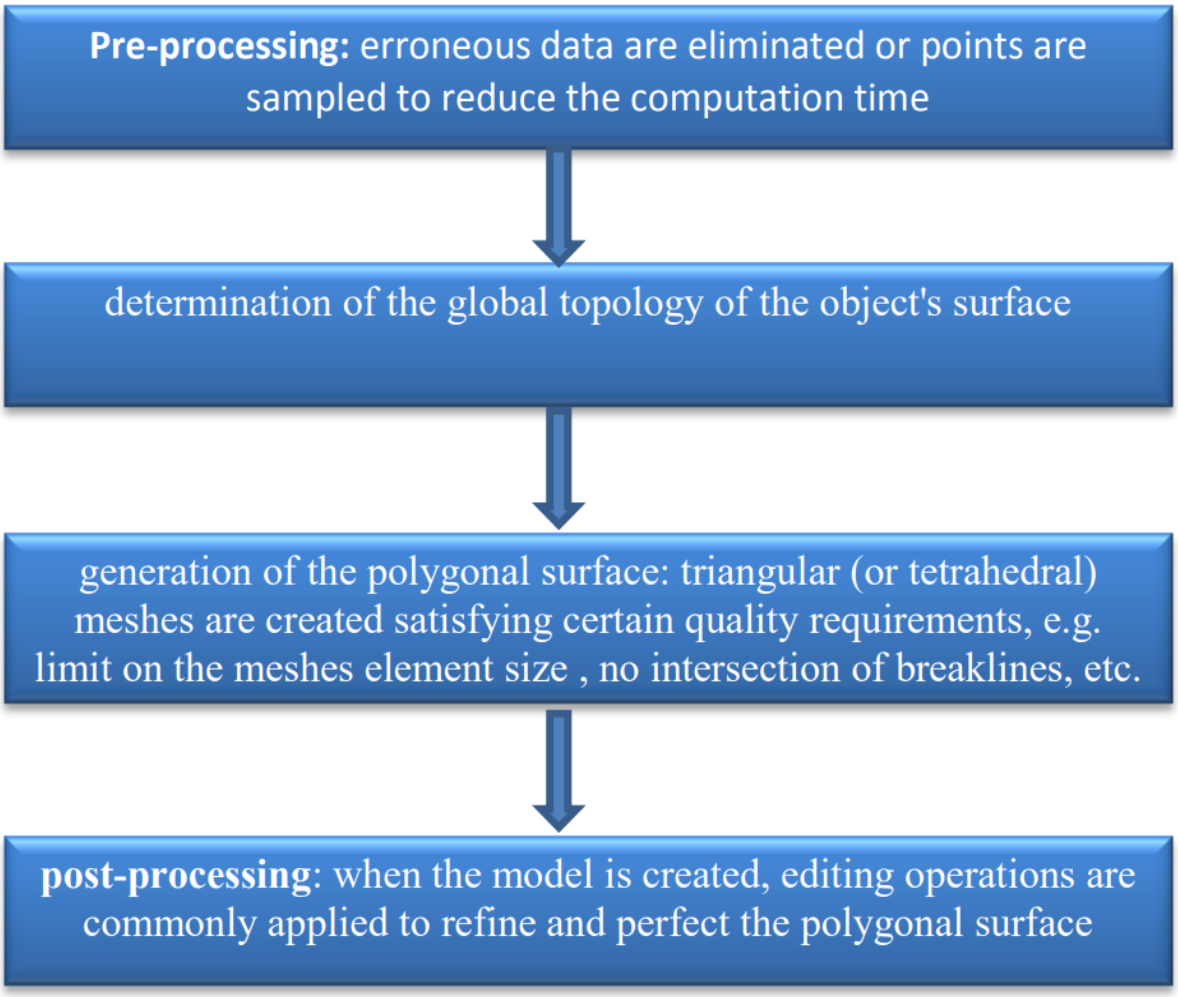
An overview of obtaining Point Cloud Data is as given below:



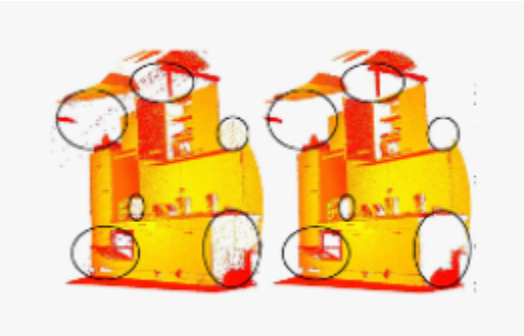
2.2 SURFACE RECONSTRUCTION

From the depth sensor we obtained an unstructured point cloud representing our terrain, and the final goal is to be able to generate a dense 3D geometry based on this point cloud, i.e., a polygonal (specifically, triangular) mesh which is the commonly accepted graphic representation with the widest support from existing software and hardware.

The general steps in any surface reconstruction algorithm are as follows:

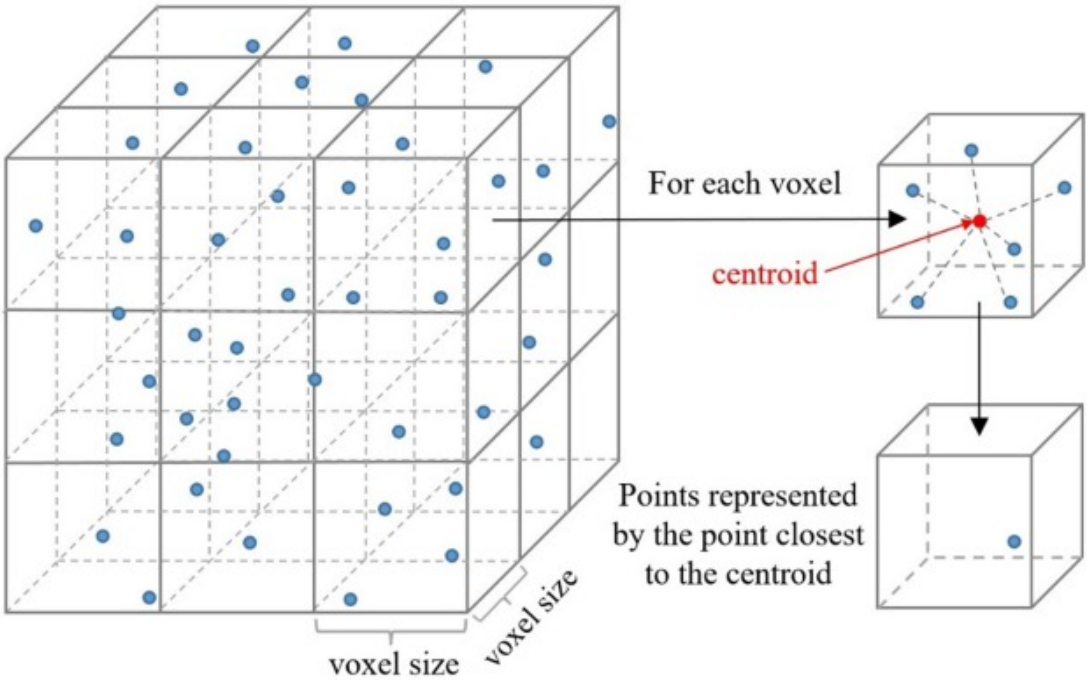


2.2.1 Pre-Processing



The point cloud data obtained might be erroneous due to inaccurate motion of the quadrotor, therefore some points might have to be eliminated to enhance the accuracy of feature estimation. In addition to inaccurate points, the dataset may also contain some redundant points, which upon being eliminated, significantly reduce the computational time at some minimal expense of accuracy. Such points are eliminated using the **VoxelGrid** Filter Mechanism in PCL.

The `VoxelGrid` class creates a 3D voxel grid (think about a voxel grid as a set of tiny 3D boxes in space) over the input point cloud data. Then, in each voxel, all the points present will be *approximated (i.e., downsampled) with their centroid*. This approach is a bit slower than approximating them with the center of the voxel, but it represents the underlying surface more accurately.

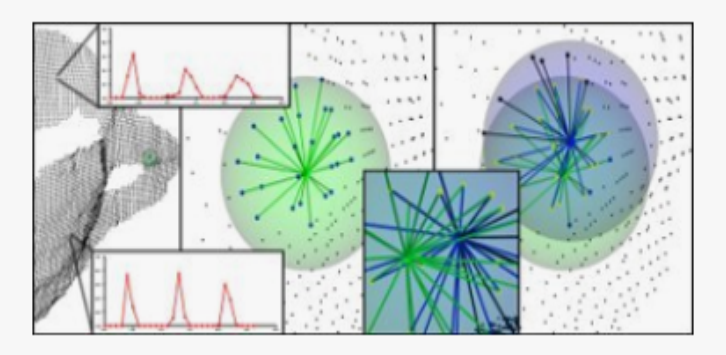


Pseudo Code:

1. We first create two pointers, two store the point cloud data pre and post filtering, and the pre-filtering data is stored in the point cloud accordingly.
2. Next, a VoxelGrid Filter object is created, to downsample the data, and the input point cloud is assigned to it.
3. Since VoxelGrid Filter operates on a grid structure of the input point cloud, we set a grid square size (or in terms of the filter, “LeafSize”), that is suitable considering the size of our input cloud.
4. When the sor.filter() function is executed, the points within one grid square are represented by a point closest to their centroid, and the resulting point cloud is stored in the other initialized pointer.

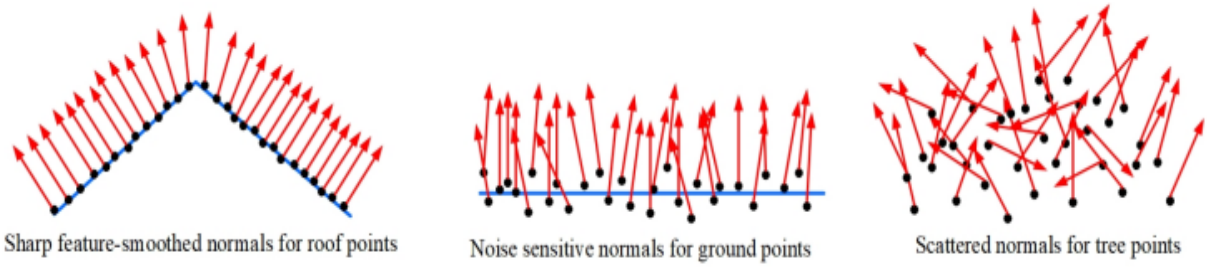
2.2.2 Feature (Normal) Estimation

In computer graphics, the surface normals are used for generating the correct shadows based on the lightning. In surface reconstruction, the normals define the direction of the polygon faces. Normal Estimation has been implemented using two algorithms in this project, which are as described in the following pages.

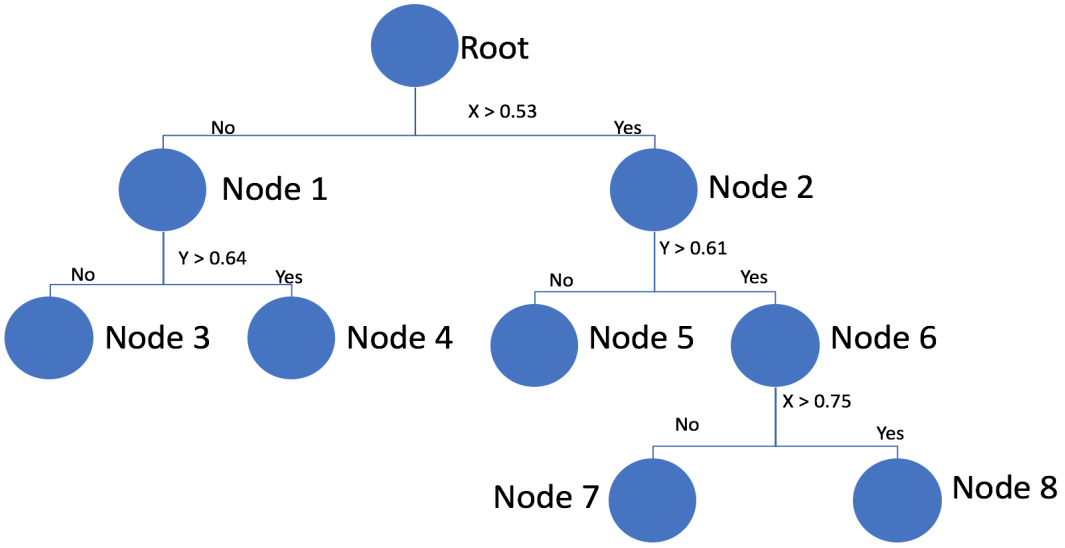


2.2.2.1 Kd-Trees & Nearest Neighbor Search Problem

- Kd tree or a k-dimensional tree is a space-partitioning data structure for organizing points in a k-dimensional space. The k-d tree is a generalization of binary search trees in which every node is a k-dimensional point.
- Every non-leaf node generates a splitting hyperplane that divides the space into two two parts, known as half-spaces. Each node in the tree is defined by a plane through one of the dimensions that partitions the set of points into left/right (or up/down) sets, each with half the points of the parent node. These children are again partitioned into equal halves, using planes through a different dimension. Partitioning stops after log n levels, with each point in its own leaf cell.
- The partitioning loops through the different dimensions for the different levels of the tree, using the median point for the partition. kd-trees are known to work well in low dimensions but seem to fail as the number of dimensions increases beyond three.



- A 2-dimensional kd-Tree can be illustrated as below:



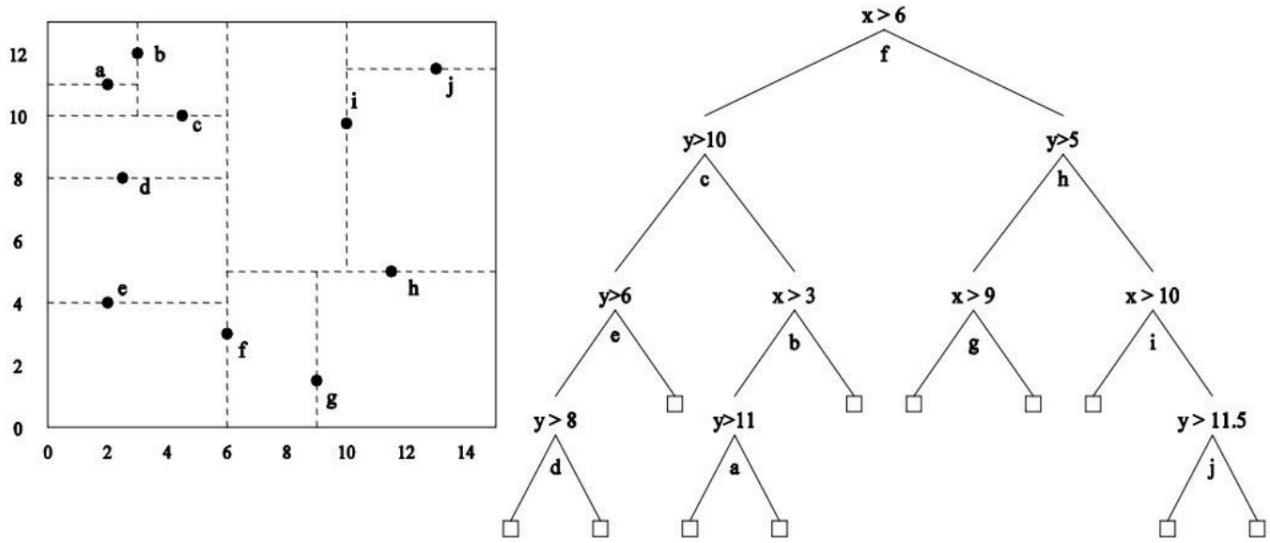
- kd-Trees ensure efficient searches, and can be utilized in the nearest neighbor search algorithm as follows:

1. locate the location where the point would be located if it were added to the tree: Starting with the root node, the algorithm moves down the tree recursively, taking decisions to follow the left or the right node depending on whether the point is less than or greater than the current node in the split dimension.
2. Once the algorithm reaches a leaf node, it saves that node point as the "current best". As the tree is traversed the distance between the point and the current node should be recorded.
3. The algorithm goes back up the tree evaluating each branch of the tree that could have points within the current minimum distance i.e. it unwinds the recursion of the tree, performing the following steps at each node:

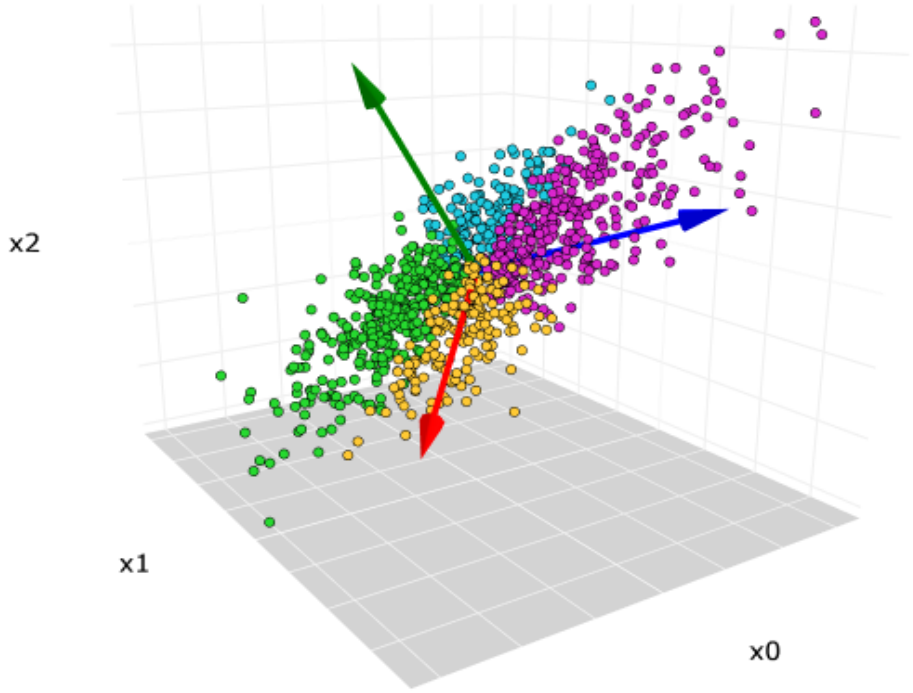
   a) If the current node is closer than the current best, then it becomes the “current best.”

b) Check the other side of the hyperplane for points that could be closer to the search point than the current best. To check this, intersect the splitting hyperplane with a hypersphere around the search point that has a radius equal to the current nearest distance. i) If the hypersphere crosses the plane, there could be nearer points on the other side of the plane. The process is repeated in this branch. ii) If the hypersphere doesn't intersect the splitting plane, then the algorithm continues walking up the tree, and the entire branch on the other side of that node is eliminated.

4. Search completes when the algorithm reaches the root node.



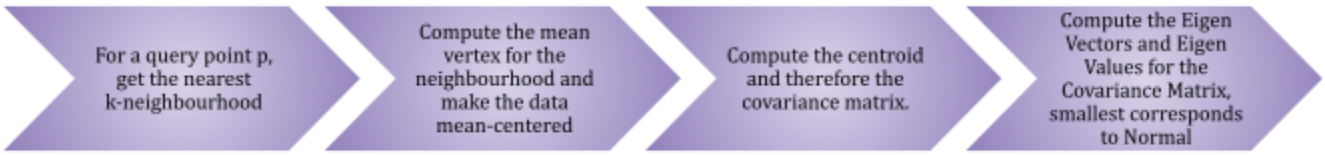
2.2.2.2 Principal Component Analysis

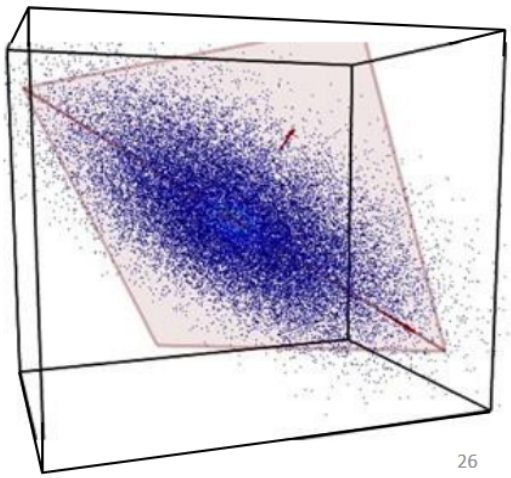


- Principal component analysis (PCA) involves a mathematical procedure that transforms a number of correlated variables into a (smaller) number of uncorrelated variables called principal components.
- The first principal component accounts for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible.
- Normals characterize a point using the information provided by its k closest point neighbors. For determining these neighbors efficiently, the input dataset is usually split into smaller chunks using spatial decomposition techniques (kD-trees in this project, see section 2.2.2.0), and then closest point searches are performed in that space.
- Depending on the application one can opt for either determining a fixed number of k points in the vicinity of p, or all points which are found inside of a sphere of radius r centered at p.
- The solution for estimating the surface normal is therefore reduced to an analysis of the eigenvectors and eigenvalues of a covariance matrix created from the nearest neighbors of the query point. Thus, the *eigenvector corresponding to the smallest eigenvalue* will approximate the surface normal n at point p.
- For each query point pi, we assemble the covariance matrix $\mathcal{C}$ as follows:

$$\mathcal{C} = \frac{1}{k}\sum_{i=1}^{k} \cdot (\boldsymbol{p}_i - \overline{\boldsymbol{p}}) \cdot (\boldsymbol{p}_i - \overline{\boldsymbol{p}})^T, \ \mathcal{C} \cdot \vec{\mathbf{v}_j} = \lambda_j \cdot \vec{\mathbf{v}_j}, \ j \in \{0, 1, 2\}$$

Where k is the number of point neighbors considered *in the neighborhood of* $\boldsymbol{p_i}$, $\overline{\boldsymbol{p}}$ represents the 3D centroid of the nearest neighbors, λ_j is the j-th eigenvalue of the covariance matrix, and $\vec{\mathbf{v}_j}$ the j-th eigenvector.





In 3D, after the first base vector is found, the data is projected onto a plane with this base vector as its normal, and we find the second base vector in this plane as the direction with largest variance in that plane.

(This "removes" the variance explained by the first base vector.)

After the first two base vectors are found, the data is projected onto a line orthogonal to the first two base vectors and the third.

Pseudo Code:

1. Continuing from the previous step, we first create a pointer of suitable type to store the normals to be computed from the downsampled point cloud.
2. Next, a Normal Estimation object is created, to estimate normals at all points to the provided cloud, and the downsampled point cloud is assigned to it.
3. Next, to look for the nearest neighbors, we set the method as kD-Trees, as explained in section 2.2.2.0. We also assign the radius in which the nearest neighbors are to be found, following which a call to the compute() function is made which computes the normals and stores them in the specified pointer.

2.2.2.3 Moving Least Squares With Maximum Likelihood Estimation

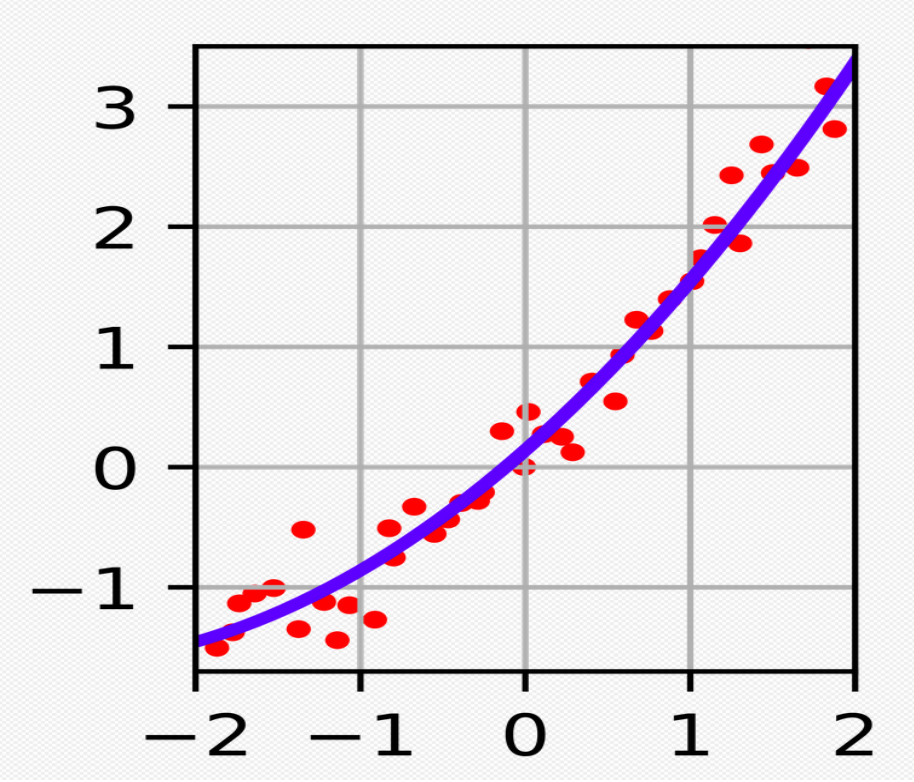
- The least squares method is a statistical procedure to find the best fit for a set of data points by minimizing the sum of the offsets or residuals of points from the plotted curve.
- The least-squares explain that the curve that best fits is represented by the property that the sum of squares of all the deviations from given values must be minimum, i.e:

$$S = \sum_{i=1}^{n} {d_i}^2$$

$$S = \sum_{i=1}^{n} [y_i - f_{x_i}]^2$$

$$S = {d_1}^2 + {d_2}^2 + {d_3}^2 + \cdots + {d_n}^2$$

- This method is called the "Moving" Least Squares Method is when the data is fed continuously, and simultaneously we get output of all the data present by applying the least squares method.



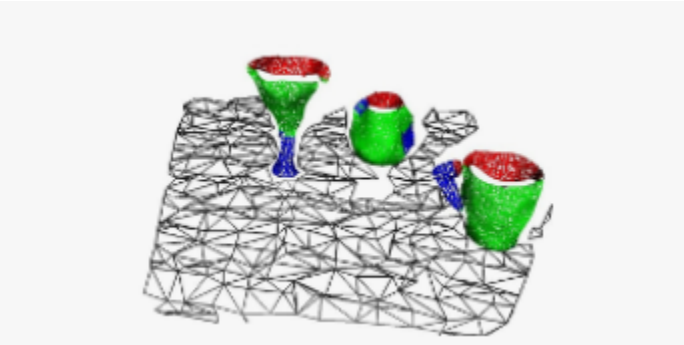
- Moving Least Squares (MLS) surface reconstruction method can be used to smooth and resample noisy data. Some of the data irregularities (caused by small distance measurement errors) are very hard to remove using statistical analysis.
- To create complete models, glossy surfaces as well as occlusions in the data must be accounted for. In situations where additional scans are impossible to acquire, a solution is to use a resampling algorithm, which attempts to recreate the missing parts of the surface by higher order polynomial interpolations between the surrounding data points. By performing resampling, these small errors can be corrected and the "double walls" artifacts resulting from registering multiple scans together can be smoothed.

- On the left side we see the noisy data which has been smoothed using the moving least square algorithms. MLS predominantly is used for smoothening of the normals computed in a fashion similar to that used in Principal Component Analysis (see section 2.2.2.1), by the application of least squares method.

Pseudo Code:

```
{ In the main function
        pass the point cloud data as pointer to the mls function;
        set compute normals as true, it will calculate the normals of the point cloud data
        set the search method as kd trees
        set the search radius according to the data set
        process and save the output
        end
}
```

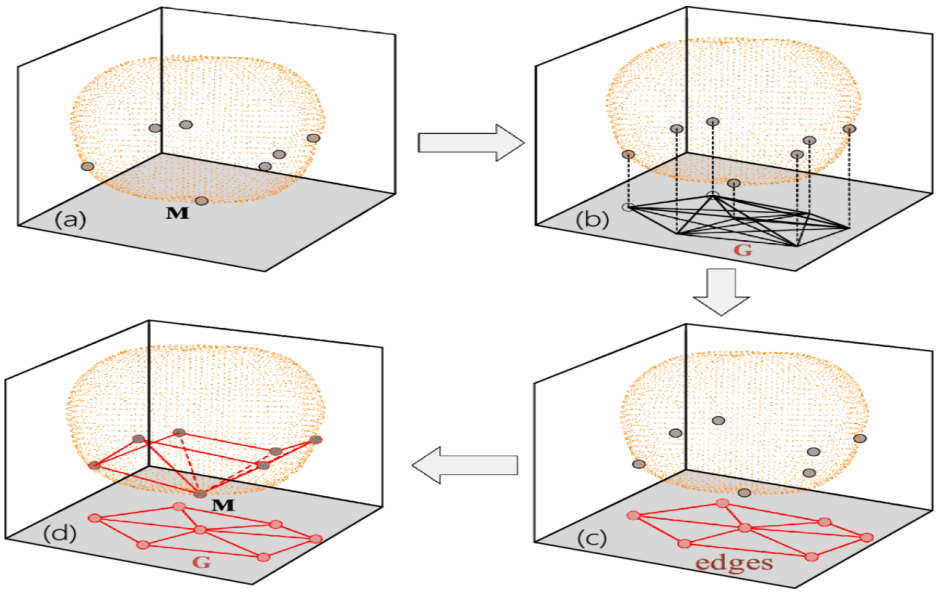
2.2.3 Meshing



- A mesh is a collection of triangular (or quadrilateral) contiguous, non-overlapping faces joined together along their edges. A mesh therefore contains vertices, edges and faces and its easiest representation is a single face. Sometimes it is also called a "Triangulated Irregular Network" (TIN).
- Meshing is a general way to create a surface out of points. The current algorithms available within PCL create a mesh representation or a smoothed/resampled surface with normals, two of which are as explained below:

2.2.3.1 Greedy Projection Triangulation

- The Greedy Projection algorithm works by maintaining a list of points from which the mesh can be grown ("fringe" points) and extending it until all possible points are connected.
- Triangulation is performed locally, by projecting the local neighborhood of a point along the point's normal and connecting unconnected points.



- For each point ‘p’ in the point cloud, a k-neighborhood is selected. This neighborhood is created by searching the reference point’s nearest k-neighbors within a sphere of radius r. (see section 2.2.2.0)
- The radius is defined as μ.d, where d is the distance of the point p from its closest neighbor and μ is the user-specified constant to take into account the point cloud density.
- The points in the cloud are assigned various states depending on their interaction with the algorithm: *free*, *fringe*, *boundary*, and *completed*.

    a. Initially, all the points in the cloud are in the free state and free points are defined as those points which have no incident triangles.

    b. When all the incident triangles of a point have been determined, the point is referred to as “completed”.

    c. When a point has been chosen as a reference point but has some missing triangles due to the maximum allowable angle parameter, it is referred to as a “boundary” point.

    d. “Fringe” points are the points that have not yet been chosen as a reference point.
- The neighborhood is projected on a plane that is approximately tangential to the surface formed by the neighborhood and ordered around p.
- The points are pruned by visibility and distance criterion, and connected to p and to consecutive points by edges, forming triangles that have a maximum angle criterion and an optional minimum angle criterion. The points in the point cloud are pruned depending on many criterions.

    a. Pruning by distance criterion: a distance criterion is applied to prune down the search for candidate adjacent points in the spatial proximity of the current reference point using kd-tree. Further points which lie outside the sphere of influence centered at reference point are rejected. The chosen points are referred to as the “candidate” points. The candidate set of points obtained after applying the distance criterion are projected on the approximate tangent plane.
- The input to this algorithm is a point cloud with estimated surface normals and the curvature data that can be calculated using either Principal Component Analysis (see section 2.2.2.1) or Moving Least Squares method (see section 2.2.2.2).

Pseudo Code:

4. Continuing from the previous step, we first create a pointer of suitable type to store the normals to be computed from the downsampled point cloud.
5. Next, a Normal Estimation object is created, to estimate normals at all points to the provided cloud, and the downsampled point cloud is assigned to it.
6. Next, to look for the nearest neighbors, we set the method as kD-Trees, as explained in section 2.2.2.0. We also assign the radius in which the nearest neighbors are to be found, following which a call to the compute() function is made which computes the normals and stores them in the specified pointer.

Following are the defining parameters of Greedy Algorithm:

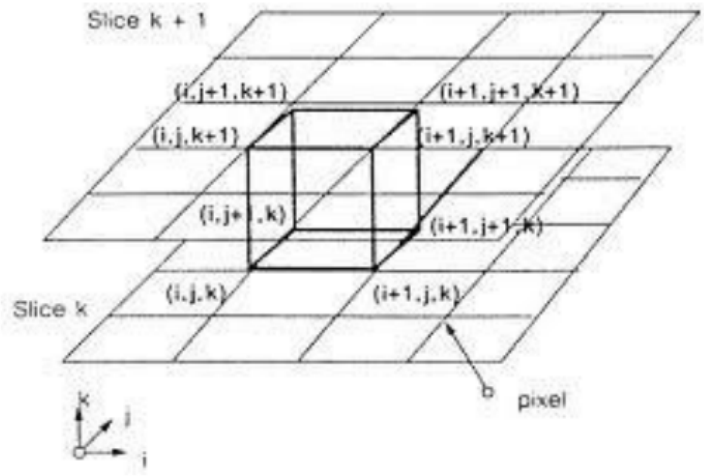
- **nnn_(unsigned int)** and **mu_(double)**: These parameters control the size of the neighborhood of projection. The former defines the maximum number of neighbors that are searched for, while the latter specifies the maximum acceptable distance for a point to be considered, relative to the distance

of the nearest point (in order to adjust to changing densities). These parameters are referenced by the functions setMaximumNearestNeighbors() and setMu() respectively.

- **search_radius_ (double)** is practically the maximum edge length for every triangle. This has to be set by the user such that to allow for the biggest triangles that should be possible. It is defined by the product of mu_ and the distance of the nearest neighbour in the k-neighbourhood in order to accomodate varying point densities. Referenced by the function setSearchRadius().
- **minimum_angle_ (double)** and **maximum_angle_ (double)** are the minimum and maximum angles in each triangle.
- **maximum_angle_** and **consistent_(bool)** are meant to deal with the cases where there are sharp edges or corners and where two sides of a surface run very close to each other. To achieve this, points are not connected to the current point if their normals deviate more than the specified angle (note that most surface normal estimation methods produce smooth transitions between normal angles even at sharp edges). This angle is computed as the angle between the lines defined by the normals (disregarding the normal's direction) if the normal-consistency-flag is not set, as not all normal estimation methods can guarantee consistently oriented normals. Typically, 45 degrees (in radians) and false works on most datasets.

2.2.3.2 Marching Cubes Algorithm Using Radial Basis Function

- The Marching Cube (MC) algorithm is the most used for the isosurface reconstruction. The algorithm subdivides the region of space into 3D array cubes, also known as voxels, formed by 8 vertices and 12 edges.



- The algorithm instructs to 'march' through each of the cubes while testing the corner points and replacing the cube with an appropriate set of polygons. The sum total of all polygons generated will be a surface that approximates the one the data set describes.
- The process flow can be explained in the following steps:

STEP 1: define a cube and number its vertices according to the Paul Bourke convention.

STEP 2: determination of the index and use the index as a pointer in the Lookup table, which defines the set of intersections of the interested surface by the cube edges

STEP 3: Calculate the intersection points on the cube edges by linear interpolation.

3. IMPLEMENTATION

3.1 PACKAGE OVERVIEW

Let's take a look at the Drone-3d-topography Package:

```
📦Drone-3D-topography
 ├ 📂assets
 ├ 📂config
 ├ 📂include
 │  └ 📜DialogKeyboard.h
 │  └ 📜drone_object_ros.h
 │  └ 📜pid_controller.h
 │  └ 📜plugin_drone.h
 │  └ 📜plugin_ros_cam.h
 │  └ 📜sensor_model.h
 │  └ 📜util_ros_cam.h
 ├ 📂launch
 │  └ 📜simple.launch
 ├ 📂models
 │  └ 📂kinect
 │  │  ├ 📂materials
 │  │  ├ 📂meshes
 │  │  ├ 📜model.config
 │  │  └ 📜model.sdf
 ├ 📂plugins
 │  └ 📜libplugin_drone.so
 │  └ 📜libplugin_ros_cam.so
 ├ 📂scripts
 │  ├ listener.cpp
 ├ 📂src
 │  ├ 📜DialogKeyboard.cpp
 │  └ 📜DialogKeyboard.ui
 │  └ 📜drone_keyboard.cpp
 │  └ 📜drone_object_ros.cpppid_controller.cpp
 │  └ 📜plugin_drone.cpp
 │  └ 📜plugin_ros_cam.cpp
 │  └ 📜plugin_ros_init.cpp
 │  └ 📜util_ros_cam.cpp
 ├ 📂urdf
 │  └ 📜sjtu_drone.urdf
 ├ 📂worlds
 │  ├ 📜terrain.world
 ├ 📜CMakeLists.txt
 ├ 📜README.md
 └ 📜package.xml
```

1. **config**: Contains all RViz configuration files.
2. **include**: Contains all the necessary header files for keyboard control and other plugins installed on the quadrotor.
3. **launch**: Contains the simple.launch file which is used to load the model into Gazebo.
4. **models**: Contains the meshes and textures for the quadrotor model.
5. **scripts**: Contains the main C++ script (listener.cpp) used to interact with the simulation of the quadrotor.
6. **src**: Contains the custom plugins used with the model.
7. **worlds**: Contains the world file which loads in a simple open sky world with our sample terrain and the ground. This file gets launched automatically when the simple.launch file gets launched.
8. **CMakeLists.txt**: Contains the instructions to link all these files to their dependencies when the package is built.

3.2 QUADROTOR MODEL

The quadrotor drone model we incorporated within the project ("sjtu_drone") was developed by Shanghai Jiao Tong University. It is a simulation developed with ROS + Gazebo, and is used for testing visual SLAM algorithms aiding with different sensors, such as IMU, sonar range finder and laser range finder.

Because Gazebo and ROS are separate projects that do not depend on each other, sensors from the gazebo_models repository (such as depth cameras) do not include ROS plugins by default. So, we were required to make a custom camera based on those in the Gazebo model repository, and then add our own <plugin> tag to make the depth camera data publish point clouds and images to ROS topics. The Microsoft Kinect depth camera was used with our model.

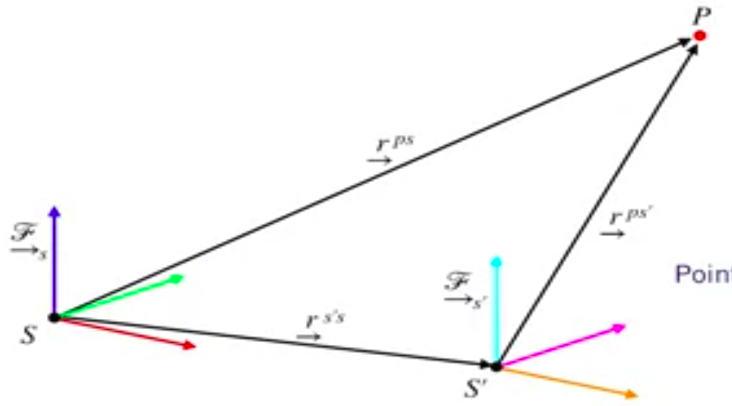
3.3 CODE

The code utilized to get the Point Cloud Data of the terrain from ROS and to process it is effectively contained within the C++ script (listener.cpp) and is implemented as follows:

In a frame when the drone hovers over a particular region of the terrain, the Point Cloud Data of that region is published via the depth camera to the topic "/3d_image/3d_cloud/" for which a subscriber node has been created.

However, the coordinates of every point in that frame were obtained considering the drone itself as the origin, and hence to get the coordinates of each point with respect to the origin of the world map, we used the get_model_state() service in order to obtain the coordinates of the drone. We then applied the logic of translation and rotation to get the actual coordinates which were then appended ("pushed back") in order to create the source PCD file.

Translation in Point Clouds:



From vector addition

$$\vec{r}^{\,ps'} = \vec{r}^{\,ps} - \vec{r}^{\,s's}$$

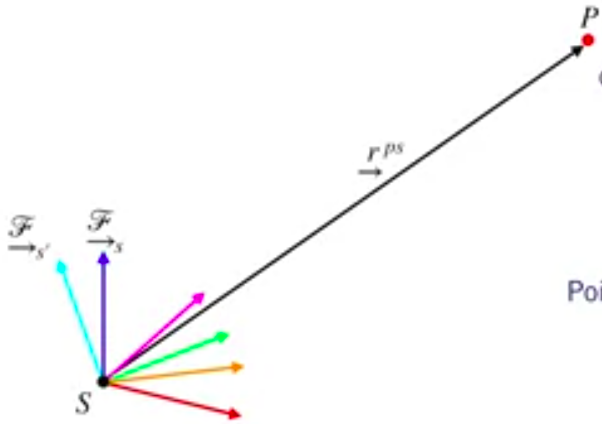
Point coordinates in the translated frame

$$\mathbf{p}_{s'}^{(j)} = \mathbf{p}_{s}^{(j)} - \mathbf{r}_{s}^{s's} \quad \text{each point}$$

$$\mathbf{P}_{s'} = \mathbf{P}_{s} - \mathbf{R}_{s}^{s's} \quad \text{whole cloud}$$

$$\mathbf{R}_{s}^{s's} = \left[\mathbf{r}_{s}^{s's} \quad \mathbf{r}_{s}^{s's} \quad \cdots \quad \mathbf{r}_{s}^{s's}\right]$$

Rotation in Point Clouds:



The *rotation matrix* takes coordinates from frame *s* to frame *s'*

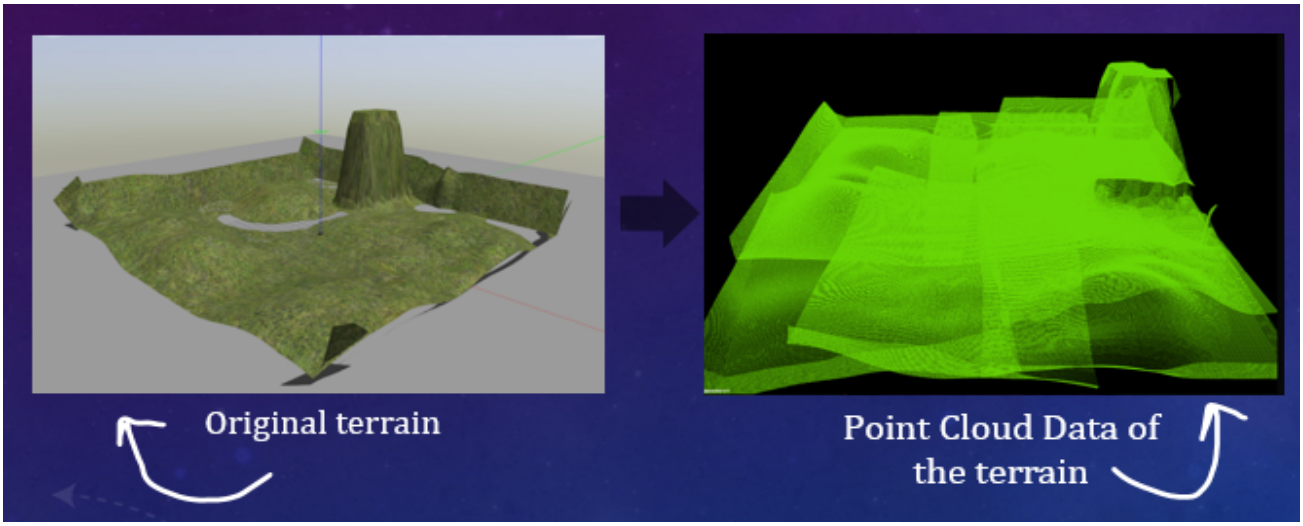
$$\mathbf{r}_{s'} = \mathbf{C}_{s's}\,\mathbf{r}_{s}$$

Point coordinates in the rotated frame

$$\mathbf{p}_{s'}^{(j)} = \mathbf{C}_{s's}\,\mathbf{p}_{s}^{(j)} \quad \text{each point}$$

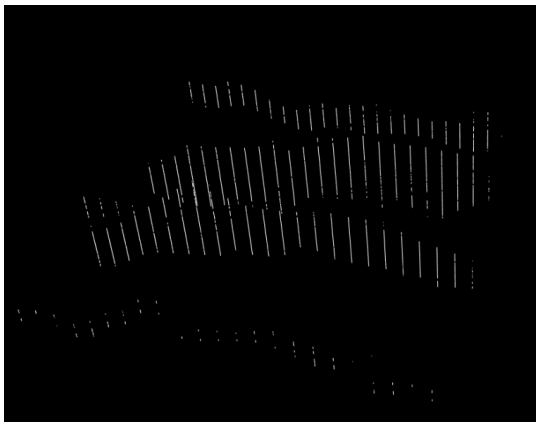
$$\mathbf{P}_{s'} = \mathbf{C}_{s's}\,\mathbf{P}_{s} \quad \text{whole cloud}$$

Obtained Point Cloud Data from ROS is as given below:

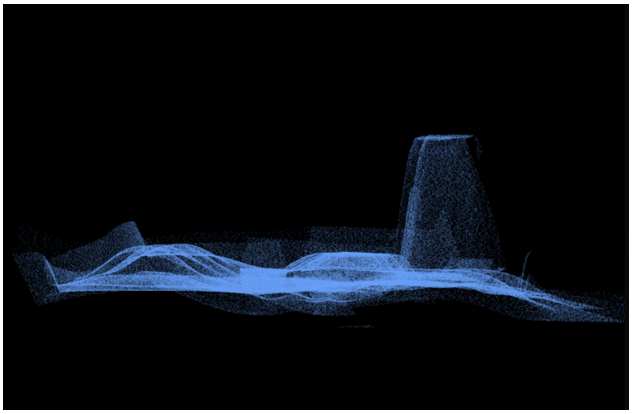


Once we obtained the source point cloud of the terrain post translation and rotation, we moved on to reconstructing the 3D Model of that terrain, as described below:

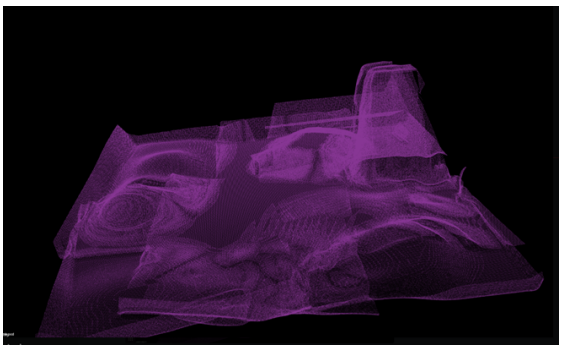
- We initially worked on estimating the normals of our Point Cloud Data set, using two algorithms aka Principal Component Analysis and Moving Least Squares, explained theoretically in detail in the sections 2.2.2.1 and 2.2.2.2 respectively.
- Upon considering highly inaccurate outputs we got after our initial attempts in implementing PCA, we went back to consider what our algorithm was actually doing, and realized that surface normals were being computed for all the points in our dataset, that included a significant amount of erroneous that had to be eliminated. This output is as given below:

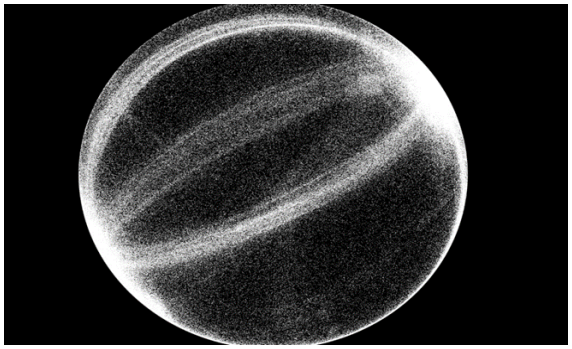


- Here onwards, we looked up PCL-supported methods for this purpose, and found the VoxelGrid filter to be a suitable approach. This filter essentially fits our dataset as a grid, and then consolidates each grid square to be represented by its centroid. This approach is a bit slower than considering the center point, but is more accurate. The terrain post downsampling is as follows:

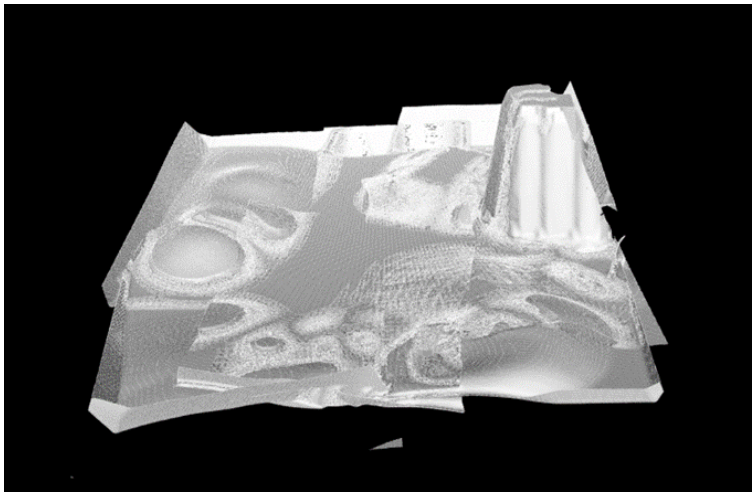


- MLS and PCA outputs post downsampling respectively are as follows:





- Following a considerably better output of PCA and MLS algorithms, we moved on to implement the Meshing Algorithms, aka Greedy Projection Triangulation and Marching Cubes Algorithm. During this implementation due to time constraints, we focused on implementing the Greedy Algorithm appropriately following it with implementation of MC.
- A good amount of time was spent in fine-tuning our parameters for GA, since this algorithm is supposed to be a better fit for convex objects as opposed to our terrain.
- Final Mesh via Greedy Projection Triangulation is as follows:



Thus, we were able to reconstruct the terrain by incorporating Voxel Grid Downsampling for filtering erroneous points, Moving Least Squares method to estimate the normals and Greedy Projection Triangulation to mesh the normals.

4. APPLICATIONS

Drone mapping has a wide scope and potential for construction, agriculture, mining, infrastructure inspection, and real estate. Having a clear, accurate photograph or 3D model of your project area, complete with measurements, is advantageous in terms of decision-making. Following are some applications of drone data:

1. **Civil Engineering Design**: Civil Engineering design requires accurate and precise topographic mapping data to ensure that design parameters reflect real-world conditions. The low altitude requirement of the drone means denser surface sampling which often results in higher accuracy than conventional ground surveys. Drone mapping can also be supplemented with conventional survey measurements on critical hard surfaces and drainage structure invert elevations when needed. Turnaround times are faster and drones can provide digital orthophotos, video, and images to support planning, design, and future project monitoring.

2. **GIS Base Mapping and Data Acquisition**: With an easy-to-deploy aerial mapping drone, we can capture accurate aerial imagery and transform it into 2D orthomosaics (maps) and 3D models of small- and medium-sized sites. Using an aerial drone means we can take to the skies virtually whenever we need. With low operating heights, cloud cover is rarely an issue.

3. **Smart Cities and Commercial Applications**: Drones are becoming a necessity for smart city growth in India. They have become a popular tool for city inspections, business advancement, and disaster relief. With built-in **3D mapping** and cloud technology, drones provide cities the ability to track and record data they never could before. One large benefactor of smart city drones has been streamlined building inspections. Drone surveillance

allows city planners clearer visuals over construction and development sites. Areas that were previously too hard for inspectors to access are now within reach with the help of drones.

4. **Defense Applications**: In many recent wars and even peacetime missions, the drones have performed roles traditionally carried out by manned aircraft. Drones offer two main advantages over manned aircraft. They eliminate the risk to a pilot's life and their aeronautical capabilities such as endurance are not constrained by human limitations. Generally, drones are cheaper to procure and operate as compared to manned aircraft and are employed by militaries around the world for intelligence, surveillance, reconnaissance, electronic warfare, and strike missions. In the future, they could be employed for resupply, combat search and rescue, aerial refueling, and air combat.

5. CONCLUSION & FUTURE WORK

5.1 CONCLUSION

In conclusion, surface reconstruction is an essential, state-of-the-art technique incorporated by several 3D vision applications. Initially, there were a lot of challenges in obtaining the point cloud data of our selected terrain, which led us to abandon our initial approach and use the sjtu_drone model instead. From there onwards were multiple issues as well, which, since we were new to ROS and Gazebo, weren't resolved swiftly but over the course of the entire project. However once the ROS framework was set into place and the parameters of our reconstruction algorithms were fine tuned as far as possible, we were able to reconstruct the terrain in quite an efficient manner. Over these past four weeks we have learnt a lot about drones as well as about various surface reconstruction algorithms in general. This knowledge of theory as well as the first-hand experience in implementing will prove valuable in the future.

5.2 IMPLEMENTING CGAL

Surface reconstruction is a long standing discussion, and over the years several algorithms have been devised in order to tackle this problem. One such meshing algorithm is the Delaunay Triangulation, which in theory, is expected to provide better results for a single frame of Point Cloud Data. However, since there is no PCL support for Delaunay Triangulation, we weren't able to explore this algorithm during the course of our project. DT can be implemented using CGAL (Computational Geometry Algorithms Library) for non-continuous stream of data.

5.3 SURFACE RECOGNITION

Once the surface has been reconstructed, the next target would be to identify what kind of a surface it is, for instance, differentiating between the topography of a hill and a plateau. The pcl_recognition library provides support for recognising objects, and can be explored further in this context.

We also plan on to implement these algorithms on a GPU using OpenGL, it will not only facilitate fast triangulation of meshes but also can help in meshing of the terrain in real time.

Further we would like to increase the scope of our project by implementing surface segmentation into our source file so that meshing of different parts of the terrain can be done more efficiently and we could also include more features or vegetation.

6. REFERENCES

6.1 BASIC UNDERSTANDING

- Gazebo-Tutorials
- Playlist-for-Linear-Algebra
- ROS-Tutorials
- Point-Cloud-Library-Tutorials

6.2 QUADROTOR MODEL

- sjtu-drone

6.3 REFERENCE PAPERS

- Master Thesis on Surface Reconstruction, Navpreet Kaur Pawar
- Comparing Normal Estimation Methods for the Rendering of Unorganized Point Clouds, INGEMAR MARKSTRÖM Section 2.3, Normal Estimation
- Greedy Projection Algorithm