3D Point Cloud Filtering

5LSH0, Computer Vision and 3D Image Processing - Q2 (2023)

Full Name	Student ID
Yokesh Dhanabal	2011468

Eindhoven, February 4, 2024

Contents

1 | Importing point cloud

1.1 | Point cloud generation using stereo camera

When we capture images using stereo camera, the 3D space is projected into 2D space and all the depth information is lost. But, reconstructing the 3D point cloud from 2D images is crucial for various applications such as self-driving cars.
It is possible to get depth information and point cloud from 2D images. This method is called stereo vision and it is similar to depth perception achieved human vision. By comparing what the left and right eye sees our brain processes the disparity and understands the depth of the environment. Following are the steps to generate points cloud from stereo camera.

1. **Image collection**

   As single image has no depth, we need at least two high-quality images captured from different angles with overlap to create a 3D point cloud. This can be achieved by using a stereo camera, a type of camera with two or more lenses with a separate image sensor or film frame for each lens simulating human binocular vision. These lenses capture the views of the same scene at different angles, and these images can be used to calculate the depth of objects in the image by further processing. By this way, the stereo camera can be used to create a discrete set of data points in space (point cloud).

   Additionally, the collected images must include crucial information such as the image's height and width, camera specifications, exposure time, and most importantly, the focal length

2. **Calculate Disparity**

   Disparity is the apparent motion of objects between a pair of stereo images. For generating disparity map, we first match every pixel in one of the captured image with its corresponding pixel in another image. Then we compute the distance for each pair of matching pixels. Finally, the distance values are represented as an intensity image to get the disparity map as shown in figure 1.1.

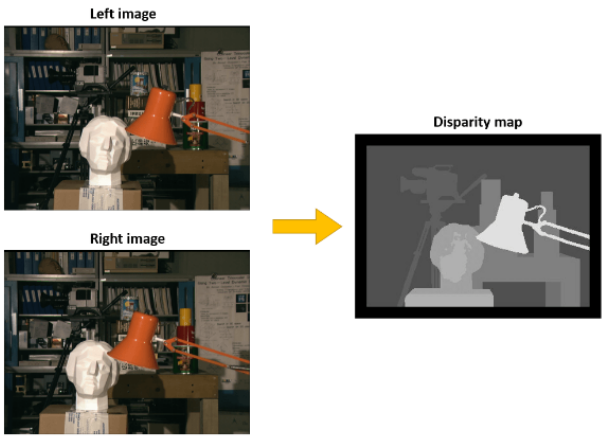

Figure 1.1: Stereo Images to Disparity map

3. **Generate depth map**

   A Depth Map is the size of the original image, but every pixel is encoded with the distance information rather than the colour. Also depth is inversely proportional to disparity. Figure 1.2 shows the relationship between depth and disparity in case of a stereo vision setup.

   If we know the geometric arrangement of the cameras, then the disparity map can be converted into a depth map using triangulation. Below is the formula for calculation depth D, from disparity δ.

$$D = \frac{fB}{\delta}$$

.

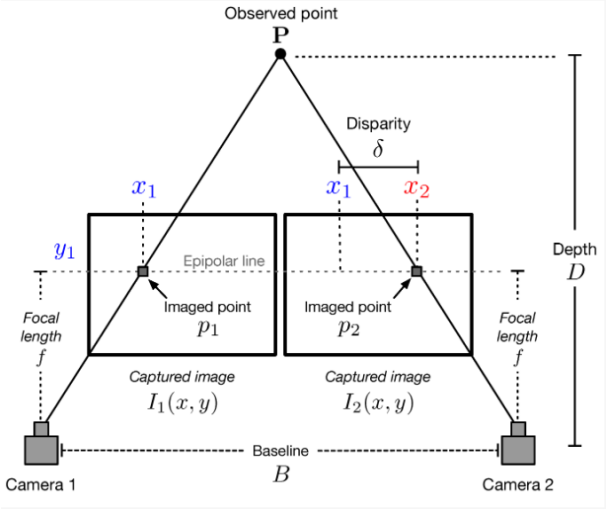

Figure 1.2: Relation between depth (D) and disparity (δ) in stereo image setup

4. Reconstruct 3D point cloud from depth map

A depth image is a 2D image that stores the distance from the camera to each pixel. A point cloud is a set of 3D points that represent the surface of objects. Now we have the depth image and we have to estimate the point cloud from it, that is, we have to transform each depth pixel from the depth image (2D coordinate system) to the camera 3D coordinate system (x, y and z).

The first step is to calibrate the stereo camera to estimate the camera matrix (Intrinsic and Extrinsic matrix) and then use it to compute the point cloud. To find the calibration matrix, we can use certain algorithms such as chess board algorithm, By using this matrix, we can then map each pixel in the depth image to 3D space using the inverse relation from 2D image co-ordinates to 3D world co-ordinates.

1.2 | Reasons for noise in point cloud generated using stereo camera

There are several reasons for noise in the point cloud generated using a stereo camera and they are discussed below

- As discussed in previous section, point cloud generation using stereo camera relies on finding corresponding pixels between the two images to compute the depth. However, this process can be affected by various factors, such as low texture, illumination changes, lens distortion, or sensor noise. Additionally, errors in measurement leads to sparse outliers which corrupt the results even more. These factors can cause incorrect matches or missing matches, resulting in noisy or distorted depth values which results in a noisy point cloud.
- Stereo camera is a passive device, as they do not emit any light pulses to estimate depth. They capture the light reflected from objects and this leads to various noises that originates from those objects such as reflections, shadows, or background objects. This leads to low quality depth images and resulting point cloud.
- The depth value of each pixel is usually stored as an integer value in a certain range, such as 0 to 255. This means that the depth value is rounded to discrete levels, which can introduce errors during 3D reconstruction. This error can be reduced by using a higher bit range for storing pixels.
- Projection errors: The depth value of each pixel is projected to a 3D point using the camera parameters, such as the focal length and the principal point etc. However, these parameters may not be accurate or consistent for different cameras or views, leading to errors during reconstruction to 3D coordinates. The projection errors can be minimized by calibrating the camera parameters or using a camera intrinsic matrix.

1.3 | Node subscribing to point cloud topic

When the bag containing raw point cloud is played using ŕosbag play; it publishes the messages under the '/cloud_map' topic. So, a subscriber node called 'filtering_node' is created and it subscribes to the topic called '/cloud_map'. Figure 1.3 shows the graph visualization of the subscriber node (filtering_node) subscribing to the topic ('/cloud_map') in rqt graph. Figure 1.4 shows the messages displayed (in VS

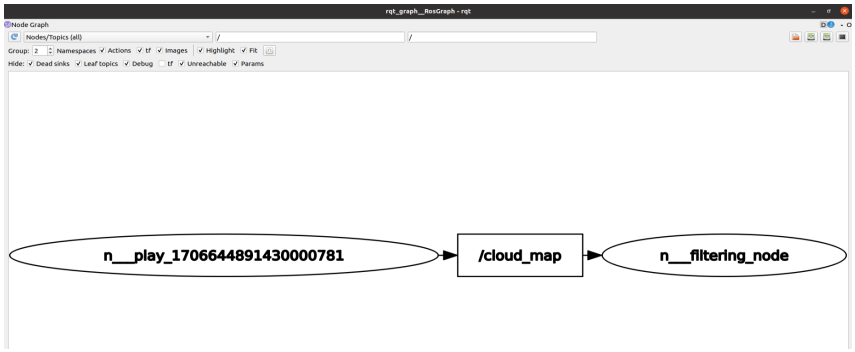

Figure 1.3: Visualization of the filtering_node by rqt graph

code terminal) after subscribed by the 'filtering_node'.

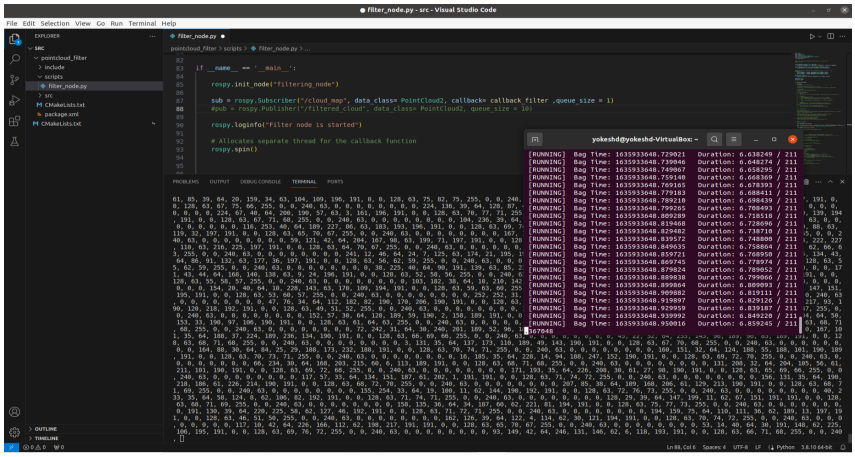

Figure 1.4: Messages subscribed by filtering_node

2 | Designing filtering node in ROS

(Click **here** to download filtered bag file)

2.1 | Designing filtering node

As the subscriber (filtering_node) has been implemented, let us extend the functionality of previously designed 'filtering_node' to do the filtering process and publish the filtered point clouds. The filtering logic is written in the call back function of the node.
Figure 2.1 shows the top view and front view of unfiltered point cloud visualized using RViz. It can be observed that there is a lot of noise and it prevents us to see the distinct structures of the building.

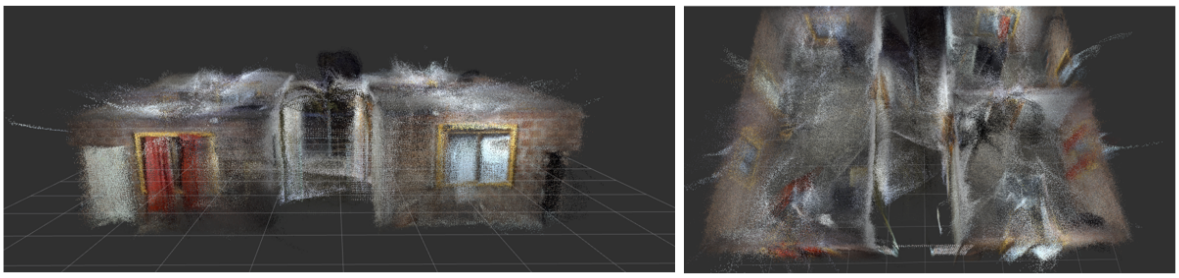

Figure 2.1: Front view (left) and top view (right) of unfiltered point cloud visualized using RViz

The filtering_node manipulates the noisy point cloud data (Steps discussed in next section) and publishes the filtered point cloud under the topic called '/filtered_cloud' which can be recorded in a bag file. It also stores the filtered point cloud in the 'filtered_cloud.pcd' file (In Point Cloud Data file format). Figure 2.2 and 2.3 shows the top view with some markings of distinct structures and front view of filtered point cloud respectively. Now the structures of the building can be clearly seen as shown in figure 2.2

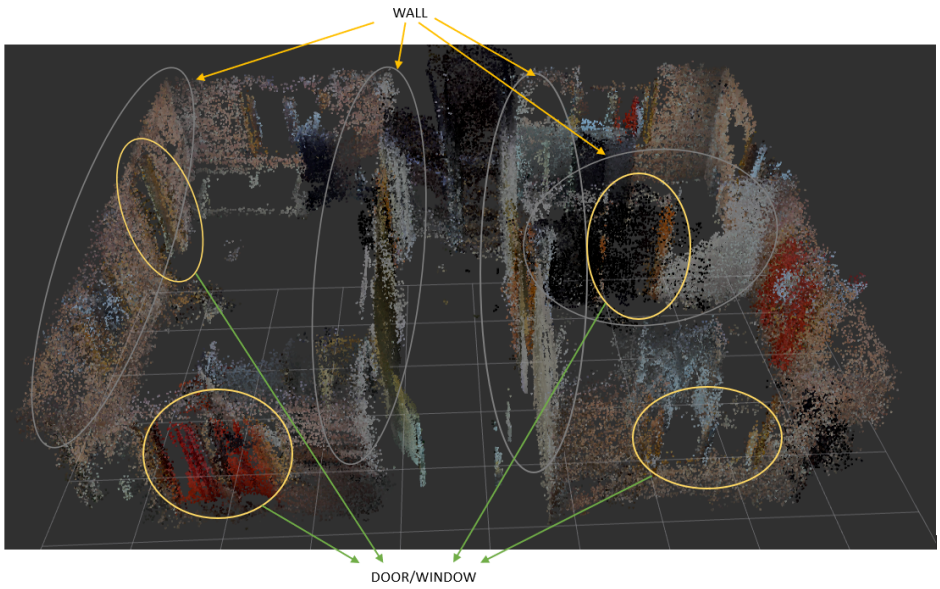

Figure 2.2: Top view of filtered point cloud with markings

2.2 | Filtering procedure

Open3D is primarily used for processing, manipulation and visualization of point cloud data and it can be installed by executing 'pip install open3d'. Figure 2.5 shows the visualization of filtered point cloud using Open3D.

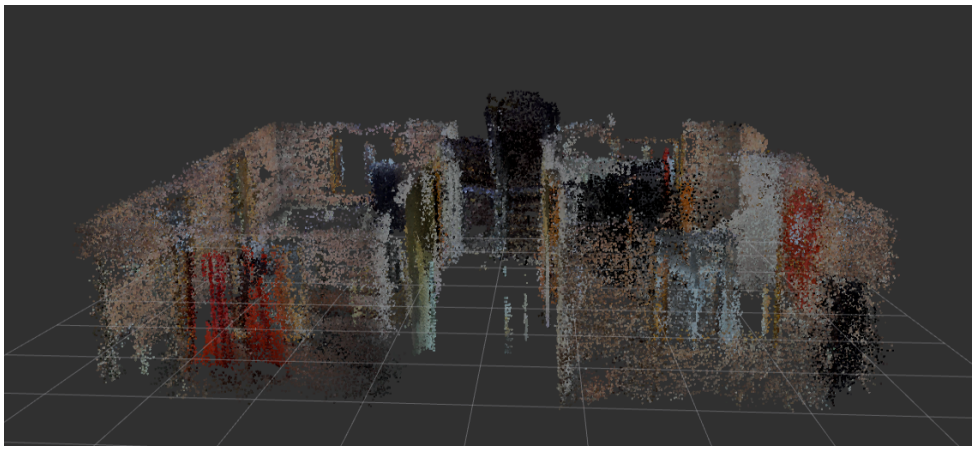

Figure 2.3: Front view of filtered point cloud

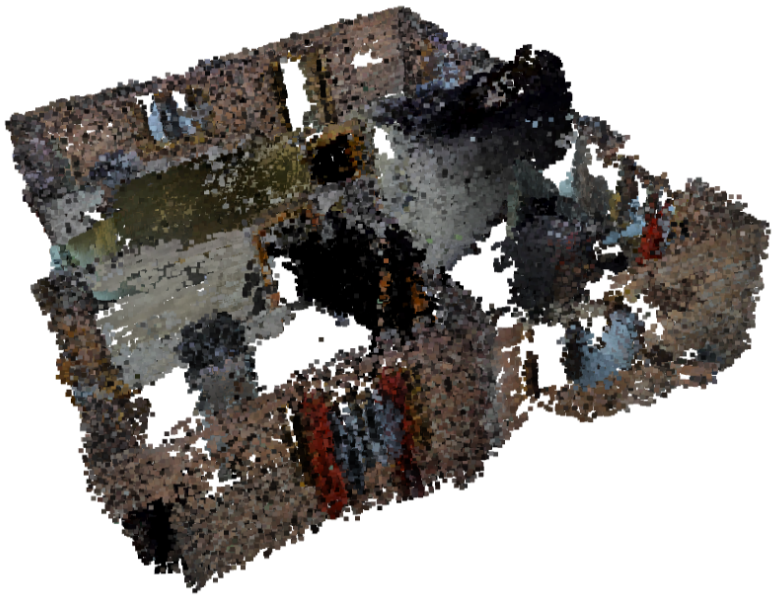

Figure 2.4: Visualization of filtered point cloud using Open3D

The whole filtering logic is written in the call back function (callback_filter) of the filtering node. The callback_filter takes the current point cloud data subscribed by the filtering node as input argument.
Step 1: As the first step, the data type of subscribed messages is converted from sensor_msgs/PointCloud2 to open3d.geometry.PointCloud() object.
Step 2: Next, we estimate normal vectors for each point in the point cloud using open3d.geometry.estimate_normals(). Based on the normal vectors, we filter out the points belonging to ceiling and floor. This is done by removing the points if their normal vector has 'z' component with magnitude greater than 0.1. The estimated normal vectors for down-sampled point cloud (before filtering) is shown in figure 2.6.
Step 3: Next, to filter out the outliers, we use an in-built radius outlier removal (remove_radius_outlier()) method available in open3D. It removes the points that have few neighbors in a given sphere around them. In our case, the number of neighbours is 20 and the radius of sphere is chosen as 0.1. Figure 2.7 shows the overall process of point cloud filtering.
Finally, the filtered point cloud is published under the topic called '/filtered_cloud' and recorded in this bag file. The Rqt graph visualization of filtering node can be seen in figure 2.7.

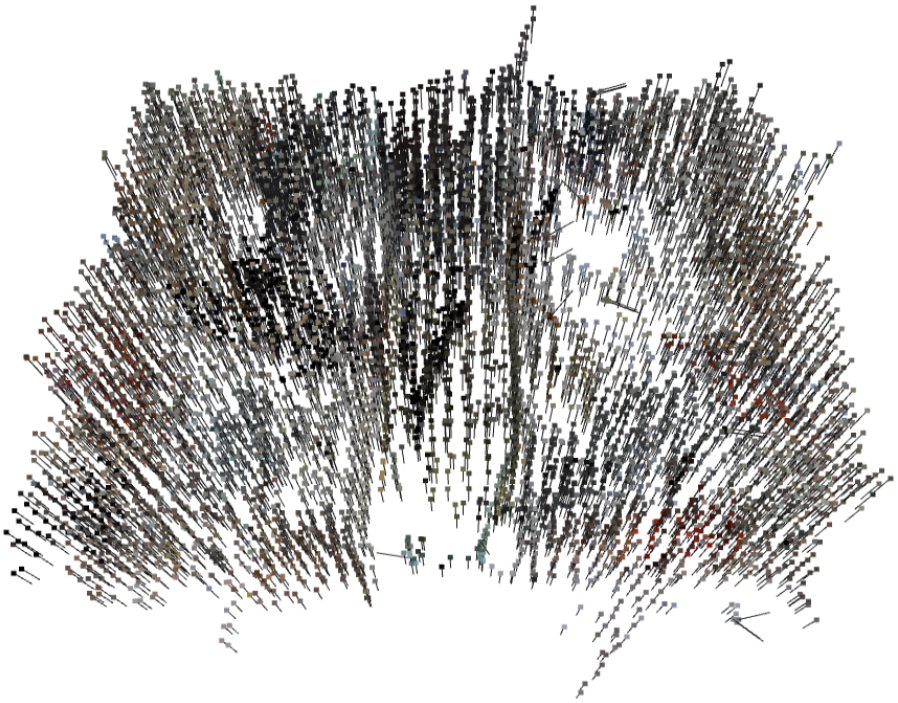

Figure 2.5: Estimated normal vectors before filtering

2.3 | Further improvements in filtering node

- Currently, the filtering node is implemented using Python. In-order to improve the speed of the filter node and overall process, we can use C++ instead of python.
- Open3D library and its in-built functions are used for filtering of noisy point cloud and these functions are implemented in the most efficient way. However, as we make these function calls from our call back function, it can reduce the speed of execution of the call back function. To reduce this, we can try to implement required logic by user-defined functions for whenever possible and use it for filtering.
- For this project, the whole logic for filtering is written inside the call back function and it takes too long to process the message. Thus the subscriber misses the next message or blocks the event loop, causing degradation in performance. The callback code can be optimized by using asynchronous operations, caching mechanisms, or message filters

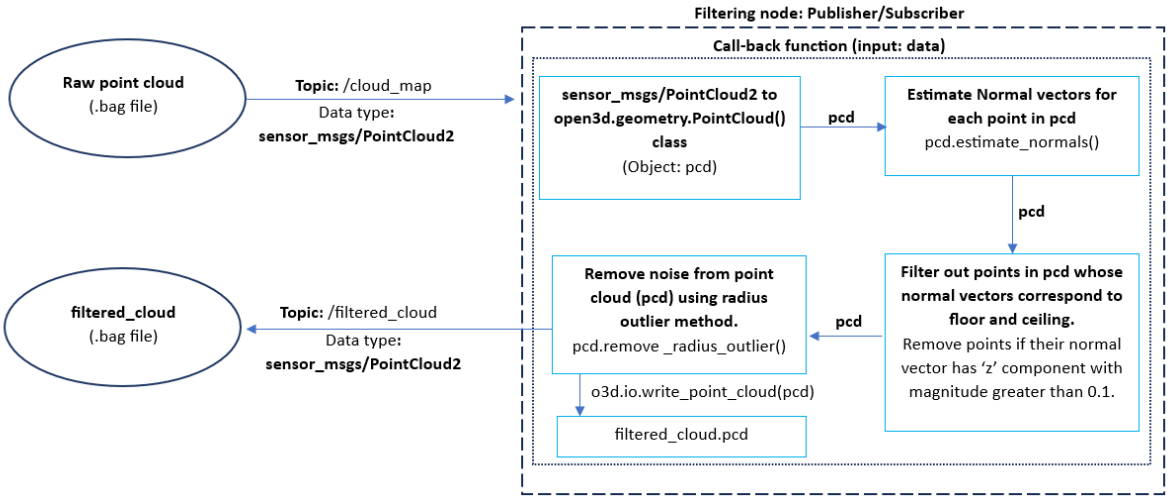

Figure 2.6: Flowchart showing the steps involved in point cloud filtering

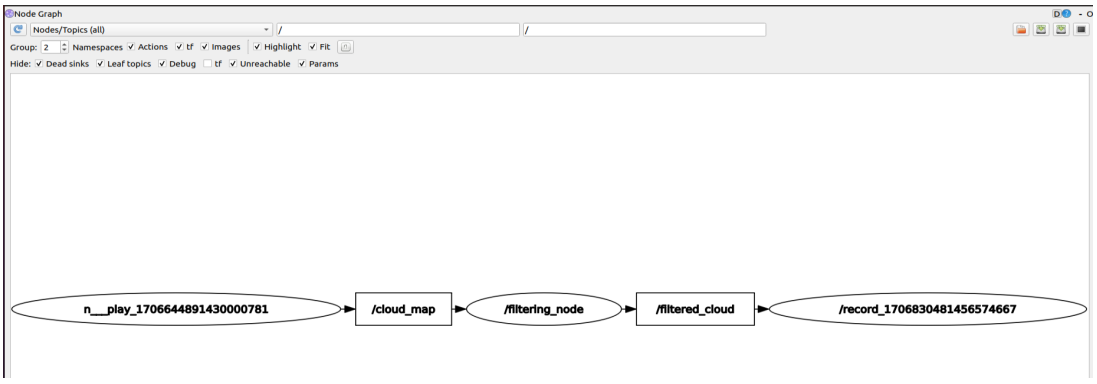

Figure 2.7: Rqt graph visualization of filtering node