Point Cloud Segmentation using Projection: A ROS Implementation

Phuc Tran

Hamburg University of Applied Sciences, Berliner Tor 7, 20099 Hamburg, Germany

Abstract. This paper implements a ROS package for segmenting Point Clouds using Gradual Point Cloud Processing (GPCP) [1]. The implementation is integrated with RTAB-Map to produce a segmented point cloud of the environment. The RGB-D SLAM Dataset [4] was used to validate the implementation. The results showed that GPCP can be applied in robotics to achieve simple and low cost point cloud segmentation, although there is still room for improvements.

Keywords: Gradual Point Cloud Processing · Point Cloud Segmentation · ROS Implementation.

1 Introduction

3D point cloud is one of the most popular representations of 3D data, especially in the robotics. Robots sense their surroundings by combining the sensor inputs. Nowadays, robots can navigate around a building and create a 3D map of the inside with relative high quality using Visual SLAM methods.

Understanding a list of points in 3D space still remains a challenge. While pixels in image that are near to each other can resemble a feature so that convolutional layers can extract meanings from them, point cloud is solely an unordered list of points in 3D space.

Various methods have been proposed to extract features from 3D point clouds. PointNet [6] and PointNet++ [7] uses spatial transformers and complex neural network architectures to work directly with the data structure of point clouds. VoxelNet [8] groups points into voxels, and random sampling is utilized to fix the voxel size, then region proposal network architecture is applied in order to detect features. The mentioned methods, however, require high computing capability and can hardly scale to work with large 3D point clouds. In addition, the size of 3D data in real world applications could be unbounded.

In the previous works [1, 2], Gradual Point Cloud Processing (GPCP) was proposed. GPCP requires odometry (or a generated position and pose of the camera in 3D space) and takes advantage of perspective projection to map the points back into an 2D plane. The projection can be used for feature extraction and segmentation using a conventional 2D model. When RGB image from the camera is available, it can also be used to produce results with higher quality.

This approach focuses on the reusability of existing state-of-the art 2D segmentation models, which are publicly available. GPCP also requires lower computing capability since it only uses 2D models.

This paper focuses on the ROS implementation of GPCP. The implementation is then validated using ROSBags from the RGB-D SLAM Dataset [4].

2 Implementation

This section presents the ROS implementation of GPCP. The nodes are implemented in C++ and Python with standard libraries of ROS. The PCL C++ Library [5] is used for point cloud data handling.

2.1 Overview

Figure 1 depicts the overview of the data flow in this work. The point cloud map of the environment and the odometry information are produced by a node that implements Visual SLAM.

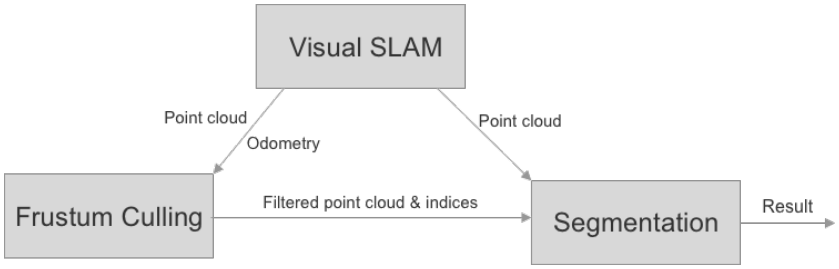


Fig. 1. Overview of the data flow

The culling node, implemented in C++, extracts the camera pose from the odometry information and filter the point cloud. The segmentation node performs projection and image segmentation only on the filtered point cloud. The segmentation result is projected back to the original point cloud. In order to reuse the pre-trained models easily, the segmentation node is implemented in Python and uses the pre-trained models from Pytorch Hub. The next sections will discuss the implementation and the design choices.

2.2 Visual SLAM: RTAB-Map

Real-Time Appearance-Based Mapping (RTAB-Map) [3] is one of the most popular Visual SLAM method in the robotics. RTAB-Map localizes the robot and map the environment incrementally using vision sensors and a loop closure detector. This approach is suitable for large-scale SLAM and can be done in real-time.

This work uses the "rtabmap_ros" package, which is an official ROS wrapper of RTAB-Map. The RGB-D odometry node extracts features from RGB images and depth information from depth images. A random sample consensus approach (RANSAC) is applied for computing the transformation between consecutive images. This node subscribes to the following topics from the cameras:

- /camera/rgb/camera_info: RGB camera metadata
- /camera/rgb/image_color: RGB image
- /camera/depth/image: Depth image

and publishes odometry information to the topic /rtabmap/odom. The RGB-D odometry node is configured as follows:

```
<node pkg="rtabmap_ros" type="rgbd_odometry" name="rgbd_odometry"
    output="screen">
   <remap from="rgb/image" to="/camera/rgb/image_color"/>
   <remap from="depth/image" to="/camera/depth/image"/>
   <remap from="rgb/camera_info" to="/camera/rgb/camera_info"/>

   <param name="Odom/Strategy" type="string" value="0"/>
   <param name="Odom/ResetCountdown" type="string" value="15"/>
   <param name="Odom/GuessSmoothingDelay" type="string" value="0"/>

   <param name="frame_id" type="string" value="/kinect"/>
   <param name="queue_size" type="int" value="10"/>
   <param name="wait_for_transform" type="bool" value="true"/>
</node>
```

The RTAB-Map node subscribes to the three topics of RGB-D camera and the odometry topic published by the odometry node and is configured as follows:

```
 <node name="rtabmap" pkg="rtabmap_ros" type="rtabmap" output="screen"
     args="--delete_db_on_start">
   <param name="subscribe_depth" type="bool" value="true"/>
   <param name="Rtabmap/StartNewMapOnLoopClosure" type="string"
       value="false"/>
   <param name="RGBD/CreateOccupancyGrid" type="string" value="false"/>
   <param name="Grid/VoxelSize" type="string" value="0.01"/>
   <param name="Grid/RayTracing" type="bool" value="true"/>
   <param name="Grid/CellSize" type="string" value="0.01"/>
   <param name="Rtabmap/CreateIntermediateNodes" type="string"
       value="true"/>
   <param name="Grid/GroundIsObstacle" type="string" value="true"/>

   <param name="frame_id" type="string" value="/kinect"/>
   <remap from="rgb/image" to="/camera/rgb/image_color"/>
   <remap from="depth/image" to="/camera/depth/image"/>
   <remap from="rgb/camera_info" to="/camera/rgb/camera_info"/>

   <param name="queue_size" type="int" value="10"/>
```

```
</node>
```

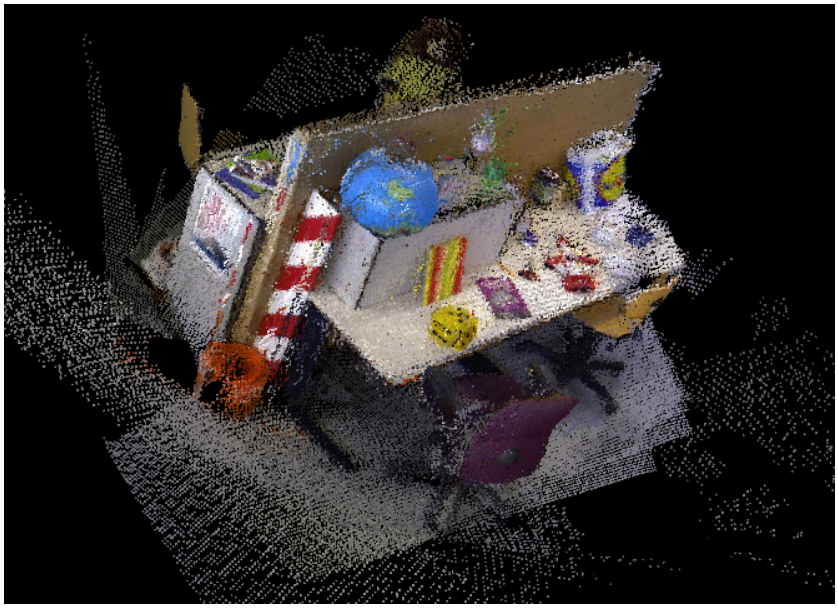


Fig. 2. The point cloud generated by RTAB-Map

The point cloud generated by RTAB-Map is shown in Figure 2. The RTAB-Map node publishes the point cloud to /rtabmap/cloud_map. The size of the point cloud, however, is unbounded, and a large point cloud may have negative impact on performance. In addition, the coordinates of the points in the point cloud use the coordinate system of RTAB-Map, while perspective projection requires the camera to be located in the origin of the coordinate system. Therefore, frustum culling and coordinate transformation are utilized to solve the problems.

2.3 Coordinate Transformation

Coordinate transformation is an important aspect in the robotics. For instance, a point in the coordinate system of the camera will have a different x, y and z coordinates in the coordinate system of a robot arm. Without considering points in different coordinate systems, the calculations required to solve problems in the robotics will quickly become extremely complex.

Coordinates in ROS belong to a frame, which has a coordinate system. There are transforms between the frames, which contain the translations and rotations required to transform one frame into another. The transforms in ROS create a tree structure, so that one can convert a arbitrary point in one frame to another frame.

The odometry information belong to the "odom" frame. In order to perform frustum culling (will be discussed in the next section), the camera position must be transformed into the same coordinate system of the point cloud ("map"). The frustum culling implementation of the PCL Library assumes a coordinate system where X is forward, Y is up and Z is right. However, the coordinate system of the the odometry used in this work is different, where X is left, Y is backward and Z is up. Therefore, the coordinate is multiplied with the following transformation matrix to be converted to the assumed coordinate system.

$$\begin{bmatrix} 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The implementation to convert the odometry data to the camera pose is as follows:

```
Eigen::Matrix4f odom2matrix4f(const geometry_msgs::PoseStamped
    pose){
  auto orientation = pose.pose.orientation;
  auto position = pose.pose.position;
  Eigen::Quaterniond quat;
  quat.w() = orientation.w;
  quat.x() = orientation.x;
  quat.y() = orientation.y;
  quat.z() = orientation.z;
  Eigen::Isometry3d isometry = Eigen::Isometry3d::Identity();
  isometry.linear() = quat.toRotationMatrix();
  isometry.translation() = Eigen::Vector3d(position.x,
      position.y, position.z);

  Eigen::Matrix4f cam2robot;
  // Swap the axes according to the robot
  cam2robot << 0, 0, -1, 0, -1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
      1;
  return isometry.matrix().cast<float>() * cam2robot;
}

void onOdomReceived(const nav_msgs::Odometry &odom_ptr){
  try {
    auto pose = geometry_msgs::PoseStamped();
    pose.pose = odom_ptr.pose.pose;
    pose.header = odom_ptr.header;
    const auto t = tf_buffer.lookupTransform("map",
        pose.header.frame_id, ros::Time(0));
    pose.header.stamp = t.header.stamp;
```

```
        const auto transformed_odom = tf_buffer.transform(pose,
            "map");
        camera_pose = odom2matrix4f(transformed_odom);
        processAndPublish();
      } catch (std::exception& e){
        std::cout << "An unexpected error occured, skipping this
            update: " << e.what() << std::endl;
      }
    }
```

The same principle applies to the filtered point cloud, which was originally produced by RTAB-Map, since it belongs to the "map" frame. In order to perform perspective projection, the points must be transformed to the coordinate system (or the frame) of the camera ("openni_rgb_optical_frame"), where the camera is located at the origin.

```
auto transform =
    tf_buffer.lookupTransform("openni_rgb_optical_frame",
    "map", ros::Time(0));
pcl_ros::transformPointCloud(target, transformed_target,
    transform.transform);
```

2.4 Frustum Culling

In order to handle large point clouds, frustum culling is utilized to drop the points that are not located in the camera's field of view. The view frustum is the field of view of the camera and contains therefore everything that is potentially visible on the image plane. Frustum culling tests intersection of the view frustum and the objects. Only the objects that intersect the frustum are selected.

The Frustum Culling Filter in the PCL C++ Library is used for the implementation in this work. The camera horizontal field of view is set to 80 degrees and the vertical field of view is set to 60 degrees. The camera pose is recomputed upon receiving the odometry information from the RGB-D odometry node.

```
pcl::FrustumCulling<pcl::PointXYZRGB> fc(true);
fc.setInputCloud(cloud);
fc.setCameraPose(camera_pose);
fc.setHorizontalFOV(80);
fc.setVerticalFOV(60);
fc.setNearPlaneDistance(0);
fc.setFarPlaneDistance(50);

pcl::PointCloud<pcl::PointXYZRGB> target;
fc.filter(target);
pcl::Indices indices;
```

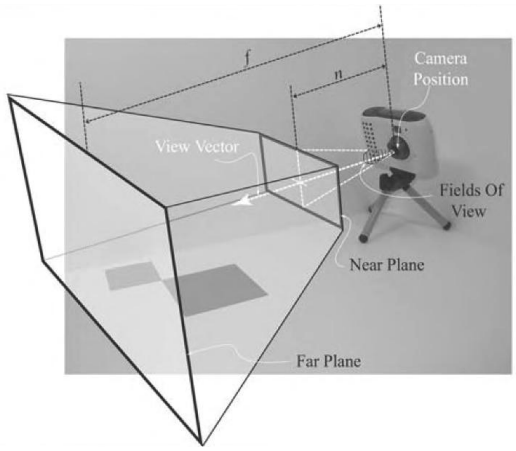


Fig. 3. Illustration of the view frustum [9]

```
fc.filter(indices);
```

The result of frustum culling is a subset of points. Thus, the indices of the selected points are no longer corresponding to those in the original point cloud. In order to allow the back-projection to work properly, the original indices must be maintained. The filtered point cloud and the indices are packed in a message and published to the topic /culling/pc_with_indices.

As mentioned in the concept paper [1], frustum culling does not exclude occluded objects. This leads to the case where multiple points are projected into the same coordinate on an image plane. In order to solve this problem, a Z-Buffering approach must be applied, which will be discussed in the section 2.5.

2.5 Perspective Projection

As explained in the concept paper [1], perspective projection is used to transform input into 2D image, so that it can be processed by an 2D image processing model such as convolutional neural network.

Perspective projection helps to gain information about the location of the points in the camera's image plane at one instance, so that the segmentation result of the RGB image can be projected back to the points in 3D space.

The perspective projection is implemented in the segmentation node, which is written in Python. Upon receiving a point cloud or odometry information, the computation process is started. The perspective projection function consists of solely matrix multiplication and a PIL Image creation for visualization purpose.

```
def project2d(X, colors, P, w=730, h=530):
    """
    :param X: List of points (N, 3)
```

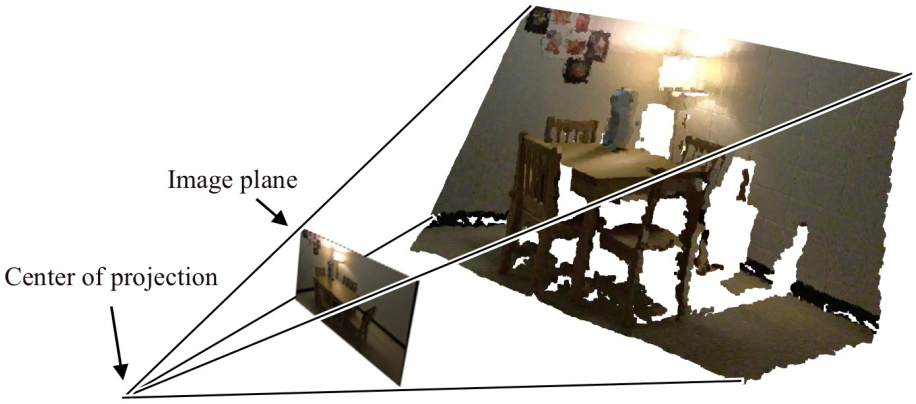


Fig. 4. Perspective projection [2]

```
    :param colors: (N, 3)
    :param P: Projection matrix
    :param w: image width
    :param h: image height
    """
    X = X.swapaxes(0, 1) # (N, 3) -> (3, N)
    X = np.vstack([X, np.ones((1, X.shape[1]))])
    X_raw = (P @ X)
    # Normalize by Z
    Z = X_raw[2, :]
    Xp = np.swapaxes((X_raw / Z)[:2, :], 0, 1).astype(int)

    # Make image
    img = PILImage.new(mode='RGB', size=(w, h), color='black')
    pixels = img.load()
    for i in range(Xp.shape[0]):
        x, y = Xp[i]
        if (x >= w or w < 0) or (y >= h or y < 0):
            continue
        pixels[int(x), int(y)] = tuple(colors[i].astype(int))
    return img, Xp
```

The result of the projection is shown in Figure 5. The projected image is published to the topic "/projection/image" and the list of indices (the variable Xp in the code above) is used for the back-projection stage, which is discussed in section 2.8

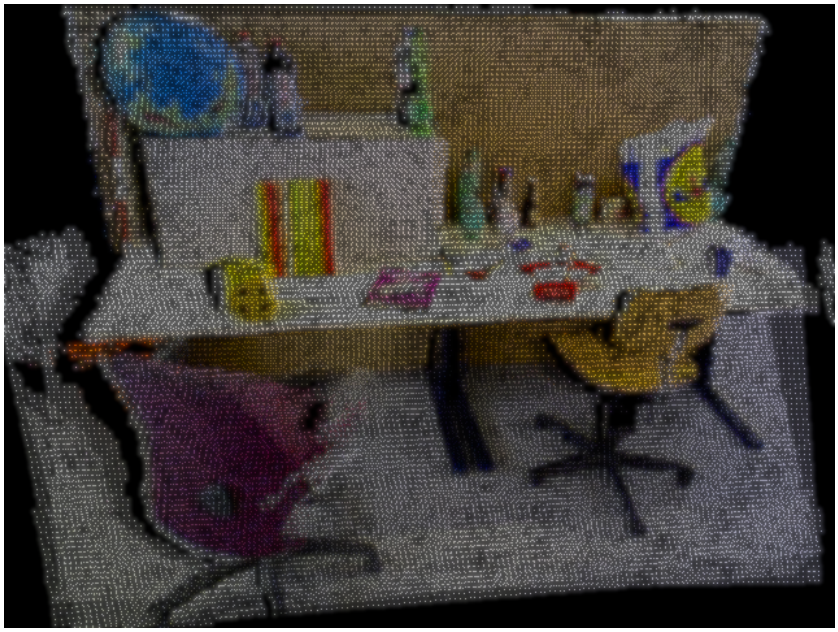


Fig. 5. The result of perspective projection

2.6 RGB Image and Projection Matching

The communication in ROS is asynchronous, and the publish rate of RGB images is different from that of point cloud projection. Therefore, a component to synchronize the 2D projection and its corresponding RGB image is required. This work implements an image buffer with fixed capacity to solve the problem.

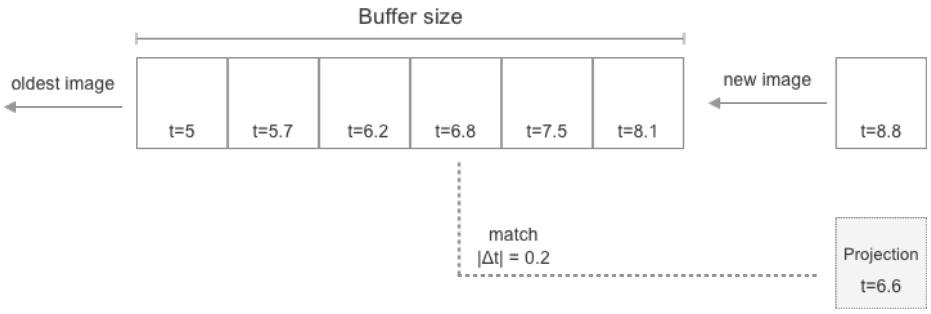


Fig. 6. Illustration of the image buffer

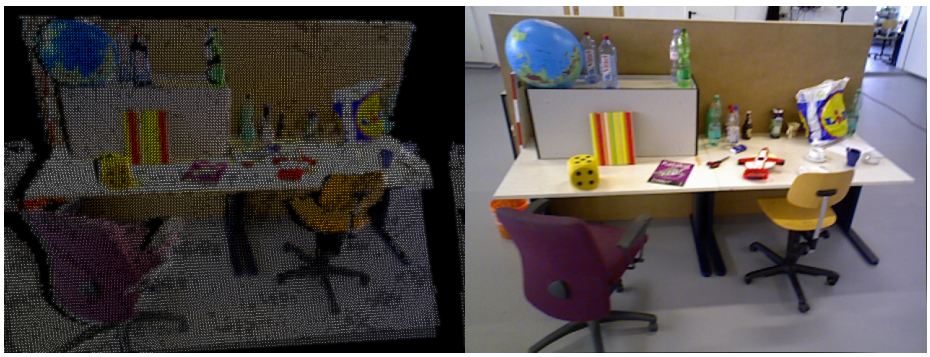
A buffer with fixed size of 20 is used to store the latest RGB images received from the camera, each image is marked with a timestamp t_i. In the lookup stage,

the buffer receives a lookup request that contains a timestamp t_p. The image i will be chosen when the time difference $\Delta_{t_i} = |t_p - t_i|$ reaches the minimum at i, and Δ_{t_i} is smaller than a given tolerance.

The implementation of the lookup function is as follows:

```
def lookup(self, stamp: rospy.Time, tolerance=1):
   stamp_ns = stamp.to_nsec()
   min_diff, min_idx = sys.float_info.max, -1
   for idx, img in enumerate(self._buf):
      diff = abs(stamp_ns - img.header.stamp.to_nsec())
      if diff < min_diff:
      min_diff, min_idx = diff, idx
   if min_idx < 0 or min_diff > tolerance * (10 ** 9):
      return None
   dt = (min_diff / (10 ** 9))
   print(f"Found a image with time diff = {dt:2f}s")
   return self._buf[min_idx]
```

As shown in Figure 7, the image buffer successfully found the matching RGB image for the given projection.



(a) The projection (b) The corresponding RGB image

Fig. 7. The result of the lookup from the image buffer

2.7 Segmentation

This work uses DeepLabV3 model [10] for object segmentation, which has ResNet50 [11] as the feature extractor. The model is pre-trained on the subset of COCO 2017 (train2017) [13]. The model achieved Mean IOU of 66.4% and Global Pixelwise Accuracy of 92.4%.

Fig. 8. The segmentation result produced by the pre-trained DeepLabV3 model

The image is normalized by ImageNet's means and standard deviations. Then the model consumes the image and returns the segmented image as usual.

```
def perform_segmentation(input_image):
   preprocess = transforms.Compose([
      transforms.ToTensor(),
      transforms.Normalize(
         mean=[0.485, 0.456, 0.406],
         std=[0.229, 0.224, 0.225])
   ])
   input_tensor = preprocess(input_image)
   # create a mini-batch as expected by the model
   input_batch = input_tensor.unsqueeze(0)
   if torch.cuda.is_available():
      input_batch = input_batch.to('cuda')
      model.to('cuda')
   with torch.no_grad():
      output = model(input_batch)['out'][0]
   output = output.argmax(0)
```

The result of the segmentation is shown in Figure 8. Only chairs (green) and bottles (blue) are recognized, since these are the only two objects that both exist in the train dataset and the RGB-D SLAM dataset that was used to build the point cloud.

2.8 Back-projection

Frustum culling does not remove occluded objects. Therefore, there may be a case where two points are projected to the same location on the image. This work uses a Z-Buffering approach, where a candidate with the smaller distance to the camera will win.

Since the camera is located at the origin, the point with the smallest distance d will be chosen.

$$d^2 = x^2 + y^2 + z^2$$

The implementation uses an 3-dimensional array of shape $H \times W \times 2$ to store a pair of point index and the squared distance d^2.

The implementation in Python is as follows:

```
# last dimension: [point index, squared distance to camera]
candidate_map = np.ones(result.shape + (2,), dtype=np.float) *
    (-1)
candidate_map[:, :, -1] = sys.float_info.max
for i in range(Xp.shape[0]):
   xi, yi = Xp[i]
   if (xi >= IMG_W or xi < 0) or (yi >= IMG_H or yi < 0):
      continue
   other_candidate_idx, other_candidate_d = candidate_map[yi, xi]
   this_candidate_d = X_dsquared[i]
   if this_candidate_d < other_candidate_d or other_candidate_idx
       < 0:
      candidate_map[yi, xi, :] = [i, this_candidate_d]
```

After performing the Z-Buffering process, the variable candidate_map stores the index of the selected point at each pixel (x_i, y_i). The back-projection process simply consists of a iteration over the point list and a lookup in the segmentation result.

```
X_class = np.ones(Xp.length) * (-1)
for i in range(Xp.shape[0]):
   xi, yi = Xp[i]
   if (xi >= IMG_W or xi < 0) or (yi >= IMG_H or yi < 0) \
   or result[yi, xi] == 0 \
   or candidate_map[yi, xi][0] != i:
      class_i = -1
   else:
      class_i = result[yi, xi]
   X_class[i] = class_i
```

The result of the back-projection is shown in Figure 9. It can be observed that the result was correctly projected back to the 3D space.

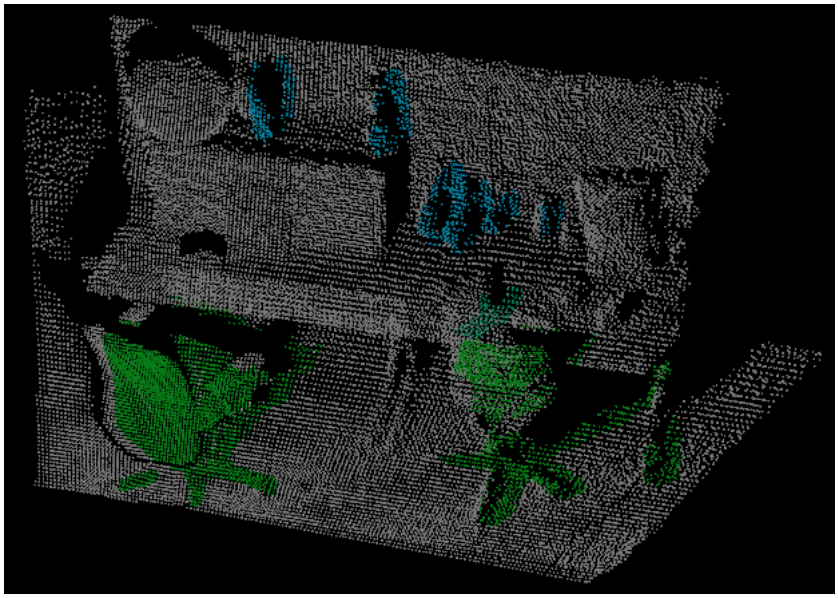


Fig. 9. The segmented point cloud

However, it was found that the current implementation cannot work with RTAB-Map in real-time, since RTAB-Map completely re-create the point cloud after every loop closure instead of building it incrementally. This leads to the problem that the list of indices stored in the memory is no longer valid.

3 Conclusion

This work implements a ROS package for GPCP and demonstrates the potential of GPCP in real-world applications. The implementation is validated using a rosbag from the RGB-D SLAM dataset. GPCP successfully segmented the RGB image and projected the result back to the point cloud in 3D space. However, GPCP cannot work in real-time with RTAB-Map since the implementation of RTAB-Map does not preserve the point ordering after loop closure. Nevertheless, GPCP can still work with RTAB-Map on localization mode, where the map is already built and therefore, the point ordering is preserved.

In order to use GPCP in real-time, a Visual SLAM implementation is required to build the point cloud incrementally and preserve the point ordering of the point cloud.

References

1. Tran, Phuc: Self-supervised gradual 3D point clouds processing for robots, `https://users.informatik.haw-hamburg.de/~acf530/pub/gradual_processing.pdf`. Last accessed 10 Oct 2021

2. Tran, Phuc: An Implementation of Gradual Point Cloud Processing, `https://users.informatik.haw-hamburg.de/~acf530/pub/gradual_processing_impl.pdf`. Last accessed 05 Apr 2022
3. Labbé, Mathieu, and François Michaud. "RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation." Journal of Field Robotics 36.2 (2019): 416-446.
4. Sturm, Jürgen, et al. "A benchmark for the evaluation of RGB-D SLAM systems." 2012 IEEE/RSJ international conference on intelligent robots and systems. IEEE, 2012.
5. Rusu, Radu Bogdan, and Steve Cousins. "3d is here: Point cloud library (pcl)." 2011 IEEE international conference on robotics and automation. IEEE, 2011.
6. Qi, Charles R., et al. "Pointnet: Deep learning on point sets for 3d classification and segmentation." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.
7. Qi, Charles R., et al. "Pointnet++: Deep hierarchical feature learning on point sets in a metric space." arXiv preprint arXiv:1706.02413 (2017).
8. Zhou, Yin, and Oncel Tuzel. "Voxelnet: End-to-end learning for point cloud based 3d object detection." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.
9. Sung, Kelvin, Peter Shirley, and Steven Baer. Essentials of interactive computer graphics: concepts and implementation. CRC Press, 2008.
10. Chen, Liang-Chieh, et al. "Rethinking atrous convolution for semantic image segmentation." arXiv preprint arXiv:1706.05587 (2017).
11. He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.
12. Fei-Fei, Li, Jia Deng, and Kai Li. "ImageNet: Constructing a large-scale image database." Journal of vision 9.8 (2009): 1037-1037.
13. Lin, Tsung-Yi, et al. "Microsoft coco: Common objects in context." European conference on computer vision. Springer, Cham, 2014.