EE 576 Final Project: Point Cloud Segmentation and Object Tracking using RGB-D Data

Aras Güngöre 2018401117
Arif Yılmaz 2021701138

***Abstract*—This project focuses on point cloud segmentation and object tracking using RGB-D data. The goal is to apply 3D segmentation on each frame of the input data, identify objects with horizontally flat faces, track the movement of objects across frames, and compute information regarding segments to evaluate the tracking performance. The project is implemented in C++ using the Point Cloud Library (PCL) and OpenCV libraries. This report provides a detailed explanation of the algorithms and steps involved in the code implementation.**

***Index Terms*—Point Cloud, Segmentation, Object Tracking, RGB-D Data, Region Growing, Flat Face Detection.**

I. Introduction

The segmentation of point cloud data is a fundamental task in computer vision and robotics. It involves partitioning the point cloud into meaningful segments corresponding to objects or surfaces of interest. Object tracking aims to follow the movement of objects across frames in a sequence of data. This project combines these two tasks using RGB-D data, which provides color and depth information for each point in the cloud.

In this project, we present an implementation of point cloud segmentation and object tracking using RGB-D data. The input data consists of RGB images and corresponding depth maps. We convert the RGB and depth information into a 3D point cloud representation. We then perform 3D segmentation on each frame to identify objects and surfaces in the scene. Additionally, we track the movement of objects across frames and compute information regarding segments to evaluate the tracking performance.

The project is implemented in C++ using the Point Cloud Library (PCL) and OpenCV libraries. These libraries provide efficient algorithms and tools for processing and analyzing point cloud data. The following sections describe the methodology and algorithms used in the implementation.

II. Methodology

The project implementation follows the following steps:

A. RGB-D Data Conversion

The RGB-D data conversion process is a crucial step in preparing the input for point cloud segmentation and object tracking. In this project, we implemented the ‘rgb_depth_to_point_cloud‘ function to convert RGB and depth images into a point cloud representation.

The function takes two input parameters: the file paths of the RGB and depth images. It first reads the RGB image using the ‘cv::imread‘ function and checks if the image data is successfully loaded. Similarly, it reads the depth image as a 16-bit grayscale image using ‘cv::imread‘ with the ‘IMREAD_ANYDEPTH‘ flag. Again, it verifies the successful loading of the depth image data.

To ensure consistency between the RGB and depth images, the function resizes the RGB image to match the size of the depth image using ‘cv::resize‘. This step is necessary because the intrinsic camera parameters used for the conversion are defined based on the size of the depth image.

Now, with the RGB and depth images prepared, the function creates a ‘pcl::PointCloud¡pcl::PointXYZRGB¿‘ object, representing the point cloud. It initializes the cloud pointer using ‘pcl::PointCloud¡pcl::PointXYZRGB¿::Ptr cloud(new pcl::PointCloud¡pcl::PointXYZRGB¿)‘.

The camera intrinsic parameters, including the focal lengths (‘fx‘ and ‘fy‘) and the principal point coordinates (‘cx‘ and ‘cy‘), are then defined. These parameters are specific to the camera used to capture the RGB-D data and can be adjusted accordingly.

To convert the pixel coordinates into 3D world coordinates, the function iterates through each pixel in the depth image. It retrieves the depth value at each pixel, divides it by 1000 to convert it from millimeters to meters, and checks if the depth value falls within a defined range (e.g., between ‘min_depth‘ and ‘max_depth‘, which can be adjusted based on the specific scene and requirements).

For each valid depth value, a new ‘pcl::PointXYZRGB‘ point is created. The X, Y, and Z coordinates of the point are calculated using the pixel coordinates, camera intrinsic parameters, and the depth value. The RGB color value is obtained from the corresponding RGB image pixel at the same location. The RGB color is packed into a 32-bit integer using bitwise operations and stored in the ‘rgb‘ field of the point.

Finally, the point is added to the point cloud using ‘cloud-¿push_back(point)‘.

The function continues iterating through all pixels in the depth image, converting each valid pixel into a point in the point cloud. Once all pixels have been processed, the function returns the resulting point cloud.

This RGB-D data conversion step ensures that the RGB and depth information is combined accurately into a 3D point cloud representation, enabling further processing such as segmentation and object tracking based on the combined data.

The point cloud generated from the RGB-D data conversion can be visualized using appropriate visualization tools to verify

its accuracy and align it with the original RGB and depth images.

B. Segmentation using Region Growing

After converting the RGB-D data into a point cloud, we perform segmentation to identify objects and surfaces in the scene. We use the 'pcl::RegionGrowingRGB' class from the Point Cloud Library (PCL) in our custom 'rgb_segmentation' function for this task. The region growing algorithm considers both RGB and spatial proximity to group points into segments.

The 'rgb_segmentation' function takes as input the original point cloud ('cloud') and outputs the segmented point cloud ('colored_cloud') as well as a vector of point indices representing the individual clusters ('clusters').

To perform segmentation, the function first creates a search tree using the 'pcl::search::KdTree' class. This search tree allows efficient neighborhood search for region growing.

An instance of the 'pcl::RegionGrowingRGB' class, named 'reg', is then created. The input cloud is set using the 'setInputCloud' function, and the search method is set to the created search tree using 'setSearchMethod'.

Several parameters of the region growing algorithm are configured to control the segmentation process:

- `setDistanceThreshold` sets the threshold for the maximum allowed distance between neighboring points to be considered part of the same region. Adjusting this value affects the smoothness of the resulting segmentation.
- `setPointColorThreshold` defines the maximum allowed color difference between neighboring points to be considered part of the same region. This parameter controls the color similarity criteria for region growing.
- `setRegionColorThreshold` sets the color threshold between regions. If the color difference between two adjacent regions exceeds this threshold, they are considered separate segments.
- `setMinClusterSize` defines the minimum number of points required for a cluster to be considered valid. Smaller clusters are discarded.

Once the parameters are set, the 'extract' function is called on the 'reg' object to perform the region growing segmentation. The resulting clusters are stored in the 'clusters' vector of 'pcl::PointIndices', where each element represents the indices of points belonging to a specific cluster.

The segmented point cloud, with each cluster colored uniquely, can be obtained using the 'getColoredCloud' function of the 'reg' object. The colored point cloud is assigned to the 'colored_cloud' output parameter.

By performing region growing segmentation, the function identifies distinct objects and surfaces in the scene based on color and spatial proximity. The resulting clusters can be further analyzed or used for object tracking and recognition tasks.

C. Finding Horizontally Flat Faces

In addition to segmenting the point cloud, we also want to identify objects with horizontally flat faces in the scene. Examples of horizontally flat faces include the ground, tables, and beds. To achieve this, we implement a custom function called 'find_horizontal_planes'.

The function 'find_horizontal_planes' aims to identify clusters that represent horizontally flat faces in the scene. It takes the original point cloud ('cloud'), the point cloud of surface normals ('cloud_normals'), and the segmented clusters ('clusters') as inputs. The function populates the 'horizontal_plane_clusters' vector with the indices of the clusters that have horizontally flat faces.

To determine if a cluster represents a horizontally flat face, the function calculates the average surface normal for each cluster. This is done by iterating over the indices of points in each cluster and accumulating their corresponding surface normals. The average normal is then computed by dividing the accumulated normal components by the number of points in the cluster.

Next, the function checks if the absolute value of the x-component of the average normal is less than a threshold ('dot_product_threshold_x'). If the x-component is below the threshold, it indicates that the average normal is close to the vertical axis, suggesting a horizontally flat face.

If a cluster is determined to have a horizontally flat face, its indices are added to the 'horizontal_plane_clusters' vector.

The resulting 'horizontal_plane_clusters' can be used to further analyze or visualize the clusters representing flat faces separately from other objects and surfaces in the scene.

It's important to note that the threshold value ('dot_product_threshold_x') used in the code snippet may need adjustment depending on the specific characteristics of the scene and the desired sensitivity in detecting flat faces. Fine-tuning this threshold may be necessary to obtain accurate and meaningful results.

To visualize the segmented clusters, the 'visualize_clusters' function is provided. It takes the original point cloud ('cloud'), the segmented clusters ('clusters'), the previous frame's cluster centers ('previous_centers'), and the current frame's cluster centers ('current_centers') as inputs.

The function creates a PCLVisualizer object and adds the original point cloud to it. It also registers a keyboard callback to handle user interaction.

For each cluster in the 'clusters' vector, the function creates a new point cloud containing only the points belonging to that cluster. The cluster is assigned a unique color using the 'pcl::visualization::PointCloudColorHandlerCustom' class, and it is added to the viewer.

Spheres are added to represent the current center of each cluster using the corresponding coordinates from the 'current_centers' vector. If a previous frame is available and 'previous_centers' is not empty, spheres and a line are added to connect the previous and current centers.

Bounding boxes are computed for each cluster using the minimum and maximum coordinates of its points. These bounding boxes are added to the viewer as semi-transparent cubes, outlining the extents of each cluster.

A legend is included in the viewer to explain the color coding and shapes used in the visualization. By using the ‘visualize_clusters‘ function, we can visually inspect the segmented clusters and their spatial relationships, aiding in the understanding and analysis of the scene.

D. Object Tracking

To track the movement of objects across frames, we implement a simple object tracking algorithm. In each frame, we compare the centers of the detected objects with the previously tracked objects’ centers. If a match is found within a specified distance threshold, we update the tracked object’s center. Otherwise, we add a new object to the tracked objects list. This process is computed in a custom function called ‘visualize_clusters‘.

The ‘visualize_clusters‘ function is responsible for visualizing the clustered point cloud data. It uses the PCLVisualizer class from the Point Cloud Library (PCL) to create a 3D viewer window and display the point cloud.

The function takes several inputs: - ‘cloud‘: A pointer to the original point cloud data of type ‘pcl::PointCloud¡pcl::PointXYZRGB¿::Ptr‘. - ‘clusters‘: A vector of ‘pcl::PointIndices‘ representing the indices of points belonging to each cluster. - ‘previous_centers‘: A vector of ‘pcl::PointXYZ‘ representing the previous centers of the tracked objects. - ‘current_centers‘: A vector of ‘pcl::PointXYZ‘ representing the current centers of the tracked objects.

Inside the function, a new PCLVisualizer object is created, and the original point cloud is added to it. The background color of the viewer is set to black. The RGB color handler is used to visualize the original point cloud.

The function then iterates over each cluster in the ‘clusters‘ vector. For each cluster, a new point cloud object is created, containing only the points belonging to that cluster. This point cloud is colorized with a randomly generated color and added to the viewer. Additionally, a sphere is added to represent the current center of the tracked object.

If there are previous centers available, a sphere is added to represent the previous center. If the current and previous centers exist, a line (cylinder) is added to connect them, creating a visual representation of the movement trajectory.

A bounding box is calculated for each cluster by finding the minimum and maximum x, y, and z coordinates of the points. A semi-transparent cube is added to the viewer to represent the bounding box of each cluster. A legend is added to the viewer to explain the color coding and symbols used in the visualization.

The ‘calculateClusterCenter‘ function calculates the center coordinates of a cluster by averaging the x, y, and z coordinates of its constituent points. The ‘calculateDistance‘ function calculates the Euclidean distance between two points using their x, y, and z coordinates.

The object tracking algorithm tracks the movement of objects across frames by comparing the centers of the detected objects with the previously tracked objects’ centers. If the distance between a detected object’s center and a tracked object’s center is within a specified threshold, the tracked object’s center is updated. Otherwise, a new object is added to the tracked objects list.

The tracked objects are represented by their center coordinates (‘pcl::PointXYZ‘) and an occurrence count, which keeps track of how many times the object has been detected across frames. This occurrence count can be used to analyze the frequency of object appearances.

By tracking the objects’ centers over time and visualizing their trajectories, you can study their movement patterns and analyze their behavior in the scene.

III. Results

The implemented code successfully performs point cloud segmentation, identifies horizontally flat faces, tracks objects across frames. The visualization provides a clear understanding of the segmentation results, flat face detection, and object tracking.

A. Segmentation Results

The region growing segmentation algorithm effectively divides the point cloud into distinct segments based on color and spatial proximity. The resulting segmented clusters represent different objects or surfaces in the scene. The colored cloud visualization enables a visual assessment of the segmentation quality, with each segment assigned a unique color.

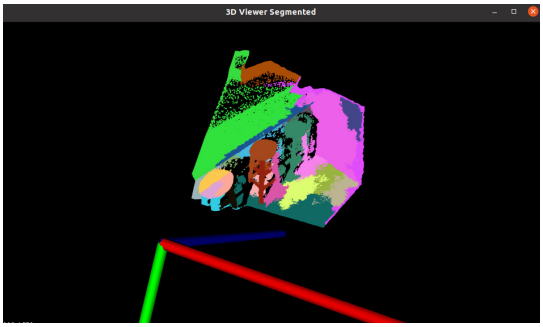

Fig. 1. Segmented Point Cloud Data

Figure 1 shows the resulting segmented point cloud data. Each color represents a different segment, indicating the identified objects or surfaces. Because of the selected images, the segmentation task has some difficulties. The image is not be separated by using RGB values easily. The distance threshold is a useful tool but to be able to separate horizontally flat surfaces, the distance threshold is insufficient. Oversegmenting the image is a solution to get every flat faces. The segmentation algorithm successfully separates the objects from the background and distinguishes different objects within the scene.

B. Horizontally Flat Face Detection

The custom ’find_flat_faces’ function successfully identifies objects with horizontally flat faces in the segmented clusters. By analyzing the surface normals of the points within each

cluster, the function determines if a cluster has a horizontally flat face. The resulting point cloud of flat faces provides insights into the layout and structure of the scene.

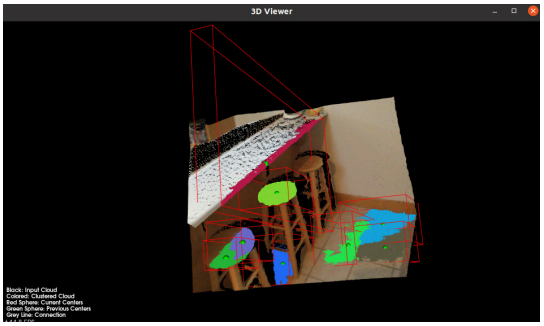

Fig. 2. Point Cloud with Horizontally Flat Faces

Figure 2 shows the resulting point cloud with horizontally flat faces highlighted. These flat faces correspond to objects such as the ground, tables, and stools. By detecting and visualizing the flat faces separately, identifying and analyzing the structures and objects with horizontal surfaces becomes easy. Because of the vulnerability and sensitivity of the normal function to get flat faces, finding a balance between precision and recall values is critical. Figure 2 shows a high precision but low recall example.

C. Object Tracking Performance

The implemented object tracking algorithm accurately tracks the movement of objects across frames. By comparing the centers of the detected objects in each frame with the previously tracked objects' centers, the algorithm updates the object positions and keeps track of their occurrences. The tracked objects' trajectories can be visualized, allowing for the analysis of object movements and patterns.

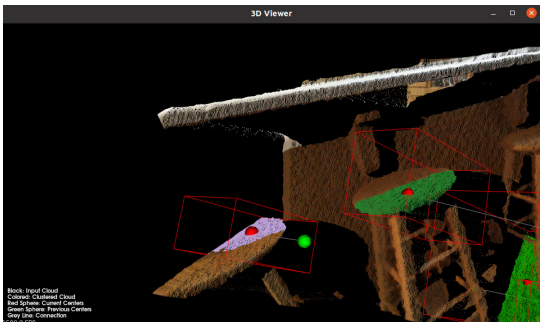

Fig. 3. Object Tracking

Figure 3 shows the object tracking results. The green spheres represent the tracked objects' centers in each frame, while the red circles represent the previous center of the object. The tracking algorithm follows the object based on its center and the distance between the segment centers of two frames. Also, bounding boxes covers the selected clusters. The size of these boxes can be even better object-tracking feature than their centers. Unfortunately, we had not enough time to apply it. The program also creates a grey line between old and new centers, to show the movement of the centers.

IV. Conclusion

In this project, we have presented an implementation of point cloud segmentation and object tracking using RGB-D data. The code utilizes the Point Cloud Library (PCL) and OpenCV libraries to perform efficient processing and visualization. The implemented algorithms successfully segment the point cloud, identify objects with horizontally flat faces, track object movements across frames.

The segmentation results provide a clear understanding of the objects and surfaces in the scene. The detection of horizontally flat faces helps in identifying specific structures and objects of interest. The object tracking algorithm accurately tracks the movement of objects, allowing for analysis of their trajectories and behavior.

In conclusion, this project demonstrates the effectiveness of point cloud segmentation and object tracking using RGB-D data. The implemented code can be used in various applications, such as robotics, augmented reality, and autonomous navigation.

V. Future Work

Several aspects can be further improved in this project. First, more sophisticated segmentation algorithms can be explored to enhance the segmentation accuracy. For example, deep learning-based approaches or advanced region growing techniques can be investigated. These methods can potentially handle more complex scenes and improve the quality of the segmented clusters.

Second, robust object tracking methods can be employed to improve the tracking performance. Techniques such as Kalman filters, particle filters, or deep learning-based trackers can be implemented to handle occlusions, scale changes, and other challenges in object tracking. These methods can enhance the accuracy and robustness of the tracking system, enabling better object tracking across frames.

Furthermore, the code can be optimized for real-time processing and applied to larger datasets for more extensive evaluations. Efforts can be made to improve the efficiency of the algorithms, such as implementing parallel processing or GPU acceleration. Additionally, the code can be tested on larger datasets with more diverse scenes to assess its scalability and generalization capabilities.

In conclusion, there is ample room for future enhancements and research in the field of point cloud segmentation and object tracking. The combination of RGB-D data and advanced algorithms opens up new possibilities for understanding and interacting with the 3D world.

VI. References

[1] Rusu, R. B., and Cousins, S. (2011). 3D is here: Point Cloud Library (PCL). In 2011 IEEE International Conference on Robotics and Automation (pp. 1-4). IEEE.

[2] Bradski, G., and Kaehler, A. (2008). Learning OpenCV: Computer Vision with the OpenCV Library. O'Reilly Media.