3D Point Cloud Processing & Analysis Project Documentation

Introduction

In this project, I developed a comprehensive pipeline for 3D room segmentation using point cloud data. The main objectives included data acquisition, preprocessing, segmentation and visualization, with a focus on applying advanced techniques such as plane fitting, PointNet and various clustering methods. The project utilized publicly available datasets, including the Stanford 2D-3D Indoor Scenes Dataset and the KITTI dataset, to demonstrate the effectiveness of the pipeline across different types of 3D data. By integrating models like PointNet, I aimed to enhance the accuracy and robustness of 3D segmentation tasks.

Import Libraries.

First, I ensured that the necessary libraries were installed using pip install for open3d, numpy and matplotlib. Once installed, I imported the required libraries. The os library was used for handling directory paths, numpy for numerical operations and torch along with related modules for deep learning tasks, such as using PointNet. Additionally, I imported open3d for point cloud processing and matplotlib.pyplot for plotting and visualization. This setup prepared the environment for subsequent steps in the 3D room segmentation.

Data Acquisition

To begin data acquisition, I set the dataset_base_dir to the path where the Stanford 2D-3D Indoor Scenes Dataset is stored. The dataset contains multiple areas, each representing different indoor scenes. I listed all directories within the base directory to ensure that the dataset was correctly loaded.

Next, I selected one specific area (Area 1) for exploration and listed the contents of this directory to identify all point cloud files available.

Load and Visualize Point Cloud Data

After identifying the point cloud files in Area 1, I selected one specific file for further processing. The chosen file contains point cloud data for "office_1".

In this step, I specified the path to the office_1 point cloud file. The file was loaded using numpy.loadtxt with space as the delimiter, as the data is formatted with XYZ coordinates followed by RGB color values.

Next, I created an Open3D Point Cloud object and assigned the loaded XYZ coordinates to it. If the file contains RGB values (which it does in this case), I normalized these values to the range [0, 1] and assigned them to the point cloud object.

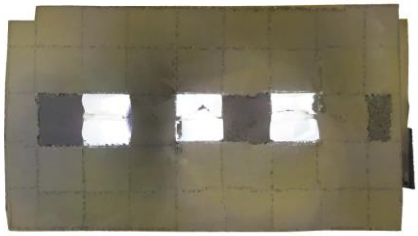
Finally, I visualized the point cloud using Open3D's visualization tools, which allows for an interactive exploration of the 3D point cloud data.

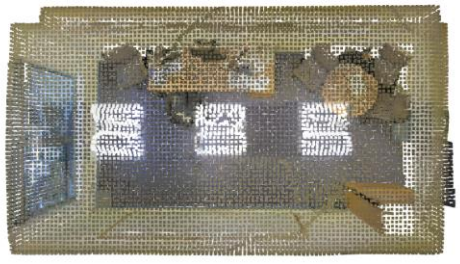
Data Preprocessing

Voxel Downsampling

For the data preprocessing stage, I started with voxel downsampling to reduce the number of points in the point cloud, making it more manageable for further processing. I defined a voxel size of 0.05 and used Open3D's voxel_down_sample function to downsample the point cloud.

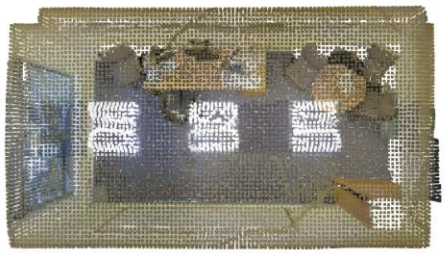
Next, I visualized the original and downsampled point clouds side by side for comparison. This step allowed me to ensure that the downsampling process retained the essential features of the point cloud while significantly reducing the number of points.





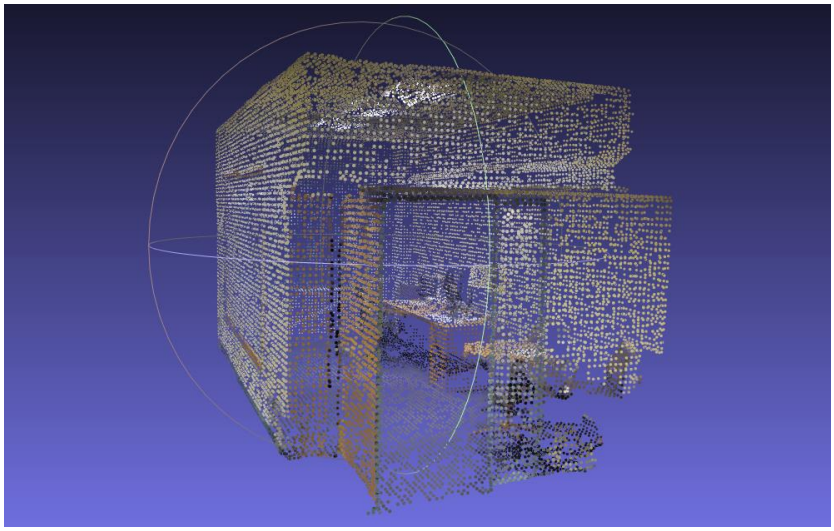
Statistical Outlier Removal

Next, I applied statistical outlier removal to clean the point cloud by removing noise and outliers. I defined parameters for the number of neighbors (nb_neighbors = 20) and the standard deviation ratio (std_ratio = 2.0). Using Open3D's remove_statistical_outlier function, I separated the inliers (points that are part of the main structure) from the outliers (noise). I then visualized the inliers and outliers separately to confirm the effectiveness of the outlier removal process.



Radius Outlier Removal

To further refine the point cloud, I applied radius outlier removal. I defined a radius of 0.05 and a minimum number of neighbors (min_neighbors = 2). This method removes points that do not have a sufficient number of neighboring points within the specified radius. After applying radius outlier removal, I visualized the cleaned point cloud to ensure the removal of isolated points. Finally, I saved the cleaned point cloud as a .ply file for future use.



3D Harris Corner Detection

I implemented 3D Harris Corner Detection to identify points of interest in the point cloud that are likely to be corners. This is useful for further segmentation and feature extraction. The process involved calculating Harris response values for each point in the cloud based on its local neighborhood.

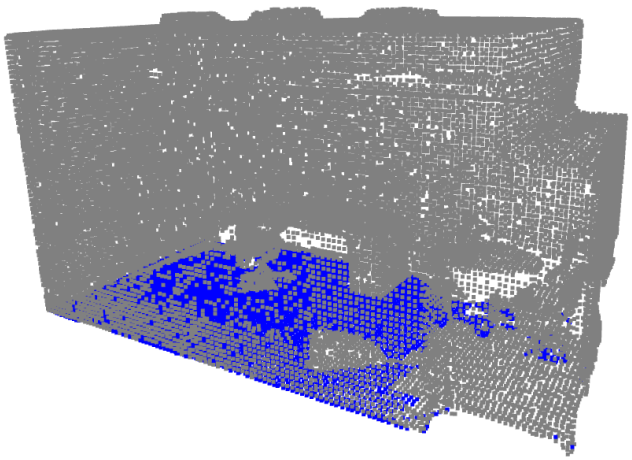
To visualize the detected corners, I painted the entire point cloud gray and highlighted the corners in red. This visualization helped in confirming that the detected corners were correctly identified. Finally, I saved the point cloud with the highlighted corners as a .ply file for future use.

Ground Plane Segmentation

For ground plane segmentation, I defined a height threshold of 1.0 to filter out points based on their Z coordinate (height). Points below this threshold were considered potential ground points. I created a new point cloud with these filtered points for more efficient processing.

Using the filtered point cloud, I applied plane segmentation using the RANSAC algorithm to identify the ground plane. The RANSAC algorithm iteratively fits a plane to the points, maximizing the number of inliers within a specified distance threshold. The result was a plane model representing the ground and a set of inliers that lay on this plane.

I extracted the ground plane points (inliers) and the remaining points (outliers). To distinguish between them, I colored the ground plane points blue and the non-ground points gray. This visual differentiation helped in verifying the accuracy of the ground plane segmentation.

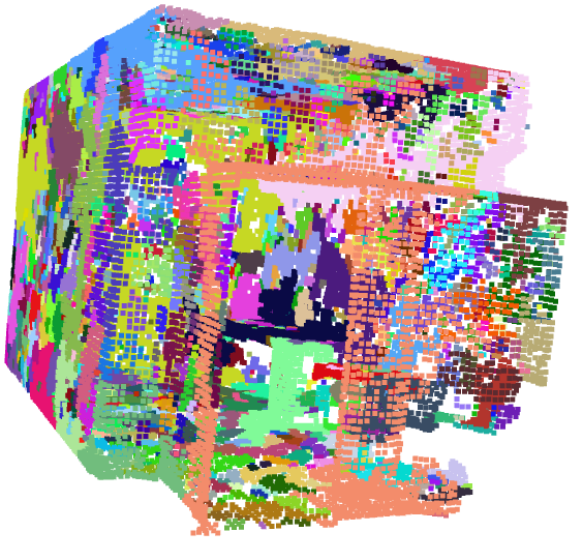


Initial Region Growing Segmentation

For initial region growing segmentation, I implemented a custom function to cluster points in the point cloud based on spatial proximity. The region_growing_segmentation function iteratively grows regions from seed points, assigning labels to points that are within a specified distance threshold. This process continues until all points are assigned to clusters or identified as noise.

I loaded the point cloud and applied the region growing segmentation on non-ground points. The function identified clusters in the point cloud, which were labeled and counted to determine the number of distinct clusters.

To visualize the clusters, I assigned random colors to each cluster, with black representing noise points.



Refined Segmentation: Large Planes and Smaller Objects

1. Normal Computation

To prepare for refined segmentation, I first computed the normals for the point cloud using Open3D's normal estimation functions. Normal vectors are essential for accurate plane segmentation and object detection.

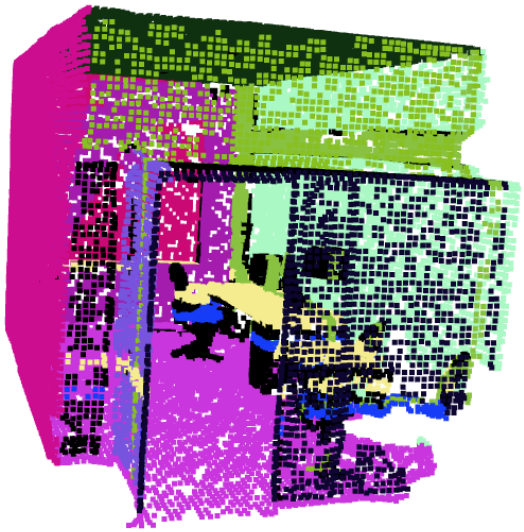
2. Large Plane Segmentation

Using the computed normals, I segmented large planes in the point cloud such as walls, ceilings and floors. The segmentation was performed using the RANSAC algorithm, which iteratively fits planes to the point cloud data. Planes with a sufficient number of points were considered significant and were extracted from the point cloud. These planes were then uniformly colored for easy visualization.

3. Smaller Object Segmentation

After removing the large planes, the remaining points represented smaller objects in the scene. I applied a custom region growing segmentation algorithm to these points, ensuring that clusters below a certain size were discarded as noise. This segmentation process helped in identifying individual objects like furniture.

I assigned random colors to each identified plane and object cluster for visualization. The combined point cloud, with both large planes and smaller objects colored distinctly, was then visualized.



Counting Task

For the counting task, I aimed to identify and count unique objects in the point cloud based on their colors. I started by loading the previously saved segmented and colored point cloud from the .ply file. A tolerance value for color comparison was defined to account for minor variations in color values. I then identified unique colors in the point cloud, each corresponding to a different object and determined the number of distinct objects by counting these unique colors.

Next, I created a dictionary to map each unique color to the corresponding points in the point cloud. This mapping allowed for easy access to the points representing each object. The count of unique objects was printed, confirming the number of distinct entities identified by their colors.

PointNet (Google Colab)

To implement PointNet for 3D point cloud classification and segmentation, I started by cloning the PointNet repository from GitHub.

Preparing the Dataset
I prepared the dataset by defining the source and destination directories. The source directory contained the Stanford 3D Dataset and the destination directory was where the dataset would be copied for use with PointNet. I used shutil to copy the dataset from the source to the destination directory. Once the dataset was copied, I listed the contents of the destination directory to ensure that all files were successfully transferred and organized correctly.

Data Collection and Preprocessing for PointNet

I defined and executed a function, run_data_collection, to automate the data collection and preprocessing process. This function set up the necessary directories, read annotation paths and processed each annotation file to collect point cloud data and their labels. The processed data was saved in a .npy format, which is efficient for storing large numerical arrays and compatible with PointNet.

This comprehensive data collection and preprocessing step ensured that the Stanford 3D Dataset was correctly formatted and organized, making it ready for effective training and evaluation of the PointNet model.

Train Model
To ensure efficient training, I first checked for GPU availability using PyTorch. Utilizing a GPU significantly speeds up the training process for deep learning models.

I trained the PointNet model for semantic segmentation using the provided script train_semseg.py. The command specified the model (pointnet2_sem_seg), test area (--test_area 1) and the directory for logging training information (--log_dir pointnet2_sem_seg). The training

parameters included a batch size of 32, 10 epochs, a learning rate of 0.001 and a decay rate of 0.0001.

During training, the model processed a total of 55,648 samples in the training set and 10,751 samples in the test set. The training and test data were loaded successfully and the training process was executed with the specified parameters.

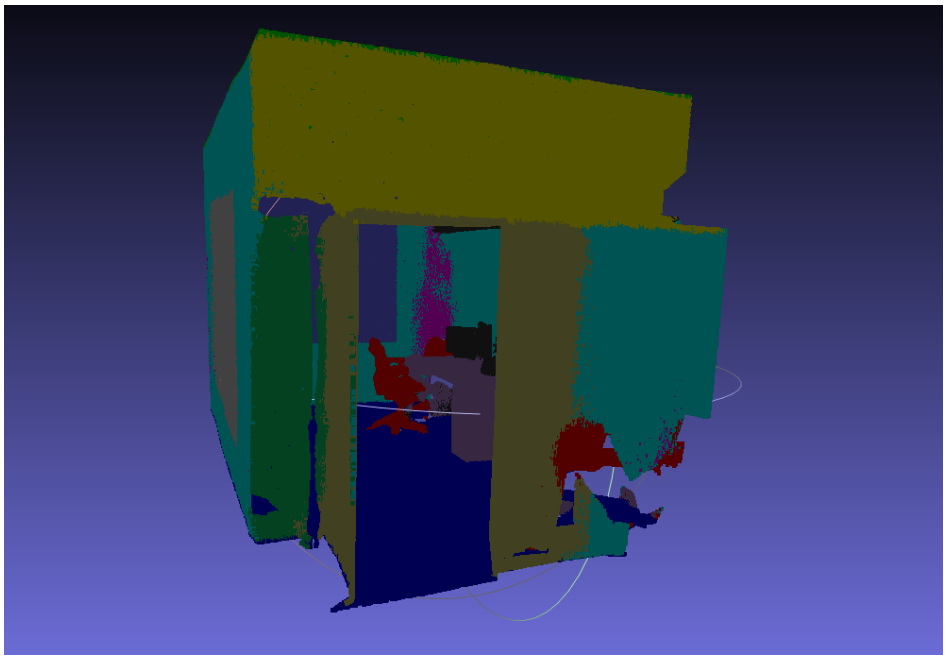
By utilizing a GPU, the training process was optimized for performance, ensuring that the PointNet model was trained efficiently for semantic segmentation tasks.

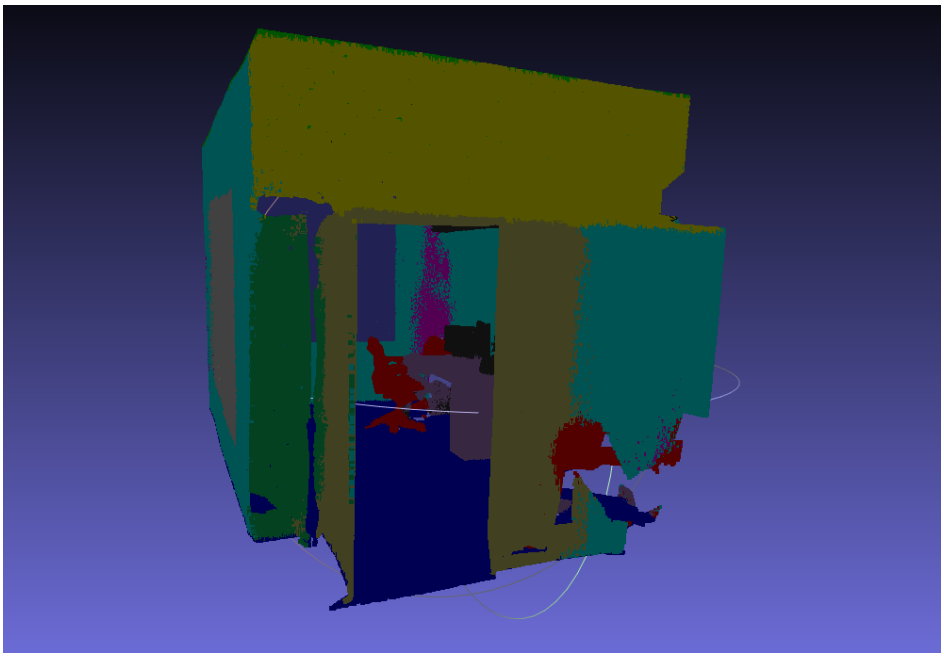
Testing the PointNet Model

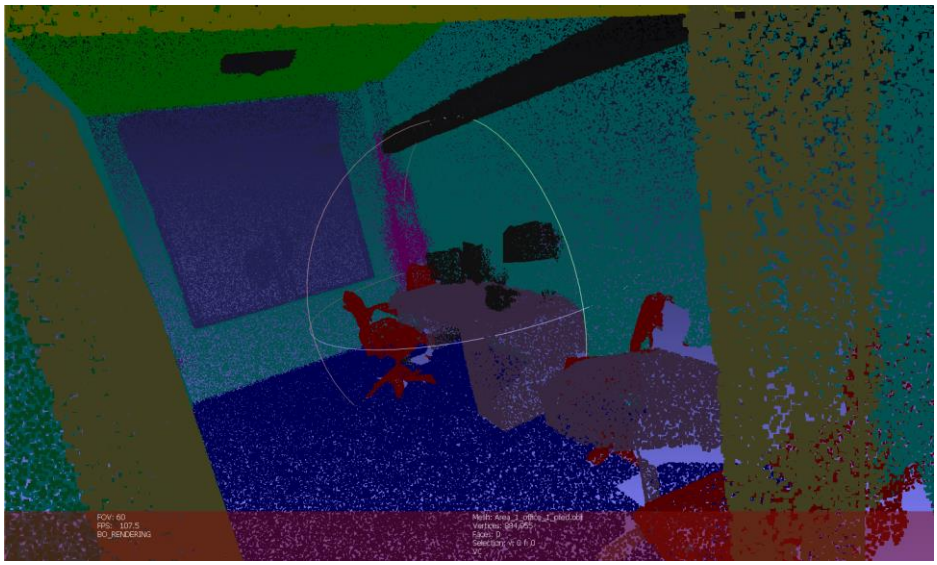
Running the Model on Test Data

After training the PointNet model, I proceeded to evaluate its performance on the test data using the test_semseg.py script. This script was executed with the previously specified log directory (--log_dir pointnet2_sem_seg), test area (--test_area 1) and visualization flag (--visual).

To display the results specifically for Area_1_office_1, I ran the model on this segment and the results demonstrated the model's effectiveness in segmenting and classifying this specific area. The Mean IoU values for Area_1_office_1 and other segments in the test area highlighted the model's high accuracy.







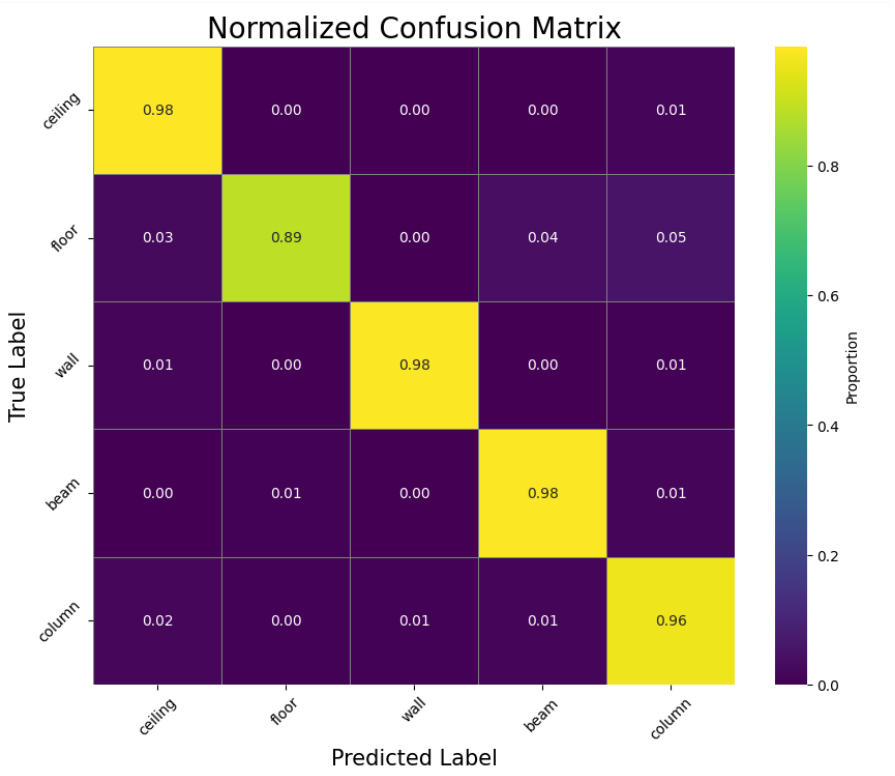
The results showed that the PointNet model was able to accurately segment and classify various regions in the test area, with Mean IoU values indicating high accuracy for most segments.

Extracting Labels and Computing Confusion Matrix

To evaluate the PointNet model's performance in detail, I used a confusion matrix to compare the predicted labels with the ground truth labels for `Area_1_office_1`. I started by defining a function to read the labels from `.obj` files. The labels were assumed to be the last element on lines starting with 'v' in the `.obj` files.

I specified the paths to the prediction and ground truth `.obj` files and read the labels using the defined function. The class names for the dataset were also defined to match the labels with meaningful names.

Using the `confusion_matrix` function from `sklearn.metrics`, I computed the confusion matrix. This matrix shows the counts of true positive, false positive and false negative predictions for each class. To make the matrix more interpretable, I normalized it by dividing each element by the sum of its row.



Test Pipeline Features on Kitti

I implemented a UI with a function to load KITTI .bin files, which contain point cloud data. The function reads the binary file and reshapes the data into a point cloud format using Open3D.

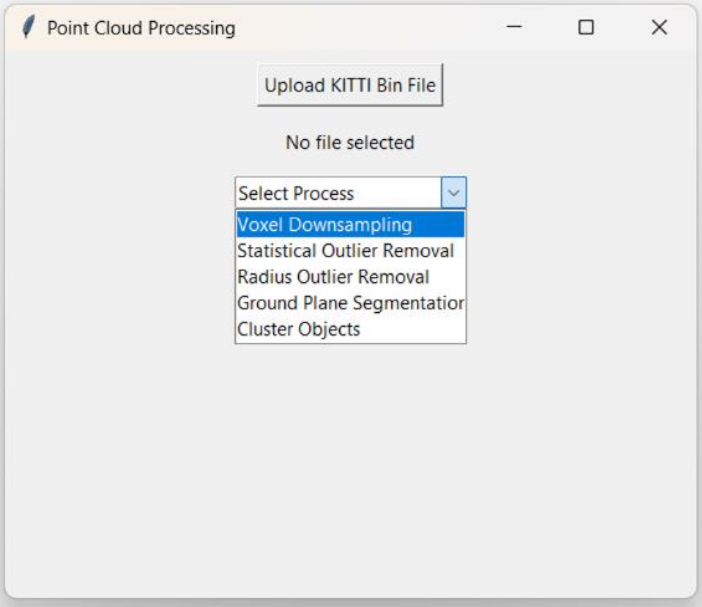
A Tkinter-based GUI was created to facilitate user interaction. The GUI allows users to upload a KITTI point cloud file and select a processing method from a dropdown menu. The available processes include voxel downsampling, statistical outlier removal, radius outlier removal, ground plane segmentation and object clustering.

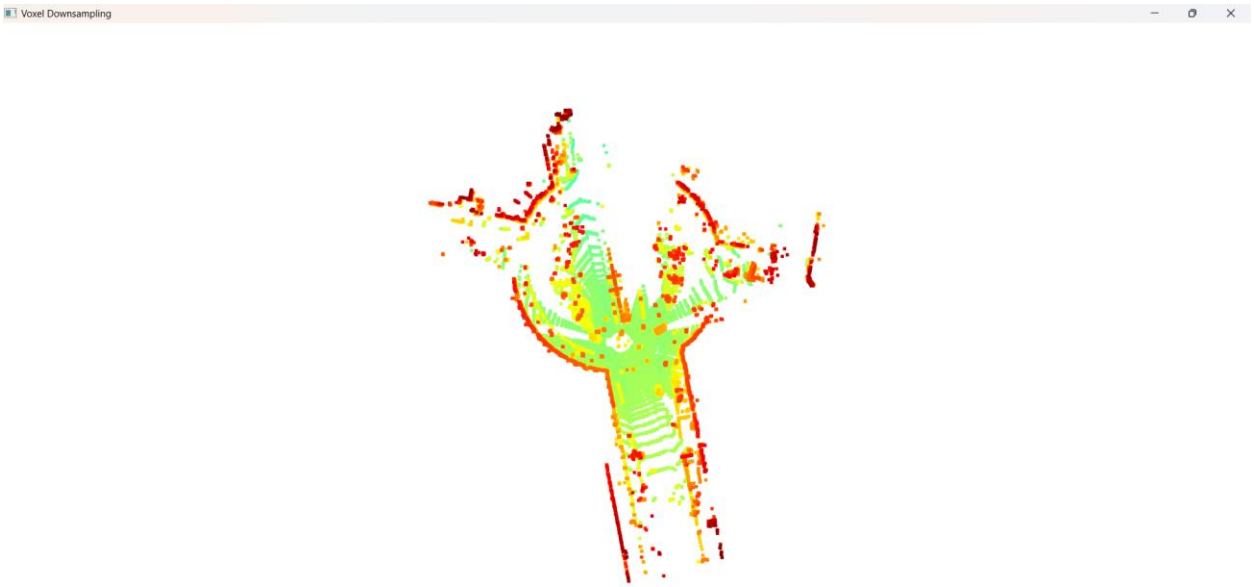
Workflow

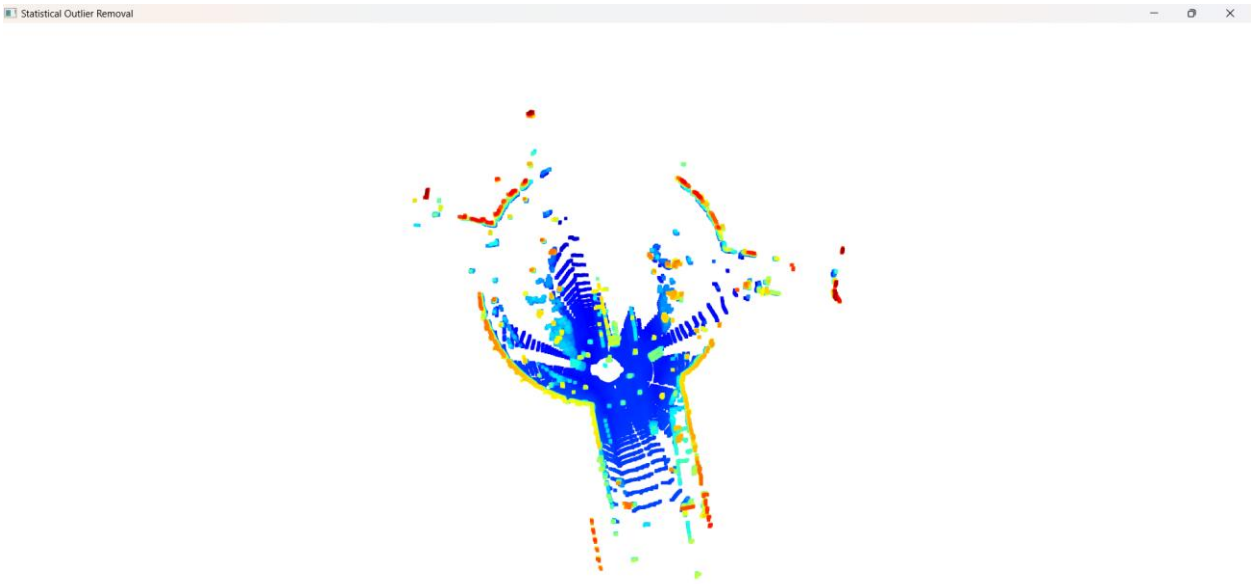
File Upload: The user uploads a KITTI .bin file using the "Upload KITTI Bin File" button.

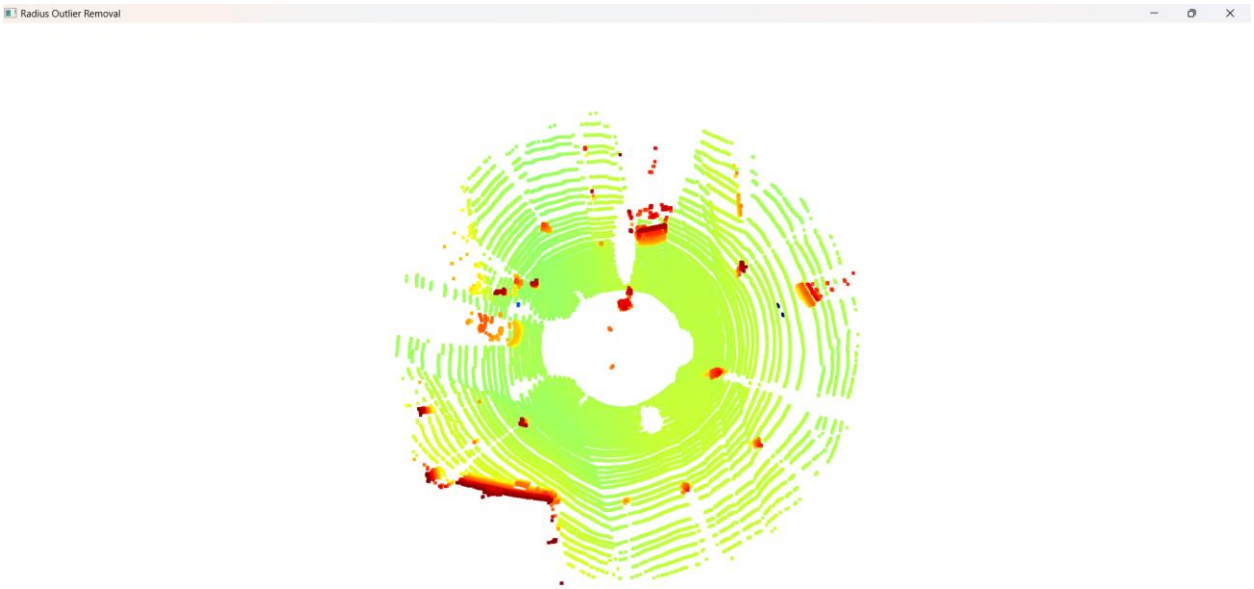
Process Selection: The user selects a processing method from the dropdown menu.

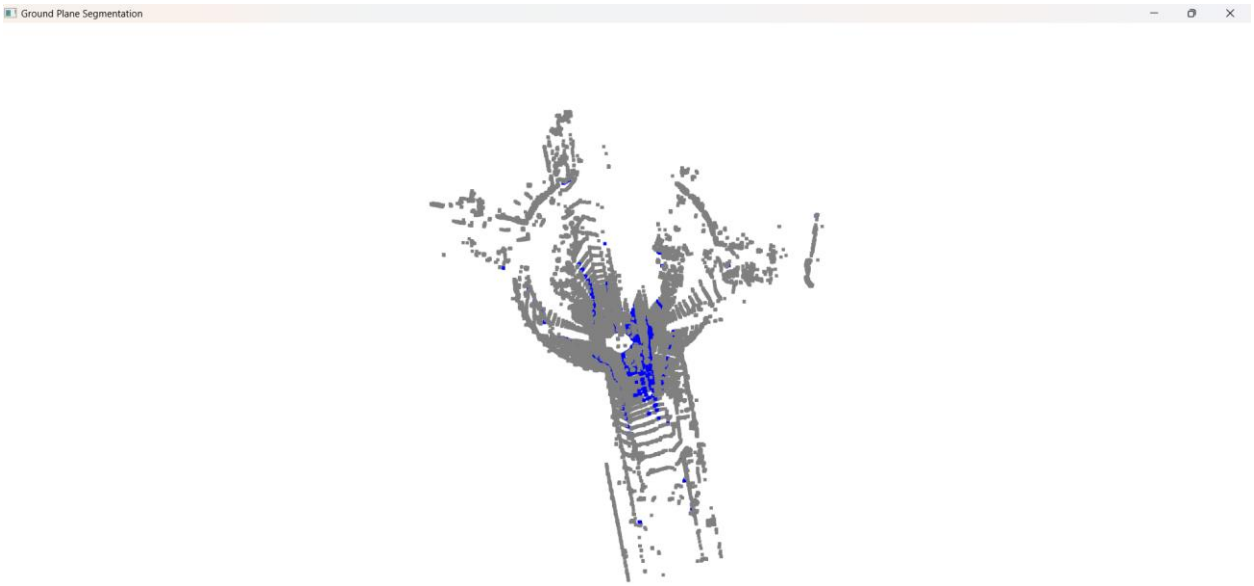
Run Process: The user clicks the "Run" button to execute the selected process on the uploaded point cloud. The results are then visualized.

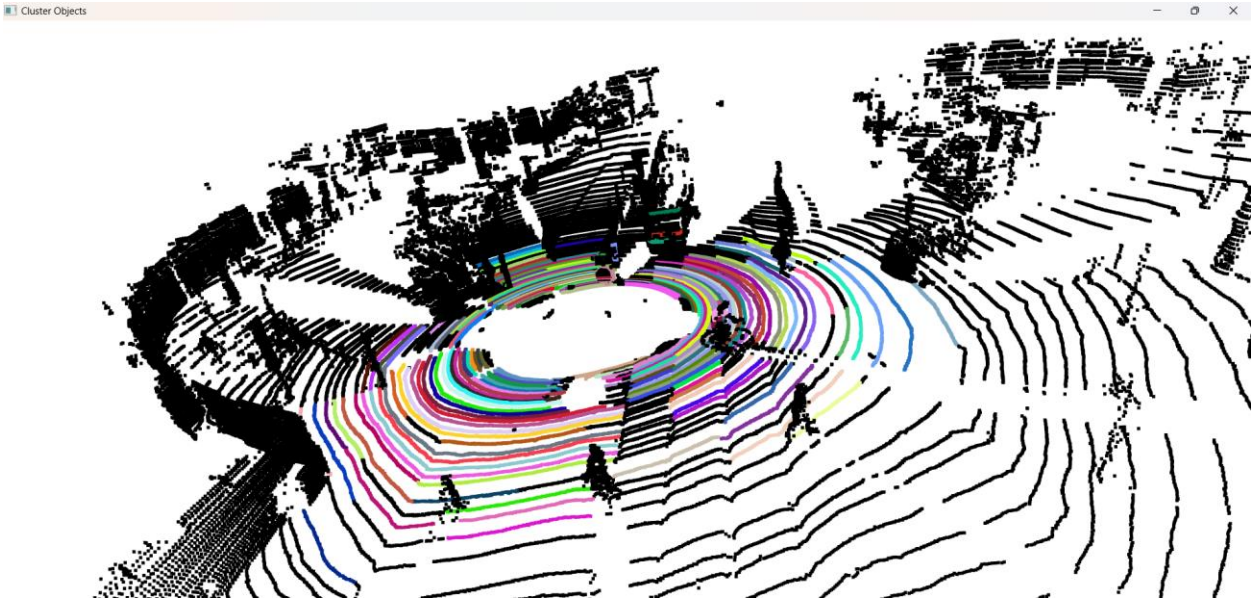





Statistical Outlier Removal


Radius Outlier Removal





Conclusion

PointNet, in particular, demonstrated significant advantages over traditional manual clustering techniques. While manual clustering methods, such as region growing and density-based clustering, rely heavily on predefined parameters and can struggle with complex, high-dimensional data, PointNet leverages deep learning to learn intricate patterns directly from the data. This capability allows PointNet to handle diverse and complex scenes more effectively, resulting in higher accuracy and robustness in segmentation tasks. PointNet's ability to generalize across different types of 3D structures reduces the need for manual parameter tuning and enables more automated and scalable solutions.

Overall, this project highlights the potential of combining traditional geometric approaches with modern deep learning techniques for 3D segmentation, paving the way for more advanced and automated 3D scene understanding applications. Future work could focus on further improving the segmentation accuracy and extending the pipeline to handle more complex and diverse 3D environments.