Summer Internship Project Report

3D Image Reconstruction using Open3D and RealSense Camera

Submitted by:

Kommareddy Diana Mary
Bachelors of Technology in Artificial Intelligence & Data Science
K L University

Under the guidance of:

Dr. Amit Shukla
Chairperson of CAIR

Center for AI and Robotics,

INDIAN INSTITUTE OF TECHNOLOGY MANDI

Summer Internship 2024

TABLE OF CONTENTS

1. INTRODUCTION

This report provides an in-depth examination of the processes and applications involved in 3D image reconstruction using Open3D and the Intel RealSense Depth Camera D455. The aim is to generate point clouds and 3D models from RGB-D images, capitalizing on the powerful data processing and visualization capabilities offered by Open3D.

1.1 BACKGROUND AND MOTIVATION

3D image reconstruction has become an essential technology in various fields such as robotics, augmented reality, medical imaging, and cultural heritage preservation. The ability to create accurate 3D models from real-world objects opens up new possibilities for analysis, interaction, and representation. The Intel RealSense Depth Camera D455, with its advanced depth sensing capabilities, provides an excellent tool for capturing high-quality RGB-D images. Combined with Open3D, an open-source library designed for 3D data processing, this technology enables efficient and effective reconstruction of 3D scenes.

2. OPEN3D OVERVIEW

Open3D is a versatile and powerful open-source library tailored for 3D data processing, making it an invaluable tool in various domains such as computer vision, robotics, and 3D reconstruction. It provides comprehensive support for a wide range of 3D data formats including point clouds, meshes, RGB-D images, and more. The library is designed to be user-friendly, promoting ease of use without compromising on its extensive functionality, which appeals to both beginners and advanced users.

3. RGB-D IMAGES

An RGB-D image is a powerful data format that combines an RGB (red, green, blue) image with a corresponding depth map. This combination provides both color and depth information, enabling more comprehensive analysis and processing of 3D environments and objects.

3.1 Components of an RGB-D Image

- **RGB Image**: The RGB component of the image captures the color information using three channels: red, green, and blue. This allows for the representation of a wide range of colors, similar to what the human eye perceives.
- **Depth Map**: The depth component is a single-channel grayscale image where each pixel's intensity represents the distance from the camera to the corresponding point in the scene. This depth information is crucial for understanding the 3D structure of the scene.

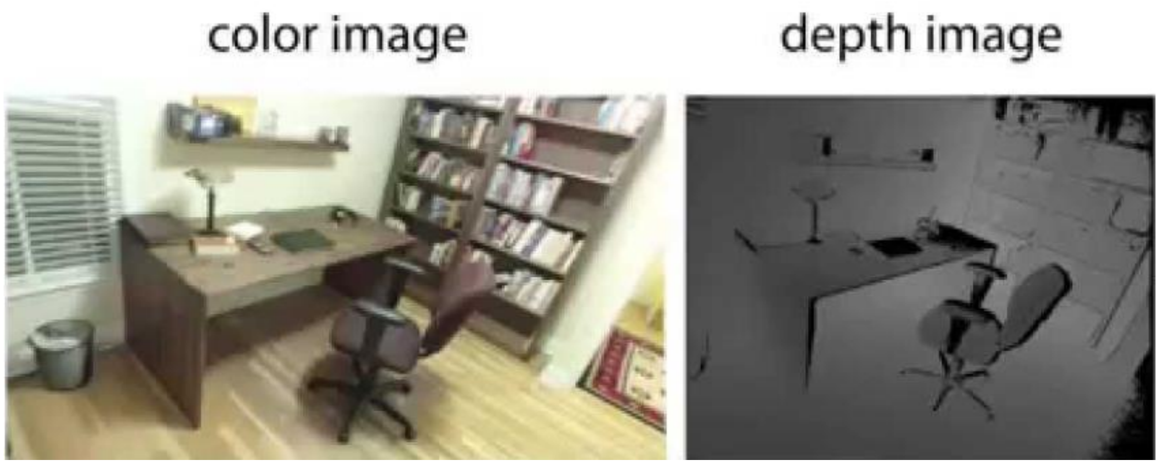


Fig1: The RGB-D image example

3.2 Depth Map Details

In the depth map, the pixel values indicate the distance of points from the camera:

- **Darker Pixels**: These represent points that are farther away from the camera. In a typical 8-bit depth map, darker pixels have values closer to 0. For instance, a pixel value of 0 indicates the maximum depth or farthest point that the camera can detect within its range.
- **Lighter Pixels**: These represent points that are nearer to the camera. In an 8-bit depth map, lighter pixels have values closer to 255. A pixel value of 255 indicates the minimum depth or the nearest point detectable by the camera.

This gradient from dark to light provides a visual representation of depth, where a smooth transition from dark to light indicates a gradual change in distance, and abrupt changes signify edges or significant depth variations in the scene.

4. POINT CLOUDS

A point cloud is a collection of data points defined in a three-dimensional coordinate system. Each point in the cloud has a set of X, Y, and Z coordinates that denote its position in 3D space. Point clouds are often generated using 3D scanning technologies such as LIDAR, stereo vision, and depth-sensing cameras like the Intel RealSense Depth Camera D455. When these points are densely packed, they can create highly detailed 3D representations of environments or objects.

4.1 Components of a Point Cloud

- **Coordinates**: Each point in the point cloud has three coordinates (X, Y, Z) that define its location in 3D space.
- **Color Information (Optional)**: In some cases, points may also contain color information (R, G, B) from the corresponding RGB image, enhancing the visual representation of the 3D model.
- **Intensity Values (Optional)**: For certain applications, points may include intensity values indicating the strength of the returned signal in scanning technologies like LIDAR.

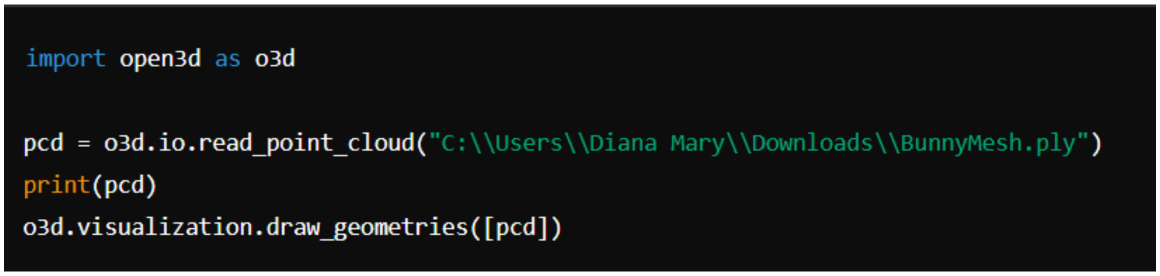
4.2 Uses of Point Clouds in Robotics

1. **3D Mapping**: Detailed 3D maps for navigation.
2. **Object Detection and Recognition**: Accurate identification and location of objects.
3. **Obstacle Avoidance**: Detecting and avoiding obstacles.
4. **Path Planning**: Planning efficient paths in complex environments.
5. **Manipulation and Grasping**: Accurate object manipulation.
6. **Environmental Interaction**: Detailed 3D understanding for effective interaction.

While working with the Intel RealSense Depth Camera D455, I've found it particularly advantageous for applications needing precise, real-time 3D data. It excels in capturing detailed depth information with high accuracy, offering ease of use and cost-effectiveness compared to technologies like LIDAR. These qualities make it well-suited for robotics, augmented reality, and industrial automation applications.

5. BASIC CODE AND OUTPUT

Code:



```
import open3d as o3d

pcd = o3d.io.read_point_cloud("C:\\Users\\Diana Mary\\Downloads\\BunnyMesh.ply")
print(pcd)
o3d.visualization.draw_geometries([pcd])
```

Output: Point Cloud with 35,947 points.

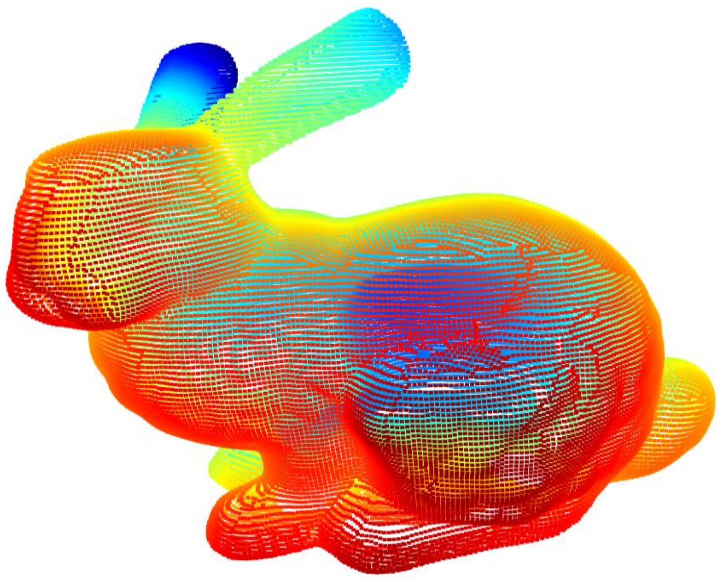


Fig 2: The point cloud Output

The following code take a .ply file as input and provides the visualization of the output point cloud along with number of point present in it. The .ply file format, short for Polygon File Format or Stanford Triangle Format, is a popular file format used for storing 3D models and point clouds. It supports a variety of data elements, including vertices (points), faces (polygons), edges, and other properties like color and normals.

6. Down sampled Point Cloud

A down sampled point cloud is a reduced-density version of the original point cloud, achieved to lower computational complexity and memory requirements while preserving essential features. This process typically involves using a voxel grid approach, where points within each voxel are replaced by a single representative point.

6.1 Voxel Grid Down sampling

- **Voxel Grid**: A voxel grid divides the 3D space into small cubic volumes called voxels. Each voxel encompasses a certain volume of space and acts as a unit for downsampling.
- **Process**: To downsample a point cloud using a voxel grid:
    1. **Define Voxel Size**: Specify the size of the voxel grid, determining the granularity of downsampling. A smaller voxel size retains more detail but requires more computational resources.
    2. **Downsampling Operation**: For each voxel, replace all points within the voxel with a single representative point. Common approaches include taking the centroid (average position) of the points or choosing a point closest to the voxel's center.
    3. **Output**: The result is a new point cloud with reduced density, where each voxel now contains a single representative point instead of multiple points.

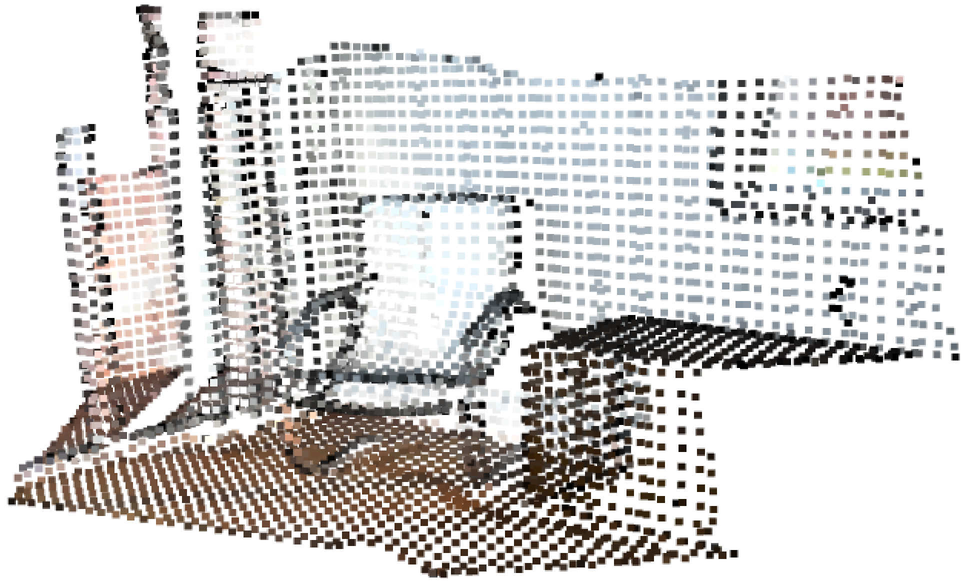


Fig3: The following point cloud is down sample by the voxel value of 0.05.

6.2 Advantages of Down sampling

- **Computational Efficiency**: Down sampling reduces the number of points, making subsequent processing tasks such as registration, segmentation, and feature extraction faster.
- **Memory Optimization**: By decreasing the number of points, down sampling reduces the memory footprint required to store and process the point cloud data.
- **Noise Reduction**: Down sampling can mitigate noise and small irregularities in the point cloud, providing a smoother and more manageable dataset for analysis.

6.3 Code sample

```
def preprocess_point_cloud(pcd, voxel_size):
    print(":: Downsample with a voxel size of %.3f." % 
voxel_size)
    pcd_down = pcd.voxel_down_sample(voxel_size)

    radius_normal = voxel_size * 2
    print(":: Estimate normal with search radius %.3f." 
% radius_normal)
    pcd_down.estimate_normals(
       o3d.geometry.KDTreeSearchParamHybrid(radius=radiu
s_normal, max_nn=30))

    radius_feature = voxel_size * 5
    print(":: Compute FPFH feature with search radius 
%.3f." % radius_feature)
    pcd_fpfh = 
o3d.pipelines.registration.compute_fpfh_feature(
       pcd_down,
       o3d.geometry.KDTreeSearchParamHybrid(radius=radiu
s_feature, max_nn=100))
    return pcd_down, pcd_fpfh
```

Explanation:

The `preprocess_point_cloud` function processes a given point cloud (`pcd`) by initially downsampling it using a voxel grid approach with a specified `voxel_size`, reducing the density of points for computational efficiency. After downsampling, surface normals are estimated for the downsampled cloud (`pcd_down`) using a neighborhood search radius (`radius_normal`). This step provides information about the orientation and curvature of surfaces within the point cloud. Subsequently, Fast Point Feature Histograms (FPFH) features are computed for `pcd_down`, describing local geometric structures using a larger search radius (`radius_feature`). These features capture detailed information about point neighborhoods, crucial for tasks like point cloud registration and object recognition in applications such as robotics and 3D reconstruction.

7. Outliers and Inliers in Point Clouds

In the context of point clouds, understanding outliers and inliers is essential for accurate analysis and processing of 3D data.

7.1 Outliers

- **Definition**: Outliers are points within a point cloud that significantly deviate from the overall pattern or expected distribution.
- **Characteristics**:
  - **Unexpected**: They often represent errors, noise, or anomalies in the data capture process.
  - **Sparse**: Outliers are typically fewer in number compared to inliers, but their presence can impact data integrity and analysis.
  - **Impact**: They can distort statistical measures, affect surface reconstruction, and hinder accurate modeling or registration tasks.
- **Detection and Handling**:
  - **Statistical Methods**: Outliers can be identified using statistical techniques such as mean and standard deviation analysis, where points falling outside certain thresholds are flagged.
  - **Spatial Context**: Contextual information like local density or neighborhood characteristics can also help distinguish outliers.
  - **Filtering**: Techniques such as spatial filtering or clustering algorithms can mitigate the impact of outliers on subsequent data processing steps.

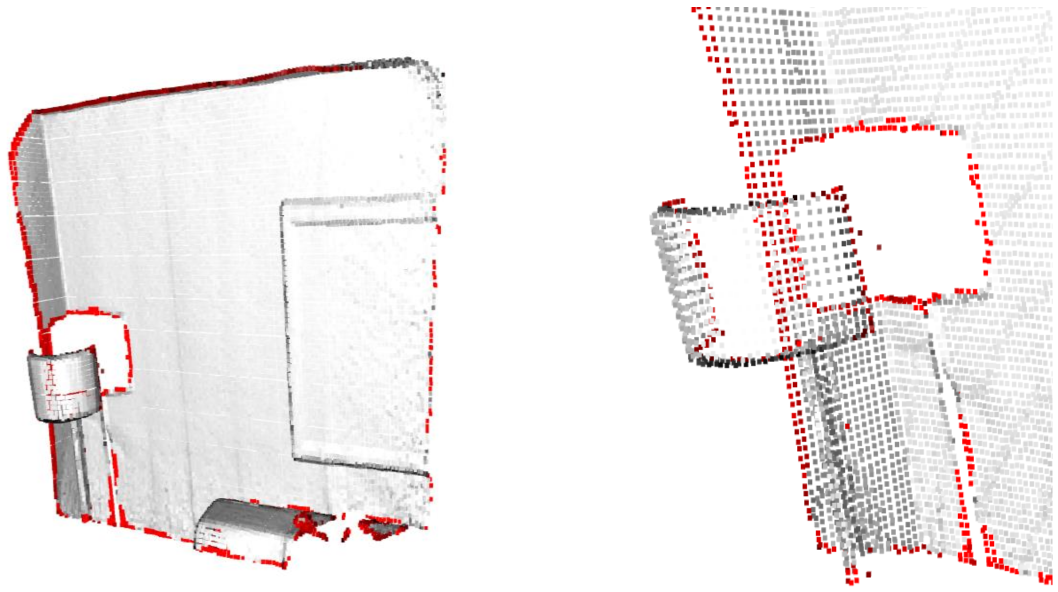


Fig 4: Showing outliers (red) and inliers (gray)

7.2 Inliers

- **Definition**: Inliers are points in a point cloud that conform to the expected distribution or shape of the main structure being represented.
- **Characteristics**:
    - **Representative**: They accurately reflect the underlying geometry or features of interest in the scene.
    - **Majority**: Inliers constitute the majority of points and form the core data set for analysis and modeling.
    - **Consistency**: They contribute to stable and reliable results in tasks like surface reconstruction, object detection, and localization.

8. POINT CLOUD FROM RGB AND DEPTH IMAGES

Point clouds can be formed from RGB and depth images of the same size.

Code:

```
import open3d as o3d
import numpy as np
import matplotlib.pyplot as plt

print("Read TUM dataset")
tum_rgbd = o3d.data.SampleSUNRGBDImage()
color_raw = o3d.io.read_image("C:\\Users\\Diana
mary\\Downloads\\SampleRGBDImageTUM\\TUM_color.png")
depth_raw = o3d.io.read_image("C:\\Users\\Diana
mary\\Downloads\\SampleRGBDImageTUM\\TUM_depth.png")
rgbd_image =
o3d.geometry.RGBDImage.create_from_tum_format(color_raw,
depth_raw)
print(rgbd_image)
```

```
plt.subplot(1, 2, 1)
plt.title('TUM grayscale image')
plt.imshow(rgbd_image.color)
plt.subplot(1, 2, 2)
plt.title('TUM depth image')
plt.imshow(rgbd_image.depth)
plt.show()

pcd = o3d.geometry.PointCloud.create_from_rgbd_image(
    rgbd_image,
    o3d.camera.PinholeCameraIntrinsic(
        o3d.camera.PinholeCameraIntrinsicParameters.Prim
eSenseDefault)
)
pcd.transform([[1, 0, 0, 0], [0, -1, 0, 0], [0, 0, -1,
0], [0, 0, 0, 1]])

vis = o3d.visualization.Visualizer()
vis.create_window()

vis.add_geometry(pcd)

view_ctl = vis.get_view_control()
view_ctl.set_zoom(0.35)

vis.run()
vis.destroy_window()
```

Output

```
RGBDImage of size
Color image : 640x480, with 1 channels.
Depth image : 640x480, with 1 channels.
Use numpy.asarray to access buffer data.
```

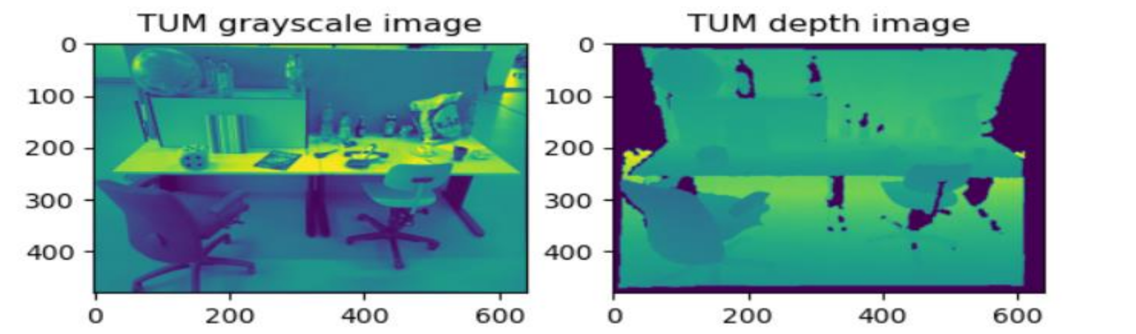


Fig 5: The grayscale and depth images of the input RGB-D images

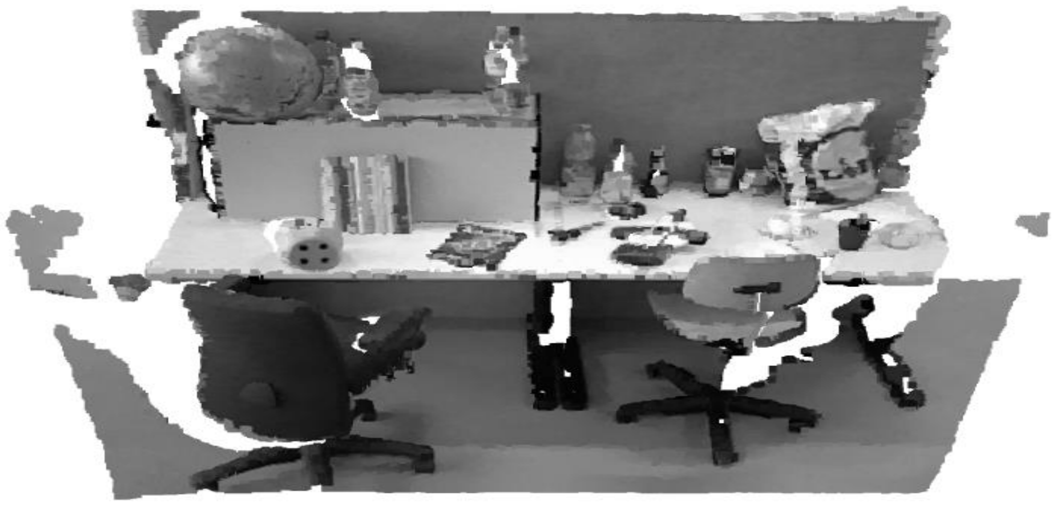


Fig 6: Point cloud of input data

We can use the same code to provide our own rgb and its corresponding depth image, to get a point cloud.

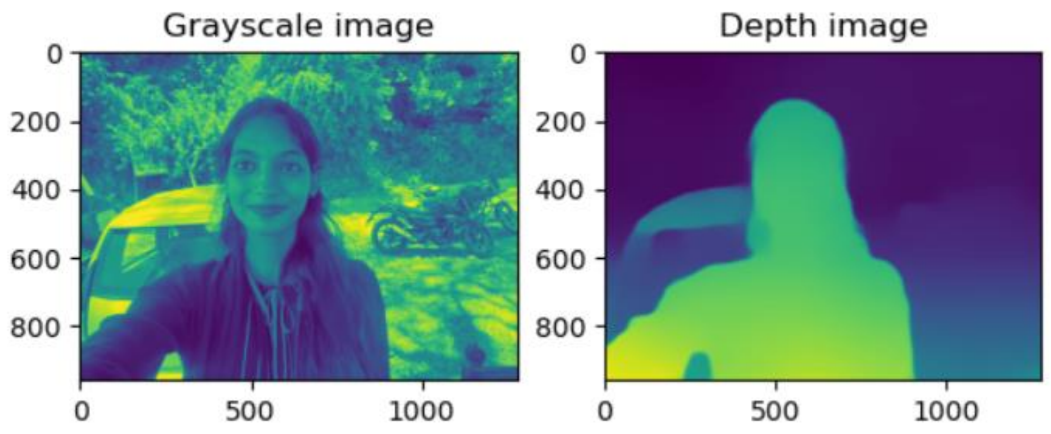


Fig 7: The grayscale and depth images

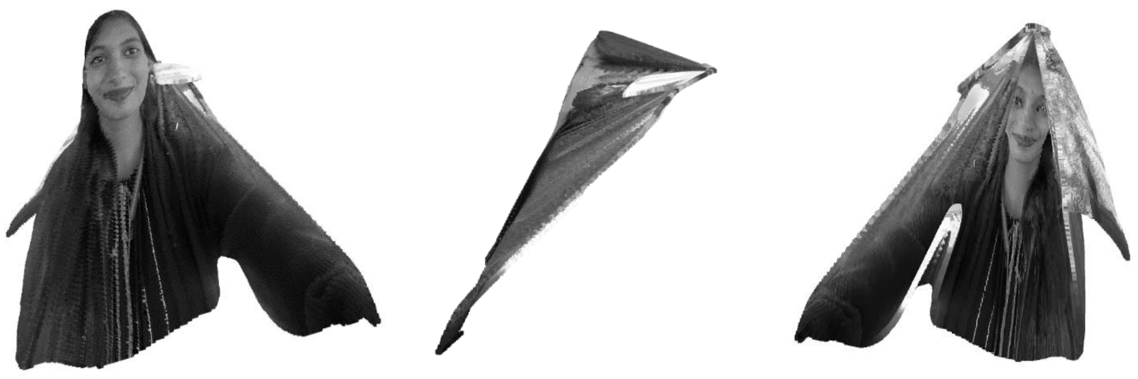


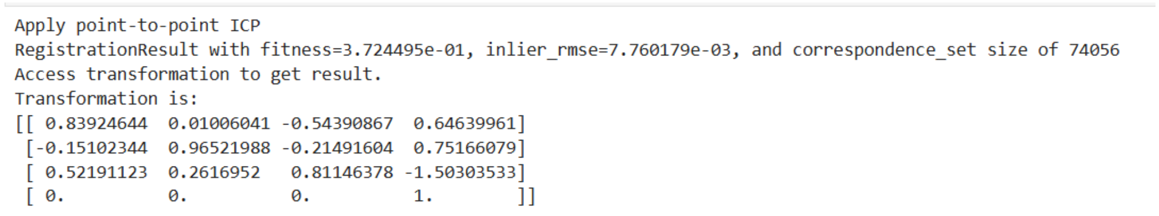
Fig 8: point cloud from front, side and back

9. ALIGNMENT

Aligning point clouds involves finding a transformation that minimizes the distance between corresponding points.

9.1 Point-to-Point Alignment

This method directly minimizes the Euclidean distance between corresponding points in the source and target point clouds. It iteratively refines the transformation by matching each point in the source point cloud to the nearest point in the target point cloud. The Iterative Closest Point (ICP) algorithm is a typical example of this method. It's simple and effective for cases where the point clouds are already roughly aligned, but it can be sensitive to noise and outliers.



```
Apply point-to-point ICP
RegistrationResult with fitness=3.724495e-01, inlier_rmse=7.760179e-03, and correspondence_set size of 74056
Access transformation to get result.
Transformation is:
[[ 0.83924644  0.01006041 -0.54390867  0.64639961]
 [-0.15102344  0.96521988 -0.21491604  0.75166079]
 [ 0.52191123  0.2616952   0.81146378 -1.50303533]
 [ 0.          0.          0.          1.        ]]
```

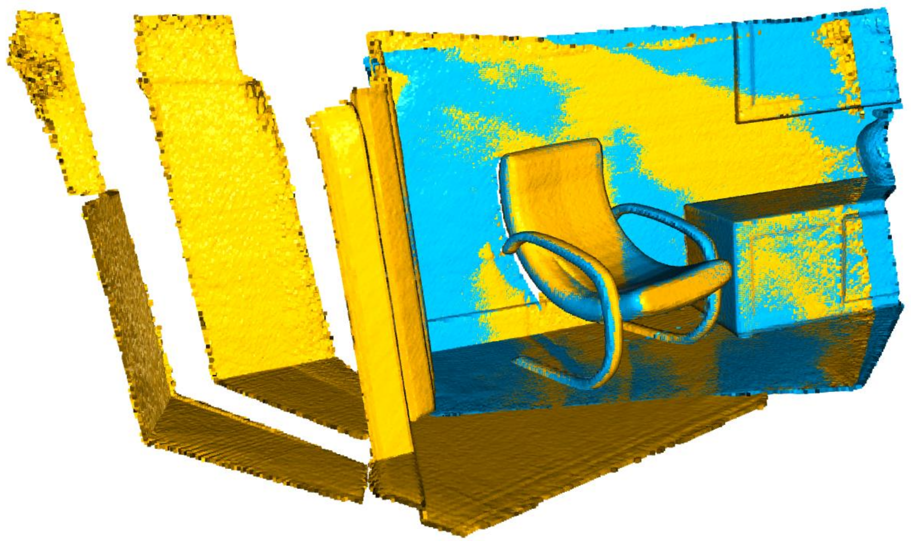
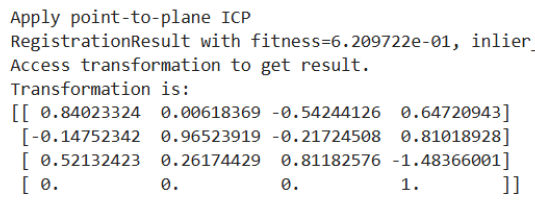


Fig 9: Point-To-Point alignment

9.2 Point-to-Plane Alignment

This faster and more accurately, especially when dealing with surfaces because it takes the local geometry into account. The point-to-plane ICP algorithm extends the basic ICP by incorporating surface normals to improve alignment accuracy. This method is particularly effective when the point clouds have a smooth surface and when dealing with partial overlaps. Method minimizes the distance between a point in the source point cloud and the plane defined by the nearest point and its normal in the target point cloud. It tends to converge



```
Apply point-to-plane ICP
RegistrationResult with fitness=6.209722e-01, inlier_rmse=6.581453e-03, and correspondence_set size of 123471
Access transformation to get result.
Transformation is:
[[ 0.84023324  0.00618369 -0.54244126  0.64720943]
 [-0.14752342  0.96523919 -0.21724508  0.81018928]
 [ 0.52132423  0.26174429  0.81182576 -1.48366001]
 [ 0.          0.          0.          1.        ]]
```

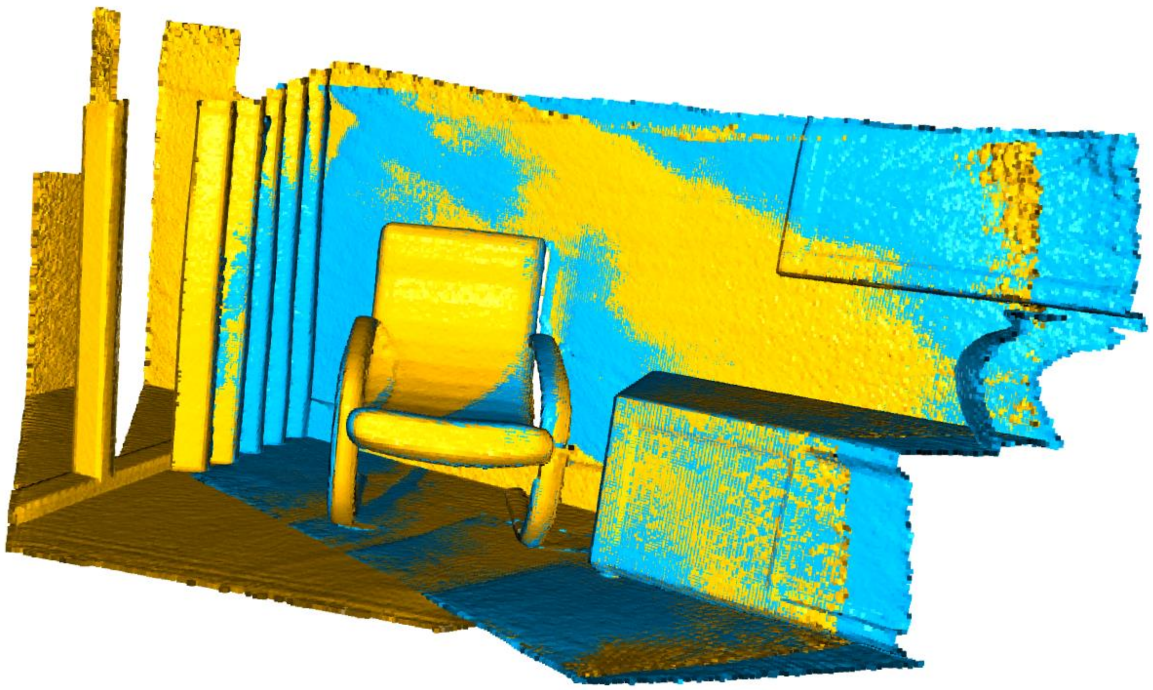


Fig 10: Point-To-Plane Alignment

"Corresponding point size" typically refers to the number or density of corresponding points between two or more-point clouds that are being aligned. We can see that in point to plane alignment, the corresponding point size is higher compare to point to point, which show that point to plane alignment is more efficient.

10. Triangle Mesh

A triangle mesh is a fundamental data structure used in computer graphics and computational geometry to represent the surface of a 3D object or scene. It consists of interconnected triangles that collectively define the geometry of the object. Here’s an expanded explanation of triangle meshes, their characteristics, and applications:

10. 1 Structure and Representation

- **Triangles**: The basic building blocks of a triangle mesh are triangles, which are composed of three vertices (points in 3D space) and their associated edges.
- **Vertex Connectivity**: Each vertex in a mesh is connected to form a network of triangles. This connectivity defines the surface topology and geometric features of the object.
- **Mesh Elements**: Apart from vertices and triangles, additional data such as vertex normals, texture coordinates, and vertex colors can be associated with each vertex or triangle to enhance visual appearance and realism.

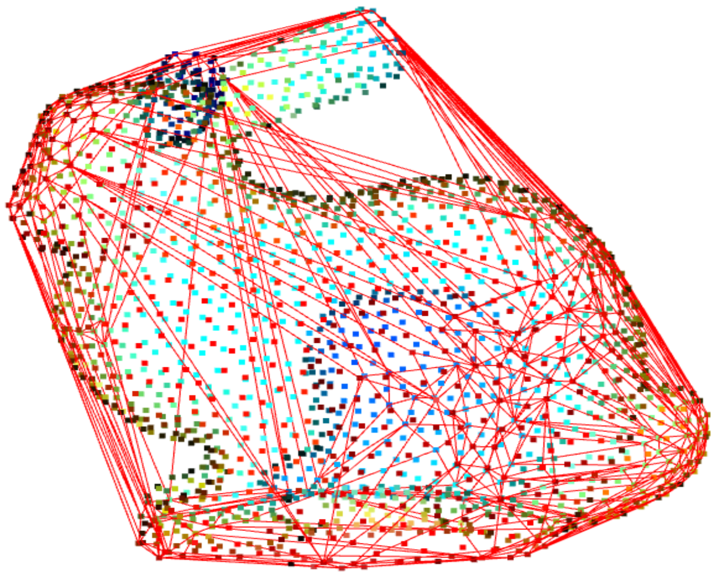


Fig 11: Triangle mesh of rabbit point cloud

11. Effect of Alpha Values on Alpha Shapes

The alpha value plays a crucial role in generating the alpha shape of a point cloud, influencing how the shape encapsulates and represents the overall structure of the point cloud. Here’s an expanded explanation of how alpha values affect alpha shapes:

11.1 Alpha Shape Overview

- **Definition**: An alpha shape is a geometric model that wraps around a set of points in 3D space, forming a boundary that connects neighboring points based on a specified alpha radius.
- **Radius Parameter**: The alpha radius (alpha) defines the maximum radius of the circumcircle around each edge of the Delaunay triangulation of the point cloud. It controls which edges are included in the alpha shape.

11.2 Effect of Alpha Values

- **Alpha Value Interpretation**: The alpha value (alpha) determines the size of the alpha shape's edges relative to the density and distribution of points in the point cloud.
- **Small Alpha Values**: When alpha is small, the alpha shape tends to include smaller circumcircles, resulting in a more detailed and intricate boundary that closely follows the contours of densely packed point clusters. This can capture fine details and irregularities in the point cloud but may also include noise or outliers.
- **Large Alpha Values**: Larger alpha values lead to larger circumcircles, which encompass more points and smooth out the boundary of the alpha shape. This results in a more generalized representation that emphasizes the overall structure and larger-scale features of the point cloud while potentially filtering out smaller-scale details and noise.

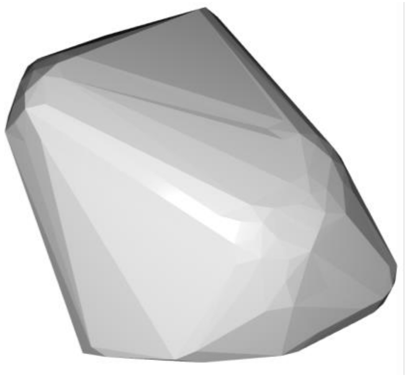
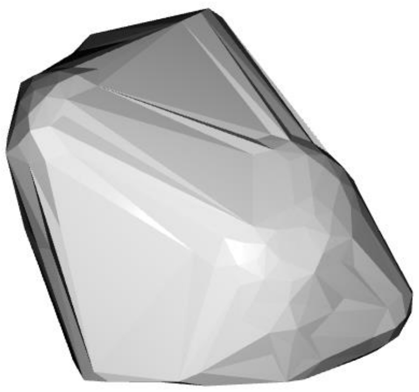
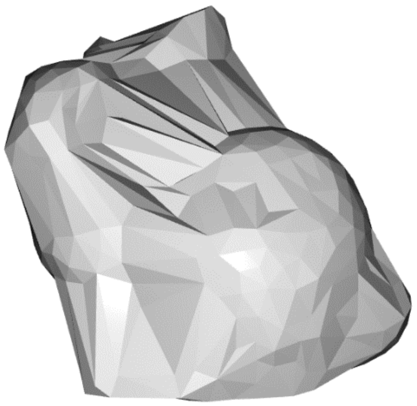
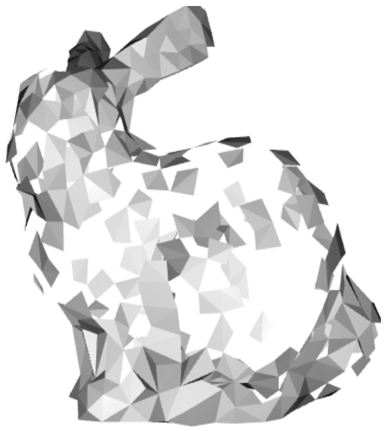


Fig 12: Alpha value =500


Alpha value = 136


alpha value = 37


Aplha value = 10

12. RGB-D Odometry

RGB-D Odometry is a technique used to estimate the motion of a camera or sensor platform by analyzing consecutive frames from an RGB-D (color and depth) camera. This method is integral to applications such as visual SLAM (Simultaneous Localization and Mapping), robotics navigation, and augmented/virtual reality. Here’s an expanded explanation of RGB-D odometry, its principles, and its applications:

12.1 Principles of RGB-D Odometry

- **Input Data**: RGB-D odometry utilizes data from RGB-D cameras, which provide both color (RGB) and depth information (D). Depth information is crucial as it enables precise estimation of scene geometry and depth changes over time.
- **Motion Estimation**: The primary goal of RGB-D odometry is to compute the camera's relative motion between consecutive frames. This is achieved by tracking visual features in the RGB images and matching corresponding depth points to compute relative depth changes.
- **Feature Tracking**: Keyframe-based methods track features such as keypoints or edges in RGB images across frames. These features provide distinctive landmarks that help in estimating camera motion accurately.
- **Depth Integration**: Depth information from the RGB-D camera is integrated to refine the motion estimation. Changes in depth between frames allow for robust estimation of camera translation and rotation.

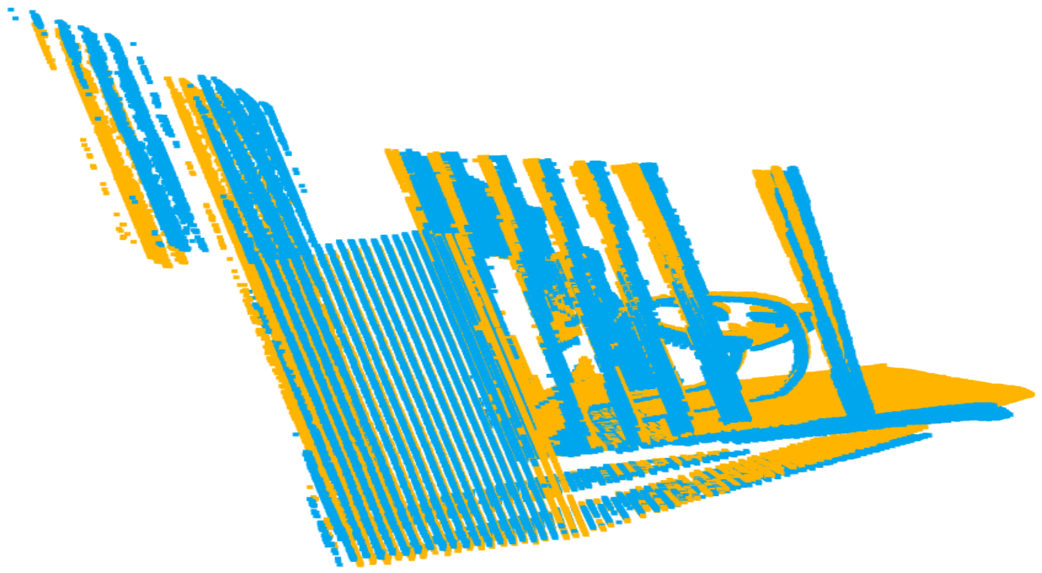


Fig 13: Veritucal Odomatry

13. Intel RealSense Depth Camera D455

The Intel RealSense Depth Camera D455 is a stereo-based depth camera designed for applications requiring precise depth sensing and 3D imaging.
Key Features:

- Stereo Depth Technology, Longer Range, Wide Field of View, High Accuracy, RGB Camera, Synchronization, IMU (Inertial Measurement Unit), USB Interface SDK Support, Depth Technology: Active IR Stereo
- Depth Resolution: Up to 1280 x 720 at 30 fps.
- RGB Resolution: Up to 1920 x 1080 at 30 fps.
- Depth Accuracy: Within 2% at a range of 4 meters.
- Minimum Depth Distance: 0.4 meters.
- Maximum Range: Up to 10 meters with reduced accuracy.
- Dimensions: 124 mm x 29 mm x 20 mm.
- Weight: Approximately 70 grams.
- Power Consumption: 1.5 W (typical).
- Operating Temperature: 0°C to 35°C.

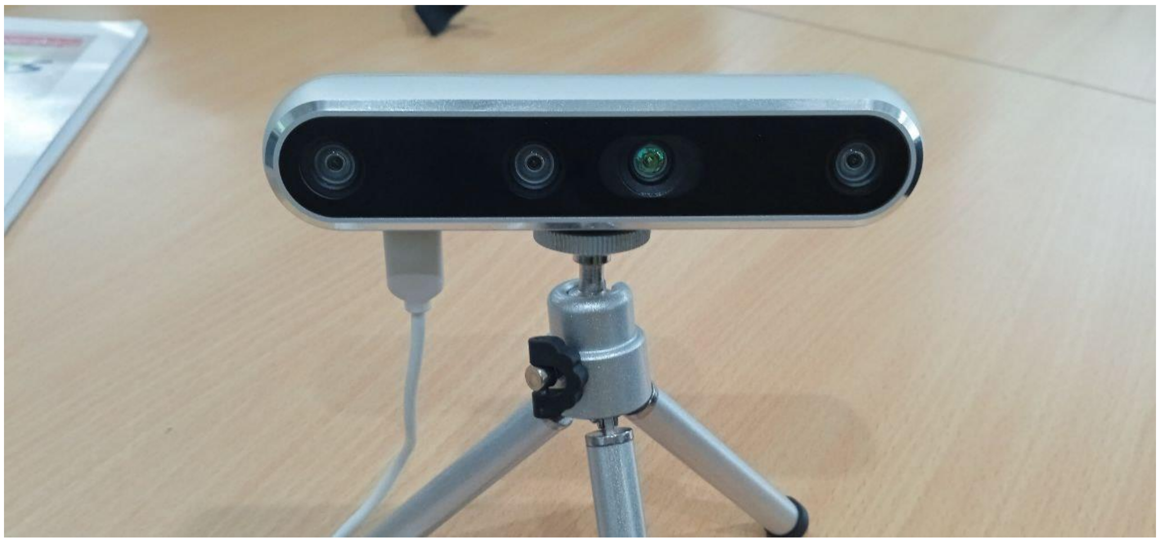


Fig 14: Intel Realsense Depth camera D455

14. Image Stitching

Image stitching is a computational technique used to combine multiple overlapping images into a single seamless panoramic image or 3D model. This process involves several key steps, including feature detection, matching, transformation estimation, and blending.

14.1 Applications of Image Stitching

- **Panoramic Photography**: Creating high-resolution panoramic images from multiple overlapping photos.
- **Virtual Tours**: Building immersive virtual tours of real-world environments or indoor spaces.
- **Medical Imaging**: Stitching together multiple medical images (e.g., MRI or CT scans) for comprehensive analysis.
- **Document Scanning**: Combining multiple scans of large documents or drawings into a single digital file.

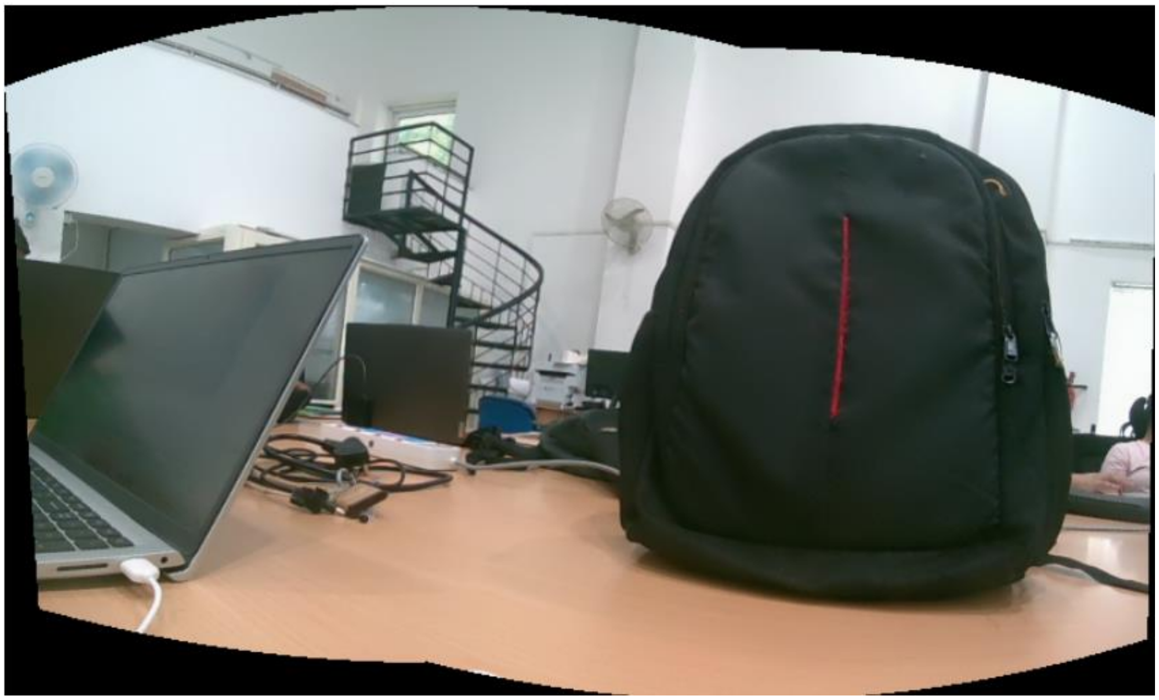


Fig 15: Stitched image

15. Point Cloud from Stitched RGB-D Images

Creating a point cloud from stitched RGB-D images involves aligning and combining multiple pairs of RGB and depth images to generate a comprehensive 3D representation of a scene. This process includes several steps, such as image alignment, depth map processing, and point cloud generation.

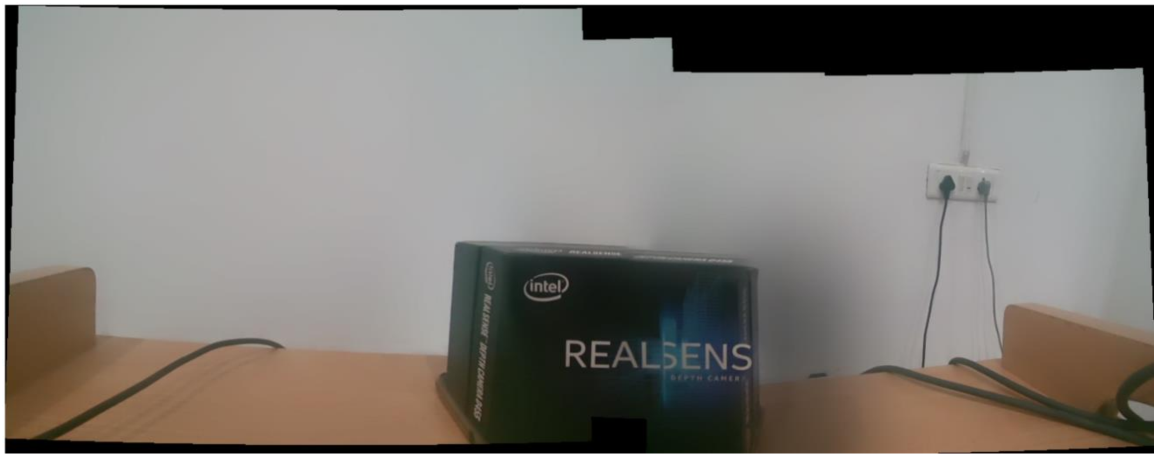


Fig 16.1 : Stitched image

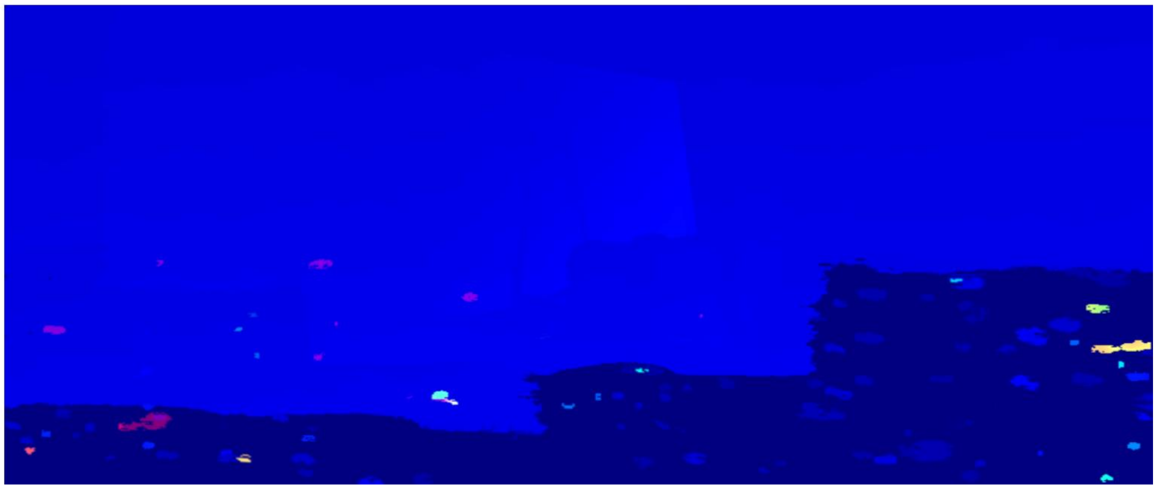


Fig 16.2 : Depth image of stitched images

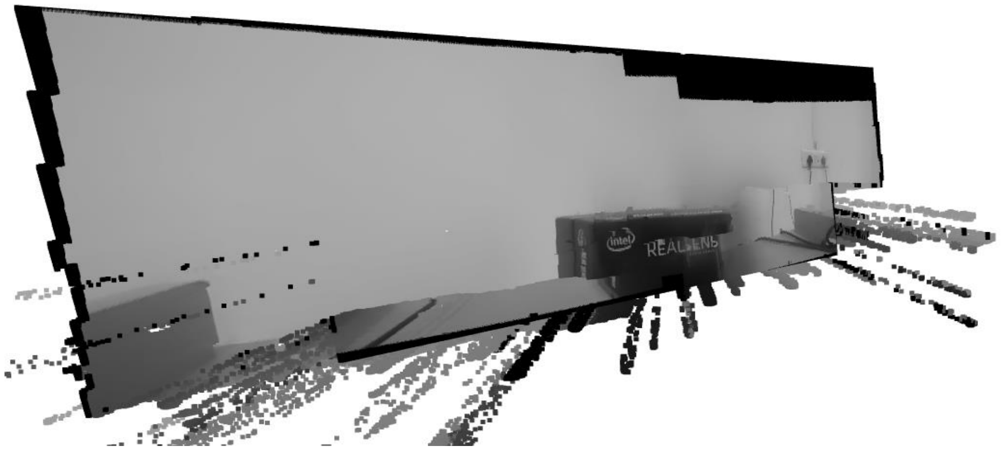


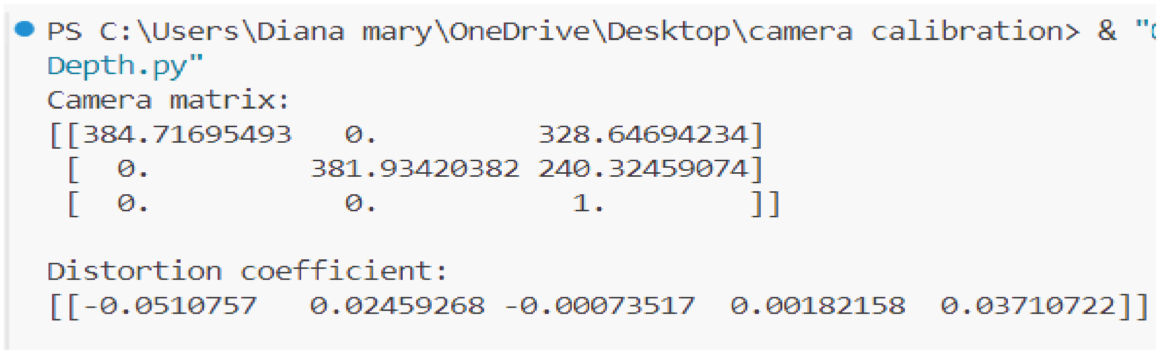
Fig 16.3: Point cloud of stitched

16. CAMERA CALIBRATION

Ensures accurate depth measurements and alignment between depth and RGB images by determining the intrinsic and extrinsic parameters of the camera.

16.1 Key Concepts

- **Intrinsic Parameters**: Define the internal characteristics of the camera, such as focal length and optical center.
- **Extrinsic Parameters**: Define the position and orientation of the camera in the world coordinate system.
- **Depth-to-Color Alignment**: Ensures accurate correspondence between depth information and RGB images.



```
PS C:\Users\Diana mary\OneDrive\Desktop\camera calibration> & "
Depth.py"
Camera matrix:
[[384.71695493   0.         328.64694234]
 [  0.         381.93420382 240.32459074]
 [  0.           0.           1.        ]]

Distortion coefficient:
[[-0.0510757   0.02459268 -0.00073517  0.00182158  0.03710722]]
```

Fig 17: Camera calibration values

17. Highlighting Specific Features in Point clouds

Point clouds, which represent 3D data collected from environments or objects, can be manipulated to highlight specific features, such as text on a box or distinctive shapes and structures. Highlighting specific features within point clouds can be crucial for various applications, including object recognition, inspection, and analysis.

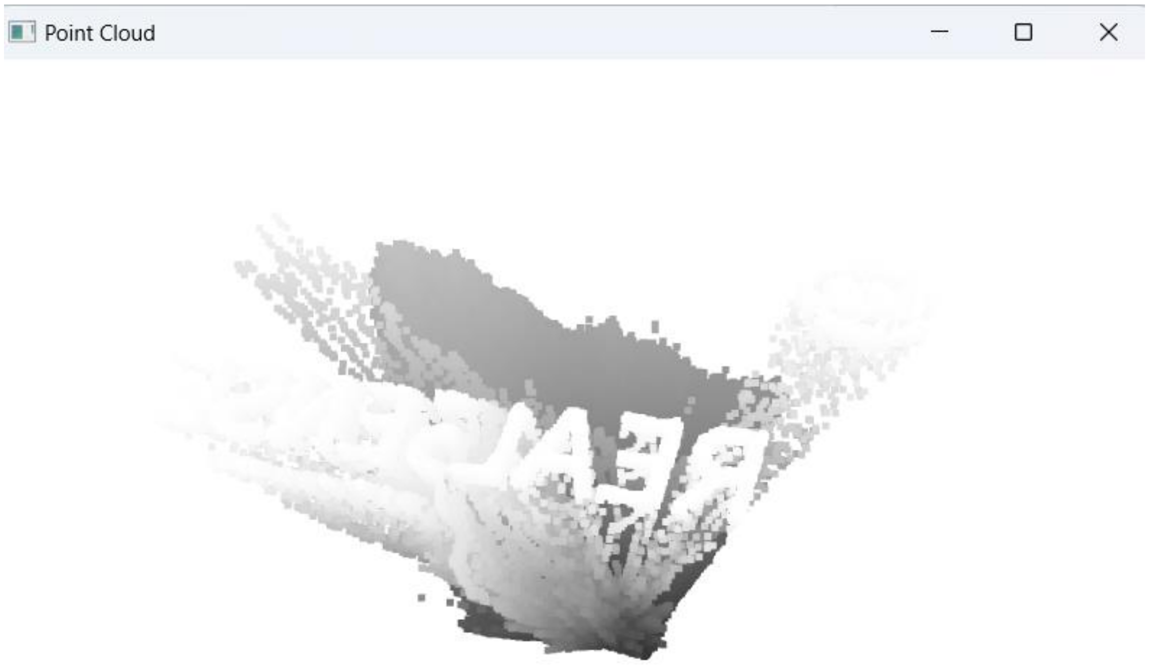


Fig 18: The point cloud highlight the text on box

The text is white, while the box and background are black, resulting in a point cloud where the name is prominently highlighted.

18. 3D Point Cloud Creation

3D point cloud creation involves combining multiple point clouds captured from different angles to form a single, comprehensive point cloud. This integrated point cloud accurately represents the complete surface geometry of an object or scene, providing a detailed and precise 3D model.

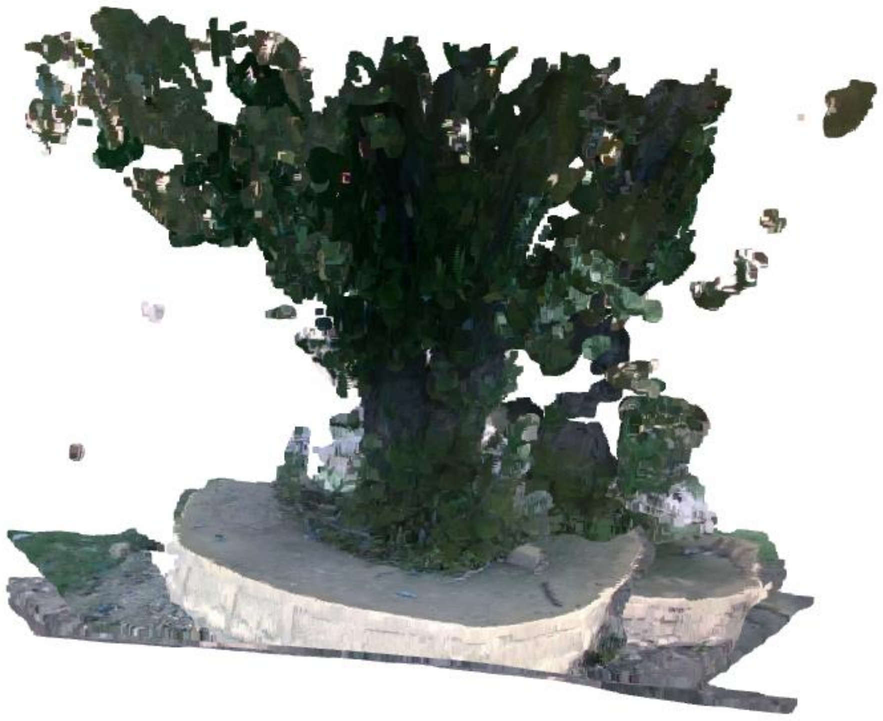


Fig 19: Re-constructed 3D image using combained point clouds

19. Conclusion

This report presents a comprehensive overview of 3D image reconstruction using Open3D and the Intel RealSense Depth Camera D455, highlighting the key processes, applications, and code examples for generating and processing point clouds and 3D models. The techniques discussed herein are crucial for advancements in computer vision, robotics, and 3D data processing.

20. References

Open 3D Documentation: Introduction - Open3D 0.18.0 documentation
Practice: Point Cloud Processing with Open3D: A Beginner's Guide to Getting Started (youtube.com)

21. Appendix

Python code of reconstruction of the 3D image, for output fig 19.

```
import open3d as o3d
import numpy as np
import cv2

point_clouds = []

# Loop to read and process each pair of depth and color images
for i in range(7):
    # Read depth image
    depth_image = cv2.imread(f'depth_image_{i}.png', cv2.IMREAD_UNCHANGED)
    # Read color image
    color_image = cv2.imread(f'color_image_{i}.png')

    # Ensure images are read correctly
    if depth_image is None or color_image is None:
        print(f"Error reading images for index {i}")
        continue

    # Convert images to Open3D format
    depth_o3d = o3d.geometry.Image(depth_image)
    color_o3d = o3d.geometry.Image(cv2.cvtColor(color_image,
cv2.COLOR_BGR2RGB))

    # Camera intrinsic parameters (adjust based on your camera's specs)
    intrinsic = o3d.camera.PinholeCameraIntrinsic(
        o3d.camera.PinholeCameraIntrinsicParameters.PrimeSenseDefault)

    # Create RGBD image from color and depth images
    rgbd_image = o3d.geometry.RGBDImage.create_from_color_and_depth(
        color_o3d, depth_o3d, convert_rgb_to_intensity=False)

    # Generate point cloud from RGBD image
    pcd = o3d.geometry.PointCloud.create_from_rgbd_image(rgbd_image,
intrinsic)

    # Append point cloud to list
    point_clouds.append(pcd)

# Visualize the captured point clouds
o3d.visualization.draw_geometries(point_clouds)

# Function for pairwise registration using ICP
def pairwise_registration(source, target):
    icp_coarse = o3d.pipelines.registration.registration_icp(
        source, target, 0.02, np.eye(4),
        o3d.pipelines.registration.TransformationEstimationPointToPlane())
    icp_fine = o3d.pipelines.registration.registration_icp(
        source, target, 0.01, icp_coarse.transformation,
        o3d.pipelines.registration.TransformationEstimationPointToPlane())
    return icp_fine.transformation

# List to store transformed point clouds
transformed_pcds = [point_clouds[0]]  # Start with the first point cloud

# Register and transform each subsequent point cloud
for i in range(1, len(point_clouds)):
    transformation = pairwise_registration(transformed_pcds[-1],
point_clouds[i])
    point_clouds[i].transform(transformation)
    transformed_pcds.append(point_clouds[i])

# Combine all the point clouds into one
combined_pcd = transformed_pcds[0]
```

```
for pcd in transformed_pcds[1:]:
    combined_pcd += pcd

# Visualize the combined point cloud
o3d.visualization.draw_geometries([combined_pcd])

# Remove outliers from the combined point cloud
clean_pcd, ind = combined_pcd.remove_statistical_outlier(nb_neighbors=20,
std_ratio=2.0)

# Downsample the point cloud
clean_pcd = clean_pcd.voxel_down_sample(voxel_size=0.01)

# Surface reconstruction using Poisson method
mesh, densities =
o3d.geometry.TriangleMesh.create_from_point_cloud_poisson(clean_pcd,
depth=9)
mesh = mesh.remove_degenerate_triangles()
mesh = mesh.remove_duplicated_triangles()
mesh = mesh.remove_non_manifold_edges()
mesh.compute_vertex_normals()

# Visualize the reconstructed mesh
o3d.visualization.draw_geometries([mesh])
```

The following code take RGB-D images as input, convert them it into point clouds, and combine these point clouds to construct a 3D image output.