DSC 210 Final Project: 3D Reconstruction

Aditya Paranjape	**Kavya Sridhar**	**Susmit Singh**	**Varadraj Bartakke**
MS Computer Science	MS Data Science	MS Data Science	MS Data Science
A69034007	A69035242	A69034300	A69034734

1 Introduction

3D reconstruction is the process of creating a three-dimensional representation of an object or scene from two-dimensional images. It involves extracting spatial information from 2D images and translating it into a 3D structure, enabling a more immersive and accurate representation of real-world objects.

In this project, we explored 3D reconstruction by generating dense point clouds using two distinct methodologies. First, we employed numerical linear algebra (NLA) based approach and then implemented state-of-the-art (SOTA) methods, including Neural Radiance Fields (NeRF) and Gaussian Splatting, to develop dense point clouds from 2D images. Finally, we compared the results of the linear algebra-based approach and the SOTA methods, analyzing their effectiveness in terms of output quality. This report outlines the methodology, implementation, and results of classical algorithms along with cutting-edge technologies for 3D reconstruction.

1.1 History/Background

3D reconstruction in computer vision involves the generation of three-dimensional models of objects or scenes from 2D images or other sensor data. Its origins trace back to photogrammetry in the mid-20th century, where photographs were used to estimate the dimensions and shapes of objects. Early methods relied on geometry and manual processing to extract depth information.

The field picked up speed in the 1980s with advances in computer vision and algorithms for stereo vision, which allowed two or more images to be used to infer depth. Epipolar geometry and bundle adjustment became foundational tools. In the 1990s and 2000s, improvements in feature detection-e.g., SIFT, SURF-and Structure from Motion techniques enabled automated and scalable 3D reconstruction from image sequences.

In the 2010s, significant breakthroughs arose with the introduction of deep learning and CNNs. These techniques improved the performance on single-image 3D shape reconstruction, bypassing several limitations of classical geometry-based approaches. Other techniques included volumetric reconstruction, point cloud generation, and differentiable rendering, which further fine-grained the results.

Today, 3D reconstruction finds important applications in virtual reality, robotics, autonomous vehicles, and the preservation of cultural heritage with more and more realistic results, leveraging multimodal inputs from LiDAR, depth cameras, and photogrammetry.

1.2 Applications

3D reconstruction has broad applications in industries that help in creating detailed 3D models for further analysis, visualization, and interaction. Medical imaging is one such field where it finds its application for surgical planning and diagnosis using 3D models of organs and tissues from MRI or CT scans. For instance, 3D reconstructions help in customizing prosthetics and implants.

It helps in the digital archiving of artifacts and historical sites in cultural heritage preservation, such as the reconstruction of Notre-Dame Cathedral after it caught fire. In entertainment, 3D reconstruction powers realistic CGI in films and games, creating life-like environments and characters.

3D mapping is critical in robotics for navigation and obstacle detection, as seen in Tesla's use of LiDAR and cameras. Architecture and real estate make use of 3D models for virtual tours and design visualization, as with platforms like Matterport. Lastly, e-commerce uses 3D models to enhance product interaction, exemplified by IKEA's augmented reality app for virtual furniture placement.

1.3 State-of-the-art

Gaussian Splatting and Neural Radiance Fields (NeRF) are advanced techniques for 3D scene representation and rendering. Gaussian Splatting uses spatially distributed Gaussian densities to efficiently render and reconstruct 3D data in real-time, excelling in computational simplicity and adaptability. NeRF employs neural networks to encode volumetric radiance fields, producing highly detailed and photorealistic 3D reconstructions from 2D views. While NeRF offers superior fidelity, Gaussian Splatting is faster and well-suited for real-time applications, highlighting complementary strengths in 3D scene rendering.

2 Problem Formulation

2.1 Relation to numerical linear algebra

Numerical Linear Algebra (NLA) forms the backbone of our 3D reconstruction process, particularly in creating dense point clouds. The workflow begins with camera calibration, where systems of linear equations are solved to estimate intrinsic and extrinsic camera parameters, establishing the transformation between the 3D world and its 2D projections. In the next step, SIFT (Scale-Invariant Feature Transform) is used to detect and match key points across multiple images, leveraging matrix operations to identify correspondences. The relationship between these correspondences is then encapsulated in the fundamental matrix, which is computed by solving an overdetermined system of linear equations using methods like Singular Value Decomposition (SVD). Following this, camera pose estimation determines the relative positions and orientations of cameras through matrix factorizations and eigenvector computations, ensuring consistency with the fundamental matrix. Finally, triangulation reconstructs 3D points from their 2D correspondences by solving linear equations derived from camera projection matrices. Each step heavily relies on NLA techniques to ensure precise and efficient computation, making it an indispensable component of our methodology for generating dense point clouds.

2.2 Approach description

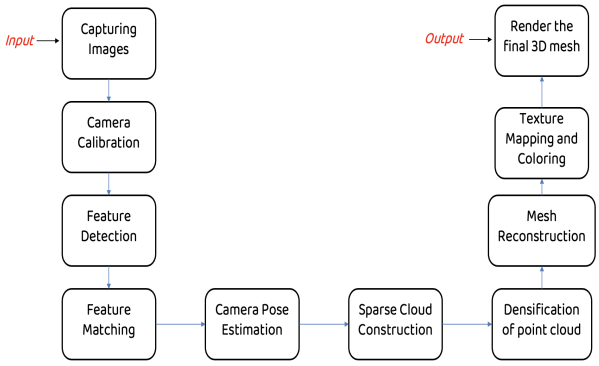


Figure 1: Numerical Linear Algebra Approach

2.2.1 Capturing Images

Capturing images for 3D renders from 2D images entails capturing an object or scene from several perspectives in order to collect the essential data for reconstruction. The goal is to obtain high-resolution, distortion-free photos from multiple angles with adequate overlap between them. We captured 47 images of 3000 x 4000 resolution

2.2.2 Camera Calibration

Camera calibration determines a camera's intrinsic and extrinsic parameters to map 3D points to 2D coordinates. It involves the intrinsic matrix K, extrinsic parameters $[R \mid t]$, and distortion coefficients. The intrinsic matrix K, representing focal lengths (f_x, f_y) and the principal point (c_x, c_y), is defined as:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}.$$

The extrinsic matrix $[R \mid t]$ describes the camera's orientation and position in the world, where R is a 3×3 rotation matrix and t is a 3×1 translation vector.

The projection of a 3D point $X_w = [X, Y, Z]^T$ from the world coordinate system onto the 2D image plane is mathematically described as:

$$x = K[R \mid t]X_w,$$

where $x = [u, v, 1]^T$ represents the homogeneous coordinates of the 2D image point.

The estimation of intrinsic and extrinsic parameters involves combining these parameters into a camera projection matrix P, defined as $P = K[R \mid t]$. This 3×4 matrix maps 3D world points to 2D image points. Using multiple point correspondences, the values of the projection matrix P are calculated. The relationship can be expressed as:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}.$$

Expanding this equation, the expressions for u and v are derived as:

$$u \cdot (p_{31}X + p_{32}Y + p_{33}Z + p_{34}) = p_{11}X + p_{12}Y + p_{13}Z + p_{14},$$

$$v \cdot (p_{31}X + p_{32}Y + p_{33}Z + p_{34}) = p_{21}X + p_{22}Y + p_{23}Z + p_{24}.$$

These equations can be compactly represented as $A \cdot p = 0$, where p is the vector of projection matrix coefficients, and A is a matrix formed using the point correspondences. Finally, the RQ decomposition is performed on the matrix P to retrieve the intrinsic matrix K, the rotation matrix R and the translation vector t.

2.2.3 Feature Detection with SIFT

The SIFT (Scale-Invariant Feature Transform) technique is a common method for feature recognition and description because it can withstand scale, rotation, and lighting variations.

Step 1: Scale-Space Construction
Scale-space construction is a key step in the SIFT algorithm. It identifies and characterizes local image features, ensuring robustness to scale, rotation, and minor viewpoint changes. This process involves two main steps: creating the scale space and finding the difference between Gaussians.

The scale space of an image is created by progressively blurring the image using a Gaussian filter. The scale space representation $L(x, y, \sigma)$ is defined as:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y),$$

where $G(x, y, \sigma)$ is the Gaussian kernel with standard deviation σ, $I(x, y)$ is the input image, and $*$ denotes the convolution operation.
Key points are detected by computing the Difference of Gaussians (DoG), which subtracts blurred images at nearby scales:

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma),$$

where k is a constant multiplicative factor.

Step 2: Key Point Detection
For each pixel in the DoG image, its value with its 26 neighbours is compared:

- 8 neighbours in the current scale.
- 9 neighbours in the scale above.
- 9 neighbours in the scale below.

A pixel is considered a key point if it is a local maximum or minimum.
Low-contrast points are deleted to improve stability. A key point is discarded if its intensity in the DoG image falls below a predetermined level.
Edges often produce unstable key points. To remove these, the *Hessian matrix* H is calculated for each key point:

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$

The eigenvalues of H are used to measure the curvature. Key points with a high ratio of eigenvalues (indicating a strong edge response) are rejected.

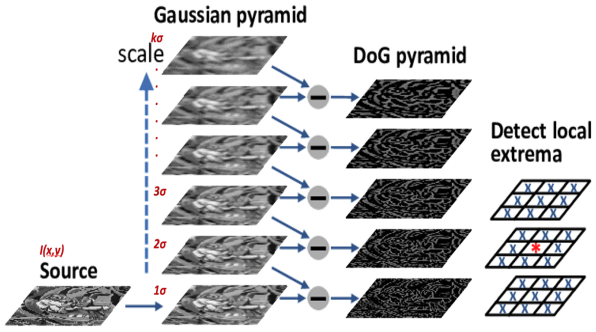


Figure 2: Key Point Detection

Step 3: Orientation Assignment
For rotation invariance, an orientation is assigned to each key point based on the local image gradient. For a region around the key point, compute the gradient magnitude $m(x, y)$ and orientation $\theta(x, y)$, as follows:

$$m(x, y) = \sqrt{(L_x)^2 + (L_y)^2}$$

$$\theta(x, y) = \tan^{-1}\left(\frac{L_y}{L_x}\right)$$

Gradients are weighted by their magnitudes and *binned* into a *histogram of 36 bins* (covering 360°). The orientation with the highest peak in the histogram is assigned to the key point. Additional key points may be created for secondary peaks.

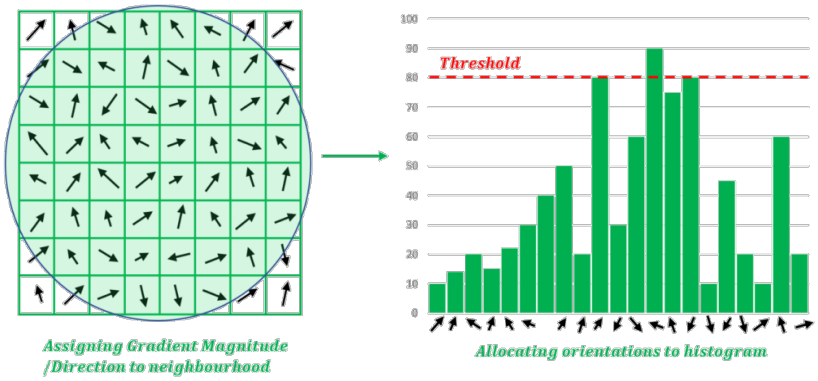


Figure 3: Orientation Histogram

Step 4: Key Point Descriptor A descriptor is generated for each key point to describe its local appearance. Around each key point, a 16×16 window is taken and divided into 4×4 subregions. For each subregion, an 8-bin histogram of gradient orientations is computed. A Gaussian window weights the gradient magnitudes to reduce the influence of distant gradients. Concatenate the histograms from all 16 subregions to form a descriptor vector $4 \times 4 \times 8 = 128$.

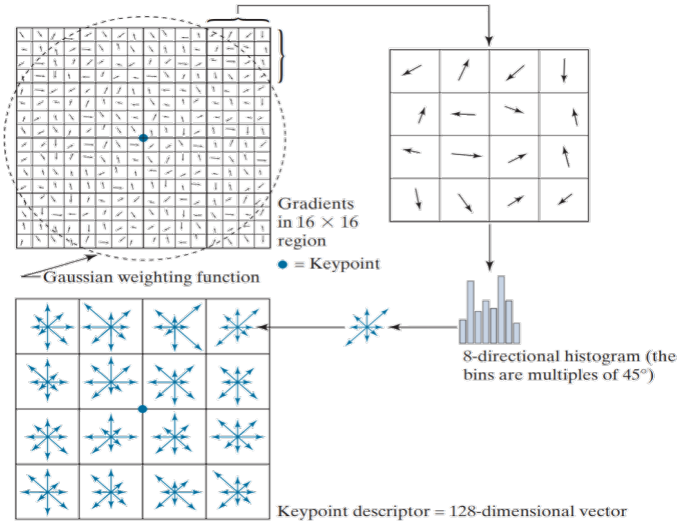


Figure 4: Key Point Descriptor

2.2.4 Feature Matching (Euclidean Distance and KNN Based)

Once the descriptors are generated for key points in two images, match them using:

- Nearest Neighbor Search (Euclidean distance).
- *Lowe's Ratio Test*: Retain matches where the distance to the nearest neighbor is significantly smaller than to the second nearest.

Let d_1 and d_2 represent the distances to the closest and second-closest matches, respectively. The Lowe's ratio is calculated as:

$$\text{Ratio} = \frac{d_1}{d_2}$$

If the ratio is below the threshold, the match is considered reliable; otherwise, the match is discarded:

$$\frac{d_1}{d_2} < \text{threshold}$$

2.2.5 Camera Pose Estimation

The goal of camera pose estimation is to determine the **relative position and orientation** of the camera for each image. This is achieved using *epipolar geometry*, which describes the geometric relationship between two views.

Mathematically, the relationship is expressed as:

$$\mathbf{x'}^T \mathbf{F} \mathbf{x} = 0$$

where $\mathbf{F}$ is the Fundamental matrix (3×3), and

$$\mathbf{x} = \begin{bmatrix} u_1 \\ v_1 \\ 1 \end{bmatrix}, \quad \mathbf{x'} = \begin{bmatrix} u_2 \\ v_2 \\ 1 \end{bmatrix}$$

are corresponding points in two images.

The Fundamental Matrix can be decomposed to estimate the relative **rotation** $\mathbf{R}$ and **translation** $\mathbf{t}$ between the two views.

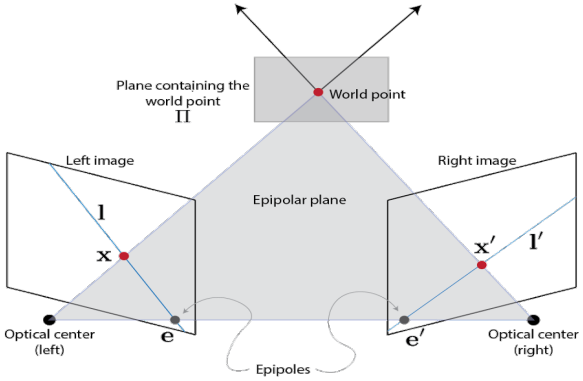


Figure 5: Camera Pose Estimation

2.2.6 Fundamental matrix calculation

The 8-point algorithm is a method for estimating the Fundamental Matrix F using at least 8 pairs of corresponding points from two images.

$$\begin{bmatrix} u_2 & v_2 & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ v_1 \\ 1 \end{bmatrix} = 0$$

$$f_{11}u_2u_1 + f_{12}u_2v_1 + f_{13}u_2 + f_{21}v_2u_1 + f_{22}v_2v_1 + f_{23}v_2 + f_{31}u_1 + f_{32}v_1 + f_{33} = 0 \quad \ldots \quad \text{(i)}$$

This gives a single equation for a pair of corresponding points $(\mathbf{x}, \mathbf{x'})$:

$$[u_2u_1, u_2v_1, u_2, v_2u_1, v_2v_1, v_2, u_1, v_1, 1] \, [f_{11}, f_{12}, f_{13}, f_{21}, f_{22}, f_{23}, f_{31}, f_{32}, f_{33}]^T = 0$$

Here, the coefficients of $\mathbf{F}$ are flattened into a 9-dimensional vector:

$$\text{vec}(\mathbf{F}) = [f_{11}, f_{12}, f_{13}, f_{21}, f_{22}, f_{23}, f_{31}, f_{32}, f_{33}]^T$$

If there are n pairs of corresponding points $(x, x')_{i=1}^{n}$, stack the equations into a single matrix A:

$$A = \begin{bmatrix} u_2^1 u_1^1 & u_2^1 v_1^1 & u_1^1 u_2^1 & u_1^1 v_1^1 & v_2^1 u_1^1 & v_2^1 v_1^1 & u_1^1 & v_1^1 & 1 \\ u_2^2 u_1^2 & u_2^2 v_1^2 & u_1^2 u_2^2 & u_1^2 v_1^2 & v_2^2 u_1^2 & v_2^2 v_1^2 & u_1^2 & v_1^2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ u_2^n u_1^n & u_2^n v_1^n & u_1^n u_2^n & u_1^n v_1^n & v_2^n u_1^n & v_2^n v_1^n & u_1^n & v_1^n & 1 \end{bmatrix}$$

Each row corresponds to one pair of points, and the matrix A has dimensions $n \times 9$ (at least $n = 8$). Homogeneous System can be rewritten as: The stacked equations form a homogeneous linear system:

$$A \cdot \text{vec}(F) = 0$$

Solve Using Singular Value Decomposition (SVD), Perform SVD on A Decompose A using SVD:

$$A = U \cdot \Sigma \cdot V^T$$

- U: $n \times n$ orthogonal matrix.
- Σ: $n \times 9$ diagonal matrix of singular values.
- V: 9×9 orthogonal matrix

The solution to $A \cdot \text{vec}(F) = 0$ corresponds to the singular vector of A associated with the smallest singular value. This vector is the last column of V:

$$\text{vec}(F) = V[:, -1]$$

Reshape $\text{vec}(F)$ back into a 3×3 matrix F. The Fundamental Matrix $\mathbf{F}$ should have rank 2 because it represents a mapping between two planes in projective geometry. Decompose $\mathbf{F}$ using SVD:

$$\mathbf{F} = \mathbf{U}_F \cdot \mathbf{\Sigma}_F \cdot \mathbf{V}_F^T$$

Where $\mathbf{\Sigma_F} = \text{diag}(\sigma_1, \sigma_2, \sigma_3)$ are the singular values of $\mathbf{F}$. Set the smallest singular value σ_3 to 0 to enforce the rank-2 constraint:

$$\mathbf{\Sigma}'_{\mathbf{F}} = \text{diag}(\sigma_1, \sigma_2, 0)$$

Reconstruct the rank-2 Fundamental Matrix:

$$\mathbf{F} = \mathbf{U}_F \cdot \mathbf{\Sigma}'_F \cdot \mathbf{V}_F^T$$

2.2.7 Camera pose calculation

Estimate the camera pose (relative rotation and translation) between the two camera views. The relationship between $\mathbf{F}$ and camera pose comes from the equation:

$$\mathbf{F} = [\mathbf{t}]_\times \cdot \mathbf{R}$$

Where: $[\mathbf{t}]_\times$: Skew-symmetric matrix of the translation vector $\mathbf{t}$. $\mathbf{R}$: Rotation matrix.

$$[\mathbf{t}]_\times = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}$$

Use the decomposition of Fundamental Matrix $\mathbf{F}$ done earlier:

$$\mathbf{F} = \mathbf{U}_F \cdot \mathbf{\Sigma}'_F \cdot \mathbf{V}_F^T$$

From the SVD, we compute candidate solutions for $\mathbf{R}$ and $\mathbf{t}$:

Translation: $\mathbf{t} = \mathbf{U}[:, 3]$ (Third column of $\mathbf{U}$) Rotation: Construct two possible rotations using:

$$\mathbf{W} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_1 = \mathbf{U} \cdot \mathbf{W} \cdot \mathbf{V}^\top$$

$$\mathbf{R}_2 = \mathbf{U} \cdot \mathbf{W}^\top \cdot \mathbf{V}^\top$$

The decomposition yields four possible solutions:

- $(\mathbf{R}_1, \mathbf{t})$
- $(\mathbf{R}_1, -\mathbf{t})$
- $(\mathbf{R}_2, \mathbf{t})$
- $(\mathbf{R}_2, -\mathbf{t})$

To determine the correct $(\mathbf{R}, \mathbf{t})$, we use the Cheirality condition, which ensures that the reconstructed 3D points lie in front of both cameras.

2.2.8 dense cloud generation by Triangulation

In 3D geometry, a reconstructed 3D point $\mathbf{x}$ is considered valid if it lies in front of both cameras. Mathematically, this means the depth Z (the third component of $\mathbf{x}$ in each camera's coordinate system) must be positive for both cameras:

- For the first camera: $Z_1 > 0$
- For the second camera: $Z_2 > 0$

The correct pose $(R, \mathbf{t})$ will satisfy this condition for most points. The cameras are defined in a coordinate system such that:

- For the first camera: $\mathbf{P}_1 = [\mathbf{I}, \mathbf{0}]$ This is the canonical projection matrix for the first camera, with identity rotation and zero translation.
- For the second camera: $\mathbf{P}_2 = [\mathbf{R}, \mathbf{t}]$ where $\mathbf{R}$ and $\mathbf{t}$ are derived from the decomposition of $\mathbf{F}$.

Given two corresponding points $\mathbf{x} = [u_1, u_2, 1]^\top$ in Image 1 and $\mathbf{x}' = [u_1', u_2', 1]^\top$ in Image 2, the 3D point $\mathbf{P} = [X, Y, Z, 1]^\top$ is reconstructed by solving the following system of equations:

$$\mathbf{x}^\top \cdot (\mathbf{P}_1 \cdot \mathbf{P}) = 0$$
$$\mathbf{x}'^\top \cdot (\mathbf{P}_2 \cdot \mathbf{P}) = 0$$

Expanding for both cameras: For the first camera,

$$u_1 X + v_1 Y + Z = 0$$

For the second camera,

$$u_2(\mathbf{R}_{11}X + \mathbf{R}_{12}Y + \mathbf{R}_{13}Z + t_x) + v_2(\mathbf{R}_{21}X + \mathbf{R}_{22}Y + \mathbf{R}_{23}Z + t_y) + (\mathbf{R}_{31}X + \mathbf{R}_{32}Y + \mathbf{R}_{33}Z + t_z) = 0$$

The above system of equations can be rewritten into a linear system:

$$\mathbf{A} \cdot \mathbf{P} = 0$$

Where $\mathbf{A}$ is a 4×4 matrix constructed as:

$$A = \begin{bmatrix} u_1 P_{1,3}^T - P_{1,1}^T \\ v_1 P_{1,3}^T - P_{1,2}^T \\ u_2 P_{2,3}^T - P_{2,1}^T \\ v_2 P_{2,3}^T - P_{2,2}^T \end{bmatrix}$$

Here, $P_{1,i}$ and $P_{2,i}$ are the rows of $\mathbf{P}_1$ and $\mathbf{P}_2$. Solve $\mathbf{A} \cdot \mathbf{P} = 0$ using SVD, decompose $\mathbf{A}$ as: $\mathbf{A} = \mathbf{U} \cdot \mathbf{\Sigma} \cdot \mathbf{V}^\top$ The solution for $\mathbf{P}$ is the last column of $\mathbf{V}$ (corresponding to the smallest singular value) $\mathbf{P} = \mathbf{V}[:, -1]$ Normalize $\mathbf{P}$ so that its fourth component is 1

$$\mathbf{P} = \frac{\mathbf{P}}{\mathbf{P}[3]}$$

To resolve pose ambiguities using triangulation:

Compute the depth of $\mathbf{P}$ in the first camera's frame:

$$Z_1 = \mathbf{P}[3]$$

Compute the depth of $\mathbf{P}$ in the second camera's frame:

$$Z_2 = (\mathbf{R} \cdot \mathbf{P} + \mathbf{t})[3]$$

For a valid solution, both Z_1 and Z_2 must be positive.

2.3 SOTA approach description

Gaussian Splatting:
Gaussian Splatting represents 3D scenes as spatially distributed Gaussians in a volumetric grid, enabling efficient real-time rendering and reconstruction. By smoothing point cloud data with Gaussian kernels, it creates a density field that can be rendered directly without complex neural network computations. This approach excels in scenarios requiring speed and computational efficiency, making it ideal for real-time applications such as virtual reality and gaming.

Neural Radiance Fields (NeRF):
NeRF models encode volumetric radiance fields using neural networks, learning to map 3D coordinates and viewing directions to RGB values and density. By optimizing over multiple 2D views, NeRF generates high-fidelity 3D reconstructions with remarkable detail and photorealism. While computationally intensive, NeRF has become a benchmark for 3D scene representation in applications such as visual effects and digital content creation.

2.4 Evaluation

In this project, we have not trained any model; therefore, we do not have ground truth data for evaluation. As a result, the usual methods for quantitative comparison cannot be applied. Instead, the results must be evaluated visually to determine if the generated point clouds accurately represent the object.

Since texture and color mapping are beyond the scope of this project, the dense point clouds will only provide a rough representation of the object's geometry.

Feature matching results can be assessed by plotting straight lines that connect the matched keypoints between two consecutive images. This visual representation helps in distinguishing between good and bad feature matches, providing a clear understanding of the matching quality.

3 Experiments

3.1 Setup and logistics

- We used Python as the programming language to code the algorithms for dense point cloud generation. Google Colab CPU and GPU runtime was used to run and test the codes. Github was used to version control the project and for code base reference.
- OpenCV was used for image processing. Matplotlib was used for plots and image processing results. Numpy was used for algebraic operations performed on matrices and other numerical operations. Open3D was used for triangulation and creating the dense point clouds.
- COLMAP was used to perform Structure-from-Motion (SfM) and Multi-View Stereo (MVS) for the dataset. The camera intrinsic and extrinsic parameters derived from COLMAP serve as essential inputs for the project's 3D reconstruction and depth estimation tasks.

3.2 Dataset

The dataset used in this project was developed by our team, and no open-source datasets were utilized. For 3D reconstruction, we required images of the object of interest such that all views of the object were captured. Using artificial lighting and a textureless background, we captured a 360-degree view of the object. Each image was taken with approximately 80% overlap, which is essential for matching keypoints and ultimately aids in dense point cloud generation.

A total of 47 image (3000px x 4000px) of the object (a small bottle of spice) were used for this project. This dataset was finalized after several trials with different objects, methods of capturing images, and varying numbers of images. In earlier datasets, we applied our algorithm but identified shortcomings during the initial stages, leading us to discard them. The current dataset was finalized after overcoming these challenges.

Figure 6: Dataset

3.3 Implementation

First of all we have to perform the camera calibration to obtain the internal parameters of the camera, for calibration we used a checkerboard with 13 horizontal and 9 vertical corner. Checkboard images were also resized to 750 x 1000 resolution.

```
import cv2
import numpy as np
import glob

# Checkerboard dimensions (internal corners)
CHECKERBOARD = (13, 9)  # 13 horizontal and 9 vertical internal
    corners

# Prepare object points assuming unit square size (relative scale)
objp = np.zeros((CHECKERBOARD[0] * CHECKERBOARD[1], 3), np.float32)
objp[:, :2] = np.mgrid[0:CHECKERBOARD[0], 0:CHECKERBOARD[1]].T.reshape
    (-1, 2)

# Arrays to store object points and image points from all images
objpoints = []  # 3D points in the real world
```

```
imgpoints = []  # 2D points in the image plane

# Load all checkerboard images
images = glob.glob('resized_checkerboard_images/*.jpg')  # Update this path

for image_file in images:
    image = cv2.imread(image_file)
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Find the checkerboard corners
    ret, corners = cv2.findChessboardCorners(gray, CHECKERBOARD, None)

    if ret:
        objpoints.append(objp)
        imgpoints.append(corners)

# Perform camera calibration
ret, K, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, None)

# Output the intrinsic matrix and distortion coefficients
if ret:
    print("Intrinsic Matrix (K):\n", K)
    print("\nDistortion Coefficients:\n", dist)
else:
    print("Camera calibration failed. Ensure the checkerboard images are correct.")
```

The camera calibration was performed on local system (PyCharm IDE). We will manually input the instrinsic parameter values in later steps (Essential Matrix calculation). From here onwards all the code was implemented on Google Colab.

First of all we have to mount our google drive so that we can import the dataset from it.

The images can also be directly upload to the disk space of Google Colab but it won't persist after the session crashes. If the dataset is uploaded on the Google Drive, all we have to do is just mount the Drive on the Colab notebook.

```
from google.colab import drive
drive.mount('/content/drive')
```

Except for Open3D, all the packages which we needed were already present in the python environment of the Colab runtime. So, we will install the Open3d using the following command.

```
!pip install open3d
```

The image from the dataset needs to loaded for further processing.

```
import cv2
import os
import glob

# Set dataset path
dataset_path = '/content/drive/MyDrive/DSC 210 Final Project/Dataset'

# Load images
images = []
image_paths = sorted(glob.glob(os.path.join(dataset_path, '*.jpg')))

for img_path in image_paths:
    img = cv2.imread(img_path)
    images.append(cv2.cvtColor(img, cv2.COLOR_BGR2GRAY))

print(f"Loaded {len(images)} images.")
```

After the images are loaded, they need to be downscaled to smaller resolution. The original images were of size 3000 x 4000 each. This resolution is too high for downstream tasks and will take unnecessarily high compute and time. So, we bring it down to one-fourth the original resolution.

```
# Set the desired resolution
desired_width = 750  # Adjust as needed
desired_height = 1000  # Adjust as needed
desired_resolution = (desired_width, desired_height)

# Downscale images to the specific resolution
downscaled_images = []

for img in images:
    # Resize image to the desired resolution
    resized_img = cv2.resize(img, desired_resolution, interpolation=
    cv2.INTER_AREA)
    downscaled_images.append(resized_img)

print(f"Images resized to {desired_width}x{desired_height} resolution.
    ")
# Use downscaled_images for further processing
images = downscaled_images
```

Perform a sanity check on all the images. Print their resolution to check if they got resized correctly

```
# Print the resolution of images
for idx, img in enumerate(images):
    print(f"Image {idx+1}: Resolution = {img.shape[1]} x {img.shape
    [0]}")
```

Now we found the keypoints in the images using Scale Invariant Feature Transform (SIFT) algorithm. We visualize each of the image along with their detected keypoints.

```
import matplotlib.pyplot as plt

# SIFT Feature Detection
sift = cv2.SIFT_create()
keypoints_list, descriptors_list = [], []

# Iterate through each image for feature detection
for i, img in enumerate(images):
    keypoints, descriptors = sift.detectAndCompute(img, None)
    keypoints_list.append(keypoints)
    descriptors_list.append(descriptors)

    # Visualize the detected keypoints
    img_with_keypoints = cv2.drawKeypoints(
        img, keypoints, None, flags=cv2.
    DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
    )

    # Display the image with keypoints
    plt.figure(figsize=(8, 6))
    plt.title(f"Feature Detection using SIFT for Image {i}")
    plt.imshow(cv2.cvtColor(img_with_keypoints, cv2.COLOR_BGR2RGB))
    plt.axis('off')
    plt.show()

print("Feature detection and visualization complete.")
```

Figure 6: Feature Detection using SIFT

Now we match the detected key points between consecutive image pairs and visualize them.

```python
# Feature Matching

bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=False)  # Set crossCheck to False for knnMatch
matches = []
ratio_thresh = 0.75  # Lowe's ratio test threshold

# Iterate through image pairs for matching and visualization
for i in range(len(images) - 1):
    # Perform kNN matching
    knn_matches = bf.knnMatch(descriptors_list[i], descriptors_list[i + 1], k=2)

    # Apply Lowe's ratio test
    good_matches = []
    for m, n in knn_matches:
        if m.distance < ratio_thresh * n.distance:
            good_matches.append(m)
    matches.append(good_matches)

    # Draw the good matches
    img_matches = cv2.drawMatches(
        images[i], keypoints_list[i],  # First image and its keypoints
        images[i + 1], keypoints_list[i + 1],  # Second image and its keypoints
        good_matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
    )

    # Display the matches using Matplotlib
    plt.figure(figsize=(12, 6))
    plt.title(f"Feature Matching between Image {i} and Image {i + 1}")
    plt.imshow(cv2.cvtColor(img_matches, cv2.COLOR_BGR2RGB))
    plt.axis('off')
    plt.show()

print("Feature matching visualization complete.")
```

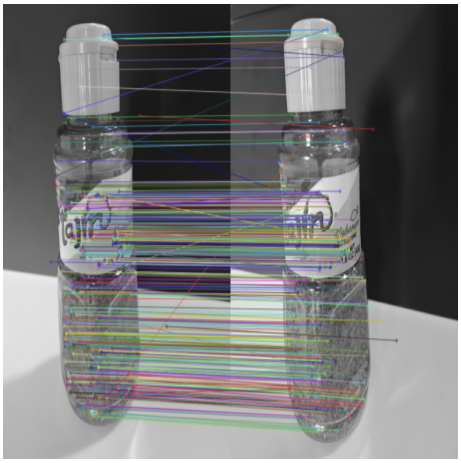


Figure 7: Feature Matching

Then we find the fundamental matrix which is needed to find the camera poses.

```
# Fundamental Matrix
import numpy as np
fundamental_matrices = []
for i, match in enumerate(matches):
    pts1 = np.float32([keypoints_list[i][m.queryIdx].pt for m in match
    ])
    pts2 = np.float32([keypoints_list[i + 1][m.trainIdx].pt for m in
    match])
    F, mask = cv2.findFundamentalMat(pts1, pts2, cv2.FM_RANSAC)
    fundamental_matrices.append(F)

print("Fundamental matrices calculated.")
```

After the fundamental matrix is calculated the Essential matrix is calculated with the help of Instrinsic parameter matrix of the camera which we found out during camera calibration.

```
# Camera Pose Recovery
# Replace with the actual intrinsic parameters of your camera or
    dataset
fx = 704.95  # Example focal length in x
fy = 704.70  # Example focal length in y
cx = 382.04  # Example principal point x
cy = 496.37  # Example principal point y

K = np.array([[fx, 0, cx], [0, fy, cy], [0, 0, 1]])
poses = []

for F in fundamental_matrices:
    E = K.T @ F @ K
    _, R, t, _ = cv2.recoverPose(E, pts1, pts2, K)
    poses.append((R, t))

print("Camera poses recovered.")
```

Now we perform Triangulation using keypoint correspondences and camera poses.

```
# Triangulation
points_3d = []

for i, (R, t) in enumerate(poses):
    P1 = K @ np.hstack((np.eye(3), np.zeros((3, 1))))
    P2 = K @ np.hstack((R, t))
    pts4D = cv2.triangulatePoints(P1, P2, pts1.T, pts2.T)
```

```
    points_3d.append(cv2.convertPointsFromHomogeneous(pts4D.T))

print("Triangulation complete.")
```

After Triangulation, we create disparity maps which gives us the depth of 3D point in the point cloud

```
# Disparity Maps

# Resize images to the same size
reference_size = images[0].shape[:2]  # (height, width)
images_resized = [cv2.resize(img, (reference_size[1], reference_size
    [0])) for img in images]

# Create StereoBM matcher

stereo = cv2.StereoBM_create(numDisparities=64, blockSize=15)

# Directory to save disparity maps
output_dir = '/content/drive/MyDrive/DSC 210 Final Project/
    disparity_maps'
os.makedirs(output_dir, exist_ok=True)

# Generate disparity maps for consecutive pairs
for i in range(len(images_resized) - 1):
    img_left = images_resized[i]
    img_right = images_resized[i + 1]

    # Compute disparity map
    disparity = stereo.compute(img_left, img_right)

    # Normalize disparity for visualization
    disparity_normalized = cv2.normalize(disparity, None, alpha=0,
    beta=255, norm_type=cv2.NORM_MINMAX)
    disparity_normalized = cv2.convertScaleAbs(disparity_normalized)

    # Save the disparity map
    output_path = os.path.join(output_dir, f'disparity_map_{i:02d}.png
    ')
    cv2.imwrite(output_path, disparity_normalized)
    print(f"Saved disparity map: {output_path}")
```

From the disparity maps we create the depth maps.

```
# Convert Disparity to Depth Map
focal_length = 704  # Focal length in pixels
baseline = 0.1  # Baseline in meters


disparity_dir = '/content/drive/MyDrive/DSC 210 Final Project/
    disparity_maps'
depth_output_dir = '/content/drive/MyDrive/DSC 210 Final Project/
    depth_maps'
os.makedirs(depth_output_dir, exist_ok=True)
# Load disparity images and compute depth maps
disparity_images = sorted([os.path.join(disparity_dir, f) for f in os.
    listdir(disparity_dir) if f.endswith('.png')])

for i, disparity_path in enumerate(disparity_images):
    disparity = cv2.imread(disparity_path, cv2.IMREAD_UNCHANGED).
    astype(np.float32)
    # Avoid division by zero
    disparity[disparity == 0] = 1e-6

    # Compute depth
    depth = (focal_length * baseline) / disparity
```

```
    # Normalize and save depth map
    depth_normalized = cv2.normalize(depth, None, alpha=0, beta=255,
    norm_type=cv2.NORM_MINMAX)
    depth_normalized = np.uint8(depth_normalized)
# Dense Point Cloud

point_cloud_dir = '/content/drive/MyDrive/DSC 210 Final Project/
    point_clouds'
os.makedirs(point_cloud_dir, exist_ok=True)

for i, disparity_path in enumerate(disparity_images):
    depth_path = os.path.join(depth_output_dir, f'depth_map_{i:02d}.
    png')
    depth = cv2.imread(depth_path, cv2.IMREAD_UNCHANGED).astype(np.
    float32)

    # Reproject depth map to 3D points
    h, w = depth.shape
    Q = np.float32([[1, 0, 0, -w / 2],
                    [0, -1, 0, h / 2],
                    [0, 0, 0, -focal_length],
                    [0, 0, 1 / baseline, 0]])
    points_3d = cv2.reprojectImageTo3D(depth, Q)

    # Mask invalid points
    mask = depth > 0
    points = points_3d[mask]

    # Save point cloud using Open3D
    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(points)
    point_cloud_path = os.path.join(point_cloud_dir, f'point_cloud_{i
    :02d}.ply')
    o3d.io.write_point_cloud(point_cloud_path, pcd)
    print(f"Saved point cloud: {point_cloud_path}")
```

3.4 Results

The following results were achieved, dense point clouds generated using the linear algebra approach.

Figure 8: Dense Point CLoud

3.5 SOTA Implemenation

3.5.1 Gaussian Splatting Implementation

The implementation of the Gaussian splatting process involves multiple stages, including data loading, feature detection, feature matching, 3D reconstruction, and Gaussian splatting. Below, we detail the key components of the implementation using code snippets and accompanying explanations.

1. Data Loading: The first step involves loading the images from the dataset directory. Each image is converted from BGR to RGB format for visualization and processing.

```
import os
import cv2

# Function to load images
def load_images(path):
    images = []
    for file_name in sorted(os.listdir(path)):
        if file_name.lower().endswith(('png', 'jpg', 'jpeg')):
            img = cv2.imread(os.path.join(path, file_name))
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            images.append(img)
    return images
```

Explanation: This function iterates through the directory, reads valid image files, and appends them to a list for further processing. It ensures the images are compatible with OpenCV and visualization libraries.

2. Feature Detection and Matching: Features are detected using the SIFT algorithm, and matches between consecutive images are identified using a brute-force matcher.

```
# Function to detect keypoints and descriptors (SIFT)
def detect_features(image):
    sift = cv2.SIFT_create()
    gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    keypoints, descriptors = sift.detectAndCompute(gray, None)
    return keypoints, descriptors

# Function to match features between two images
def match_features(kp1, kp2, des1, des2):
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    matches = bf.match(des1, des2)
    matches = sorted(matches, key=lambda x: x.distance)
    return matches[:400]
```

Equations: Feature matching relies on minimizing the Euclidean distance between descriptor vectors:

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

Here, p and q are descriptor vectors of the matched keypoints.

3. 3D Reconstruction: To reconstruct 3D points, we use the essential matrix to estimate the camera motion and triangulate points from matched keypoints.

```
# Function to reconstruct 3D points
def reconstruct_3d(features, K):
    points_3d = []
    for i in range(len(features) - 1):
        matches = match_features(features[i][0], features[i+1][0],
                                 features[i][1], features[i+1][1])
```

```
        pts1 = np.float32([features[i][0][m.queryIdx].pt for m in
    matches])
        pts2 = np.float32([features[i+1][0][m.trainIdx].pt for m in
    matches])

        E, _ = cv2.findEssentialMat(pts1, pts2, K, method=cv2.RANSAC)
        _, R, t, _ = cv2.recoverPose(E, pts1, pts2, K)

        P1 = K @ np.hstack((np.eye(3), np.zeros((3, 1))))
        P2 = K @ np.hstack((R, t))
        points_4d = cv2.triangulatePoints(P1, P2, pts1.T, pts2.T)
        points_3d.append((points_4d[:3] / points_4d[3]).T)

    return np.vstack(points_3d) if points_3d else np.array([])
```

Equations: The relationship between 3D points and image points is given by:

$$\mathbf{x} = \mathbf{K}[\mathbf{R}|\mathbf{t}]\mathbf{X}$$

Where $\mathbf{x}$ is the 2D point in image coordinates, $\mathbf{X}$ is the 3D point in world coordinates, $\mathbf{K}$ is the intrinsic camera matrix, and $[\mathbf{R}|\mathbf{t}]$ is the extrinsic transformation matrix.

4. Gaussian Splatting: The reconstructed points are processed into a voxel grid, and a Gaussian filter is applied to achieve splatting.

```
# Function to process point cloud and apply Gaussian splatting
def process_point_cloud(points_3d):
    grid_size = 1024
    grid = np.zeros((grid_size, grid_size, grid_size))
    voxel_coords = (points_3d * (grid_size - 1)).astype(int)
    for point in voxel_coords:
        grid[point[0], point[1], point[2]] += 1
    return gaussian_filter(grid, sigma=1.0)
```

Explanation: The function normalizes 3D points into voxel grid coordinates and applies a Gaussian filter to create a density representation.

Equations: The Gaussian filtering process is mathematically defined as:

$$G(x, y, z) = \frac{1}{(2\pi\sigma^2)^{3/2}} \exp\left(-\frac{x^2 + y^2 + z^2}{2\sigma^2}\right)$$

Here, $G(x, y, z)$ represents the smoothed value at the voxel coordinate (x, y, z) after applying the Gaussian filter. σ is the standard deviation of the Gaussian distribution, controlling the spread of influence from each voxel. x, y, z are the distances of a voxel from the center of the Gaussian kernel in the respective dimensions.

3.5.2 Neural Radiance Fields (NeRF)

NeRF is a deep neural network that models a scene as a continuous volumetric representation, enabling the creation of photorealistic images from novel viewpoints. Instead of using traditional 3D models, NeRF represents the scene with a neural network that predicts color and density at any point in 3D space, given a specific camera viewpoint. This allows the generation of new images from different camera angles by learning both the geometry and the lighting of the scene.

Training NeRF

Training NeRF involves several key steps:

1. Collecting Multi-view Images

A set of images is taken from different viewpoints of the scene. These images do not need to cover the full 360-degree range but must provide diverse angles to learn the scene's geometry and lighting.

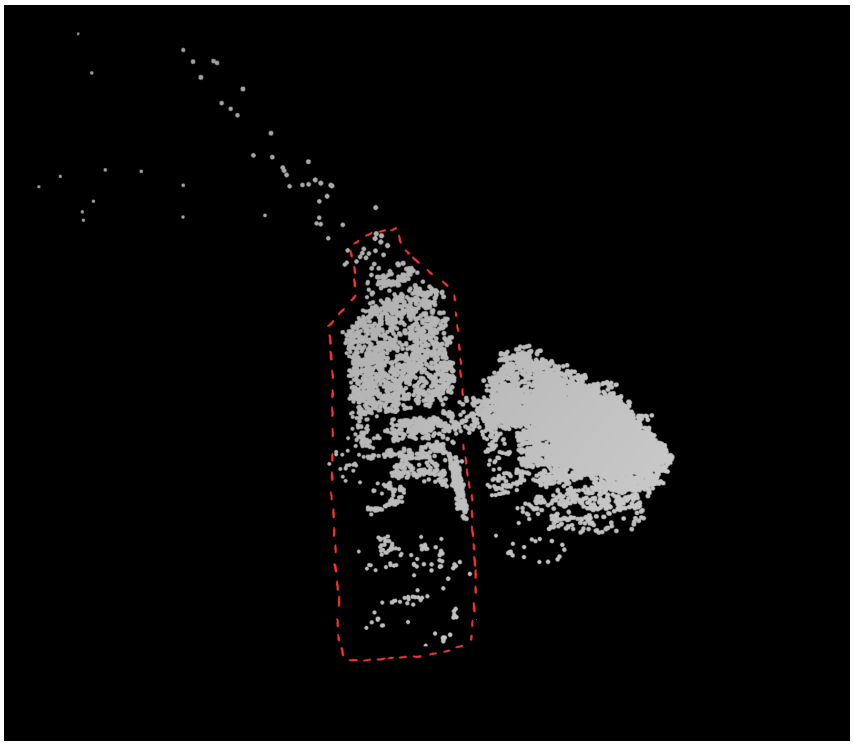


Figure 9: Gaussian point cloud.

2. Preprocessing

The images are preprocessed to normalize camera parameters, including intrinsics (such as focal length and principal point) and extrinsics (camera positions and orientations).

3. Model Training

The images are fed into the NeRF model, along with their corresponding camera viewpoints. The network computes the predicted color and opacity for each point along the rays passing through the scene.

3.5.3 Applications of NeRF

1. View Synthesis

NeRF can create high-quality images from novel viewpoints, even if the original dataset is dense or incomplete. Given a set of images from different angles, NeRF can generate photorealistic renderings from any viewpoint in 3D space.

2. 3D Reconstruction

NeRF is used for 3D scene reconstruction. By learning from multiple images taken from various angles, NeRF models the scene's geometry in continuous 3D space, enabling detailed 3D reconstructions.

3. Depth Estimation

NeRF can be applied to depth estimation. Since it learns a volumetric representation, it inherently captures depth information, which can be extracted by tracing rays from the camera and computing distances to the scene points.

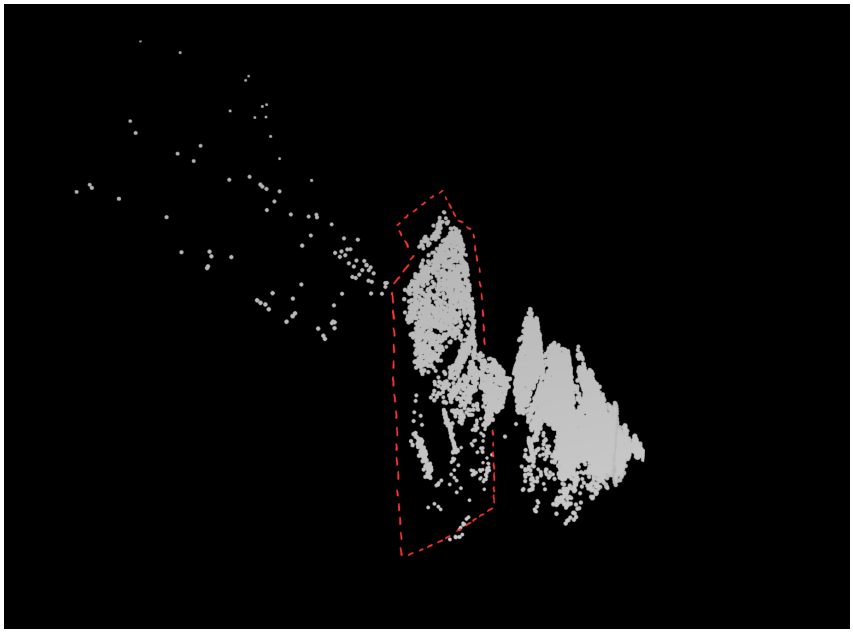


Figure 10: Gaussian point cloud.

4. Object Reconstruction

NeRF can be extended to model individual objects, enabling photorealistic reconstructions of complex 3D objects, such as sculptures or products, from 2D images.

3.5.4 Limitations of NeRF

1. Computational Complexity:
NeRF is computationally expensive, both during the training process and when rendering images. Rendering high-quality images can take several seconds or even minutes per frame.

2. Data Requirements
NeRF requires a substantial number of images from diverse viewpoints for effective training. For detailed scenes, a large dataset may be needed to capture the nuances of lighting and geometry accurately.

3. Limited to Static Scenes
NeRF is primarily designed for static scenes. Handling dynamic objects or moving cameras is much more complex and is an ongoing research challenge.

3.5.5 NeRF Implementation

NeRF: Structure from Motion (SfM)

Structure from Motion (SfM) is a technique used to estimate the 3D structure of a scene from multiple 2D images taken from different viewpoints. In this context, we use tools like **COLMAP** to estimate the camera parameters (intrinsics and extrinsics) by solving a problem known as **bundle adjustment**.

The relationship between corresponding points in two views is described by **Essential** and **Fundamental matrices**:

- The **Essential matrix** (**E**) encodes the relative rotation and translation between two camera views.
- The **Fundamental matrix** (**F**) relates the corresponding points in stereo images through the equation:

$$\mathbf{x_2}^T \mathbf{F} \mathbf{x_1} = 0$$

Solving for $\mathbf{E}$ and $\mathbf{F}$ typically involves **Singular Value Decomposition (SVD)**, which helps in determining the transformation between the two views.

NeRF: Camera Model

The **pin-hole camera model** is a fundamental concept in computer vision and photogrammetry that describes how 3D points in the world are mapped to 2D points in an image. The transformation is represented by the equation:

$$\mathbf{x} = \mathbf{K} [\mathbf{R} \mid \mathbf{t}] \mathbf{X}$$

Where:

- $\mathbf{x}$: Image coordinates (2D, in homogeneous form)
- $\mathbf{K}$: Camera's intrinsic matrix, which includes properties like focal length and optical center
- $\mathbf{R}$: Rotation matrix that aligns the 3D world coordinates with the camera's coordinate system
- $\mathbf{t}$: Translation vector, which defines the position of the camera in the 3D space
- $\mathbf{X}$: 3D coordinates in the world

This mathematical framework allows us to project a 3D scene onto a 2D plane. It uses **linear algebra** principles like **matrix multiplication** to map 3D points to their 2D projections. We also use **homogeneous coordinates** to handle transformations efficiently, including scaling and translation.

NeRF: Feature Detection and Matching

Feature detection involves identifying keypoints in images that are distinctive and can be tracked across different views. The **ORB** algorithm (Oriented FAST and Rotated BRIEF) is commonly used for this purpose. It extracts keypoints and describes them using a **binary descriptor**.

```
#1. NeRF Feature Detection

import cv2
import matplotlib.pyplot as plt

# Define the path to the images folder on Google Drive
base_path = "/content/drive/MyDrive/NeRF_implementation/"
images_path = base_path + "images_folder/"

# Load the images (update with the correct paths from Google Drive)
image1 = cv2.imread(images_path + 'Image3.jpg', cv2.IMREAD_GRAYSCALE)
image2 = cv2.imread(images_path + 'Image6.jpg', cv2.IMREAD_GRAYSCALE)

# Check if the images were loaded successfully
if image1 is None:
  print("Error loading image1.jpg")
if image2 is None:
  print("Error loading image2.jpg")

# Initialize ORB detector
orb = cv2.ORB_create(nfeatures=10000, scaleFactor=1.1, nlevels=8)

# Detect keypoints and descriptors
kp1, des1 = orb.detectAndCompute(image1, None)
kp2, des2 = orb.detectAndCompute(image2, None)

# Draw keypoints
img1_kp = cv2.drawKeypoints(image1, kp1, None, color=(0, 255, 0),
    flags=0)
```

```
img2_kp = cv2.drawKeypoints(image2, kp2, None, color=(0, 255, 0),
    flags=0)

# Plot the keypoints
plt.subplot(1, 2, 1), plt.imshow(img1_kp)
plt.subplot(1, 2, 2), plt.imshow(img2_kp)
plt.show()
```

When we have multiple images, the next step is **feature matching**, where we find corresponding keypoints across these images. This is done by calculating the **Hamming distance** between the binary descriptors of features. This helps in determining which keypoints in one image correspond to keypoints in another. The entire process is represented through matrix operations, which simplify matching and comparison tasks.

```
#2 NeRF Feature matching

#Initialize the Brute Force Matcher (BFMatcher)
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# Match descriptors between the two images
matches = bf.match(des1, des2)

# Sort the matches based on distance (the smaller the distance, the
    better the match)
matches = sorted(matches, key = lambda x: x.distance)

# Draw the matches
img_matches = cv2.drawMatches(image1, kp1, image2, kp2, matches[:20],
    None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Show the image with the matched keypoints
plt.imshow(img_matches)
plt.show()
```

NeRF: Depth Estimation and Disparity

In **stereo vision**, depth refers to the distance of points in the scene from the camera. The depth Z can be derived from the **disparity** d, which is the difference in the positions of corresponding points in two images.

The relationship is:

$$Z = \frac{f \cdot B}{d}$$

Where:

- f is the focal length of the camera
- B is the baseline, i.e., the distance between the two cameras
- d is the disparity

```
#3 NeRF Disparity and Edge Map

# Function to preprocess images
def preprocess_images(img1, img2):
    if len(img1.shape) == 3:
        img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
        img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    # Enhance contrast
    clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(8,8))
    img1 = clahe.apply(img1)
```

```
    img2 = clahe.apply(img2)

    # Bilateral filtering for denoising
    img1 = cv2.bilateralFilter(img1, 9, 75, 75)
    img2 = cv2.bilateralFilter(img2, 9, 75, 75)

    return img1, img2

# Function to create stereo matcher
def create_stereo_matcher():
    window_size = 7
    min_disp = -16
    num_disp = 192
    stereo = cv2.StereoSGBM_create(
        minDisparity=min_disp,
        numDisparities=num_disp,
        blockSize=window_size,
        P1=8 * 3 * window_size ** 2,
        P2=32 * 3 * window_size ** 2,
        disp12MaxDiff=2,
        uniquenessRatio=10,
        speckleWindowSize=50,
        speckleRange=4,
        preFilterCap=31,
        mode=cv2.STEREO_SGBM_MODE_HH
    )
    return stereo

# Function to post-process disparity map
def post_process_disparity(disparity):
    # Convert to float32
    disparity = disparity.astype(np.float32) / 16.0

    # Apply bilateral filter for smoothing
    disparity = cv2.bilateralFilter(disparity, 9, 75, 75)

    # Detect edges
    edges = cv2.Canny(np.uint8(disparity), 50, 150)

    # Combine disparity with edges
    disparity_with_edges = disparity.copy()
    disparity_with_edges[edges > 0] = 255

    return disparity_with_edges

# Load images
image1 = cv2.imread(image1_path)
image2 = cv2.imread(image2_path)

# Check if images are loaded
if image1 is None or image2 is None:
    print("Error: One or both images could not be loaded.")
else:
    # Preprocess images
    processed_left, processed_right = preprocess_images(image1, image2
    )

    # Compute disparity
    stereo = create_stereo_matcher()
    disparity = stereo.compute(processed_left, processed_right)

    # Post-process disparity
    filtered_disparity = post_process_disparity(disparity)

    # Visualization
```

```
    plt.figure(figsize=(12, 5))
    plt.subplot(121)
    plt.imshow(cv2.normalize(filtered_disparity, None, 0, 255, cv2.
    NORM_MINMAX), cmap='viridis')
    plt.title('Disparity Map with Edges')
    plt.colorbar()

    plt.subplot(122)
    plt.imshow(cv2.Canny(np.uint8(filtered_disparity), 50, 150), cmap=
    'gray')
    plt.title('Edge Map')
    plt.show()
```

The **disparity map** is generated by finding matching points between images. Mathematically, this involves solving systems of equations, and the result is a map that gives depth information for each pixel.

```
#4 NeRF Depth Estimation

# Set paths to the images
image1_path = images_path + 'Image3.jpg'
image2_path = images_path + 'Image6.jpg'

# Check if files exist
if not os.path.exists(image1_path):
    print(f"Error: {image1_path} does not exist.")
if not os.path.exists(image2_path):
    print(f"Error: {image2_path} does not exist.")

def preprocess_images(img1, img2):
    if len(img1.shape) == 3:  # Convert to grayscale if not already
        img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
        img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    # Enhance contrast using CLAHE
    clahe = cv2.createCLAHE(clipLimit=4.0, tileGridSize=(8, 8))
    img1 = clahe.apply(img1)
    img2 = clahe.apply(img2)

    # Apply bilateral filtering for noise reduction
    img1 = cv2.bilateralFilter(img1, 11, 75, 75)
    img2 = cv2.bilateralFilter(img2, 11, 75, 75)

    return img1, img2

def create_stereo_matcher():
    window_size = 9
    min_disp = -32
    num_disp = 256

    stereo = cv2.StereoSGBM_create(
        minDisparity=min_disp,
        numDisparities=num_disp,
        blockSize=window_size,
        P1=8 * 3 * window_size ** 2,
        P2=32 * 3 * window_size ** 2,
        disp12MaxDiff=1,
        uniquenessRatio=5,
        speckleWindowSize=200,
        speckleRange=2,
        preFilterCap=63,
        mode=cv2.STEREO_SGBM_MODE_HH
    )
    return stereo

```

```
# Compute disparity map
disparity = stereo.compute(processed_left, processed_right)

# Check if disparity map is valid
if disparity is None:
    print("Error: Failed to compute disparity map.")
else:
    # Post-process disparity map
    filtered_disparity = post_process_disparity(disparity)

    # Normalize disparity for visualization
    depth_map = cv2.normalize(filtered_disparity, None, 0, 255, cv2.NORM_MINMAX)
    depth_map = np.uint8(depth_map)  # Ensure valid image type for plotting

    # Visualize the depth estimation
    plt.figure(figsize=(10, 8))
    plt.imshow(depth_map, cmap='plasma')  # Use the "plasma" colormap for depth
    plt.colorbar(label='Depth (relative)')
    plt.title('Depth Estimation Map')
    plt.show()
```

3.6 SOTA Results

The reconstructed points and splatted grid are visualized, and the results are saved.

```
# Visualization
def visualize(points_3d, splatted_grid=None):
    fig = plt.figure(figsize=(15, 5))
    ax1 = fig.add_subplot(121, projection='3d')
    ax1.scatter(points_3d[:, 0], points_3d[:, 1], points_3d[:, 2], c='blue', s=1)
    ax1.set_title("Raw 3D Points")
    if splatted_grid is not None:
        ax2 = fig.add_subplot(122, projection='3d')
        points = np.argwhere(splatted_grid > splatted_grid.mean())
        ax2.scatter(points[:, 0], points[:, 1], points[:, 2], c='green', s=1)
        ax2.set_title("Gaussian Splatted Points")
    plt.show()
```

Figures:

Figure 11: Top feature matches between two consecutive images.

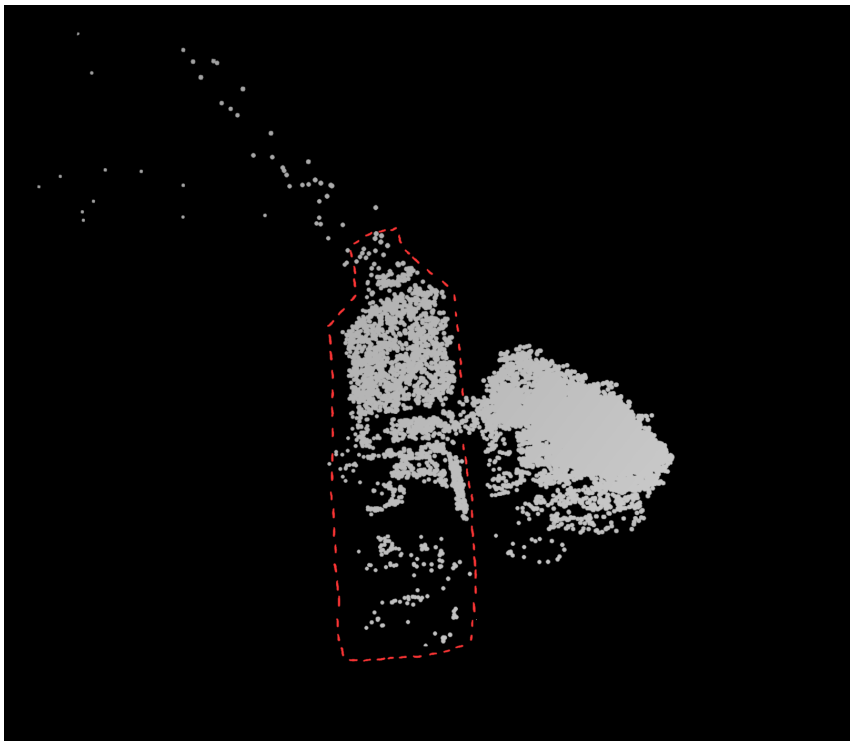
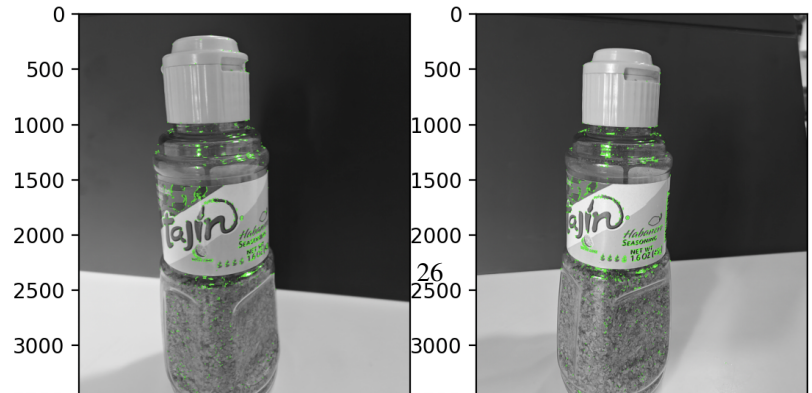


Figure 12: Gaussian point cloud.



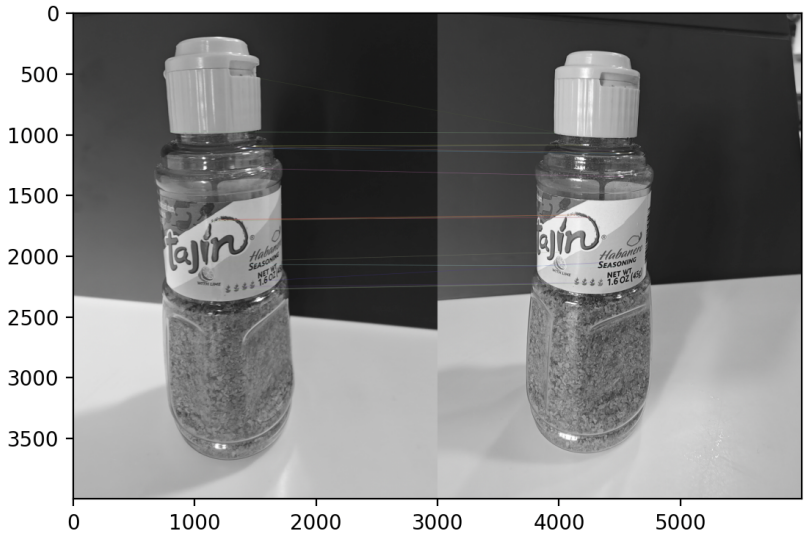


Figure 14: Feature Matching using NeRF

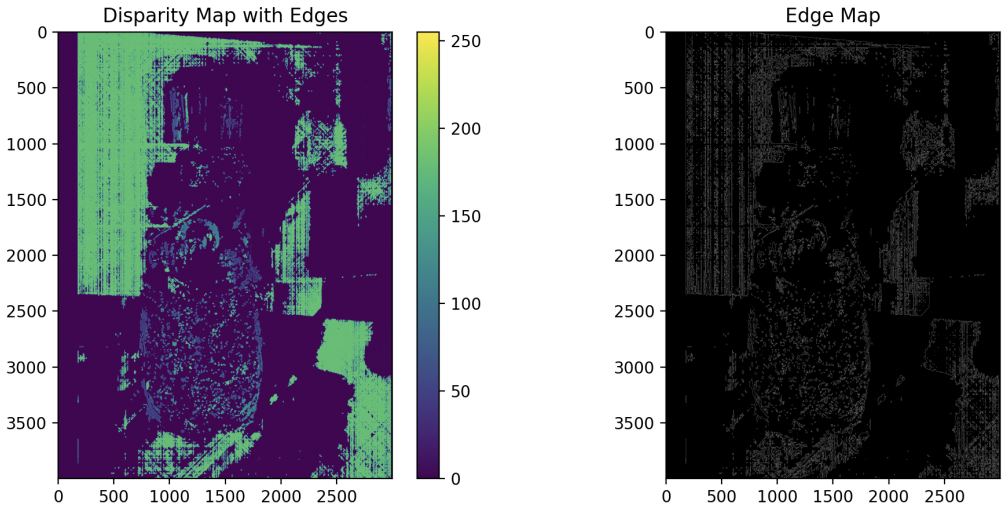


Figure 15: Disparity and Edge Map

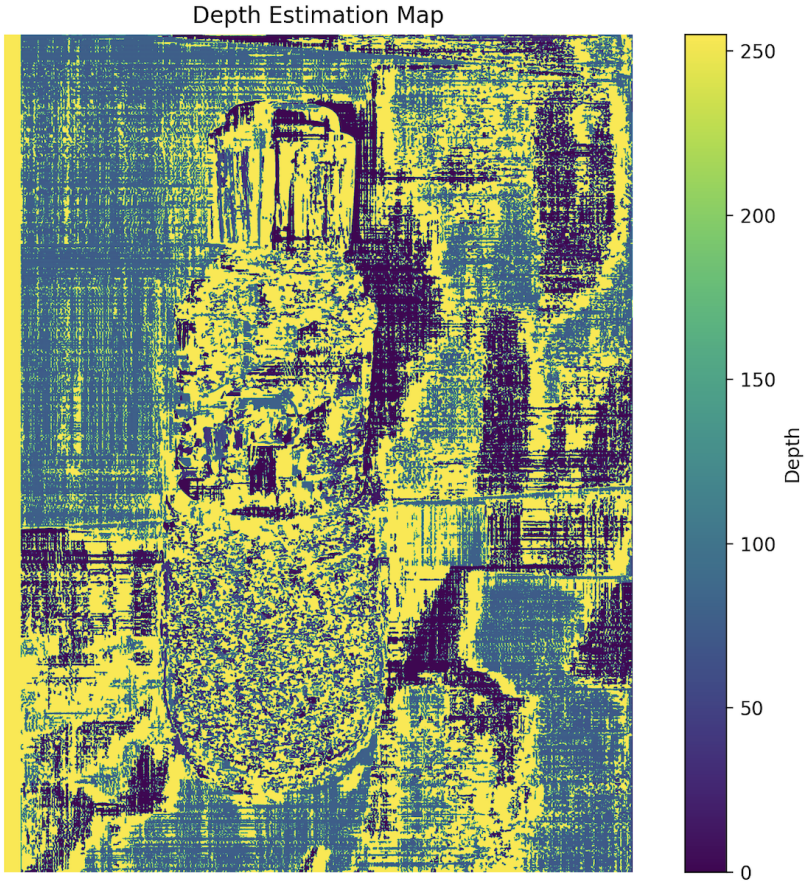


Figure 16: Depth Map using NeRF

3.7 Compare and contrast

Comparing results from the Linear Algebra approach and NeRF implementation:

Figure 17: Feature Detection using NLA(SIFT)

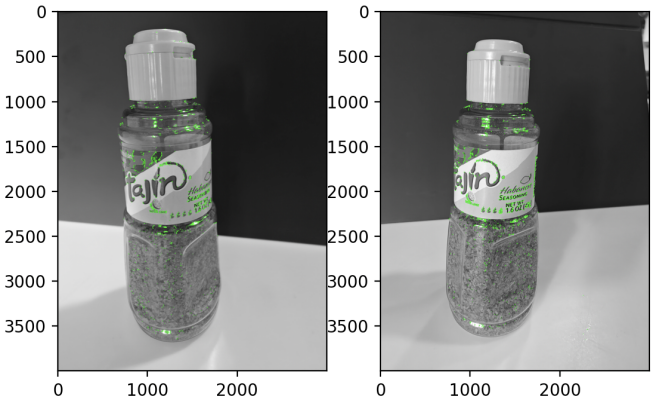


Figure 18: Feature Detection using NeRF(ORB)

Figure 19: Feature Matching using NLA

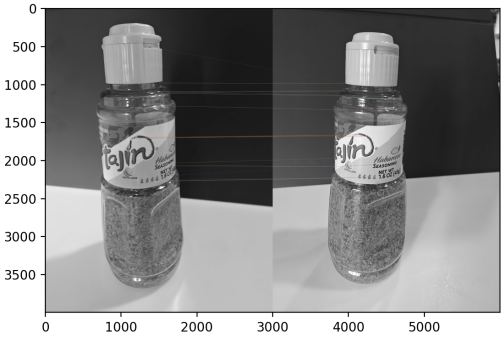


Figure 20: Feature Matching using NeRF

Figure 21: NLA: Disparity

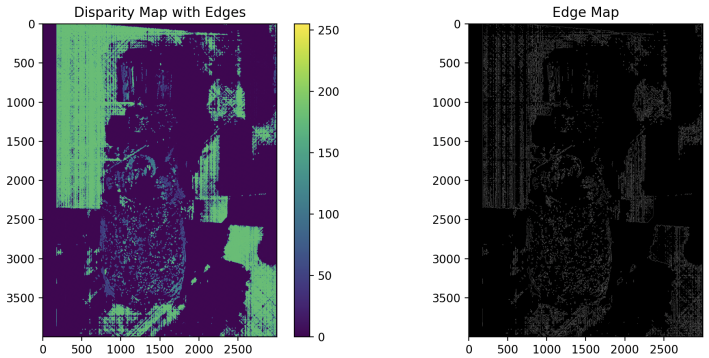


Figure 22: NeRF: Disparity and Edge Map

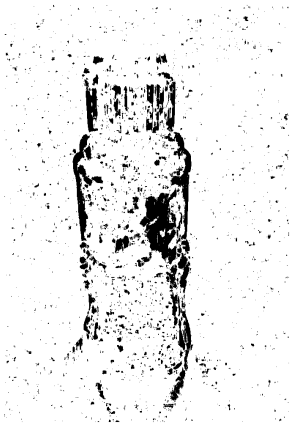


Figure 23: NLA Depth Estimation

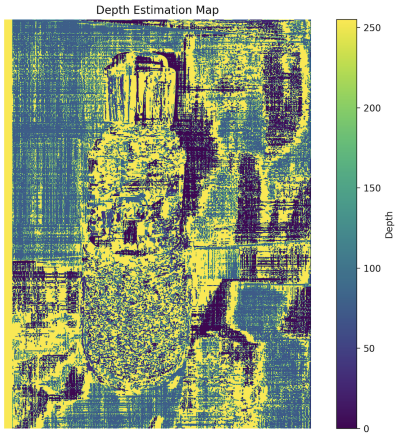


Figure 24: Depth Map using NeRF

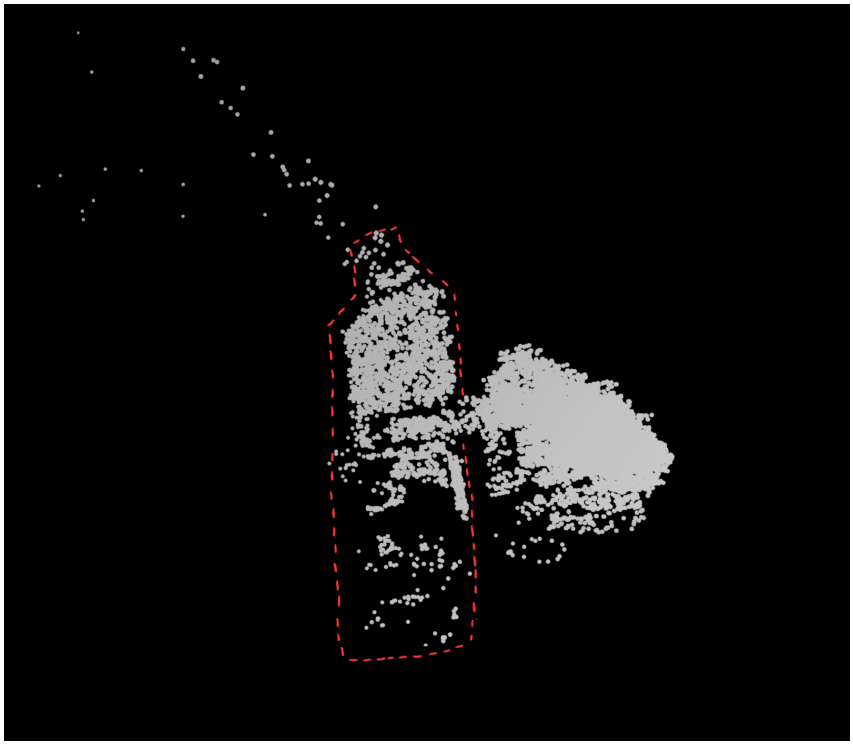


Figure 25: Gaussian point cloud

4 Conclusion

This project focuses on 3D reconstruction using numerical linear algebra techniques, Gaussian splitting, and Neural Radiance Fields (NeRF).

- We captured images to create a custom dataset for the project.
- Libraries/Tools used include NumPy, SciPy, Matplotlib, Plotly, Open3d OpenCV, COLMAP and executed our code on Google Colab.

Feature Matching:

- NLA approach shows varying line transparencies and densities.
- NLA method emphasizes surface textures and reflective properties.
- NeRF visualization maintains clear and distinct matching lines.
- NeRF approach focuses on geometric accuracy and form definition.

Feature Matching:

- NLA approach shows varying line transparencies and densities.
- NLA method emphasizes surface textures and reflective properties.
- NeRF visualization maintains clear and distinct matching lines.
- NeRF approach focuses on geometric accuracy and form definition.

Disparity representation:

- NLA map appears more discrete in its depth representation.
- NLA shows more noise and less defined edges.
- NeRF map shows smoother gradients and better depth transitions.
- NeRF provides clearer object boundaries and more consistent depth values.

Depth representation:

- NLA approach shows a binary-like segmentation with sharp contrast.
- NLA result shows some scattered noise elements that appear as isolated black spots.
- NeRF method displays a continuous depth gradient that offers more depth transitions.
- NeRF visualization exhibits a more structured noise pattern.

5 Acknowledgement

We are really grateful to Prof. Tsui-Wei Weng, Halıcıoğlu Data Science Institute, San Diego for her continuous support, encouragement, and willingness to help us throughout this project.

References

[1] G. Chen, W. Liu, H. Ling, and J. Dai. A survey on 3d gaussian splatting. *arXiv preprint arXiv:2401.03890*, 2024.

[2] R. Hartley and A. Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[3] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4), July 2023.

[4] Y. Mei, Y. Fan, Y. Zhou, L. Xu, M. Nießner, and S. Wu. Reference-based controllable scene stylization with gaussian splatting. *arXiv preprint arXiv:2407.07220*, 2024.

[5] A. Patranabis, S. Banerjee, S. Chakraborty, and A. Roy. A generalized framework for bayesian data association using poisson multi-bernoulli mixture filters. *arXiv*, 2024.

[6] R. Szeliski. *Computer vision: algorithms and applications*. Springer Nature, 2022.

[7] J. Tang, J. Ren, H. Zhou, Z. Liu, and G. Zeng. Dreamgaussian: Generative gaussian splatting for efficient 3d content creation. In *International Conference on Learning Representations*, 2024.

[8] Z. Zhang. Determining the epipolar geometry and its uncertainty: A review. *International journal of computer vision*, 27:161–195, 1998.

[5] [1] [2] [3] [4] [5] [6] [7] [8]