Final Report: Structure from Motion and NeRF

3D Data Analysis

Lastname and Name	Student ID
1. Turriate Llallire, Jorge Victor	**20235603**

1. Structure from Motion

I.1 General description

Structure from Motion (SfM) is a computer vision technique used to reconstruct 3D structures from a set of 2D images taken from different viewpoints. It estimates the camera poses and 3D coordinates of scene points by identifying and matching key features across images. The process involves several steps, including camera calibration, feature detection, feature matching, epipolar geometry estimation, camera pose estimation, triangulation, and bundle adjustment to refine the reconstruction. SfM is widely used in applications such as 3D scanning, robotics, and augmented reality.

I.2 Dataset description and Object selection:

For this project, I captured 20 images using my iPhone 14 Pro from different viewpoints around the selected object to ensure complete and accurate 3D reconstruction. The images were taken under consistent lighting conditions, avoiding reflections and excessive shadows that could interfere with feature detection and matching. The camera's intrinsic parameters were estimated through a calibration process, ensuring accurate reprojection of 3D points.

I selected a laptop support as the object for reconstruction because of its distinct geometric features. The support has a ramp-like shape with an "X" pattern on the slanted side, which provides well-defined edges and texture variations that help improve feature detection and matching. Compared to other objects I initially tested, the laptop support produced better reconstruction results, as it minimized noisy data and avoided complications associated with objects having reflective surfaces or highly intricate structures.

Fig.1 Laptop Support

This selection was crucial for achieving a robust Structure from Motion (SfM) pipeline, as well-defined features enhance the accuracy of camera pose estimation and triangulation, ultimately improving the 3D reconstruction quality.

I.3 Implementation:

Camera Calibration

Camera calibration is a fundamental step in **Structure from Motion (SfM)** that allows us to determine the **intrinsic parameters** of the camera, which are necessary for accurately projecting 3D points onto a 2D image plane. The intrinsic parameters are represented by the **camera matrix (K)**, which includes the focal length and principal point, and the **distortion coefficients**, which model lens distortions that may affect image accuracy.
For this project, I performed **camera calibration** using my **iPhone camera** by capturing multiple images of a **chessboard pattern** from different angles. The chessboard pattern is commonly used for calibration because it provides well-defined **corner points**, which can be easily detected and used to compute the camera parameters.

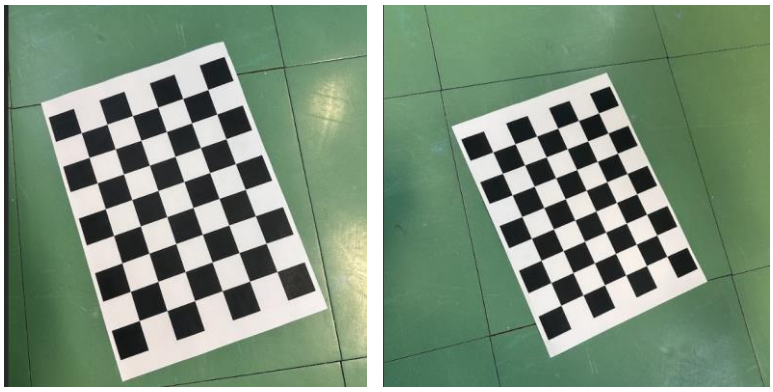


Fig.2 Chessboard used for Camera Calibration

The calibration process was implemented using **OpenCV**, following these steps:

1. **Chessboard Detection**: The program detects the **inner corners** of the chessboard in each image using OpenCV's findChessboardCorners() function. These detected points are then refined using cornerSubPix() to improve accuracy.
2. **Computing Camera Parameters**: Once enough chessboard images are processed, the function calibrateCamera() estimates the **intrinsic camera matrix (K)** and **distortion coefficients** by minimizing the reprojection error.
3. **Saving the Calibration Results**: The computed **K matrix** and **distortion coefficients** were stored and later used throughout the reconstruction process to **correct image distortions** and **improve feature matching and camera pose estimation**.

```
Import cv2
import numpy as np
import os
# Chessboard size (internal corners)
chessboard_size = (8, 6)
square_size = 1  # Realative size (optional)
# 3D points position inside the chessboard
obj_points = np.zeros((chessboard_size[0] * chessboard_size[1], 3), np.float32)
obj_points[:, :2] = np.mgrid[0:chessboard_size[0], 0:chessboard_size[1]].T.reshape(-1, 2)
obj_points *= square_size
# Storing 2D and 3D points
obj_points_list = []  # Real 3D points
img_points_list = []  # 2D Image points
path= "/content/drive/MyDrive/SFM/chess"
```

```
images= os.listdir(path)
for fname in images:
    img = cv2.imread(os.path.join(path, fname))
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Detecta las esquinas del tablero
    ret, corners = cv2.findChessboardCorners(gray, chessboard_size, None)
    if ret:
        obj_points_list.append(obj_points)
        img_points_list.append(corners)
# Calibration
ret, K, dist, rvecs, tvecs = cv2.calibrateCamera(obj_points_list, img_points_list, gray.shape[::-1],
None, None)
print("Intrinsics Matrix (K):\n", K)
print("Distortion coefficients (dist):\n", dist)
```

By calibrating the camera beforehand, we ensure that the 3D reconstruction is geometrically accurate and free from distortions that could degrade the quality of feature detection and matching.

```
Intrinsics Matrix (K):
 [[2.88498063e+03 0.00000000e+00 1.50810456e+03]
 [0.00000000e+00 2.89392408e+03 2.06858771e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion coefficients (dist):
 [[ 7.38983902e-02  1.44364069e+00  2.30518419e-03  9.60029464e-04
  -8.80069551e+00]]
```

Fig.3 Intrinsics matrix (K) and Distortion coefficients

Feature Extraction

Feature extraction is a crucial step in Structure from Motion (SfM), as it allows us to identify key points and descriptors in the images that will be used for matching and 3D reconstruction. The goal is to detect distinctive features in each image that are invariant to changes in scale, rotation, and illumination, ensuring reliable correspondences between multiple views. The most used Feature Extraction algorithms are:

- FAST (Features from Accelerated Segment Test): A high-speed corner detector but does not compute descriptors.
- ORB (Oriented FAST and Rotated BRIEF): A fast alternative to SIFT, but less robust for complex texture variations.
- SURF (Speeded-Up Robust Features): A faster version of SIFT with similar performance but is patented.
- SIFT (Scale-Invariant Feature Transform): One of the most robust feature detectors, invariant to scale, rotation, and illumination changes.

In our implementation, we used OpenCV's SIFT detector to extract features from all images. We created the following function to perform the Feature extraction:

```
def extract_features(config,dict_cameras):
    '''
    extract features
    '''
    t0=time.time()
```

```
for I in range(config.n_cameras):
    img=dict_cameras[i]['img']
    kp, des = config.detector.detectAndCompute(img,None)
    dict_cameras[i]['kp'],dict_cameras[i]['des']=kp,des
t1=time.time()
print('Feature detection takes %d seconds' % (t1-t0))
return dict_cameras
```

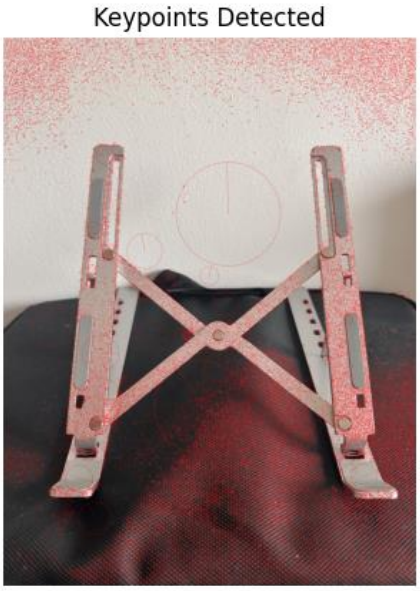


Fig.4 Feature extraction using SIFT detector

We can notice that we got some feature key points on the wall, it happened because our wall had certain points in its relief that caused this detection.

Feature Matching

Once features are extracted from each image, the next step is to establish correspondences between features in different images. This process, known as feature matching, is essential for estimating camera poses and performing 3D reconstruction. Feature matching algorithms compare descriptors extracted from different images and find the best correspondences based on similarity measures. Common matching algorithms include:

- Brute-Force Matcher (BFMatcher)
    - This method compares each descriptor in one image with all descriptors in the other image using a chosen distance metric (e.g., Euclidean distance for SIFT).
    - It is simple and effective but computationally expensive, especially for large datasets.
- FLANN (Fast Library for Approximate Nearest Neighbors)
    - A more efficient alternative to brute-force matching.
    - Uses k-d trees and hierarchical clustering to approximate the nearest neighbors of each descriptor.
    - Faster than brute-force, making it suitable for large datasets.

In our implementation, we used FLANN-based matching to efficiently find correspondences between images.

Fig.5 FLANN-based matching

However, even after filtering with Lowe's Ratio Test, some incorrect matches may still exist due to **occlusions, reflections, or noise**. To refine the matches further, we use **RANSAC (Random Sample Consensus)**:

1. **Estimate a Transformation Model**:
    - We estimate the **Essential Matrix** or **Homography** using a subset of the matched points.
2. **Iteratively Improve the Model**:
    - We check which points fit the model well (inliers) and discard outliers.
3. **Keep the Best Model**:
    - The best model with the most inliers is selected, ensuring robust feature correspondences.

By using **FLANN-based matching with Lowe's Ratio Test and RANSAC**, we obtain reliable feature correspondences, which are crucial for accurate camera pose estimation and 3D reconstruction.

Fig.6 Outlier rejection with RANSAC

Essential Matrix Calculation

Once we have established feature correspondences between images, we need to determine the geometric relationship between two views. This is done using the **Fundamental Matrix (F)** and the **Essential Matrix I**, which encode constraints on corresponding points between two images.

Fundamental Matrix (F)
The **Fundamental Matrix (F)** is a **3×3 matrix** that relates points in one image to their corresponding points in another image. If **x** and **x'** are corresponding points in two images, they satisfy the **epipolar constraint**:

$$x'^T.F.x = 0$$

Where:

- x and x′ are homogeneous coordinates of the points in the two images.
- F is the **Fundamental Matrix**, which depends only on the **relative motion** between cameras and is independent of intrinsic camera parameters.

In our implementation, we estimate **F** using the **RANSAC algorithm**, which robustly filters out outlier matches and ensures a more accurate estimate.

```
# compute F
F, mask = cv.findFundamentalMat(pts1,pts2,cv.FM_RANSAC,1,0.99999)
# filter the outliers
pts1 = pts1[mask.ravel()==1]
pts2 = pts2[mask.ravel()==1]
```

Essential Matrix I
The **Essential Matrix I** is a **scaled version** of the Fundamental Matrix that incorporates the **camera intrinsic parameters (K)**:

$$E = K^T.F.K$$

where:

- K is the **camera intrinsic matrix** (obtained from **camera calibration**).
- F is the **Fundamental Matrix** estimated previously.
- E contains information about the **relative rotation I and translation (t)** between the two camera views.

In our implementation, we computed **E** using:

```
# get Essential matrix from calibration matrix and fundamental matrix
E=np.dot(np.dot(config.K.T,F),config.K)
```

Camera Pose Estimation

Once we have computed the Essential Matrix I, the next step is to estimate the relative camera pose, which consists of the rotation matrix I and translation vector (t) between two views.

Decomposing the Essential Matrix to Obtain (R, t)
The Essential Matrix I encodes the relative motion of the camera but does not directly provide the camera pose. To extract the rotation I and translation (t), we perform a Singular Value Decomposition (SVD) of E:

$$E = U.\Sigma.V^T$$

Where:

- U and V^T are orthogonal matrices.
- Σ is a diagonal matrix with singular values.

Using **SVD**, we obtain two possible rotation matrices **R** and two possible translation vectors **t**, leading to four potential solutions.

$$\begin{cases} C_1 = U(:3), & R_1 = U\Sigma V^T \\ C_2 = -U(:3), & R_2 = U\Sigma V^T \\ C_3 = U(:3), & R_3 = U\Sigma^T V^T \\ C_4 = -U(:3), & R_4 = U\Sigma^T V^T \end{cases}$$

Correction for Invalid Rotation Matrices: If the determinant of R is -1, the pose must be corrected as: C = -C, R = -R

We choose the correct solution by ensuring that the reconstructed 3D points are in front of both cameras (i.e., they have positive depth).
In our implementation, we use OpenCV's recoverPose function, which simplifies this process:

```
_, R, t, mask = cv2.recoverPose(E, pts1, pts2, K)
```

Triangulation

Once we have estimated the camera poses from the Essential Matrix, the next step is to compute the 3D coordinates of the matched feature points from two views. This process is known as **triangulation**, which reconstructs the 3D structure of the scene by finding the intersection of the back-projected rays from the camera centers through the matched 2D feature points.
We would check the **chirality condition** (ensuring that the triangulated 3D points have positive depth in both views and lie in front of cameras) to choose the correct camera pose.

Given a point x_1 in the first image and its corresponding point x_2 in the second image, we know their **homogeneous coordinates** $x_1 = (u_1, v_1, 1)$ and $x_2 = (u_2, v_2, 1)$

Each camera has a **projection matrix**:

- The first camera is at the **origin**, so its projection matrix is: P_1=K·[I|0]
- The second camera has a **relative rotation I and translation (t)**, so its projection matrix is: P_2=K·[R|t]

Using these two projection matrices, the 3D point X=(X,Y,Z,1) must satisfy:

$$x_1 = P_1.X,\ x_2 = P_2.X$$

This gives us a system of linear equations, which we solve using **Singular Value Decomposition (SVD)** to obtain the best 3D point estimate.

```
Def linearTriangulation(P1,x1s,P2,x2s,K):
    '''
    Given two projection matrice and calibrated image points, triangulate them to get 3-D points
    and also get the reprojection error [pixel]
    Input:
        P1:     4 x 4
        P2:     4 x 4
        x1s:    N x 3
        x2s:    N x 3
```

```
        K:      3 x 3
    Output:
        XS:     4 x N
        error:  N x 4
    '''
    XS=np.zeros((4,x1s.shape[0]))
    for k in range(x1s.shape[0]):
        r1=x1s[k,0]*P1[2,:]-P1[0,:]
        r2=x1s[k,1]*P1[2,:]-P1[1,:]
        r3=x2s[k,0]*P2[2,:]-P2[0,:]
        r4=x2s[k,1]*P2[2,:]-P2[1,:]
        A=np.vstack((r1,r2,r3,r4))
        _,_,Vh=np.linalg.svd(A)
        XS[:,k]=Vh.T[:,-1]/Vh.T[3,3]
    # get the reprojection errors
    error_1=reprojection_error(XS,x1s,P1,K)
    error_2=reprojection_error(XS,x2s,P2,K)
    error=np.hstack((error_1,error_2))
    return XS,error
```

Bundle Adjustment

After obtaining the initial 3D structure through triangulation, the next step is to refine both the 3D points and the camera parameters to minimize the reprojection error. This optimization process is called Bundle Adjustment (BA).

Triangulation provides an initial estimate of the 3D structure, but this estimate may contain errors due to:

- Imprecise feature matching
- Small errors in camera pose estimation
- Noise in the image data

Bundle Adjustment corrects these errors by simultaneously optimizing:

1. The 3D coordinates of reconstructed points
2. The camera parameters (rotation, translation, intrinsic matrix)
3. The distortion coefficients (if considered)

The goal is to minimize the reprojection error, which is defined as:

$$\sum_{i,j} ||x_{ij} - P_i X_j||^2$$

where:

- x_{ij} is the observed 2D point in image i
- P_i is the projection matrix of camera iii
- X_j is the 3D point being observed

In our code, new cameras are added one by one to improve the reconstruction. Each time a new camera is registered, we refine the 3D points and update the list of registered cameras:

```
for I in range(n_cameras-2):
```

```
    print('--------------------------------------------------------------------------------
----------')
    print('Registering %dth camera.....' % (i+3))
    config,dict_cameras,n_observations,n_points_3d=register(config,dict_cameras)
    print('%d observations, %d 3D points' %(n_observations,n_points_3d))
```

After registering a new camera, we filter the observed 3D points to remove duplicates and enhance the reconstruction:

```
    mask=[]
    for crn_camera_index in config.indice_registered_cameras:
        mask.extend(dict_cameras[crn_camera_index]['point_indice'].tolist())
    mask=list(set(mask))
    reconstructed_points_3d=config.reconstructed_points_3d[mask]
```

These filtered points are then saved progressively to txt files:

```
    np.savetxt('results/%s/%d.txt' % (SAVE_NAME,i+3),reconstructed_points_3d,delimiter=';')
```

The register function applied for each camera has the following definition and it performs Bundle adjustment and RansacPNP to refine the points.

```
Def register(config,dict_cameras):
    ###########################################################################
    # determine the next camera to be registered and linked
    ###########################################################################
    index_camera_2d,index_camera_3d,n_pairs=next_camera(config,dict_cameras)
    ###########################################################################
    # ransacPnP
    ###########################################################################
    crn_match=dict_cameras[index_camera_3d]['matches'][index_camera_2d] # match from 3D to 2D
    camera_3d=dict_cameras[index_camera_3d]
    camera_2d=dict_cameras[index_camera_2d]
    camera_param,indice_points_2d,indice_points_3d =PnP(config,crn_match,camera_2d,camera_3d)
    # update the parameters of new camera
    dict_cameras[index_camera_2d]['camera']=camera_param
    dict_cameras[index_camera_2d]['indice_registered_2d']=indice_points_2d.flatten()
    dict_cameras[index_camera_2d]['point_indice']=indice_points_3d.flatten()
    print('Use %d out of %d points to register camera %d by camera %d' %
(len(indice_points_2d),n_pairs,index_camera_2d,index_camera_3d))
    ###########################################################################
    # reconstruct 3d points AND link new observations with all the other images
    ###########################################################################
    P1=recover_projection_matrix(camera_param)
    tmp=config.indice_registered_cameras
    for index_camera in tmp:
        crn_match=dict_cameras[index_camera]['matches'][index_camera_2d]
        P2=recover_projection_matrix(dict_cameras[index_camera]['camera'])
        camera_new=dict_cameras[index_camera_2d]
        camera_exist=dict_cameras[index_camera]
        dict_cameras[index_camera_2d],dict_cameras[index_camera],config=reconstruct_3d_and_link_2d(config,crn_match,P1
,P2,camera_new,camera_exist)
```

```
    ####################################################
    # run BA
    ####################################################
    config.indice_registered_cameras.append(index_camera_2d)
    n_cameras=len(config.indice_registered_cameras)
    n_points=config.reconstructed_points_3d.shape[0]
    # points_3d, n_points x 3
    points_3d=config.reconstructed_points_3d
    # get camera params,points_2d, camera_indice
    camera_params=np.zeros((n_cameras,6))
    camera_indices=[]
    point_indices=[]
    points_2d=[]
    for j in range(n_cameras):
        crn_camera_index=config.indice_registered_cameras[j]
        camera_params[j,:]=dict_cameras[crn_camera_index]['camera']
        camera_indices.extend([j]*dict_cameras[crn_camera_index]['indice_registered_2d'].shape[0])
        point_indices.extend(dict_cameras[crn_camera_index]['point_indice'].tolist())
        for ele in dict_cameras[crn_camera_index]['indice_registered_2d']:
            points_2d.append(dict_cameras[crn_camera_index]['kp'][ele].pt)
    points_2d=np.float64(points_2d)
    camera_indices=np.array(camera_indices)
    point_indices=np.array(point_indices)
    # calibrate the 2d image pts to calculate residual
    if(points_2d.shape[1]==2):
        points_2d=np.hstack((points_2d,np.ones((points_2d.shape[0],1))))
    points_2d=np.dot(np.linalg.inv(config.K),points_2d.T).T
    points_2d=points_2d[:,:2]
    # optimize
    print("Camara Params")
    print(camera_params)
    print(camera_params.shape)
    x0 = np.hstack((camera_params.ravel(), points_3d.ravel()))
    #print(np.atleast_1d(fun(x0,n_cameras, n_points, camera_indices, point_indices, points_2d,config.K)))
    A = bundle_adjustment_sparsity(n_cameras, n_points, camera_indices, point_indices)
    #f0 = fun(x0, n_cameras, n_points, camera_indices, point_indices, points_2d, config.K)
    #print("Initial residuals:", f0)
    res = least_squares(fun, x0,jac='3-point',jac_sparsity=A, verbose=1, x_scale='jac', ftol=1e-5,
method='trf',loss='soft_l1',
                        args=(n_cameras, n_points, camera_indices, point_indices, points_2d,config.K))
    ###############################################################################
    # filter the big outliers and distant 3D points by  irs. ng the camera dictionary
    ###############################################################################
    optimized_params=res.x
    camera_params=optimized_params[:n_cameras*6].reshape((n_cameras,6))
    print("Camera_params after optimization")
    print(camera_params)
    print(camera_params.shape)
```

```
    points_3d=optimized_params[n_cameras*6:].reshape((n_points,3))
    # filter by reprojection error
    ff=np.abs(res.fun.reshape((-1,2)))
    ii_inlier=(ff<config.post_threshold).sum(1)==2
    # filter by the distance to the origin
    pts_3d=points_3d[point_indices]
    jj_inlier=np.linalg.norm(pts_3d-config.ORIGIN,axis=1)<config.dist_threshold
    mask=np.logical_and(ii_inlier,jj_inlier)
    # filter if there's only 1 support FOR 3D point
    left_point_indices=point_indices[mask]
    count_support=dict(Counter(left_point_indices.tolist()))
    remove_indice=[]
    for key,value in count_support.items():
        if(value==1):
            remove_indice.append(key)
    ii_mask=[True if ele not in remove_indice else False for ele in point_indices]
    mask=np.logical_and(mask,ii_mask)
    # update camera dictionary
    n_observations=mask.sum()
    n_points_3d=len(list(set(point_indices[mask])))
    count=0
    for j in range(n_cameras):
        crn_camera_index=config.indice_registered_cameras[j]
        n_eles=dict_cameras[crn_camera_index]['indice_registered_2d'].shape[0]
        ii_mask=mask[count:count+n_eles]
        count+=n_eles
        dict_cameras[crn_camera_index]['indice_registered_2d']=dict_cameras[crn_camera_index]['indice_registered_2d'][
ii_mask]
        dict_cameras[crn_camera_index]['point_indice']=dict_cameras[crn_camera_index]['point_indice'][ii_mask]
        dict_cameras[crn_camera_index]['camera']=camera_params[j,:]
    config.reconstructed_points_3d=points_3d


    return config,dict_cameras,n_observations,n_points_3d
```

Running the simulation

To reconstruct the 3D structure from multiple images, we follow a structured pipeline that processes each step of Structure from Motion (SfM). The main steps include:

1. Loading images: Reading the 20 photos taken with the iPhone.
2. Extracting features: Detecting keypoints in each image using the SIFT detector.
3. Matching features: Using FLANN matcher to establish correspondences between image pairs.
4. Estimating the Essential Matrix: Using RANSAC to filter outliers and estimate the camera motion.
5. Determining Camera Poses: Computing the relative camera positions from the Essential Matrix.
6. Triangulating 3D Points: Recovering 3D positions of matched points by triangulation.
7. Bundle Adjustment: Refining the 3D structure and camera parameters to minimize reprojection errors.

The simulation runs iteratively, progressively registering new cameras and updating the 3D structure at each step.
Our approach orders the frames that have stronger correspondences, guiding the selection of the best image pairs for initializing the reconstruction. To do that, we plot the next matrix which represents the number of matched features between each pair of images.

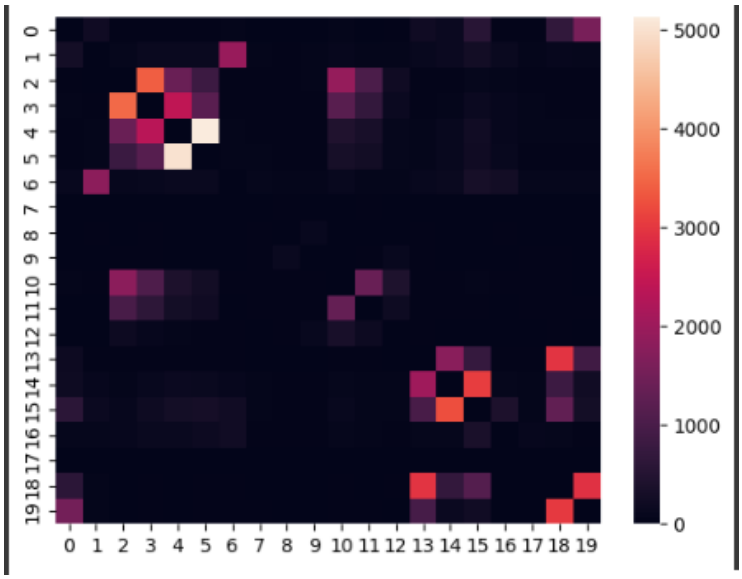


Fig.7 Number of matched features

Finally, we get the following 3D reconstruction of our laptop support.

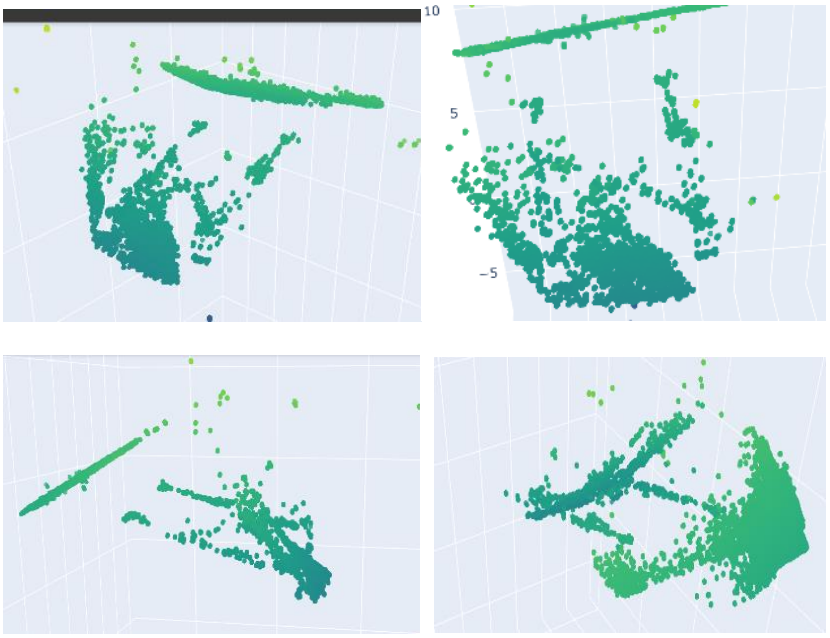


Fig.8 3D reconstruction of our laptop support

II. Neural Radiance Fields (NeRF)

II.1 General description

Neural Radiance Fields (NeRF) represents a 3D scene as a function that maps 3D positions and viewing directions (5D input) to RGB colors and volume densities (4D output). This function is parameterized by a neural network trained to reconstruct images from known viewpoints.

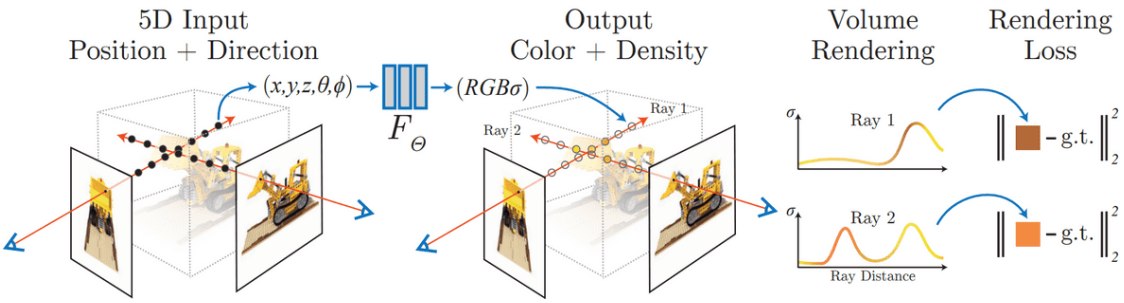


Fig.8 NeRF training process

The process consists of considering camera rays passing through each pixel of the input image, then we will apply volume rendering techniques to project color and densities into the image. Since volume rendering is differentiable, we just need the camera poses of each images (extrinsics).

The pipeline for NeRF is:

1. We create rays through the scene in each pixel to generate a sampled set of 3D points
2. We feed the Neural network with these 3D points (position of rays) and their view directions (2D vector) to produce a set of color and densities.
3. We render these rays by using volume rendering to map the color and densities.

We do this process by using Gradient descent to minimize the Rendering loss which is the error between each observed image and the corresponding views rendered from our representation.

II.2 Dataset: TinyNeRF

For training and evaluating our NeRF model, we used the TinyNeRF dataset, a small-scale dataset designed for quick experimentation with Neural Radiance Fields. This dataset was introduced as part of the official NeRF implementation and is available at: http://cseweb.ucsd.edu/~viscomp/projects/LF/papers/ECCV20/nerf/tiny_nerf_data.npz

To load the dataset, we used the following command:

```
if not os.path.exists('tiny_nerf_data.npz'):
    !wget http://cseweb.ucsd.edu/~viscomp/projects/LF/papers/ECCV20/nerf/tiny_nerf_data.npz
```

This dataset consists of 100 training images and one test image, each captured from different camera viewpoints around a 3D object. The dataset also provides the corresponding camera poses and intrinsic parameters (such as focal length), which are necessary for ray tracing and view synthesis in NeRF. The images have a relatively small resolution of 100×100 pixels, making them computationally efficient for training.

We chose this dataset because:

1. Simplicity: It provides a minimal setup for implementing NeRF without the complexities of large-scale datasets.
2. Computational efficiency: The small image resolution allows us to train the model without requiring high-end GPU resources.
3. Benchmarking: It is widely used for testing NeRF implementations, making it a good reference for evaluating model performance.

This dataset allowed us to validate our implementation and experiment with various aspects of NeRF, such as positional encoding, volumetric rendering, and ray sampling, without excessive computational costs.

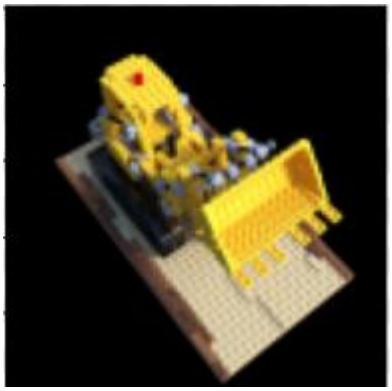


Fig.9 TinyNeRF dataset

II.3 Implementation:

Positional Encoding

In standard neural networks, inputs such as pixel coordinates or 3D positions are directly fed into the model. However, Neural Radiance Fields (NeRF) suffer from a limitation when using raw coordinates: deep networks struggle to learn high-frequency details from low-dimensional inputs. To overcome this, NeRF employs Positional Encoding (PE), a technique inspired by the Fourier feature mapping, which transforms low-dimensional inputs into a higher-dimensional space with sinusoidal functions.

Given an input coordinate x, we map it to a higher-dimensional space using sine and cosine functions at different frequencies:

$$\gamma(x) = (\sin(2^0\pi x), \cos(2^0\pi x), \sin(2^1\pi x), \cos(2^1\pi x), \dots, \sin(2^{L-1}\pi x), \cos(2^{L-1}\pi x))$$

where L is the number of frequency bands used for encoding.

Positional encoding is used in NeRF mainly for these 2 reasons:

- **Capturing High-Frequency Details**: PE enables the network to represent fine details in textures and lighting variations. Without PE, NeRF models tend to blur high-frequency signals.
- **Overcoming the Spectral Bias**: Neural networks naturally prefer learning low-frequency functions (smooth variations). PE helps balance this bias, allowing NeRF to reconstruct sharp details.

```
Def encoding(x, L=10):
    rets = [x]
    for I in range(L):
```

```
        for fn in [torch.sin, torch.cos]:  # Use torch functions
            rets.append(fn(2. ** I * x))
    return torch.cat(rets, dim=-1)  # Concatenate along the last dimension
```

NeRF Model

The NeRF Model is a MLP designed to reconstruct a 3D scene from a set of 2D images taken from different viewpoints. It works by learning a continuous function that maps spatial coordinates and viewing directions to color and density values, enabling the synthesis of novel views.

Input Representation
The input to the model consists of:

1. **3D Positions** (x,y,z) of sampled points in space.
2. **Viewing Directions** (θ,ϕ) representing the camera's perspective.
3. **Positional Encoding** is applied to both inputs before being fed into the network to capture high-frequency details.

Our architecture is represented with the same image:

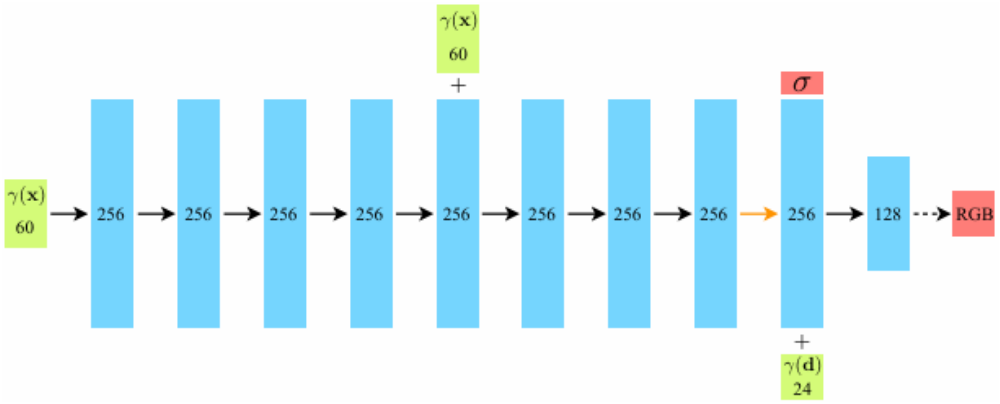


Fig.10 NeRF MLP architecture

We can notice that we use a skip connection to concatenate the original encoding with intermediate features which help preserve spatial information and prevent vanishing gradients and we use separate branches for density and color prediction ensure that the model effectively learns geometry and appearance independently.

```
Class NeRF(nn.Module):


  def __init__(self, pos_enc_dim = 63, view_enc_dim = 27, hidden = 256) -> None:
     super().__init__()
     self.linear1 = nn.Sequential(nn.Linear(pos_enc_dim,hidden),nn.ReLU())
     self.pre_skip_linear = nn.Sequential()
     for _ in range(4):
      self.pre_skip_linear.append(nn.Linear(hidden,hidden))
      self.pre_skip_linear.append(nn.ReLU())
     self.linear_skip = nn.Sequential(nn.Linear(hidden+pos_enc_dim,hidden),nn.ReLU())
     self.post_skip_linear = nn.Sequential()
```

```
    for _ in range(2):
     self.post_skip_linear.append(nn.Linear(hidden,hidden))
     self.post_skip_linear.append(nn.ReLU())
    self.density_layer = nn.Sequential(nn.Linear(hidden,1), nn.ReLU())
    self.linear2 = nn.Linear(hidden,hidden)
    self.color_linear1 = nn.Sequential(nn.Linear(hidden+view_enc_dim,hidden//2),nn.ReLU())
    self.color_linear2 = nn.Sequential(nn.Linear(hidden//2, 3),nn.Sigmoid())
    self.relu = nn.ReLU()
    self.sigmoid = nn.Sigmoid()
def forward(self,input):
  # Extract pos and view  irs.
  positions = input[..., :3]
  view_dirs = input[...,3:]
  # Encode
  pos_enc = encoding(positions,L=10)
  view_enc = encoding(view_dirs, L=4)
  x = self.linear1(pos_enc)
  x = self.pre_skip_linear(x)
  # Skip connection
  x = torch.cat([x, pos_enc], dim=-1)
  x = self.linear_skip(x)
  x = self.post_skip_linear(x)
  # Density prediction
  sigma = self.density_layer(x)
  x = self.linear2(x)
  # Incoroporate view encoding
  x = torch.cat([x, view_enc],dim=-1)
  x = self.color_linear1(x)
  # Color Prediction
  rgb = self.color_linear2(x)
  return torch.cat([sigma, rgb], dim=-1)
```

Volume Rendering

This technique is used in NeRF (Neural Radiance Fields) to synthesize novel views of a scene by integrating color and density values along a camera ray. Instead of explicitly representing surfaces as meshes or point clouds, NeRF models a scene as a continuous radiance field, where each sampled 3D point contributes to the final pixel color based on its density and color properties.

To explain it, given a camera ray:

$$r(t) = o + t.d$$

Where: o is the camera origin, d is the normalized direction of the ray, and t is the depth parameter that determines the position of the ray along the ray.

NeRF estimates the color CI of a pixel by integrating the contributions of all sampled points along the ray. This is done using the volume rendering equation:

$$C(r) = \sum_{i=1}^{N} T_i \alpha_i c_i$$

Where:

- c_i is the RGB color at the i-th sample,
- α_i is the transparency derived from the density **σ** using:

  $$\alpha_i = 1 - e^{-\sigma_i \delta_i}$$

  Where δ_i is the distance between adjacent samples
- T_i is the accumulated transmittance, representing the probability that the ray has not been absorbed before reaching the i-th sample:

  $$T_i = \prod_{j=1}^{i-1} (1 - \alpha_j)$$

This equation ensures that closer points contribute more to the final color than distant points that are occluded by dense regions.
We use Volume rendering because:

- Handles occlusions naturally: Unlike traditional surface rendering, volume rendering allows for soft transparency and gradual transitions between objects.
- Enables novel view synthesis: By modeling a continuous 3D field, NeRF can generate realistic perspectives from unseen viewpoints.
- Supports view-dependent effects: Since the model considers viewing angles, it captures complex phenomena like reflections and specular highlights.

```
Def render_rays(network_fn, rays_o, rays_d, near, far, N_samples, device, rand=False, embed_fn=None, chunk=1024*4):
    def batchify(fn, chunk):
        return lambda inputs: torch.cat([fn(inputs[i:i+chunk]) for I in range(0, inputs.shape[0], chunk)], 0)
    # Sampling
    z_vals = torch.linspace(near, far, steps=N_samples, device=device)
    if rand:
        z_vals += torch.rand(*z_vals.shape[:-1], N_samples, device=rays_o.device) * (far - near) / N_samples
    pts = rays_o[...,None,:] + rays_d[...,None,:] * z_vals[...,:,None]
    # Normalize view directions
    view_dirs = rays_d / torch.norm(rays_d, dim=-1, keepdim=True)
    view_dirs = view_dirs[..., None, :].expand(pts.shape)
    input_pts = torch.cat((pts, view_dirs), dim=-1)
    raw = batchify(network_fn, chunk)(input_pts)
    # Apply activations here instead of in network
    sigma_a = raw[...,0]  # Shape: [batch, N_samples]
    rgb = raw[...,1:]    # Shape: [batch, N_samples, 3]
    # Improved volume rendering
    dists = z_vals[..., 1:] - z_vals[..., :-1]  # Shape: [batch, N_samples-1]
    dists = torch.cat([dists, torch.tensor([1e10], device=device)], -1)
    # No need to manually expand dists as broadcasting will handle it
    alpha = 1. - torch.exp(-sigma_a * dists)  # Shape: [batch, N_samples]
    alpha = alpha.unsqueeze(-1)  # Shape: [batch, N_samples, 1]
    # Computing transmittance
    ones_shape = (alpha.shape[0], 1, 1)
    T = torch.cumprod(
        torch.cat([
            torch.ones(ones_shape, device=device),
```

```
            1.   - alpha + 1e-10
        ], dim=1),
        dim=1
    )[:, :-1]  # Shape: [batch, N_samples, 1]
    weights = alpha * T  # Shape: [batch, N_samples, 1]
    # Compute final colors and depths
    rgb_map = torch.sum(weights * rgb, dim=1)  # Sum along sample dimension
    depth_map = torch.sum(weights.squeeze(-1) * z_vals, dim=-1)  # Shape: [batch]
    acc_map = torch.sum(weights.squeeze(-1), dim=-1)  # Shape: [batch]
    return rgb_map, depth_map, acc_map
```

Training

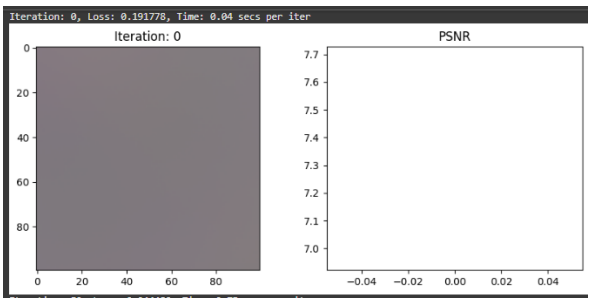
The train function trains a NeRF model using ray sampling and volume rendering to reconstruct 3D scenes. In each iteration, a random training image is selected along with its camera pose, and rays are sampled from the image. The model predicts RGB colors and densities for points along these rays, which are then composited using volume rendering to generate the final pixel colors. The loss is computed against the ground-truth image using Mean Squared Error (MSE), and the model is optimized through backpropagation using the Adam optimizer.
Every 50 iterations, validation is performed by rendering an image from the test pose, computing the PSNR to measure reconstruction quality, and displaying both the generated image and the PSNR curve. A learning rate scheduler is used to gradually reduce the learning rate, improving convergence. The final trained model can be used to synthesize novel views of the scene, ensuring high-quality 3D reconstruction from 2D images.

```
Def train(images,poses,H,W,focal,testpose,testimg,device):
    print(f"Using device: {device}")
    model = NeRF().to(device)
    criterion = nn.MSELoss(reduction='mean')
    optimizer = torch.optim.Adam(model.parameters(),lr=5e-4)
    scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.99)
    n_iter = 1000
    n_samples = 64
    i_plot = 50
    psnrs = []
    iternums = []
    t = time.time()
    # Convert data to tensors and move to device ONCE
    images_tensor = torch.from_numpy(images).float().to(device)
    poses_tensor = torch.from_numpy(poses).float().to(device)
    for I in range(n_iter):
        img_i = np.random.randint(images.shape[0])
        target = images_tensor[img_i]  # Use the corresponding image
        pose = poses_tensor[img_i]     # Use the corresponding pose
        rays_o, rays_d = get_rays(H, W, focal, pose)
        optimizer.zero_grad()
        rgb, depth, acc = render_rays(model, rays_o, rays_d, near=2., far=6., N_samples=n_samples, device=device,
rand=True)
```

```
        rgb = rgb.reshape(H,W,3)
        loss = criterion(rgb, target)
        loss.backward()
        optimizer.step()
        if I % i_plot == 0:
            print(f'Iteration: {i}, Loss: {loss.item():.6f}, Time: {(time.time() - t) / i_plot:.2f} secs per iter')
            t = time.time()
            with torch.no_grad():
                rays_o, rays_d = get_rays(H, W, focal, testpose)
                rgb, depth, acc = render_rays(model, rays_o, rays_d, near=2., far=6.,
                                              N_samples=n_samples, device=device)
                rgb = rgb.reshape(H, W, 3)
                loss = criterion(rgb, testimg)
                psnr = -10. * torch.log10(loss)
                psnrs.append(psnr.item())
                iternums.append(i)
                plt.figure(figsize=(10,4))
                plt.subplot(121)
                plt.imshow(rgb.cpu().detach())
                plt.title(f'Iteration: {i}')
                plt.subplot(122)
                plt.plot(iternums, psnrs)
                plt.title('PSNR')
                plt.show()
    return model
```

Results

After the training was performed, we can see the evolution of the PNSR vs Iteration and how the reconstruction of the views was. Initially, PSNR starts at a low value and quickly rises, indicating that the model is rapidly improving its reconstruction quality in the early training phase.



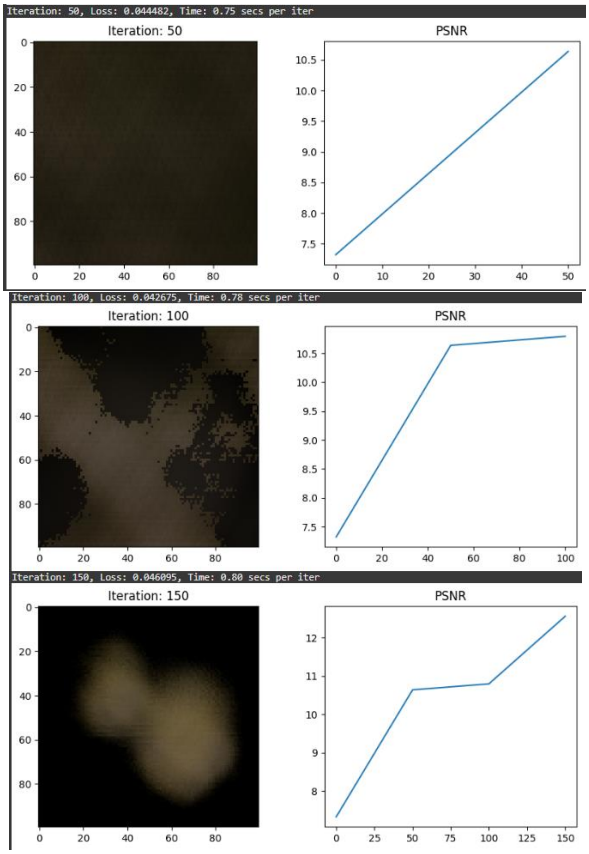


Fig.11 PSNR vs Iterations in the first iterations

As training continues beyond 450 iterations, the PSNR shows a more gradual increase, converging around 25 dB with minor fluctuations. The fluctuations suggest some variation in the model's predictions, possibly due to scene complexity, but the overall trend remains upward and stable. This indicates that the model is converging well, achieving high-quality 3D reconstruction.

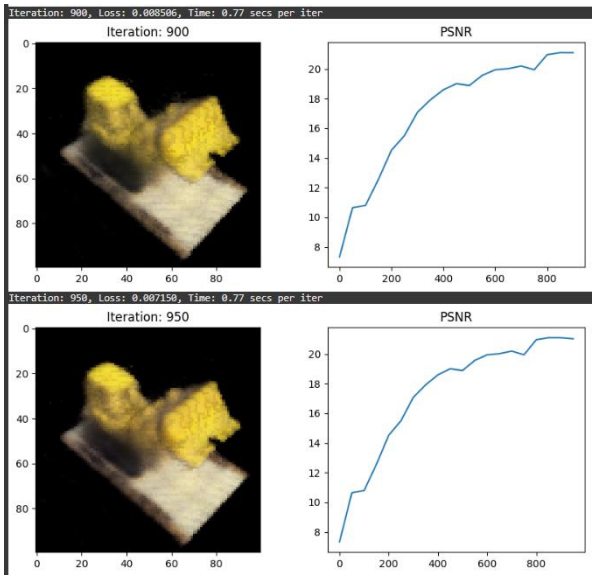


Fig.12 PSNR vs Iterations in the last iterations

Finally, we can render a little video an check how our NeRF was able to reconstruct successfully the 3D Object.

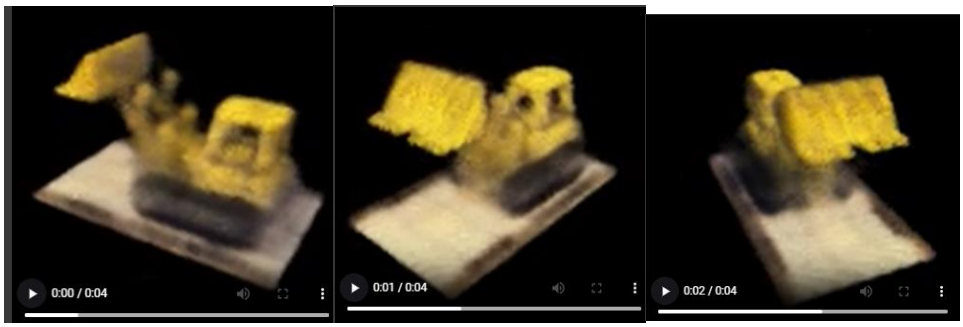


Fig.13 Some novel views generated by our NeRF network

III. Conclusions

- NeRF and SFM are powerful techniques for 3D scene reconstruction, but they differ in approach and application.
- SfM reconstructs a sparse 3D point cloud by estimating camera motion from multiple images using feature detection and triangulation.
- NeRF learns a continuous volumetric representation of a scene by modeling color and density fields, enabling realistic novel view synthesis with fine details and lighting effects.