Project 4: Neural Radiance Fields (NeRF)

Eason Wei | CS 180 / 280A – Fall 2025

Introduction

In this project, I implement a full NeRF pipeline—from capturing a real object with a phone, to calibrating the camera, estimating poses, training a 2D neural field, and finally training a complete 3D Neural Radiance Field capable of synthesizing novel views.

Part 0 — Camera Calibration & 3D Scanning Pipeline

I begin by printing a 6-tag ArUco grid, capturing 50 calibration images, and running `cv2.calibrateCamera` to recover the intrinsic matrix K. I then capture 50 images of my Labubu figure, detect an ArUco tag in each frame, and estimate all camera-to-world poses using the PnP algorithm.

0.1 — Camera Calibration

I printed a 6-tag ArUco grid and captured 50 high-resolution images. Since they were originally in `.heic` format, I batch-converted them into `.jpg` for processing. Each printed marker measured 0.57 units wide, which I used to define the 3D coordinates of all tag corners.

Using `cv2.aruco.detectMarkers`, I detected all visible markers and applied a mask to ensure consistent indexing across frames. Finally, I used `cv2.calibrateCamera` to recover the camera's intrinsic matrix K and distortion coefficients.

0.2 — Object Capture

I captured ~50 images of my Labubu figure beside a single ArUco tag. The camera remained 10–20 cm away, ensuring the object filled roughly half the frame. I kept exposure consistent, avoided motion blur, and varied the angle around a circular path to maximize multi-view coverage.

0.3 — Pose Estimation (PnP)

For each image, I detect the tag, extract its 2D corners, and solve for camera extrinsics using `cv2.solvePnP`. The projection model follows:

$$x_i \sim K[R|t] \begin{bmatrix} X_i \\ 1 \end{bmatrix}$$

The rotation vector is converted using Rodrigues:

$$R = \text{Rodrigues}(\mathbf{r})$$

World-to-camera:

$$\text{w2c} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$$

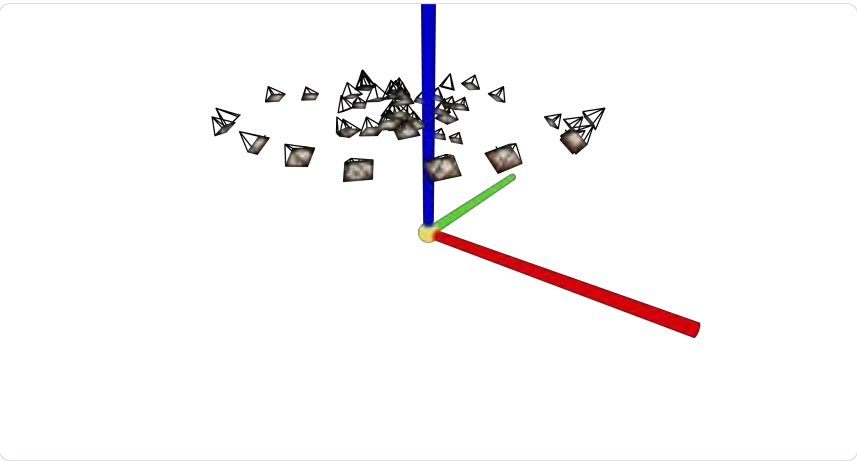
And camera-to-world:

$$\text{c2w} = \begin{bmatrix} R^\top & -R^\top t \\ 0 & 1 \end{bmatrix}$$

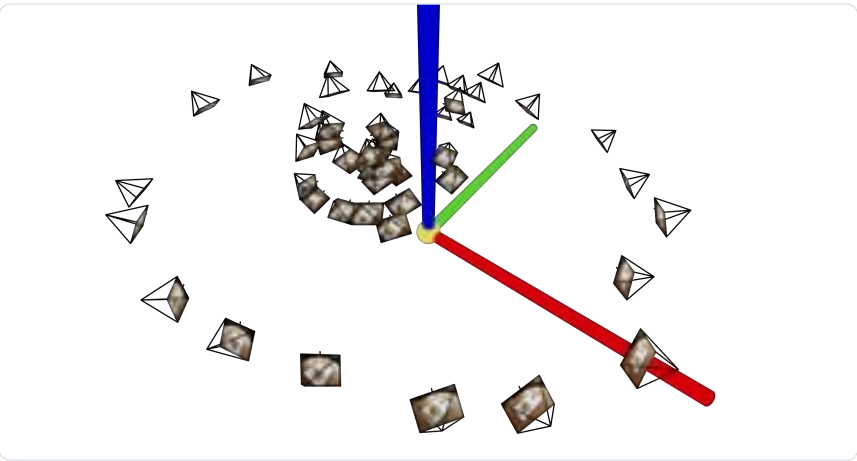
Because NeRF uses a different coordinate convention, I apply a diagonal flip:

$$D = \text{diag}(1, -1, -1)$$

Below are the Viser-rendered camera frustums:



Dome-like coverage — dense sampling of azimuth angles.



Full 360° pose sweep ensures strong multi-view parallax.

0.4 — Dataset Packaging

In this step, I load the 50 resized Labubu images and their camera poses from Part 0.3, then undistort each image using the scaled intrinsic matrix K and the distortion coefficients from the original calibration. This produces a clean, distortion-free set of inputs for NeRF.

I then split the 50 images into **train/val/test** sets using a 70/15/15 ratio. The split is done over image indices with a fixed random seed, and each subset contains both the undistorted RGB images and their corresponding camera-to-world matrices.

The focal length is computed from the resized intrinsics as $f = (K_{00} + K_{11})/2$, and all data—images, poses, intrinsics, and focal—is packaged into `my_data.npz` for NeRF training.

This single file is used for every remaining stage of the NeRF pipeline.

Part 1 — Neural Field for 2D Image Reconstruction

In this part, I train a coordinate-based neural network to reconstruct a 2D image. The model learns a continuous mapping

$$F_\theta(x, y) \to \mathrm{RGB},$$

where each pixel is represented by normalized (x, y) coordinates in $[0, 1]^2$. With positional encoding and a small MLP, the network gradually learns to reproduce the full-resolution image.

Neural Field Architecture

Input: 2D pixel coordinate (x, y)

Positional Encoding: 10 frequency bands (also compared with 2)

MLP:

- Hidden width: 256 (also tested 64)
- Two Linear + ReLU layers
- Final Linear → Sigmoid (RGB)

Loss: MSE **Optimizer:** Adam (lr = 1e−2)

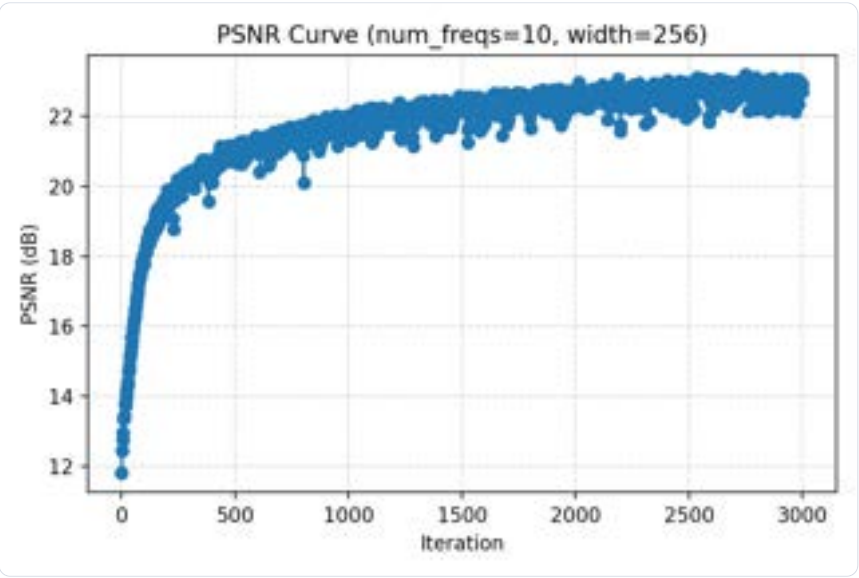
Below are examples of the reconstruction quality at different training iterations.

During training, the MLP learns a continuous mapping from 2D coordinates → RGB values using batches of randomly sampled pixels. At early iterations (e.g., 50–100), the network captures only coarse color blobs because it first fits the low-frequency structure of the image. As training progresses, higher-frequency details emerge: edges become sharper, textures appear, and colors stabilize. By 1000–2000 iterations, the model converges to a smooth, high-fidelity reconstruction that closely matches the original image. This progression highlights how neural fields naturally learn images from coarse-to-fine detail through gradient descent.

The hyperparameter sweep clearly shows how positional frequency and network width affect reconstruction quality. With only **2 frequency bands**, the model cannot represent high-frequency detail, producing overly smooth and blurry results even with a wider MLP. Increasing to **10 frequencies** dramatically sharpens edges and textures, enabling the network to reproduce fine details in the fur and background. Width also matters: the narrow **64-unit** MLP struggles with capacity, introducing graininess and losing subtle shading, while the **256-unit** version produces smoother colors and more faithful contours. Overall, **high frequency + large width** yields the most accurate reconstruction, demonstrating that spatial detail is controlled primarily by positional encoding, while network width governs representational capacity.

reconstruction for flower image

reconstruction using a 2x2 grid of results



psnrs for flower image

Part 2.1 — Create Rays from Cameras

In this step, I convert each pixel in an image into a **3D camera ray** that NeRF can train on. For every pixel coordinate (u, v), I first back-project it through the intrinsic matrix K to obtain a point at unit depth in *camera coordinates*. I then transform this point into *world coordinates* using the camera-to-world matrix `c2w`, which gives me the ray origin (the camera center) and a normalized ray direction. This produces one ray per pixel — the fundamental input representation for NeRF.

$$\mathbf{r}(t) = \mathbf{o} + t\,\mathbf{d}$$

I verify the implementation by round-tripping a test point through `c2w` and its inverse, and I also visualize a small pixel grid's ray directions to ensure they are consistent with the image geometry. These checks confirm that my camera-to-ray conversion is correct.

Part 2.2 — Sampling Points Along Rays

After generating camera rays, the next step is to sample 3D points along each ray so the NeRF MLP can predict color and density at those positions. I first randomly select an image, then randomly choose pixel indices and convert them into ray origins $\mathbf{o}$ and directions $\mathbf{d}$. This produces a batch of rays and their corresponding ground-truth RGB values.

For each ray, I sample **64 points** between a near and far bound (2.0 → 6.0). I use **stratified sampling**, which jitters each interval to produce unbiased Monte-Carlo samples and reduce aliasing. This is the standard NeRF sampling approach.

$$t_i \sim \text{Uniform}(t_{i,\text{lower}},\ t_{i,\text{upper}}), \qquad \mathbf{x}_i = \mathbf{o} + t_i\,\mathbf{d}.$$

The result is a tensor of sampled 3D points of shape $(N, 64, 3)$, and corresponding distances t_i. These samples are later fed into the NeRF network and volume renderer. I also confirm the implementation by printing example rays, directions, and their first few sampled 3D points.

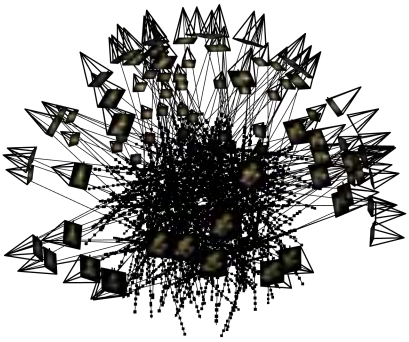
Part 2.3 — Putting the Dataloading All Together

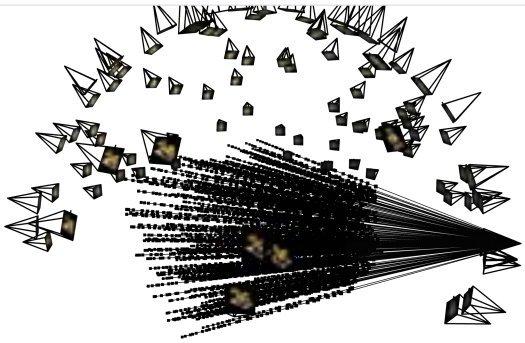
In this step, I build a **RaysData** class that converts the entire training set into a unified ray dataset. For every image, I generate a full grid of pixel centers, convert each pixel into a ray using its camera pose, and flatten all rays across all views into:

$$\text{rays\_o},\ \text{rays\_d},\ \text{pixels} \in \mathbb{R}^{(N\,H\,W)\times 3}.$$

I implemented **two complementary sampling modes**:

- ① **Global Ray Sampling** — I flatten all rays from every training image into one array: `N_images × H × W` rays. A minibatch is created by random indexing, which gives uniform coverage across all viewpoints and is used for the actual NeRF training loop.
- ② **Single-Camera Sampling (for debugging)** — To avoid visual clutter in debugging, I also allow sampling **only from one selected camera pose**. This makes Viser visualizations clean and easy to interpret, since all rays originate from the same location.





Part 2.4 — Neural Radiance Field (NeRF)

I implement a full NeRF model whose goal is to learn a volumetric scene representation that maps 3D world coordinates and viewing directions into **density** and **RGB color**. The model follows the original NeRF architecture: positional encoding on both spatial coordinates and directions, an 8-layer MLP with a skip connection, and separate heads for density and color prediction.

> **NeRF Architecture (My Implementation)**
> - **Positional Encoding:** xyz → 3 + 2×3×L_{xyz} (with L_{xyz} = 10) dir → 3 + 2×3×L_{dir} (with L_{dir} = 4)
> - **MLP Depth:** 8 fully connected layers
> - **Hidden Width:** 256
> - **Skip Connection:** after layer 4 (concatenate xyz-encoding)
> - **Outputs:**
>   - Density σ(x) ∈ ℝ (ReLU)
>   - Color c(x,d) ∈ [0,1]³ (Sigmoid)

Positional Encoding

For both 3D location $x \in \mathbb{R}^3$ and viewing direction $d \in \mathbb{S}^2$, NeRF applies multi-frequency sinusoidal encoding:

$$\gamma(x) = \left[x,\ \sin(2^0\pi x), \cos(2^0\pi x),\ \ldots,\ \sin(2^{L-1}\pi x), \cos(2^{L-1}\pi x)\right].$$

MLP with Skip Connection

The encoded position $\gamma(x)$ flows through the first 4 layers. After layer 4, NeRF concatenates the original $\gamma(x)$ to form a controlled skip connection that preserves high-frequency geometry:

$$h_4 = F_4(\gamma(x)), \qquad h_{\text{skip}} = [h_4,\ \gamma(x)].$$

The following layers compute a feature vector f, then predict:

$$\sigma = \text{ReLU}(W_\sigma h_{\text{skip}}), \qquad f = W_f h_{\text{skip}}.$$

Directional encoding $\gamma(d)$ is concatenated with features to produce RGB:

$$c = \text{Sigmoid}(W_c[f,\ \gamma(d)]).$$

Output

Given many 3D sample points along a ray, NeRF predicts both **volume density** and **view-dependent color**. These outputs feed into the volume rendering equation in Part 2.5 to synthesize pixel colors.

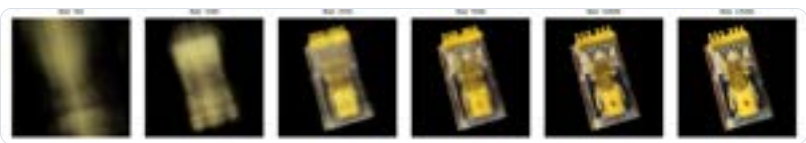
Part 2.5 — Volume Rendering

Once the NeRF MLP predicts **density** σ and **color** $\mathbf{c}$ for every sampled 3D point, the next step is to convert these values into a final pixel color using **volume rendering**. For each ray, I accumulate contributions from all samples using alpha compositing, which models how light is absorbed and emitted along the ray.

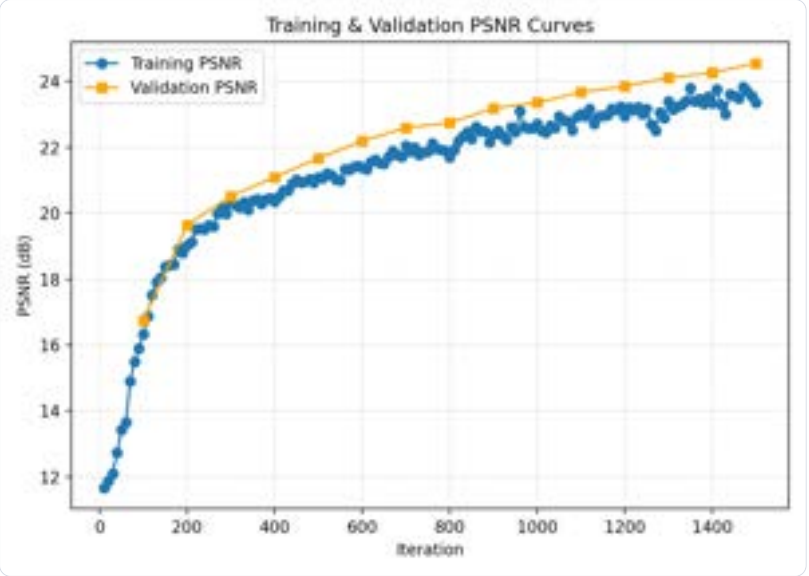
I implement the standard volume rendering equation:

$$C(\mathbf{r}) = \sum_{i=1}^{N} T_i\, \alpha_i\, \mathbf{c}_i, \qquad T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j\, \delta_j\right), \qquad \alpha_i = 1 - \exp(-\sigma_i\, \delta_i).$$

The model uses a numerically stable formulation based on cumulative log-transmittance. After calculating ray colors for thousands of rays per iteration, I compute the MSE loss against ground-truth RGB values and backpropagate through the entire rendering pipeline.



volume rendering of images at different iterations



psnrs curves for both training and validation set

Replay GIF

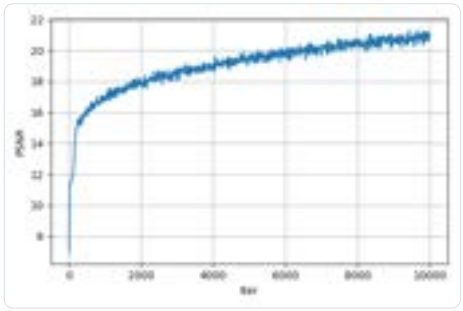
Part 2.6 — Training NeRF on My Own Data

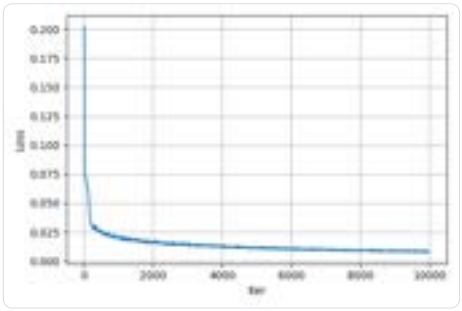
> **Training Hyperparameters**
> - **Iterations:** 10,000
> - **Batch size:** 8192 rays
> - **Samples per ray:** 64
> - **Near bound:** 0.01
> - **Far bound:** 1.5
> - **Learning rate:** 3e-4 (Adam)
> - **Model:** 8-layer NeRF, width 256, skip connection at layer 4
> - **image size is downscaled from 3024 by 4032 to 600 by 800 for efficiency)**

Reconstruction Progress

Below are six snapshots of the validation view rendered at different iterations[200, 1500, 2500, 5000, 8000, 10000] during training comparing to the original undistorted image. These demonstrate how the scene gradually becomes sharper and more consistent as NeRF learns geometry and color.

Training Curves





Final Novel View Synthesis

I generate 2 360° orbit GIF novel views around the reconstructed object by rotating a virtual camera around the estimated scene center.

Training a full NeRF on my own dataset turned out to be significantly more challenging than the earlier parts. The validation renderings were initially low-resolution and often contained artifacts such as black borders, missing geometry, and incorrect colors. Even after 5,000 iterations, the validation PSNR consistently plateaued below 19 dB, indicating that the model was failing to generalize well. I experimented with increasing `N_samples` and the batch size, but these changes provided only marginal visual improvements while greatly increasing runtime. Training on my MacBook CPU became impractical, so I switched to running the NeRF on an MBS GPU, which allowed more iterations and larger batches—but still required extensive hyperparameter tuning.

The most impactful improvement came from increasing the size of the training set. Instead of using 70% of the 35 captured images, I expanded the training subset to 90% of 45 images and reserved only 5 for validation(out of total 50 images). This provided the model with much denser multi-view coverage, noticeably improving stability and reducing artifacts. In the final reconstruction, the Labubu figure and the ArUco marker both become clearly recognizable, although some fine facial details

remain missing—likely due to small pose-estimation inaccuracies and imperfect extrinsics during data capture.

Despite these challenges, the final 360° novel-view GIFs are reasonably smooth and visually coherent. While not perfect, it demonstrates that with sufficient training coverage and careful tuning, a full NeRF can be reconstructed from a small, real-world dataset.