NeuralView: NeRF Pipeline for Novel View Synthesis

Atharv Gupta
Electrical Engineering, IIT Roorkee
Enrollment No: 25115029
Email: atharv g@ee.iitr.ac.in

BYOP End-Term Report — January 2026

1 Abstract

The goal of this project is to implement a complete Neural Radiance Field (NeRF) system from scratch in order to understand the internal working of neural 3D reconstruction. NeRF models a scene as a continuous function that takes a 3D position and viewing direction and outputs radiance and volume density.

During the mid-term phase, I focused on implementing the core NeRF architecture from scratch in PyTorch and building a preprocessing pipeline using COLMAP for camera pose estimation. In the end-term phase, the project was extended to improve robustness on real-world data, and explore faster NeRF variants such as TinyNeRF and Instant-NGP.

2 Methodology

This project uses two types of datasets: the standard NeRF Synthetic dataset and real images captured using a mobile phone. Using both kinds of data helped me understand the difference between working with ideal, perfectly prepared inputs and real-world images where pose estimation is more challenging.

2.1 Problem Statement

My interest in this idea started long before I heard about NeRF. Back in grade 9, I used to explore Unity and create small games as a hobby. Whenever I needed a simple object for my games, I would try to model it manually in Blender. This used to take a lot of time as blender is a difficult tool to learn and master.

I realized that instead of manually designing 3D models, NeRF makes it possible to reconstruct 3D objects directly from images.This felt like a natural solution to a problem I had faced years ago.

Real world use cases - NeRF-based methods are applicable in areas such as real estate and architecture, where indoor or outdoor spaces can be reconstructed from photographs to enable virtual walkthroughs. In cultural heritage and restoration, NeRF can help preserve artifacts or historical structures by creating digital reconstructions without physical contact

These applications highlight how NeRF bridges the gap between computer vision and graphics, enabling 3D understanding of real-world scenes using only image data

.2.2 Data Collection and Preprocessing

Two categories of data were used:

1. **NeRF Synthetic Dataset**
   The NeRF Synthetic dataset consists of clean, high-resolution rendered images with ground-truth camera parameters. These datasets provide perfectly consistent lighting and stable camera trajectories, making them ideal for testing the correctness of a NeRF implementation. I mainly used scenes such as Lego, Drums, and for testing.

2. **Real-World Images**
   For real data, I captured 40–50 images of objects in my room by slowly moving my phone around the object in a circular path. I tried to keep the movement smooth and ensure that each new image had enough overlap with the previous ones so COLMAP could detect common keypoints. These images were then used for pose estimation and later as training inputs.

2.2 Preprocessing Pipeline

The preprocessing stage is essential because NeRF depends heavily on accurate camera intrinsics, extrinsics, and a consistent dataset structure. Below I describe the steps I implemented and the theory behind them.

Camera Model and Projection

I used the pinhole camera model to understand how 3D points are projected into 2D images. A 3D point X in world coordinates is projected using:

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

where *K* is the intrinsic matrix containing focal lengths and the principal point, and [*R* |*t*] represents the extrinsic matrix that moves a point from the world coordinate system into the camera coordinate system.

For NeRF, we need the camera-to-world transform instead. This is computed as

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

This matrix tells us where the camera is located in the scene and which direction each ray

should be cast during volume rendering.

COLMAP Pose Estimation

To estimate camera poses from real images, I used the COLMAP pipeline:

1. Feature extractor 2. Exhaustive matcher 3. Mapper

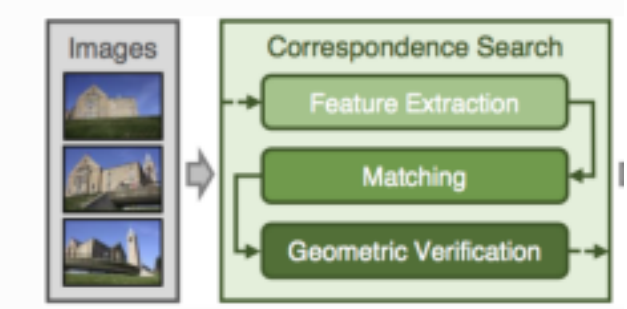

Figure 1: COLMAP pipeline.

COLMAP detects keypoints and matches them across frames, then uses bundle adjustment to compute camera intrinsics and extrinsics. When COLMAP succeeds, the resulting sparse point cloud and camera trajectory (Figure 2) are accurate enough for NeRF to train properly.

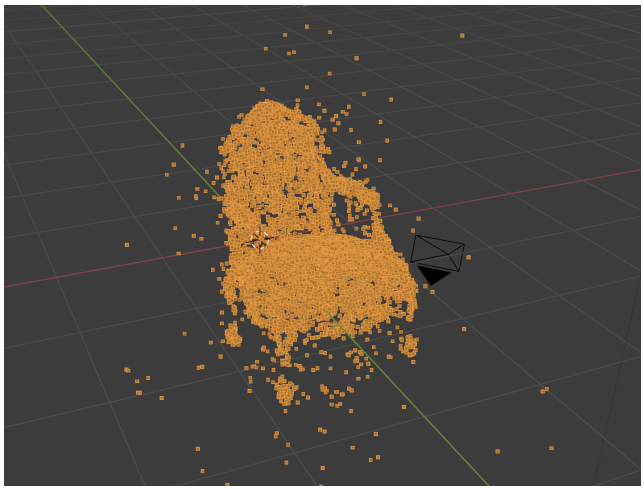

Figure 2: Example of COLMAP sparse reconstruction and recovered camera positions

Converting COLMAP Output to transforms.json

NeRF expects a specific JSON file format containing camera intrinsics, file paths, and the camera-to-world matrices. COLMAP does not output this format directly, so I wrote a custom conversion script. The script performs the following:

1. Reads the COLMAP camera parameters from the cameras.txt and images.txt files. 2. Converts COLMAP's world-to-camera matrices into camera-to-world matrices. 3. Adjusts axes to match NeRF's right-handed coordinate system. 4. Computes the horizontal field-of-view needed for camera angle x. 5. Stores each frame as:

"Transform Matrix": T_{cw}.

The final transforms.json is then used directly by the NeRF training code.


```
"transform_matrix": [
  [
    -0.8771350586446799,
    0.17098990148384738,
    0.4487722612717265,
    2.2771104838577054
  ],
  [
    0.3832921187790467,
    -0.3137607969463591,
    0.8687009347188797,
    -1.1301675610533768
  ],
  [
    0.28934622959054385,
    0.9339789161914361,
    0.20967151340038162,
    3.3651458717656375
  ],
  [
    0.0,
    0.0,
    0.0,
    1.0
  ]
]
```


Figure 3: Sample Transform.json

Background Removal (Optional)

For real images, background removal was tested after pose estimation. While this helped in some scenes, it also introduced bugs and degraded results in others, leading to mixed outcomes.

Moreover, Rembg (The python library I used to remove the background in batches) cant remove background of complex scenes properly leading to improper input to the model

Figure 3: The chair background wasn't removed properly

3 Model Architecture

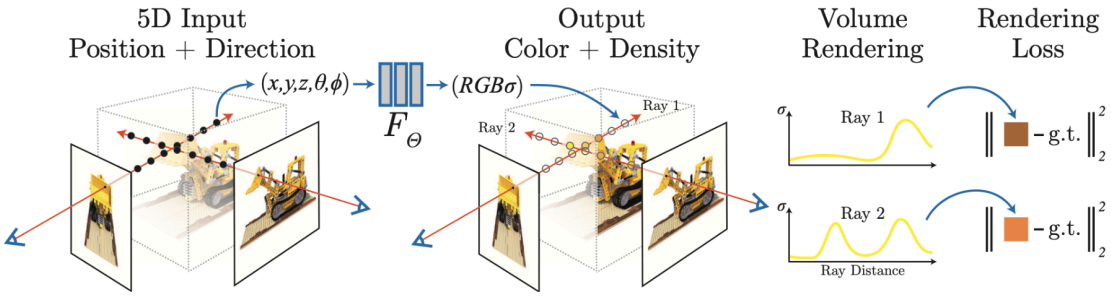

Figure 4: An overview of the neural radiance field scene representation and differentiable rendering procedure.

The NeRF model attempts to learn a continuous function mapping:

$$F_\Theta(\mathrm{x}, \mathrm{d}) \rightarrow (\mathrm{c}, \sigma),$$

where x denotes the 3D location, d represents the view direction, σ is the density, and c is the predicted color.

3.1 Input

The input to the NeRF model is a single continuous 5D coordinate, which consists of the positionally encoded spatial location (x, y, z) and the viewing direction (θ, ϕ).

During each training iteration, a batch of rays is randomly sampled from the input images, and points are sampled along each ray.

3.2 Processing Stages

Positional Encoding

Figure 4: Effect of positional Encoding (Left - no positional encoding , Right - positional encoding)

The NeRF pipeline consists of several key processing stages. First, 3D points along each camera ray are sampled between predefined near and far bounds. These points are then transformed using positional encoding, which applies sinusoidal functions at multiple frequencies to allow the MLP to learn high-frequency and sharper details.

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

In my implementation, I used **L = 10** for the spatial coordinates (x, y, z).And the final encoded 3D position becomes:

$$\gamma(\mathbf{x}) = [\gamma(x), \gamma(y), \gamma(z)] \in \boldsymbol{R}^{63}.$$

For the viewing direction, I applied a smaller positional encoding with **L = 4**.Giving a final direction embedding of:

$$\gamma(\mathbf{d}) \in \boldsymbol{R}^{27}.$$

3.3 Model Structure

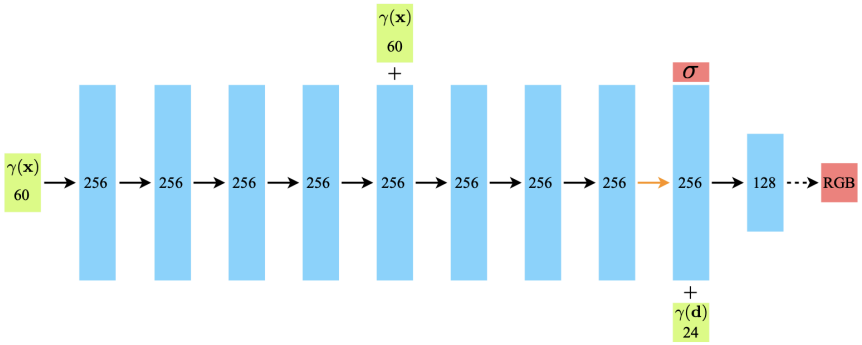

Figure 4: model architecture.

1. **Density MLP** This first MLP takes the encoded 63–dimensional spatial input and predicts:
    - the volume density σ
    - a 256–dimensional feature vector **h** that stores scene appearance information
2. **Color MLP** (View–Dependent Color Branch) This second MLP takes the feature vector h and the encoded viewing direction (27–dimensional) and produces the final RGB color c.

The density network consists of 8 fully connected layers, each with 256 channels. I used ReLU activation after each linear layer. To help the network learn both global structure and fine detail, a skip connection is added after the fifth layer. Specifically, the original positional encoding input is concatenated back into the hidden representation:

$$\mathbf{h}_5 = \mathbf{ReLU}([\mathbf{h}_4, \gamma(\mathbf{x})]\boldsymbol{W}_5 + \boldsymbol{b}_5).$$

After the eight layers, the density σ is predicted using a ReLU activation to ensure that density is non–negative: The remaining hidden features h (dimension 256) are then sent to the second stage of the network. This feature vector is concatenated with the positional encoding of the input viewing direction, and is processed by an additional fully-connected ReLU layer with 128 channels.

The final RGB color is predicted with a sigmoid activation to keep the output between 0 and 1:

Full Forward Pass Summary

Putting everything together, a single forward pass works as follows:

1. Encode (x, y, z) into a 63–dimensional vector.
2. Encode (θ, ϕ) into a 27–dimensional vector.
3. Pass the encoded position through the 8–layer density MLP with a skip connection

at layer 5. 4. Obtain density σ and a 256–dimensional feature vector.

5. Concatenate the feature vector with the encoded direction.
6. Pass through the color MLP to obtain the final RGB value c.

Hierarchical Sampling

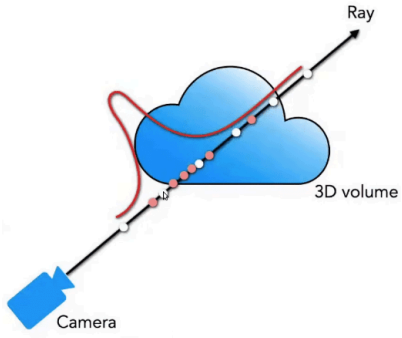

Figure 5: (White points are the points that are being sampled using stratified sampling (uniform sampling), where each point is equidistant. After using these points for volume rendering we get the weights w_i, which is interpreted as the contribution of the color at point i. Based on the weights we can decide where the surface of the volume lies. After we know the surface we sample more points near that region, generally a PDF sampler is used to sample points in that area)

NeRF uses hierarchical sampling to make the rendering process more efficient while still producing high quality results. Instead of sampling all points along a ray uniformly, the model first performs a coarse pass. In this stage, a smaller number of uniformly-spaced points is evaluated, giving a rough idea of where the scene may contain surfaces or density. This is called Stratified Sampling

Using this rough estimate, NeRF performs a second stage called fine sampling. In this stage, the model takes more samples only in the regions that the coarse network marked as important(PDF Sampling). This means that the network spends more effort on areas where geometry actually exists, instead of wasting computation on empty space. The coarse network therefore acts like a guide, and the fine network refines the details.

Optimizer and Regularization

I use the Adam optimizer and for Regularization (to prevent overfitting) I use skip connections which allows the original input (positional encoding) to be reintroduced at deeper layers of the network.This reduces the risk of vanishing gradients and makes the model easier to optimize.

Loss Function for Coarse and Fine Outputs

Both the coarse and fine networks predict a color value for each pixel. During training, these predictions are compared with the ground-truth pixel color. The final loss used to train the model is simply the sum of the two reconstruction losses:

$$\mathcal{L} = \left\| C_{\text{coarse}} - C_{\text{gt}} \right\|_2^2 + \left\| C_{\text{fine}} - C_{\text{gt}} \right\|_2^2 .$$

Here, C_{coarse} is the color rendered using the coarse network's samples, and C_{fine} is the color rendered using the fine network's samples.

This combined training encourages the coarse network to provide a good approximate layout of the scene and allows the fine network to focus its capacity on generating sharp, accurate details in the final image.

PSNR Evaluation

During implementation, I also evaluate the model using Peak Signal-to-Noise Ratio (PSNR). PSNR is a widely used metric in image reconstruction tasks because it measures how close the predicted image is to the ground-truth image. Higher PSNR values indicate that the reconstruction is more accurate and has less error.

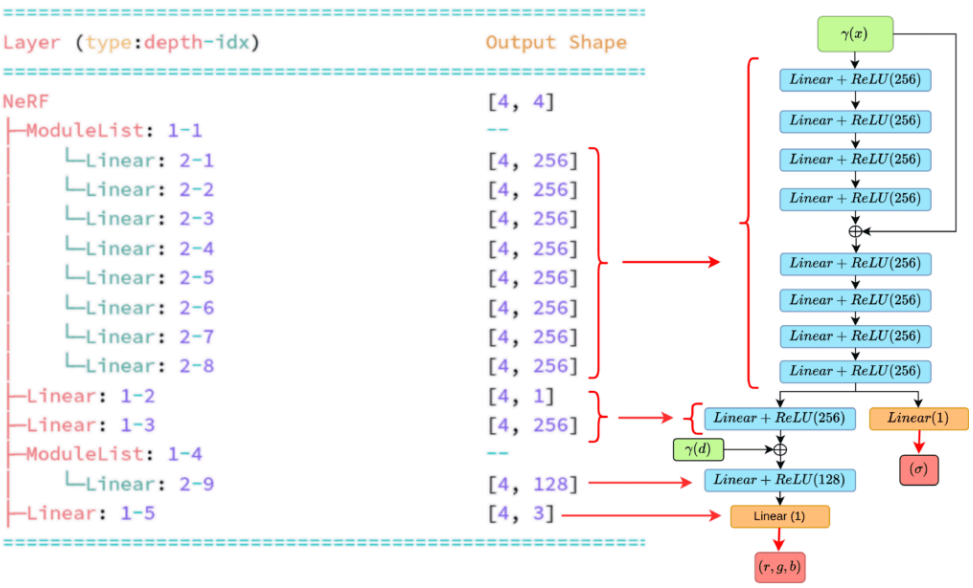

Figure 6 Model Architecture Side by Side comparison(Dimensional Clarity)

3.4 Tiny NeRF and Instant NGP

Tiny-NeRF is a simplified version of the original NeRF,The main difference between Tiny-NeRF and the original NeRF lies in the level of complexity and the time it takes to train on a scene.

The Tiny-NeRF network consists of eight fully connected layers with 256 neurons per layer, each followed by a ReLU activation function.To improve training stability and preserve spatial information, skip connections are introduced after every four layers, where the original encoded input is concatenated back into the network. Unlike the original NeRF . Tiny-NeRF predicts both color and density using a single network and does not model view-dependent effects

Moreover we don't use hierarchical sampling and use positional encoding only on the spatial location (*x, y, z*) with L = **6**

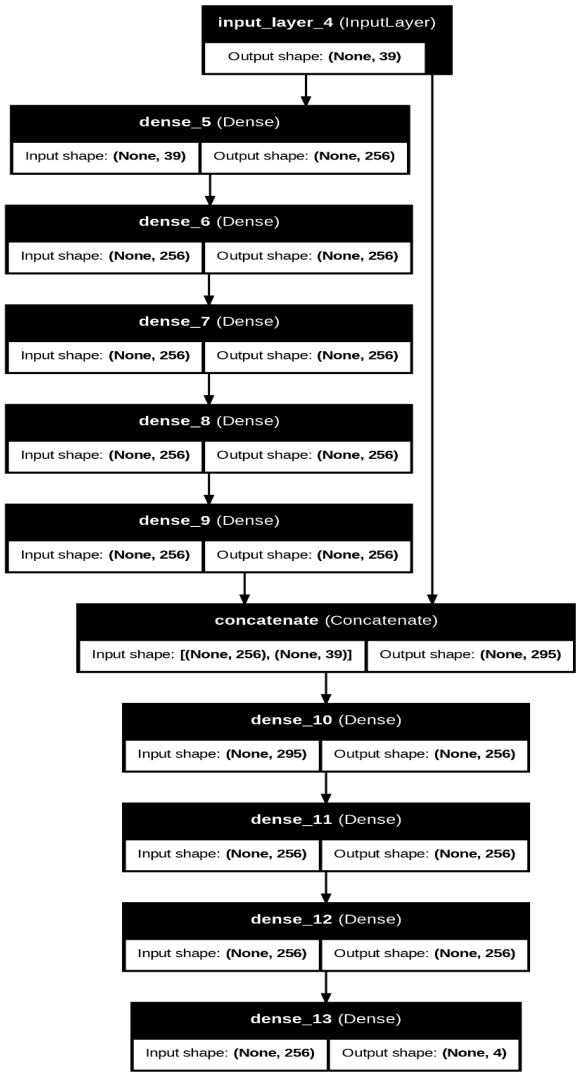

Figure 7 Tiny NeRF architecture

Instant-NGP is a fast NeRF-based method that can reconstruct 3D scenes in a few minutes instead of hours. It achieves this speed-up by using a multi-resolution hash-grid encoding instead of traditional positional encoding.(I used this to play around with the cutting edge tech in 3d reconstruction also to see where the NeRF is lacking in)

4 Result

4.1 Results on Synthetic NeRF dataset

The first set of experiments was conducted on the synthetic NeRF dataset, which provides clean images, accurate camera poses, and consistent lighting conditions. These datasets are used as a benchmark for evaluating NeRF implementations and were ideal for validating the correctness of my pipeline.

Table 1: PSNR comparison between my implementation and the original NeRF authors.

Dataset	Iterations	My PSNR	Author PSNR
Lego	5000	25 dB	32 dB (50,000 iterations)
Drums	8000	22 dB	25 dB (50,000 iterations)

Although my PSNR values are lower than those reported by the original NeRF authors, this difference is expected. My models were trained for significantly fewer iterations due to GPU and time constraints. Despite this, the results were visually reasonable.

Figure 8 Novel view synthesis of the *Drums* dataset after 8,900 iterations. Rendering views across a 360° camera path allows the creation of a 3D orbit video, helping visualize the reconstructed object from all angles.

4.2 Results on Real NeRF dataset

After validating the pipeline on synthetic data, I moved on to real-world images captured using a phone. These datasets were significantly more challenging due to: 1) Not enough overlap in

between the images 2) Background clutter

For real-world objects such as shoes and chairs, the model was able to learn coarse geometry and approximate appearance. However, the final quality was noticeably lower compared to synthetic scenes

Figure 9 Right: The original image Left: One of the novel views generated by the model, It is clear that the model is able to learn the geometry of the shoes as well as the color decently but fails to reconstruct thinner and smaller details like the shoe laces as well as the brand logo. A potential fix could be to increase the position encoding dimensions for the x,y,z coordinates and to remove/reduce the background clutter.

4.3 Results using Instant Ngp

Working with Instant-NGP allowed me to experience the cutting edge of 3D reconstruction firsthand. The fact that it can produce high-quality results in just 1–2 minutes where a traditional NeRF often takes several hours(and lots of gpu power) was quite impressive and astonishing

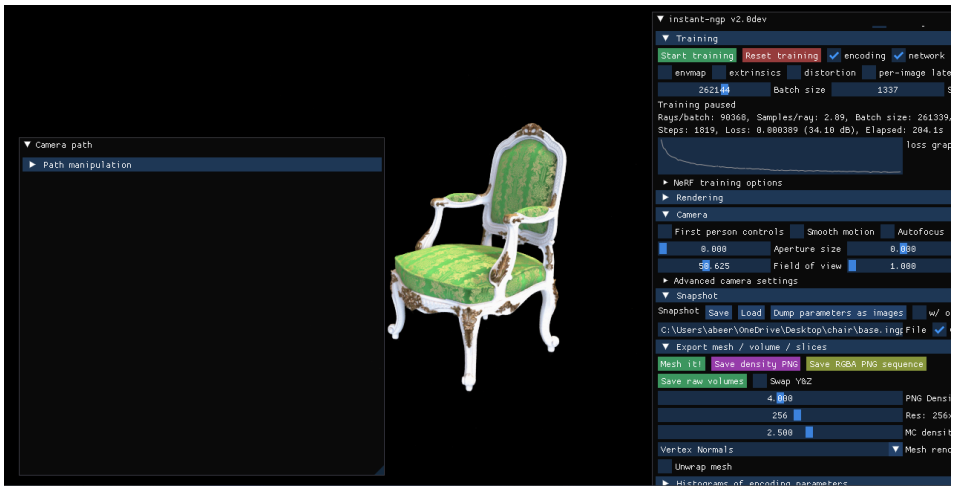

Figure 10 3d reconstruction of a chair using instant ngp, On the right there can is the psnr loss plot

5 Final Deliverable

The final deliverable of this project is a complete, end-to-end Neural Radiance Fields (NeRF) pipeline that allows a user to reconstruct a 3D representation of an object using only a set of overlapping images captured from multiple viewpoints.

5.1 End-to-End Preprocessing Pipeline

The repository includes a full preprocessing pipeline that prepares raw user images for NeRF training. The preprocessing steps include:

- **Image Organization & Renaming (Rename.py)**
- **COLMAP-Based Camera Pose Estimation (Runcolmap.py)**
- **Generation of transforms.json (GenTransform.py)**
- **Optional Background Removal (Removebg.py)**

5.2 Model Selection

Allows users to choose between two model (Tiny and Full NeRF) variants depending on their computational and time constraints .

5.3 3d orbit video and Mesh generation

After training, novel camera poses are sampled along a circular trajectory around the object to synthesize a 360° orbit video

The learned volumetric density field can be converted into an explicit 3D mesh using marching cubes. This allows the reconstruction to be exported as a standard mesh file.

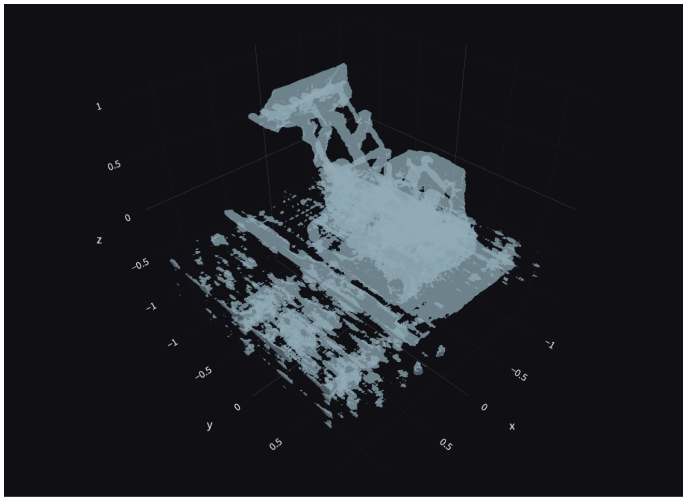

Figure 5: Mesh extracted using marching cubes from the learned density

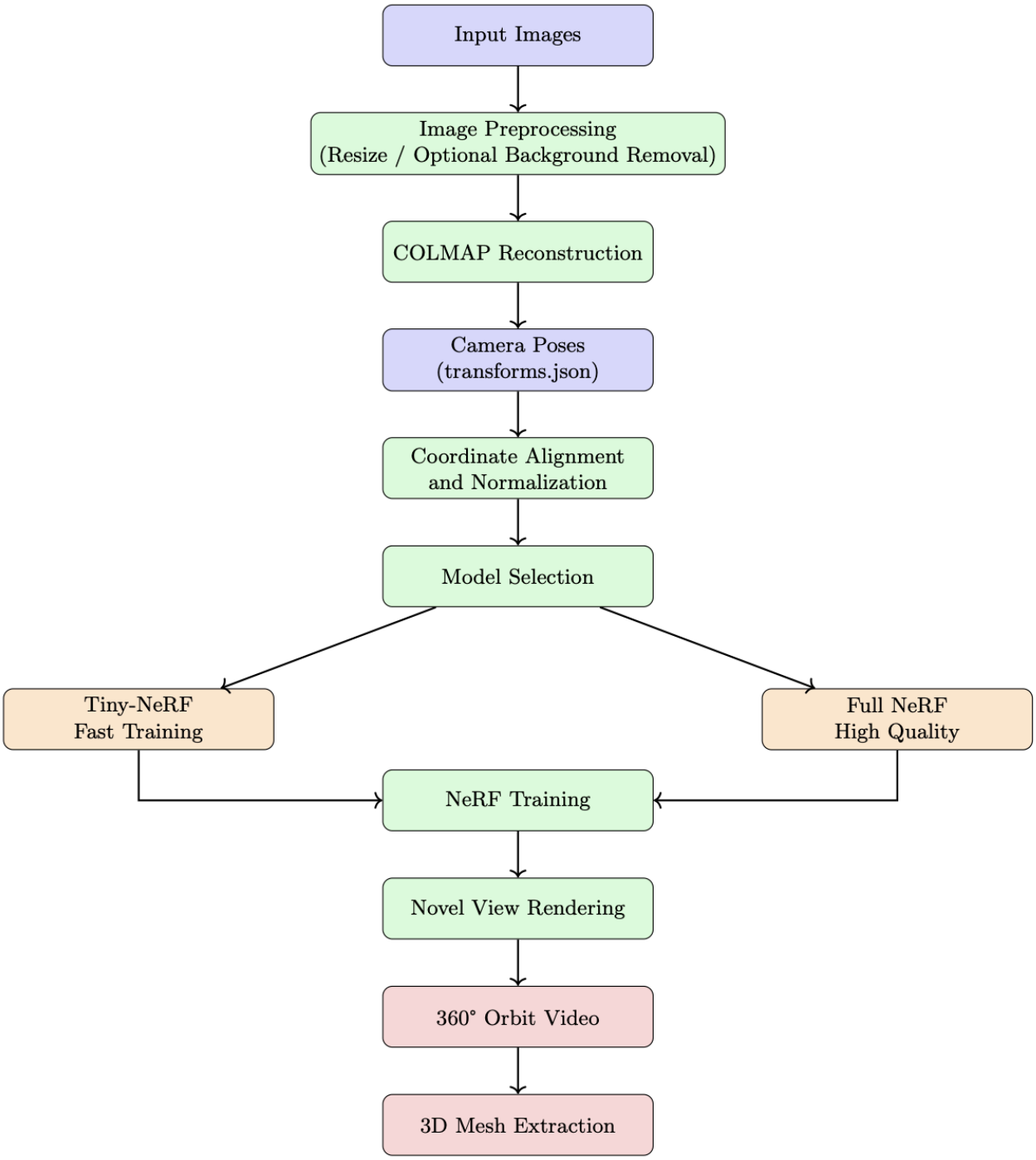

Figure 11 Overview of the complete pipeline. Input images are preprocessed and passed through COLMAP to estimate camera poses. After coordinate alignment, the user may choose between Tiny-NeRF for faster training or Full NeRF for higher-quality reconstructions. The trained model enables novel view synthesis, 360° orbit video generation, and 3D mesh extraction

6 Challenges Faced

6.1 Gpu Constraints

I initially trained the model on my local machine using Apple's Metal Performance Shaders (MPS) as the accelerator. This setup was fine for small experiments with low-resolution images (around 100 × 100), but it quickly became a bottleneck when I tried scaling up to higher resolutions like 800 × 800. Training became extremely slow and often ran out of memory, making meaningful experimentation difficult.

To get around this, I switched to Kaggle, where I had access to two NVIDIA T4 GPUs (each with 15 GB of VRAM). This made a big difference, as I could increase both the batch size and image resolution, and I used *nn.DataParallel* to train on both GPUs simultaneously

One of the main problems with kaggle is the 30 hours limit and random server shutdown which was quite annoying and frustrating prompting me to make about 5 different kaggle accounts!.

6.2 Black Outputs with Fixed PSNR

During multiple experiments, training resulted in completely black images while PSNR remained constant across iterations. This was traced back to a combination of incorrect background handling, sigma activation instability, and near–far plane misconfiguration.

To address this, I modified the pipeline to allow the user to explicitly choose between a black or white background during training. This simple change significantly improved stability and made the model more robust across different types of objects and lighting conditions.

6.3 Instant-Ngp Setup

Running Instant-NGP turned out to be quite a headache, especially on macOS since it is not supported natively and requires CUDA to function properly. I initially tried running it on Google Colab, but repeatedly ran into issues where the installation was incomplete, often missing the testbed binary. This forced me to retry the setup multiple times, which was both time-consuming and frustrating.

Eventually, I switched to a Windows machine, which provides an official pre-built Instant-NGP setup with CUDA support. This finally allowed me to run Instant-NGP successfully.

7 Secondary Goals

One of the goals I initially aimed to achieve was building a complete Streamlit-based interface that would allow a user to upload their own images and run the entire pipeline end-to-end, including preprocessing, COLMAP pose estimation, NeRF training, and final visualization, all from a single web interface. While parts of this integration were planned and partially explored, I was unable to complete it within the given time constraints.

The main challenges were the long training times of NeRF models, GPU memory limitations . Given the limited time and compute resources available, I decided to prioritize correctness, robustness, and reproducibility of the core pipeline rather than risk an unstable demo.

In future work, this project could be extended by fully integrating the pipeline into a Streamlit or web-based interface, possibly by offloading heavy computation to background jobs or cloud-based GPU services. Additional improvements could include automated dataset validation, smarter background handling, and tighter integration with faster methods such as Instant-NGP to provide near real-time feedback to the user.

Moreover, I also wanted to extend the system to allow users to upload a short video of an object instead of individual images, from which frames could be automatically extracted and used to generate a 3D reconstruction

8 References

[1] Ben Mildenhall et al., “NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis,” ECCV, 2020.

[2] Tensorflow Implementation of NeRF

[3] Tensorflow implementation of Tiny NeRF

[4] Colmap documentation

[5] Instant Ngp

[6] Pytorch end to end implementation with explanation

[7] Nerf Synthetic dataset

[8] Great Demonstration website by the authors of the paper

[9] Nerf Studio

9 Appendix

9.1 Failed Masking-Based Training Approach

During the development of this project, I experimented with a masking-based training strategy where the background was explicitly removed from each image using automatic background removal tools, and the NeRF model was trained only on the remaining foreground object. The motivation behind this idea was fairly intuitive: by removing the background, the model would be forced to focus entirely on the object geometry, potentially reducing noise during training and improving reconstruction quality for real-world images

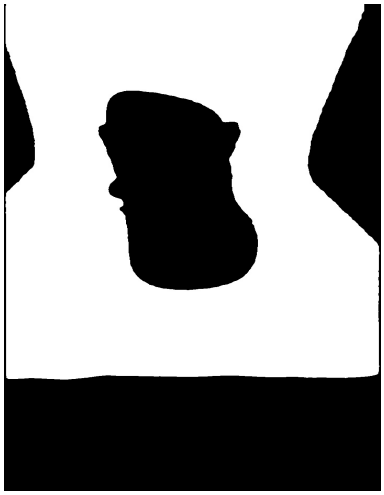

Figure 12 Mask generated by using SAM (Segment anything model)

However, after multiple training runs, this approach consistently produced poor results. One major issue was that the masking process removed more than just the background , it also eliminated subtle color gradients, soft shadows, and boundary transitions near the object edges. These details turned out to be important cues for NeRF's volumetric rendering process.

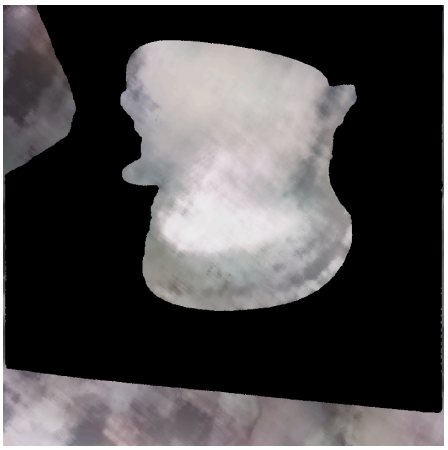

Figure 13:Result obtained using the masking-based approach even after 8,000 iterations. The reconstruction failed to recover meaningful geometry and appeared blurry and inconsistent, whereas the normal (non-masked) training setup produced much cleaner and more stable results.