LONDON'S GLOBAL UNIVERSITY
Department of Computer Science

On Neural Radiance Fields (NeRFs) for Analysing Dark Cave Tunnels

University College London

BSc Computer Science

Supervisors: Simon Julier

Submission Date: 26th April 2023

This report is submitted as part requirement for the BSc Degree in Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Acknowledgements

First, I would like to thank my project supervisor Dr. Simon Julier who I have had a chance to meet every week to discuss the progress of the project and provide me with valuable feedback. His enthusiastic and passionate nature made me enjoy the research process and he has also been a great mentor.

Secondly, I want to give a special thanks to Oliver Kingshott, a PhD student who attended my project meetings and found an interest in my project. His insight and knowledge have helped me to understand key concepts to a deeper level and encouraged me to work at my potential, which has helped me to tackle the challenging problems in my project.

Another person I would like to thank is Dr. Karoly Zsolnai-Feher who runs the YouTube channel "Two Minute Papers". His videos on the latest research in machine learning and computer vision have been inspirational and have made the research process an enjoyable experience.

Thank you to all the lecturers and teaching assistants who have taught me over the past 3 years, as I have been able to apply the knowledge they have given me during the BSc Computer Science course to this project.

Last, but not least, I want to express my appreciation to my parents for their support and encouragement throughout this project.

Abstract

Neural Radiance Fields (NeRFs) are hugely popular in the field of Computer Vision. They can reconstruct novel views of a 3D scene, however, most research has focused on rendering objects and environments in good lighting conditions. This project aims to investigate the effectiveness of NeRFs in low-lighting environments.

We explored the Onboard Illumination Visual- Inertial Odometry (OIVIO) dataset, in particular the images of a dark underground cave tunnel. We employed the COLMAP (structure-from-motion package) pipeline to extract the camera poses and intrinsic parameters necessary for training a NeRF model. After training the model for 85,000 iterations, we carried out an analysis of the reconstructed cave tunnel's structure.

We present "explore mode", a unique way to navigate through novel view predictions using keyboard input. In this mode, we could plot opacity values of a ray through a scene and create disparity maps to highlight surfaces closer or further away from the camera. We also demonstrated specular reflection of cave walls using fixed-pose inferencing in a series of videos.
Keywords: Neural Radiance Fields, Novel View Synthesis, 3D Reconstruction, Visual-Inertial Odometry Systems, COLMAP

Table of Contents

1 | Introduction

1.1 Problem Outline

Neural Radiance Fields (NeRFs) can reconstruct 3D scenes and synthesise novel views from a set of images and their corresponding camera poses. As NeRF models are able to compactly represent the world, they have many applications, such as in large/city-scale urban mapping, 3D surface analysis, and robotics/autonomous navigation.

A vast amount of research on NeRFs has been focusing on improving their inference speed or the quality of their renders. However, less work has been done to investigate the effectiveness of NeRF models in dark, dimly-lit environments. Many of the popular datasets used to train a NeRF are captured under good lighting conditions. Dark scenes pose a challenge as the images can contain more noise have a limited dynamic range between the brightest and darkest parts of the scene. All of this makes it harder for a NeRF model to capture fine details and accurately infer the underlying 3D structure of the scene.

Therefore, the objective of this work is to apply NeRFs to a novel dataset of a dark cave tunnel, observe how well it reconstructs the 3D scene, and perform analysis on the structure of the cave walls.

1.2 Project Aims

- Understand the theory behind NeRFs
- Learn to use the COLMAP software package for extracting camera poses
- Learn to combine COLMAP and NeRFs to train a model on a new dataset
- Explore the Onboard Illumination Visual-Inertial Odometry (OIVIO) dataset and then train a NeRF on images of a cave tunnel
- Understand how well NeRFs can represent a scene in reverse

1.3 Project Goals

- A literature review covering the background of NeRFs and COLMAP
- A trained NeRF model of an underground cave tunnel from the OIVIO dataset
- Ability to explore through a NeRF model using keyboard input
- Disparity maps to see which surfaces are closer or farther away from the camera
- Use the NeRF model to produce novel views of the cave in the reverse direction by turning the camera 180°
- Videos showing the specular reflection of light moving around the cave walls

1.4 Research Approach

Throughout the project, we used an iterative approach to conducting research and developing the code for analysis. On the research side, our goal was to build up a comprehensive understanding of the theory behind Neural Radiance Fields and COLMAP. We

accomplished this by starting to read the original papers and then using review surveys to find related papers that covered the progress and latest research in the field.

On the development side, we began by running an existing Python notebook that implements a simple version of a NeRF on a toy dataset and exploring how to use the COLMAP software package.

Once we had understood how to import a dataset, train a NeRF model, and analyse its output, we could apply what we had learnt to the novel dataset (OIVIO) and train a NeRF model to study the problem and carry out our analysis.

We organised weekly meetings with the project supervisor and a PhD student to receive feedback on our latest progress and plan the next set of tasks to complete. We used the project management software, Trello, to create a list of tasks for each week, which helped to keep track of the project's progress by marking off completed tasks.

1.5 Overview of Report

Chapter 2 introduces the reader to a literature review on Neural Radiance Fields (NeRFs) and explores a simple implementation of one. In Chapter 3, the COLMAP software package and its pipeline for extracting camera pose data are described. Chapter 4 combines the data from COLMAP with a set of images of a building to train a NeRF model. Chapter 5 applies the process of using COLMAP and NeRFs to the OIVIO dataset, which consists of dark underground cave tunnels. Finally, in Chapter 6, we use the trained NeRF model to analyse the structure of cave tunnels.

2 | Exploring NeRFs

In this chapter, we introduce the formal theory of a NeRF model and explain how it can learn to represent a 3D scene and be used to generate a new view from any viewpoint. We explore the implementation of a simple NeRF in a Python notebook, where we discuss loading the input data, understanding the various hyperparameters, and analysing the output from the model.

2.1 Neural Radiance Fields (NeRFs)

NeRFs (Neural Radiance Fields) [1] are fully-connected deep neural networks that generate novel views of a 3D scene by training on a sample set of 2D images and then approximating the scene as a radiance field. In a radiance field, every point continuously emits light of a certain colour. The volume density of that point determines the intensity of the radiance.

The input to the neural network is a 5D vector, $[x, y, z, \theta, \phi]$, where $[x, y, z]$ is the location of a point in the scene and $[\theta, \phi]$ is the viewing direction. Points are sampled from a ray pointing in the viewing direction.

The outputs are the view-dependent colour, $[R, G, B]$, and the volume density, σ. Colour is a function of both the location and viewing direction, whereas, volume density is only a function of the location. This is because if the viewing direction is changed, the density of a point in space remains the same, but its emitted colour changes, as different amounts of light rays enter the camera.

View-dependence allows a NeRF to learn non-Lambertian effects [2]. This is when light scatters off smooth and shiny surfaces, resulting in specular reflection. By fixing the camera's render pose, but changing the inputted viewing direction, we can render the

reflected light moving around a static scene, which creates a unique effect as shown on this website [1].

2.1.1 Training a NeRF

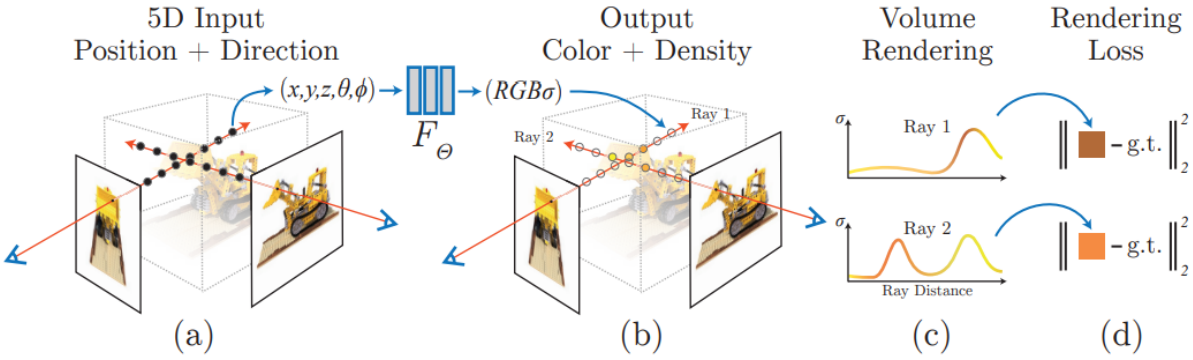


Figure 2.1: 4 steps to rendering a novel view. In (a), a ray is sent through the scene to sample 5D coordinates. In (b), the NeRF model predicts the RGB view-dependent colour and volume density for each sample. In (c), the final RGB colour of a pixel is calculated using volume-rendering techniques. In (d), the squared error loss is minimised using gradient descent to improve the model. (Reference: Figure 2: [1])

There are 4 steps to training a NeRF and then rendering a novel view from the model. These are labelled a, b, c, and d in Figure 2.1.

In step (a), a camera ray is sent through the scene for each pixel in the resulting image. The 5D coordinates $[x, y, z, \theta, \phi]$ are generated by sampling locations along the camera ray and combining them with the viewing direction, which is passed to the NeRF model.

Next, in (b) the model outputs its prediction of the view-dependent colour $[R, G, B]$ and volume density σ for each sample on the ray.

Then, in step (c), classical volume rendering techniques [3] are used to integrate the functions of density and colour between the starting t_1 and ending t_2 bounds of the ray, resulting in a $[R, G, B]$ colour for each pixel.

[1] View-Dependent Appearance Videos: https://www.matthewtancik.com/nerf

Finally, in step (d), the loss is computed, which is the squared error between the predicted pixel colour and the ground truth colour from the training image. The loss is minimised using gradient descent, resulting in a trained NeRF that can generate novel views.

2.1.2 Volume Rendering

We will now go into more detail about the volume rendering process described in step (c). We start by defining the camera ray as, $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, where t is the parametric length of the ray, $\mathbf{o}$ is the camera origin vector, and $\mathbf{d}$ is the viewing direction vector.

The colour of a given ray through a pixel is given by,

$$\mathbf{C}(\mathbf{r}) = \int_{t_1}^{t_2} T(t) \cdot \sigma\left(\mathbf{r}(t)\right) \cdot \mathbf{c}\left(\mathbf{r}(t), \mathbf{d}\right) dt, \tag{2.1}$$

where $T(t)$ is a function of accumulated transmittance. This refers to the probability that the ray is not blocked by a particle from the starting bound t_1 to the current location t and is defined as,

$$T(t) = \exp(-\int_{t_1}^{t} \sigma\left(\mathbf{r}(u)\right) du) \tag{2.2}$$

In step (d), the loss function is given by,

$$L = \sum_{r \in R} ||\hat{\mathbf{C}}(\mathbf{r}) - \mathbf{C}(\mathbf{r})||_2^2, \tag{2.3}$$

where $\hat{\mathbf{C}}(\mathbf{r})$ is the model's predicted colour and $\mathbf{C}(\mathbf{r})$ is the ground-truth colour of the pixel from the training image. R is the batch of rays, so the loss is summed over all rays.

2.1.3 Optimisation of a NeRF

The 4 steps we described in 2.1.1 would result in a NeRF model that performs poorly on complex scenes. What we mean by a complex scene is high-frequency variations such as sharp changes in lighting and shadows, fine-grained details (e.g., tree leaves), or small bumps on a surface (e.g., pebbles, dirt).

We can optimise a NeRF to better model such scenes using two techniques. Positional encoding, which creates a high-frequency function, and Hierarchical sampling, which allows us to sample from this high-frequency function.

2.1.3.1 Positional Encoding

When a NeRF is trained using the original 5D input coordinates, $[x, y, z, \theta, \phi]$, it is unable to learn high-frequency variations. Findings by Rahaman *et al.* [4] show that deep neural networks tend to be biased towards learning low-frequency functions.

A clever trick is to map the input to a higher dimensional space and then pass this encoded input to the NeRF allowing it to learn high-frequency changes. The mapping, from $\mathbb{R}$ to $\mathbb{R}^{2N}$, is performed using a composition of sine and cosine functions. Given an input vector $\mathbf{v}$, its encoding is given by,

$$\texttt{ENCODING}(\mathbf{v}) = \left(\sin(2^0\pi\mathbf{v}), \cos(2^0\pi\mathbf{v}), \sin(2^1\pi\mathbf{v}), \cos(2^1\pi\mathbf{v}), \cdots, \sin(2^{N-1}\pi\mathbf{v}), \cos(2^{N-1}\pi\mathbf{v})\right),$$

where N is a hyper-parameter for the number of pairs of sines and cosines used. The encoding is applied separately to the three coordinates in the location, $\mathbf{x}$, and the three components of the cartesian viewing direction, $\mathbf{d}$. In the original NeRF paper, the authors set $N = 10$ for the location and $N = 4$ for the viewing direction.

2.1.3.2 Hierarchical Sampling

In most 3D scenes, the volume is sparse, meaning that many points don't contribute much to the final rendered view. The simple approach to evaluating the integral for volume rendering is to use stratified sampling, which samples N points uniformly along the ray. However, this results in sampling unnecessary points in free space or regions occluded from the camera.

Hierarchical Sampling is a technique that proportionally over-samples parts of the ray that have a high likelihood of contributing to the final render. In Figure 2.2, the red ray has more samples in areas that correspond to useful objects/features in the scene and fewer in areas that are sparse.

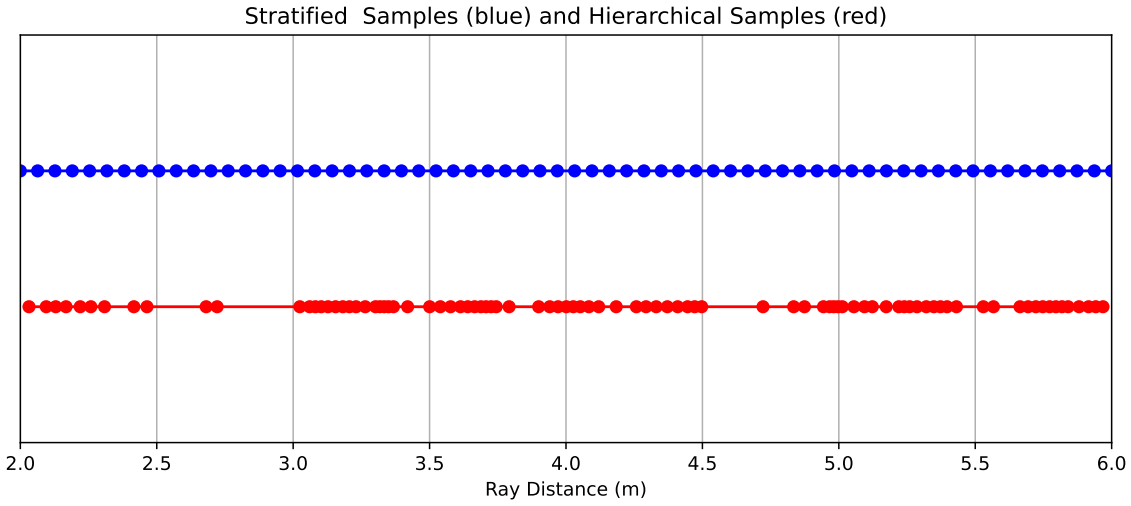


Figure 2.2: The difference between a ray sampled using a Stratified Sampling (blue) and a Hierarchical Sampling (red). In Stratified Sampling, the samples are evenly spread across the ray, whereas in Hierarchical Sampling, certain regions are sampled more as these correspond to important features in the scene.

2.1.3.3 Coarse and Fine Models

To achieve a more accurate reconstruction of a scene, both stratified and hierarchical sampling are utilised. Instead of training a single neural network, a NeRF can be divided into two models: "coarse" and "fine".

Both models are optimised simultaneously and represent different structures in the scene. Large features, such as a building or a car are modelled by the *coarse* network. First, stratified sampling generates the input points to evaluate the "coarse" model on. Then, using the output of the "coarse" model, hierarchical sampling generates a more detailed sampling of points along each ray by focusing on the relevant parts of the scene. These samples are used to evaluate the "fine" model.

2.2 Implementation of a Simple NeRF

We started researching implementations and found one in an article on Towards Data Science titled *"It's NeRF From Nothing: Build A Complete NeRF with PyTorch"* by Mason McGough [2].

The article provided a Google Colab notebook [3] to train a basic NeRF model. McGough used most of the code from an implementation by the original authors of the NeRF paper but modified it to run on the free resources provided by a Google Colab environment.

His modification included using chunking, which performs the forward pass in chunks for each batch to address potential memory issues, and adjusting the hyperparameters, such as reducing the number of hidden layers.

[2]Article: https://towardsdatascience.com/its-nerf-from-nothing-build-a-vanilla-nerf-with-pytorch-7846e4c45666

[3]Colab notebook: https://colab.research.google.com/drive/1TppdSsLz8uKoNwqJqDGg8se8BHQcvg_K

2.2.1 Loading a dataset

Recall that a NeRF model takes 5D coordinates, $[x, y, z, \theta, \phi]$, as input. However, in practice, this notebook loads a dataset of a specific format and chooses to represent the input differently by setting the viewing direction as a 3D unit vector, $\mathbf{d}$.

The dataset consists of a set of RGB images of the scene taken from multiple viewpoints; the extrinsic camera poses corresponding to each image which are stored in 4×4 matrices; and the focal length which is an intrinsic camera parameter. We discuss what extrinsic and intrinsic camera parameters are in more detail in Section 3.2.

2.2.1.1 Lego Tractor Dataset

The notebook uses a popular dataset of a synthetic Lego tractor (Figure 2.3) created in the 3D software Blender by the original authors of the NeRF paper. The dataset is stored as a Numpy zip, so after downloading it, the images, poses, and focal length can be loaded as shown below,

```
data = np.load("lego_tractor.npz")

images = data["images"]
poses = data["poses"]
focal = data["focal"]
```

Figure 2.3: A ground-truth pose from the Lego Tractor dataset. The tractor is a synthetic 3D model created in Blender.

The dataset contains 106 images of size $(100 \times 100 \times 3)$. As the object was created in synthetically, the camera poses were already given. They were generated randomly on a hemisphere around the vertical z-axis (an example is shown in Figure 2.4). This is important as a dataset of images captured in the real world won't have ground-truth poses already given, instead, they would need to be calculated. Nevertheless, for the purposes of training a NeRF, this dataset is a great starting point.

Figure 2.4: An example of generating camera poses around the z-axis on a hemisphere for the drums dataset. (Reference: Figure 1: [1])

2.2.2 NeRF Hyperparameters

The NeRF model contains many hyperparameters, but we will focus on the most important ones. We have divided the hyperparameters into different sections to make it clear what they control, and we have also included what their default values are set to.

Encoders

Hyperparameter	Default Value
`d_input`	3
`n_freqs`	10

These are for the positional encoder. `d_input` is the number of dimensions of the input point, which is set to 3 as the points are in 3D space. `n_freqs` is the number of pairs of sines and cosines functions used in the encoding.

Sampling

Hyperparameter	Default Value
`n_samples`	64
`pertub`	True
`n_samples_hierarchical`	64
`perturb_hierarchical`	False

These are for the stratified and hierarchical sampling methods. `n_samples` is how many samples are to be taken on the ray and `pertub` is a Boolean that, when set, adds some random noise to the locations of each sample.

NeRF Model

Hyperparameter	Default Value
`lr`	5e-4
`d_filter`	128
`n_layers`	2
`use_fine_model`	True
`d_filter_fine`	128
`n_layers_fine`	6

These are for the neural network. `lr` sets the learning rate for the optimiser. The `d_filter` is the number of neurons in each layer and `n_layers` is how many layers there are. These two hyperparameters are for the coarse model and there are also two for the fine model.

Training

Hyperparameter	Default Value
`n_iters`	10,000
`batch_size`	2^{14}
`chunksize`	2^{14}

The model will train for `n_iters`. `batch_size` is the batch size which sets how many rays are sent. `chunksize` is a unique hyperparameter just for this notebook which divides the forward pass into chunks. This is needed to manage the limited memory provided in a Google Colab environment.

Early Stopping

Hyperparameter	Default Value
`warmup_iters`	100
`warmup_min_fitness`	10.0
`n_restarts`	10

Early stopping causes the training process to restart if it has stalled in a local minimum. Restarting only occurs if the number of iterations is below `warmup_iters`. If the PSNR metric hasn't increased more than `warmup_min_fitness`, then a restart occurs. The number of restarts is capped at `n_restarts`.

2.2.3 Analysing the Output

While the NeRF model is being trained, the notebook outputs a PNG image (Figure 2.5), which contains four figures, every 25 iterations to keep track of how the model is learning. The first figure is the predicted RGB map of the view and the second figure is the target ground-truth RGB map that was taken from the test set. The third figure shows a plot of the Peak Signal-to-Noise Ratio (PSNR) for the training set (shown in red) and the validation set (shown in blue). The higher the PSNR, the closer the prediction is to the target. Finally, the fourth figure is a plot of the stratified samples (shown in blue) vs hierarchical samples (shown in red) sampled from the ray going through the center of the scene.

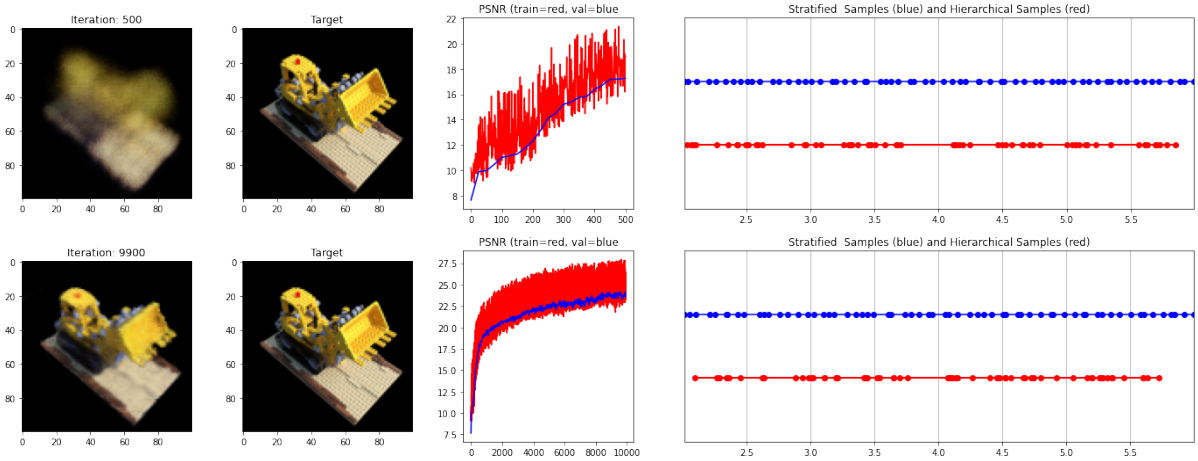


Figure 2.5: Plots showing the NeRF training process. The top row was after **500** Iterations and the bottom row was after **9900** Iterations. The first column is the predicted novel view, the second column is the ground-truth test pose, the third column is the PSNR metric, and the fourth column is the plot of the stratified vs hierarchical samples from the ray going through the center of the scene.

To calculate the PSNR, we first compute the Mean-Squared-Error (MSE) between two images, I_1 and I_2,

$$\text{MSE} = \frac{1}{mn} \sum_{i=1}^{m-1} \sum_{j=1}^{n-1} \left[I_1(i,j) - I_2(i,j) \right]^2 \tag{2.4}$$

The PSNR is then defined as,

$$\text{PSNR} = 10 \cdot \log_{10} \left(\frac{(I_{MAX})^2}{\text{MSE}} \right), \tag{2.5}$$

where I_{MAX} is the maximum possible pixel value in the image.

The PSNR value is measured in decibels (dB) and provides a measure of how much the predicted image deviates from the original image. A higher PSNR value indicates less distortion in the predicted image and therefore better image quality.

2.3 Summary

This chapter introduced the reader to Neural Radiance Fields (NeRFs). We explained how they take a 5D vector as input and output radiance and volume density.

We then described the 4-step process of training these networks, which started by firing rays into a scene, predicting the output, applying classical volume rendering to produce an RGB map, and performing stochastic gradient descent to minimise the loss. Furthermore, we looked into techniques to optimise a NeRF, such as positional encoding and hierarchical sampling.

We analysed an implementation of a simple NeRF in a Google Colab notebook, where we described the popular Lego tractor dataset and outlined the various hyperparameters used to train NeRFs. Finally, we studied the output the notebook produced, which included the prediction and PSNR evaluation metric.

3 | COLMAP and Pose Extraction

In Chapter 2, we studied NeRFs and presented the Lego tractor dataset, a popular collection of images used to train a NeRF model. We mentioned that this dataset provides the camera pose information for each image's view as they were generated synthetically in software. However, pose information is not readily available, especially for a custom set of images taken in the real world.

In this chapter, we will start by understanding the COLMAP application and how its pipeline can extract pose information from a set of images of a scene. We will then explain camera intrinsics and extrinsics and demonstrate how a Python script can extract all this data from COLMAP, which can then be packaged up into a dataset ready for training a NeRF.

3.1 COLMAP

COLMAP is a Structure-from-Motion [5] software package that provides a pipeline to reconstruct a 3D scene from a set of 2D images. The images overlap and are taken from different viewpoints around a particular object, such as a building, tree, or car. COLMAP estimates a sparse point cloud of the world, as well as the camera poses for each image's viewpoint.

3.1.1 COLMAP Pipeline

We present the reader with the full COLMAP pipeline for structure-from-motion reconstruction in Figure 3.1, however, we have chosen to expand upon the most important sections of the process.

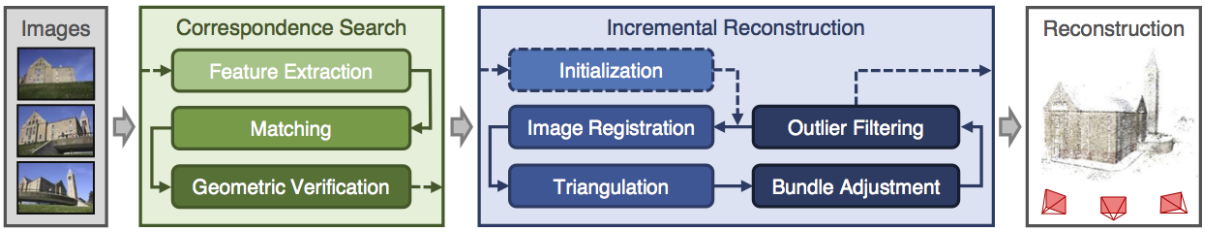


Figure 3.1: A diagram of the COLMAP Pipeline which shows the different processing stages to estimate the camera poses and sparse point cloud of the scene from the set of images. (Reference: Figure 2: [5])

Feature Extraction

The first stage of the pipeline is about detecting the particular features, such as points and corners, that appear in multiple images. This allows COLMAP to estimate the location and orientation of the camera relative to other images. One widely-used method is SIFT [6] which is robust to noise, changes in illumination, and affine distortion. Additionally, this method can be used to match features across a wide range of scales and orientations.

Feature Matching

Using the features extracted in the previous stage, COLMAP searches for correspondences between features. They indicate which points in one image correspond to the same points in another image. The K-Nearest-Neighbour algorithm [7] is one technique that is used to find the k closest features in one image for each feature in another image.

Reconstruction Initialisation

COLMAP initialises the pose estimation by choosing two images that are more likely to give a robust estimate of the pose. It will choose from a dense location in its image graph where there are many overlapping images.

Triangulation

Once COLMAP has estimated the poses for two images, it will solve for a new feature that is in these two images. This process is called triangulation because the two image poses are two vertices, and the new feature is the third vertex.

Bundle Adjustment

After estimating the poses for each image, COLMAP will optimise them further by minimising the reprojection error in a procedure known as bundle adjustment [8]. This is the error between the predicted 2D projection of a feature and its ground truth location in an image.

3.2 Extrinsic and Intrisic Camera Parameters

In this section, we will explain the two types of camera parameters that are needed to train a NeRF. Then, in Section 3.3, we will show how we extracted this data from COLMAP.

3.2.1 Extrinsic Camera Parameters

Extrinsic camera parameters define the location and rotation of a camera in the 3D space. These can be represented by a 4×4 matrix, also known as the pose matrix, which applies a transformation to points from the camera's reference frame to the world reference frame.

$$\begin{bmatrix} \boldsymbol{R}_{3\times3} & \mathbf{t}_{3\times1} \\ 0_{1\times3} & 1 \end{bmatrix}, \tag{3.1}$$

where $\boldsymbol{R}$ is the 3×3 rotation matrix and $\mathbf{t}$ is the 3×1 translation vector, which stores the origin position of the camera. The last row, $[0, 0, 0, 1]$, is a 4D homogeneous coordinate that ensures the transformation maps vectors to vectors and points to points.

3.2.1.1 Quaternions

Instead of a 4×4 pose matrix, COLMAP returns camera extrinsics as quaternions and translation vectors. A quaternion is a 4D complex vector that represents just the rotation of the camera but is more compact than the 9-element rotation matrix.

$$\mathbf{q} = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}, \tag{3.2}$$

where w is the real part and x, y, z are the imaginary parts corresponding to the 3 rotation axes.

Given a quaternion, the following computation gives the corresponding rotation matrix,

$$R(\mathbf{q}) = \begin{pmatrix} 2(q_0^2 + q_1^2) - 1 & 2(q_1q_2 - q_0q_3) & 2(q_1q_3 + q_0q_2) \\ 2(q_1q_2 + q_0q_3) & 2(q_0^2 + q_2^2) - 1 & 2(q_2q_3 - q_0q_1) \\ 2(q_1q_3 - q_0q_2) & 2(q_2q_3 + q_0q_1) & 2(q_0^2 + q_3^2) - 1 \end{pmatrix} \tag{3.3}$$

The Python library SciPy has tools to convert from a quaternion to its rotation matrix, which we use in Section 3.3.3.

3.2.2 Intrinsic Camera Parameters

Intrinsics are parameters specific to the internal components of the camera used to capture the scene. COLMAP supports multiple types of cameras, including Pinhole, Radial, and Fisheye. The default is a Radial camera, and its intrinsics are focal lengths, optical centers, and the skew coefficient. These can be represented in the following 3×3 matrix

$$\begin{pmatrix} f_x & k & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{3.4}$$

The focal length for a camera can differ for the x-axis and y-axis, but COLMAP assumes they are the same and returns one value. The optical centers will be assumed to be half the resolution, $(c_x = W/2, c_y = H/2)$, where W is the width and H is the height. The skew coefficient, k, is the amount by which a pixel is shifted in the x-axis. It is set to 0 if the axes are perpendicular.

3.3 Extracting Data from COLMAP

After COLMAP completes its reconstruction of the scene, it stores the predicted extrinsic and intrinsic parameters in binary files. First, we needed to convert these to text files, which we did using the provided binary-to-text tool. Once converted, we were able to read the text files and extract the necessary data required for creating a dataset to train a NeRF.

The intrinsic parameters are stored in the `cameras.txt` file and the extrinsic parameters are stored in the `images.txt` file.

3.3.1 Cameras Text File

We have shown the data from the `cameras.txt` file in table 3.1. Each row stores data on a different camera, but for our reconstruction, COLMAP returns the data for a simple radial camera. In the column headers, CX and CY correspond to c_x and c_y, the center of projection, and K is the pixel skew, k. From this row, the only intrinsic parameter we need to extract is the focal length.

CAMERA_ID	MODEL	WIDTH	HEIGHT	FOCAL	CX	CY	K
1	SIMPLE_RADIAL	3072	2304	2558.42	1536	1152	-0.020

Table 3.1: `cameras.txt` data

3.3.2 Images Text File

We have shown the data from the `images.txt` file in table 3.2. There are N rows for each image from the input set. In each row, $[QW, QX, QY, QZ]$ are the values of a quaternion and $[TX, TY, TZ]$ are the values of a translation vector.

Image_ID	QW	QX	QY	QZ	TX	TY	TZ	CAMERA_ID	FILENAME
1	0.862	0.0185	0.485	-0.146	-0.677	1.002	3.645	1	P1180141.JPG

Table 3.2: `images.txt` data

Furthermore, the filename entry is also useful, as our dataset will need to store the pixel data of each image. We used OpenCV to read an image's filename from its source directory, converted the data into RGB format, normalised the values to the range $[0, 1]$, and saved it in NumPy arrays. The final array storing the images would be of size $(N \times \text{HEIGHT} \times \text{WIDTH} \times 3)$.

3.3.3 Creating the Pose Matrix

Recall that the extrinsics pose matrix, as shown below, is 4×4 and consists of the camera's rotation matrix and translation vector.

$$\boldsymbol{M} = \begin{pmatrix} \boldsymbol{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \tag{3.5}$$

To get a rotation matrix from the quaternion, $[QW, QX, QY, QZ]$, we needed to use the SciPy library which comes with many useful functions for handling rotations. One such function is `Rotation.from_quat(quat)` which takes a quaternion and outputs the corresponding (3×3) rotation matrix. The input quaternion needs to be of the format $[QX, QY, QZ, QW]$, so we had to re-arrange the array to have the scalar component, QW, as the last entry.

To get the translation vector, all we needed to do was store each component, $[TX, TY, TZ]$, in a NumPy array.

To create the pose matrix, we used `np.column_stack()` to concatenate the rotation matrix and translation vector column-wise. Then we added the additional row, $[0, 0, 0, 1]$, to the bottom which resulted in the (4×4) matrix. This was repeated for each row in the `images.txt` file, resulting in N pose matrices for N images.

3.3.3.1 Converting from world to camera coordinates

When we applied the process above on a test dataset, we realised that the poses were incorrect. The cameras pointed in seemingly random directions instead of forming a hemisphere, which was how the images were taken.

We learned that the quaternions and translation vectors from COLMAP were stored in world-to-camera (W2C) coordinates. However, we actually wanted the data to be stored in camera-to-world (C2W) coordinates, therefore, we needed to convert from W2C to C2W.

The conversion process starts by getting the inverse pose matrix, $\boldsymbol{M}^{-1}$. Then we needed to flip the signs of entries in the y and z axes (the second and third columns) to rotate the scene 180°, as shown in this source code [1], by multiplying $\boldsymbol{M}^{-1}$ with the following flip matrix,

$$\boldsymbol{F} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.6}$$

[1] Line 340 in the source code: https://github.com/NVlabs/instant-ngp/blob/master/scripts/colmap2nerf.py

3.4 Summary

In this chapter, we introduced the COLMAP application, which allows us to extract camera poses from a set of images. This is necessary to train a NeRF model and COLMAP's pipeline is used when the ground truth poses are not provided.

The pipeline consists of several stages, as we showed in Figure 3.1, but we chose to explain the most important steps, which were: Feature Extraction, Feature Matching, Reconstruction Initialisation, Triangulation, and Bundle Adjustment.

We then presented the mathematics of a camera's extrinsic and intrinsic parameters, which led us to describe the process of extracting these parameters from the binary files that COLMAP outputs. Finally, we used the extracted data to construct the pose matrix.

4 | Applying COLMAP and NeRF

In Chapter 2, we introduced NeRFs, which are Neural Radiance Fields that can be trained to learn a 3D scene from a set of images. We mentioned that the input dataset requires camera extrinsics, which store the location and rotation, as well as the intrinsic parameters, such as focal length. Then, in Chapter 3, we discussed the COLMAP software package and how its pipeline can be used to extract the extrinsic and intrinsic camera parameters from a set of images.

In this chapter, we will be training a NeRF model on a new dataset that isn't one of the synthetic datasets. This dataset will be a collection of real world images, for which the camera parameters are not available. Therefore, we will be making use of COLMAP to extract the extrinsics/intrinsics.

4.1 The Building Dataset

The tractor dataset was created synthetically in Blender, so we wanted to train a NeRF on a dataset of images taken in the real world. One such dataset we discovered is of the South Building at the University of North Carolina, Chapel Hill which we downloaded from [1].

It contains 128 images with a resolution of (3072×2304), captured at ground level by rotating 360 degrees around the building. The images feature quite a bit of vegetation, including bushes and small trees, that cover the lower part of the building. The structure of the building is composed of bricks of varying colors and numerous glass windows that reflect the surrounding light.

This dataset has the advantage of having all images taken with the same camera, lens, and focal length, which enhances COLMAP's accuracy in computing the camera poses.

[1] Datasets: https://colmap.github.io/datasets.html

4.2 Running COLMAP on the Building Dataset

Before we can train the NeRF, we needed to create the input dataset which consists of the set of images, camera poses, and the focal length intrinsic parameter. We used the COLMAP pipeline to perform the extraction of camera data from the images.

In the COLMAP GUI, there is an "Automatic Reconstruction" tool that outputs the sparse and dense reconstructions in a workspace folder. We gave the location of the images folder and set the quality to "Low". The reason for this is that extracting camera poses doesn't require higher processing, which is intended for higher-quality point clouds. We then ticked the "Shared Intrinsics" toggle, which tells COLMAP that the same camera was used in all the images. Finally, we enabled GPU acceleration and initiated the process.

When the whole process has been completed, there is a sparse and dense folder. The sparse folder stores the camera's extrinsic and intrinsic parameters, while the dense folder stores the 3D point cloud mesh of the building.

4.2.1 MeshLab

Although the 3D mesh is not important for training a NeRF, it helped us to visualise how accurate COLMAP's reconstruction was. If we noticed issues in the render, then there might be problems with the predicted camera poses.

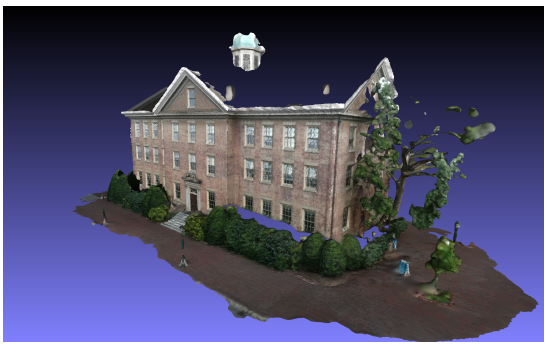
We used the mesh processing software, MeshLab [2], to verify the reconstruction of the building by importing the ".ply" (Polygon File Format) file that COLMAP outputted. Shown below are the front (a) and side (b) views of the building.

Notice that the roof is missing in the render because the images were taken on the ground floor, so they didn't capture the upper section of the building. However, the building

[2]Meshlab: https://www.meshlab.net/

itself was rendered well, which meant that we could go ahead and use the estimated camera parameters.

(a) Front View of the dense reconstruction from COLMAP



(b) Side View of the dense reconstruction from COLMAP

4.3 Downsampling

In Section 2.2, we looked at McGough's Google Colab notebook [3], which trained a simple NeRF model. He had used the tractor dataset which had images of resolution (100×100). If images of a higher resolution we used, the colab environment would then struggle to load them onto the GPUs, which resulted in an out-of-memory error.

Recall that the images from the building dataset have a resolution of (3072×2304), which is significantly larger. We realised that we needed to downsample the images before we could train a NeRF on them.

Since downsampling doesn't have an impact on the camera's orientation, we could still use the camera extrinsics that COLMAP extracted using the original resolution images. However, the focal length intrinsic would need to be divided by the same scale factor used for downsampling the images.

[3] https://towardsdatascience.com/its-nerf-from-nothing-build-a-vanilla-nerf-with-pytorch-7846e4c45666

We chose to downsample the images by a scale factor of 32, reducing the resolution to (96×72) so that neither the width nor height would exceed 100 pixels.

4.3.1 Extracting data from COLMAP

We created a Python notebook in order to extract the relevant data from the sparse folder.

In the first cell, we import the necessary libraries, which include: CV2 for loading each image, NumPy for storing the translation and quaternion vectors, Pandas for reading data from the text files, Matplotlib for plotting the camera rays, and the SciPy `spatial.transform.Rotation` package for converting from quaternions to the rotation matrix.

After executing all the cells, we generated three NumPy arrays with the following shapes,

Array	**Shape**
`Images`	(128, 72, 96, 3)
`Poses`	(128, 4, 4)
`Focal`	(1)

The camera's focal length for the original images was 2558.42, but after dividing by the scale factor 32, it became 79.95.

We now present the output of the notebook for a sample image and its corresponding pose at index 0,

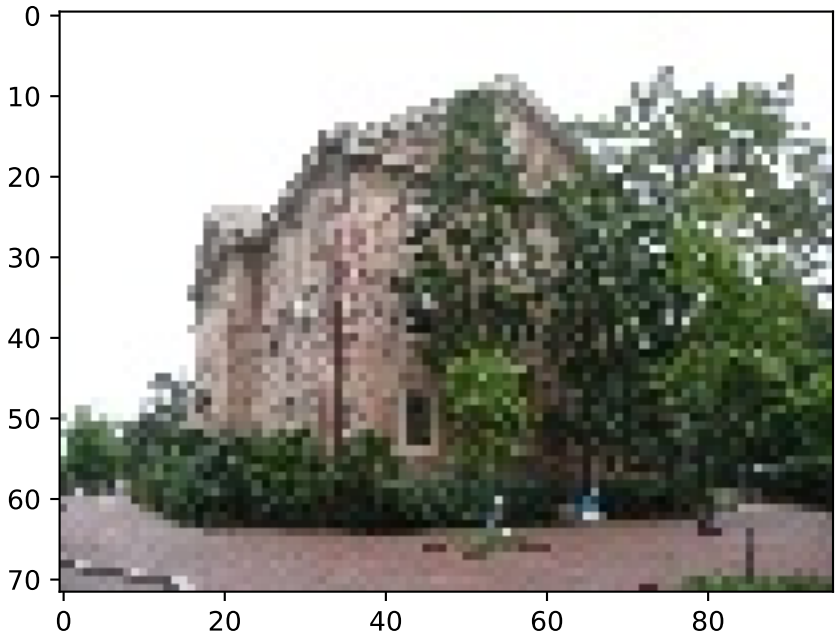


Figure 4.2: The first (index 0) ground-truth image of the building

```
Pose from COLMAP of the image at index 0:
[[ 0.4862282   0.23385237  0.8419592   3.6334524 ]
 [ 0.2698713  -0.95661026  0.10984683 -0.3757273 ]
 [ 0.8311147   0.17380999 -0.5282409  -1.1884611 ]
 [ 0.          0.          0.          1.        ]]
```

The final cell of the notebook combines the three arrays into one NumPy zip file,

```
np.savez("building_dataset.npz", images=images, poses=poses,
          focal=focal_length)
```

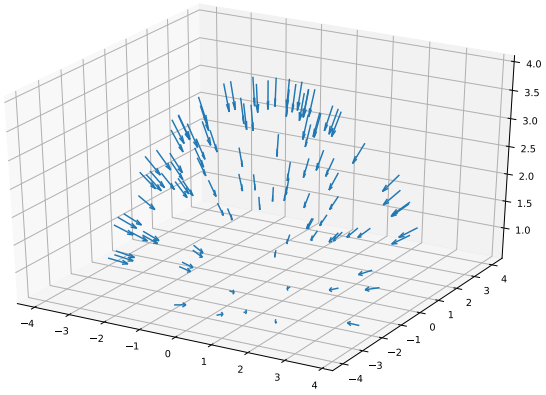
The ".npz" file, which stores the three arrays, is in the same format as the tractor dataset, making it straightforward to load into the NeRF notebook.

4.4 Visualising Camera Poses

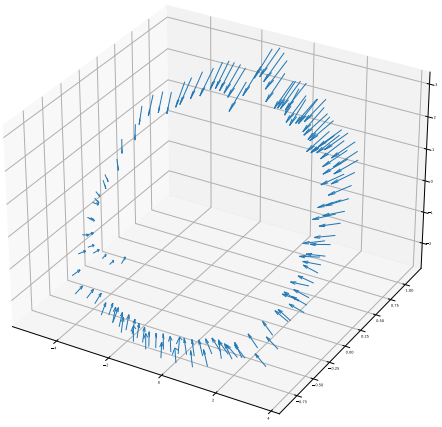
In the NeRF notebook, McGough included a cell to render the origins and directions of all cameras using their pose matrices. The result is a Matplotlib 3D plot that shows vectors to represent the rays.

McGough used this to visualise the camera poses of the tractor dataset, as seen in Figure 4.3 (a). The plot confirms our understanding that the camera poses were synthetically generated around a hemisphere.

We used the code to visualise the poses of the building dataset to evaluate COLMAP's estimation. Since the images were captured 360 degrees around the building, we expected a circular ring. As you can see in Figure 4.3 (b), the rays do form around a circular ring, but they are at an incline.



(a) Camera pose directions from COLMAP for the Lego Tractor dataset



(b) Camera pose directions from COLMAP for the Building dataset

Figure 4.3

The reason for this is that COLMAP lacks the concept of a ground plane, which means it doesn't enforce any particular orientation for the cameras. This explains why the ground-truth tractor poses form a hemisphere around the z-axis, while the predicted building poses form a ring around an arbitrary axis.

In Figure 4.4, we visualise the building in Meshlab with the default view and enable the world axis (Z is blue, X is red, and Y is green). It becomes clear that the building, seen on its side, is aligned vertically to some direction close to the world's y-axis, hence the camera rays were plotted at an angle.

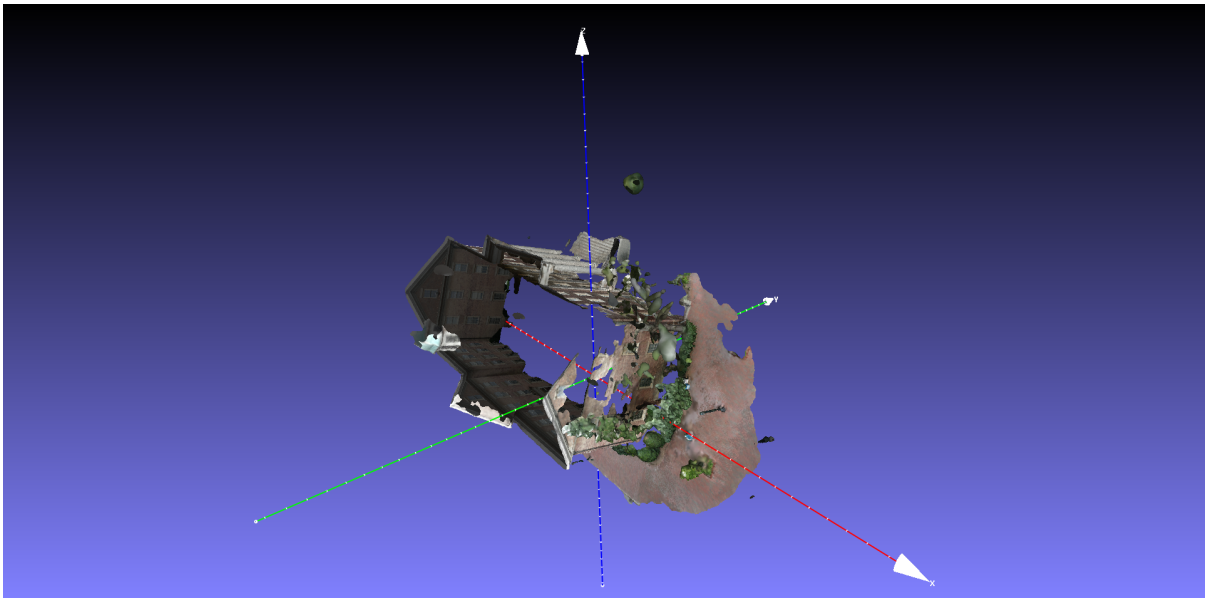


Figure 4.4: Meshlab render of the reconstructed building rotated to be aligned with the world axis. Z-axis is blue, X-axis is red, and Y-axis is green.

Fortunately, these poses would have no impact on training a NeRF. However, when querying the model with novel views, we would need to use poses that are aligned similarly to the training poses.

4.5 Training a NeRF to Reconstruct the Building

Loading the NumPy zip dataset into the NeRF notebook was straightforward as the format is the same as the Lego tractor, so the following lines of code were enough,

```
data = np.load("building.npz")

images = data["images"]
poses = data["poses"]
focal = data["focal"]
```

Recall that we had downsampled the building images to a resolution of 72×96. The hyperparameters for training the NeRF model were set to handle the tractor images, which are sized at 100×100. Given that our images are slightly smaller, we determined that adjusting the hyperparameters was unnecessary, as the available Google Colab free GPU resources would be capable of handling our dataset during training.

4.5.1 NeRF training results

We trained the NeRF script for 10,000 iterations and now present three different stages throughout the process. The target image is from the test set, so we are measuring the model's performance on a novel view.

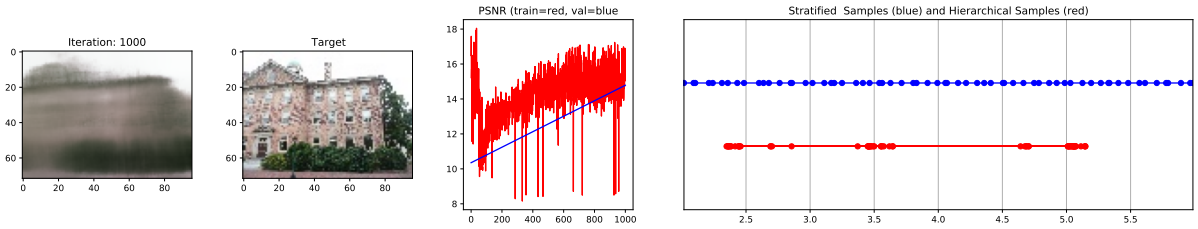


Figure 4.5: NeRF training output at 1000 iterations

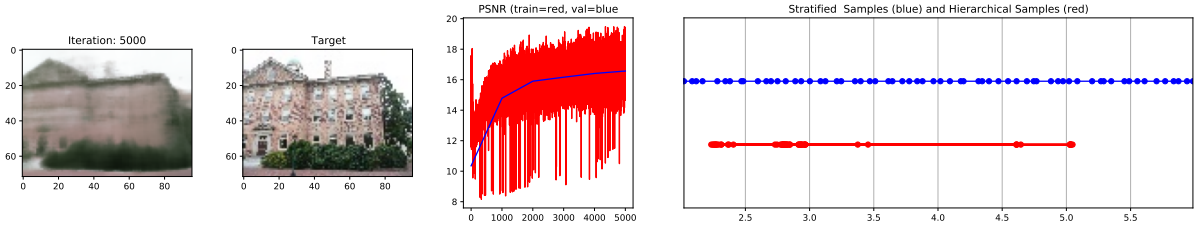


Figure 4.6: NeRF training output at 5000 iterations

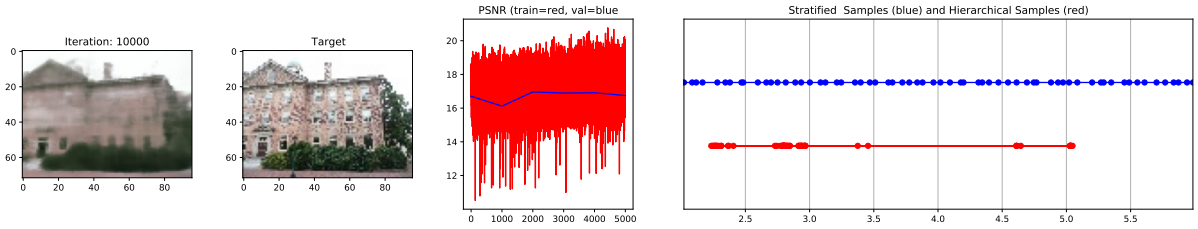


Figure 4.7: NeRF training output at 10,000 iterations

We should mention that the reason the PSNR plot's x-axis in Figure 4.7 ranges from 0 to 5000, instead of 0 to 10,000, is beacause we trained the NeRF in batches. Therefore, this axis actually represents the last 5000 iterations. In other words, it is the PSNR values for iterations 5000 to 10,000.

From these outputs, it is clear the NeRF's prediction gets closer to the target image. This is reflected in the increase in the PSNR values. Since this metric is logarithmic, a small increase represents a greater change in quality.

4.6 Summary

In this chapter, we explored how to train a NeRF on real-life images of a building at the University of North Carolina, allowing us to move beyond the synthetic Lego tractor dataset. We explained that using such a dataset meant that there weren't any ground-truth camera extrinsic and intrinsic parameters available, therefore, we needed to use the COLMAP pipeline, introduced in Chapter 3, to calculate this data for the building dataset. We also discussed the downsampling process that was necessary to reduce the image sizes and focal length intrinsic to make training a NeRF possible with the free resources given by a Google Colab environment.

After displaying the extracted camera data from COLMAP, we visualised the camera poses of the building images and compared them to the synthetic camera poses of the tractor dataset. By using MeshLab, we then investigated why the poses were not aligned with world axes.

Finally, we demonstrated the results of training a NeRF model on the building dataset, which showed an improvement in the PSNR metric, and therefore the prediction quality, of an unseen pose as the number of iterations increased.

5 | Training a NeRF on the OIVIO Dataset

In the previous chapter, we applied COLMAP on a real-life dataset of a building and then trained a NeRF model to predict novel views of the 3D scene.

In this chapter, we will repeat the process of running COLMAP to extract camera poses and training a NeRF model. However, we will move on from the building dataset to one that will be the subject of our research investigation into analysing underground dark cave tunnels.

5.1 The OIVIO Dataset

The Autonomous Robotics and Perception research group at the University of Colorado have created the OIVIO dataset [9], which stands for Onboard Illumination Visual-Inertial Odometry. The training set contains over 44,000 frames of data, while the testing set contains over 20,000 frames of data. The dataset includes the ground truth trajectory of the robot and calibration information for both the camera and the inertial sensors, however, it doesn't provide camera poses, for which we will employ the COLMAP pipeline.

They recorded various sequences of dark environments, including a cave tunnel, a lab with the lights off, and even a forest at night. To capture this data, they built a sensor rig robot with an Intel RealSense stereo camera, as well as an onboard 100W, white LED flashlight. Using the flashlight, they set three different intensity levels: 15%, 50%, and 100% of its full capacity, and sent the robot down the same route with the three different illumination settings.

The dataset was designed for evaluating visual-inertial odometry algorithms in dim lighting conditions, where the amount of light would vary significantly throughout the data collection process. This inspired us to train a NeRF on the underground tunnel and

analyse how well it could reconstruct the scene, given the challenging low-lighting environment. Note that this was not an issue in the previous datasets we had started with, such as the Lego tractor or the building dataset.

5.1.1 The underground tunnel sequence

We began with using the mine dataset at 15% lighting as it had the smallest size compared to the other lighting level datasets, coming in at 5.01GB. The duration of the robot's route took approximately 249 seconds, and as the stereo cameras were recording at 30Hz, this resulted in 7498 images per camera, but we only selected the images from one camera. The resolution of these images was 1280×720, therefore, we would need to downsample them later.

Out of the 7498 images, we selected 150 images of a specific section of the tunnel to train a NeRF on. While most of the tunnel happens to be quite repetitive, this section contains various features, such as a meandering path, wooden structural pillars, and ceiling lamps. Together, these would result in a more interesting 3D reconstruction for analysis.

Shown below, in Figures 5.1, 5.2, 5.3, are images of the start, middle, and end of the specific section of the tunnel we chose.

Figure 5.1: Ground-truth view of the tunnel from its starting location

Figure 5.2: Ground-truth view of the tunnel in the middle

Figure 5.3: Ground-truth view of the tunnel from the ending location

5.2 NeRF PyTorch Repository

Up to this point, we have been using the Google Colab notebook by McGough to train NeRF models for the Lego tractor dataset and the building dataset. However, due to memory limitations in the Colab environment, the notebook implements a modified version that is simpler to run with the free resources available. It became clear that we needed to transition to a NeRF implementation that matched what the authors of the original paper had used.

We found a PyTorch NeRF GitHub repository [1] that implements a NeRF as the original authors had designed and can run on GPUs using a script, `run_nerf.py`. The advantage of this version is that we can run it on our high-performance lab PCs, which have the following specifications: i9 processors, 128 GB RAM, and Nvidia GeForce RTX 3090 graphics cards. This will allow us to train larger models much faster than if we had continued using the Colab notebook.

5.3 Loading a NumPy zip dataset

The first issue we came across was loading our NumPy zip dataset that we had created specifically for the Google Colab notebook. Although the NeRF repository still requires the same data (i.e., images, extrinsics, and intrinsics), it needed to be stored in different formats.

The repo provided scripts to load various formats, which included blender data (in JSON files), deepvoxels (3D feature embeddings in an object-oriented format), and Local Light Field Fusion (LLFF) data. The repository also came with the popular tractor dataset, but it was stored in the blender format.

[1] https://github.com/yenchenlin/nerf-pytorch

Therefore, we started by implementing our own script, `load_npz.py`, to load datasets in the ".npz" format. After extracting the images, poses, and focal length, we also needed to split the data into train, validation, and test sets which would store the index positions of the images.

Let `indexes` be an array of integers in the range $[0, N]$, where N is the total number of images. We first shuffled the array to ensure the data is randomly distributed between the sets, `indexes=SHUFFLE(indexes)`. We decided to split the sets into the following amounts,

Set	**Percentage**
Train	80%
Validation	10%
Test	10%

With the following code, we can get the index positions of the end of each set,

```
train_index = int(NUM_IMAGES*0.8)
val_index = train_index + int(NUM_IMAGES*0.1)
test_index = val_index + int(NUM_IMAGES*0.1)
```

For example, if $N = 100$, then `train_index` = 80, `val_index` = 90, and `test_index` = 100. With these indexes, we can extract the sets using the following slices,

```
train_set = indexes[:train_index]
validation_set = indexes[train_index:val_index]
test_set = indexes[val_index:test_index]
```

Our script also needed to return a subset of the camera poses, called render poses. These poses would be used to create the reconstruction video. We chose to sample across all the poses with even spacing, which depended on the number of render poses we wanted. If N_r is the number of render poses and N_p is the total number of poses (equal to the

total number of images), then the step between the poses would be `step` $= \lfloor \frac{N_p}{N_r} \rfloor$. The set of render poses can then be extracted with the following slice,

```
render_poses = poses[::STEP,...]
```

5.4 Results from the NeRF Model

After the NeRF has been trained to a desired number of iterations, it uses the given render poses to generate RGB images of those viewpoints, as well as converting each frame into an MP4 video.

The render poses were chosen sequentially with even spacing between each one, allowing us to render the views of the robot travelling down the tunnel. Recall that we shuffled the set of indexes corresponding to the input images, which meant that the training set contained viewpoints randomly throughout the tunnel. As a result, some of the render frames in the video may have come from the training set, whereas others may have come from the test set.

If we only wanted to output novel views, we could add the `--render_test` flag, which creates a folder with RGB images of the test set poses. These indicate whether the NeRF model has learnt the 3D scene well. On the following page, we have shown one of the images from a test set pose when the model had been trained for 10,000 (Figure 5.4) and 85,000 (Figure 5.5) iterations.

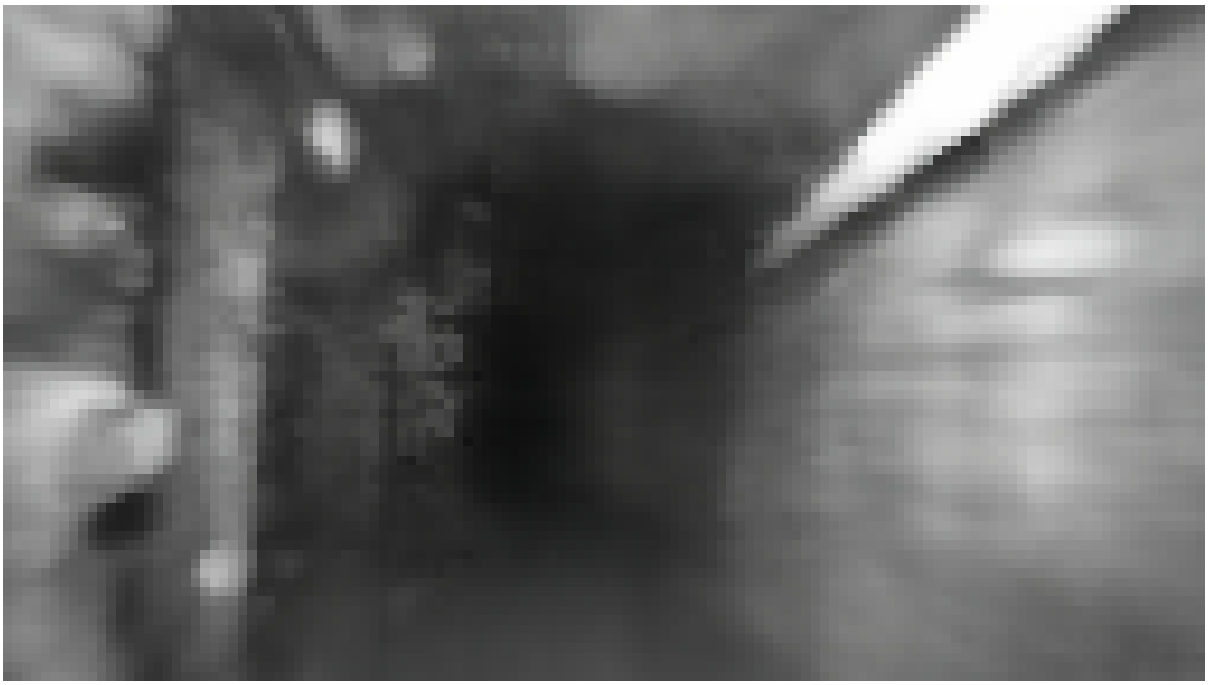


Figure 5.4: NeRF prediction of a test pose in the middle of the tunnel after 10,000 iterations

Figure 5.5: NeRF prediction of a test pose in the middle of the tunnel after 85,000 iterations

Note that the reconstructed images are pixelated as the model was trained on down-sampled images. We can visually observe an improvement in the output from 10,000 to 85,000 iterations, which is supported by the increase in the PSNR metric,

Iterations	**Test PSNR (dB)**
10,000	17.36
85,000	**22.85**

We have uploaded a video showing a sequence of all the RGB maps for each render pose which helps the reader to visualise the NeRF's reconstruction of the tunnel. The link to the video is: https://www.youtube.com/watch?v=6JWP4jrUqeM

5.5 Summary

In this chapter, we introduced the Onboard Illumination Visual- Inertial Odometry (OIVIO) dataset, which contains images taken by a robot in dark environments. Our focus was to train a NeRF on the underground tunnels in order to analyse the structure of caves. We acknowledged that the low lighting conditions would pose a challenge as most input images used to train a NeRF are well illuminated.

We then explained our decision to transition from the Google Colab notebook, which we had previously used to train the Lego tractor and building datasets, to a PyTorch-based NeRF implementation from a GitHub repository that matches the original paper, allowing us to take advantage of our high-performance lab PCs.

Finally, we presented our results of the RGB images of a test pose when the model had been trained for 10,000 and 85,000 iterations and compared the PSNR values, concluding that the output at 85,000 iterations was better.

6 | Analysing Cave Tunnels with the NeRF model

In Chapter 5, we trained a NeRF model on the OIVIO dataset to 85,000 iterations, specifically on the collection of underground cave tunnel images.

In this chapter, we will use the model to analyse how the neural network represents the structure of the tunnels. We will demonstrate our "explore mode", a unique way to move through a NeRF using keyboard input to generate novel views.

In addition to predicting the colour of a ray, NeRFs also output the density for each sample. By converting it to an opacity value, we will create a disparity map that shows the difference between structures closer to the camera and farther away in a grayscale image.

We will also attempt to analyse the back of the cave's structure by turning the camera around 180° and viewing the disparity map from this viewpoint.

Using the opacities, we will also plot their distribution for different parts of the tunnel to understand the geometry of the cave.

Finally, we will analyse the specular reflection effect on the cave walls by querying the NeRF model from a fixed viewpoint.

6.1 Explore Mode

After having trained a NeRF model, we wanted to explore through its reconstructed 3D scene by querying novel views. The difficulty was that the training set of camera poses were not aligned with the world axis, as we had explained in Section 4.4, therefore, we couldn't just move along, say the x-axis.

We implemented an "explore mode" which can be run by adding the `--explore` flag to the `run_nerf.py` script. This opens up an image, shown using the CV2 package, of the initial pose (i.e. pose 0 from the training set). We used the WASD navigation keys to update the pose to a novel view. Keys W and S move the camera forwards and backwards, while keys A and D turn the camera left and right by a constant angle. This allows us to translate as well as rotate the camera to view the tunnel from multiple viewing directions.

6.1.1 Navigation in explore mode explained

We will now provide more details on how we update the poses based on the keyboard input. The first step is to calculate the rotation vector, $\mathbf{r}$, from the camera's (3×3) pose matrix. This vector points in the direction of the camera, which we will use to move the camera a scalar multiple in that direction. If the scalar multiple, also called the step size s, is positive, the camera will move forwards in the scene. If it is negative, then the camera will move backwards. To get the updated pose matrix, we increment the translation component, which is the 4^{th} column, by $s\mathbf{r}$, which is the rotation vector multiplied by the step size.

In order to move the camera left or right by an angle θ, we need to calculate a vector, $\mathbf{r}'$, orthogonal to the rotation vector, $\mathbf{r}$. Using $\mathbf{r}'$ and θ, we can obtain the (3×3) rotation matrix representing a rotation of θ around the axis given by $\mathbf{r}'$. By multiplying

the camera's current pose matrix with the new pose matrix, we can update the pose, which corresponds to a horizontal rotation. Conventionally, rotation angles are counter-clockwise; thus, a positive θ will rotate the camera to the left, while a negative value of θ will rotate it to the right.

6.2 Disparity Map

A disparity map is a grayscale image that stores information associated with the distance of points from the camera at a particular viewpoint. A depth map is similar but stores data in metric units, such as meters, whereas disparity is unitless. We can convert the depth map into a disparity map by mapping it to the range $[0, 1]$. A value of 0 is shown as black and indicates the furthest point from the camera, while a value of 1 is shown as white and indicates the point closest to the camera.

6.2.1 Creating the depth map

The first stage is to create the depth map for a given pose. When we fire rays into the scene, we integrate over the following bounds, $[t_n, t_f]$. Let's partition the range into N (the number of samples) equally-spaced bins, so that t_i is the i^{th} bin. This allows us to calculate the distance between a sample and the next one on the ray,

$$\delta_i = t_{i+1} - t_i \tag{6.1}$$

The value of δ_N is set to ∞ as it is the last sample on the ray.

The NeRF model outputs a density, σ_i, for each sample, but instead we will use alpha-composting [10] to calculate the opacity, α_i, of a sample by mapping density to $[0, 1]$ using the following formula,

$$\alpha_i = 1 - e^{-\sigma_i \delta_i} \tag{6.2}$$

We also need to calculate accumulated transmittance, T_i, which is the probability that the ray has not hit any particles from the near bound up to the current bin, $[t_1, t_{i-1}]$, and is given by,

$$T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_i \delta_i\right) \tag{6.3}$$

The intuition behind this formula is that as the accumulated densities, $\sum_{j=1}^{i-1} \sigma_i \delta_i$, increase, the exponential decay causes the probability of the ray not hitting any particles to decrease.

We are now ready to calculate the depth of a ray, $D(\mathbf{r})$, which is given by,

$$D(\mathbf{r}) = \sum_{i=1}^{N} T_i \cdot \alpha_i \cdot t_i \tag{6.4}$$

We use the bin the sample is in, t_i, to ensure the depth is a metric unit in the range $[t_n, t_f]$.

6.2.2 Creating the disparity map

The depth map can't be plotted unless it is in the range $[0, 1]$, which is why we need to calculate the disparity of a ray, $\hat{D}(\mathbf{r})$. We take the depth map and inverse it,

$$\hat{D}(\mathbf{r}) = \frac{1}{\max(\epsilon, D(\mathbf{r}))}, \tag{6.5}$$

where ϵ is a small constant, e.g., 10^{-10}, to avoid the result from being infinite.

The final step will be to normalise the disparity map, which has the effect of increasing the contrast between objects closer and farther away from the camera as the pixels will have a higher dynamic range. We first calculate the minimum and maximum disparity

across all rays, and then normalise,

$$\text{normalised}(\hat{D}(\mathbf{r})) = \frac{\hat{D}(\mathbf{r}) - \min(\hat{D})}{\max(\hat{D}) - \min(\hat{D})} \tag{6.6}$$

6.2.3 Disparity map output

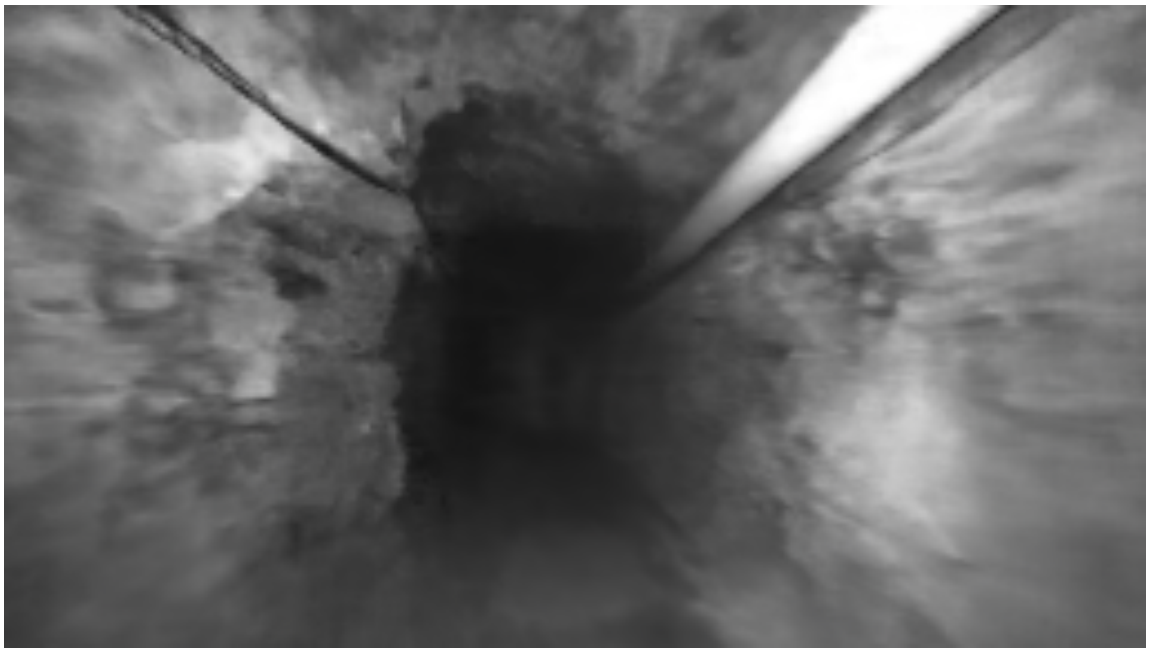


Figure 6.1: RGB map of the tunnel from its starting location

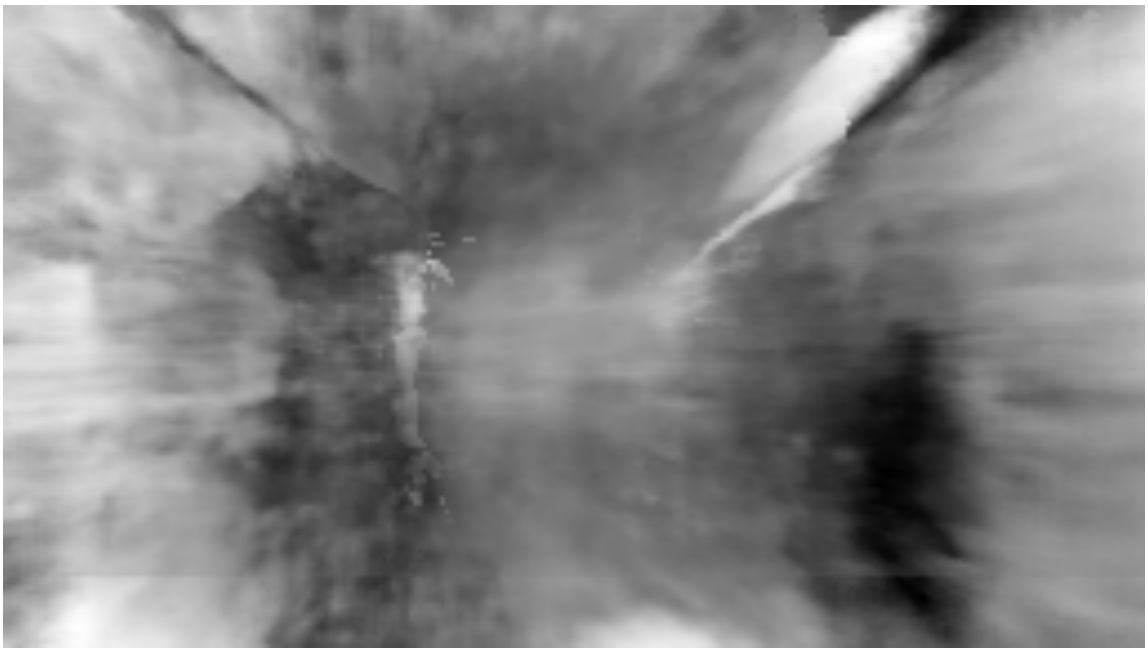


Figure 6.2: Disparity map of the tunnel from its starting location

We plotted the disparity map (Figure 6.2) for the starting view of the tunnel (Figure 6.1). Recall that white pixels refer to points closer to the camera, while black pixels refer to points furthest away. We can observe that the wall to the left of the camera appears bright, and the center of the tunnel is mostly grey. Interestingly, we notice that the structural pillar, which is not visible in the RGB map due to being occluded by a wall, is visible in the disparity map.

Figure 6.3 below shows a histogram of the pixel values (intensities) and their frequencies in the disparity map. A wide distribution is desired, as it ensures a significant contrast between surfaces closer and farther away from the camera.

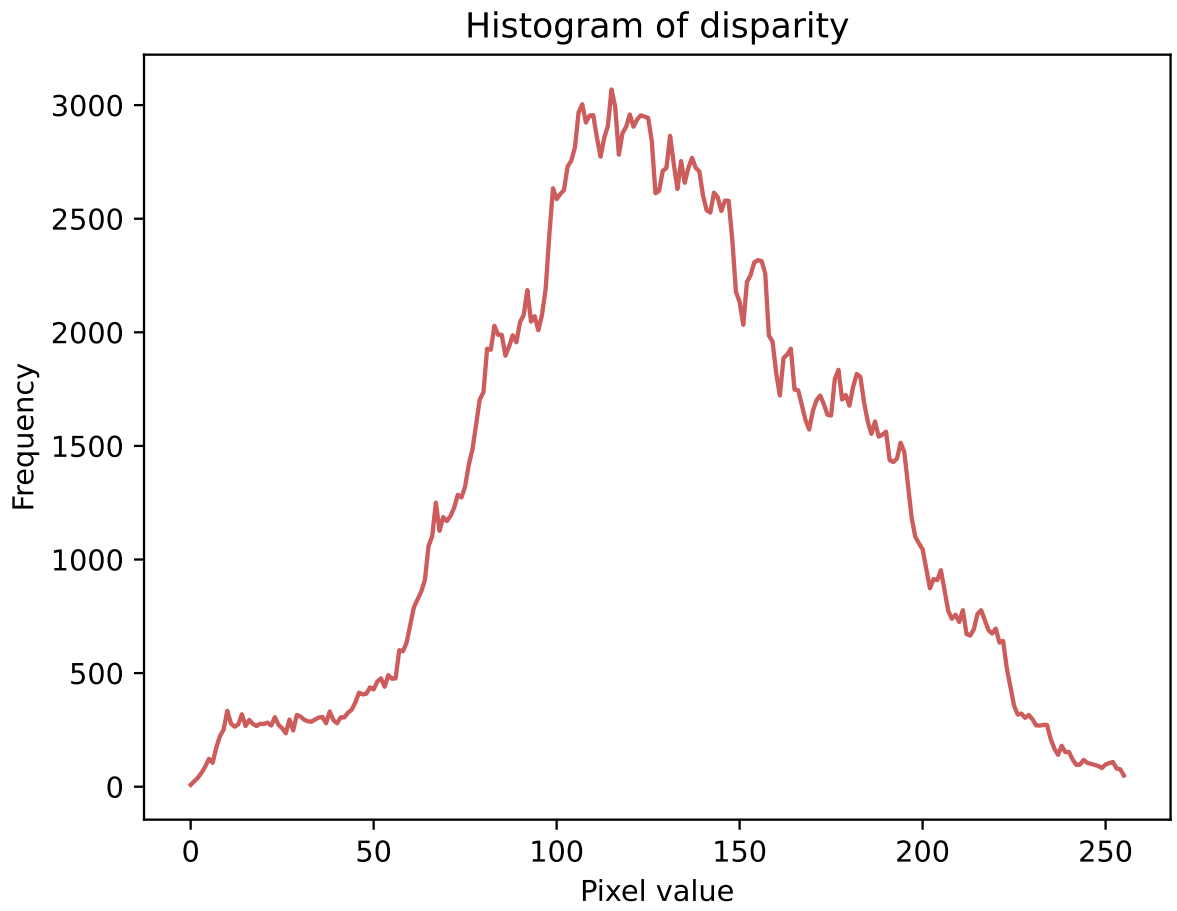


Figure 6.3: Histogram showing the distribution of pixel intensities in the disparity map grayscale image. A wide distribution is desired, as it ensures a significant contrast between surfaces closer and farther away from the camera.

6.3 Analysis of the Back of the Cave

Now that we are able to generate a disparity map, we wanted to analyse how well the NeRF model reconstructs a scene in reverse. This means having to rotate the camera 180° around so that it is facing in the opposite direction.

This is an interesting challenge because the training set of images was generated by sending the robot down the tunnel in only one direction. As a result, there was no training data for the tunnel in the reverse direction. Furthermore, the concept of rendering a novel view by turning the camera around is not present in much of the literature on NeRFs. Therefore, we believe that researching this would be an important contribution.

In Figure 6.4, we have shown the RGB map after the camera had been turned around in the middle of the cave so that the view would now show the start of the cave at the far end. Notice how the white pipe going down the tunnel is now rendered on the left-hand side, as opposed to being on the right (as seen in Figure 6.1), which tells us that the NeRF understands how the view changes after rotating the camera 180°. Another interesting aspect of the render is that the tunnel is seen curving towards the right, instead of towards the left in the normal direction.

Unfortunately, quite a bit of detail from the cave walls is lost in this render and the quality is far below that of the normal direction RGB map. The resulting PSNR for the reverse RGB map is 14.29dB, whereas it was 22.85dB for one of the test poses in the forward direction (Figure 5.5).

We generated the corresponding disparity map (Figure 6.5) and attempted to analyse the structure of the cave, however, we struggled to understand how the NeRF model was representing the reversed scene. While some parts of the tunnel walls were rendered white, meaning they were closer to the camera, the center part of the image is a mixture of black, grey, and white pixels. This indicates that the NeRF does a poor job of predicting

the correct density values for rays passing through the scene. This might explain why the RGB map is lacking detail as the colour of a pixel is a function of the opacities for that ray.

We feel further research needs to be done to understand how NeRFs render scenes in the reverse direction of the training images. We suggest taking a look at more recent architectures of NeRFs (Mip-NeRF, Ref-NeRF, etc.,) as they may be better at rendering the necessary details of the cave walls.

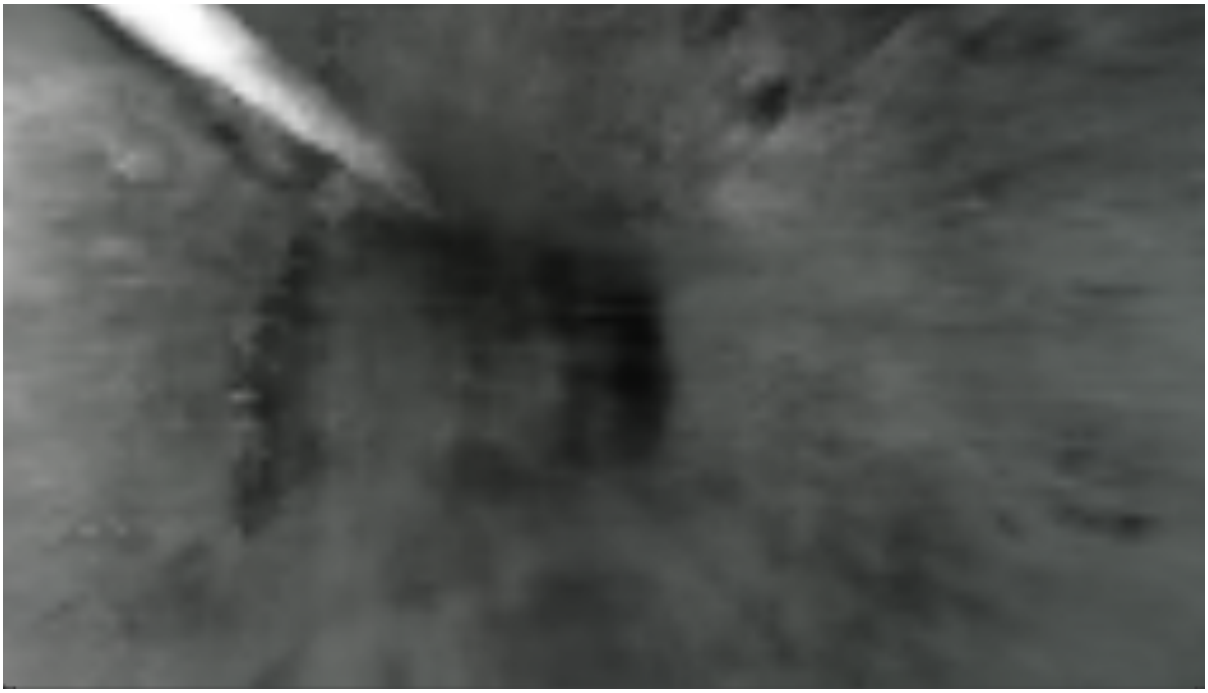


Figure 6.4: RGB map in the middle of the tunnel after the camera was turned 180° around.

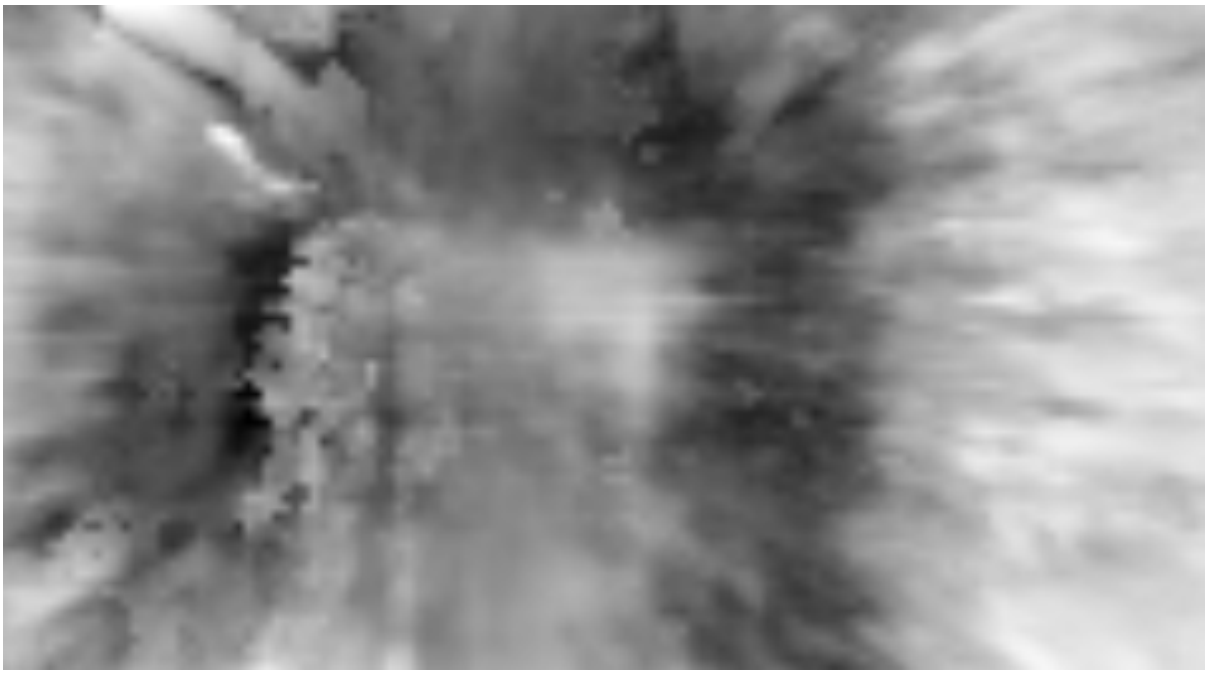


Figure 6.5: Disparity map in the middle of the tunnel after the camera was turned 180° around.

6.4 Plotting Ray Opacities

The density values predicted for each sample along a ray can give additional information about the geometry of the NeRF's representation of the cave. We will actually used opacity, α, which if you recall is density, σ, mapped to the range $[0, 1]$, using the following formula,

$$\alpha_i = 1 - e^{-\sigma_i \delta_i} \tag{6.7}$$

We hypothesised that a ray sent down the tunnel would have low opacity values in regions of empty space until it hits a tunnel wall, at which point there would be a spike to indicate the higher opacity surface of the walls.

To test whether the NeRF model has learnt this, we extended the "explore mode" to plot a graph of a ray's opacities when we press the "SPACE BAR" key. By using the "explore mode", we could navigate the camera to different viewpoints and then select a ray through that pose.

6.4.1 Selecting the center ray

We chose to select a ray that corresponded to the center of the viewpoint. As the opacities of all rays were stored in an array with the size (Num_Rays, Num_Samples), all we needed was the index position of the center ray, $\mathbf{r}_{\text{center}}$, to obtain its opacities. Given the viewpoint's width, W, and height, H, the index of the center ray is,

$$\mathbf{r}^{(i)}_{\text{center}} = W \cdot \frac{H}{2} + \frac{W}{2} \tag{6.8}$$

Note that when dividing by 2, if the height or width is odd, we round down to the nearest integer.

However, when we tried this approach, we received an index out-of-range error. It turns out that the rendering process for a viewpoint is performed in batches. If the ray batch size is smaller than the total number of pixels, Num_Rays $< W \cdot H$, then there will be multiple opacity arrays corresponding to each batch of rays. Therefore, the opacities of the center ray will be in only one of the batches.

Our solution was to keep track of the cumulative number of rays, CT, by adding up the number of rays in each batch. When this cumulative total is greater than the original center ray index, the current batch stores the opacities of the center ray. By subtracting the center ray index from the cumulative total, CT $- \mathbf{r}^{(i)}_{\text{center}}$, we get a new index position that can be used to access the opacities of the center ray.

6.4.2 Ray opacity plots

In the following figures, we have shown the RGB map for a particular viewpoint, along with its corresponding opacity plot. Each view includes a red pixel to highlight the center ray, indicating where the ray is being directed. One common feature among all the opacity plots is that the opacity reaches a maximum value of 1.0 at 6.0m. This occurs because the NeRF model assumes the scene is bounded and predicts a very large density value for the final sample on the ray at the far bound. This value then gets mapped to an opacity of 1.0.

In Figure 6.6, we used the initial render pose from the start of the tunnel because the center ray points straight down the tunnel. We expected to see very low opacities along the whole ray, as the middle of the tunnel is empty space, which is reflected in the plot. A slight bump in the opacity values after 5.5m is the result of the ray hitting a wall at the far end, which is not visible in the RGB map.

In Figure 6.7, we turned the camera towards a cave wall. In the plot, notice that the opacity values are low up to 3.4m before they start increasing. This is a result of the

distance between the camera and the wall, hence we can use opacity plots to estimate how far away surfaces are. Additionally, we observe from the plot that after 3.5m, the opacities increase and decrease in waves. This is because the cave wall is rough, with many uneven surface bumps that cause the ray to enter a dense section, come out of it, and then enter another dense section. This cycle repeats until about 5.7m when the opacity drops back to a low value, indicating that the ray has passed the cave wall. The NeRF model has assumed that there is empty space beyond the cave walls.

Finally, in Figure 6.8, we moved the camera down the tunnel to point at a structural pillar. Our hypothesis was that the opacities would be low until the ray hit the pillar, after which there would be a spike in values as the ray moved through the pillar, and then back to low values. The plot confirms our hypothesis, showing a spike just after 4.0m that lasts about 0.5m, which gives an estimate of the pillar's depth.

We then noticed a small bump in the opacity values at 5.0m, which could represent the wall behind the pillar that is not visible in the RGB map. The fact that these opacities fall back after a few meters suggests that the reconstructed wall appears to be thin.

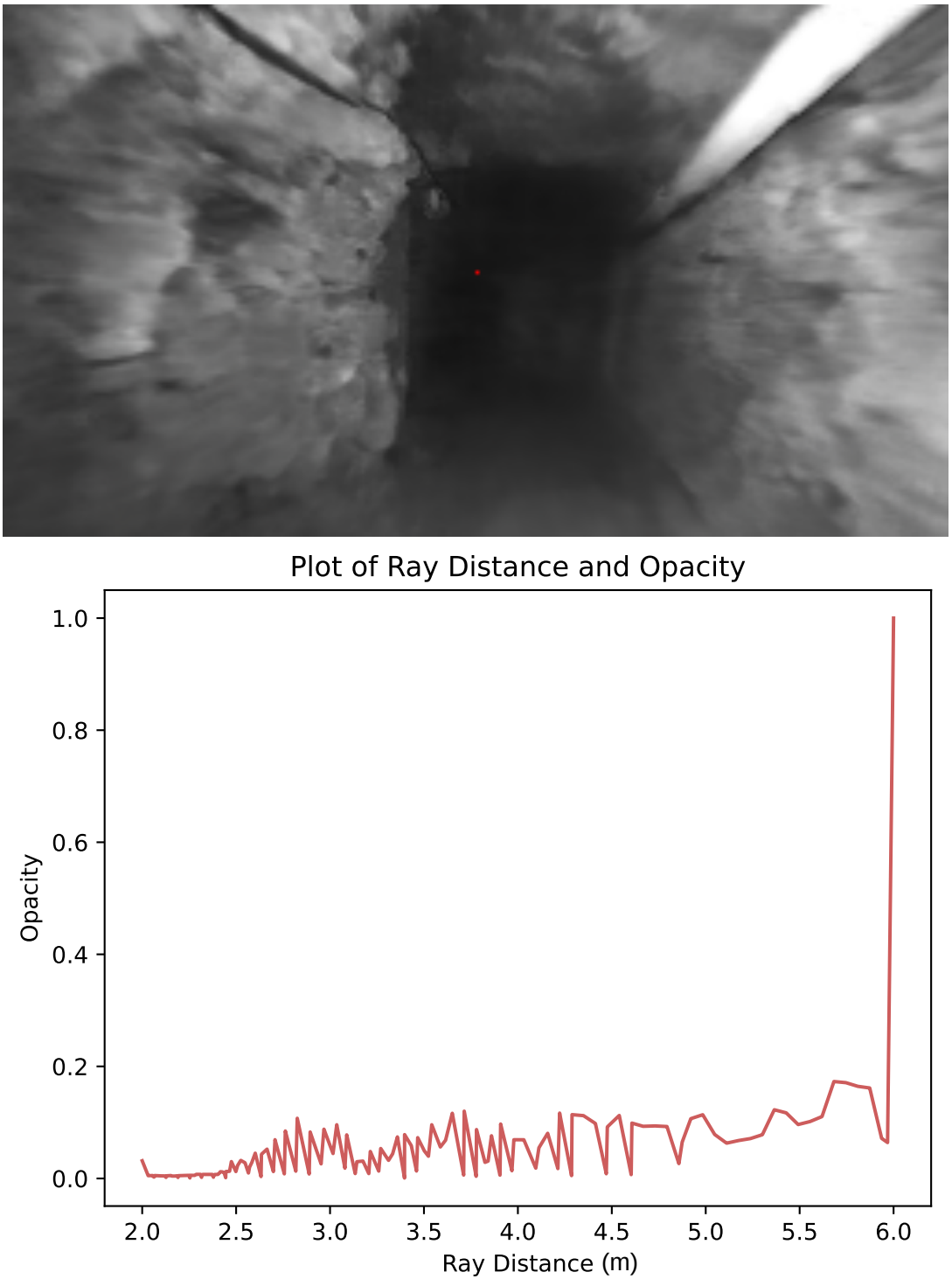


Figure 6.6: Top image is the RGB map of the tunnel from its starting location. The bottom image is a plot of the opacity values along the ray going through the center of this scene.

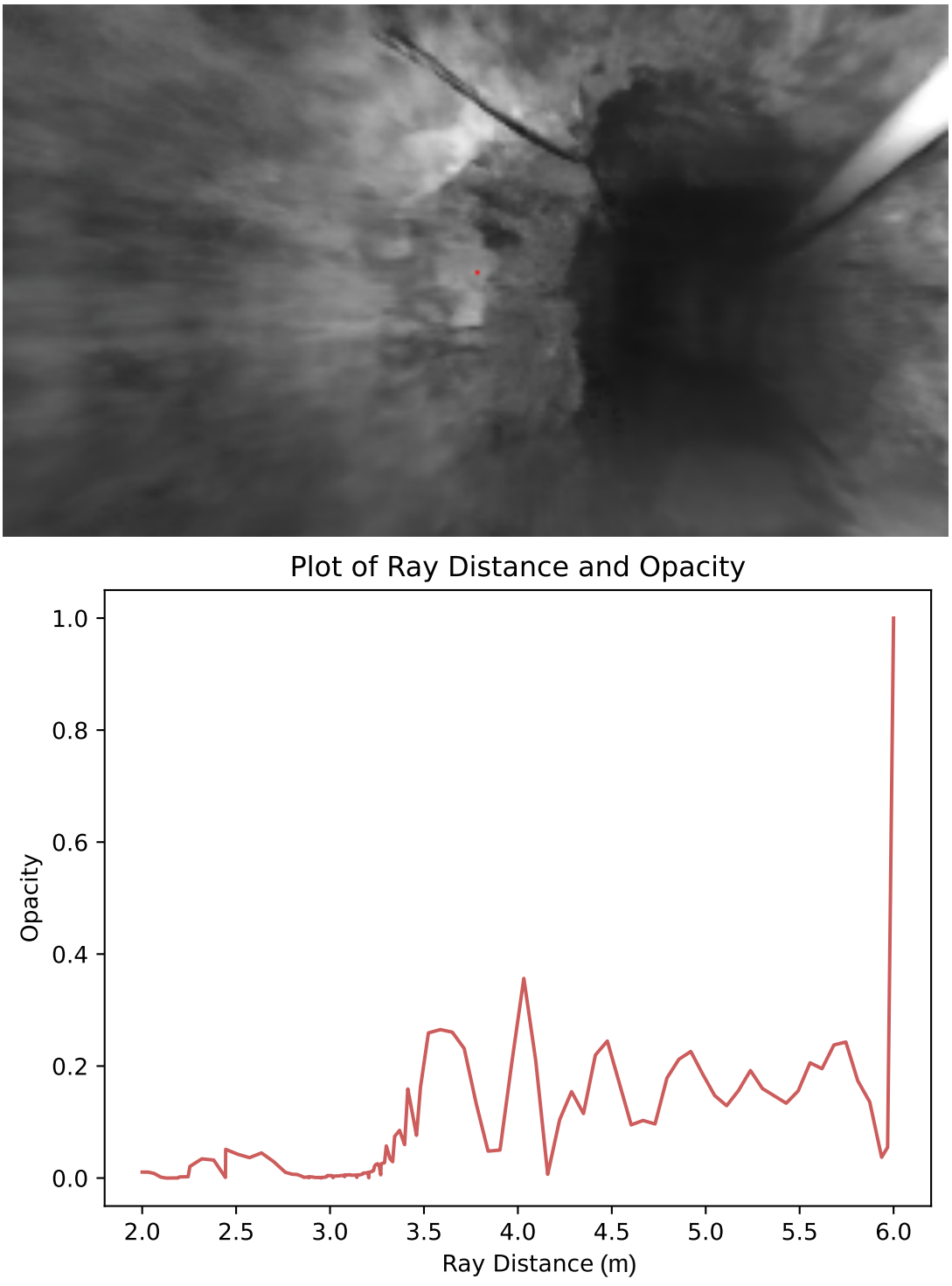


Figure 6.7: Top image is the RGB map of the camera oriented towards a wall. The bottom image is a plot of the opacity values along the ray going through the center of this scene.

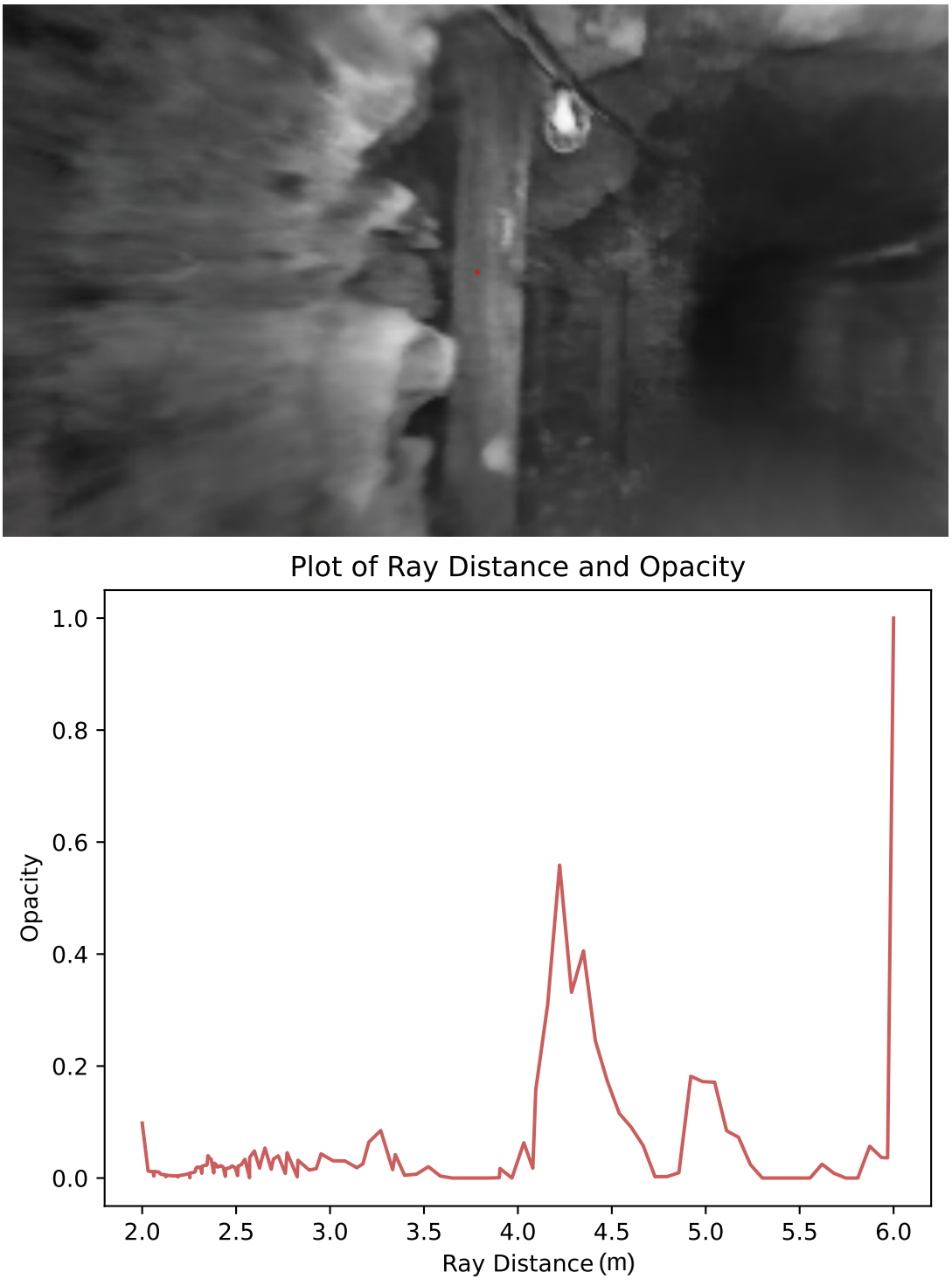


Figure 6.8: Top image is the RGB map of the camera looking at a structural pillar and the surrounding walls. The bottom image is a plot of the opacity values along the ray going through the center of this scene. Pillar View and its Opacity Plot

6.5 Fixed Viewpoint Specular Reflection

Back in Chapter 2, we explained that when implementing NeRFs, the output colour is a function of both the location and the viewing direction, while the volume density is only a function of the location as changing the camera's pose has no effect on the opacity of a point. We mentioned that we could take advantage of this by querying a NeRF model with a fixed pose but varying the input viewing direction. This technique allows us to produce non-Lambertian effects. Recall that this is when light scatters off smooth and shiny surfaces, which results in specular reflection.

Using our NeRF model, we wanted to observe how the light from the robot's flashlight reflected off the cave walls. By selecting a pose, we queried the NeRF model for all the render poses and viewed it from the fixed pose. We created a flag `--fixed_pose_index <index>` that takes an index position of a render pose to fix the view from. We then ran the process on three poses: 0, 25, and 42, which corresponded to the start, middle and end of the tunnel. Finally, we combined the renders from each viewing direction into MP4 videos.

Tunnel View	**Video Link**
Start	https://www.youtube.com/watch?v=Bz5xKuFbVR8
Middle	https://www.youtube.com/watch?v=mZ8gzh4r0x4
End	https://www.youtube.com/watch?v=Lrqmi_x6YdM

6.6 Summary

In this chapter, we performed an analysis of the cave tunnel using the trained NeRF model we had at the end of Chapter 5. We presented "explore mode", which enabled us to navigate through the NeRF model by generating novel views through translation and rotation using keyboard input.

We then explained how to calculate the disparity map given the densities along each ray, which allowed us to visualise how much closer or farther away a surface is from the camera in the grayscale output.

We attempted to analyse the structure of the scene in reverse by turning the camera around. While we were able to explain some parts of the RGB map, we struggled to understand the disparity map, and therefore, we suggested further research needed to be taken to improve the reconstruction of scenes in the opposite direction.

By converting the ray densities to opacities, we were able to plot a graph of the ray distance and the opacity value for the ray passing through the center of the scene. Using these plots, we could analyse the structure of the reconstructed cave walls and even identify structures not visible in the RGB map.

Finally, we demonstrated the specular reflection effect of light moving around the cave walls by fixing the viewpoint but varying the viewing directions as input to the NeRF model. We generated videos showing the effect for three different positions in the tunnel: start, middle, and end.

7 | Conclusion

7.1 Summary of Achievements

This project has achieved all the goals set out in the Introduction. A comprehensive literature review has been written up about Neural Radiance Fields and the COLMAP software. A NeRF model has been trained to 85,000 iterations on images of a cave tunnel from the OIVIO dataset and an analysis on the structure of cave walls has been carried out using the model. We were able to explore through the NeRF model using keyboard input, generate disparity maps, and produce videos showing the specular reflection on the cave walls.

7.2 Critical Evaluation

The first aim we set out in the Introduction was to understand the theory behind Neural Radiance Fields. After reading the original paper and related papers, we believe we have developed this understanding by covering it in Chapter 2.

The second aim was to learn how to use the COLMAP software package for extracting camera poses. We first built up an understanding of structure-from-motion for pose estimation the COLMAP pipeline. After running the software on the building dataset, where ground-truth poses were not available, we managed to build a script to extract the data from binary files.

For our third aim, we were successful in combining COLMAP and NeRF to train a model on a dataset of building images, which laid the foundation for applying this workflow to the OIVIO dataset. We overcame any out-of-memory issues by creating a script to downsample the input images and intrinsic camera parameters to a lower resolution.

However, our aim to understand how well NeRFs can represent a scene in reverse wasn't met fully. We were able to turn the camera around 180° and render a novel view using the model, but we struggled to understand the structure from the disparity map. If we had more time, we would explore more recent NeRF architectures to see if they can render the reversed scene in more detail.

In this project, we took an existing NeRF GitHub repository, which was written using the PyTorch framework to harness high-performance GPUs for training a NeRF, and extended it with several analysis tools to investigate the structure of cave tunnels. One such tool which we are proud of creating is "explore mode", which gave us a unique way to move around the NeRF model's reconstruction using keyboard input. Additionally, we were pleased with generating videos that showcase the predicted RGB maps through the tunnel and the specular reflection effect on cave walls.

Overall, we feel that this project has been a great contribution to the exciting field of research around NeRFs by focusing on applying them to dark environments and analysing how effectively they can reconstruct such scenes.

7.3 Future Work

Listed below is future work we could investigate to expand on our project

- Using more recent architectures of NeRFs, such as Mip-NeRF, Ref-NeRF, etc., to better understand the structure of the cave when turning the camera 180° around
- Investigate the effect of training a NeRF on different lighting conditions. The OIVIO dataset contains images of the same environment but taken using different amounts of illumination from the robot's flashlight.

References

[1] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: representing scenes as neural radiance fields for view synthesis," *Communications of the ACM*, vol. 65, pp. 99–106, 1 2022.

[2] D. Verbin, P. Hedman, B. Mildenhall, T. Zickler, J. T. Barron, and P. P. Srinivasan, "Ref-nerf: Structured view-dependent appearance for neural radiance fields," in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5481–5490, 2022.

[3] J. T. Kajiya and B. P. V. Herzen, "Ray tracing volume densities," *ACM SIGGRAPH Computer Graphics*, vol. 18, pp. 165–174, 7 1984.

[4] N. Rahaman, A. Baratin, D. Arpit, F. Draxler, M. Lin, F. Hamprecht, Y. Bengio, and A. Courville, "On the spectral bias of neural networks," in *Proceedings of the 36th International Conference on Machine Learning* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, pp. 5301–5310, PMLR, 09–15 Jun 2019.

[5] J. L. Schonberger and J.-M. Frahm, "Structure-from-motion revisited," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[6] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, pp. 91–110, 11 2004.

[7] C. Zhao, Z. Cao, C. Li, X. Li, and J. Yang, "Nm-net: Mining reliable neighbors for robust feature correspondences," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[8] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment — a modern synthesis," in *Vision Algorithms: Theory and Practice* (B. Triggs, A. Zisserman, and R. Szeliski, eds.), (Berlin, Heidelberg), pp. 298–372, Springer Berlin Heidelberg, 2000.

[9] M. Kasper, S. McGuire, and C. Heckman, "A Benchmark for Visual-Inertial Odometry Systems Employing Onboard Illumination," in *Intelligent Robots and Systems (IROS)*, 2019.

[10] A. Tagliasacchi and B. Mildenhall, "Volume rendering digest (for nerf)," 8 2022.

Bibliography

The following resources are additional materials that are not included in the references but have aided in the research for this project.

- McGough, Mason. (2022) *It's NeRF From Nothing: Build A Complete NeRF with PyTorch*. https://towardsdatascience.com/its-nerf-from-nothing-build-a-vanilla-nerf-with-pytorch-7846e4c45666
- Anwar, Aqeel. (2022) *What are Intrinsic and Extrinsic Camera Parameters in Computer Vision?*. https://towardsdatascience.com/what-are-intrinsic-and-extrinsic-camera-parameters-in-computer-vision-7071b72fb8ec
- COLMAP Documentation. https://colmap.github.io/
- Point Cloud & Camera Basics Notebook. https://colab.research.google.com/drive/1sF3kF9NDMA9PZSgKHghMiYhJYNc3VXol#scrollTo=MbAKBlH5qffG&uniqifier=7

A | Project Plan

This is a copy of the Project Plan submitted on 9th November 2022.

Project Title: Using Neural Radiance Fields (NeRFs) on Dark Cave Scenes

Supervisor's Name: Simon Julier

External Supervisor's Name: N/A

Aim: To learn how to represent dark/dimly-lit cave environments using Neural Radiance Fields (NeRFs). Then trying to query the neural network to produce novel views and synthesising them.

Objectives:

1. Read and understand the original NeRF paper and find existing code that implements it on a toy dataset
2. Investigate the Onboard Illumination Visual-Inertial Odometry (OIVIO) dataset which consists of dark environments such as mines, tunnels, caves
3. Apply a NeRF to this dataset to represent dark environments which can be harder to learn (than well-lit environments)
4. Evaluate how well the NeRF can output novel views of a 3D cave scene and find applications of using this representation for robotics, scene mapping

Deliverables:

- A literary review/summary of the original NeRF paper explained to a general audience
- Results obtained from novel view synthesis using the NeRF and discussion of how significant the output is

- A fully documented and functional NeRF algorithm trained on dark cave scenes
- A specification for using the trained NeRF algorithm

Work Plan:

- Pre-October/Summer: Literary research on NeRFs, Computer Vision/Graphics and research papers from related domains
- October to Mid-November: Start initial iteration of prototyping. Each prototype will undergo design, implementing, testing, refactoring (if necessary):
  - Running an implementation of the original NeRF on toy datasets
  - Learning to use the COLMAP software package to extract pose information from a set of images
  - Using COLMAP on OIVIO dataset to create the input training data for the NeRF algorithm
  - Training a NeRF on the OIVIO dataset
- Mid-November:
  - Completing Project Plan
  - Completing Ethics review
- Mid-November to Mid-January:
  - Working on Interim Report
  - Wrapping up prototypes into a single piece of software and adding any extra features. This should ideally be completed by End-December
  - Writing up the documentation/specification of the software
- Mid-March: Finishing Video Preview

- Early-April: Making sure Final Report is nearly complete and ready for submission by MidApril

Ethics Review: I believe that this project has no ethical issues.

The project is not sensitive as it doesn't involve data of people, animals, nor sensitive/offensive material. Further, the project won't involve undertaking interviews/questionnaires with the general public.

The primary dataset I will be working with, OIVIO, can be used in research given I include the citation of their paper. The data consists of images recorded in dark environments that don't include people/animals.

B | Interim Report

This is a copy of the Interim Report submitted on 18th January 2023.

Project Title: On Neural Radiance Fields (NeRFs) for Analysing Dark Cave Scenes **Supervisor's Name:** Simon Julier **External Supervisor's Name:** N/A

Progress Made to Date

Project Plan Objectives Completed

These are the objectives I set in my Project Plan which have been completed so far:

1. Read and understood the original NeRF paper and found existing code that implements it on a toy dataset
2. Examined the Onboard Illumination Visual-Inertial Odometry (OIVIO) dataset, which comprises of environments with low lighting such as caves, tunnels, and mines. Then downloaded some sample data from this dataset to train a NeRF on later

Additional Tasks Completed

This is work that has been completed so far which includes prototyping and testing. This work builds the foundation to achieve the main objectives for the project.

1. Downloaded the COLMAP GUI and learned how to use the software to perform pose extraction from a set of images
2. Downloaded a set of images of a building to use with COLMAP for 3D reconstruction and to obtain the pose information

3. Written a script to downsample a set of images to a lower resolution to make it possible to train a NeRF model
4. Written a script to extract the data from text files generated by COLMAP. This includes extracting the images (stored as NumPy arrays), the intrinsic focal length parameter, and the extrinsic (4x4) pose matrices for each viewing direction. All of this data was then collated into a single dataset stored as a NumPy zip
5. Used an implementation of the original NeRF on a Jupyter notebook to train a model on a popular toy dataset, a Lego tractor, using the already provided pose data
6. Used the same notebook to train another model on the custom building dataset, whose pose data was extracted using COLMAP
7. Downloaded a GitHub repository that trains a NeRF using a python script instead of a notebook, which allowed me to train larger models. Applied the script to the building dataset to generate a 360° reconstruction video containing novel views of the scene
8. Applied COLMAP to the OIVIO dataset for two sections of the cave, the initial starting scene and a tunnel
9. Trained a larger NeRF model, using higher computing resources, on two parts of the OIVIO dataset - the initial scene with a calibration QR code and a tunnel sequence

Remaining Work to be Done

January:

- Use the model to generate novel views of the underground cave network
- Use keyboard/mouse input to move around the 3D scene generated by the NeRF model

February:

- Fix a viewpoint and change the viewing direction to observe the NeRF's predictions of specular reflection of the cave walls
- Use the NeRF model to analyse the structure of cave walls, such as spikiness, bumpiness, and reflections

C | Code Listing

The full project repository is available at: https://github.com/surajvkothari/Final-Year-Project-UCL

downsample.py

Downsamples the training images to a lower resolution so that they can fit into the GPU for training.

```
import cv2
import glob
import os
import tqdm

IMAGE_DIR = "../COLMAP Projects/OIVIO Tunnel/images/*"
# Makes sure downsamples folder exists (cv2 doesn't create one)
DOWNSAMPLE_DIR = "../COLMAP Projects/OIVIO Tunnel/downsamples/"

DOWNSAMPLE_SCALE_FACTOR = 10

images = glob.glob(IMAGE_DIR)  # Gets list of all images in directory

for img_path in tqdm.tqdm(images):
    image = cv2.imread(img_path)
    # Join downsampled directory with image filename
    downsample_path = os.path.join(DOWNSAMPLE_DIR, os.path.basename(
   img_path))


    # Divide width, height by scale factor
    downsample_size = (image.shape[1] // DOWNSAMPLE_SCALE_FACTOR,
```

```
                        image.shape[0] // DOWNSAMPLE_SCALE_FACTOR,)

    image_downsample = cv2.resize(image, downsample_size)

    # Save downsampled image
    cv2.imwrite(downsample_path, image_downsample)
```

get_data_for_NeRF.ipynb

This is a Jypyter notebook to extract the camera extrinsics from COLMAP. We have separated the code into blocks that correspond to the cells in the notebook.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.spatial.transform import Rotation
import cv2
import os
%matplotlib inline
```

```
downsampled_image_dir = "../COLMAP Projects/OIVIO Tunnel/downsamples/"
sparse_dir = "../COLMAP Projects/OIVIO Tunnel/workspace/sparse/0/"

SCALE_FACTOR = 10  # Same scale factor used for image downsampling
```

```
cameras_txt = os.path.join(sparse_dir, "cameras.txt")
images_txt = os.path.join(sparse_dir, "images.txt")

cameras_cols = ["Camera_ID", "Model", "Width", "Height", "Focal", "
    Param_1", "Param_2", "Param_3"]
images_cols = ["Image_ID", "qw", "qx", "qy", "qz", "tx", "ty", "tz", "
    Camera_ID", "Filename"]

```

```
# Make sure cameras data has only 1 camera (Simple_Radial)
cameras_data = pd.read_csv(cameras_txt, sep=" ", names=cameras_cols,
                           skiprows=range(3), index_col="Camera_ID")  # Skip first 3 rows

# Custom skip rows function
def image_skip_rows(index):
    # Skip first 4 rows as they are comments
    if index in range(4):
        return True
    # Skip odd indexed rows as it is unnecessary points data
    elif index % 2 == 1:
        return True
    else:
        return False

images_data = pd.read_csv(images_txt, sep=" ", names=images_cols,
                          skiprows=lambda x: image_skip_rows(x), index_col="Image_ID")  # Skip first 4 rows
images_data
```

```
focal_length = cameras_data["Focal"][1]  # Get focal length from camera 1

# Divide focal length by scale factor
focal_length = focal_length / SCALE_FACTOR
```

```
images = []
image_filenames = images_data["Filename"]
for name in image_filenames:
    image_path = os.path.join(downsampled_image_dir, name)  # Get path to downsampled image

    image = cv2.imread(image_path)  # Read image into array
```

```
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Convert to RGB
    images.append(image)

# Divide by 255 to get in range [0, 1] and convert to float
images = (np.array(images) / 255).astype(np.float32)
```

```
quaternions = images_data[["qx", "qy", "qz", "qw"]].to_numpy()  # X,Y,Z,W is the format needed for from_quat() below

# For N quaternion vectors, get the Nx3x3 rotation matrices
# Reference: https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.transform.Rotation.html
rotations = Rotation.from_quat(quaternions).as_matrix()

# Get translation vectors
translations = images_data[["tx", "ty", "tz"]].to_numpy()

poses = []
# Create Nx4x4 pose matrix for N images
for i in range(len(images)):
    pose = np.column_stack((rotations[i], translations[i]))  # Combine rotation matrix and translation vector along columns
    pose = np.vstack((pose, [0.0, 0.0, 0.0, 1.0]))  # Make matrix into square (4x4) by adding this row vector

    # COLMAP stores data in world-to-camera coordinates. Convert to camera-to-world coordinates by inverting the pose matrix
    pose = np.linalg.inv(pose)

    # Matrix to flip the sign of the y and z rotation axes. No need to flip the x rotation axis
    # Reference: https://github.com/NVlabs/instant-ngp/blob/master/scripts/colmap2nerf.py
```

```
    flip_mat = np.array([
        [1, 0, 0, 0],
        [0, -1, 0, 0],
        [0, 0, -1, 0],
        [0, 0, 0, 1]
    ])

    pose = pose @ flip_mat  # Apply flip matrix to pose matrix

    poses.append(pose)

poses = np.array(poses).astype(np.float32)  # Convert to float
```

```
# Reference: https://colab.research.google.com/drive/1TppdSsLz8uKoNwqJqDGg8se8BHQcvg_K?usp=sharing
dirs = np.stack([np.sum([0, 0, -1] * pose[:3, :3], axis=-1) for pose in poses])
origins = poses[:, :3, -1]

ax = plt.figure(figsize=(24, 64)).add_subplot(projection='3d')
_ = ax.quiver(
    origins[..., 0].flatten(),  # x-coord of origin
    origins[..., 1].flatten(),  # y-coord of origin
    origins[..., 2].flatten(),  # z-coord of origin
    dirs[..., 0].flatten(),  # x-coord of direction endpoint
    dirs[..., 1].flatten(),  # y-coord of direction endpoint
    dirs[..., 2].flatten(),  # z-coord of direction endpoint
    length=0.5, normalize=True)
plt.show()
```

```
np.savez("oivio_tunnel.npz", images=images, poses=poses, focal=focal_length)
```

load_npz.py

This script loads the data from the NPZ files.

```
import numpy as np

def load_npz_data(data_dir):
    """ Loads data from a numpy .npz dataset """
    dataset = np.load(data_dir)
    images, poses, FOCAL = dataset["images"], dataset["poses"], dataset["focal"]

    NUM_IMAGES = images.shape[0]
    HEIGHT, WIDTH = images.shape[1:3]

    train_index = int(NUM_IMAGES*0.8)
    val_index = train_index + int(NUM_IMAGES*0.1)
    test_index = val_index + int(NUM_IMAGES*0.1)

    indexes = np.arange(NUM_IMAGES)
    np.random.shuffle(indexes)  # Random shuffle indexes in-place

    # Select train, val, test from shuffled indexes
    i_split = [indexes[:train_index], indexes[train_index:val_index], indexes[val_index:test_index]]

    # Render poses are a sample of original poses
    NUM_RENDER_POSES = 50
    STEP = int(NUM_IMAGES / NUM_RENDER_POSES)  # Round up to next integer
    render_poses = poses[::STEP,...]  # Select every <STEP> poses

    return images, poses, render_poses, [HEIGHT, WIDTH, FOCAL], i_split
```

run_nerf.py

This is the main script that trains the NeRF model. We extended this script to include our analysis tools. Each subsection below is a section of this script. The whole script is provided in our GitHub repository.

Disparity Map and Ray Opacities

This function takes the raw output from the NeRF and converts it into the RGB map, the disparity map, and computes the opacities for the ray going down the center of the scene.

```
def raw2outputs(raw, z_vals, rays_d, HEIGHT, WIDTH, cumulative_num_rays
    , ray_shown, raw_noise_std=0, white_bkgd=False, pytest=False,
    get_plot_data=False):
    """Transforms model's predictions to semantically meaningful values
    .
    Args:
        raw: [num_rays, num_samples along ray, 4]. Prediction from
    model.
        z_vals: [num_rays, num_samples along ray]. Integration time.
        rays_d: [num_rays, 3]. Direction of each ray.
    Returns:
        rgb_map: [num_rays, 3]. Estimated RGB color of a ray.
        disp_map: [num_rays]. Disparity map. Inverse of depth map.
        acc_map: [num_rays]. Sum of weights along each ray.
        weights: [num_rays, num_samples]. Weights assigned to each
    sampled color.
        depth_map: [num_rays]. Estimated distance to object.
    """
    SHOW_RAY = True

```

```
16      raw2alpha = lambda raw, dists, act_fn=F.relu: 1.-torch.exp(-act_fn(
    raw)*dists)
17
18      dists = z_vals[...,1:] - z_vals[...,:-1]
19      dists = torch.cat([dists, torch.Tensor([1e10]).expand(dists
    [...,:1].shape)], -1)  # [N_rays, N_samples]
20
21      dists = dists * torch.norm(rays_d[...,None,:], dim=-1)
22
23      rgb = torch.sigmoid(raw[...,:3])  # [N_rays, N_samples, 3]
24      noise = 0.
25      if raw_noise_std > 0.:
26          noise = torch.randn(raw[...,3].shape) * raw_noise_std
27
28          # Overwrite randomly sampled data if pytest
29          if pytest:
30              np.random.seed(0)
31              noise = np.random.rand(*list(raw[...,3].shape)) *
    raw_noise_std
32              noise = torch.Tensor(noise)
33
34      # Alpha is the opacities, which is densities mapped to [0,1]
35      alpha = raw2alpha(raw[...,3] + noise, dists)  # [N_rays, N_samples]
36
37      """
38      Calculate accumulated transmittance: the probability that the ray
    travels
39      from near bound to far bound without hitting any other particle
40      """
41      # weights = alpha * tf.math.cumprod(1.-alpha + 1e-10, -1, exclusive
    =True)
42      accumulated_transmittance = torch.cumprod(torch.cat([torch.ones((
    alpha.shape[0], 1)), 1.-alpha + 1e-10], -1), -1)[:, :-1]
```

```
43      weights = alpha * accumulated_transmittance
44      rgb_map = torch.sum(weights[...,None] * rgb, -2)  # [N_rays, 3]
45
46      """
47      To calculate the depth (distance) of a ray, we multiply the
48      weights (alpha * accumulated_transmittance) by z_vals, which gives
    higher values
49      to the weights further along the ray.
50
51      This means that if the weights were distributed near the front of
    the ray, then
52      they would get a lower distance value than if the weights were
    distributed
53      towards the end of the ray.
54      """
55      depth_map = torch.sum(weights * z_vals, -1)
56
57      # Disparity is inverse of depth (1/depth)
58      disp_map = 1./torch.max(1e-10 * torch.ones_like(depth_map),
    depth_map / torch.sum(weights, -1))
59
60      # Normalise disparity map
61      disp_min, disp_max = torch.min(disp_map), torch.max(disp_map)
62      disp_map = (disp_map - disp_min) / (disp_max - disp_min)
63
64      acc_map = torch.sum(weights, -1)  # Accumulated weights map
65
66      if white_bkgd:
67          rgb_map = rgb_map + (1.-acc_map[...,None])
68
69      # Only get ray info if we want to plot and the ray hasn't been
    rendered already
70      if get_plot_data and not(ray_shown):
```

```
71          center_ray_index = WIDTH*(HEIGHT//2) + WIDTH//2  # Get index of
    ray in the flattened image
72
73          center_ray_batch_index = cumulative_num_rays - center_ray_index
     # Get index of the ray in the current batch
74
75          # Check the index is positive, otherwise the ray is not in the
    current batch
76          if center_ray_batch_index > 0:
77              # Get ray distances and opacities of the ray that goes
    through the center of the image
78              ray_distances = z_vals[-(center_ray_batch_index)]
79              opacities = alpha[-(center_ray_batch_index)]
80
81              if SHOW_RAY:
82                  # Shows ray as red pixel
83                  #rgb_map[-(center_ray_batch_index)] = torch.Tensor([1,
    0, 0])
84                  ray_shown = True
85          else:
86              ray_distances, opacities = None, None
87      else:
88          ray_distances, opacities = None, None
89
90      return rgb_map, disp_map, acc_map, weights, depth_map,
    ray_distances, opacities, ray_shown
```

Plot Ray Opacities and Histogram

Generates a plot of the ray distance with the ray opacities. The other function plots a histogram showing the frequency of each pixel's intensity in a given image.

```
def plot_ray_opacities(ray_distances, opacities):
    """ Plots a graph of the ray's distance with the opacity at those
   points """
    plt.plot(ray_distances, opacities, c="indianred")

    plt.xlabel("Ray Distance")
    plt.ylabel("Opacity")
    plt.title("Plot of Ray Distance and Opacity")

    plt.show()


def plot_image_histogram(img):
    hist = cv2.calcHist([img],[0],None,[256],[0,1])
    plt.plot(hist, c="indianred")

    plt.xlabel("Pixel value")
    plt.ylabel("Frequency")
    plt.title("Histogram of disparity")

    plt.savefig(f"histogram.pdf", format="pdf", bbox_inches="tight")

    plt.show()
```

Explore mode

Our novel approach to exploring through a NeRF model using keyboard input.

```
def explore(explore_pose, hwf, K, chunk, render_kwargs, initial_pose):
    """ Explores through the NeRF model """

    H, W, focal = hwf

    SCALE_WINDOW = 3

    # Rendering
    with torch.no_grad():
        explore_pose_tensor = torch.Tensor(explore_pose).to(device)

        rgb, disp, acc, depth, all_returns = render(H, W, K, chunk=
    chunk, c2w=explore_pose_tensor[:3,:4], **render_kwargs)

        # Convert RGB and disparity maps to Numpy arrays to be
    displayed in CV2 imshow
        rgb_map, disp_map = rgb.cpu().detach().numpy(), disp.cpu().
    detach().numpy()
        rgb_map = cv2.cvtColor(rgb_map, cv2.COLOR_RGB2BGR)
        # Scale output to be larger for displaying
        rgb_map, disp_map = cv2.resize(rgb_map, (W*SCALE_WINDOW, H*
    SCALE_WINDOW)), cv2.resize(disp_map, (W*SCALE_WINDOW, H*SCALE_WINDOW
    ))

        cv2.imshow("Explore", rgb_map)

    key = cv2.waitKey(0)

    # Up/W
    if key == ord('w'):
```

```
26          explore_pose = update_pose(key="up", pose=explore_pose)
27
28      # Down/S
29      elif key == ord('s'):
30          explore_pose = update_pose(key="down", pose=explore_pose)
31
32      # Left/A
33      elif key == ord('a'):
34          explore_pose = update_pose(key="left", pose=explore_pose)
35
36      # Right/D
37      elif key == ord('d'):
38          explore_pose = update_pose(key="right", pose=explore_pose)
39
40      # Reset view
41      elif key == ord('r'):
42          explore_pose = np.copy(initial_pose)
43
44      # Display disparity map
45      elif key == ord('m'):
46          cv2.imshow("Disparity", disp_map)
47          plot_image_histogram(disp_map)
48
49      # Plot ray density graph
50      elif key == ord(' '):
51          if "ray_distances" in all_returns and "opacities" in
    all_returns:
52              ray_distances = all_returns["ray_distances"].cpu().detach()
    .numpy()
53              opacities = all_returns["opacities"].cpu().detach().numpy()
54              plot_ray_opacities(ray_distances, opacities)
55
56      # If <ESC> (27) is pressed, stop program
```

```
57      elif key == 27:
58          return
59
60      explore(explore_pose, hwf, K, chunk, render_kwargs, initial_pose=
        initial_pose)
61
62
63  def update_pose(key, pose):
64      """ Updates pose matrix based on movement key """
65      STEP_SIZE = 0.2
66      ROTATION_ANGLE = 10
67
68      rotation_matrix = pose[:3, :3]
69      rotation_vector = Rotation.from_matrix(rotation_matrix).as_rotvec()
70
71      if key == "up":
72          # Apply 90-degree rotation to rotation vector to face forwards
73          rotation_operation = Rotation.from_euler('y', 90, degrees=True)
74          rotation_vector = rotation_operation.apply(rotation_vector)
75
76          # Move in the direction of the rotation vector
77          pose[:3, -1] += STEP_SIZE * rotation_vector
78
79      elif key == "down":
80          # Apply 90-degree rotation to rotation vector to face forwards
81          rotation_operation = Rotation.from_euler('y', 90, degrees=True)
82          rotation_vector = rotation_operation.apply(rotation_vector)
83
84          # Move in the direction of the rotation vector
85          pose[:3, -1] -= STEP_SIZE * rotation_vector
86
87      elif key == "left":
```

```
88          # Get rotation matrix for rotation around an axis by given
    angle
89          rotation_operation = Rotation.from_euler('y', ROTATION_ANGLE,
    degrees=True).as_matrix()
90          # Apply operation
91          new_rotation_matrix = rotation_matrix @ rotation_operation
92
93          pose[:3, :3] = new_rotation_matrix
94
95      elif key == "right":
96          # Get rotation matrix for rotation around an axis by given
    angle
97          rotation_operation = Rotation.from_euler('y', -ROTATION_ANGLE,
    degrees=True).as_matrix()
98          # Apply operation
99          new_rotation_matrix = rotation_matrix @ rotation_operation
100
101         pose[:3, :3] = new_rotation_matrix
102
103     return pose
```

Fixed-Pose Specular Reflection Videos

The `--fixed_pose_index` flag creates a video of the specular reflection for a given pose.

```
if args.fixed_pose_index is not None:
        """ Fix camera but use render poses to change viewing direction
    """
        moviebase = os.path.join(basedir, expname, f"{expname}_{start
    +1:06d}_")
        render_kwargs_test["c2w_staticcam"] = render_poses[args.
    fixed_pose_index][:3,:4]  # Fixed camera pose (index given by
    argument)

        with torch.no_grad():
            rgbs_static, _, _ = render_path(render_poses, hwf, K, args.
    chunk, render_kwargs_test)

        render_kwargs_test["c2w_staticcam"] = None
        imageio.mimwrite(moviebase + f"fixed_view_{args.
    fixed_pose_index}.mp4", to8b(rgbs_static), fps=30, quality=8)

        return
```

D | User Manual

This user manual will provide a guide to using the extended features of this project. The GitHub repository for this project is available here: https://github.com/surajvkothari/Final-Year-Project-UCL

Install Dependencies

```
pip install -r requirements.txt
```

You will also need access to a GPU and PyTorch enabled with CUDA.

Getting the data

Our data is stored in NumPy zips (.npz) and you will need to download the datasets from this Google Drive link: https://drive.google.com/drive/folders/1lGJcPAoUxMEKT189W4GutoEF1wCMePGY?usp=share_link.

After downloading, unzip the datasets folder and move the .npz files into the **data** folder in the project.

Using a configuration

We have trained a NeRF model of the OIVIO dataset to 85,000 iterations. The same model has been applied to different resolutions of the data. When using any command, please replace <config_filename> with any of the following options:

- OIVIO_tunnel_10x_downsampled.txt
- OIVIO_tunnel_5x_downsampled.txt

- OIVIO_tunnel_2x_downsampled.txt
- OIVIO_tunnel_full_resolution.txt

Note: We recommend to use the "OIVIO_tunnel_10x_downsampled.txt" option for most hardware. The other options result in a better resolution output, however may result in an Out-Of-Memory error.

Explore Mode

Add the --explore flag,

```
python run_nerf.py --config configs/<config_filename> --explore
```

Navigating in Explore Mode

After running the explore mode command, you will be shown an image of view from the start of the tunnel. Use keyboard input to move or rotate around the NeRF model's reconstruction.

- W, S - Moves forwards and backwards
- A, D - Rotates the camera left and right
- R - Resets the viewpoint to the start of the tunnel

Note: When a key is pressed, there may be a delay until the next frame shows.

Plotting Ray Opacities

After running explore mode, move to a desired viewpoint, and the use the Space bar key which will display a plot of the opacities for the ray going through the center of the image.

Note: After the plot is displayed, keyboard input to move around the scene will stop working. Just close the plot window and the explore mode window. Then the explore mode window will re-display and will start accepting keyboard input again.

Generating a Disparity Map

After running explore mode, move to a desired viewpoint, and the use the M key which will display a disparity map of that view.

Note: After the disparity map is displayed, keyboard input to move around the scene will stop working. Just close the map window and the explore mode window. Then the explore mode window will re-display and will start accepting keyboard input again.

Creating a Fixed-Viewpoint Specular Reflection Video

Replace <pose_index> with the index position of the camera pose to view the specular reflection from. If unsure, use 0 to view from the start of the tunnel.

```
python run_nerf.py --config configs/<config_filename>
    --fixed_pose_index <pose_index>
```

Note: The video will be stored as an MP4 file in the config folder inside the log folder, "logs/<config_filename>".