RBE/CS 549 Project 2
Structure from Motion and NeRF

Blake Bruell
Worcester Polytechnic Institute
babruell@wpi.edu

Cole Parks
Worcester Polytechnic Institute
cparks@wpi.edu

***Abstract*—This report presents the results of implementing a Structure from Motion (SfM) pipeline and a Neural Radiance Field (NeRF) model. The SfM pipeline estimates the camera poses and 3D structure of a scene from a set of 2D images by creating point clouds, which recreate the scene in 3D.**

I. Phase 1: Structure from Motion

A. Introduction

Structure from motion is a computer vision technique that aims to recover the 3D structure of a scene from a set of 2D images. The basic idea is to estimate the camera poses and 3D points that best explain the observed 2D points in the images. This is a challenging problem, as it requires solving for the camera poses and 3D points simultaneously, and is sensitive to noise and outliers. In this phase, we implemented a basic structure from motion pipeline that estimates the camera poses and 3D points from a set of 2D images. We begin this section with the math and algorithms used in a basic SfM pipeline for a single pair of images, and then extend it to multiple images. We then discuss the challenges and limitations of the basic pipeline, and propose a more robust and scalable pipeline that addresses these issues. Finally, the results of the pipeline applied to Unity Hall at WPI are presented.

B. Estimating the Fundamental Matrix

The first step in the Structure from Motion (SfM) pipeline is to estimate the fundamental matrix. The fundamental matrix describes the epipolar geometry between two images, relating the points in one image to the corresponding epipolar lines in the other image. This relationship is captured in the following mathematicaly expression, refered to as the epipolar constraint:

$$\mathbf{x}_i'^{\top}\mathbf{F}\mathbf{x}_i = 0 \tag{1}$$

where $\mathbf{x}_i$ and $\mathbf{x}_i'$ are the homogeneous coordinates of the corresponding points in the two images, and $\mathbf{F}$ is the fundamental matrix. This equation is then expanded to the following form:

$$\begin{bmatrix} x_i' & y_i' & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = 0 \tag{2}$$

which can be transformed into a the following linear equation:

$$\begin{bmatrix} x_1'x_1 & x_1'y_1 & x_1' & y_1'x_1 & y_1'y_1 & y_1' & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n'x_n & x_n'y_n & x_n' & y_n'x_n & y_n'y_n & y_n' & x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix} = 0 \tag{3}$$

Note that we need *at least 8* point correspondences to solve for Equation 3, as each correspondence only contributes 1 contraint to the system, as the epipolar constraint is a scalar equation.

To solve Equation 3 we use singular value decomposition (SVD) to find the least squares solution to the linear equation. The fundamental matrix is then constructed from the least squares solution, and the rank-2 constraint is enforced by setting the smallest singular value to zero.

To increase the stability of the solution, the 8 points from each image are normalized using basic normalization matrices, $\mathbf{T}$ and $\mathbf{T}'$. Once the eight-point algorithm is applied to the normalized points, the fundamental matrix is denormalized using the following equation:

$$\mathbf{F}' = \mathbf{T}'^{\top}\mathbf{F}\mathbf{T} \tag{4}$$

It is important to note, however, that the eight-point algorithm is sensitive to noise and requires a sufficient number of point correspondences for accurate results. Additionally, it can be affected by degenerate configurations, such as coplanar points or degenerate camera motion. To solve these solutions one more step is required.

The resulting fundamental matrix can be visualized by looking at the epipolar lines in the two images. Epipolar lines are lines which pass though a feature point one image and location of the other camera in that image. The epipolar lines are shown in Figure 1.

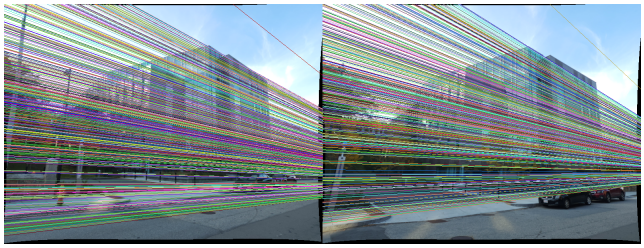


Fig. 1: Epipolar lines in the two images

C. Match Outlier Rejection with RANSAC

To address the aforementioned issues with the naive eight-point algorithm, we use the RANdom SAmple Consensus (RANSAC) algorithm to reject outlier correspondences, and thus obtain a more accurate estimate of the fundamental matrix. RANSAC is an iterative algorithm that selects a random subset of the data and fits a model to that subset, in our case a random set of 8 correspondences. It then evaluates the model on the remaining data, and the points that are consistent with the model are considered inliers. This process is repeated for a specified number of iterations, and the model with the most inliers is chosen as the best estimate

One aspect of the algorithm which was glossed over is the evaluation of the model on the remaining data. The naive approach would be to simply consider the algebraic error of Equation 1, and apply a threshold. This, while functinoal, does not reflect the physical reality of what we are trying to model, which is the geometric error. The better option, and what was used, is Sampson distance, which is a first order approximation to geometric error. The Sampson distance is given by the following equation:

$$d(\mathbf{x}_i, \mathbf{x}_i')^2 = \frac{(\mathbf{x}_i'^\top \mathbf{F} \mathbf{x}_i)^2}{\|\mathbf{F}\mathbf{x}_i\|_2^2 + \|\mathbf{F}^\top \mathbf{x}_i'\|_2^2} \tag{5}$$

The output of the fundamental mattix RANSAC with Sampson distance is shown in Figure 2. It can clearly be seen that RANSAC has effectively removed the outlier correspondences, resulting in a more accurate set of matches.

D. Estimating the Essential Matrix from the Fundamental Matrix

The next step of the SfM pipeline is to estimate the essential matrix from the fundamental matrix. The essential matrix is a 3×3 matrix that relates the corresponding points in two images, assuming that the cameras obey the pinhole model.

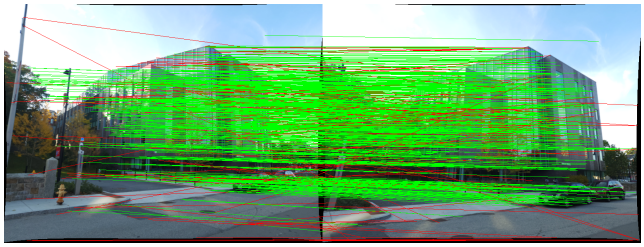


Fig. 2: Matched features with discarded matches shown in red

It can be computed from the fundamental matrix using the following relationship:

$$\mathbf{E} = \mathbf{K}^\top \mathbf{F} \mathbf{K} \tag{6}$$

Due to noise in the calculation of the fundamental matrix, the essential matrix is not guaranteed to be of rank 2. To enforce this constraint, we use singular value decomposition to decompose the essential matrix into its constituent parts, and then reconstruct it using the following equation:

$$\mathbf{E} = \mathbf{U} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{V}^\top \tag{7}$$

E. Estimating Camera Pose from the Essential Matrix

Once the essential matrix has been estimated, the next step is to estimate the camera pose. The essential matrix is a representation of the relative pose between the two cameras, and it can be decomposed into the rotation and translation components. The decomposition of the essential matrix is given by the following equation:

$$\mathbf{E} = [\mathbf{t}]_\times \mathbf{R} \tag{8}$$

Sadly, the decomposition of the essential matrix is not unique, and can result in four possible camera poses. To resolve this ambiguity, we use the following method:

1) Compute the four possible camera poses using the essential matrix
2) Triangulate the 3D points for each of the four camera poses
3) Enforce the cheirality condition

Step 1 is accomplished by decomposing $\mathbf{E}$ using SVD into $\mathbf{U}$, $\mathbf{D}$, and $\mathbf{V}$. The four possible camera poses are then computed using the following equations:

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

1) $\mathbf{R}_1 = \mathbf{U}\mathbf{W}\mathbf{V}^\top$ and $\mathbf{C}_1 = \mathbf{U}(:, 3)$
2) $\mathbf{R}_2 = \mathbf{U}\mathbf{W}\mathbf{V}^\top$ and $\mathbf{C}_2 = -\mathbf{U}(:, 3)$
3) $\mathbf{R}_3 = \mathbf{U}\mathbf{W}^\top\mathbf{V}^\top$ and $\mathbf{C}_3 = \mathbf{U}(:, 3)$
4) $\mathbf{R}_4 = \mathbf{U}\mathbf{W}^\top\mathbf{V}^\top$ and $\mathbf{C}_4 = -\mathbf{U}(:, 3)$

where $(:, 3)$ denotes the third column of the matrix, $\mathbf{R}_i$ is a rotation matrix, and $\mathbf{C}_i$ is the camera center. The triangulation process for the second step will be discussed in the next section. The output of this process is shown in Figure 3.

The cheirality condition is simply the condition that the 3D points are in front of the camera. This is enforced by checking the sign of the depth of the 3D points, and discarding the camera poses that do not satisfy the condition. The cheirality condition is given by the following equation:

$$\mathbf{R}(:, 3)^\top (\mathbf{X} - \mathbf{C}) > 0 \tag{9}$$

Due to noise, though, Equation 9 is not guaranteed to be satisfied for all 3D points in the correct camera. To resolve this issue, we simply accept the camera pose that satisfies the

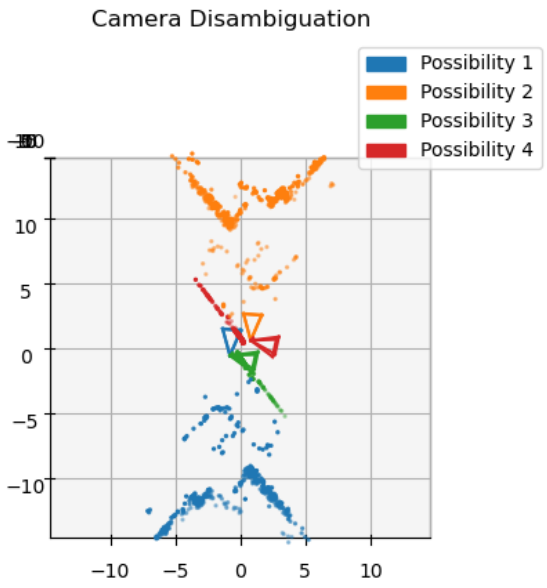


Fig. 3: Initial triangulation plot with disambiguity, showing all four possible camera poses

cheirality condition for the maximum number of 3D points. The threshold for the cheirality condition was also changed to > 0.1, to account for noise.

F. Linear Triangulation

With two camera poses ($\mathbf{R}_i$, $\mathbf{C}_i$) and the corresponding 2D points in the images ($\mathbf{x}_i$, $\mathbf{x}_i'$) we can calulate X, or the world point of each correspondence. To find a solution to the problem of triangulatoin, we begin with the pinhole projection model:

$$\begin{bmatrix}\mathbf{x}\\1\end{bmatrix} = \alpha\mathbf{P}\begin{bmatrix}\mathbf{X}\\1\end{bmatrix} \tag{10}$$

where $\mathbf{P}$ is the matrix:

$$\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}] \tag{11}$$

which can alternatively be written as:

$$\left[\begin{bmatrix}\mathbf{x}\\1\end{bmatrix}_\times \mathbf{P}\right]\begin{bmatrix}\mathbf{X}\\1\end{bmatrix} = 0 \tag{12}$$

We stack Equation 12 for each camera pose and its corresponding image point, and then solve for $\mathbf{X}$ using singular value decomposition. The result of this process is shown in Figure 5.

G. Non-Linear Triangulation

Given the linearly estimated 3D world points from the previous step, we refined their locations to minimize the reprojection error. The linear triangulation method minimizes the algebraic error, but the reprojection error is a more geometrically meaningful error that can be computed by measuring the geometrix error between an image points and the world point projected into its image plane. Since there are likely

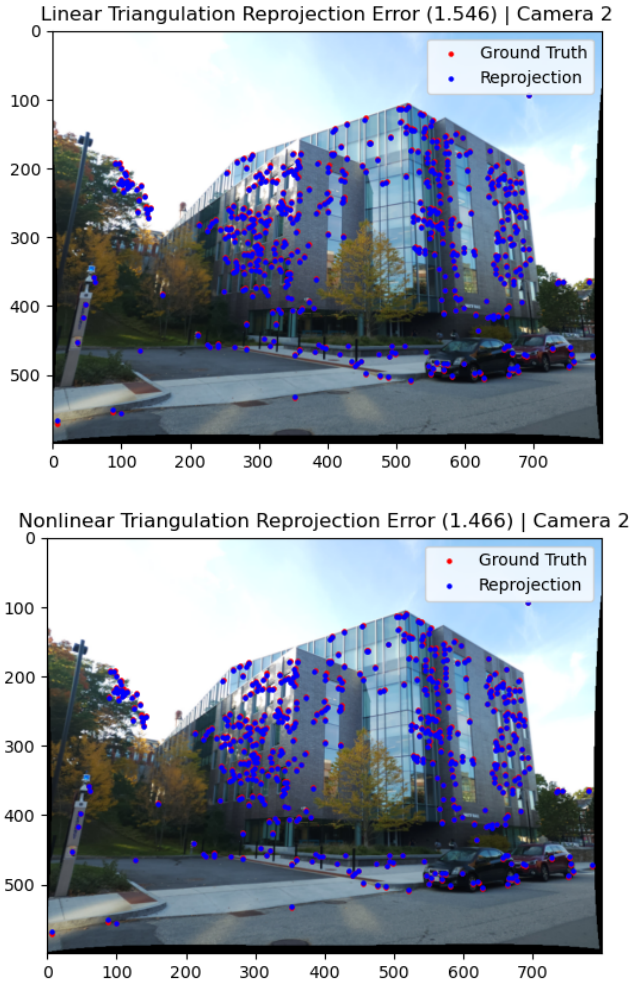


Fig. 4: Comparison of projections between non-linear and linear triangulation for Image 1

nonlinearities in the camera model, this is a more accurate method for estimating the 3D points. A comparison of the projections between the non-linear and linear triangulation methods is shown in Figure 4.

This method *did* improve the accuracy of the 3D points, as shown in results section of this phase, in Table I, but the optimization step was likely not needed, as only in some cases was the error significantly reduced, and even in those cases the error to begin with was not that high. This reflects the fact that the linear triangulation method is already quite accurate.

H. SfM Pipeline for a Single Pair of Images

With all the building blocks in place, we can now summarize the SfM pipeline for a single pair of images:

1) Use RANSAC to reject outlier correspondences and find the fundamental matrix
2) Estimate the essential matrix from the fundamental matrix

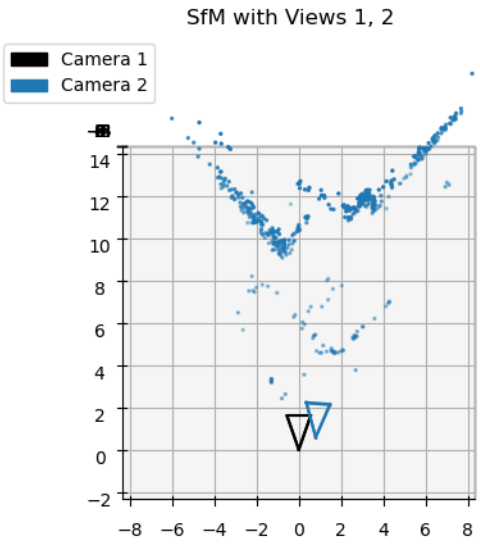


Fig. 5: Final reconstruction of the scene using the SfM pipeline for a single pair of images

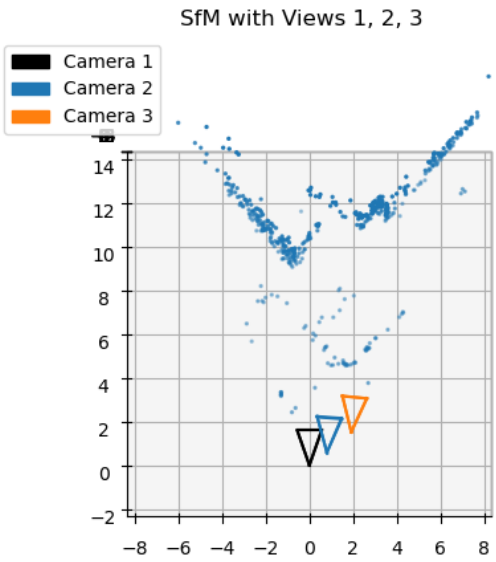


Fig. 6: Camera 3's estimated pose using PnP RANSAC

3) Estimate the 4 possible camera poses from the essential matrix
4) Linearly triangulate the 3D points for each possible pose
5) Disambiguate the camera poses using the cheirality condition
6) Non-linearly triangulate the 3D points to minimize the reprojection error

This pipline works very well, and an output of the pipeline is shown in Figure 5.

I. Perspective-N-Points (PnP) and PnP RANSAC

To extend this pipeline to multiple images, we need to estimate the camera pose for each image. This is accomplished using the Perspective-n-Points (PnP) algorithm, which estimates a camera pose from a set of 3D points and their corresponding 2D projections. The PnP algorithm can take in n, hence the name, but we used $n = 6$ which allows the math to remain simple.

PnP first consists of forming a linear estimate of the camera pose using, formed by transforming Equation 10 into the following form:

$$\begin{bmatrix} X, Y, Z, 1, 0, 0, 0, 0, -xX, -xY, -xZ, -x \\ 0, 0, 0, 0, X, Y, Z, 1, -yX, -yY, -yZ, -y \end{bmatrix} \begin{bmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{14} \\ p_{21} \\ p_{22} \\ p_{23} \\ p_{24} \\ p_{31} \\ p_{32} \\ p_{33} \\ p_{34} \end{bmatrix} = \mathbf{0} \tag{13}$$

and then stacking it n times for each point used in the linear PnP. This is then solved using SVD, yielding the projection matrix $\mathbf{P}$. $\mathbf{P}$ is then decomposed into $\mathbf{R}$, and $\mathbf{t}$:

$$\begin{aligned} \mathbf{R} &= \mathbf{K}^{-1}\mathbf{P}(:, 1:3) \\ \mathbf{t} &= \mathbf{K}^{-1}\mathbf{P}(:, 4) \end{aligned} \tag{14}$$

which is then cleaned up:

$$\begin{aligned} \mathbf{U}, \mathbf{D}, \mathbf{V} &= \mathrm{SVD}(\mathbf{R}) \\ \mathbf{R} &= \mathbf{U}\mathbf{V}^{\top} \\ \mathbf{t} &= \mathbf{t}/\mathbf{D}_{11} \end{aligned} \tag{15}$$

Finally, if $\det(\mathbf{R}) = -1$, we flip the sign of of $\mathbf{R}$ and $\mathbf{t}$.

This process yields a decent estimate of the camera pose, but is very sensistive to noise and outliers. To address this the RANSAC algorithm is applied, with 6 random points chosen each iteration, and with inliers being determined using reprojection error. The result of this process is shown in Figure 6.

Fig. 7: Comparison of reprojected points between non-linear and linear PnP for Camera 3

J. NonLinear PnP

While the linear PnP RANSAC algorithm is a good start, it is not perfect. To address this, we use the Levenberg-Marquardt algorithm to refine the camera pose estimate. The Levenberg-Marquardt algorithm is a non-linear optimization algorithm that minimized a sum of squares of residuals. The residuals for the algorithm are simply given by the difference between the reprojected points and the true image points:

$$\mathbf{r} = \mathbf{x} - \mathbf{PX} \tag{16}$$

which means that Levenberg-Marquardt is minizing the following cost function:

$$\mathbf{C} = \sum_{i=1}^{n} \|\mathbf{x}_i - \mathbf{PX}_i\|_2^2 \tag{17}$$

The result of this process is shown in Figure 7.

K. SfM Pipeline for Multiple Images

With PnP added to our toolbox, we can now summarize the basic SfM pipeline for multiple images:

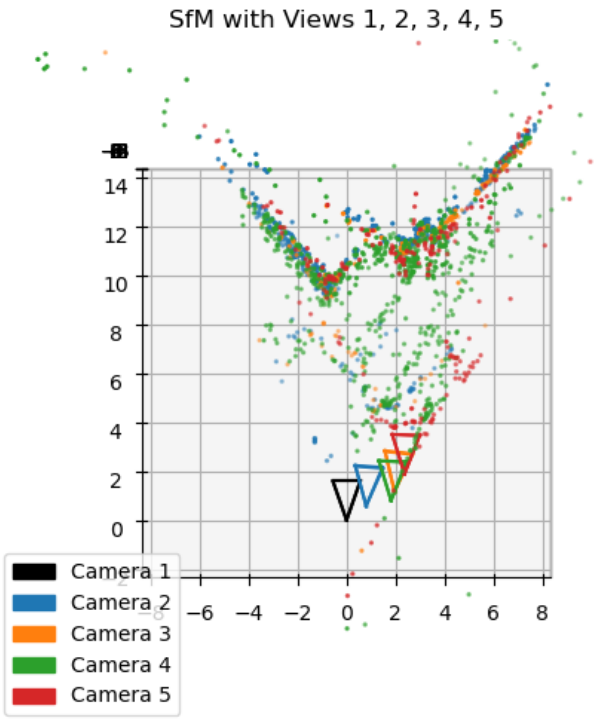


Fig. 8: Output from 5 differnt view of Unity Hall, without bundle adjustment

1) Perform the SfM pipline for a single pair of images on the first two images
2) For each subsequent image:
   a) Determine all image points which have a known 3D world-point
   b) Estimate the camera pose using PnP RANSAC
   c) Refine the camera pose using non-linear PnP
   d) Triangulate all points which have a match to a previos image, and do not already have a known 3D world-point
   e) Add the new 3D points to the set of known world points

The output of this pipeline is shown in Figure 8.

This pipeline may seem fairly straight forward, but looking closer at the steps for subsequent images reveals a major challenge: determining which image points do and do not have a known 3D world-point. This is a non-trivial problem, and is the focus of the next section.

L. World Point Set Data Structure

To be able to efficiently determine which image points have a known 3D world-point, we need to store the world points in a data structure that keeps track of which image points are associated with each world point. This is accomplished using a pair of structures:

1) A list of world points and the associated image points in each image

2) A dictionary the following structure:
   a) Keys: image ids (denoted as `id1`)
   b) Value: Dictionary
      i) Keys: all image points (`ip1`) in image `id1`
      ii) Value: Dictionary
         A) Color: color of `ip1`
         B) Has_WP: boolean indicating if `ip1` has a known 3D world-point
         C) Key: image id (`id2`) of each image with a match to point `ip1`
         D) Value: matched point (`ip2`) in image `id2`

These data structures allow one to efficiently recover the all matches for a given point, identidied by a `(id1, ip1)` pair, and to efficiently recover all matches for a given image, identified by `id1`. The correspoding world point list simply keeps track of every world point and its color, as well as the image points in each image that are associated with it. This means the world point list encode the *tracks* in the set of images.

A *track* is a set of image points that are all associated with the same world point. The world point list and the dictionary are then used to efficiently determine which image points for a particular have a known 3D world-point, and to efficiently determine which points matched with in a point in a particular image do not have a known 3D world-point, crucial for the SfM pipeline for multiple images.

M. Bundle Adjustment

The final aspect of the SfM pipeline is bundle adjustment, which refines the camera poses and 3D points simultaneously by minimizing the reprojection error of all the image points which have a known world point. The optimization problem can be expressed as follows:

$$\min_{\{\mathbf{C}_i,\mathbf{q}_i\}_{i=1}^{I},\{\mathbf{X}_j\}_{j=1}^{J}} \sum_{i=1}^{I}\sum_{j=1}^{J} \mathbf{V}_{ij}\left(\|\mathbf{x}_j - \mathbf{P}_i\mathbf{X}_j\|_2^2\right) \tag{18}$$

where $\mathbf{C}_i$ and $\mathbf{q}_i$ are the camera center and rotation quaternion (used for more stable convergence) for the ith camera, $\mathbf{X}_j$ is the jth world point, and $\mathbf{P}_i$ is the projection matrix for the ith camera, and $\mathbf{V}_{ij}$ is a binary matrix that indicates if the jth world point is visible in the ith image.

This is a slow optimization problem, as it involves a large number of parameters, but is made feasible by the using a sparsity matrix for the Jacobian, which is derived from $\mathbf{V}$. The Trust Region Reflective algorithm is used to solve the optimization problem.

This formulates the last step of our final pipeline, which is as follows:

1) Perform the SfM pipline for a single pair of images on the first two images
2) For each subsequent image:
   a) Determine all image points which have a known 3D world-point
   b) Estimate the camera pose using PnP RANSAC
   c) Refine the camera pose using non-linear PnP
   d) Triangulate all points which have a match to a previos image, and do not already have a known 3D world-point
   e) Add the new 3D points to the set of known world points
   f) Perform bundle adjustment for all images/cameras and world points

Now the part you have been waiting for, the results. The SfM pipeline described above was applied to 5 images taken of Unity Hall at WPI. The results of the pipeline with bundle adjustment is shown in Figure 9, and with final result with color is shown in Figure 10.

The triangulation reprojection error for each image pair is before and after nonlinear optimization is shwon in Table I, the PnP reprojection error for each added view is shown in Table II, and the bundle adjustment reprojection error each set of views for which it was performed is shown in Table III.

View 1	View 2	Number of Points	Error (Linear)	Error (Nonlinear)
1	2	508	1.507424	1.503899
1	3	55	1.996084	1.934641
2	3	184	0.898506	0.897582
1	4	34	1.375734	1.371658
2	4	102	1.216846	1.216223
3	4	629	0.741905	0.587245
1	5	7	1.694281	1.672964
2	5	18	1.669900	1.374472
3	5	104	1.932718	1.340792
4	5	92	1.675483	1.603067

TABLE I: Triangulation reprojection error for each pair of views, before and after nonlinear optimization.

New View	Number Points	Error (Linear)	Error (Nonlinear)
3	191	25.663625	5.625782
4	390	10.415050	3.000878
5	496	18.373742	7.761963

TABLE II: PnP reprojection error for each added view, before and after nonlinear optimization.

Views	Error (Before)	Error (After)
1, 2, 3	1.866829	1.042487
1, 2, 3, 4	1.119583	0.943924
1, 2, 3, 4, 5	1.751347	1.015014

TABLE III: Bundle adjustment reprojection error for all views.

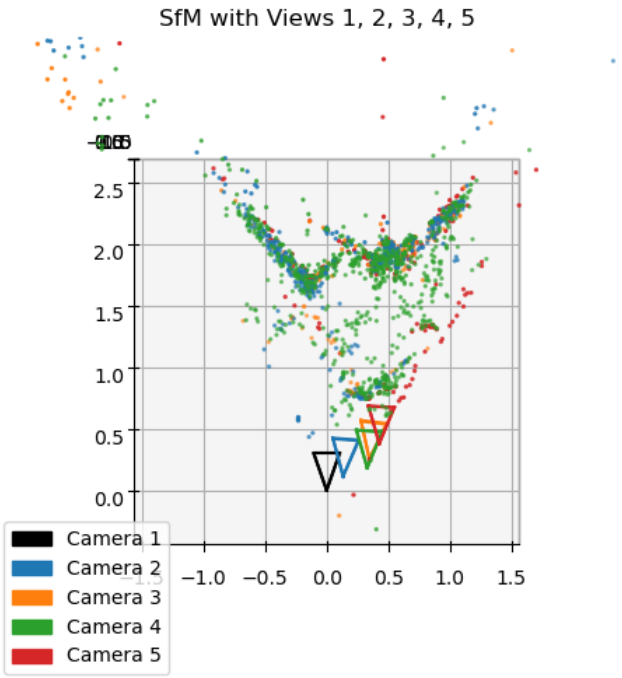


Fig. 9: Final reconstruction of the scene using the SfM pipeline with bundle adjustment for 5 images of Unity Hall

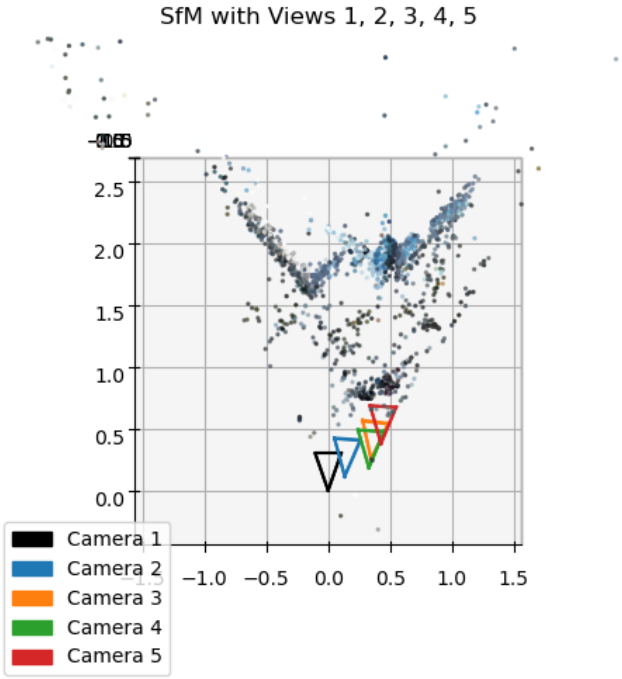


Fig. 10: Final reconstruction of the scene using the SfM pipeline for multiple images

II. Phase 2: Neural Radiance Fields (NeRF)

In Phase 2, we implemented the well-known NeRF model, which is a method for synthesizing novel views of complex scenes. NeRF is a fully-connected deep neural network that takes 5D coordinates encoding position and view direction, and outputs the radiance emitted at the position in the view direction. The model was trained on a set of images of a scene and the associated camera extrincs for each image, and can was used to render novel views of the scene from any viewpoint. We implemented the NeRF model as described in the original NeRF paper [1].

A. Data Loading

Before being able to train the network, we needed to be able to produce sample rays with a known color. This was accomplished by simply creating one ray for each pixel in an image. The direction of the ray was determined by assuming the principal point was in the center of the image, and using the known focal length of the camera in pixels. The origin of the rays for a particular image were shifted to the origin of the camera, and the ray directions were rotated by the camera rotation the camera of the frame. For a dataset of N images, each with width W and height H, the result would be a full dataset of $N \times W \times H$ rays. During training a random subset of `batch_size` (a hyperparameter) rays were selected from the entire set. For predictions, rays were generated from a given camera position, image size, and focal length, using the same process as used for input data.

NDC Ray Space: For the forward facing scenes which we were dealing with, rays can be embedded into NDC ray space. This technique was only applied to the ship dataset, as when applied to the lego dataset, the networks trained to around 21 PSNR, and then did not train anymore regardless of doing more iterations. For the ship dataset, the effect of the transform was to move the sampled points into a wider range of values, thus allowing the network to learn more effectively. The details for how this technique were implemented are explained in the original nerf paper.

B. Point Sampling for NeRF

The nerf network takes in points and directions, not the rays in dataset, and so sampling along the rays is performed. At the most basic level, the region between the distances of `near` and `far` along each ray are split into `N_coarse` bins, and a random point from bin is selected. This is performed for each ray in the training batch, and then all points an rgb and σ value are predicted for each point. These are then used in a radiance rendering pipeline to calculate the predicted color for the ray, on which loss can then be calculated. In practice this means a set of rays of shape $B \times (3+3)$ (a 3 vector for position and 3 vector for direction) is expanded to a set of points and directions of shape $B \times N\_coarse \times (3+3)$, passed into the network as $B * N\_coarse \times (3+3)$, and then reshaped back to $B \times (N\_coarse) \times (3+3)$, and finally using the radiance volume rendering technique collapsed into just

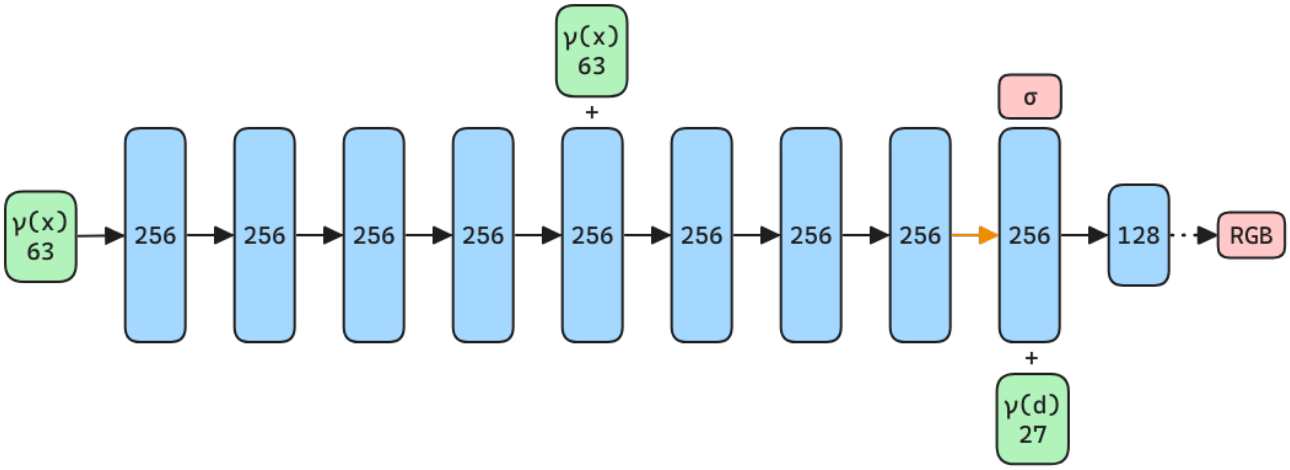


Fig. 11: NeRF Network Architecture

$B \times 3$, which are the colors of the rays. This pipeline is used while training and rendering images.

C. Positional Encoding

In order for the network to learn fine detail in the model, the points passed into the network must be encoded into a higher dimension, which is done via positional encoding. This step is performed by the network itself, and not as a preprocessing step. Each ray is expanded from a 3 vector position $\mathbf{x}$ and 3 vector direction $\mathbf{d}$ via the following transform:

$$\begin{aligned}\gamma(x) = (&\sin(2^0\pi x), \cos(2^0\pi x),\\ &\ldots,\\ &\sin(2^{L-1}\pi x), \cos(2^{L-1}\pi x))\end{aligned} \tag{19}$$

which is applied to each coordinate value in $\mathbf{x}$ and each component of $\mathbf{d}$. This gives rise to two hyperparameters, namely `pos_enc_x` and `pos_enc_d`, which give the value of L for $\mathbf{x}$ and $\mathbf{d}$ respectively.

D. TinyNeRF

To initially test and validate the implementation implementation our data loading, we built a simpler implementation of NeRF, called TinyNeRF implementation, which is a single four-layer fully-connected network. It is designed to be easy to train on, and to allow for quick iteration and debugging which helped us ensure that the our model was able to learn to render novel views of a scene from a set of images.

1) Training: The network architecture is shown in Figure 12. This network was trained with 100 images of size 100×100 , with hyperparameters `batch_size = 1024`, `pos_enc_x = 6` and `pos_enc_x = 4`. The optimizer used was `Adam`, with default parameters, and a learning rate of 5×10^{-3}

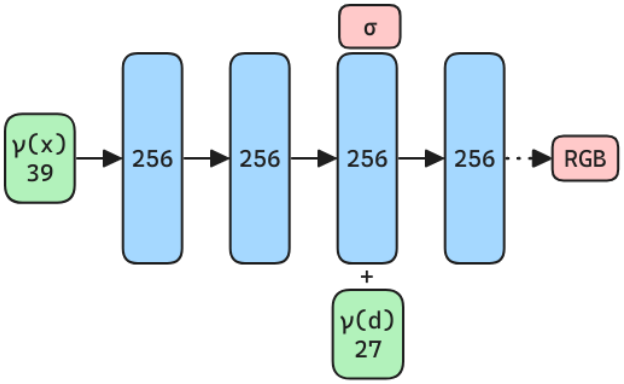


Fig. 12: TinyNeRF Network Architecture

E. NeRF Model

The main NeRF model is a set of two fully-connected networks, one for finding the coarse colors for each point, and one for finding the fine colors, which uses the coarse network and estimates the final colors for the output image. The network architecture is shown in Figure 11.

1) Hierarchical Sampling: The input points for the coarse network are exactly as discussed in previously, but the fine network is a little different. Based on the points with the highest density predicted by the coarse network, an extra `N_fine` points are sampled along each ray, using inverse transform sampling to focus the extra samples around the parts of the ray which intersect the object being learned (as predicted by the coarse model). This means that the fine network is trained on $N\_coarse + N\_fine$ points for each ray, as the samples locations initially used for the coarse network are added to the points sampled using the hierarchical sampling method. The specifics of this technique are described in the original NeRF paper. Both networks used the same hyperparameters.

2) Training: This network was trained with 800×800 images, with hyperparameter `batch_size = 4096`,

TABLE IV: NeRF Metrics

PSNR↑

	Lego	Lego (No Positional)	Ship
Original	32.54	27.75	28.65
Ours	28.944	25.951	20.3

SSIM↑

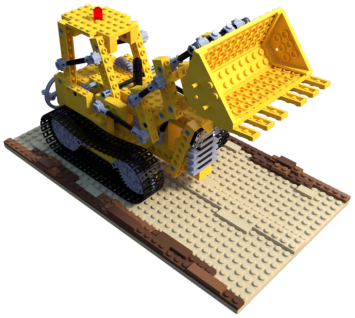
	Lego	Lego (No Positional)	Ship
Original	32.54	27.75	28.65
Ours	0.9629	0.9295	XX

`N_coarse = 64`, `N_fine = 128`, `pos_enc_x = 10`, `pos_enc_x = 4`. The optimizer used was `Adam`, with default parameters, and a learning rate of 5×10^{-4} which decayed exponentially to 5×10^{-5} over 250000 iterations.

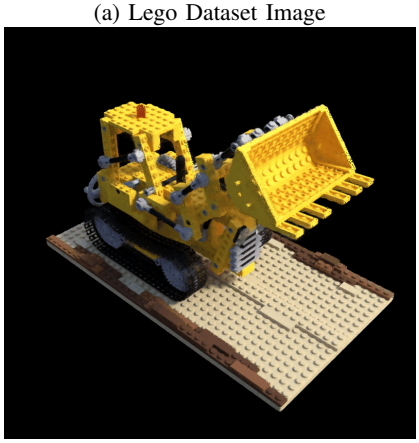
F. Results and Discussion

When rendering the Lego model, the reconstructed images were very sharp, with low amounts of noise (albeit higher than the original NeRF paper implementation). Reflections and complex lighting conditions were captured well, as were fine details. The accuracy of the output depended on multiple factors, and one such factor was the use of positional encoding. When using positional encoding for the Lego dataset, the model was able to capture the fine details of the structure, such as the individual studs on the top of the Lego bricks. However, when positional encoding was removed, the model was not able to capture the fine details of the model, and the output was much more smoothed, as if there was motion blur. The final PSNR and SSIM results are reported in Table IV.

When training on the ship dataset, which had even more complex lighting scenarios with opaque water and very thin rigging, the model had a much more difficult time. Our outputs clearly show a ship, but only the larger sails and the general shape of the ship are captured. The water is not captured well at all. This was trained with the same parameters as the Lego dataset, and the only difference was the dataset itself, so it is clear that the complexity of the dataset has a large impact on the model's ability to capture the scene.



(a) Lego Dataset Image



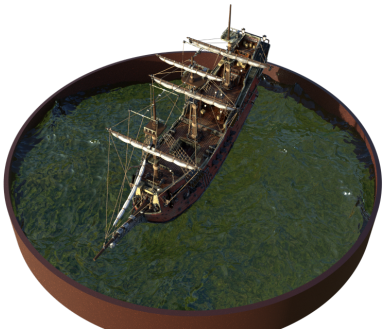
(b) Lego Output with Positional Encoding

(c) Lego Output without Positional Encoding

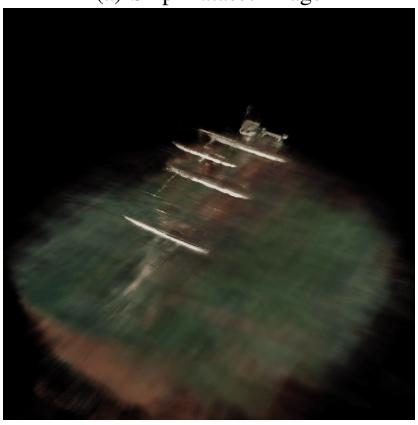
Fig. 13: Lego NeRF Outputs

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*, 2020. arXiv: 2003.08934 [cs.CV].



(a) Ship Dataset Image



(b) Ship NeRF Output

Fig. 14: Ship NeRF Outputs

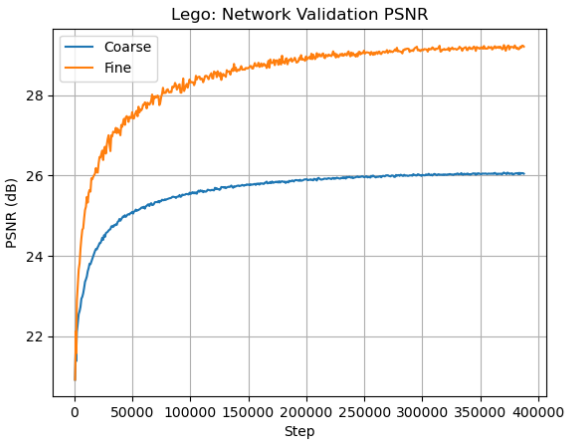


Fig. 15: Validation Loss During Training on Lego Dataset

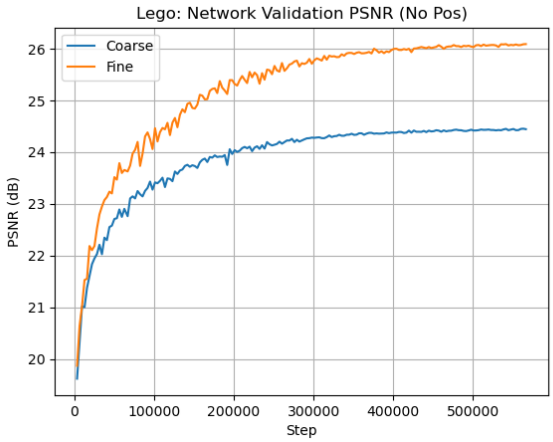


Fig. 16: Validation Loss During Training on Lego Dataset Without Positional Encoding

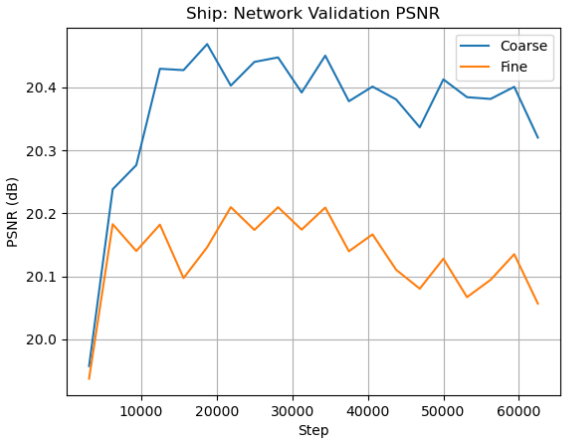


Fig. 17: Validation Loss During Training on Ship Dataset