Novel view synthesis with neural radiance fields

Maryush Soroka [1] **Anton Labutin** [1] **Vladislav Mityukov** [1] **Yunseok Park** [1] **Marco Offidani** [1]

Abstract

In this project, we aimed to replicate and extend the Neural Radiance Field (NeRF) algorithm for novel view synthesis, which has recently achieved impressive photorealistic image synthesis results. We studied the methodology of the technology, trained and evaluated NeRF on synthetic and real datasets, and compared the performance of NeRF trained with known and estimated camera-to-world matrices and intrinsic parameters. Additionally, we explored a method based on hash ideas like Instant-NGP, which aim to improve the training and inference time of NeRF.

Github repo: NeRF this
Presentation file: Presentation Team15.zip

1. Introduction

The Neural Radiance Field (NeRF) algorithm, introduced in (*Mildenhallet al.*, 2020) , has revolutionized the field of computer graphics and computer vision by enabling the synthesis of photorealistic novel views from 3D scenes. This is achieved by modeling the underlying scene geometry and appearance using a continuous, differentiable function learned by a multi-layer perceptron (MLP). Since its introduction, NeRF has generated significant interest in the research community, with numerous extensions and variations proposed to improve its efficiency, accuracy, and applicability.

One of the main challenges of NeRF is its computational cost, with training and rendering times being prohibitively high for many practical applications. Recent research has proposed various methods to address this challenge, including the use of volume grids or hash-based approaches like Instant-Nearest-Grid-Point (Instant-NGP). These methods aim to reduce the number of MLP evaluations required for rendering an image, thus improving the efficiency of NeRF.

[1]Skolkovo Institute of Science and Technology, Moscow, Russia. Correspondence to: Marco Offidani <marco.offidani@skoltech.ru>.

Final Projects of the Machine Learning 2023 Course, Skoltech, Moscow, Russian Federation, 2023.

The motivation for this project is to replicate and extend the NeRF algorithm, and evaluate its performance on both synthetic and real-world datasets. We aim to investigate the accuracy and efficiency of NeRF under different camera-to-world matrices and intrinsic parameters, and compare the performance of NeRF with and without estimated parameters. Furthermore, we propose to explore the potential of hash-based methods like Instant-NGP to improve the efficiency of NeRF.

Recent related work has shown promising results for hash-based methods like Instant-NGP, which achieves significant speedups in rendering times while maintaining the photorealistic quality of the synthesized images. Our project builds upon these developments, providing a comprehensive evaluation of the effectiveness of Instant-NGP compared to the vanilla NeRF algorithm.

The main contributions of this report are as follows:

1. Deriving and explaining the formula for casting rays
2. Replicating and comparing the results of NeRF algorithm for the problem of novel view synthesis in two scenarios: synthetic and real
3. Extracting train and test camera-to-world matrices and intrinsic parameters by COLMAP framework and comparing the results with the case when poses are known
4. Exploring the research direction and testing methods based on the hash idea like Instant-NGP and comparing the results with vanilla NeRF

2. Preliminaries

2.1. Forward imaging model

The process of converting a world scene into an image can be described by a mathematical model known as a forward imaging model. This model involves starting from a point in the world coordinate frame and then transforming it into the camera coordinate frame through coordinate transformation. Finally, the camera coordinates are projected onto the image plane using projection transformation. We can visualize the model in Figure 1.

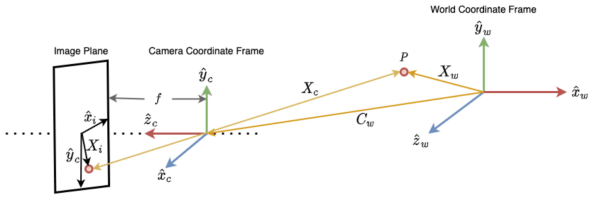

Figure 1. Forward imaging model.

World coordinate frame. The real-world shapes and objects we observe are situated in a three-dimensional (3D) frame of reference known as the world coordinate frame. With the help of this frame, it is relatively simple to determine the position of any point or object in 3D space

Let's take the point P in the $3D$ space as shown in Figure .

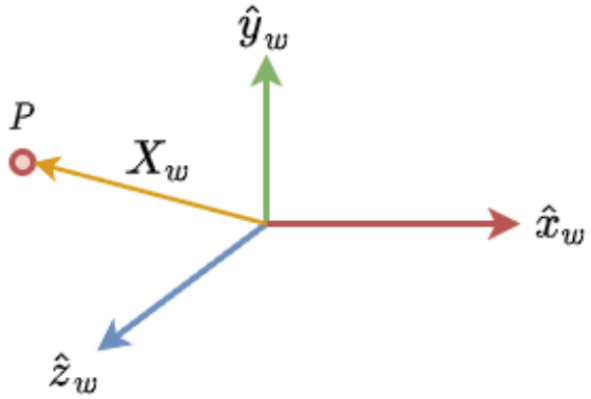

Figure 2. World coordinate frame.

Here, $\hat{x}_w, \hat{y}_w, \hat{z}_w$ represent the three axes in the world coordinate frame. The location of the point P is expressed through the vector $X_w = [x_w, y_w, z_w]^\top$.

Camera coordinate frame. Similarly to the world coordinate frame, there exists another frame of reference known as the camera coordinate frame, as illustrated in Figure 3.

The camera coordinate frame is positioned at the center of the camera, and unlike the world coordinate frame, it is not a fixed frame of reference. As the camera moves to capture an image, this frame can be shifted accordingly.

In the world coordinate frame, the location of a point is identified by the X_w vector, whereas in the camera coordinate frame, it is represented by the $X_c = [x_c, y_c, z_c]^\top$ vector.

Coordinate transformation. Having introduced the world coordinate frame and the camera coordinate frame, we can now establish a mapping between the two.

Consider the point P depicted in Figure 2. Our aim is to

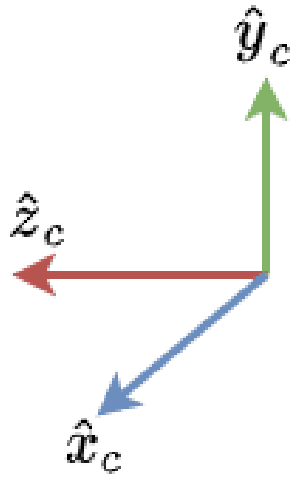

Figure 3. Camera coordinate frame.

establish a connection between the camera coordinates X_c and the world coordinates X_w. From Figure 2, we derive the following relationship:

$$X_c = R \times (X_w - C_w)$$

Here, $R = (r_{i,j})i, j = 1^3$ represents the orientation of the camera coordinate frame relative to the world coordinate frame. The vector $[r1, 1, r_{1,2}, r_{1,3}]$ denotes the direction of x_c in the world coordinate system, while $[r_{2,1}, r_{2,2}, r_{2,3}]$ represents the direction of y_c in the world coordinate frame. Similarly, $[r_{3,1}, r_{3,2}, r_{3,3}]$ indicates the direction of z_c in the world coordinate frame. Additionally, C_w represents the position of the camera coordinate frame relative to the world coordinate frame.

The previous equation can be extended as shown below:

$$X_c = R \times X_w - R \times C_w = R \times X_w + t,$$

where $t = -R \times C_w$ represents the translation matrix.

Although the mapping between the two coordinate systems has been established, it is not fully complete. The equation above involves a combination of matrix multiplication and matrix addition, which can be simplified by condensing it into a single matrix multiplication. To achieve this, we can introduce the concept of homogeneous coordinates.

The homogeneous coordinate system enables us to represent an N dimensional point $x = [x_0, x_1, \ldots, x_n]$ in an $N + 1$ dimensional space $\tilde{x} = [\tilde{x}_0, \tilde{x}_1, \ldots, \tilde{x}_n, w]$ using a hypothetical variable $w \neq 0$ such that:

$$x_0 = \frac{\tilde{x}_0}{w}, x_1 = \frac{\tilde{x}_1}{w}, \ldots, x_n = \frac{\tilde{x}_n}{w}.$$

By introducing the homogeneous coordinate system, we can convert X_w (3D) into $\tilde{X}_w$ (4D) using the following equation:

$$X_w \equiv \begin{bmatrix} x\ y\ z\ 1 \end{bmatrix} \equiv \begin{bmatrix} wx_w\ wy_w\ wz_w\ w \end{bmatrix} \equiv \begin{bmatrix} \tilde{x}\ \tilde{y}\ \tilde{z}\ w \end{bmatrix} \equiv \tilde{X}_w \tag{1}$$

By leveraging homogeneous coordinates, we can simplify the equation to a single matrix multiplication:

$$\tilde{X}_c = \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \times \tilde{X}_w = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

$$\tilde{X}_c = C_{ex} \times \tilde{X}_w,$$

Here, C_{ex} represents the matrix containing the camera coordinate frame's orientation and position. This matrix is referred to as the *Camera Extrinsic* matrix since it includes external properties of the camera, such as rotation and translation.

Projective transformation. Initially, we considered a point P and its corresponding homogeneous world coordinates $\tilde{X}w$. The *Camera Extrinsic* matrix C_{ex} was employed to transform $\tilde{X}_w$ into its homogeneous camera coordinates $\tilde{X}_c$.

Subsequently, we move to the final step of generating an image from the $3D$ camera coordinates $\tilde{X}_c$, as illustrated in Figure 4.

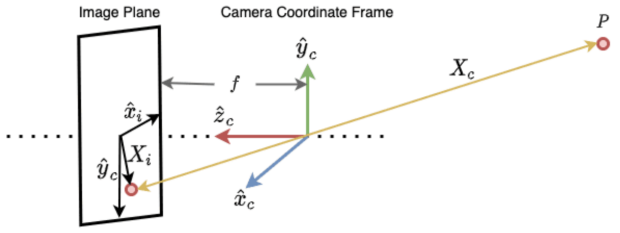

Figure 4. Projecting the point on the image plane.

The key concept to understand projective transformation, is similarity between triangles. From their properties (Figure 5) we can derive that

$$\frac{x_i}{f} = \frac{x_c}{z_c}; \frac{y_i}{f} = \frac{y_c}{z_c}$$

Subsequently it follows:

$$x_i = f\frac{x_c}{z_c},\ y_i = f\frac{y_c}{z_c}$$

It is essential to note that the image plane is not a virtual plane, but instead, it comprises an array of image sensors.

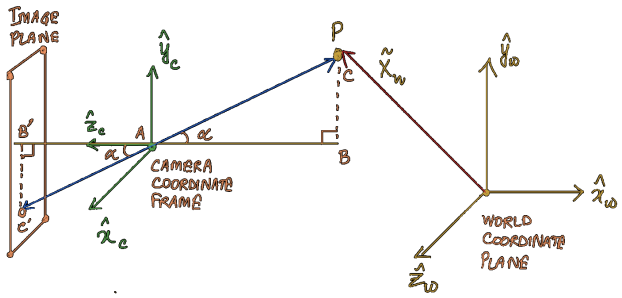

Figure 5. Triangles $AB'C'$ and ABP are similar.

The $3D$ scene is projected onto these sensors, which generates the final image. As a result, the image plane's x_i and y_i coordinates can be replaced with pixel values u and v:

$$u = f\frac{x_c}{z_c},\ v = f\frac{y_c}{z_c}$$

To position a pixel on an image plane, we begin from the upper left-hand corner $(0, 0)$. However, to align the pixels with the center of the image plane, it is necessary to shift them:

$$u = f\frac{x_c}{z_c} + o_x,\ v = f\frac{y_c}{z_c} + o_y$$

In this case, o_x and o_y correspond to the center coordinates of the image plane.

With the point from the $3D$ camera space now represented in terms of u and v on the image plane, we need to convert the pixel values into homogeneous representation to ensure that the matrices are compatible.

Homogeneous representation of u, v,
where $u = \frac{\tilde{u}}{\tilde{w}}$ and $v = \frac{\tilde{v}}{\tilde{w}}$:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} \equiv \begin{bmatrix} z_c u \\ z_c v \\ z_c \end{bmatrix}$$

This can be expanded as:

$$\begin{bmatrix} z_c u \\ z_c v \\ z_c \end{bmatrix} = \begin{bmatrix} fx_c + z_c o_x \\ fy_c + y_c o_y \\ z_c \end{bmatrix} = \begin{bmatrix} f & 0 & o_x & 0 \\ 0 & f & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

So, we have:

$$\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} f & 0 & o_x & 0 \\ 0 & f & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

which can be expressed compactly as

$$\tilde{u} = C_{in} \times \tilde{x}_c$$

In this equation, $\tilde{x}_c$ denotes the vector containing the point's coordinates in the camera coordinate space, while $\hat{u}$ represents the vector containing the point's coordinates on the image plane. Similarly, C_{in} denotes the vector of values required to transform a point from $3D$ camera space to $2D$ space.

$$C_{in} = \begin{bmatrix} f & 0 & o_x & 0 \\ 0 & f & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

We can call C_{in} the camera intrinsic since it represents values like focal length and center of the image plane along x and y axes, both of which are internal properties of the camera.

3. Related work

The problem of novel view synthesis, which aims to synthesize an image of a scene from a viewpoint that has not been captured, has been extensively studied in computer vision and graphics communities. In this section, we review some of the relevant old, recent, and state-of-the-art methods for solving this problem.

One of the classic approaches for view synthesis is based on the geometry of the scene, where the visible portions of the scene from different viewpoints are reconstructed and combined to form a novel view. This approach requires a precise estimation of the scene geometry and camera poses, which can be obtained by Structure from Motion (SfM) and Multi-View Stereo (MVS) (Furukawa et al., 2015) techniques. Although this method can produce high-quality images, it is computationally expensive and sensitive to noise and inaccuracies in the scene geometry estimation.

Another classical approach for view synthesis is based on the image warping technique, where the input images are warped to the target viewpoint and then combined to form the novel view. This approach requires the estimation of a dense correspondences between the input images and the target view, which can be obtained by optical flow or stereo matching algorithms. However, this method can result in artifacts and blurring due to errors in the correspondences.

Recently, deep learning-based methods have shown remarkable success in the field of view synthesis. One of the most notable works in this direction is Neural Radiance Fields (NeRF), which represents a scene as a continuous 3D function and learns this function using a neural network. NeRF has achieved state-of-the-art results in terms of visual quality, but it is computationally expensive and requires a large amount of memory to represent the entire scene.

To overcome the computational limitations of NeRF, several works have proposed extensions and improvements. For example, NeRF++ (Zhang et al., 2020) introduces an octree structure to efficiently represent the scene and accelerate rendering. Another approach is to use a sparse voxel-based representation, such as Sparse Voxel Fields (SVF), which achieves real-time performance on modern GPUs. In addition, some works propose to use hash-based techniques, such as Instant-NGP, to speed up the inference time while maintaining a comparable level of visual quality.

Despite the success of these methods, they still suffer from some limitations, such as high memory and computational requirements, difficulty in handling large-scale scenes, and limited generalization to unseen viewpoints.

4. Algorithms and Models. Experiments and Results.

Github repo: Novel view synthesis with neural radiance fields

4.1. Algorithm

Shortly saying, NeRF pipeline is as follows:

1. Generating rays through each pixel of the image.
2. Sampling the points on the rays.
3. Volume rendering: Passing these points into an MLP to predict the color and density.
4. Calculating photometric loss to perform backpropagation on the MLP.

4.1.1. Neural Radiance Field Scene Representation

Each ray in the $3D$ space is represented by the origin and the direction (vector). So we can represent a continuous scene as a $5D$ vector-valued function whose input is a $3D$ location $\mathbf{x} = (x, y, z)$ and $2D$ viewing direction $\mathbf{d} = (\theta, \phi)$, and whose output is an emitted color $\mathbf{c} = (r, g, b)$ and volume density σ. We approximate this continuous $5D$ scene representation with an MLP network $F_\Theta : (\mathbf{x}, \mathbf{d}) \rightarrow (\mathbf{c}, \sigma)$ and optimize its weights Θ to map from each input $5D$ coordinate to its corresponding volume density and directional emitted color.

It is easy to locate the origin of the pixel points but a little challenging to get the direction of the rays. From 2.1 we have:

$$\tilde{X}_c = C_{ex} \times \tilde{X}_w \Rightarrow \tilde{X}_w = C_{ex}^{-1} \times \tilde{X}_c$$

The camera-to-world matrix from the data set is the C_{ex}^{-1} that we need:

$$C_{ex}^{-1} = \begin{bmatrix} r'_{11} & r'_{12} & r'_{13} & t'_x \\ r'_{21} & r'_{22} & r'_{23} & t'_y \\ r'_{31} & r'_{32} & r'_{33} & t'_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To define the direction vector, we do not need the entire camera-to-world matrix; instead, we use the matrix that defines the camera's orientation:

$$R'_{ex} = \begin{bmatrix} r'_{11} & r'_{12} & r'_{13} \\ r'_{21} & r'_{22} & r'_{23} \\ r'_{31} & r'_{32} & r'_{33} \end{bmatrix}$$

With the rotation matrix, we can get the unit direction vector by the following equation:

$$\vec{d} = \frac{R'_{ex} \times X_c}{|R'_{ex} \times X_c|}$$

The rays' origin will be the translation vector of the camera-to-world matrix:

$$t'_{ex} = \begin{bmatrix} t'_x \\ t'_y \\ t'_z \end{bmatrix}$$

We encourage the representation to be multi-view consistent by restricting the network to predict the volume density σ as a function of only the location x, while allowing the RGB color $\mathbf{c}$ to be predicted as a function of both location and viewing direction. To accomplish this, the MLP F_Θ first processes the input $3D$ coordinate $\mathbf{x}$ with 8 fully-connected layers (using ReLU activation and 256 channels per layer), and outputs σ and a 256-dimensional feature vector. This feature vector is then concatenated with the camera ray's viewing direction and passed to one additional fully-connected layer (using a ReLU activation and 128 channels) that output the view-dependent RGB color.

See Figure 6 for an example of how our method uses the input viewing direction to represent non-Lambertian effects. As shown in Figure 7, a model trained without view dependence (only $\mathbf{x}$ as input) has difficulty representing specularities (*Mildenhallet al.*, 2020).

4.1.2. Volume Rendering with Radiance Fields

Our $5D$ neural radiance field represents a scene as the volume density and directional emitted radiance at any point in space. We render the color of any ray passing through the scene using principles from classical volume rendering. The volume density $\sigma(\mathbf{x})$ can be interpreted as the differential probability of a ray terminating at an infinitesimal

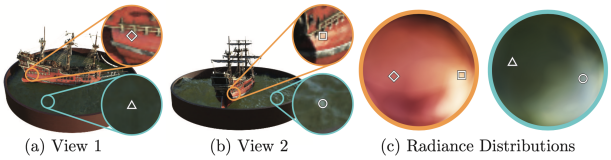

Figure 6. A visualization of view-dependent emitted radiance. Our neural radiance field representation outputs RGB color as a $5D$ function of both spatial position $\mathbf{x}$ and viewing direction $\mathbf{d}$. Here, we visualize example directional color distributions for two spatial locations in our neural representation of the Ship scene. In (a) and (b), we show the appearance of two fixed $3D$ points from two different camera positions: one on the side of the ship (orange insets) and one on the surface of the water (blue insets). Our method predicts the changing specular appearance of these two $3D$ points, and in (c) we show how this behavior generalizes continuously across the whole hemisphere of viewing directions.

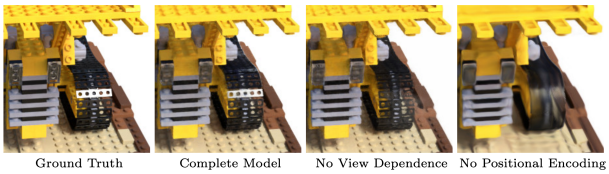

Figure 7. Here we visualize how our full model benefits from representing view-dependent emitted radiance and from passing our input coordinates through a high-frequency positional encoding. Removing view dependence prevents the model from recreating the specular reflection on the bulldozer tread. Removing the positional encoding drastically decreases the model's ability to represent high frequency geometry and texture, resulting in an over-smoothed appearance.

particle at location $\mathbf{x}$. The expected color $C(\mathbf{r})$ of camera ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ with near and far bounds t_n and t_f is:

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt,$$
$$T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right) \quad (2)$$

The function $T(t)$ denotes the accumulated transmittance along the ray from t_n to t, i.e., the probability that the ray travels from t_n to t without hitting any other particle. Rendering a view from our continuous neural radiance field requires estimating this integral $C(\mathbf{r})$ for a camera ray traced through each pixel of the desired virtual camera.

We numerically estimate this continuous integral using quadrature. Deterministic quadrature, which is typically used for rendering discretized voxel grids, would effectively

limit our representation's resolution because the MLP would only be queried at a fixed discrete set of locations. Instead, we use a stratified sampling approach where we partition $[t_n, t_f]$ into N evenly-spaced bins and then draw one sample uniformly at random from within each bin:

$$t_i \sim \mathcal{U}\left[t_n + \frac{i-1}{N}(t_f - t_n), t_n + \frac{i}{N}(t_f - t_n)\right] \quad (3)$$

Although we use a discrete set of samples to estimate the integral, stratified sampling enables us to represent a continuous scene representation because it results in the MLP being evaluated at continuous positions over the course of optimization. We use these samples to estimate $C(\mathbf{r})$ with the quadrature rule discussed in the volume rendering review by Max:

$$\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i\delta_i))\mathbf{c}_i, \quad T_i = \exp\left(-\sum_{j=1}^{i-1}\sigma_j\delta_j\right), \quad (4)$$

where $\delta_i = t_{i+1} - t_i$ is the distance between adjacent samples. This function for calculating $\hat{C}(\mathbf{r})$ from the set of $(\mathbf{c}_i, \sigma_i)$ values is trivially differentiable and reduces to traditional alpha compositing with alpha values $\alpha_i = 1 - \exp(\sigma_i\delta_i)$. (*Mildenhallet al.*, 2020)

4.1.3. Optimizing a Neural Radiance Field

In the previous section we have described the core components necessary for modeling a scene as a neural radiance field and rendering novel views from this representation. However, we observe that these components are not sufficient for achieving state-of-the-art quality. We introduce two improvements to enable representing high-resolution complex scenes. The first is a positional encoding of the input coordinates that assists the MLP in representing high-frequency functions, and the second is a hierarchical sampling procedure that allows us to efficiently sample this high-frequency representation.

Positional encoding. Despite the fact that neural networks are universal function approximators, we found that having the network F_Θ directly operate on $xyz\theta\phi$ input coordinates results in renderings that perform poorly at representing high-frequency variation in color and geometry. This is consistent with recent work by Rahaman et al. (*Rahamanet al.*, 2019), which shows that deep networks are biased towards learning lower frequency functions. They additionally show that mapping the inputs to a higher dimensional space using high frequency functions before passing them to the network enables better fitting of data that contains high frequency variation.

We leverage these findings in the context of neural scene representations, and show that reformulating F_Θ as a composition of two functions $F_\Theta = F'_\Theta \circ \gamma$, one learned and one not, significantly improves performance (see Figure 7). Here γ is a mapping from $\mathbb{R}$ into a higher dimensional space $\mathbb{R}^{2L}$, and F'_Θ is still simply a regular MLP. Formally, the encoding function we use is:

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

This function $\gamma(\dot{p})$ is applied separately to each of the three coordinate values in $\mathbf{x}$ (which are normalized to lie in $[-1, 1]$) and to the three components of the Cartesian viewing direction unit vector $\mathbf{d}$ (which by construction lie in $[-1, 1]$). In our experiments, we set $L = 10$ for $\gamma(\mathbf{x})$ and $L = 4$ for $\gamma(\mathbf{d})$.

A similar mapping is used in the popular Transformer architecture, where it is referred to as a positional encoding. However, Transformers use it for a different goal of providing the discrete positions of tokens in a sequence as input to an architecture that does not contain any notion of order. In contrast, we use these functions to map continuous input coordinates into a higher dimensional space to enable our MLP to more easily approximate a higher frequency function.

Hierarchical volume sampling. Our rendering strategy of densely evaluating the neural radiance field network at N query points along each camera ray is inefficient: free space and occluded regions that do not contribute to the rendered image are still sampled repeatedly. We draw inspiration from early work in volume rendering and propose a hierarchical representation that increases rendering efficiency by allocating samples proportionally to their expected effect on the final rendering.

Instead of just using a single network to represent the scene, we simultaneously optimize two networks: one "coarse" and one "fine". We first sample a set of N_c locations using stratified sampling, and evaluate the "coarse" network at these locations as described in Eq. 3 and 4. Given the output of this "coarse" network, we then produce a more informed sampling of points along each ray where samples are biased towards the relevant parts of the volume. To do this, we first rewrite the alpha composited color from the coarse network $\hat{C}_c(\mathbf{r})$ in Eq. 4 as a weighted sum of all sampled colors c_i along the ray:

$$\hat{C}_c(\mathbf{r}) = \sum_{i=1}^{N_c} w_i c_i, \quad w_i = T_i(1 - \exp(-\sigma_i\delta_i)) \quad (5)$$

Normalizing these weights as $\hat{w}_i = w_i / \sum_{j=1}^{N_c} w_j$ produces a piece wise-constant PDF along the ray. We sample a second set of N_f locations from this distribution using inverse transform sampling, evaluate our "fine" network at the union of the first and second set of samples, and compute the final rendered color of the ray $\hat{C}_f(\mathbf{r})$ using Eq. 5 but using all $N_c + N_f$ samples. This procedure allocates more samples to regions we expect to contain visible content. This addresses a similar goal as importance sampling, but we use the sampled values as a nonuniform discretization of the whole integration domain rather than treating each sample as an independent probabilistic estimate of the entire integral.

Implementation details. We optimize a separate neural continuous volume representation network for each scene. This requires only a dataset of captured RGB images of the scene, the corresponding camera poses and intrinsic parameters, and scene bounds (we use ground truth camera poses, intrinsics, and bounds for synthetic data, and use the COLMAP structure-from-motion package to estimate these parameters for real data). At each optimization iteration, we randomly sample a batch of camera rays from the set of all pixels in the dataset, and then follow the hierarchical sampling described in section 4.1.3 to query N_c samples from the coarse network and $N_c + N_f$ samples from the fine network. We then use the volume rendering procedure described in section 4.1.2 to render the color of each ray from both sets of samples. Our loss is simply the total squared error between the rendered and true pixel colors for both the coarse and fine renderings:

$$\mathcal{L} = \sum_{\mathbf{r} \in \mathcal{R}} \left[\|\hat{C}_c(\mathbf{r}) - C(\mathbf{r})\|_2^2 + \|\hat{C}_f(\mathbf{r}) - C(\mathbf{r})\|_2^2 \right] \quad (6)$$

where $\mathcal{R}$ is the set of rays in each batch, and $C(\mathbf{r}), \hat{C}_c(\mathbf{r})$, and $\hat{C}_f(\mathbf{r})$ are the ground truth, coarse volume predicted, and fine volume predicted RGB colors for ray $\mathbf{r}$ respectively. Note that even though the final rendering comes from $\hat{C}_f(\mathbf{r})$, we also minimize the loss of $\hat{C}_c(\mathbf{r})$ so that the weight distribution from the coarse network can be used to allocate samples in the fine network.

In our experiments, we use a batch size of 4096 rays, each sampled at $N_c = 64$ coordinates in the coarse volume and $N_f = 128$ additional coordinates in the fine volume. We use the Adam optimizer with a learning rate that begins at 5×10^{-4} and decays exponentially to 5×10^{-5} over the course of optimization (other Adam hyperparameters are left at default values of $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 10^{-7}$). The optimization for a single scene typically take around 100–300k iterations to converge on a single NVIDIA V100 GPU (about 1–2 days).

4.2. Metrics

4.2.1. PSNR (Peak Signal-to-Noise Ratio)

PSNR is a widely used metric in image processing and computer vision. It measures the ratio between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation. In other words, PSNR measures the quality of the reconstructed image by comparing it to the original image. It is calculated as:

$$PSNR = 10 \log_{10} \left(\frac{MAX_I^2}{MSE} \right) \quad (7)$$

where MAX_I is the maximum pixel value (255 for an 8-bit color image), and MSE is the mean-square error between the original and reconstructed images.

One of the main drawbacks of PSNR is that it does not take into account the perceptual quality of the image. Human perception of image quality is complex and depends on various factors such as texture, contrast, and color. PSNR only measures the mean squared error between the original and reconstructed images, which may not be a good indicator of the perceived image quality.

Moreover, PSNR assumes that the noise in the reconstructed image is additive and follows a Gaussian distribution. However, this assumption may not always hold in practice, and PSNR may not be a reliable measure of image quality in such cases.

Another limitation of PSNR is that it depends on image size, as it is directly affected by amount of pixels, therefore is sensitive to certain image transformations like scaling and compression.

Therefore an additional metric SSIM is used to estimate the quality of the rendering result.

4.2.2. SSIM (Structural Similarity Index)

SSIM is a metric that measures the structural similarity between two images, taking into account luminance, contrast, and structure. It is designed to better reflect the human perception of image quality. SSIM ranges from -1 to 1, where 1 indicates perfect similarity between the two images. The formula for SSIM is:

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \quad (8)$$

where μ is the mean of the image, σ is the standard deviation, and c_1 and c_2 are constants to avoid division by zero. Because SSIM takes into account the structural information of the image, it is less affected by changes in scale and compression, and can provide a more accurate measure of image quality than PSNR in these scenarios.

4.3. Datasets

Both datasets where taken from the same creative common open licensed set used by (Mildenhall et al., 2020)

4.3.1. LEGO

Lego is a dataset consisting of 400 .jpg images, split between 100 (train) 100 (validate) and 200 (test). Each image is a 800x800 pixel 32 bits per pixel. The main object is surrounded by a white void (synthetic). Pictures are taken from all angles and directions (360 degree).

4.3.2. FERN: A REAL FACE FORWARD IMAGE

Fern dataset consist of 20 .jpg. Each images has 3 level of qualities, 4032 x 3024 pixels, 1008 x 756 and 504 x 378 pixels, all 24 bits per pixels. Object is surrounded by its background (real). All 20 pictures are taken from very similar angles (face forward). We used the lowest quality image, using images 0, 8 and 16 for test and validation.

4.4. Data preprocessing

Our method renders views by querying the neural radiance field representation at continuous $5D$ coordinates along camera rays. For experiments with synthetic images, we scale the scene so that it lies within a cube of side length 2 centered at the origin, and only query the representation within this bounding volume. Our data set of real images contains content that can exist anywhere between the closest point and infinity, so we use normalized device coordinates to map the depth range of these points into $[-1, 1]$. This shifts all the ray origins to the near plane of the scene, maps the perspective rays of the camera to parallel rays in the transformed volume, and uses disparity (inverse depth) instead of metric depth, so all coordinates are now bounded.

4.5. Training parameters

For real scene data, we regularize our network by adding random Gaussian noise with zero mean and unit variance to the output σ values (before passing them through the ReLU) during optimization, finding that this slightly improves visual performance for rendering novel views.

To render new views at test time, we sample 64 points per ray through the coarse network and $64 + 128 = 192$ points per ray through the fine network, for a total of 256 network queries per ray. Our realistic synthetic data set requires 640k rays per image, and our real scenes require 762k rays per image, resulting in between 150 and 200 million network queries per rendered image. On an NVIDIA V100, this takes approximately 30 seconds per frame.

We used 250000 iteration for FERN data set and 200000 for LEGO one.

As an optimizer we used the Adam one with $lr = 5 \cdot 10^{-3}$. Also we used a scheduler with $lr\_decay = 250$ and $lr\_decay\_factor = 0.1$.

The weights initialization is a standard one for PyTorch modules.

Figure 8 shows our fully-connected network architecture.

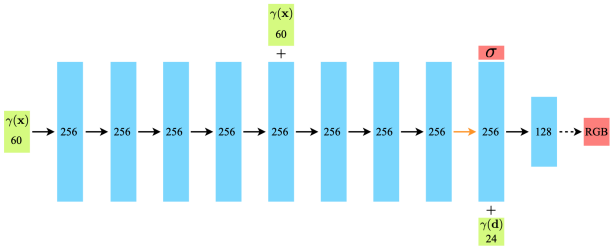

Figure 8. A visualization of our fully-connected network architecture. Input vectors are shown in green, intermediate hidden layers are shown in blue, output vectors are shown in red, and the number inside each block signifies the vector's dimension. All layers are standard fully-connected layers, black arrows indicate layers with ReLU activation, orange arrows indicate layers with no activation, dashed black arrows indicate layers with sigmoid activation, and "+" denotes vector concatenation. The positional encoding of the input location ($\gamma(\mathbf{x})$) is passed through 8 fully-connected ReLU layers, each with 256 channels. We follow the DeepSDF architecture and include a skip connection that concatenates this input to the fifth layer's activation. An additional layer outputs the volume density σ (which is rectified using a ReLU to ensure that the output volume density is non negative) and a 256-dimensional feature vector. This feature vector is concatenated with the positional encoding of the input viewing direction ($\gamma(\mathbf{d})$), and is processed by an additional fully-connected ReLU layer with 128 channels. A final layer (with a sigmoid activation) outputs the emitted RGB radiance at position $\mathbf{x}$, as viewed by a ray with direction $\mathbf{d}$.

5. Implementation of Nerf Algorithm

5.1. Introduction:

The NeRF algorithm works by representing a 3D scene as a continuous 5D function that maps any 3D point in space to its corresponding radiance (i.e., the amount of light that enters the eye from that direction). This function is learned using a neural network that takes in a set of 2D images of the scene as input and produces the corresponding 5D radiance field as output.

The radiance field is represented as a multi-layer perceptron (MLP), where each layer of the MLP corresponds to a set of neurons that learn to represent a different level of detail in the scene. By learning to predict the radiance of every point in 3D space, NeRF is able to synthesize highly detailed and realistic images of complex 3D scenes, even in the presence of occlusions and other challenging visual effects.

5.2. Methodology:

We used the provided GitHub repository to access the NeRF algorithm's implementation. We executed the code on two datasets: the synthetic 360 Lego dataset and the real forward facing LLFF Fern dataset. We replicated (Mildenhall et al., 2020) results.

5.3. Experimental results

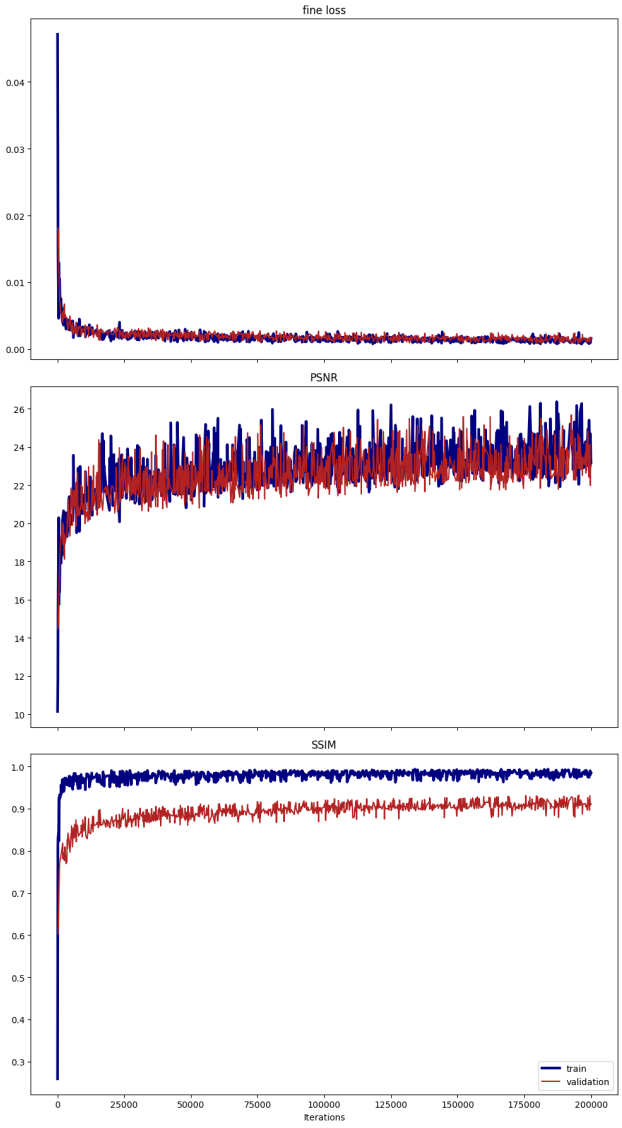

Figure 9. Loss, PSNR, SSIM for Lego dataset

Table 1 shows a significant difference in PSNR and SSIM metrics between the two datasets. Estimated poses result in a noticeable decrease in NeRF model performance due to its heavy reliance on accurate camera poses for generating realistic novel views. Artifacts in generated images can be attributed to the algorithm's limitations, such as its inability to handle large-scale scenes and the need for substantial training data. Future research can investigate techniques for handling large-scale scenes and developing methods to handle sparse camera view datasets better.

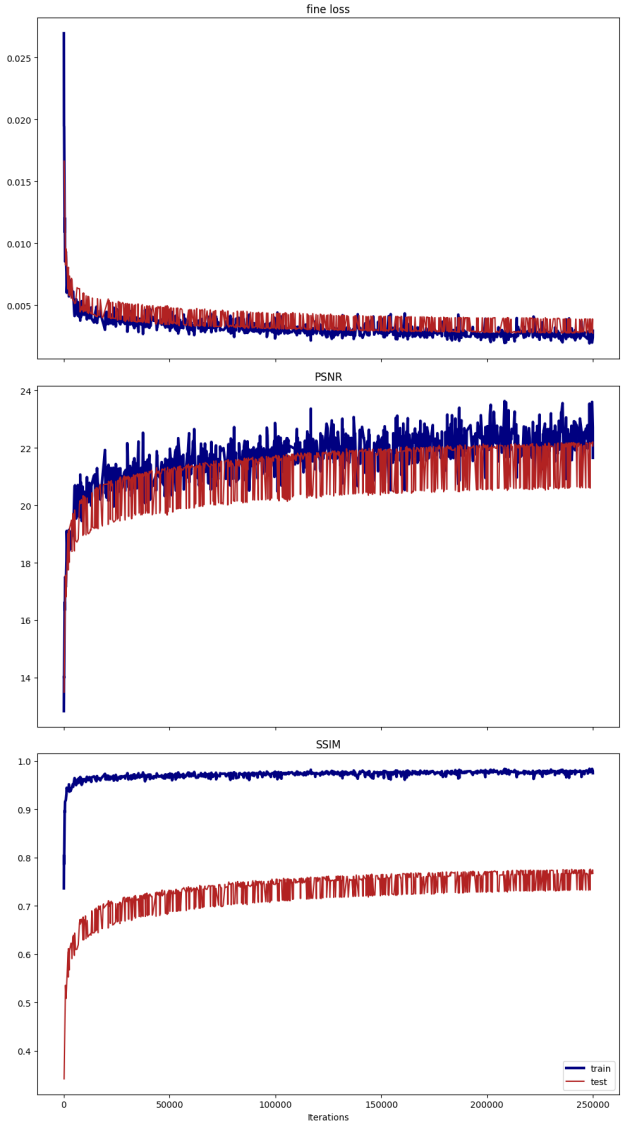

Figure 10. Loss, PSNR, SSIM for Fern dataset

Table 1. Comparison of Skoltech and Berkeley train metrics on 2 scenes

	PSNR(s)	PSNR(b)	SSIM(s)	SSIM(b)
Fern	21.66	25.17	0.73	0.79
Lego	22.84	32.54	0.97	0.96

Table 2. Time of reconstructing the images

	Mean	Std
Fern	2.626	0.003
Lego	1.504	0.001

5.4. Conclusion

The NeRF algorithm demonstrates impressive results in novel view synthesis tasks, producing high-quality images with realistic lighting and textures. However, the performance heavily depends on the accuracy of camera poses and intrinsic parameters. The artifacts observed in the generated images can be attributed to the algorithm's limitations, such as the inability to handle large-scale scenes and the requirement for substantial training data. The slow and impact results of the NeRF algorithm encourage us to find a way to provide additional data to the neural network. The idea is to provide the camera to world transformation matrices provided by COLMAP.

6. Retraining NeRF with Colmap poses

6.1. Introduction

COLMAP is a computer vision software used for 3D reconstruction from images, which is open source and based on various research papers. It provides a suite of algorithms for feature detection, feature matching, and 3D structure estimation from 2D images. In particular it can extract the position of the camera angles and the matrix transformations from each camera to world coordinates.

6.2. Methodology

The synthetic dataset was utilized to train and test the NeRF model with exact camera-to-world matrices and intrinsic parameters instantly provided by COLMAP. In contrast, the real dataset required extensive use of the COLMAP framework to extract camera-to-world matrices and intrinsic parameters. To address the convergence issue with the COLMAP algorithm, we fine-tuned the hyperparameters and optimization strategies using a Python script, resulting in more accurate pose estimations for the real LLFF Fern dataset. NeRF performance on synthetic and real datasets: The NeRF model was trained and tested on both the synthetic Lego dataset and real LLFF Fern dataset.

6.3. Experimental results

Table 3. Train and test metrics for NeRF and NeRF with COLMAP on 2 scenes

	PSNR(tr)	PSNR(t)	SSIM(tr)	SSIM(t)
NeRF				
Fern	16.52	21.16	0.97	0.73
Lego	22.84	22.59	0.98	0.89
NeRF with COLMAP				
Fern	14.98	12.37	0.84	0.32
Lego	16.52	7.32	0.85	0.51

Table 4. Training time and time per image of NeRF and NeRF with COLMAP on 2 scenes

Time for each alg	Lego	Fern
NeRF (train time)	10.66h	12.50h
NeRFC (train time)	9.00h	11.50h
NeRF (time/img)	1.504 ± 0.001s	2.626s $\pm$0.003s
NeRFC (time/img)	1.50s	2.62s

6.4. Conclusion

The COLMAP framework can be beneficial in real-world scenarios where accurate pose estimation is challenging or time-consuming. Nevertheless, the decrease in performance when using estimated poses emphasizes the importance of accurate pose estimation for the NeRF algorithm. The artifacts observed in the generated images can be attributed to the algorithm's limitations, such as the inability to handle large-scale scenes and the requirement for substantial training data.

7. Improvement of the NeRF Algorithm with Instant NGP

7.1. Introduction

"Instant Neural Graphics Primitives" (iNGP) is a method proposed in a research paper by NVIDIA (MÜLLER et al., 2022) to encounter it's drawbacks, such as slow training and rendering time, effective only with static object, requires many different camera positions. The paper proposes a neural rendering framework that can represent complex 3D scenes with a small set of neural primitives. The primitives are learned from training data and can be used to generate new views of the scene with high visual quality and high efficiency.

Graphic Primitives are the components needed to create complex images, including Points, Line, and Polygon. Graphic Primitives are encoded from images using a neural network, and then NeRF (Neural Radiance Fields) is generated through an inflation process. However, the size of the encoding results that can produce various changes is very large, so it should be used effectively, and for this purpose, a multi-resolution hash table is used, which is learned and made by a neural network. The encoded information contains multi-resolution information, and it can be said that the hash table was used to make efficient use of it.

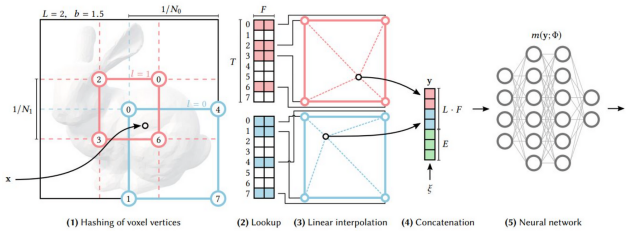

Figure 11. Multiresolution hash encoding

Here are the following steps of Instant-NGP:

- For a given input of coordinate x, we find a surrounding voxels at L resolution levels and assign indices to their corners by hashing their coordinates
- For all resulting corner indices, we look up the corresponding F-dimensional feature vectors from the has tables θ_l
- Linearly interpolate them according to the relative position of x within the respective l-th voxel
- Concatenate the result of each level as well as auxiliary inputs, producing encoded MLP input, which is evaluated the last
- To train the encoding, loss gradients are backpropagated through the MLP, the linear interpolation and then accumulated in the looked up feature vectors

The Instant-NGP method has shown promising results in generating high-quality images of complex 3D scenes with efficient rendering times. The method has potential applications in fields such as virtual reality, gaming, and computer graphics.

7.2. Methodology

We integrated the Instant-NGP approach into the NeRF algorithm by replacing the standard neural radiance fields with Instant-NGP-based neural primitives. This involved modifying the NeRF model architecture, training procedures, and rendering equations to accommodate the Instant-NGP framework. We then trained and tested the modified NeRF model on the synthetic Lego dataset and the real LLFF Fern dataset.

7.3. Experimental results

Table 5. Train and test metrics for NeRF and Instant-NGP on 2 scenes

	PSNR(tr)	PSNR(t)	SSIM(tr)	SSIM(t)
NeRF				
Fern	21.66	21.16	0.97	0.73
Lego	22.84	22.59	0.98	0.89
Instant-NGP				
Fern	39.8	37.8	0.98	0.97
Lego	16.5	11.2	0.87	0.69

Table 6. Training time and time per image of NeRF and Instant-NGP on 2 scenes

Time for each alg	Lego	Fern
NeRF (training time)	10.66h	12.50h
Instant-NGP (training time)	0.50h	0.40h
NeRF (time per image)	1.50s	2.63s
Instant-NGP (time per image)	0.536s	4.4s

7.4. Conclusion

Instant-NGP is significantly faster than NeRF with a small loss of quality.

8. Computing infrastructure

In this section, we describe the computing infrastructure used to conduct our experiments and analyze our results.

8.1. Hardware

Hardware

Our experiments were performed on a cluster of computing resources consisting of the following hardware:

- Nvidia Tesla V100 16GB
- Skoltech cloud 100 GB RAM, CPU unknown

8.2. Software

We used a variety of software tools to conduct our experiments and analyze our results, including:

- CUDA 10.2
- Python 3.8
- GCC/G++ 8
- CMake v3.21
- COLMAP 3.8

With the following dependencies

- channels:
  - pytorch
  - conda-forge
  - defaults
- dependencies:
  - python>=3.7 -
  - pip
  - pytorch>=1.3
  - commentjson
  - cuda100
  - cudatoolkit>=10.1
  - cudatoolkit-dev
  - imageio
  - pybind11
  - pyquaternion
  - numpy
  - scipy
  - opencv-python-headless
  - pyyaml
  - tensorboard
  - torchvision
  - tqdm
  - pip:
    - opencv-python-headless
    - git

8.3. Computing Environment

Our computing environment was configured as follows:

- CentOS Linux release 7 (Kernel 5.4)

References

Furukawa, Y., Hernández, C., et al. Multi-view stereo: A tutorial. *Foundations and Trends® in Computer Graphics and Vision*, 9(1-2):1–148, 2015.

Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., and Ng, R. Nerf: Representing scenes as neural radiance fields for view synthesis, 2020.

MÜLLER, T., EVANS, A., SCHIED, C., and KELLER, A. Instant neural graphics primitives with a multiresolution hash encoding, 2022. URL https://nvlabs.github.io/instant-ngp/assets/mueller2022instant.pdf.

Rahaman, N., Baratin, A., Arpit, D., Draxler, F., Lin, M., Hamprecht, F. A., Bengio, Y., and Courville, A. On the spectral bias of neural networks, 2019.

Zhang, K., Riegler, G., Snavely, N., and Koltun, V. Nerf++: Analyzing and improving neural radiance fields. *arXiv preprint arXiv:2010.07492*, 2020.

A. Appendix A: Contributions of the Team Members

Explicitly stated contributions of each team member to the final project.

Mariush Soroka (20% of work)

- Study the mathematical model
- Analyzing logs
- Conducting experiments and computing metrics
- Setting up additional remote network for additional computing resources
- worked on the code base
- added matrix and time calculation to train loop
- added random batching
- compiled Instant NGP and python binding

Yunseok Park (20% of work)

- Coding the main algorithm
- Preparing the report
- Doing the task 2
- Preparing the slides for the presentation

Anton Labutin (20% of work)

- Doing the task 3
- Preparing the slides for the presentation
- Preparing the report
- Literature review

Vladislav Mityukov (20% of work)

- Owner of the GitHub Repository
- System administrator
- Responsible for reproduciblity of NeRF algorithm

Marco Offidani (20% of work)

- Doing the task 3
- Writing the report

B. Reproducibility checklist

Answer the questions of following reproducibility checklist. If necessary, you may leave a comment.

1. A ready code was used in this project, e.g. for replication project the code from the corresponding paper was used.
   - ☑ Yes.
   - ☐ No.
   - ☐ Not applicable.

   General comment: Our code is based on another GitHub *repo*. We used the procedures for downloading the data sets. Changed the train loop, added the calculation of the metrics, implemented random batching. Conducted experiments with caching the rays from data sets but it took too much time to read the rays files from a disk on each iteration. That is why we decided to calculate the rays on each iteration.

2. A clear description of the mathematical setting, algorithm, and/or model is included in the report.
   - ☑ Yes.
   - ☐ No.
   - ☐ Not applicable.

3. A link to a downloadable source code, with specification of all dependencies, including external libraries is included in the report.
   - ☑ Yes.
   - ☐ No.
   - ☐ Not applicable.

4. A complete description of the data collection process, including sample size, is included in the report.
   - ☑ Yes.
   - ☐ No.
   - ☐ Not applicable.

5. A link to a downloadable version of the dataset or simulation environment is included in the report.
   - ☑ Yes.
   - ☐ No.
   - ☐ Not applicable.

   Students' comment: None

6. An explanation of any data that were excluded, description of any pre-processing step are included in the report.
   - ☑ Yes.
   - ☐ No.
   - ☐ Not applicable.

7. An explanation of how samples were allocated for training, validation and testing is included in the report.
   - ☑ Yes.
   - ☐ No.
   - ☐ Not applicable.

8. The range of hyper-parameters considered, method to select the best hyper-parameter configuration, and specification of all hyper-parameters used to generate results are included in the report.
   - ☐ Yes.
   - ☑ No.
   - ☐ Not applicable.

9. The exact number of evaluation runs is included.
   - ☑ Yes.
   - ☐ No.
   - ☐ Not applicable.

10. A description of how experiments have been conducted is included.
    - ☑ Yes.
    - ☐ No.
    - ☐ Not applicable.

11. A clear definition of the specific measure or statistics used to report results is included in the report.
    - ☑ Yes.
    - ☐ No.
    - ☐ Not applicable.

12. Clearly defined error bars are included in the report.
    - ☑ Yes.
    - ☐ No.
    - ☐ Not applicable.

13. A description of the computing infrastructure used is included in the report.
    - ☑ Yes.
    - ☐ No.
    - ☐ Not applicable.