Bachelor Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

The NeRF Utilization in Dense Reconstruction

Jakub Sakař

Supervisor: Ing. Michal Polic
Field of study: Open Informatics
Subfield: Artificial Intelligence and Computer Science
May 2023

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Sakař Jakub**	Personal ID number:	**499233**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

The NeRF Utilization in Dense Reconstruction

Bachelor’s thesis title in Czech:

Využití NeRF v rámci husté rekonstrukce

Guidelines:

1) Review the state-of-the-art NeRF-based methods (i.e., Mip-NeRF, DS-NeRF, and Efficient-NeRF) for rendering new views from a set of RGB images.
2) Unify the notation and describe the fundamental ideas of listed methods.
3) Discuss in detail the DS-NeRF loss function and its utilization for optimization of the depth maps.
4) Integrate the DS-NeRF loss function into the Mip-NeRF and verify its functionality on a set of synthetic RGB-D images.
5) Modify the input data from an AR device to a suitable format and verify the functionality of step 4) this dataset.
6) [Optional] Add the efficient sampling and hashing of the 3D space voxels utilizing Efficient-NeRF.

Bibliography / sources:

[1] Barron, Jonathan T., et al. "Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields." Proceedings of the IEEE/CVF International Conference on Computer Vision. 2021.
[2] Deng, Kangle, et al. "Depth-supervised nerf: Fewer views and faster training for free." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2022.
[3] Hu, Tao, et al. "EfficientNeRF Efficient Neural Radiance Fields." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2022.

Name and workplace of bachelor’s thesis supervisor:

Ing. Michal Polic Applied Algebra and Geometry CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **02.01.2023** Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Ing. Michal Polic	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student’s signature

Acknowledgements

I would mainly like to thank my supervisor, Michal Polic, for valuable guidance and motivation which helped me a lot to work on the thesis consistently. Furthermore, I would like to thank my family and friends who have mentally supported me throughout the work on this thesis. Lastly, I would like to thank the Czech Institute of Informatics, Robotics and Cybernetics for equipment that was used for experiments.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 25, 2023 ....

Signature

Abstract

This thesis provides an overview of methods for Novel view synthesis based on Neural Radiance Fields. It focuses on improving mip-NeRF by incorporating dense depth information. That is achieved by implementing depth supervision loss functions from the DS-NeRF method into the MultiNeRF implementation of mip-NeRF. One of the loss functions is then modified and implemented into our code repository. The loss functions were used for experiments that compare the results of the modified mip-NeRF method to the results of the unmodified mip-NeRF implementation. Finally, all methods have been evaluated on synthetic and real-world datasets.

Keywords: Artificial intelligence, Computer graphics, Novel view synthesis, NeRF, 3D reconstruction

Supervisor: Ing. Michal Polic
Czech Institute of Informatics, Robotics and Cybernetics (CIIRC CTU),
Jugoslávských partyzánů 1580/3,
160 00 Praha 6

Abstrakt

Tato práce poskytuje přehled metod využívaných k syntéze nových snímků 3D scény, které jsou založené na metodě Neural Radiance Fields. Klade si za cíl zlepšit metodu mip-NeRF s využitím husté informace o hloubce. Toho je dosaženo vložením ztrátových funkcí, založených na hloubkové supervizi, z metody DS-NeRF do metody mip-NeRF implementované v repozitáři MultiNeRF. Jedna ze ztrátových funkcí byla navíc modifikována. Tyto ztrátové funkce byly použity k experimentům, které porovnávají výsledky modifikované metody mip-NeRF s výsledky její původní varianty na syntetických datech i datech z reálného světa.

Klíčová slova: Umělá inteligence, Počítačová grafika, Novel view synthesis, NeRF, 3D rekonstrukce

Překlad názvu: Využití NeRF v rámci husté rekonstrukce

Contents

Figures

Tables

Chapter 1

Introduction

The task of synthesizing a target image with an arbitrary target camera pose from given source images and their camera poses is known as novel view synthesis [7]. In recent years, the popularity of novel view synthesis has grown steadily due to its increasing capacity to generate highly realistic images. This performance improvement can be attributed to the release of Neural Radiance Fields, a method that introduced an innovative approach to synthesizing novel views using neural networks.

Although multiple novel view synthesis methods based on neural radiance fields achieved realistic results, none of them utilized information about depth. One of the methods, DS-NeRF, used depths in the sense that it computed distances of several keypoints from each camera in the scene. However, none of the methods researched in this thesis use dense depth data like LiDAR depth images. Using the depth supervision that DS-NeRF provides while leveraging the dense depth data could lead to faster convergence and robustness to a low number of input images.

This thesis aims to research recent novel view synthesis methods based on Neural Radiance Fields. Two of these methods will be tested on a prepared set of datasets, where one is synthetic while the other is created from real-world photographs. The results of these two methods will be compared and discussed. The depth supervision used in the DS-NeRF method will then be added to the existing mip-NeRF implementation. It will be first tested on the synthetic dataset to see if the depth supervision can improve the mip-NeRF method's results. The same code will be run for the real-world dataset to check if dense depth data readily available in AR devices like iPad can improve the training process by increasing its convergence.

Chapter 2

Preliminaries

2.1 Multilayer Perceptron

The construct NeRF-based methods use to represent a 3D scene is called the *multilayer perceptron* (MLP). Its name originates from the idea of Perceptron, initially created in 1957 by Frank Rosenblatt [8]. *Perceptron* is an algorithm that was initially created as an image recognition machine. It defines a *neuron* as a unit combining several inputs in a weighted sum and producing a single scalar output. Perceptron was initially used for binary classification based on the sign of the weighted sum.

Formally, the perceptron is trained on a set of data, which has the following form

$$\mathcal{T} = \{(\mathbf{x}_1, k_1)...(\mathbf{x}_L, k_L)\} \tag{2.1}$$

where $\mathbf{x}_i$ is an i-th *feature vector* and k_i is its corresponding *class*. Because perceptron is a binary classifier, there are only two possible classes for each feature vector, 1 and -1. The goal of the perceptron algorithm is to find *weights* $\mathbf{w}$ and a *bias* w_0 (often denoted b) such that the following condition is satisfied

$$\begin{aligned} &\mathbf{w} \cdot \mathbf{x}_j + w_0 > 0 \quad \text{if} \quad k_j = 1, \quad (\forall j \in \{1, 2, ..., L\}) \\ &\mathbf{w} \cdot \mathbf{x}_j + w_0 < 0 \quad \text{if} \quad k_j = -1 \end{aligned} \tag{2.2}$$

This condition can be further simplified by concatenating the weights with the bias and concatenating each feature vector with one and multiplying both inequalities with the class k_j, which results in a single inequality [27]

$$\mathbf{w}' \cdot \mathbf{x}_j k_j > 0 \quad (\forall j \in \{1, 2, ..., L\}) \tag{2.3}$$

$$\mathbf{x}' = \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} \quad \mathbf{w}' = \begin{bmatrix} w_0 \\ \mathbf{w} \end{bmatrix} \tag{2.4}$$

Perceptron was later enhanced by transforming the dot product $\mathbf{w}' \cdot \mathbf{x}'_j$ using a non-linear function, which is also called an activation function. One of the earliest activation functions is the *sigmoid function*, which is defined as

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{2.5}$$

Another function that is also used very frequently is the *rectified linear unit* (ReLU) function, which is defined as

$$R(x) = \max(0, x) \tag{2.6}$$

Because both functions map all values to the range $[0, 1]$, the inequality formerly defined for the perceptron algorithm can no longer be used for binary classification. However, these functions became much more useful with the advent of multilayer perceptron [26].

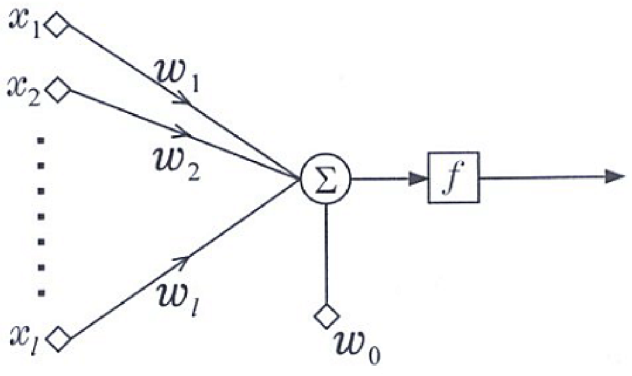


Figure 2.1: Perceptron with an activation function f, from [27]

Multilayer perceptron was created by interconnecting multiple neurons and creating a network that consists of several layers. The first layer is called the input layer, and its inputs are used as the inputs to the MLP, while the last layer is called the output layer, and its outputs are used as the outputs of the MLP. The remaining layers are called hidden layers; their inputs and outputs usually don't give any information about the model.

A multilayer perceptron can be considered a deep artificial neural network, which utilizes the *backpropagation* algorithm as a technique that is used to repeatedly adjust the weights to achieve desired results. These results are typically achieved using the stochastic gradient descent optimization method. The MLP computation is performed in two passes. The first pass, the *forward pass*, is used to compute the neural network output. This pass computes each layer's output based on the previous layer's output because each layer requires the previous layer to be computed as it uses the results of the previous layer to compute its results. The second pass is the *backward pass*, which is used to compute the gradient of the MLP using the chain rule. Because computing each layer's gradient requires the next layer's gradient, the computation is done in the opposite order [25].

2.2 Coordinate Systems

Neural Radiance Fields and similar related methods work with the concept of 3D coordinates and their transformations. In order to describe these transformations, we first need to define coordinate systems. *Coordinate systems* are a concept used in computer graphics, mathematics, and physics to determine an object's position in space. Coordinate systems are defined by a reference point called the *origin*, a set of axes with a specified range, and a unit vector for each axis [28].

This thesis will primarily use 3D *Cartesian coordinate systems*, which use three axes, x, y, and z. We further distinguish between *right handed* and *left handed* coordinate systems when talking about 3D Cartesian coordinate systems. A right handed coordinate system is defined such that if the thumb of the right hand points in the positive x direction and the index finger points in the positive y direction, the middle finger points in the positive z direction. We can analogously define left handed coordinate systems as can be seen in Figure 2.2 [9].

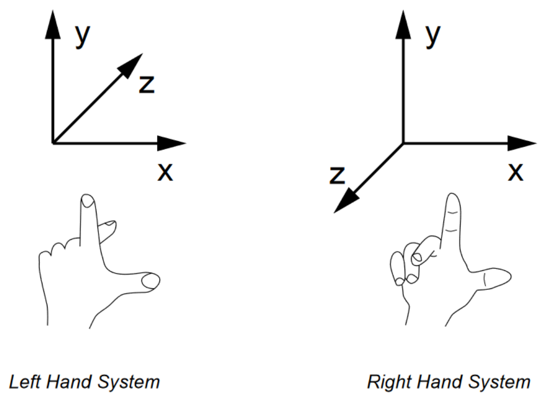


Figure 2.2: Left handed vs. right handed coordinate systems, from [9]

Left handed coordinate systems are often used in game engines like Unity or Unreal Engine. In this thesis, we will use right handed coordinate systems, which are used in OpenGL and COLMAP. See Figure 2.3 for examples. Although OpenGL and COLMAP coordinate systems are right handed, they do not have the same orientation. In order to use convert the scene from the OpenGL coordinate system to the one used by COLMAP., the system needs to be rotated by 180 degrees along the x-axis, which means that signs of both the y-axis and z-axis get flipped [29].

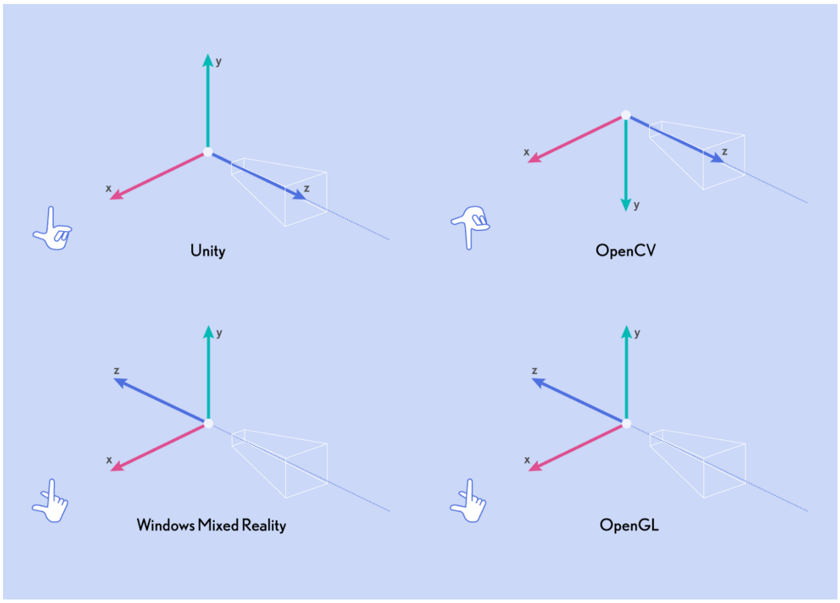


Figure 2.3: Comparison of different coordinate systems of the camera, from [10]

2.3 Transformations

The major part of computer graphics is about the issue of rendering an image from a 3D scene representation, while computer vision often deals with the opposite problem. Both fields of study thus require a way to represent transformations between different coordinate systems.

2.3.1 Homogeneous Coordinates

Unlike linear transformations, *affine transformations* in 3D cannot be generally described using 3x3 matrices. This limitation can be solved by adding a fourth coordinate, which is set to one for each 3D vector, so the coordinates of every vector can be described as

$$\mathbf{x}_h = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.7}$$

where $[x, y, z]$ are the coordinates of the original 3D vector.

Affine transformation of a vector $\mathbf{x}$ to a vector $\mathbf{x}'$ can also be represented as a linear transformation followed by a translation, which can be described using a 3x3 matrix and an offset vector:

$$\mathbf{x}' = \begin{bmatrix} a_1 & a_3 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix} \mathbf{x} + \begin{bmatrix} t_1 \\ t_2 \\ r_3 \end{bmatrix} \tag{2.8}$$

This representation can be easily converted to *homogeneous coordinates*

$$\mathbf{x}' = \begin{bmatrix} a_1 & a_2 & a_3 & t_1 \\ b_1 & b_2 & b_3 & t_2 \\ c_1 & c_2 & c_3 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{x_h} \tag{2.9}$$

where $\mathbf{x_h}$ is the vector $\mathbf{x}$ in homogeneous coordinates. After performing an arbitrary transformation in homogeneous coordinates, the fourth coordinate may no longer be equal to one. In this case, the first three coordinates have to be normalized by dividing by the fourth coordinate, so for each vector in homogeneous coordinates $[x, y, z, w]$, their corresponding vector in 3D is equal to

$$\frac{1}{w} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.10}$$

Every affine transformation can be described as a combination of translation, rotation, scale and shear, but most transformations in computer vision and graphics do not use shear, which is why it will not be mentioned in this thesis.

Translation is an operation that translates all points of an object by the same vector $\mathbf{t}$. Such transformation can be described using the following matrix

$$T = \begin{bmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.11}$$

Rotation is a transformation that rotates all points of an object around an arbitrary axis. This definition is difficult to represent using matrices, so rotation in the 3D space is defined as a set of three rotations where each is done around a different coordinate system axis. To represent rotation along the x-axis by an amount θ, we can use the following matrix

$$R_x = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.12}$$

Analogously, rotation along the y- and z-axes can be represented as

$$R_y = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.13}$$

$$R_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.14}$$

respectively. Finally, *scale* can be simply represented by changing values on the diagonal using a 3D vector $\mathbf{s}$ because the scale along each axis does not affect any other axis [30]

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.15}$$

2.3.2 Transformations in OpenGL

OpenGL is an application programming interface that was designed for graphics programming. It allows for the rendering of 3D objects using custom shaders. While the transformations used in OpenGL may differ based on the application, most transformations can be represented using a model matrix, view matrix, projection matrix, and a hidden viewport transform matrix [11]. These transformations are depicted on figures 2.4 and 2.5.

The *model matrix* transforms vertices of individual objects from local space to world space, which is shared among all geometry in the rendered scene. The *view matrix* transforms vertices in world space into the view space (also called eye space). Both model and view matrices represent an affine transformation. The model matrix is different for each model in the scene and depends on the position, rotation, and scale of the corresponding object. The view matrix is uniform for each vertex in the scene, and its values depend on the camera position and rotation [13].

The *projection matrix* transforms the scene from view space to clip space. The primary purpose of clip space is discarding everything that is not in view frustum of the camera. This frustum is defined by six values, near plane n, far plane f, left bound l, right bound r, top bound t, and bottom bound b. The projection matrix maps the frustum to a cube, which makes it easy to tell what lies outside of this camera frustum. After performing this step, the clip coordinates are normalized into a cube that lies in the range of $[-1, 1]^3$. The space that is computed by normalizing the clip space is called *normalized device coordinates* (or NDC for short).

Two types of projection can be performed: orthographic projection and perspective projection. The *orthographic projection* is more straightforward as its x and y coordinates do not depend on the z coordinate. Generally, an orthographic projection matrix can be written as

$$P_{\text{ortho}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.16}$$

The matrix can be further simplified under the assumption of symmetrical volume, which means that $r = -l$ and $t = -b$.

$$P_{\text{ortho}} = \begin{bmatrix} \frac{1}{r} & 0 & 0 & 0 \\ 0 & \frac{1}{t} & 0 & 0 \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.17}$$

The *perspective projection* assumes that objects further away from the camera appear smaller and simulates real-life camera behavior. Its matrix form looks like this

$$P_{\text{pers}} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{2.18}$$

Similarly, this matrix can be simplified if we assume a symmetrical viewing frustum.

$$P_{\text{pers}} = \begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{2.19}$$

It is also important to note that, unlike all preceding transformations, the perspective projection transformation is no longer affine as it does not preserve parallel lines [14].

Viewport transform matrix defines a transformation from normalized device coordinates to window space, which is a 2D space. The matrix is parametrized by window size and position [15].

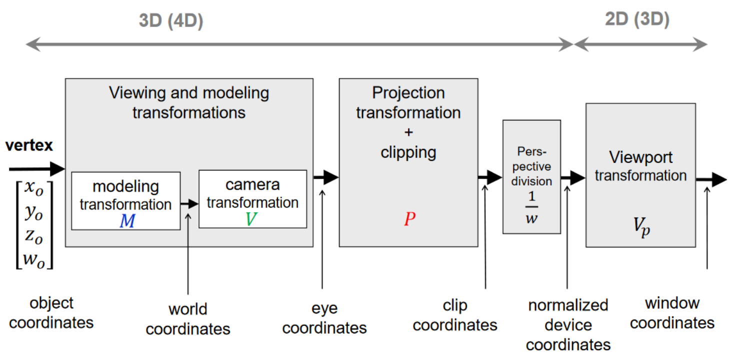


Figure 2.4: OpenGL rendering pipeline, from [12]

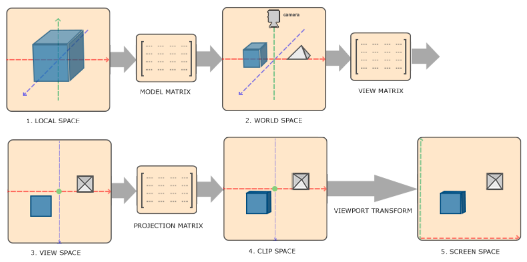


Figure 2.5: Visualization of transformations in OpenGL, from [13]

2.3.3 Transformations in Computer Vision

The idea of transformations in computer vision is similar to those in OpenGL. One of the key differences is the orientation of the y- and z-axes, as discussed in section 2.2. Another difference is the representation of the camera's intrinsic and extrinsic parameters.

Intrinsic parameters of a camera allow a mapping between camera coordinates and image coordinates, while extrinsic parameters define the position and rotation of the camera with respect to the world coordinate system [16]. These terms are not typically used in computer graphics, which is why they are described in this section.

Computer vision cameras are usually represented using a *pinhole camera model*, which does not model optical defects such as radial distortion. We will start by describing several terms that will be necessary to define the pinhole camera model. The camera is represented by its *optical center* C and the *image plane*. The distance between the optical center and the image plane is called *focal length*, denoted as f. The line perpendicular to the image plane and passing through the optical center is called the *optical axis*. The intersection of the optical axis with the image plane is called the *principal point*, denoted as $[o_x, o_y]$. The resulting image size can be further specified by the size of the *pixel footprint* on the camera photosensor $[s_x, s_y]$. All of the described parameters are the so-called intrinsic parameters of the camera. The only two extrinsic parameters of the camera are its position and orientation. Position can be described using a *translation vector* $\mathbf{t} \in \mathbb{R}^3$, and rotation can be described using a *rotation matrix* $R \in \mathbb{R}^{3\times3}$ [17].

Computer vision uses three coordinate systems, *world coordinate system*, *camera coordinate system*, and *image coordinate system* [**?**]. The world and camera coordinate systems are usually three-dimensional, while the image coordinate system is only two-dimensional. The transformation of a point in world coordinates $X_w \in \mathbb{R}^3$ to a point in camera coordinates $X_c \in \mathbb{R}^3$ is performed using a matrix $[R|\mathbf{t}]$. Because this resulting matrix is a 3x4 matrix, the point in world coordinates has to be expanded to homogeneous coordinates. The following equation describes how the point is transformed

$$X_c = \left[\; R \;\middle|\; \mathbf{t} \;\right] \begin{bmatrix} X_w \\ 1 \end{bmatrix} \tag{2.20}$$

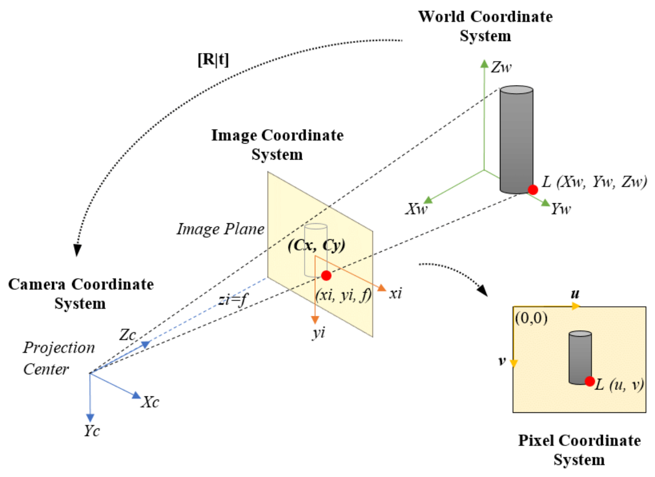


Figure 2.6: Pinhole camera model, from [18]

The transformation from camera coordinates to image coordinates is done using the *calibration matrix* $K \in \mathbb{R}^{3\times3}$, which can be constructed using the camera's intrinsic parameters. The general form of the calibration matrix is

$$K = \begin{bmatrix} \frac{f}{s_x} & \frac{f}{s_x}\cot(\theta) & o_x \\ 0 & \frac{f}{s_y} & o_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.21}$$

where θ is the angle between the x- and y-axes [17]. This matrix can be rewritten into a more readable form. First, $\frac{f}{s_x}$ and $\frac{f}{s_y}$ can be rewritten to f_x and f_y respectively. These two parameters are the focal lengths of the camera in terms of pixel dimensions in the x and y direction, respectively. The term $\frac{f}{s_x}\cot(\theta)$, which is referred to as the *skew* parameter can be denoted as s [31]. In this thesis, we will assume that this parameter is equal to zero, which means that the calibration matrix will be simplified to the following form

$$K = \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.22}$$

Using these two matrices, we can compute image coordinates of a point $\mathbf{u} \in \mathbb{R}^2$ from a point $X_w \in \mathbb{R}^3$ in world coordinates

$$\lambda \begin{bmatrix} \mathbf{u} \\ 1 \end{bmatrix} = K[\ R \mid \mathbf{t}\ ] \begin{bmatrix} X_w \\ 1 \end{bmatrix} \tag{2.23}$$

where $\lambda \in \mathbb{R}^+$ is the distance of X from the image plane of the camera [**?**]. Transformations in the pinhole camera model are depicted on Figure 2.6.

2.4 Alpha Compositing and Color Representation

¨ In order to render images from an arbitrary viewpoint, NeRF-based methods need to sample multiple points along a ray and compose colors in these points to get the resulting color. To understand how NeRF does this, we need to understand alpha compositing and how colors are represented in computer graphics. Alpha compositing, also called alpha blending, is a technique used in computer graphics, which is used to compose multiple layers of transparent images. Many different formats can represent color in computer graphics, but the most popular one is RGBA.

RGB is an additive color model inspired by the human eye's biology in the sense that the human eye contains three types of cones, each of which is sensitive to a different light wavelength. Combining these three components, red, green, and blue, we can produce any color perceivable by the human brain [32].

RGBA is an extension of the RGB color model, incorporating an additional alpha component. This component is used to represent the opacity of the given color. The most common variant of RGBA colors is a 32-bit RGBA color, where each component is represented as an 8-bit integer. This means that its values range between 0 and 255, where 0 represents a minimal intensity (black color), and 255 represents a full intensity. In the case of the alpha component, 0 represents a fully transparent color, whereas 255 represents a fully opaque color.

When performing alpha compositing, we consider a color $\mathbf{c}_1$ with an alpha component α and a background color $\mathbf{c}_0$, which is fully opaque. After blending the color $\mathbf{c}_1$ with the background the resulting color $\mathbf{c}$ can be calculated as [33]

$$\mathbf{c} = \alpha\mathbf{c}_1 + (1-\alpha)\mathbf{c}_0 \tag{2.24}$$

Chapter 3

NeRF-based Methods

3.1 Neural Radiance Fields

Neural Radiance Fields (NeRF) is a novel view synthesis method that uses an MLP to store information about a 3D scene. The perceptron is trained to predict color in RGB format and volumetric density σ for a given 3D point $\mathbf{x}$ and a given view angle $\mathbf{d}$, realized using a 3D vector, i.e.,

$$(\mathbf{x}, \mathbf{d}) \to (r, g, b, \sigma) \tag{3.1}$$

.

In order to render novel images of the scene, NeRF uses alpha compositing techniques commonly used in computer graphics. Each pixel in an image is rendered by casting a ray $\mathbf{r}$ through the scene and sampling several points on this ray. Each ray can be described using an origin $\mathbf{o}$ and a direction $\mathbf{d}$ as a parametric equation

$$\mathbf{x} = \mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \tag{3.2}$$

where $t \in \mathbb{R}$ is a parameter used to determine ray position.

After evaluating color $\mathbf{c}_i \in \mathbb{R}^3$ and volumetric density $\sigma_i \in \mathbb{R}$ for each of the sampled points, the final color $\hat{C}(\mathbf{r})$ is rendered according to following equations

$$\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i \delta_i))\mathbf{c}_i \tag{3.3}$$

$$T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right) \tag{3.4}$$

$$\delta_j = t_{j+1} - t_j \tag{3.5}$$

Because the alpha compositing weights that are being computed in equation 3.3 will often appear in the implementation, we will denote these weights as w_i

$$w_i = T_i \alpha_i \tag{3.6}$$

$$\alpha_i = (1 - \exp(-\sigma_i \delta_i)) \tag{3.7}$$

NeRF uses two MLPs with two different sampling strategies to achieve more precise results. The first MLP is called the *coarse* MLP, which is trained on data sampled uniformly along each ray. The second MLP is called the *fine* MLP, and its operation depends on the coarse MLP results. Using equation 3.6, we can compute alpha compositing weights for the coarse MLP and normalize them to get a probability distribution

$$\hat{w}_i = \frac{w_i}{\sum_{j=1}^{N_c} w_j} \tag{3.8}$$

where $N_c \in \mathbb{N}$ is a number of points sampled on each ray for the coarse MLP. This new probability distribution is then used to sample points for the fine MLP.

Since neural networks often omit low-frequency information, NeRF uses a positional encoding that encodes the MLP input using harmonic functions. Generally, this is done using a function γ, which uses a hyperparameter L. This hyperparameter defines how many pairs of harmonic functions will be used for the positional encoding. The encoding is defined as a function with a single scalar parameter applied to each input parameter separately. So, if a vector is used as an input, the function is computed for each component separately.

$$\gamma(p) = \Big(\sin(2^0 \pi p), \cos(2^0 \pi p), ..., \cos(2^{L-1} \pi p), \cos(2^{L-1} \pi p)\Big) \tag{3.9}$$

With the addition of positional encoding, both MLPs perform the following mapping

$$(\gamma_x(\mathbf{x}), \gamma_d(\mathbf{d})) \rightarrow (r, g, b, \sigma). \tag{3.10}$$

Encoding functions in this equation used subscripts because the L parameter has a different value for the encoding of the position and direction.

Another technique NeRF uses is hierarchical sampling, which means that NeRF trains two MLPs, where the results of one MLP are used to get more precise samples for the other. NeRF starts with sampling points uniformly for the coarse MLP. The output of this MLP is used to generate a non-uniform sampling, which is then used as an input for the fine MLP.

Both MLPs in NeRF are the same size and use the same parameters as input. Each MLP has eight layers with ReLU activation and 256 channels, where the first layer uses an encoded position as input. The encoded position is also concatenated to the fifth layer activation. The output of these layers consists of predicted volumetric density for the given point and a feature vector with a size of 256. This feature vector is then concatenated to the encoded view direction and passed through a single layer with 128 channels and a ReLU activation. The output layer, which uses a sigmoid activation, is a 3D vector representing the output color. For a diagram of the MLP architecture, see Figure 3.1.

3.1.1 NeRF Coordinate Systems

NeRF uses two different coordinate systems to sample the continuous 3D scene. The more straightforward approach is just using the world coordinates of the camera. However, this approach only works well for relatively small scenes and is only used for synthetic data created in Blender. The second approach assumes that the scene is "forward-facing," which means that all cameras are close to each other regarding view direction. This approach transforms the entire scene into the Normalized Device Coordinates space used in the OpenGL rendering pipeline.

Transformation to normalized device coordinates space can be derived from the perspective projection matrix

$$M = \begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{3.11}$$

where n is the near plane, f is the far plane, r is the right bound of the scene at the near plane, and t is the top bound at the near plane. If we consider the pinhole camera model, with a far bound of positive infinity, the matrix further simplifies to the following form

$$M = \begin{bmatrix} \frac{-2f_{cam}}{W} & 0 & 0 & 0 \\ 0 & \frac{-2f_{cam}}{H} & 0 & 0 \\ 0 & 0 & -1 & -2n \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{3.12}$$

where f_{cam} is the camera's focal length, and W and H are the width and height of the image in pixels. Using this matrix we can transform an arbitrary point $X \in \mathbb{R}$ in world coordinates to a point $X' \in \mathbb{R}$ in normalized device coordinates

$$X' = \frac{1}{z} M \begin{bmatrix} X \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2f_{cam}}{W} \frac{x}{z} \\ \frac{2f_{cam}}{H} \frac{y}{z} \\ -1 - \frac{2n}{z} \\ 1 \end{bmatrix} \tag{3.13}$$

In order to achieve consistency between all cameras, the origins $\mathbf{o} \in \mathbb{R}^3$ of each ray are shifted to the near plane at $z = -n$. The following set of equations defines a new origin

$$\mathbf{o}_n = \mathbf{o} + t_n \mathbf{d} \tag{3.14}$$

$$t_n = -\frac{n + o_z}{d_z} \tag{3.15}$$

The fact that the origin was moved means that after performing this transformation, all ray origins will have the same z coordinate. The negative near plane is used because NDC space is derived from the OpenGL coordinate system, where the camera faces the negative z axis [1].

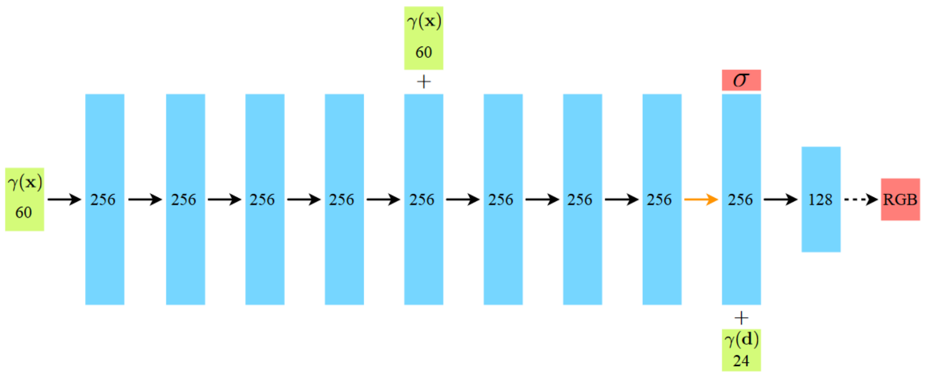


Figure 3.1: Architecture of the NeRF MLP, from [1]

3.2 mip-NeRF

Mip-NeRF extends upon the original NeRF method while removing aliasing from rays passing through the 3D scene. In order to remove aliasing, mip-NeRF replaces infinitesimally narrow rays with cones and predicts color for the entire space of conical frustums instead of just points. This approach is inspired by pre-filtering, which is one of the ways to avoid aliasing in the space domain. However, it is challenging to compute an integral over these frustums, so we use a multivariate Gaussian approximation, as shown in Figure 3.2.

The conical frustums are parameterized by their mid-point t_μ and half-width t_δ. The multivariate Gaussians are parameterized using mean distance along the ray μ_t, variance along the ray σ_t^2, and variance perpendicular to the ray σ_p^2. The following equations describe the relations between these two parametrizations

$$\mu_t = t_\mu + \frac{2t_\mu t_\delta^2}{3t_\mu^2 + t_\delta^2} \tag{3.16}$$

$$\sigma_t^2 = \frac{t_\delta^2}{3} - \frac{4t_\delta^4(12t_\mu^2 - t_\delta^2)}{15(3t_\mu^2 + t_\delta^2)^2} \tag{3.17}$$

$$\sigma_r^2 = \dot{r}^2 \left(\frac{t_\mu^2}{4} + \frac{5t_\delta^2}{12} - \frac{4t_\delta^4}{15(3t_\mu^2 + t_\delta^2)} \right) \tag{3.18}$$

where $\dot{r} \in \mathbb{R}$ is radius of the cone at the image plane.

Using these parameters, we can further compute the diagonal of the covariance matrix of the Gaussian and the mean of the Gaussian using the following equations

$$\text{diag}(\boldsymbol{\Sigma}) = \sigma_t^2(\mathbf{d} \circ \mathbf{d}) + \sigma_r^2 \left(\mathbf{1} - \frac{\mathbf{d} \circ \mathbf{d}}{\|\mathbf{d}\|_2^2}\right) \tag{3.19}$$

$$\mu = \mathbf{o} + \mu_t \mathbf{d}. \tag{3.20}$$

Now, we can perform positional encoding on the mean and covariance matrix diagonal

$$\mu_\gamma = [\mu, 2\mu, ..., 2^{L-1}\mu]^T \tag{3.21}$$

$$\text{diag}(\boldsymbol{\Sigma}_\gamma) = [\text{diag}(\boldsymbol{\Sigma}), 4\text{diag}(\boldsymbol{\Sigma}), ..., 4^{L-1}\text{diag}(\boldsymbol{\Sigma})]^T \tag{3.22}$$

$$\gamma(\mu, \boldsymbol{\Sigma}) = \begin{bmatrix} \sin(\mu_\gamma) \circ \exp(-(1/2)\text{diag}(\boldsymbol{\Sigma}_\gamma)) \\ \cos(\mu_\gamma) \circ \exp(-(1/2)\text{diag}(\boldsymbol{\Sigma}_\gamma)) \end{bmatrix}. \tag{3.23}$$

The $\circ$ symbol denotes the element-wise multiplication of vectors.

Mip-NeRF uses a 2-tap max filter on alpha compositing weights with a hyperparameter α.

$$w_i' = \frac{1}{2}(\max(w_{i-1}, w_i) + \max(w_i, w_{i+1})) + \alpha \tag{3.24}$$

Its primary purpose is to generate a smooth upper envelope on weights $\mathbf{w}$. To optimize the MLP, Mip-NeRF uses a loss function that is similar to the one in the original NeRF

$$\sum_{\mathbf{r} \in \mathcal{R}} \left(\lambda \|\mathbf{C}^*(\mathbf{r}) - \mathbf{C}^*(\mathbf{r}; \Theta^c, \mathbf{t}^c)\|_2^2 + \|\mathbf{C}^*(\mathbf{r}) - \mathbf{C}^*(\mathbf{r}; \Theta^f, \text{sort}(\mathbf{t}^c \cup \mathbf{t}^f))\|_2^2\right) \tag{3.25}$$

where $\mathbf{t^c}$ are sampling parameters of the coarse sampling and $\mathbf{t^f}$ are sampling parameters of the fine sampling and the sort function sorts values of these vectors from the smallest to the greatest. The only difference is that here, we optimize only a single MLP, although we still perform coarse and fine sampling separately, while NeRF uses a dedicated MLP for coarse sampling. In this case, C^* denotes the ground truth color rendered by the ray r.

Mip-NeRF uses three hyperparameters to fine-tune the training phase. The first parameter, L, defines the complexity of integrated positional encoding. L was set to 10 in the original Mip-NeRF implementation. The λ parameter defines how significant of an influence the coarse sampling loss function should be. It is set to 0.1 in the original Mip-NeRF implementation. Finally, the α parameter, set to 0.01 in the original Mip-NeRF implementation, is used to fine-tune the 2-tap max filter to ensure that samples appear even in empty regions of space.

Mip-NeRF extends upon the original NeRF method while removing aliasing from rays passing through the 3D scene. In order to remove aliasing, mip-NeRF replaces infinitesimally narrow rays with cones and predicts color for the entire space of conical frustums instead of just points. This approach is inspired by pre-filtering, which is one of the ways to avoid aliasing in the space domain. However, it is challenging to compute an integral over these frustums, which is why we use an approximation in the form of a multivariate Gaussian.

The conical frustums are parameterized by their mid-point t_μ and half-width t_δ. The multivariate Gaussians are parameterized using mean distance along the ray μ_t, variance along the ray σ_t^2, and variance perpendicular to the ray σ_p^2. The following equations describe the relations between these two parametrizations.

$$\mu_t = t_\mu + \frac{2t_\mu t_\delta^2}{3t_\mu^2 + t_\delta^2} \tag{3.26}$$

$$\sigma_t^2 = \frac{t_\delta^2}{3} - \frac{4t_\delta^4(12t_\mu^2 - t_\delta^2)}{15(3t_\mu^2 + t_\delta^2)^2} \tag{3.27}$$

$$\sigma_r^2 = \dot{r}^2\left(\frac{t_\mu^2}{4} + \frac{5t_\delta^2}{12} - \frac{4t_\delta^4}{15(3t_\mu^2 + t_\delta^2)}\right) \tag{3.28}$$

Using these parameters, we can further compute the diagonal of the covariance matrix of the Gaussian and the mean of the Gaussian using the following equations.

$$\operatorname{diag}(\mathbf{\Sigma}) = \sigma_t^2(\mathbf{d} \circ \mathbf{d}) + \sigma_r^2\left(\mathbf{1} - \frac{\mathbf{d} \circ \mathbf{d}}{\|\mathbf{d}\|_2^2}\right) \tag{3.29}$$

$$\mu = \mathbf{o} + \mu_t \mathbf{d} \tag{3.30}$$

Now, we can perform positional encoding on the mean and covariance matrix diagonal.

$$\mu_\gamma = [\mu, 2\mu, ..., 2^{L-1}\mu]^T \tag{3.31}$$

$$\operatorname{diag}(\mathbf{\Sigma}_\gamma) = [\operatorname{diag}(\mathbf{\Sigma}), 4\operatorname{diag}(\mathbf{\Sigma}), ..., 4^{L-1}\operatorname{diag}(\mathbf{\Sigma})]^T \tag{3.32}$$

$$\gamma(\mu, \mathbf{\Sigma}) = \begin{bmatrix} \sin(\mu_\gamma) \circ \exp(-(1/2)\operatorname{diag}(\mathbf{\Sigma}_\gamma)) \\ \cos(\mu_\gamma) \circ \exp(-(1/2)\operatorname{diag}(\mathbf{\Sigma}_\gamma)) \end{bmatrix} \tag{3.33}$$

The $\circ$ symbol denotes the element-wise multiplication of vectors. Mip-NeRF additionally uses a 2-tap max filter on alpha compositing weights with a hyperparameter α.

$$w'_{ij} = \frac{1}{2}(\max(w_{ij-1}, w_{ij}) + \max(w_{ij}, w_{ij+1})) + \alpha \tag{3.34}$$

Its main purpose is to generate a smooth upper envelope on weights w. To optimize the MLP, Mip-NeRF uses a loss function that is similar to the one in the original NeRF

$$\sum_{\mathbf{r}\in\mathcal{R}} \left(\lambda\|\mathbf{C}^*(\mathbf{r}) - \mathbf{C}^*(\mathbf{r}; \Theta^c, \mathbf{t}^c)\|_2^2 + \|\mathbf{C}^*(\mathbf{r}) - \mathbf{C}^*(\mathbf{r}; \Theta^f, \operatorname{sort}(\mathbf{t}^c \cup \mathbf{t}^f))\|_2^2\right) \tag{3.35}$$

The only difference is that here, we optimize only a single MLP, although we still perform coarse and fine sampling separately, while NeRF uses a dedicated MLP for coarse sampling. In this case, C^* denotes the ground truth color rendered by the ray **r.

Mip-NeRF uses three hyperparameters to fine-tune the training phase. The first parameter, L, defines the complexity of integrated positional encoding. L was set to 10 in the original Mip-NeRF implementation. The λ parameter is used to define how significant of an influence the coarse sampling loss function should be. It is set to 0.1 in the original Mip-NeRF implementation. Finally, the α parameter, set to 0.01 in the original Mip-NeRF implementation, is used to fine-tune the 2-tap max filter to ensure that samples will appear even in empty regions of space [2].

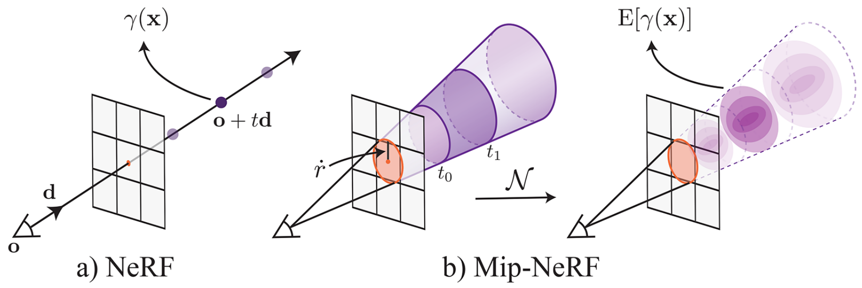


Figure 3.2: Rays in NeRF and mip-NeRF, from [2]

3.3 Mip-NeRF 360

Mip-NeRF 360 is a direct extension of mip-NeRF, which improves its ability to render the scene from all possible viewpoints while also improving the quality of objects that are very far or very close to the camera.

While mip-NeRF could render the scene from all sides, this only worked well for synthetic images that typically had no background. Scenes created from real-world images are typically "forward-facing," meaning they are only rendered from a fraction of the view space. Mip-NeRF 360 combines these two approaches by normalizing the scene uniformly in every direction. The way that mip-NeRF 360 does this is by converting the scene into a sphere with a radius of 2, which has two layers. The inner layer, which is less than one meter away from the center of the sphere, is scaled normally, while the outer layer is scaled proportionally to the inverse of the distance. Formally this conversion can be written using a contract function

$$\operatorname{contract}(\mathbf{x}) = \begin{cases} \mathbf{x} & \|\mathbf{x}\| \leq 1 \\ \left(2 - \frac{1}{\|\mathbf{x}\|}\right)\left(\frac{\mathbf{x}}{\|\mathbf{x}\|}\right) & \|\mathbf{x}\| > 1 \end{cases} \tag{3.36}$$

This contraction is described in Figure 3.3.

Mip-NeRF 360 presents two new loss functions and a different ray parametrization, allowing for better 360-degree scene rendering. Instead of coarse and fine sampling, used in the mip-NeRF method, mip-NeRF 360 uses two MLPs. The first MLP is called the proposal MLP and only predicts the volumetric

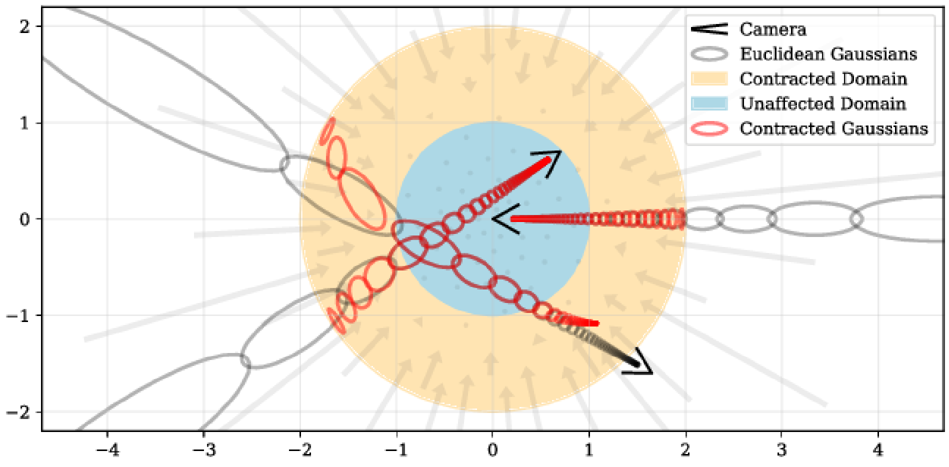


Figure 3.3: Scene normalization in mip-NeRF 360, from [6]

density, which we will denote $\hat{\tau}$. These predicted weights are then used to sample intervals for the NeRF MLP, which is much larger than the proposal MLP but is run much less often. Thanks to this approach, predicting correct colors does not take many iterations once correct densities are estimated. The NeRF MLP in Mip-NeRF 360 has a capacity that is over 20 times larger than the MLP used in mip-NeRF. However, higher MLP capacity comes with the disadvantage of higher memory requirements.

In order to train to make the weights of both MLPs consistent with each other, we need to compare their histograms of ray distances (in the sense of the weighted sum of samples for each bin). To do that, the following loss function is used

$$\mathcal{L}_{\text{prop}}(\mathbf{t}, \mathbf{w}, \hat{\mathbf{t}}, \hat{\mathbf{w}}) = \sum_i \frac{1}{w_i} \max(0, w_i - \text{bound}(\hat{\mathbf{t}}, \hat{\mathbf{w}}, T_i)) \tag{3.37}$$

$$\text{bound}(\hat{\mathbf{t}}, \hat{\mathbf{w}}, T) = \sum_{j: T \cap \hat{T}_j \neq \emptyset} \hat{w_j} \tag{3.38}$$

where T is an arbitrary interval, T_i is an i-th interval on a ray, $\mathbf{t}$ are the sampled distances used for the proposal MLP, $\mathbf{w}$ are the alpha compositing weights generated by the proposal MLP, $\hat{\mathbf{t}}$ are the sampled distances used for the NeRF MLP, and $\hat{\mathbf{w}}$ are the alpha compositing weights generated by the NeRF MLP.

Most NeRF methods suffer from two types of artifacts, floaters and background collapse. Floaters are high-density geometry that is not present in the original scene and is instead a result of trying to recover the scene from a given viewpoint, which becomes visible after attempting to render unseen viewpoints. *Background collapse* is a phenomenon that occurs when surfaces that are far away from the camera are instead rendered as semi-transparent clouds. To solve this issue, Mip-NeRF 360 proposes a loss called distortion loss, which is generally defined as the integral of interval size multiplied by weights at endpoints of this interval over all possible intervals. This loss function uses normalized distances s instead of t. Because we integrate interval lengths, this loss function prefers scenes with geometry in very thin dense intervals. The following formula defines how the distortion loss is used

$$\mathcal{L}_{\text{dist}}(\mathbf{s}, \mathbf{w}) = \sum_{i,j} w_i w_j \left| \frac{s_i + s_{i+1}}{2} - \frac{s_j + s_{j+1}}{2} \right| + \frac{1}{3} \sum_i w_i^2 (s_{i+1} - s_i) \quad (3.39)$$

where $\mathbf{w}$ are alpha compositing weights assigned to each point by the NeRF MLP and $\mathbf{s}$ are the normalized distances along a ray.

The only hyperparameter used in Mip-NeRF 360 is λ which determines the weight of the distortion loss function. It is initially set to 0.01.

3.4 DS-NeRF

NeRF-based methods require camera poses in order to create a correct 3D scene representation. These camera poses are usually derived using methods that generate sparse point clouds like COLMAP, meaning we usually have information about the original depth. Similarly, we can use information about the depth we would acquire differently (like depth cameras). The topic of depth supervision using dense depth data will be discussed later in the paper, where we discuss utilizing parts of this method while using depth images for supervision.

The crucial part of the *DS-NeRF* method is depth supervision which forces the probability distribution of ray termination predicted by the MLP to equal the depth distribution received as input. In order to do this, we will use several keypoints indexed by the letter i. using a distance of the point from a given viewpoint and a reprojection error of the keypoint, we can define a normal distribution that models the distribution of the keypoint depth. Because we now have two probability distributions, we can define a loss function that minimizes the Kullback-Leibler divergence between these two distributions. For best results, we will also need to have a different input depth variance value for each camera (which is omitted in the original NeRF paper). We will therefore denote this variance as σ_{ij}^2. The depth loss function looks like this

$$\mathcal{L}_{depth} \approx \mathbb{E}_{x_i \in X_j} - \sum_k \log(h_k) \exp\left(-\frac{(t_k - \mathbf{D}_{ij})^2}{2\hat{\sigma}_{ij}^2}\right) \Delta t_k \quad (3.40)$$

where i is an index of the selected keypoint, j is an index of a camera, k is an index of a point sampled on a ray, t_k is a distance of this point from the camera, h_k is a value of ray termination probability distribution for this point, Δt_k is a distance of this point from the next sample along a ray, $\mathbf{D}_{ij}$ is a distance of a keypoint $\mathbf{x}_i$ from a camera i, and $\hat{\sigma}_{ij}$ is a reprojection error of a keypoint $\mathbf{x}_i$ with respect to a camera with an index j.

Since the loss function is computed for each cone independently, the first two indices are omitted. It is essential to mention that measured depth variance σ_{ij}^2 was not view-dependent in the original paper since it was represented using COLMAP reprojection loss, which is the main reason why it is denoted as σ_i^2, as it does not depend on the camera index.

One key difference from the NeRF paper is the way that DS-NeRF computes weights used for rendering (according to the paper). These same weights are used to compute the depth supervision loss function

$$h(t) = \exp\left(-\int_0^t \sigma(t')dt'\right)\sigma(t). \tag{3.41}$$

This weight is also proven to be a probability distribution, specifically the probability of a ray stopping at a distance t. However, this is different from weights proposed by the NeRF paper, which have been since used in the majority of NeRF-based methods

$$w(t) = \exp\left(-\int_0^t \sigma(s)ds'\right)(1 - \exp(-\sigma(t)))$$

where $\sigma(t)$ is a volumetric density for a point sampled at distance t. The expected color of camera ray $\mathbf{r}$ is

$$\hat{\mathbf{C}}(\mathbf{r}) = \int_0^\infty h(t)\mathbf{c}(t)dt. \tag{3.42}$$

However, integrals cannot be used to compute the final color, as there is only a discrete set of samples that can be used. The rendering equation is therefore approximated using a sum

$$\hat{\mathbf{C}}(\mathbf{r}) = \sum_{i=1}^{N} h_i\mathbf{c}_i = \sum_{i=1}^{N} \exp\left(\sum_{j=1}^{i-1} \sigma_j\delta_j\right)\sigma_i\mathbf{c}_i \tag{3.43}$$

where N is the number of samples at the ray $\mathbf{r}$, $\delta_j = t_{j+1} - t_j$ is the distance between adjacent samples, σ_i is the predicted volumetric density of an i-th sample, and $\mathbf{c}_i \in \mathbb{R}$ is the predicted color of an i-th sample.

The only new hyperparameter that DS-NeRF uses is λ_d, which is used to decrease the weight of the depth supervision function. The original paper does not contain any information about its initial value, but in the DS-NeRF code, it is set to 0.1 [4].

3.4.1 Depth Supervision Loss Function

Aside from the Kullback-Leibler divergence-based loss function that DS-NeRF proposes, there is one more loss function that is briefly mentioned in the paper and used in the DS-NeRF source code. That is a loss that is based on the mean square error of depth for every ray that passes through a keypoint for a given viewpoint. This error is computed from the depth of the keypoint and the weighted mean of all samples on a given ray. According to the DS-NeRF source code, this loss function can be computed as

$$\mathcal{L}_{\text{MSE}} = \frac{1}{M}\sum^{M}(\mathbf{D}_i - \hat{\mathbf{D}}_i)^2 \tag{3.44}$$

$$\hat{\mathbf{D}}_i = \sum_{k=1}^{N_i} h_{ik} t_{ik} \tag{3.45}$$

where D_i is the target depth for a ray $\mathbf{r}_i$ which passes through a keypoint $\mathbf{x}_i$, M is the number of keypoints for a given viewpoint, N_i is the number of samples on ray $\mathbf{r}_i$, t_{ik} is the distance of a k-th sample along a ray $\mathbf{r}_i$, and h_{ik} is the weight of this sample. Index j is omitted from all of the variables [34].

Although the depth supervision loss function that DS-NeRF proposed is shown to produce better results than the MSE loss function [4], its definition is misleading. The proposed loss function is based on the Kullback-Leibler divergence between the rendered ray distribution $h(t)$ and the noisy depth distribution $\mathbb{N}(\mathbf{D}, \hat{\sigma})$. However, after rewriting the depth supervision loss by the definition of Kullback-Leibler divergence, the resulting loss function has a different definition

$$\mathcal{L}_{\text{depth}} = \sum_k \exp\left(-\frac{(t_k - \mathbf{D})^2}{2\hat{\sigma}^2}\right) \log\left(\frac{\exp\left(-\frac{(t_k-\mathbf{D})^2}{2\hat{\sigma}^2}\right)}{h_k}\right) \Delta t_k \tag{3.46}$$

$$\mathcal{L}_{\text{depth}} = \sum_k \left[-\log(h_k) - \frac{(t_k - \mathbf{D})^2}{2\hat{\sigma}^2}\right] \exp\left(-\frac{(t_k - \mathbf{D})^2}{2\hat{\sigma}^2}\right) \Delta t_k \tag{3.47}$$

Although it seems like the loss function would be the same in case the $-\frac{(t_k-D)^2}{2\hat{\sigma}^2}$ element was equal to zero, this detail is never mentioned in the paper, so it remains unclear if there was a mistake in the loss function derivation or if any mention of this omission was excluded from the paper altogether.

3.5 EfficientNeRF

EfficientNeRF is a method that improves the original NeRF method in terms of speed. This method reduces over 88% of training time and improves the rendering speed up to 200 FPS making it a reasonable choice for real-time applications. This is a significant improvement compared to other NeRF-based methods, which typically only render up to ten images per minute.

This method accelerates training using Valid Sampling (improves coarse sampling) and Pivotal Sampling (improves fine sampling). Compared to NeRF, it reduces the coarse MLP in size and uses the Spherical harmonics model from PlenOctrees to explicitly predict color parameters.

The main idea behind Valid Sampling is that once we find out that a sample $\mathbf{x}_i$ does not contribute to the final pixel color (its weight equals zero), we no

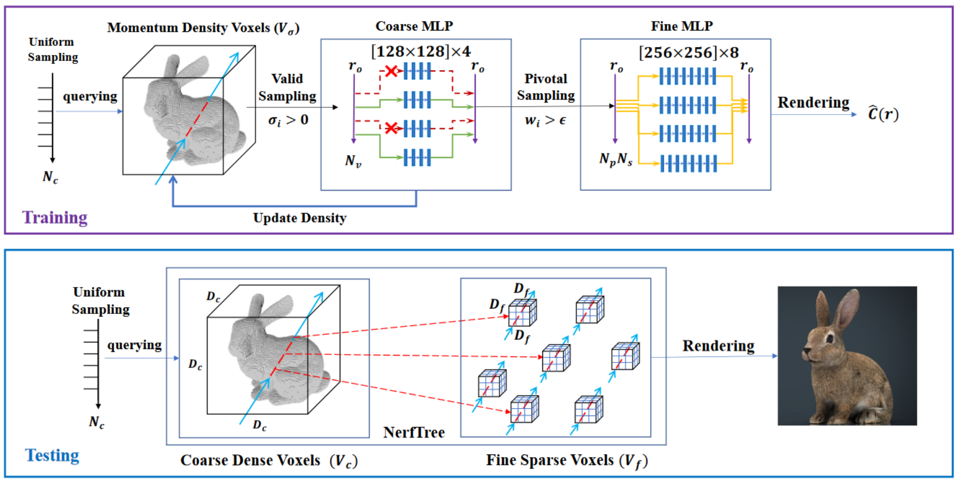


Figure 3.4: EfficientNeRF overview, from [5]

longer need to sample the area near this sample. To retain this information, EfficientNeRF proposes momentum density voxels that memorize the latest volume density values. These voxels are initially all assigned a small positive value. The corresponding voxel values are updated using volume density values received during coarse sampling

$$V_\sigma[\mathbf{i}] \leftarrow (1-\beta)V_\sigma[\mathbf{i}] + \beta\sigma_c(\mathbf{x}) \tag{3.48}$$

$$\mathbf{i} = \frac{\mathbf{x} - \mathbf{x}_{min}}{\mathbf{x}_{max} - \mathbf{x}_{min}} D \tag{3.49}$$

where $D \in \mathbb{N}$ is the resolution of the voxel grid, $\mathbf{x} \in \mathbb{R}^3$ is an arbitrary point in world coordinates, $\mathbf{i} \in \mathbb{R}^3$ is an index of a voxel that will be assigned to the point $\mathbf{x}$ and $\mathbf{x}_{min}, \mathbf{x}_{max} \in \mathbb{R}^3$ are the minimal and maximal world coordinate borders of the scene.

The Valid Sampling strategy is similar to the original coarse sampling strategy with the difference that all samples whose corresponding voxel has a non-positive value are immediately skipped.

As the name suggests, Pivotal Sampling uses pivotal samples, which are points with non-zero weight. After we pick a pivotal sample, we sample uniformly around this sample.

To improve the image rendering time, EfficientNeRF introduces a *NerfTree*, a data structure represented by a 2-depth tree, where the first depth caches the *coarse dense voxels* and the second depth caches the *fine sparse voxels*. While the coarse voxels only contain a density attribute, the fine voxels also contain color parameters. The voxel values are directly inferred from both coarse and fine MLPs (coarse voxels use the coarse MLP values, and fine voxels use the fine MLP values) [5]. The structure of NeRFTree is depicted in Figure 3.5.

3.6 Ref-NeRF

Ref-NeRF improves on Mip-NeRF and considers reflections across the surface normals instead of just using the view direction as a parameter. To improve the reflection accuracy, Ref-NeRF encodes the reflected direction using spherical harmonics.

The first part of the method is a spatial MLP which only takes in the position $\boldsymbol{x}$ as a parameter. The outputs of this MLP are volumetric density σ, diffuse color $\mathbf{c}_d$, specular tint s, surface roughness ρ, surface normal $\hat{\mathbf{n}}'$, and finally, a bottleneck vector $\mathbf{b}$. We can also compute predicted volume density to compute the surface normal directly using this equation

$$\hat{\boldsymbol{n}}(\boldsymbol{x}) = -\frac{\nabla\sigma(\boldsymbol{x})}{\|\nabla\sigma(\boldsymbol{x})\|} \tag{3.50}$$

Using the surface normal predicted by the spatial MLP and using the view direction received as input, we can compute the reflection direction

$$\hat{\boldsymbol{\omega}}_r = \mathbf{d} - 2(\mathbf{d} \cdot \hat{\mathbf{n}}')\hat{\mathbf{n}}' \tag{3.51}$$

where $\mathbf{d}$ is the direction vector.

To encode the reflection vector, we also need to use roughness predicted by the spatial MLP, more specifically its inverse, which we will call concentration κ. Following equations describe how is integrated directional encoding calculated

$$\mathrm{IDE}(\hat{\boldsymbol{\omega}}_r, \kappa) = \{A_\ell(\kappa)Y_\ell^m(\hat{\boldsymbol{\omega}}_r) : (\ell, m) \in \mathcal{M}_L\} \tag{3.52}$$

$$\mathcal{M}_L = \{(\ell, m) : \ell = 1, ..., 2^L, m = 0, ..., \ell\} \tag{3.53}$$

$$A_\ell(\kappa) \approx \exp\left(-\frac{\ell(\ell+1)}{2\kappa}\right) \tag{3.54}$$

where A_ℓ is the ℓ-th attenuation function and Y_ℓ^m is a Lapace's spherical harmonic.

Color in Ref-NeRF isn't directly acquired from an MLP. Instead, it is acquired by combining its diffuse and specular components. The diffuse color is already one of the outputs of the spatial MLP. However, since the specular color is view-dependent, we need to create one more MLP to predict the specular color component and multiply it with a specular tint (already predicted using the spatial MLP).

This new MLP, which we will call directional MLP, takes in the encoded reflection direction but also the bottleneck vector $\boldsymbol{b}$ and also the dot product of predicted normal and view direction to achieve better results. After receiving the specular color, we can finally compute the final pixel color

$$\boldsymbol{c} = \gamma(\boldsymbol{c}_d + \boldsymbol{s} \circ \boldsymbol{c}_s) \tag{3.55}$$

where γ is a fixed tone mapping function that converts linear color to sRGB.

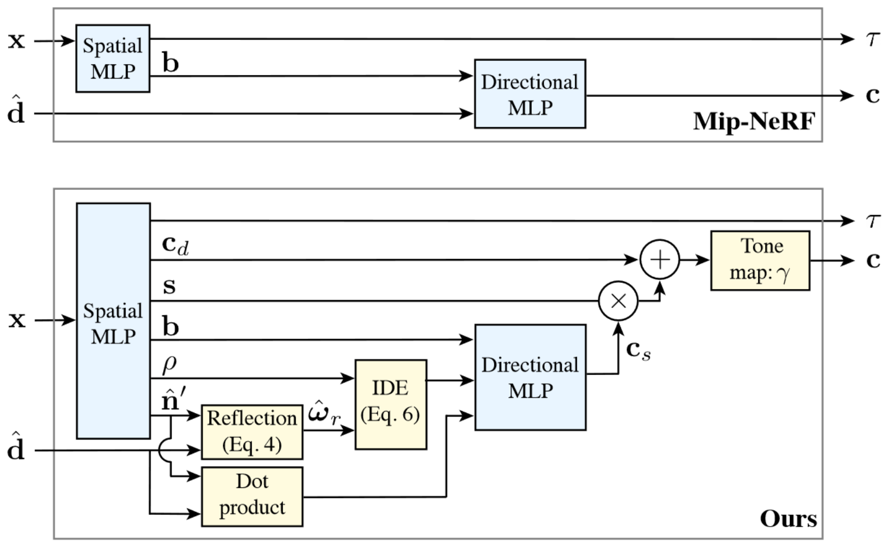


Figure 3.5: The comparison of mip-NeRF and Ref-NeRF architectures (Ref-NeRF paper used τ for the volumetric density), from [3]

To further improve the quality of Ref-NeRF renderings, two new loss functions are introduced. The first one improves the accuracy of surface normal predictions for each ray (cone)

$$L_{normal}(\boldsymbol{r}_i) = \sum_j w_{ij} \|\hat{\boldsymbol{n}}_j - \hat{\boldsymbol{n}}'_j\|^2 \tag{3.56}$$

The second loss function aims to minimize the volume density inside of objects which leads to the foggy appearance of specular surfaces. The way it is done is by penalizing normals that are oriented away from the camera

$$L_{orientation}(\boldsymbol{r}_i) = \sum_j w_{ij} \max(0, \hat{\boldsymbol{n}}'_j \cdot \hat{\boldsymbol{d}})^2 \tag{3.57}$$

Ref-NeRF does not define any additional hyperparameters [3].

Chapter 4

Experiments

The previous chapter discussed the theory behind several NeRF-based methods released in recent years. This chapter will describe how some of these methods perform on two real-life datasets and one synthetic dataset. The last part of this chapter describes how additional depth information can be utilized to improve these methods and show the results when using depth supervision.

4.1 Dataset Details

The experiments performed during this work are evaluated on three different datasets. The first dataset is a synthetic dataset created using Blender, and the other two were created from real-world images with depth maps captured using iPad. All three datasets can be found at the following link: `https://drive.google.com/drive/folders/1CvObrtawHvlLIk1IRxcQiaQAqaa1BW9_?usp=share_link`. An overview of these datasets is shown in Table 4.1.

Dataset name	Dataset type	Training viewpoints	Rendering viewpoints	Unbounded
construction_site_small2	LLFF	10	2	Yes
construction_site_normal	LLFF	20	3	Yes
ship_ds	Synthetic	20	200	No

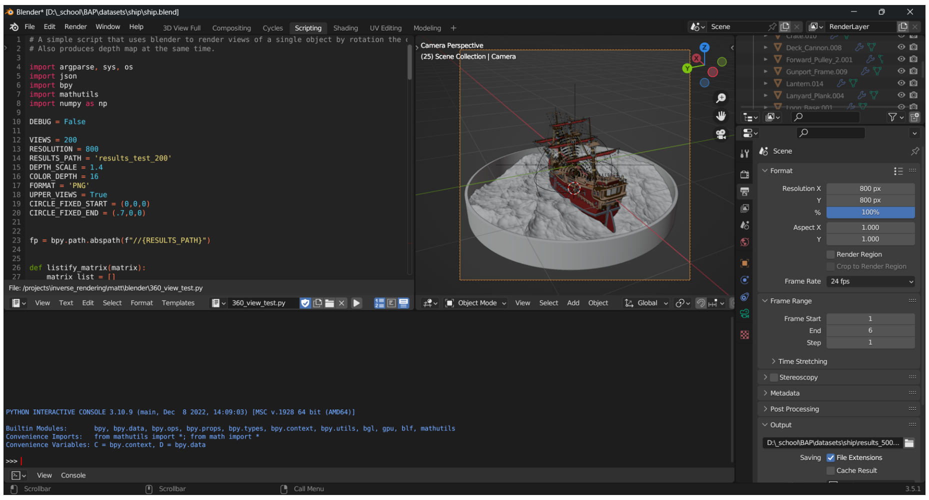
Table 4.1: Overview of used datasets

4.1.1 Synthetic Dataset

The synthetic Blender dataset used for this thesis was created using a model of a ship named "Suzanne’s Revenge," modeled by Chris Kuhn and textured by Greg Zaal. The model was posted on the website blenderartists.org in 2013 [20]. It later appeared as one of the synthetic datasets used to test the performance of the original NeRF method.

Because the dataset was not designed for depth supervision, it had to be recreated using the source Blender project to render new viewpoints along with the corresponding depth data. This source project was available in the data section of the Neural Radiance Fields website, and a screenshot of the project is shown in Figure 4.1. However, since textures were unavailable in this version, they had to be acquired from the original 2013 post. The additional problem was that the environment map, a 360-degree image used to compute realistic reflections, that was used for the synthetic dataset could not be found in the original post from 2013, so it had to be replaced by a different environment map that was used in the original project. This is why the depth supervision experiments performed on the synthetic dataset have different lighting conditions than those used for mip-NeRF and mip-NeRF 360.

The Blender project published by the NeRF authors differed from the original one since it contained several Python scripts used to render images for the dataset. Fortunately, rendering depth images was not a difficult task as the script could enable depth rendering and set up the linear mapping for the depth images. All values were mapped from range $[0, 8]$ to range $[0, 1]$. This mapping was utilized because all values that would exceed one in the rendered images were automatically clamped to one. In order to avoid aliasing of the depth values, the bit depth of a single color channel was increased from 8 to 16. This option was enabled for both depth images and RGBA images. After the depth images were rendered, they were rescaled by multiplying all values by eight, reverting the normalization to range of $[0, 1]$, and converted to the NumPy array format.


Blender* [D:_school\BAP\datasets\ship\ship.blend]
File Edit Render Window Help
3D View Full Compositing Cycles Scripting Shading UV Editing Modeling
Scene
RenderLayer
Camera Perspective
(25) Scene Collection | Camera
Deck_Cannon.008
Forward_Pulley_2.001
Gunport_Frame.009
Lantern.014
Lanyard_Plank.004
View Text Edit Select Format Templates
360_view_test.py
Object Mode View Select Add Object Global
View Console
Scrollbar
Scrollbar
Call Menu
3.5.1

```python
# A simple script that uses blender to render views of a single object by rotation the 
# Also produces depth map at the same time.

import argparse, sys, os
import json
import bpy
import mathutils
import numpy as np

DEBUG = False

VIEWS = 200
RESOLUTION = 800
RESULTS_PATH = 'results_test_200'
DEPTH_SCALE = 1.4
COLOR_DEPTH = 16
FORMAT = 'PNG'
UPPER_VIEWS = True
CIRCLE_FIXED_START = (0,0,0)
CIRCLE_FIXED_END = (.7,0,0)


fp = bpy.path.abspath(f"//{RESULTS_PATH}")


def listify_matrix(matrix):
    matrix_list = []
```

File: /projects\inverse_rendering\matt\blender\360_view_test.py

```
PYTHON INTERACTIVE CONSOLE 3.10.9 (main, Dec  8 2022, 14:09:03) [MSC v.1928 64 bit (AMD64)]

Builtin Modules:       bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context, bpy.utils, bgl, gpu, blf, mathutils
Convenience Imports:   from mathutils import *; from math import *
Convenience Variables: C = bpy.context, D = bpy.data

>>> 
```

Scene

Format

Resolution X	800 px
Y	800 px
%	100%
Aspect X	1.000
Y	1.000
	☐ Render Region
	☐ Crop to Render Region
Frame Rate	24 fps

Frame Range

Frame Start	1
End	6
Step	1

Time Stretching

☐ Stereoscopy

Metadata

Post Processing

Output

D:_school\BAP\datasets\ship\results_500...

Saving	☑ File Extensions
	☐ Cache Result

Figure 4.1: Blender project used to generate synthetic images

One problem that appeared during rendering was that all RGB pictures contained sharp light patches with a size of one pixel, likely due to aliasing. This issue was solved by enabling denoising for the Cycles renderer. The same correction was not done for the depth images to avoid using blurry data for the supervision. The number of samples was set to 512 to achieve feasible results.

The number of images that have been used for experiments has been altered as well. While the original dataset used 500 input images for training, the dataset used for our experiments contained only 20 training images to make results more comparable to results that have been evaluated for other datasets. Similar to the original dataset, our dataset used 200 viewpoints for evaluation and rendering.

4.1.2 LLFF Datasets

The two remaining datasets, which were created using real-world images, capture a construction site, and originate from a larger dataset that was initially created for the ARTWIN project, which focuses on augmented reality applications [21]. The dataset also originally contained camera poses acquired from real-time tracking of the iPad Pro 2022, but these poses were not accurate. In order to improve the dataset, the camera poses have been refined using COLMAP. While the dataset was initially composed of 194 images, it was reduced to 20 images because the original scene was so large that none of the present NeRF-based methods would be able to capture it. In order to test how different methods perform on datasets with a smaller number of input images, another dataset with 10 input images was created as well. Since these two datasets are photos of the same scene and differ only in the number of input images, they will be referred to as *the large LLFF dataset* and *the small LLFF dataset*, respectively.

Unlike the Blender dataset, this one comes with readily available depth images. The depth information was acquired by fusing iPad lidar scans with a multi-view stereo approximation of the depth images. These depth images, originally in the Matlab matrix format, were converted to NumPy array format for convenience using a simple Python script.

Most NeRF-based methods use the term LLFF to describe datasets realized as a COLMAP model. This name was used because an older novel view synthesis method called Local Light Field Fusion (LLFF for short) used the same format for its datasets. This dataset also contains a file called poses_bounds.npy, which contains an array of size Nx17, where N is the number of input images. Each line contains a flattened 3x5 pose matrix followed by two values representing the nearest and furthest point of the scene content from that point of view. The pose matrix can be defined as

$$M_{\text{pose}} = \begin{bmatrix} M_{\text{camtoworld}} & \mathbf{i} \end{bmatrix} \tag{4.1}$$

$$\mathbf{i} = \begin{bmatrix} v \\ u \\ f \end{bmatrix} \tag{4.2}$$

where v is the image height, u is the image width, f is the focal length, and $M_{\text{camtoworld}} \in \mathbb{R}^{3\times 4}$ is a camera-to-world matrix.

is a camera-to-world matrix 3x4, concatenated with a 3x1 vector containing image height, width, and focal length [24]. Because this array was not present in the original dataset, it was created using a function used in the LLFF method. This array is only used to render forward-facing scenes with a spiral path and serves no purpose in the training process.

One issue with the training data was that the original images had slight barrel distortion, which had to be corrected. However, after this undistortion was applied, several pixels in the images were black, which initially produced results that contained sharp black strips. In order to get rid of this issue, only pixels more than 10 pixels away from the border were used for training, and other pixels were discarded.

Because the code that was used for experiments did not support using synthetic blender datasets to realize the scenes in normalized device coordinates, the synthetic ship dataset was also converted to the LLFF format using COLMAP because it is later used for experiments that would involve converting the 360-degree scene to normalized device coordinates.

4.2 Implementation details

We implemented the code used for the experiments shown in this thesis in the repository called OptiNeRF. This is a modified version of the MultiNeRF repository, which Google Research published as a code release for the mip-NeRF 360, Ref-NeRF, and RawNeRF methods [22]. The main contribution of the OptiNeRF code is incorporating several depth supervision loss functions into the MultiNeRF implementation of the mip-NeRF method. The OptiNeRF repository can be found on the following link: `https://github.com/JakubSakar9/optinerf`.

Since the MultiNeRF implementation allows for saving the state of the training process before it is finished, we are easily able to see the results after an arbitrary number of training iterations. The file that stores the information is called a *checkpoint*. The number of training iterations that pass between each checkpoint can be configured. It was set to 5000 for the mip-NeRF experiments and 10000 for the mip-NeRF 360 experiments.

4.3 Testing existing methods

The following section focuses on testing the MultiNeRF implementation of two methods, mip-NeRF and mip-NeRF 360.

All methods used GPU to accelerate training and have been trained on a computer with the Nvidia RTX 3090 graphics card. However, this hardware still caused some limitations as the hardware that Google Research originally used was more powerful. The main limitation was GPU memory, which was shown to be insufficient for the training with default parameters. Due to these limitations, the batch size for each experiment had to be decreased. The synthetic Blender dataset was only used to test the mip-NeRF methods because mip-NeRF 360 is mainly used for unbounded scenes. An overview of conducted experiments is shown in Table 4.2.

	Large LLFF		Small LLFF		Blender
	NDC	No NDC	NDC	No NDC	
mip-NeRF	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
mip-NeRF 360	**Yes**	**Yes**	**Yes**	**Yes**	No

Table 4.2: Summary of experiments regarding existing methods. The table provides an overview of whether each experiment has been conducted.

4.3.1 Mip-NeRF

The mip-NeRF method was tested on all three datasets that were presented in section 4.1. using the train_llff.sh script while altering the parameters in the llff_256.gin configuration file. Experiments performed on the synthetic dataset were run using the train_blender.sh file while using the parameters in the blender_256.gin configuration file. Unlike the original mip-NeRF implementation, the MultiNeRF implementation performs the conversion to normalized device coordinates for forward-facing scenes and uses two MLPs instead of a single one. In this thesis, the LLFF dataset was tested both with and without the NDC conversion.

The two MLPs that are used for training are called proposal MLP and NeRF MLP, as mentioned in the mip-NeRF 360 paper. Both MLPs have a width of 256, and their number of layers is 4 for the proposal MLP and 8 for the NeRF MLP. While the original NeRF method used no activation function before outputting the prediction of volumetric density σ, the MultiNeRF implementation uses the Softplus activation function. However, color and density use a ReLU activation function for all the hidden layers, and the output layer for color uses the sigmoid activation function. Similarly, the implementation adds a skip connection at the fourth layer and uses an additional layer with a depth of 128 for color prediction.

The number of training iterations was reduced from the original 250000 to 100000 as the improvement of render quality was very small for the higher number of iterations. As mentioned in the previous section, the batch size also had to be decreased due to the memory limitations of the used hardware. It decreased from 16384 to 8192. In order to keep the training process consistent with the one that would be used initially, both initial and final values of the learning rate had to be halved as well. The initial value of the learning rate was set to 0.01, and the final value was set to 0.00001. This adjustment was made following the recommendation in the readme of the MultiNeRF repository [22]. As a result, the training process duration for each experiment averaged around 5 hours.

Results

The following tables show mip-NeRF renderings of both used LLFF datasets in comparison to the ground truth. These results will be later used for comparison to the modified version of mip-NeRF, which utilizes additional information about depth. Every even line in all of these tables shows a cropped section of an image which is used to demonstrate the ability of mip-NeRF to reconstruct fine details. All the renderings in this section have been created after 100000 iterations of the training process. For additional results on the performance of these methods with fewer training iterations, please refer to the appendix.

In order to make the rendered depth images comparable to the measured depth, all depth images, which contained metric depths, were normalized between zero and one. The following equation can describe this transformation

$$D' = 1 - \frac{1}{1 + D} \tag{4.3}$$

where D is a metric depth and D' is a depth that was used for visualization.

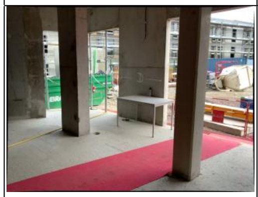
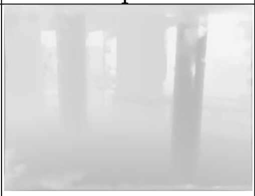
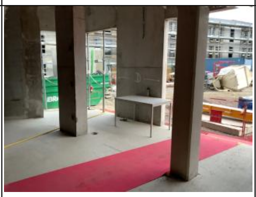
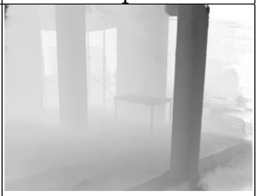
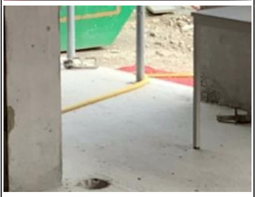
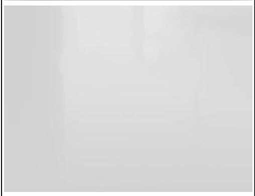
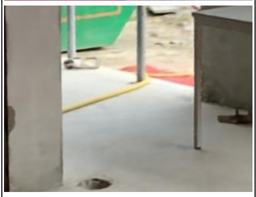
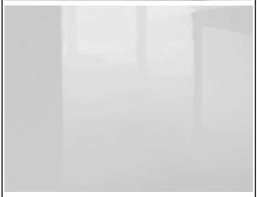
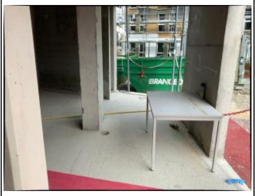
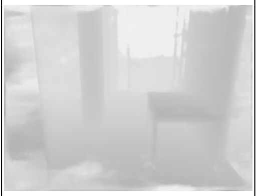
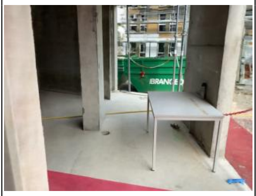
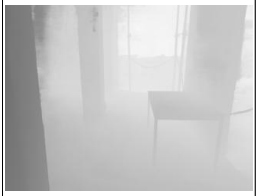
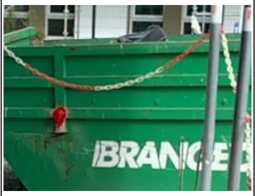
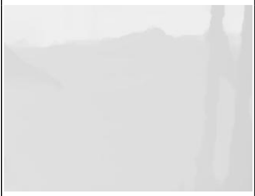
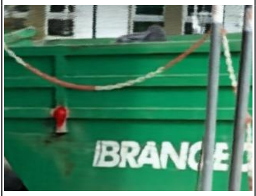
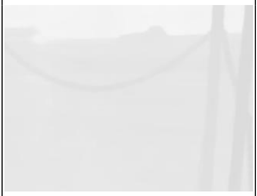
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table 4.3: Mip-NeRF renderings of the large LLFF dataset with the NDC scene normalization, 100000 iterations

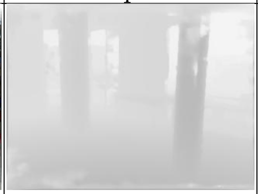
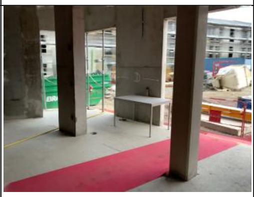
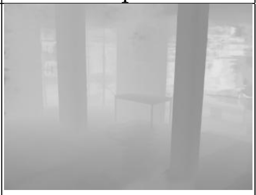
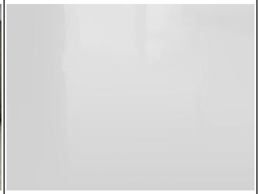
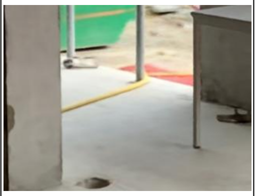
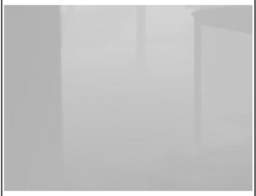
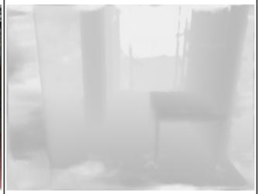
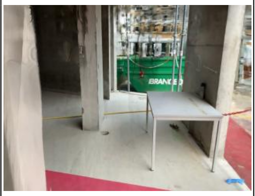
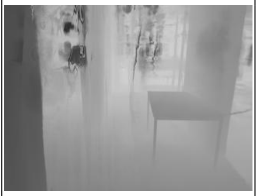
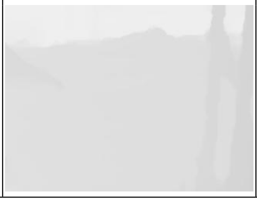
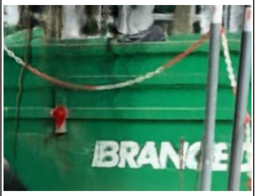
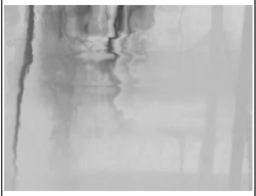
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table 4.4: Mip-NeRF renderings of the large LLFF dataset without the NDC scene normalization, 100000 iterations

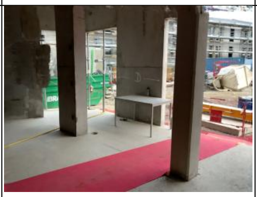
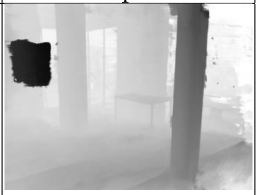
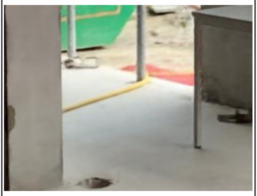
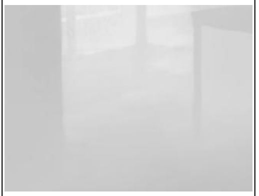
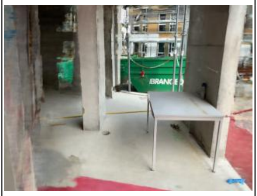
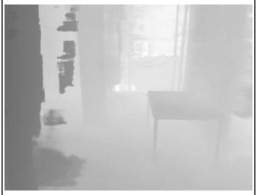
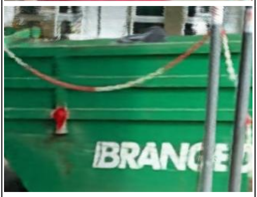
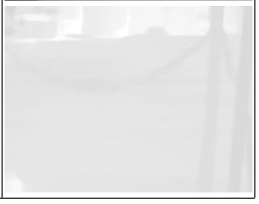
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table 4.5: Mip-NeRF renderings of the small LLFF dataset with the NDC scene normalization, 100000 iterations

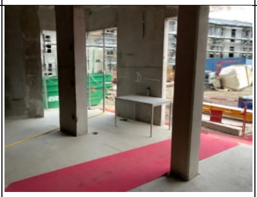
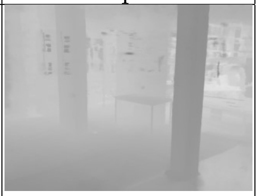
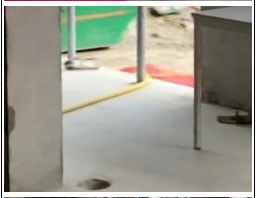
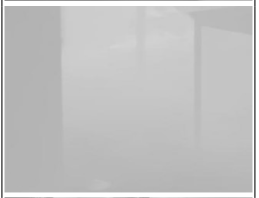
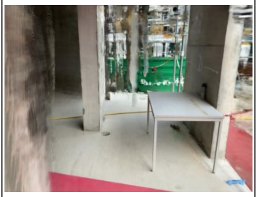
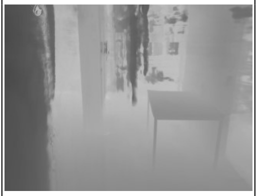
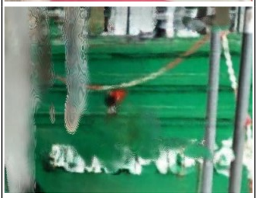
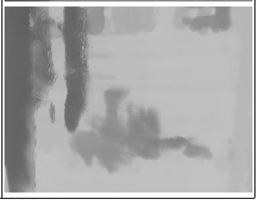
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table 4.6: Mip-NeRF renderings of the small LLFF dataset without the NDC scene normalization, 100000 iterations

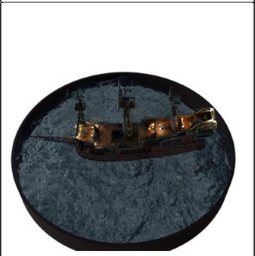
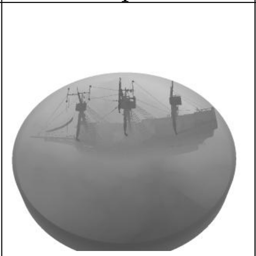
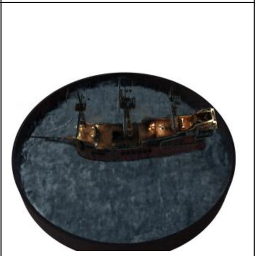
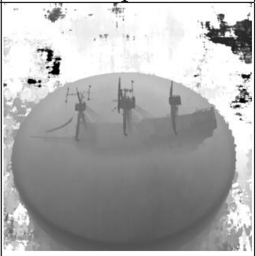
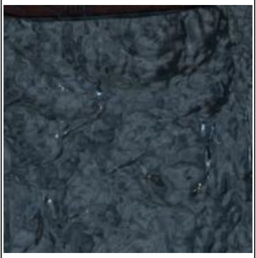
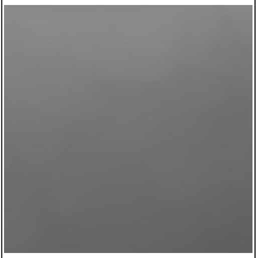
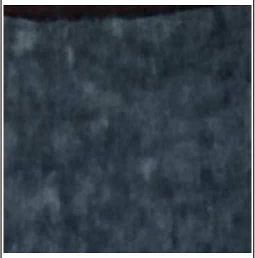
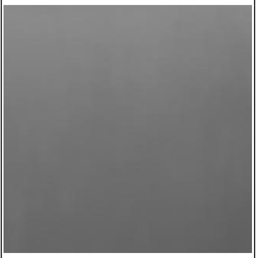
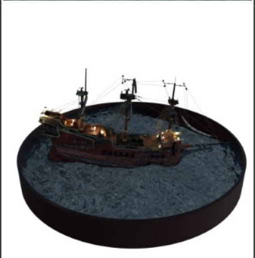
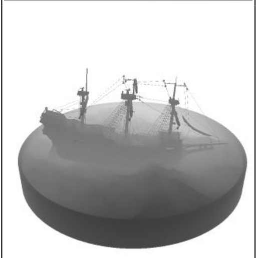
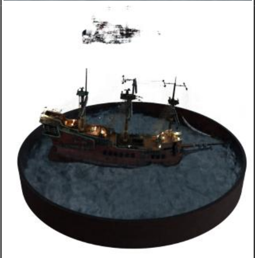
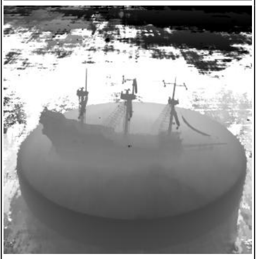
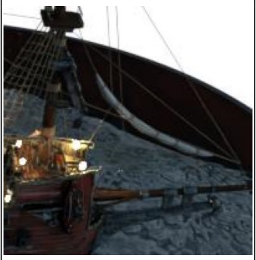
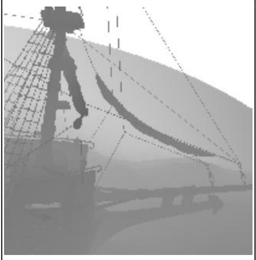
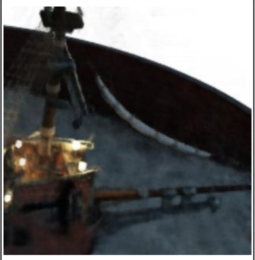
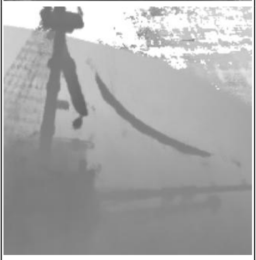
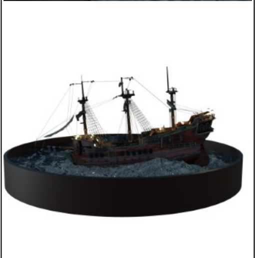
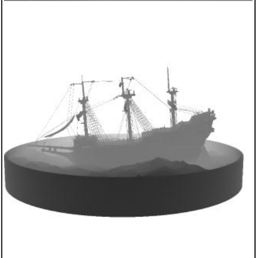
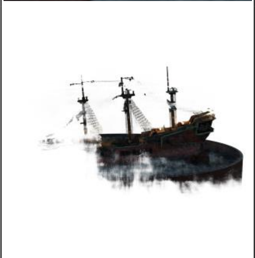
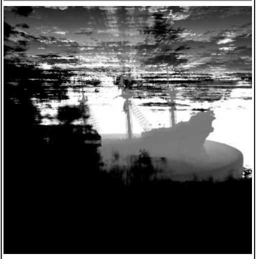
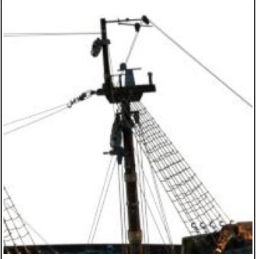
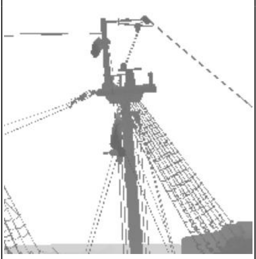
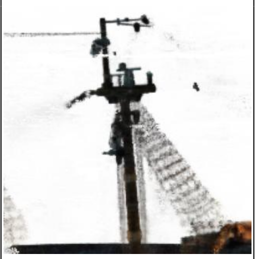
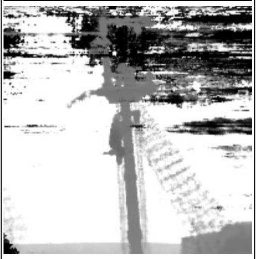
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table 4.7: Mip-NeRF renderings of the synthetic dataset without any depth supervision, 100000 iterations

As can be seen in Table 4.3, using mip-NeRF for the reconstruction of the LLFF dataset with the NDC scene normalization leads to correct results, which only fail to reconstruct fine details. Table 4.4 shows that rendering the scene without any normalization already leads to several artifacts, which can be seen for objects that are very close or very far from the camera.

Doing the same experiments for the small LLFF datasets leads to similar problems. The pillars that are present in the scene also start causing artifacts in the renders, as some areas of the scene are occluded in most input images. These experiments are shown in Tables 4.5 and 4.6.

Testing mip-NeRF on the synthetic dataset, which can be seen in the table 4.7, shows that the 3D scene is reconstructed incorrectly when given a low number of input images. This might be caused by a high dilation of input images.

4.3.2 Mip-NeRF 360

... The mip-NeRF 360 method was tested using the train_360.sh script while altering the parameters in the 360.gin configuration file. Because the MultiNeRF repository was the only official release of mip-NeRF 360 source code, the code run in the experiments is nearly identical to the code that produced the results presented in the mip-NeRF 360 paper. It was only tested on LLFF datasets as it is out of the scope of this thesis.

As mentioned in the section focusing on the theoretical background of mip-NeRF 360, two MLPs are used in the training process, proposal MLP and NeRF MLP. The proposal MLP has a depth of 4, a width of 256, and no skip layers. The NeRF MLP has a depth of 8, a width of 1024, and a skip layer at the fourth layer. The activation functions used for mip-NeRF 360 are the same as those used for mip-NeRF.

The number of training iterations decreased from 250000 to 200000 for the same reason as mip-NeRF. It was not decreased to 100000 again because mip-NeRF 360 uses more memory than mip-NeRF, which meant that the batch size had to be decreased by a more significant factor than the batch size of mip-NeRF experiments. Because the batch size for mip-NeRF 360 was 4096, half of what was set for mip-NeRF, the number of iterations had to be doubled to make up for slower learning. Both initial and final values of the learning rate have been halved as well, which means it was the initial value was set 0.005 while its final value was set to 0.000005. Given that the capacity of the NeRF MLP is double that of the MLPs utilized in mip-NeRF, and considering that the number of training iterations for mip-NeRF 360 experiments was also twice as high, it follows that the training time for mip-NeRF 360 experiments was four times longer compared to that of the mip-NeRF experiments.

Results

The following tables show mip-NeRF 360 renderings of both used LLFF datasets in comparison to the ground truth. These renderings do not involve depth maps because incorporating depth supervision into the mip-NeRF 360 is a considerably challenging task that will not be discussed in this thesis. Similarly to the mip-NeRF experiments, the renderings presented in this section are generated by the model after the training process has been completed. Intermediate results can be found in the appendix. However, because the number of iterations of the training process has been doubled compared to the mip-NeRF experiments, the respective numbers of training iterations for each of the used checkpoints have been doubled as well.

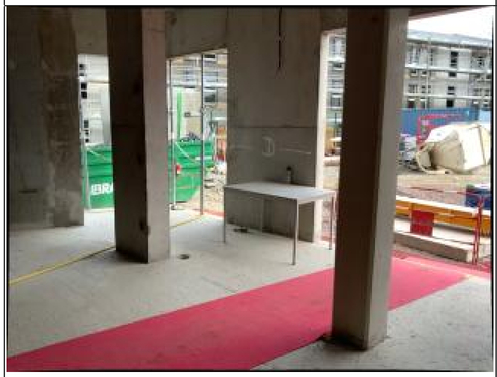
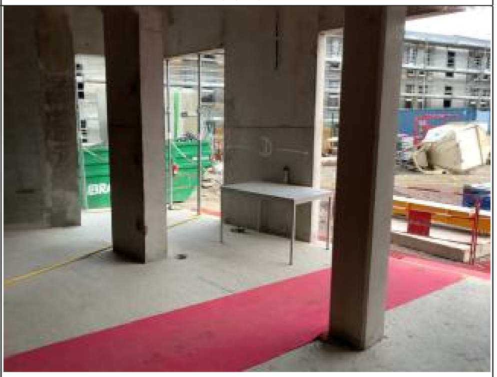
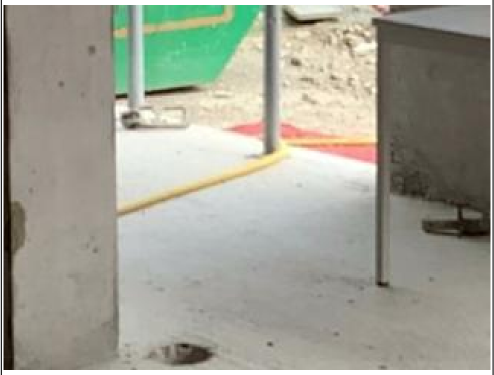
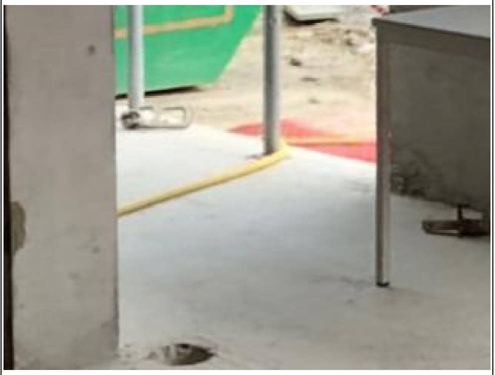
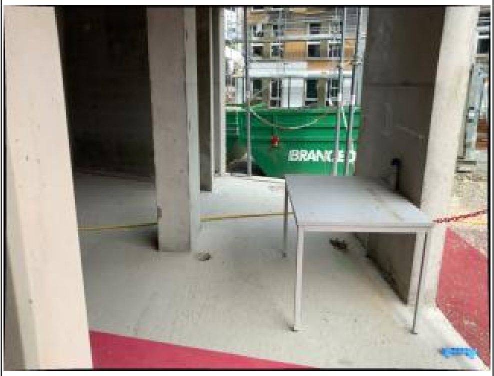
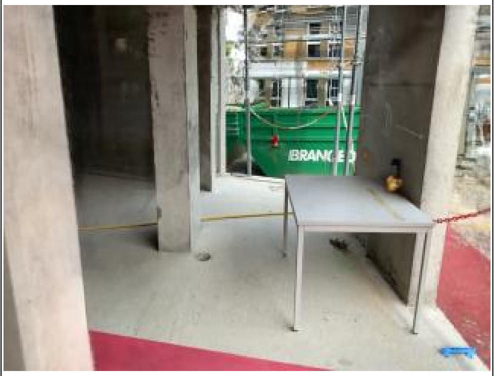
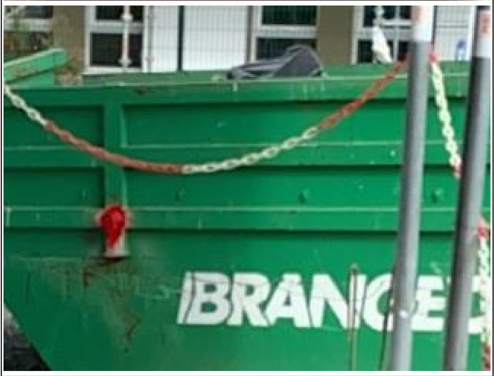
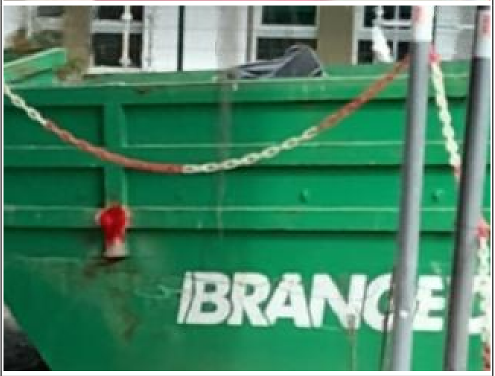
GT color	Rendered color
	
	
	
	

Table 4.8: Mip-NeRF 360 renderings of the large LLFF dataset, 200000 iterations

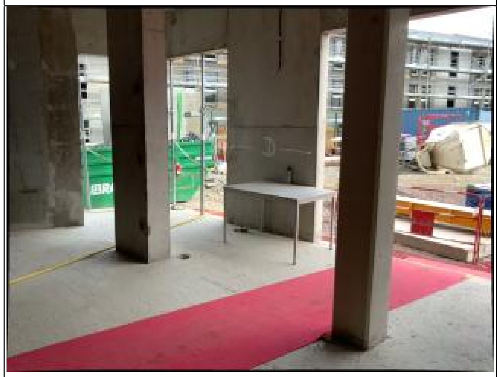
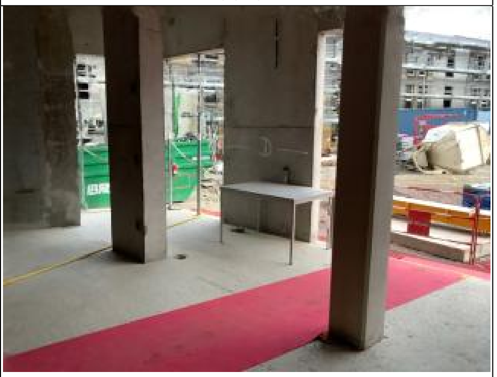
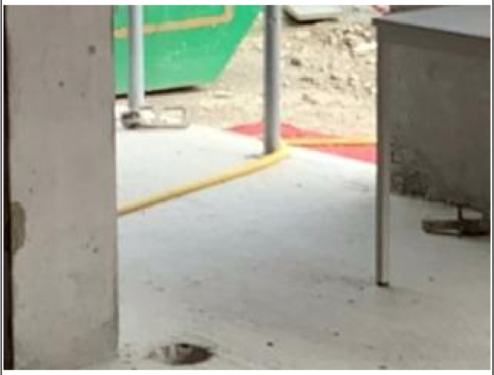
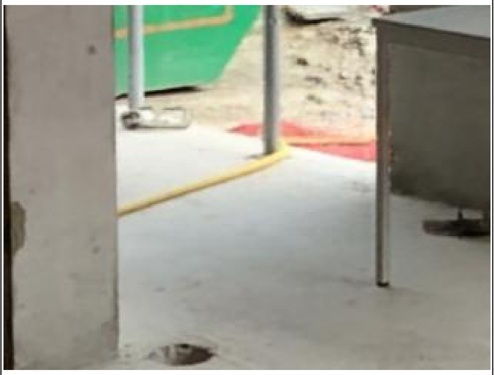
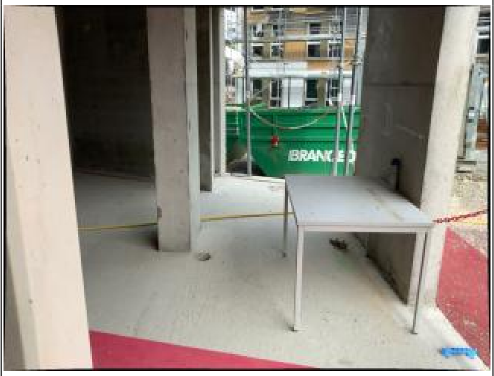
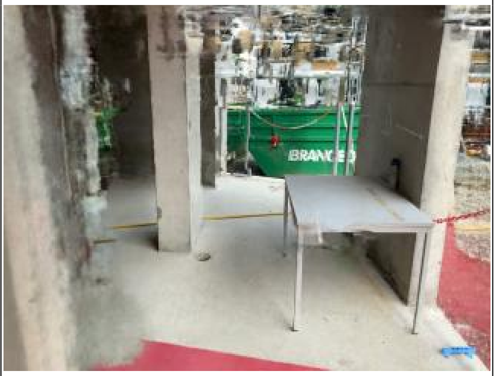
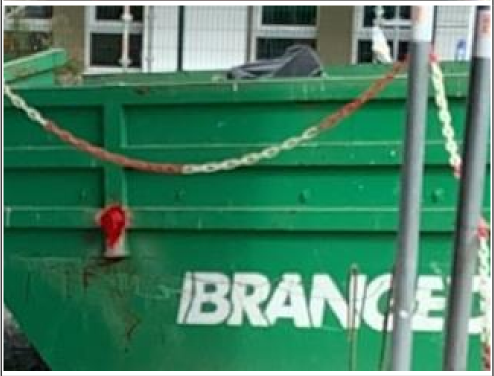
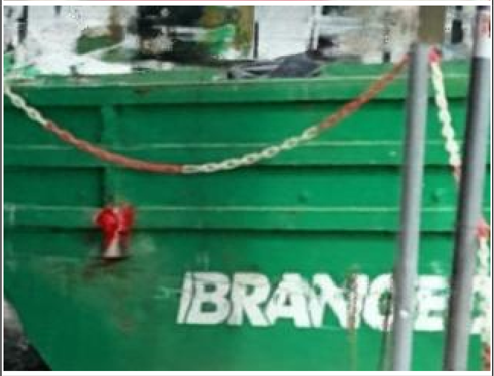
GT color	Rendered color
	
	
	
	

Table 4.9: Mip-NeRF 360 renderings of the small LLFF dataset, 200000 iterations

As well as mip-NeRF, mip-NeRF 360 managed to reconstruct the scene correctly for the large LLFF dataset (Table 4.8). Performing the same experiment for the small LLFF dataset resulted in similar problems that could be seen in Table 4.3, but managed to decrease the number of floaters in the images. Using mip-NeRF 360 for this type of scene has thus proven to be ineffective since the training time was four times longer than the training time of mip-NeRF experiments.

4.4 Depth supervision

The following section discusses how additional information about depth could be used to improve the performance of the mip-NeRF method and shows experiments performed on both LLFF datasets along with the synthetic dataset.

Although the DS-NeRF paper initially inspired the depth supervision process, the way depth data was utilized in performed experiments differs. DS-NeRF did not use any dense depth data and instead computed depths from a point cloud that was available after generating COLMAP models. DS-NeRF also used errors defined as reprojection errors of 3D points in COLMAP. This realization of errors is not straightforward to use when using depth images, but fortunately, the used dataset contains variance of depth data as well. An overview of experiments performed in this section can be seen in Table 4.10.

	Large LLFF		Small LLFF		Blender
	NDC	No NDC	NDC	No NDC	
DS NeRF loss					
$\lambda_d = 0.01,\ \xi_d = 1.000$	No	No	No	No	No
$\lambda_d = 0.10,\ \xi_d = 1.000$	No	No	No	No	**Yes**
$\lambda_d = 1.00,\ \xi_d = 1.000$	No	No	No	No	No
$\lambda_d = 1.00,\ \xi_d = 0.999$	No	No	No	No	No
KL loss					
$\lambda_d = 0.01,\ \xi_d = 1.000$	No	No	No	No	**Yes**
$\lambda_d = 0.10,\ \xi_d = 1.000$	No	No	No	No	**Yes**
$\lambda_d = 1.00,\ \xi_d = 1.000$	No	No	No	No	**Yes**
$\lambda_d = 1.00,\ \xi_d = 0.999$	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
MSE loss					
$\lambda_d = 0.01,\ \xi_d = 1.000$	No	No	No	No	No
$\lambda_d = 0.10,\ \xi_d = 1.000$	No	No	No	No	**Yes**
$\lambda_d = 1.00,\ \xi_d = 1.000$	No	No	No	No	No
$\lambda_d = 1.00,\ \xi_d = 0.999$	No	No	No	No	No

Table 4.10: Summary of depth-supervised experiments. The table provides an overview of whether each experiment has been conducted.

4.4.1 Implementation details

Implementing the depth supervision loss function into the MultiNeRF code required creating a new loss function and adding it to the total loss function if it was enabled in the configuration file. Because both depth images and depth variances were converted to NumPy arrays, the loading was done using the NumPy library.

Both depth images and depth variances also needed to be transformed for forward-facing experiments because these experiments use normalized device coordinates to sample the space. Because the conversion to normalized device coordinates makes the sampling parameter t linear in disparity, the depth images were also transformed to be linear in disparity using the following equation

$$D' = 1 - \frac{1}{D}. \tag{4.4}$$

where D is the original depth in world coordinates, and D' is the transformed depth in normalized device coordinates.

On the other hand, transforming the depth variances is not a straightforward process since they rely on the values present in the depth images. Therefore, they were adjusted in such a way that the square root of the transformed variances decreased by the same factor as the depth images.

As mentioned in section 3.4, DS-NeRF uses a parameter λ_d, which determines the weight of the depth supervision loss function. Because the main purpose of depth supervision is to speed up the training process in its early phase, the weight of the loss function could be decreased. In order to do that, we can use a *decay rate* ξ_d to exponentially decrease the depth supervision loss function weight. This decrease can be formulated using the following equation

$$\lambda_d(t) = \lambda_{d0}\xi_d^t \tag{4.5}$$

where $\lambda_d(t)$ is the depth supervision loss function multiplier at training step t and λ_{d0} is the initial value of this multiplier.

4.4.2 Synthetic Dataset

Before we attempt to apply depth supervision to real-life data, we verify its functionality on synthetic data. Therefore, the following section will start by showing how mip-NeRF performs on a synthetic dataset with a low number of input images and then demonstrate how different depth supervision loss functions and their parameters perform in improving these results.

To make the depth maps more feasible for visualization, we can transform them by normalizing the range between the near plane and the far plane and clamping everything that falls out of it to its boundaries. This can be done using the following equation

$$D' = \text{clamp}\left(\frac{D - t_n}{t_f - t_n}, 0, 1\right) \tag{4.6}$$

where t_n is the distance of the near clipping plane, t_f is the distance of the far clipping plane, D is a metric distance from the camera, and D' is a distance from the camera that is suitable for visualization. In the case of our experiments, $t_n = 2$ and $t_f = 6$.

Unlike the mip-NeRF and mip-NeRF 360 experiments sections, this section will focus on the performance of depth-supervised mip-NeRF after a low number of training iterations. In this case, the checkpoint with the lowest number of training iterations has been generated after 5000 iterations, which is the checkpoint that was used for these renderings. The experiments were run using the train_blender_ds.sh script while altering the parameters in the blender_ds.gin configuration file. Renderings that have been produced after a larger number of training iterations can be found in the appendix.

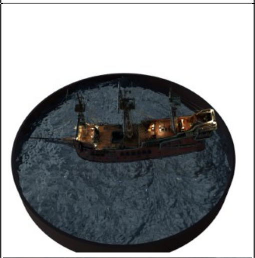
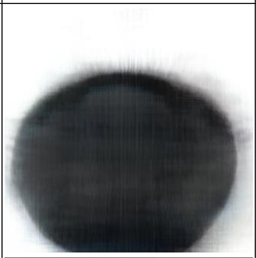
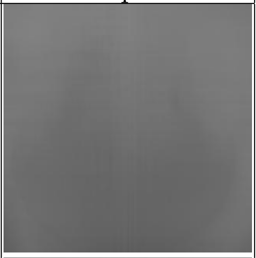
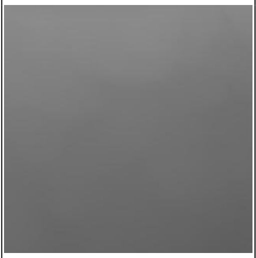
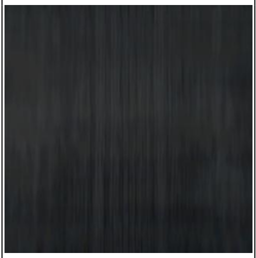
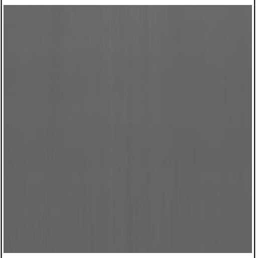
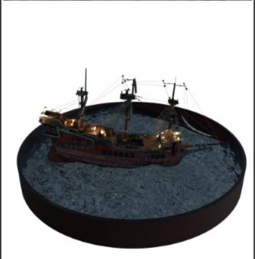
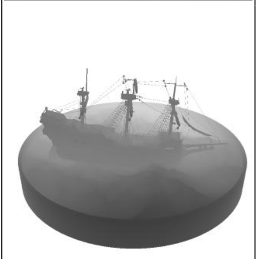
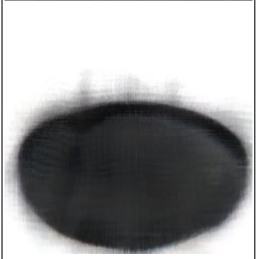
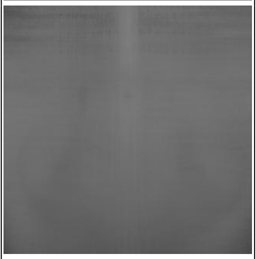
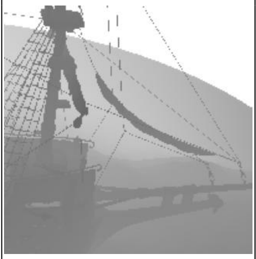
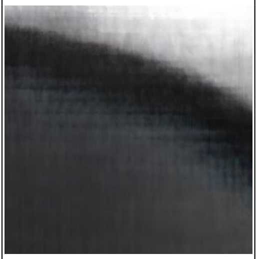
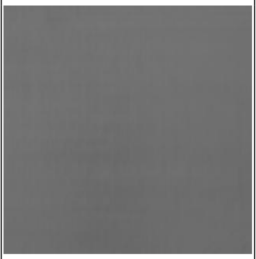
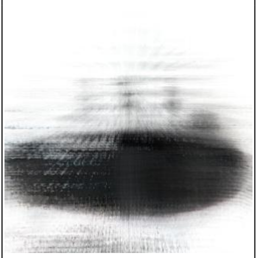
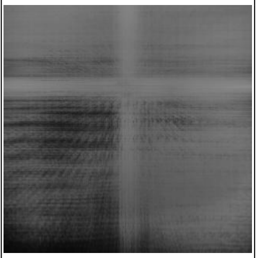
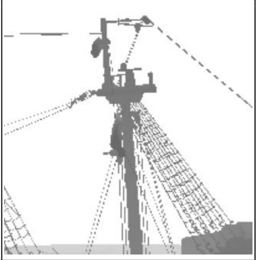
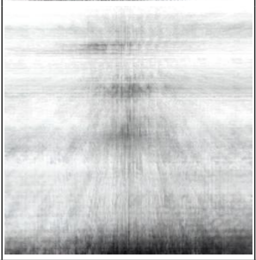
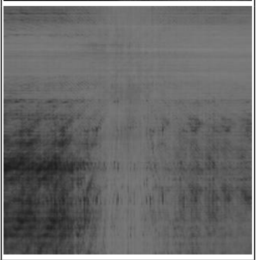
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table 4.11: Depth-supervised mip-NeRF renderings of the synthetic dataset with the original DS-NeRF loss function, $\lambda_d = 0.1$, 5000 iterations

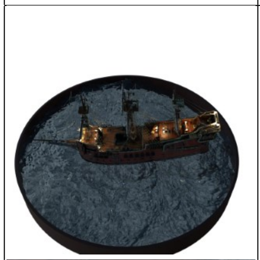
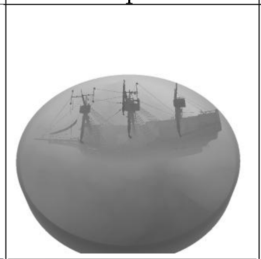
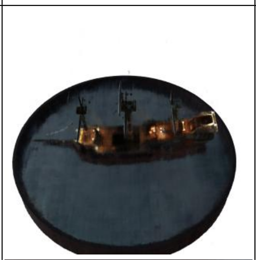
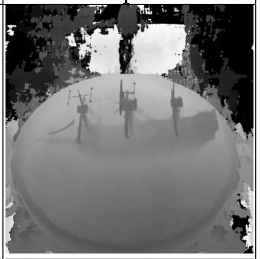
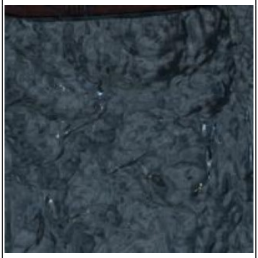
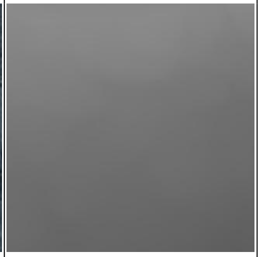
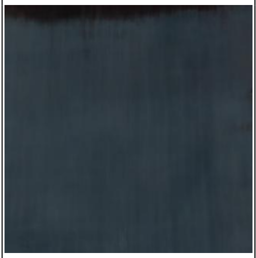
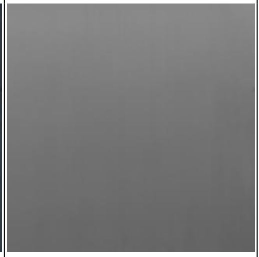
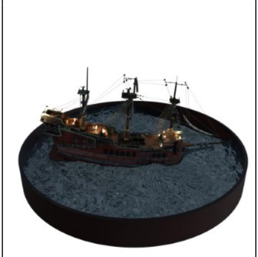
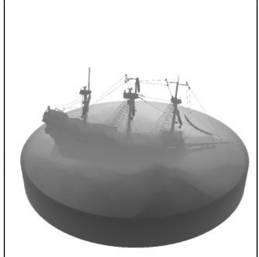
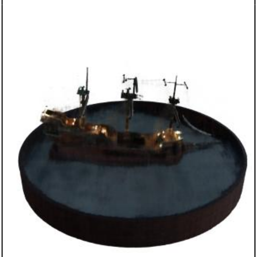
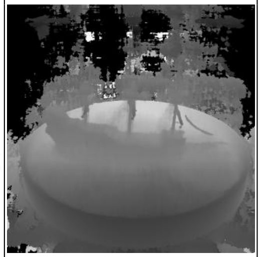
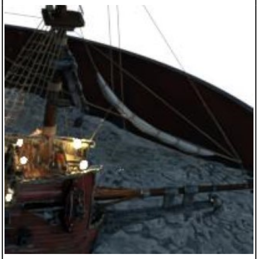
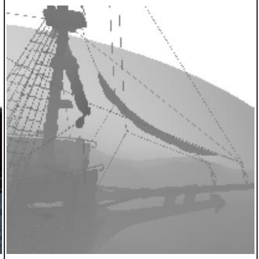
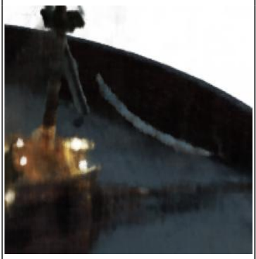
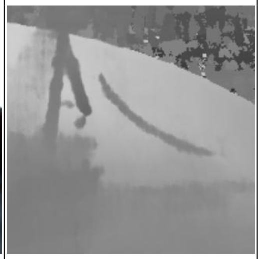
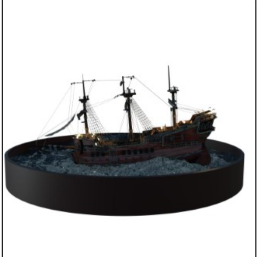
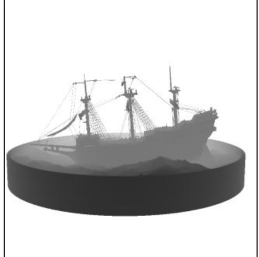
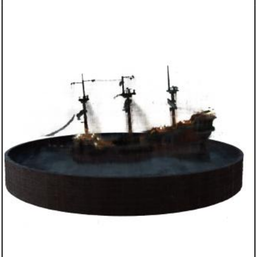
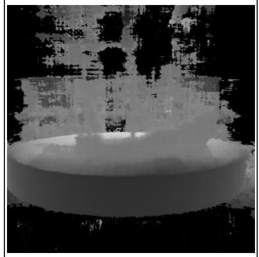
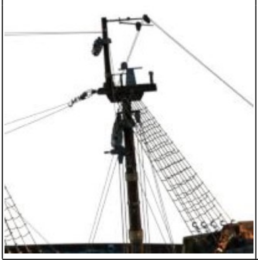
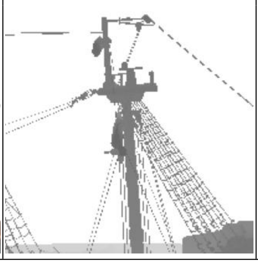
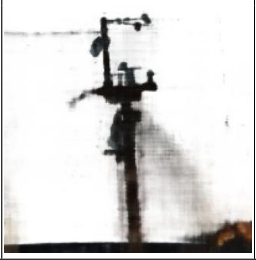
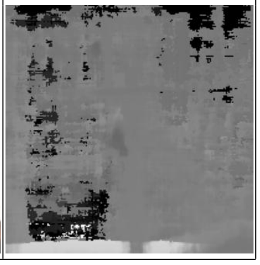
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table 4.12: Depth-supervised mip-NeRF renderings of the synthetic dataset with the derived KL divergence loss function, $\lambda_d = 0.01$, 5000 iterations

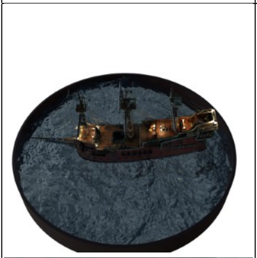
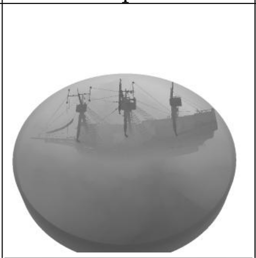
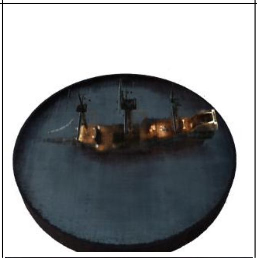
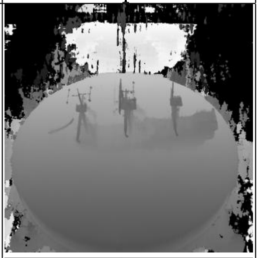
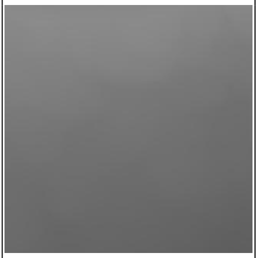
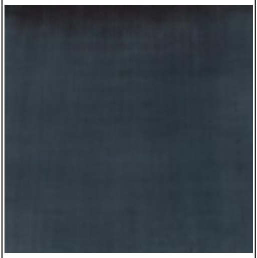
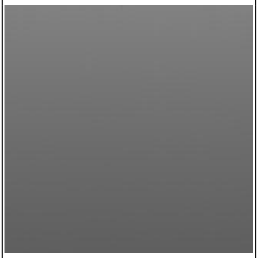
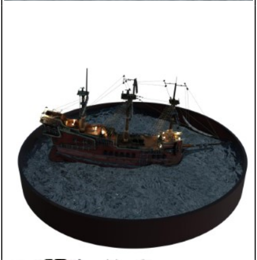
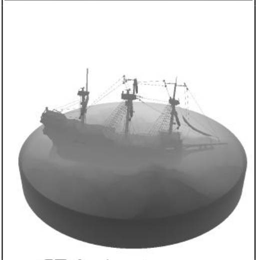
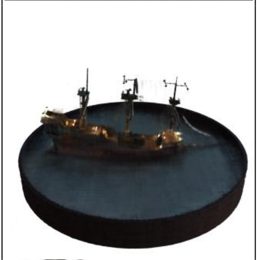
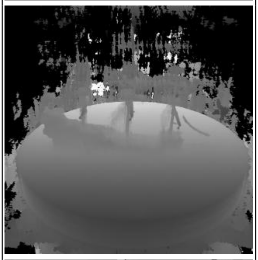
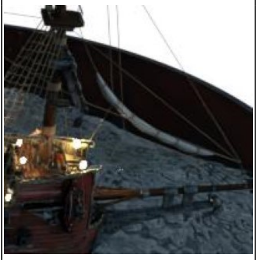
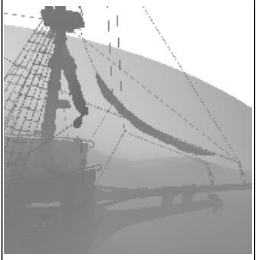
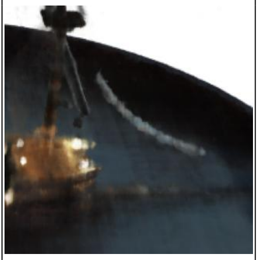
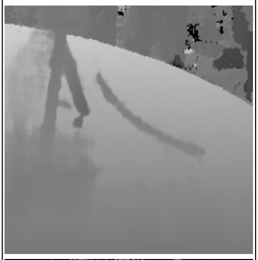
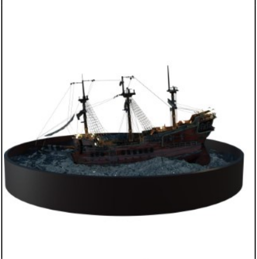
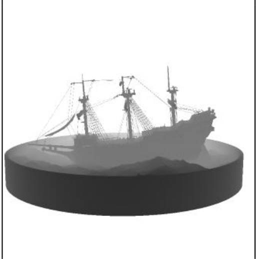
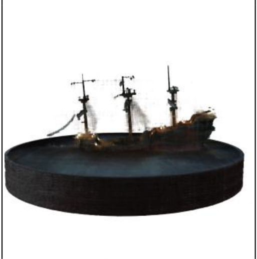
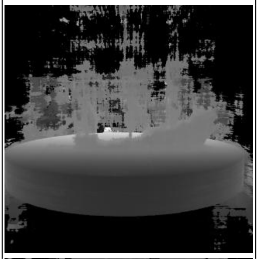
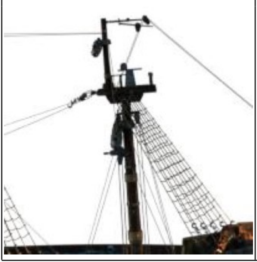
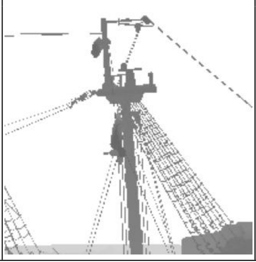
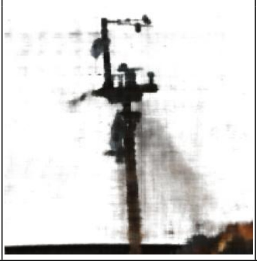
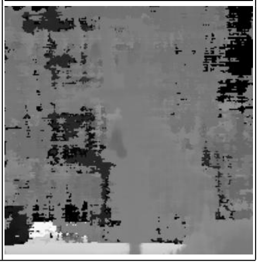
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table 4.13: Depth-supervised mip-NeRF renderings of the synthetic dataset with the derived KL divergence loss function, $\lambda_d = 0.1$, 5000 iterations

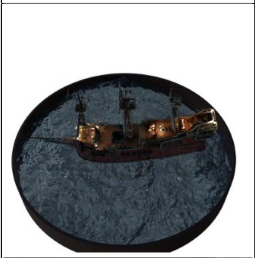
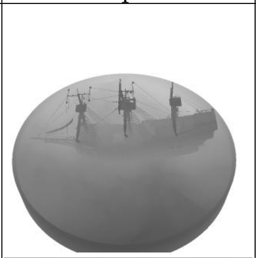
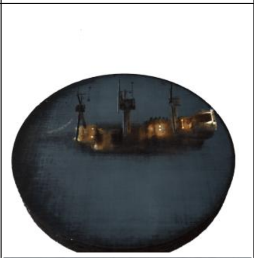
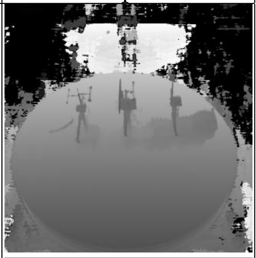
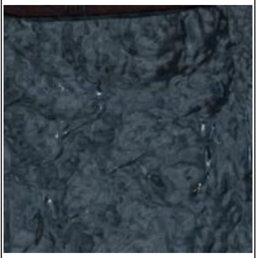
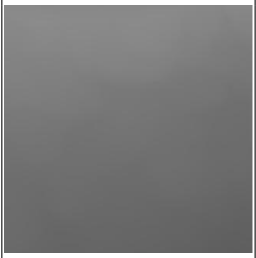
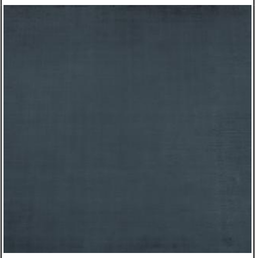
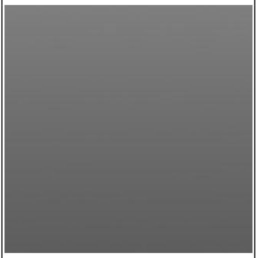
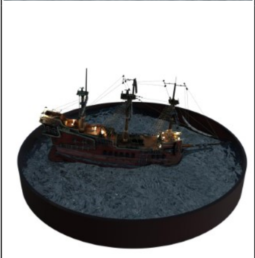
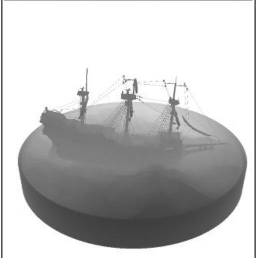
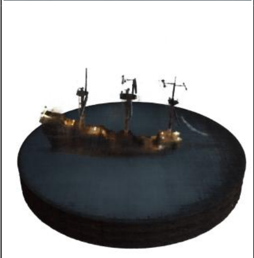
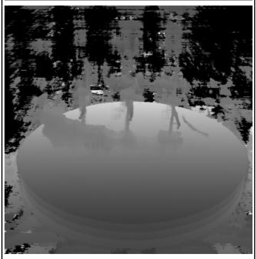
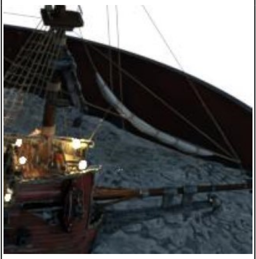
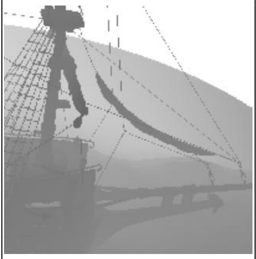
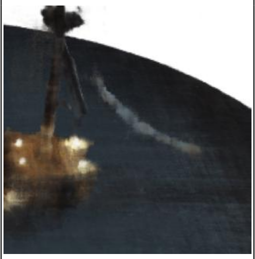
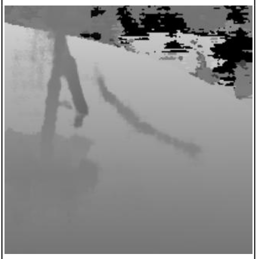
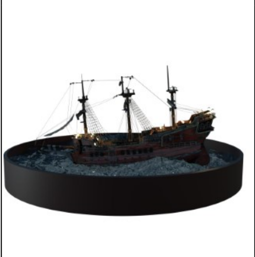
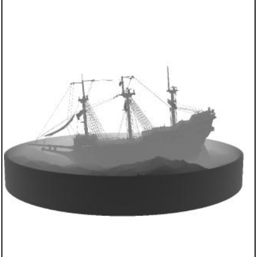
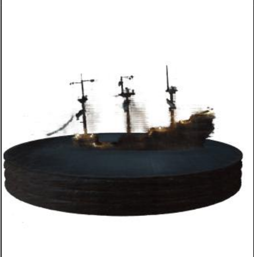
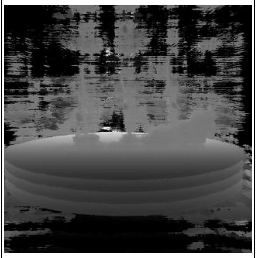
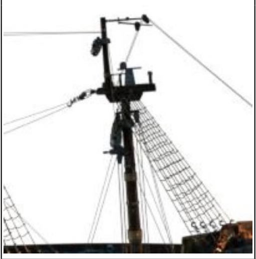
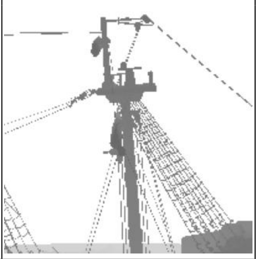
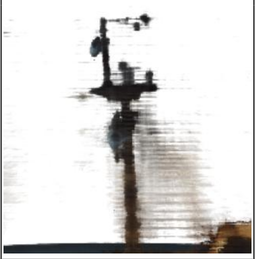
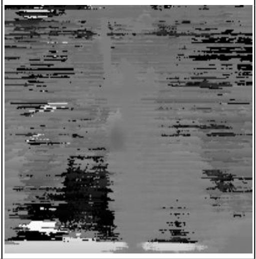
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table 4.14: Depth-supervised mip-NeRF renderings of the synthetic dataset with the derived KL divergence loss function, $\lambda_d = 1.0$, 5000 iterations

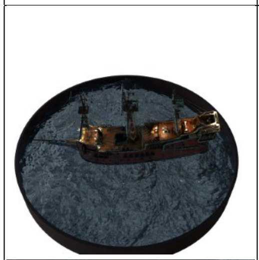
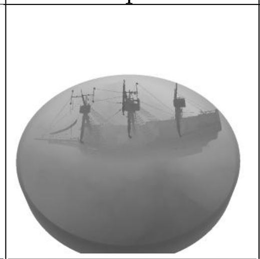
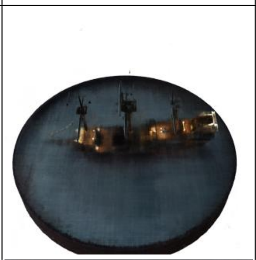
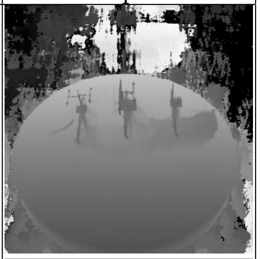
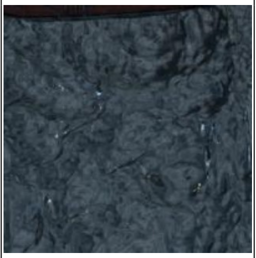
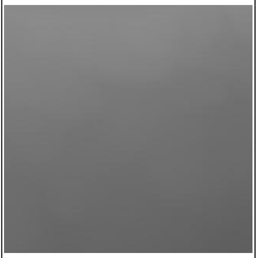
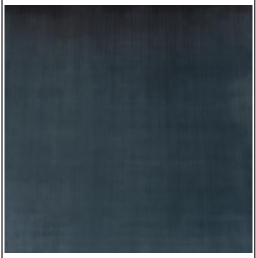
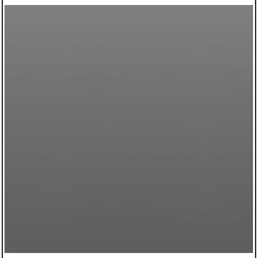
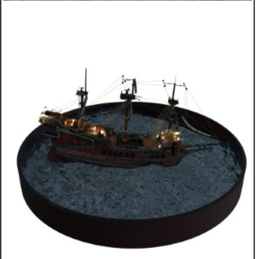
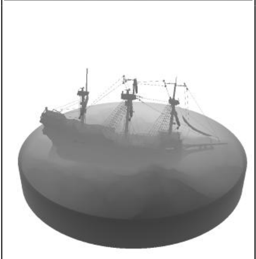
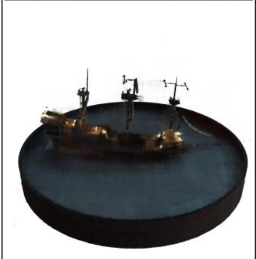
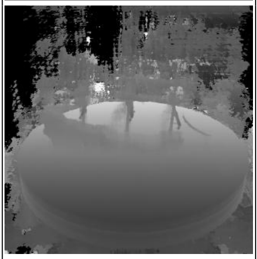
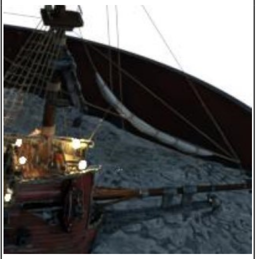
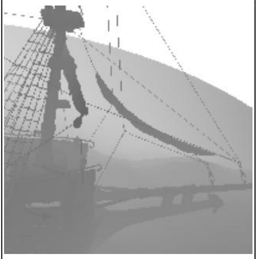
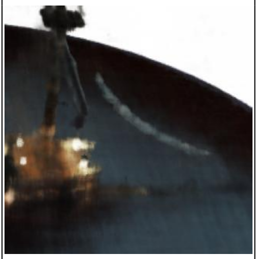
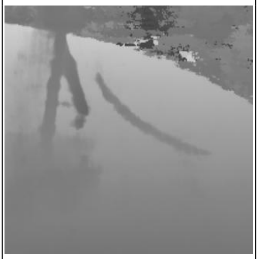
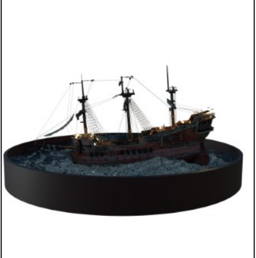
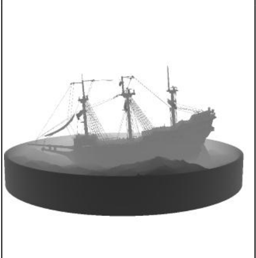
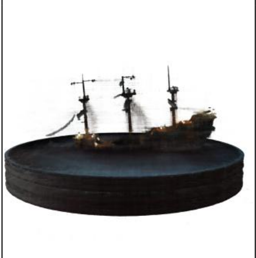
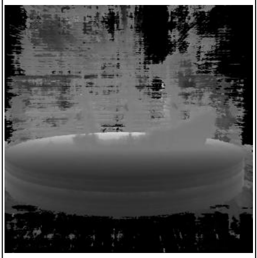
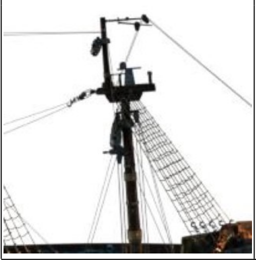
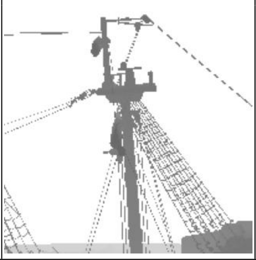
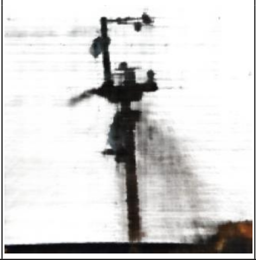
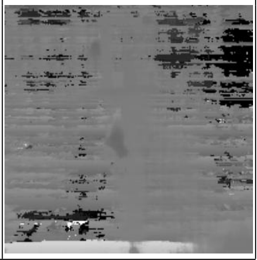
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table 4.15: Depth-supervised mip-NeRF renderings of the synthetic dataset with the derived KL divergence loss function, $\lambda_d = 1.0$ with decay rate $\xi_d = 0.999$, 5000 iterations

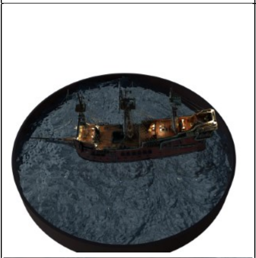
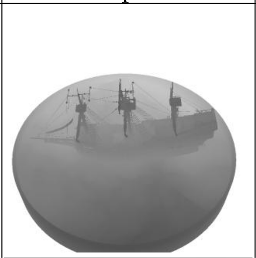
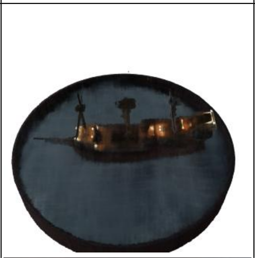
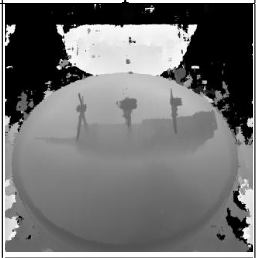
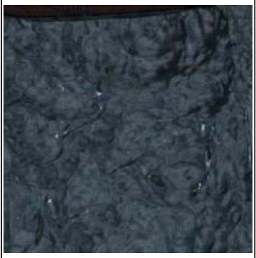
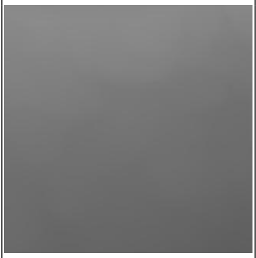
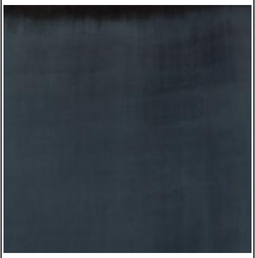
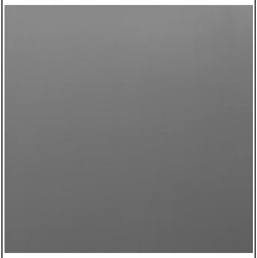
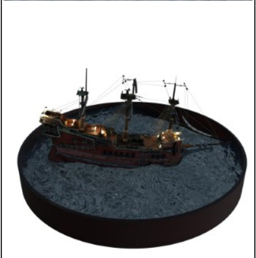
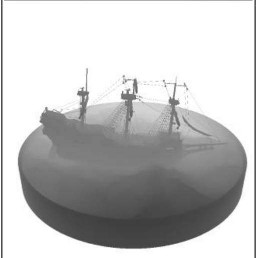
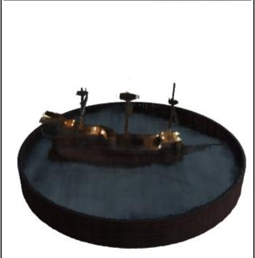
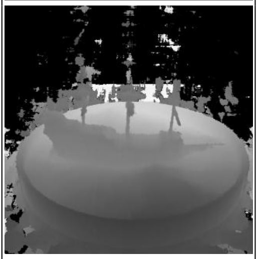
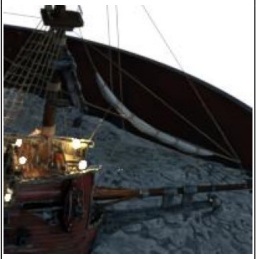
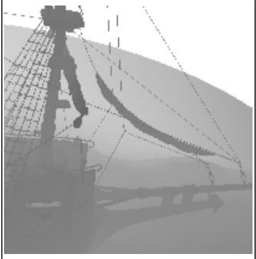
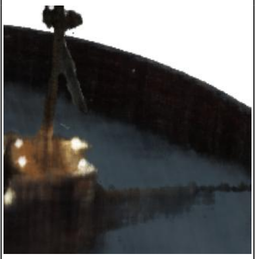
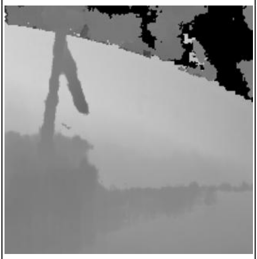
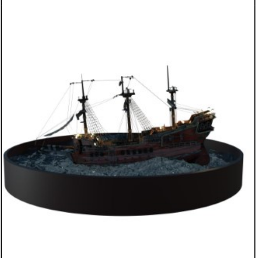
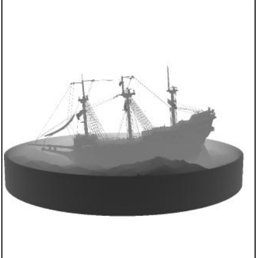
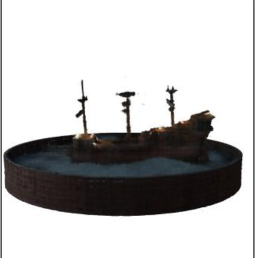
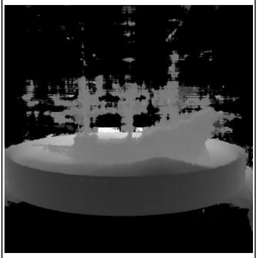
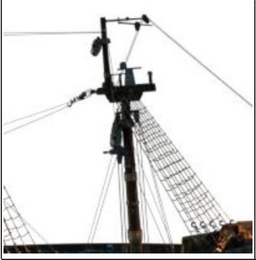
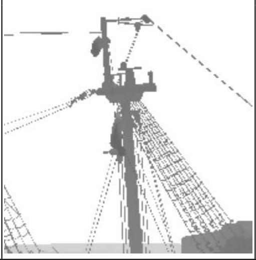
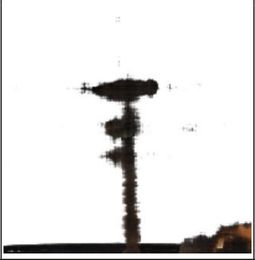
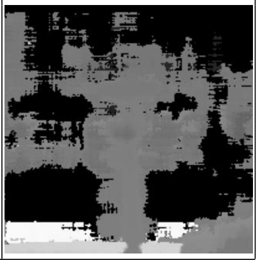
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table 4.16: Depth-supervised mip-NeRF renderings of the synthetic dataset with the mean square loss function, $\lambda_d = 0.1$, 5000 iterations

Discussion

In section 4.1, we have shown that mip-NeRF does not perform well on a bounded synthetic dataset with a small number of viewpoints.

The first experiment, which uses the loss function proposed by the DS-NeRF paper, produces incorrect results with depth maps that are completely incorrect, as can be seen in table 4.11. On the contrary, using the depth loss function that was derived from the Kullback-Leibler divergence of the predicted depth distribution and the measured depth distribution produced results that recovered more details than mip-NeRF without any depth supervision. This experiment has been conducted for different weights of the depth supervision loss function λ_d. Results of these experiments can be seen in tables 4.12, 4.13, and 4.14.

The idea of decreasing the λ_d parameter over time using a specified decay rate ξ_d was tested in one of the experiments as well. For a lower number of iterations, it seems that such an experiment performs no better than those without any decay, as can be seen in table 4.15. However, running such an experiment for a higher number of iterations produces results that are better than those of any other experiments, which can be seen in table A.26.

Finally, using a mean square distance-based loss function leads to results that are sharper than the results produced by mip-NeRF without any depth supervision, but this approach also fails to reconstruct fine detail such as thin ropes. An example of such an experiment is shown in table 4.16.

4.4.3 LLFF Dataset

In the next section, we examine how our suggested depth supervision method performs on a real-life dataset. Since the loss function derived from KL divergence gave the best results for the synthetic dataset, we will only test that for the real-world dataset. The experiments were run using the train_llff_ds.sh script while altering the parameters in the llff_ds.gin configuration file.

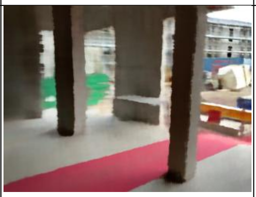
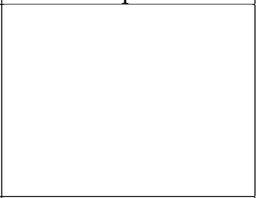
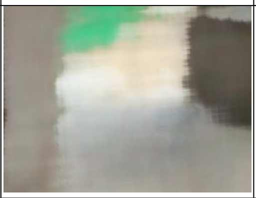
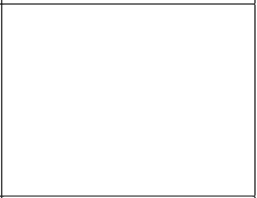
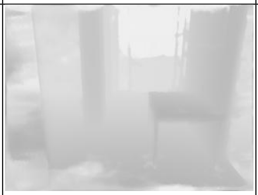
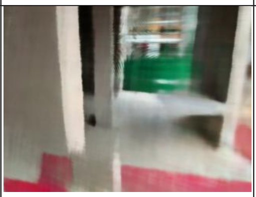
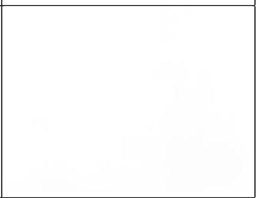
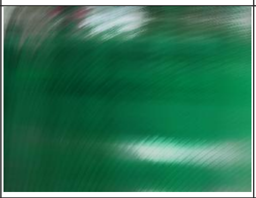
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table 4.17: Depth-supervised mip-NeRF renderings of the large LLFF dataset with the NDC scene normalization; with the derived KL divergence loss function and conversion to NDC, $\lambda_d = 1.0$ with decay rate $\xi_d = 0.999$, 5000 iterations. The rendered depth images are not missing from the table. They are all white in color because the depth for all viewpoints is equal to one.

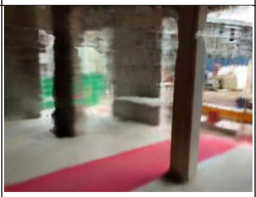
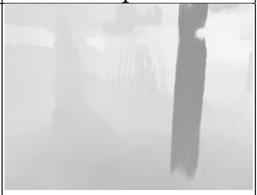
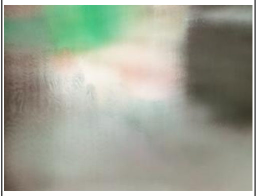
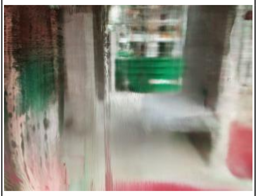
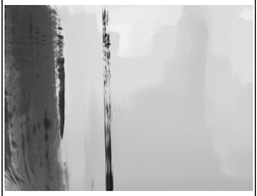
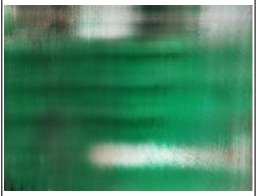
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table 4.18: Depth-supervised mip-NeRF renderings of the large LLFF dataset without the NDC scene normalization; with the derived KL divergence loss function and conversion to NDC, $\lambda_d = 1.0$ with decay rate $\xi_d = 0.999$, 5000 iterations

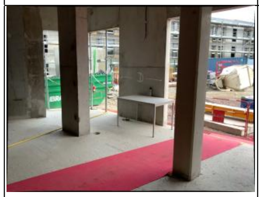
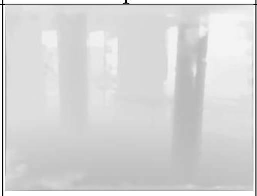
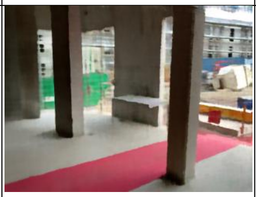
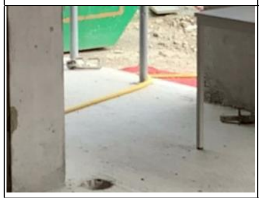
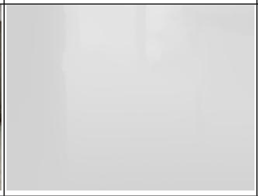
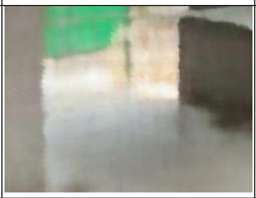
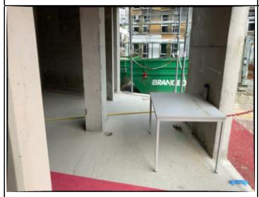
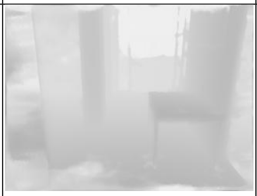
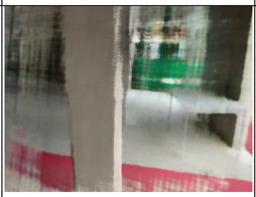
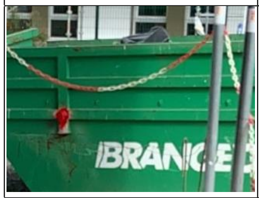
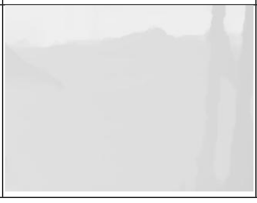
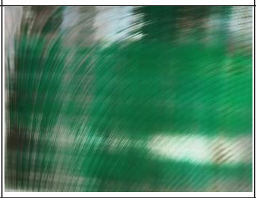
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table 4.19: Depth-supervised mip-NeRF renderings of the small LLFF dataset with the NDC scene normalization; with the derived KL divergence loss function and conversion to NDC, $\lambda_d = 1.0$ with decay rate $\xi_d = 0.999$, 5000 iterations. They are all white in color because the depth for all viewpoints is equal to one.Jk

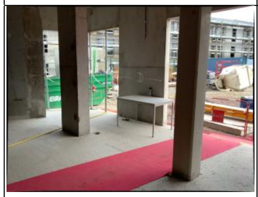
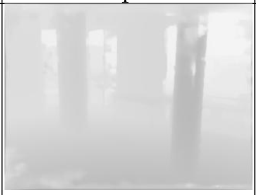
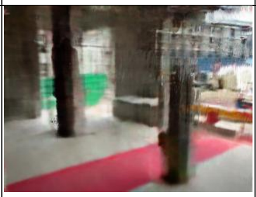
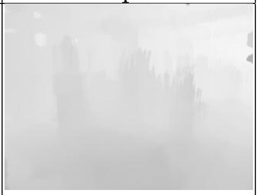
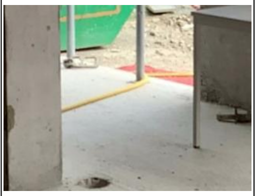
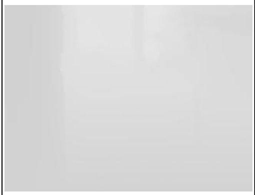
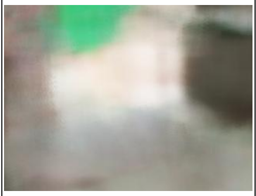
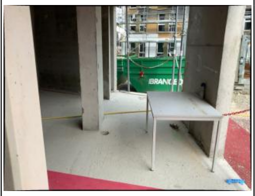
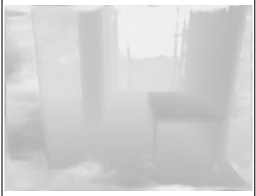
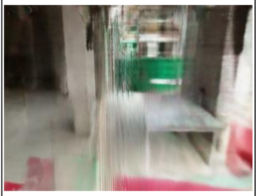
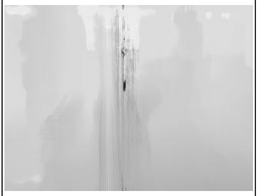
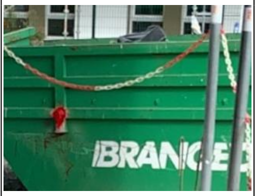
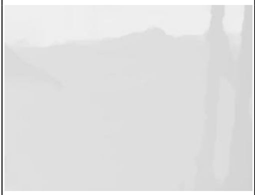
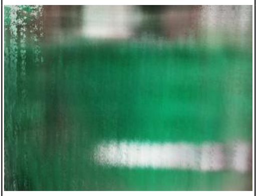
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table 4.20: Depth-supervised mip-NeRF renderings of the small LLFF dataset without the NDC scene normalization; with the derived KL divergence loss function and conversion to NDC, $\lambda_d = 1.0$ with decay rate $\xi_d = 0.999$, 5000 iterations

Discussion

Our experiments have shown that the proposed depth supervision loss function led to results that completely failed to reconstruct the depth of the scene when using normalized device coordinates, leaving all depth maps entirely white (4.17, 4.19). Performing the same depth supervision for a scene that is not normalized results produces inaccurate results (4.18, 4.20).

One possible cause of the first issue is that the proposed depth transformation of the depth map in equation 4.4 is incorrect, while the second issue is probably caused by inaccurate input depth maps. This is evident as using the same procedure on synthetic datasets resulted in improved outcomes compared to mip-NeRF without any depth supervision.

Chapter 5

Conclusion

The conducted experiments have shown that dense information about depth can be used to improve the results of the mip-NeRF method if the number of input images is low. Furthermore, one of the experiments has shown that the depth supervision loss function that was taken from the DS-NeRF method does not improve the training process when used in the context of dense reconstruction. However, a loss function was derived from the definition of Kullback-Leibler divergence improved the results that were produced by the mip-NeRF method.

Applying these findings to the task of reconstructing a real-life scene did not yield satisfactory results, as the generated renderings were of lower quality compared to those produced by the mip-NeRF method.

Bibliography

[1] Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., & Ng, R. (2020). NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. *arXiv*. `https://doi.org/https://arxiv.org/abs/2003.08934v2`

[2] Barron, J. T., Mildenhall, B., Tancik, M., Hedman, P., & Srinivasan, P. P. (2021). Mip-NeRF: A Multiscale Representation for Anti-Aliasing Neural Radiance Fields. *arXiv*. `https://doi.org/https://arxiv.org/abs/2103.13415v3`

[3] Verbin, D., Hedman, P., Mildenhall, B., Zickler, T., Barron, J. T., & Srinivasan, P. P. (2021). Ref-NeRF: Structured View-Dependent Appearance for Neural Radiance Fields. *arXiv*. `https://doi.org/https://arxiv.org/abs/2112.03907v1` 2021.

[4] Verbin, D., Hedman, P., Mildenhall, B., Zickler, T., Barron, J. T., & Srinivasan, P. P. (2021). Ref-NeRF: Structured View-Dependent Appearance for Neural Radiance Fields. *arXiv*. `https://doi.org/https://arxiv.org/abs/2112.03907v1`

[5] Hu, T., Liu, S., Chen, Y., Shen, T., & Jia, J. (2022). EfficientNeRF: Efficient Neural Radiance Fields. *arXiv*. `https://doi.org/https://arxiv.org/abs/2206.00878v1`

[6] Barron, J. T., Mildenhall, B., Verbin, D., Srinivasan, P. P., & Hedman, P. (2021). Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields. *arXiv*. `https://doi.org/https://arxiv.org/abs/2111.12077v3`

[7] Meta AI Research (n.d.). *Novel View Synthesis*. Papers With Code. `https://paperswithcode.com/task/novel-view-synthesis`

[8] Bento, C. (2021, September 21). *Multilayer Perceptron Explained with a Real-Life Example and Python Code: Sentiment Analysis*. Towards Data Science. `https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141`

[9] House, D. H. (2014, January 21). Coordinate Systems [Lecture notes]. `https://people.computing.clemson.edu/~dhouse/courses/405/notes/coord-systems.pdf`

[10] Krautz, C. (2019, March 25). *What are the coordinates?* Medium. `https://medium.com/@christophkrautz/what-are-the-coordinates-225f1ec0dd78`

[11] Shreiner, D., Sellers, G., Kessenisch, J., & Licea-Kane, B. (2013). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3.* (8th ed.) Addison-Wesley.

[12] Felkel, P. (2022, April 9). *Transformations (in OpenGL)* [PowerPoint Slides]. `https://cent.felk.cvut.cz/courses/PGR/lectures/04_Transform_1.pdf`

[13] De Vries, J. (n.d.). *Coordinate Systems.* LearnOpenGL. `https://learnopengl.com/Getting-started/Coordinate-Systems`

[14] Songho, H. A. (n.d.). *OpenGL Transformations.* Songho.ca. `http://www.songho.ca/opengl/gl_transform.html`

[15] OpenGL Wiki contributors (2021, February 11). Vertex Post-Processing. OpenGL Wiki. `https://www.khronos.org/opengl/wiki/Vertex_Post-Processing#Viewport_transform`

[16] Sala, P. L. (2006, July 20). *Camera Models and Parameters* [PowerPoint Slides]. `https://ftp.cs.toronto.edu/pub/psala/VM/camera-parameters.pdf`

[17] Fusiello, A. (n.d.). Elements of Geometric Computer Vision. `https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/FUSIELLO4/tutorial.html`

[18] Ortiz, L., Gonçalves, L., & Cabrera, E. (2017). *A Generic Approach for Error Estimation of Depth Data from (Stereo and RGB-D) 3D Sensors.* `https://doi.org/http://dx.doi.org/10.20944/preprints201705.0170.v1`

[19] Ms Aerin. (2021, September 21). *Camera Calibration: Camera Geometry and The Pinhole Model.* Towards Data Science. `https://towardsdatascience.com/camera-calibration-fda5beb373c3`

[20] Zaal, G., & Kuhn, C. (2013, February). *Suzanne's Revenge - Pirate Ship (+Blend).* Blenderartists.org. `https://blenderartists.org/t/suzannes-revenge-pirate-ship-blend/565664`

[21] (2013, February 1). *The Project.* Artwin. `https://artwin-project.eu/`

[22] Mildenhall, B., Verbin, D., Srinivasan, P. P., Hedman, P., Martin-Brualla, R., & Barron, J. T. (2022, October 13). *MultiNeRF: A*

Code Release for Mip-NeRF 360, Ref-NeRF, and RawNeRF. GitHub. https://github.com/google-research/multinerf

[23] Kusche, C., Reclik, T., Freund, M., Al-Samman, T., Kerzel, U., & Korte-Kerzel, S. (2019). *Large-area, high-resolution characterisation and classification of damage mechanisms in dual-phase steel using deep learning*. PLoS One, 14(5), e0216493.

[24] Mildenhall, B., Srinivasan, P. P., Ortiz-Canyon, R., Kalantari, N. K., Ramamoorthi, R., Ng, R., & Kar, A. (2021, January 21). *Local Light Field Fusion*. GitHub. https://github.com/Fyusion/LLFF

[25] Leonel, J. (2018, October 29). *Multilayer Perceptron*. Medium. https://medium.com/@jorgesleonel/multilayer-perceptron-6c5db6a8dfa3

[26] Baheti, P. (2021, May 27). *Activation Functions in Neural Networks [12 Types & Use Cases]*. V7Labs. https://www.v7labs.com/blog/neural-networks-activation-functions

[27] Hlaváč, V., Matas, J., & Drbohlav, O. (2020, October 29). *Perceptron Classifier, Empirical vs Structural Risk Minimization* [Lecture Slides].

[28] Dourmashkin, P. (2023). *Classical Mechanics* (p. 32). LibreTexts. https://batch.libretexts.org/print/Letter/Finished/phys-24413/Full.pdf

[29] Schoenberger, J. L. (n.d.). *Output Format*. COLMAP. https://colmap.github.io/format.html

[30] Shene, C. K. (2011, May 4). *Geometric Transformations* [Lecture Notes]. https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/geometry/geo-tran.html

[31] Hartley, R., & Zisserman, A. (2000). *Multiple View Geometry in Computer Vision* (2nd ed.). Cambridge University Press.

[32] Ladislav, Č. (2022, November 3). *VGO - Color Perception and Representation* [Lecture Slides].

[33] Eck, D. J. (2021). *Introduction to Computer Graphics* (p. 19). Hobart and William Smith Colleges. https://math.hws.edu/eck/cs424/downloads/graphicsbook-linked.pdf

[34] Deng, K., Liu, A., Zhu, J. Y., & Ramanan, D. (2023, January 19). Depth-supervised NeRF: Fewer Views and Faster Training for Free. GitHub. https://github.com/dunbar12138/DSNeRF#depth-supervised-nerf-fewer-views-and-faster-training-for-free

Appendix A
Additional Results

The following chapter contains several tables with results that were not relevant enough to show in the Experiments chapter but still present valuable results.

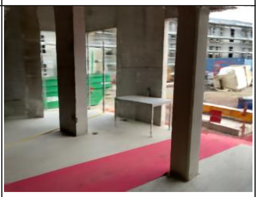
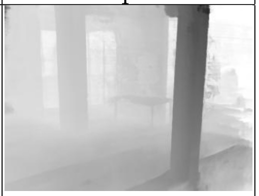
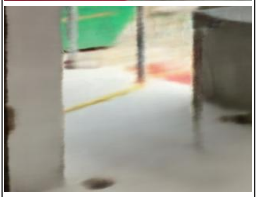
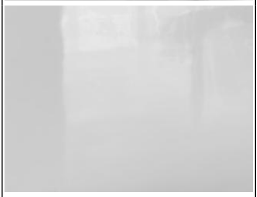
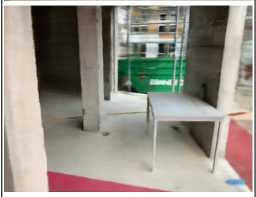
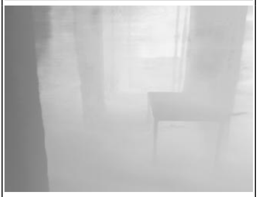
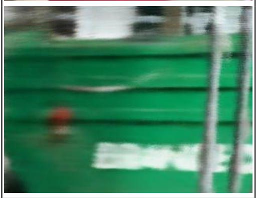
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table A.1: Mip-NeRF renderings of the large LLFF dataset with the NDC scene normalization, 5000 iterations

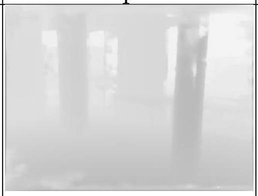
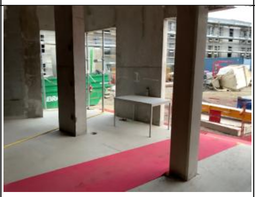
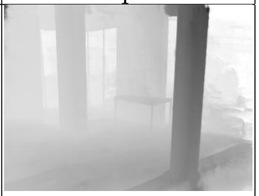
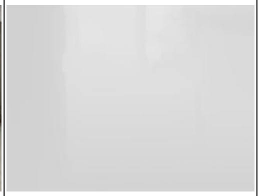
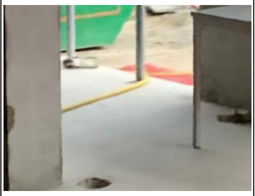
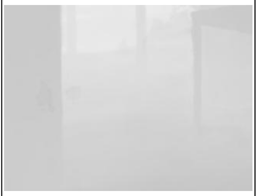
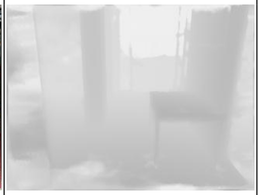
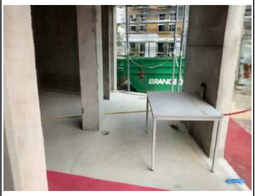
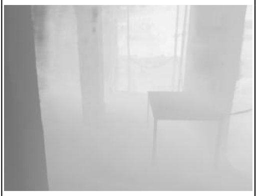
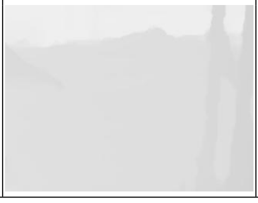
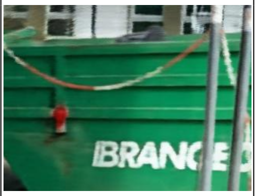
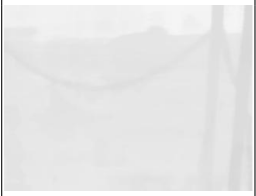
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table A.2: Mip-NeRF renderings of the large LLFF dataset with the NDC scene normalization, 25000 iterations

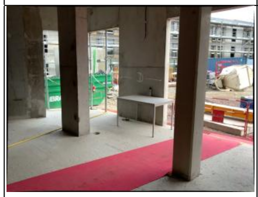
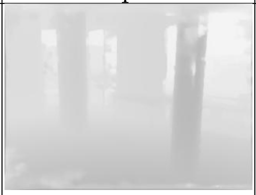
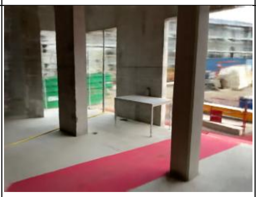
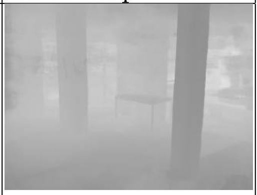
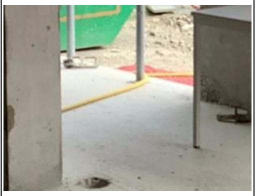
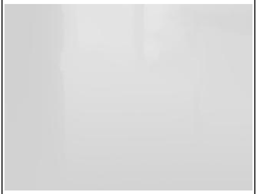
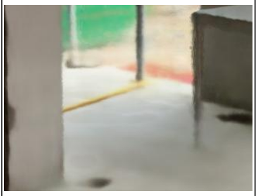
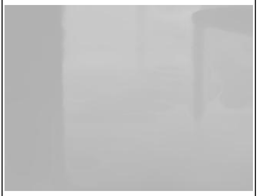
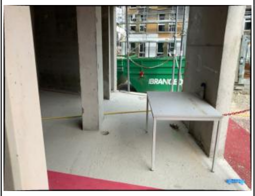
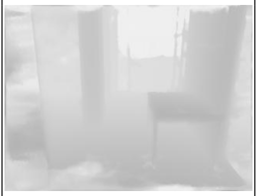
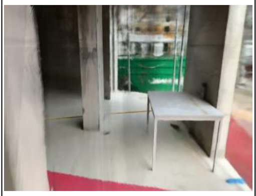
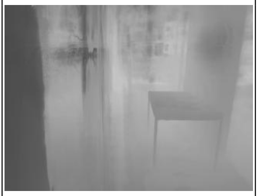
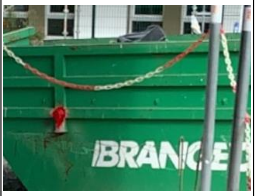
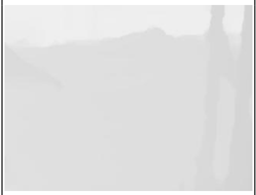
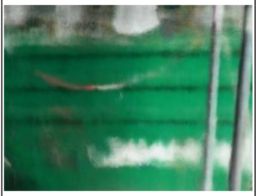
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table A.3: Mip-NeRF renderings of the large LLFF dataset without the NDC scene normalization, 5000 iterations

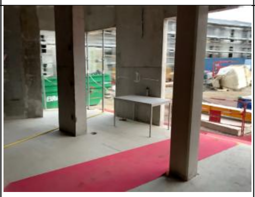
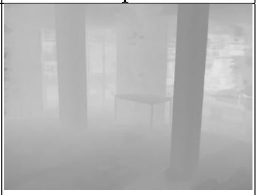
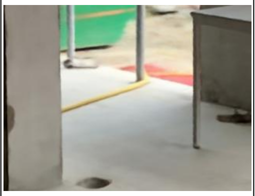
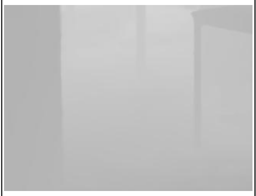
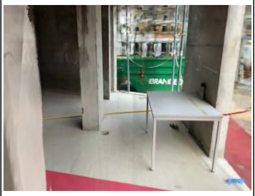
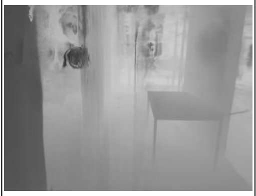
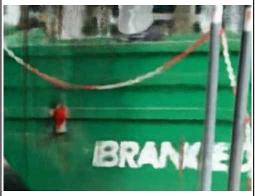
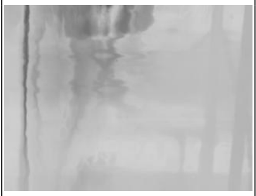
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table A.4: Mip-NeRF renderings of the large LLFF dataset without the NDC scene normalization, 25000 iterations

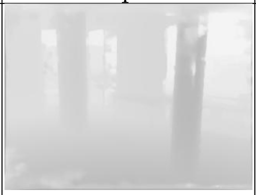
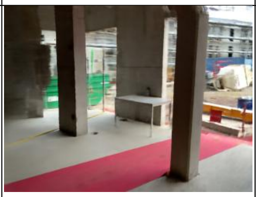
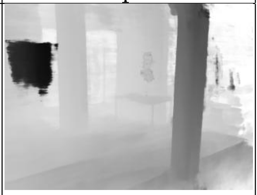
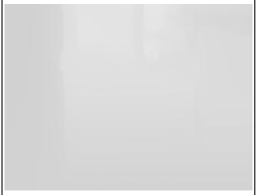
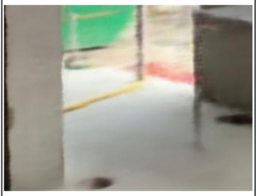
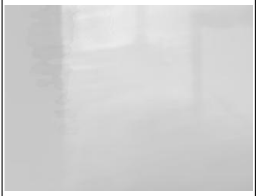
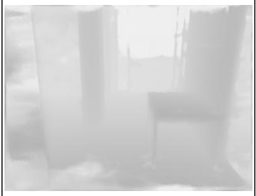
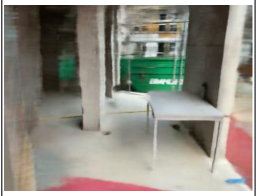
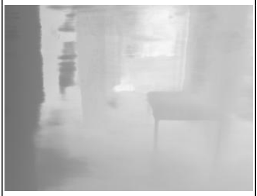
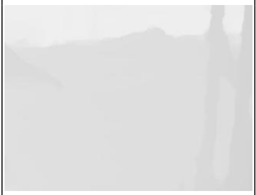
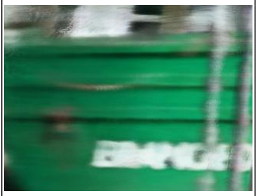
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table A.5: Mip-NeRF renderings of the small LLFF dataset with the NDC scene normalization, 5000 iterations

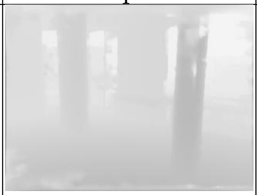
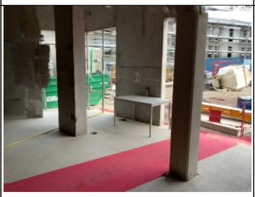
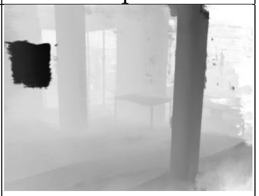
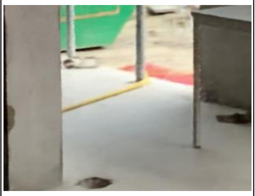
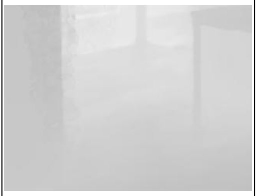
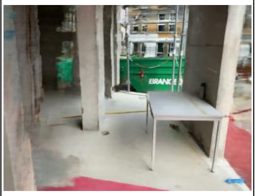
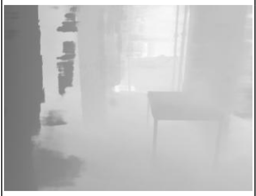
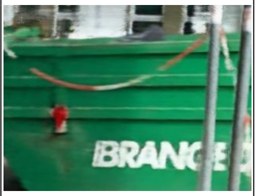
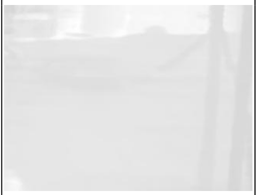
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table A.6: Mip-NeRF renderings of the small LLFF dataset with the NDC scene normalization, 25000 iterations

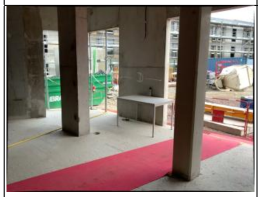
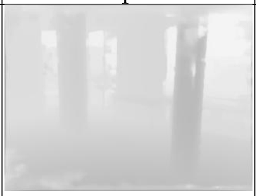
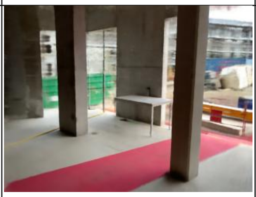
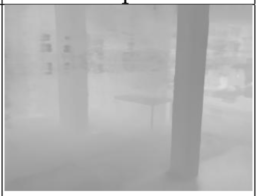
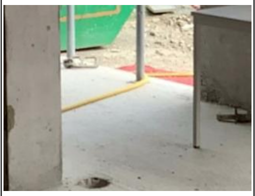
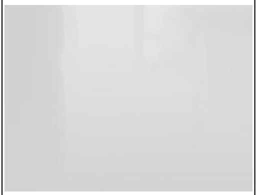
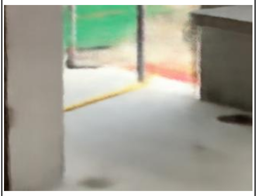
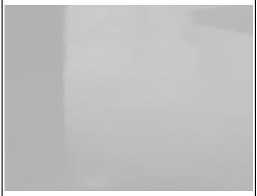
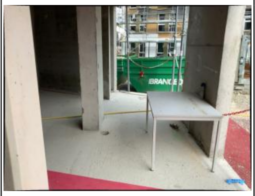
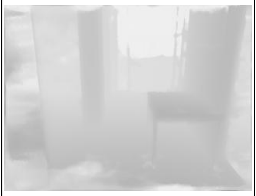
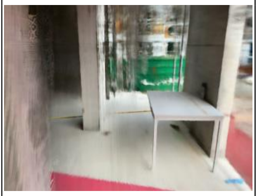
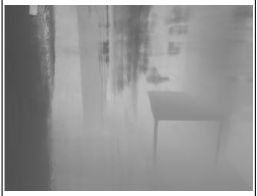
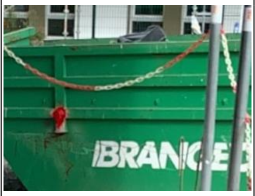
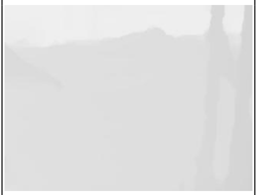
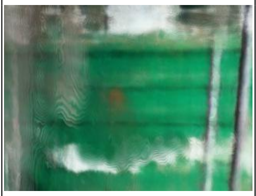
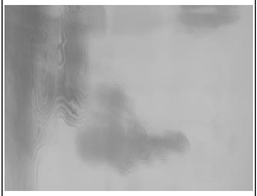
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table A.7: Mip-NeRF renderings of the small LLFF dataset without the NDC scene normalization, 5000 iterations

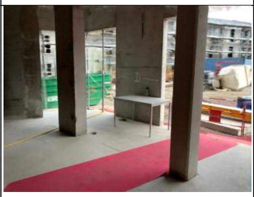
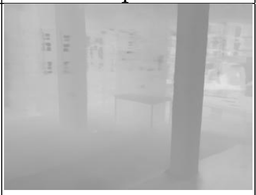
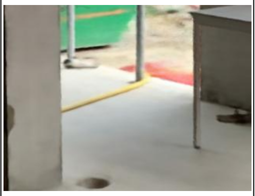
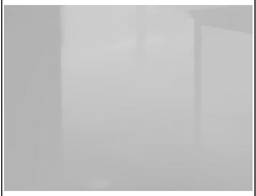
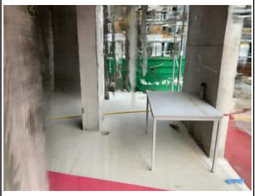
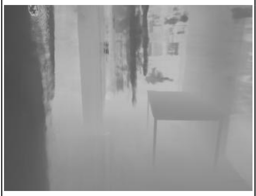
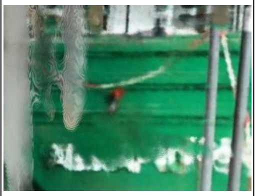
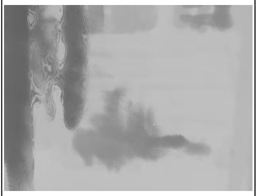
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			

Table A.8: Mip-NeRF renderings of the small LLFF dataset without the NDC scene normalization, 25000 iterations

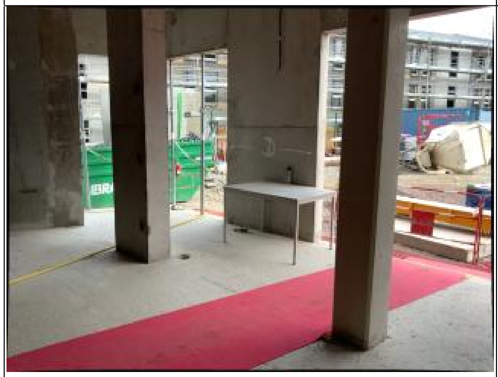
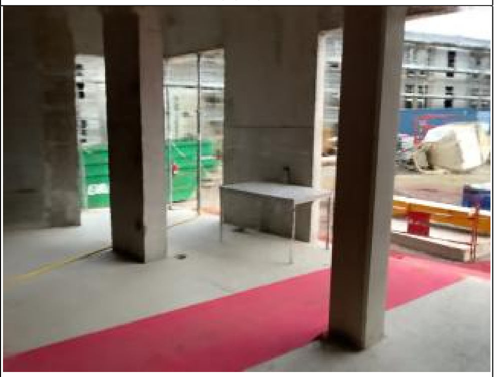
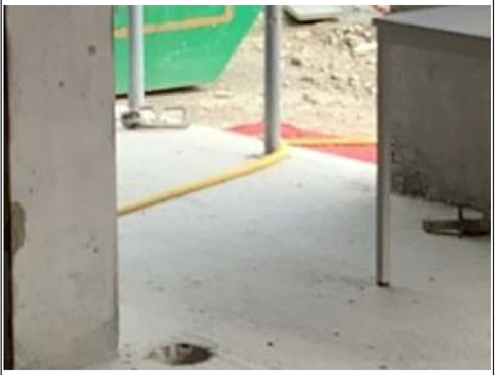
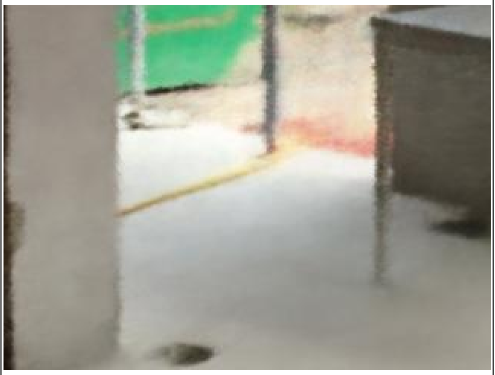
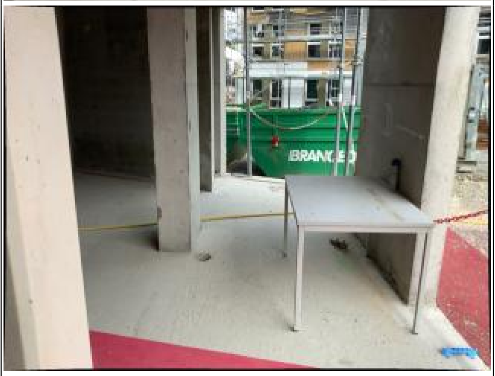
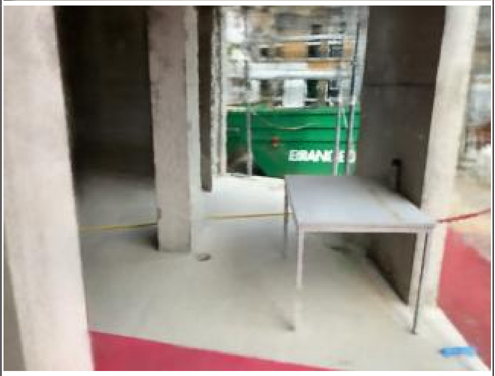
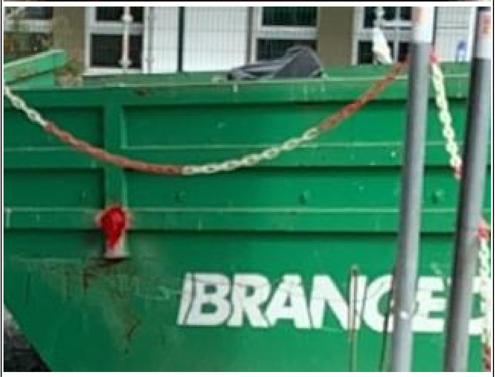
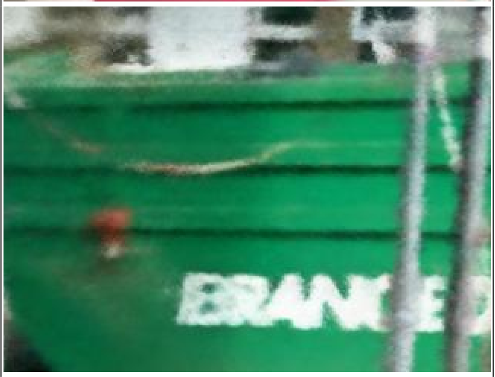
GT color	Rendered color
	
	
	
	

Table A.9: Mip-NeRF 360 renderings of the large LLFF dataset, 10000 iterations

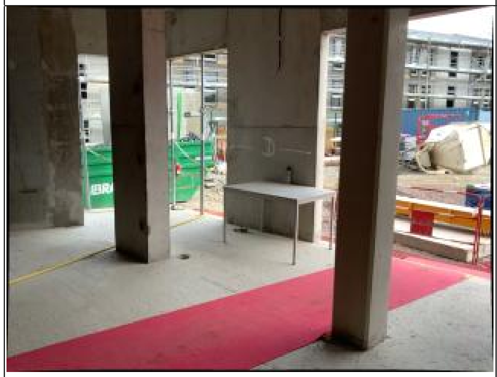
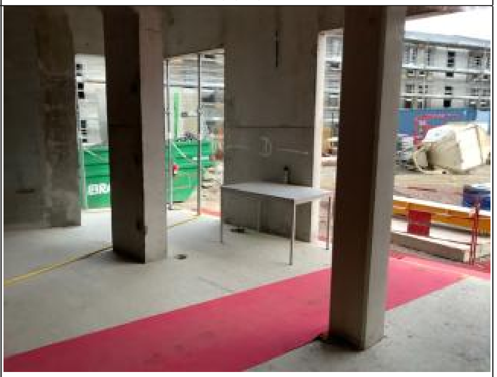
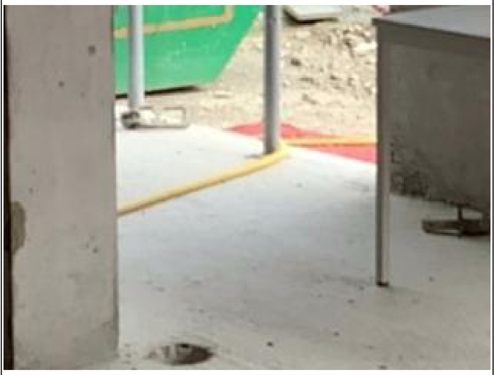
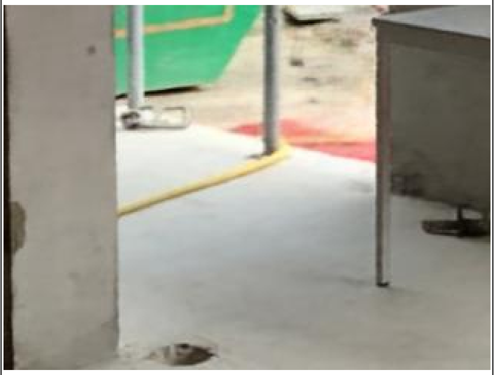
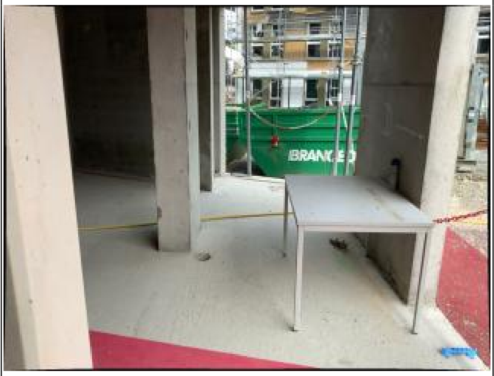
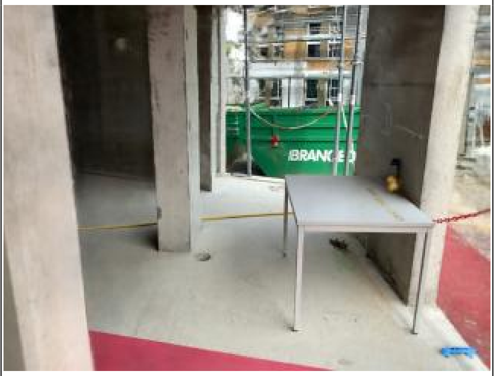
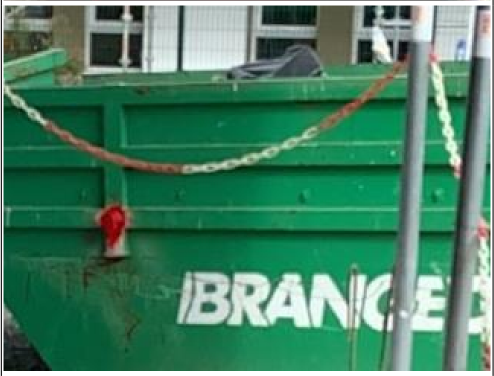
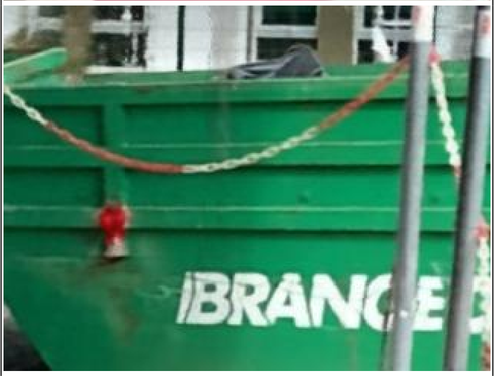
GT color	Rendered color
	
	
	
	

Table A.10: Mip-NeRF 360 renderings of the large LLFF dataset, 50000 iterations

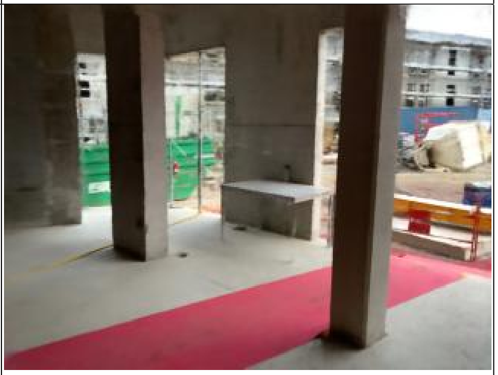
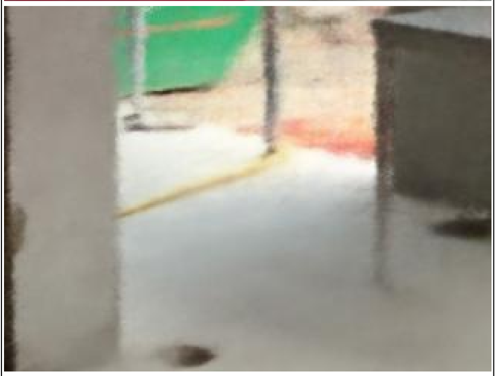
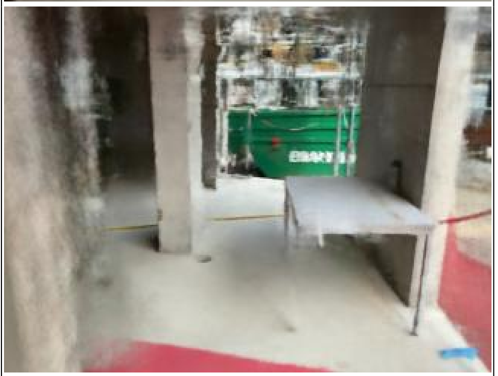
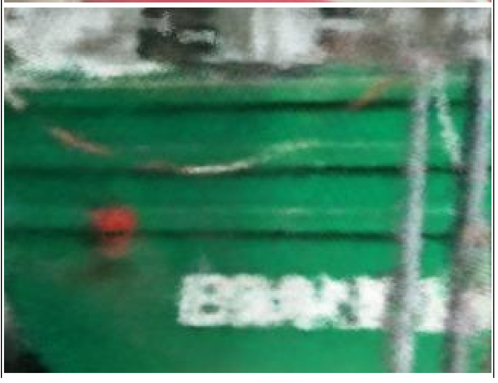
GT color	Rendered color
	
	
	
	

Table A.11: Mip-NeRF 360 renderings of the small LLFF dataset, 10000 iterations

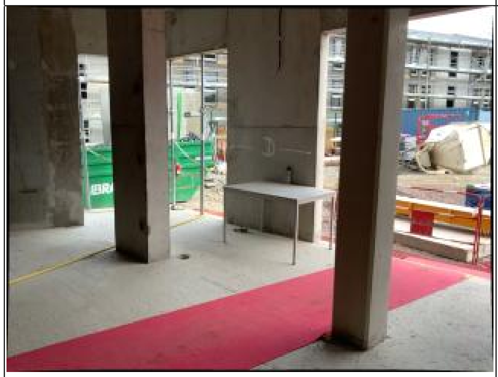
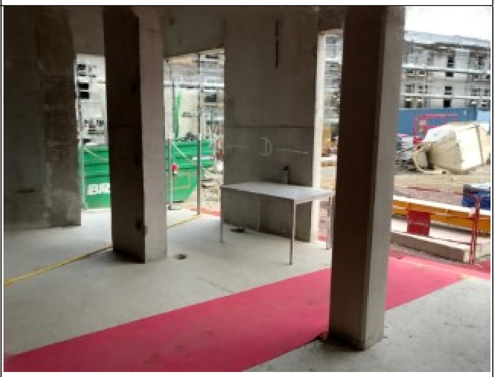
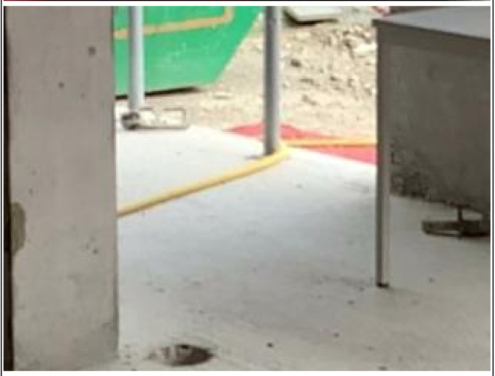
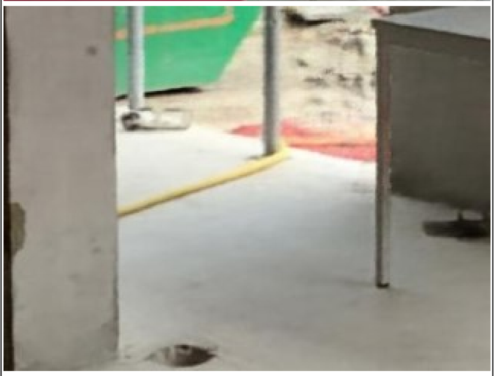
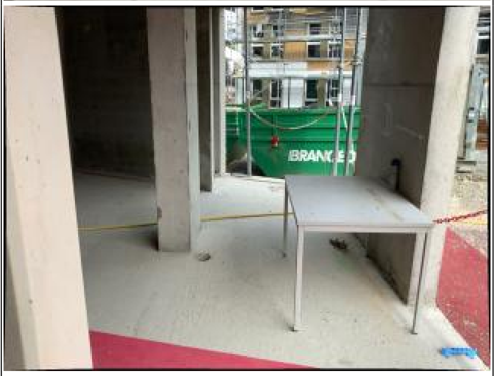
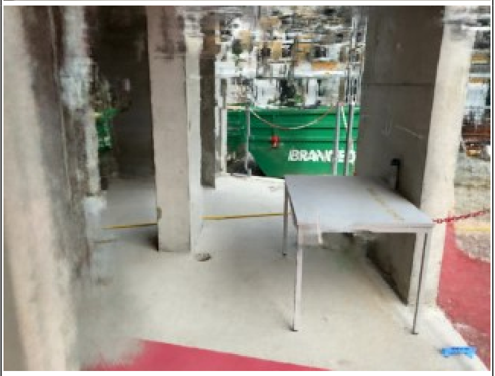
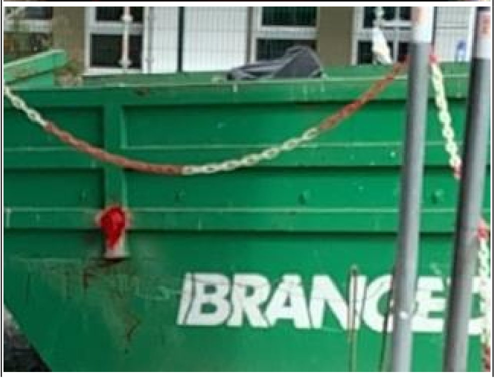
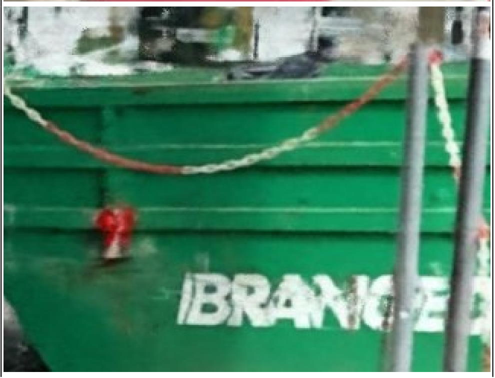
GT color	Rendered color
	
	
	
	

Table A.12: Mip-NeRF 360 renderings of the small LLFF dataset, 50000 iterations

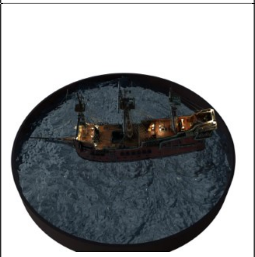
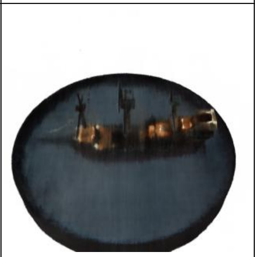
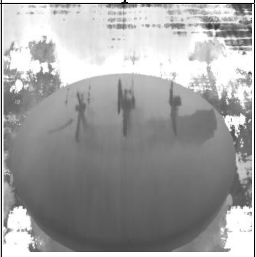
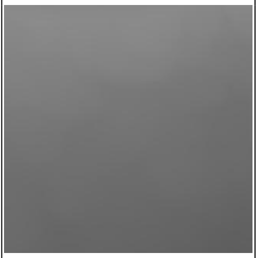
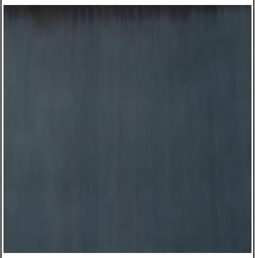
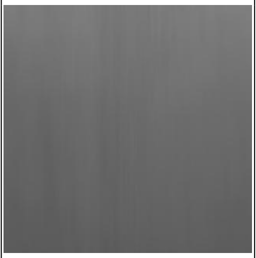
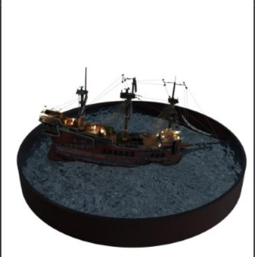
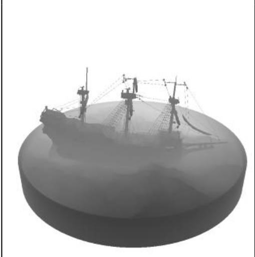
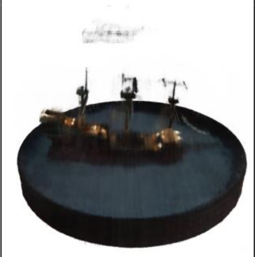
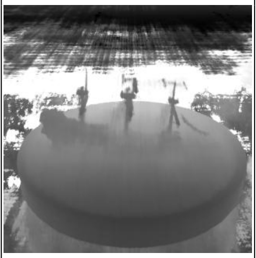
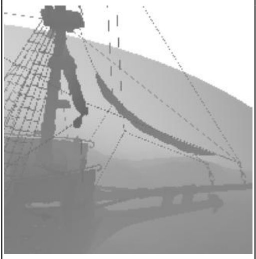
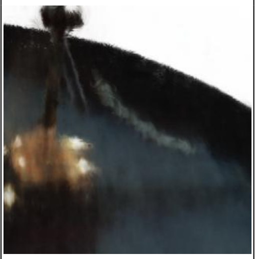
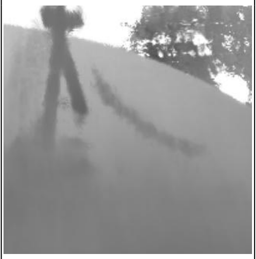
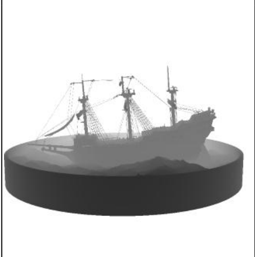
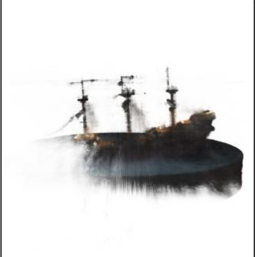
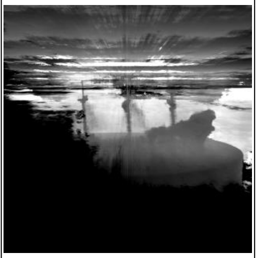
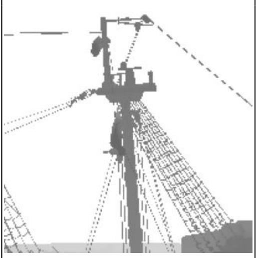
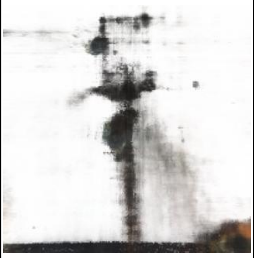
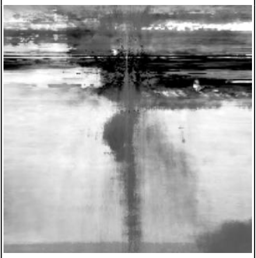
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.13: Mip-NeRF renderings of the synthetic dataset without any depth supervision, 5000 iterations

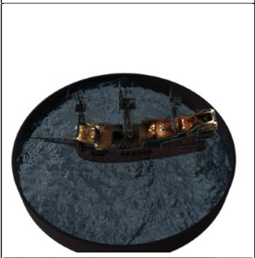
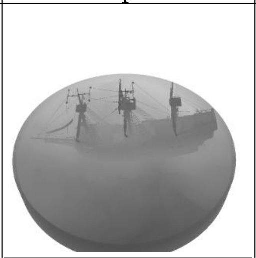
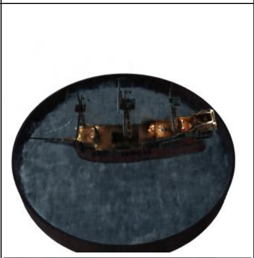
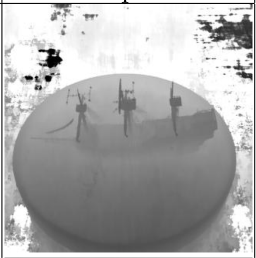
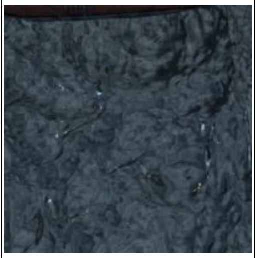
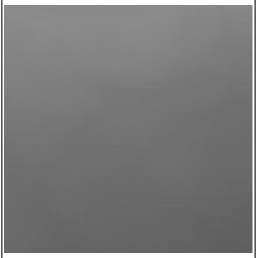
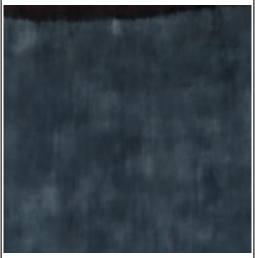
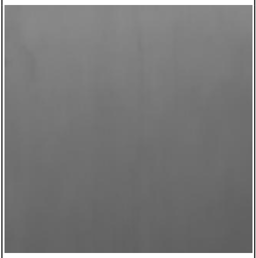
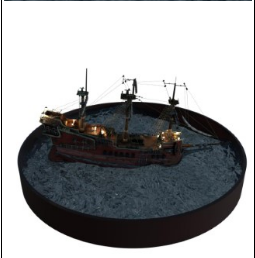
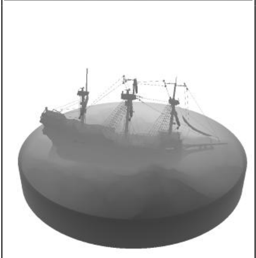
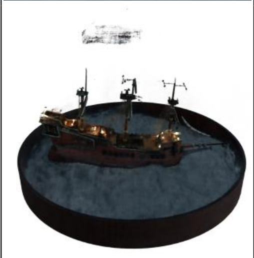
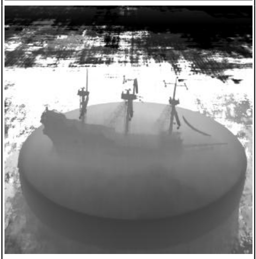
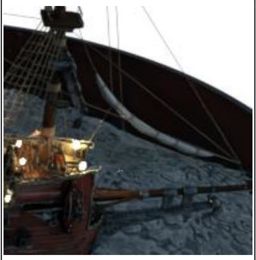
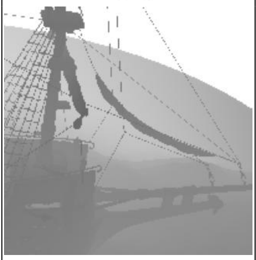
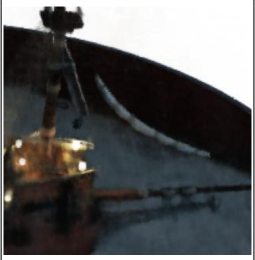
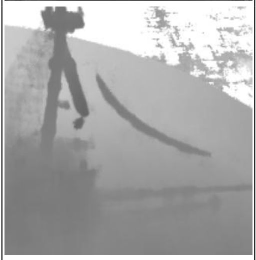
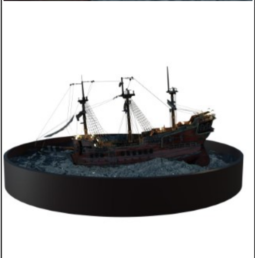
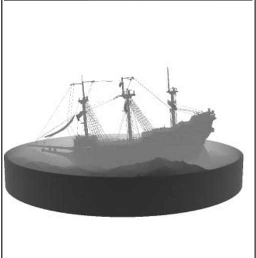
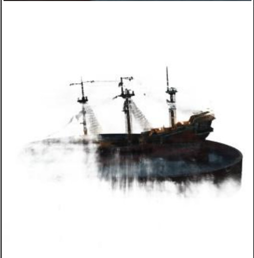
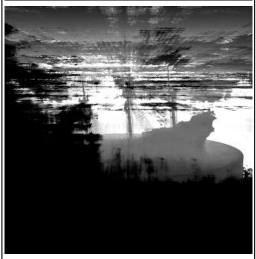
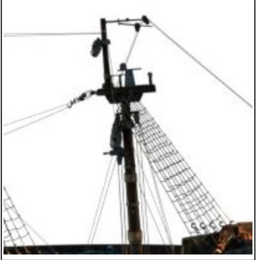
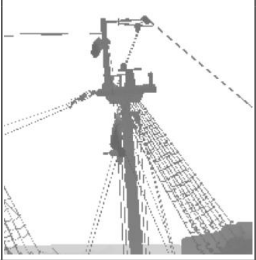
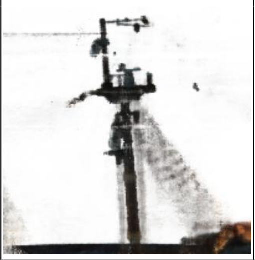
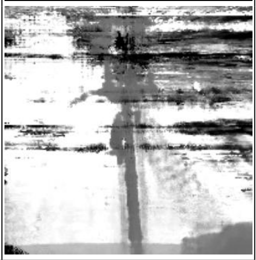
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.14: Mip-NeRF renderings of the synthetic dataset without any depth supervision, 25000 iterations

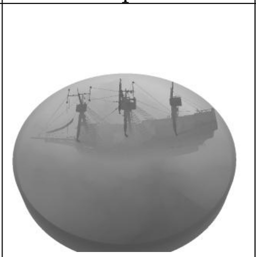
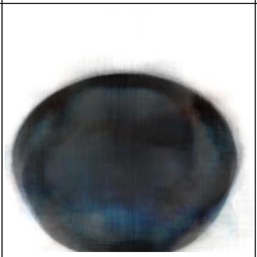
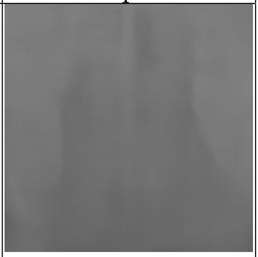
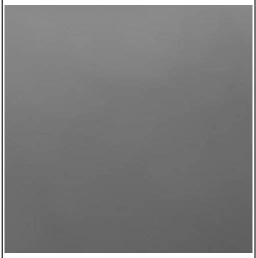
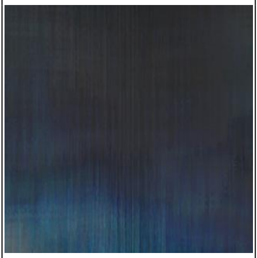
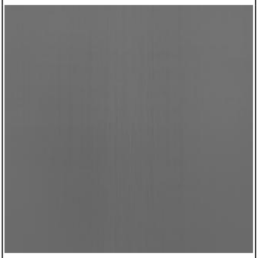
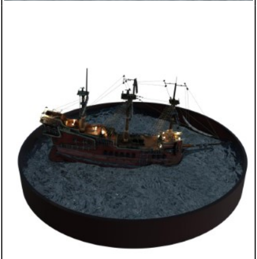
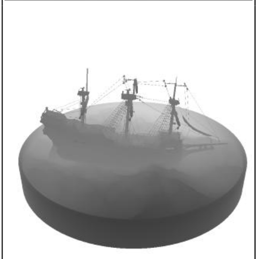
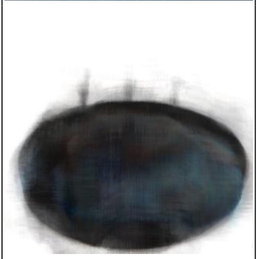
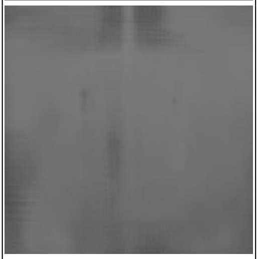
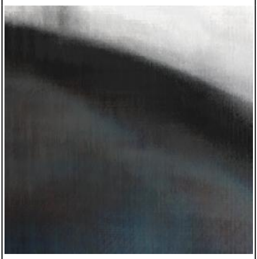
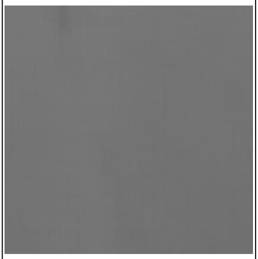
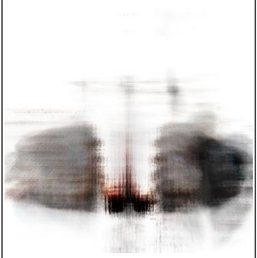
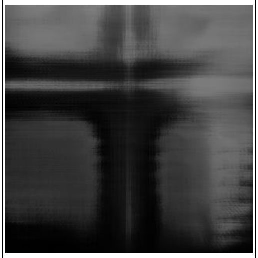
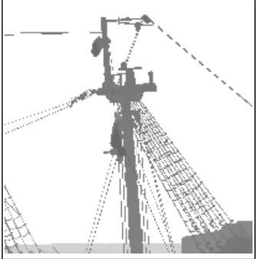
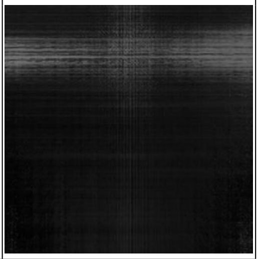
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.15: Depth-supervised mip-NeRF renderings of the synthetic dataset with the original DS-NeRF loss function, $\lambda_d = 0.1$, 25000 iterations

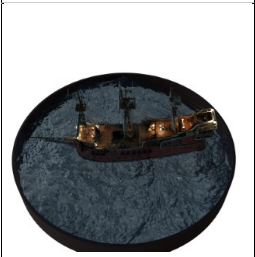
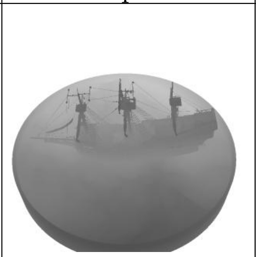
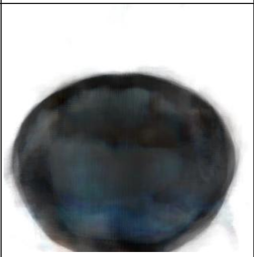
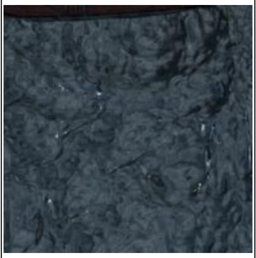
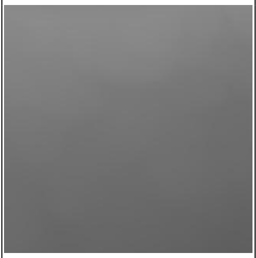
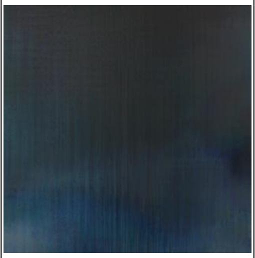
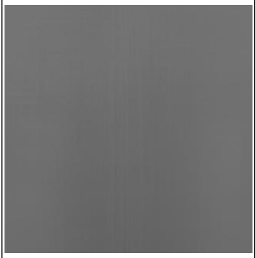
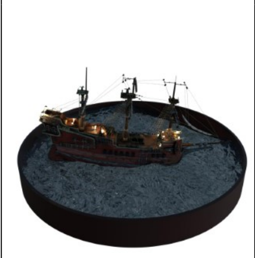
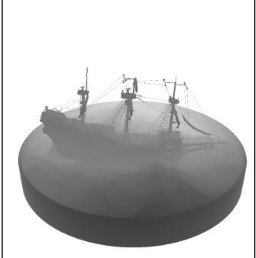
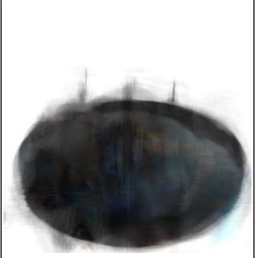
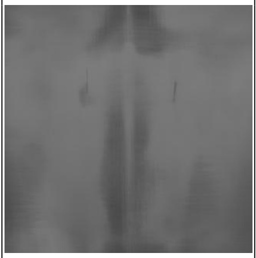
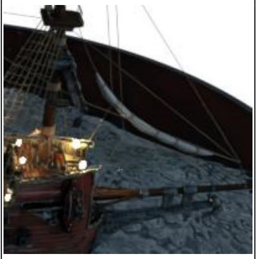
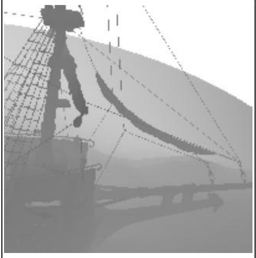
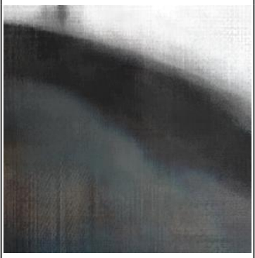
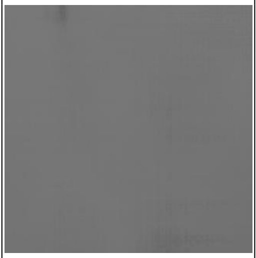
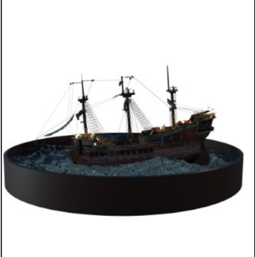
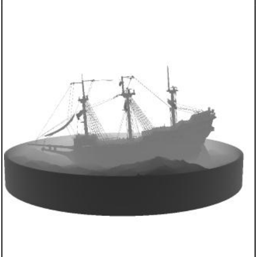
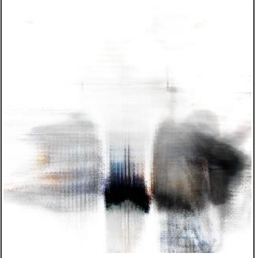
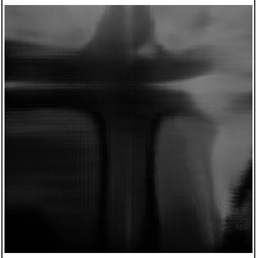
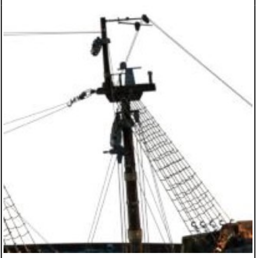
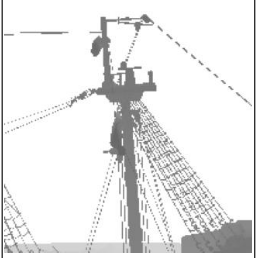
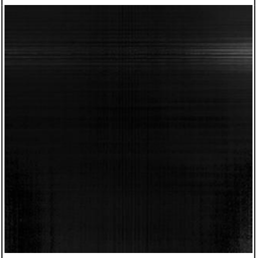
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.16: Depth-supervised mip-NeRF renderings of the synthetic dataset with the original DS-NeRF loss function, $\lambda_d = 0.1$, 100000 iterations

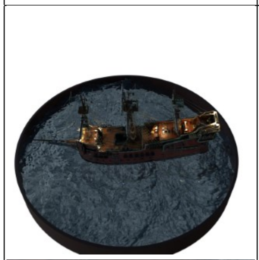
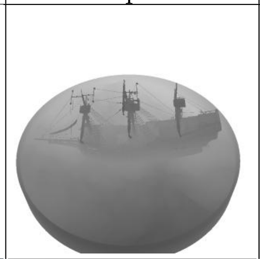
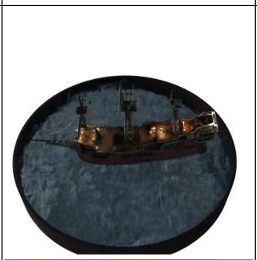
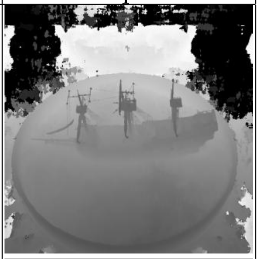
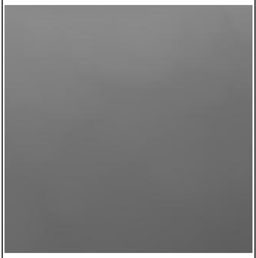
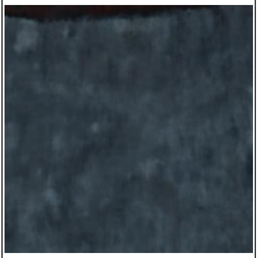
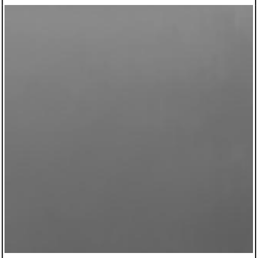
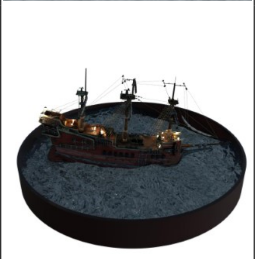
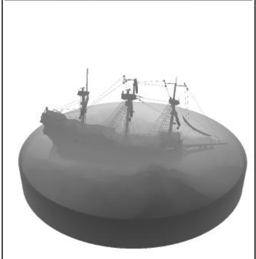
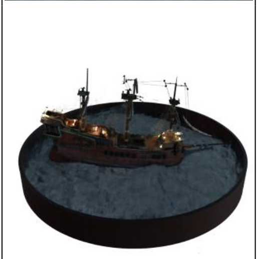
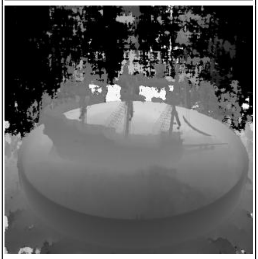
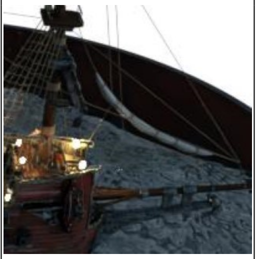
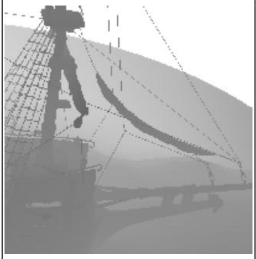
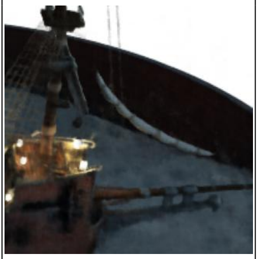
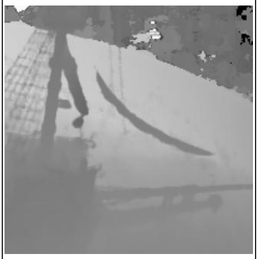
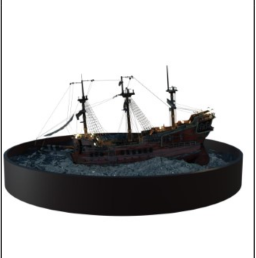
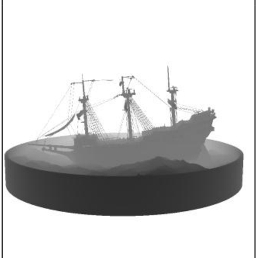
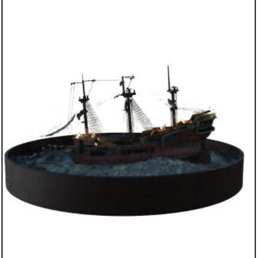
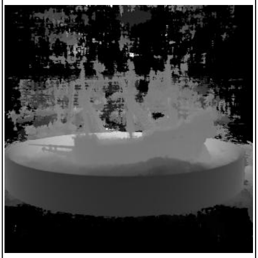
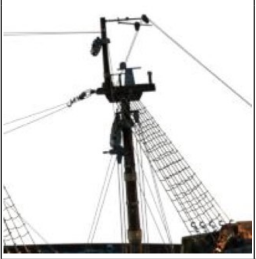
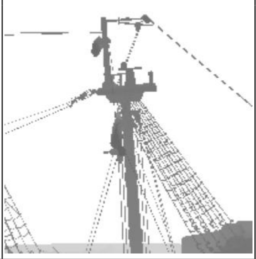
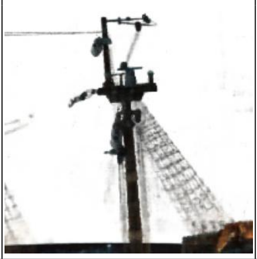
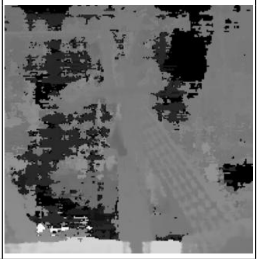
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.17: Depth-supervised mip-NeRF renderings of the synthetic dataset with the derived KL divergence loss function, $\lambda_d = 0.01$, 25000 iterations

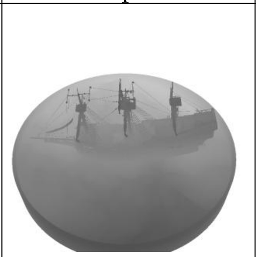
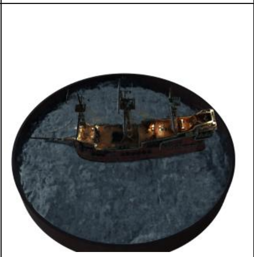
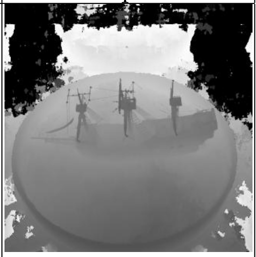
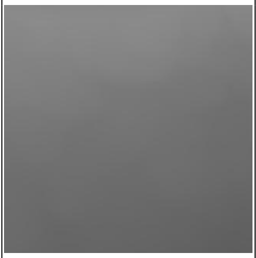
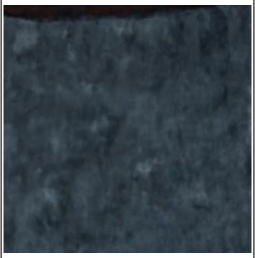
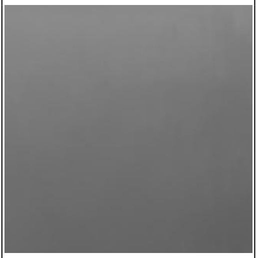
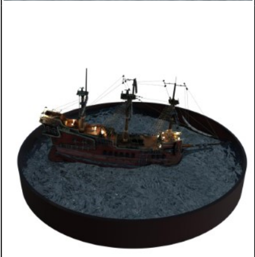
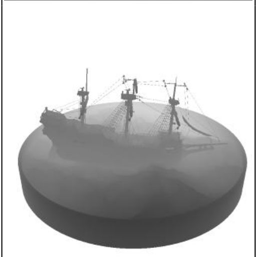
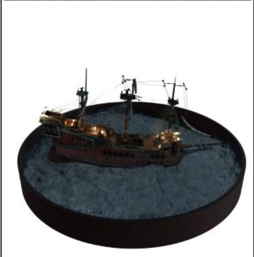
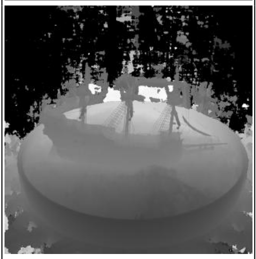
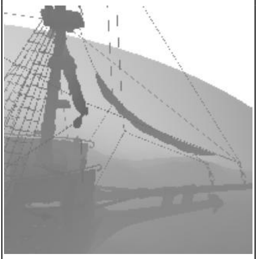
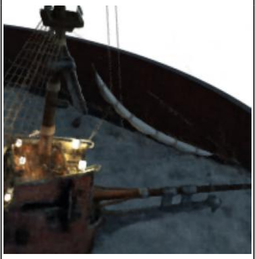
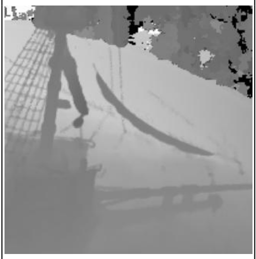
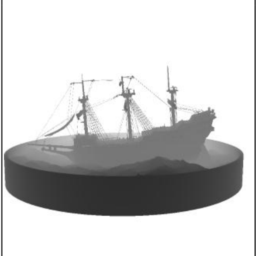
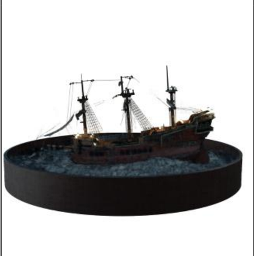
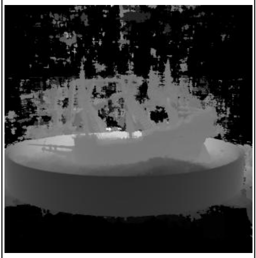
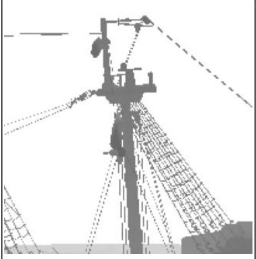
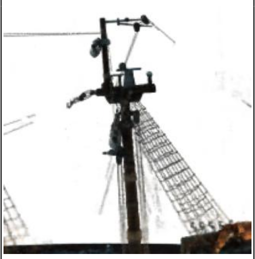
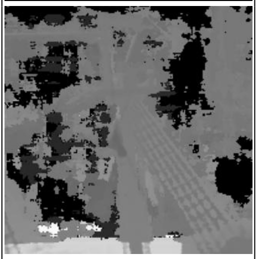
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.18: Depth-supervised mip-NeRF renderings of the synthetic dataset with the derived KL divergence loss function, $\lambda_d = 0.01$, 100000 iterations

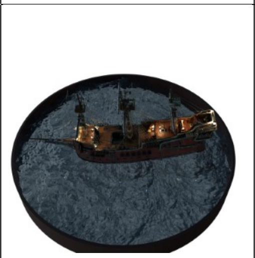
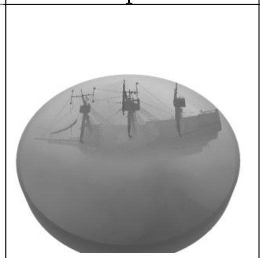
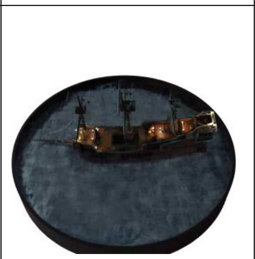
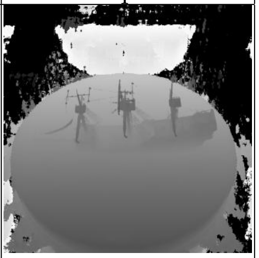
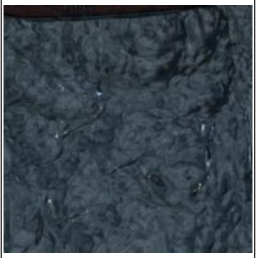
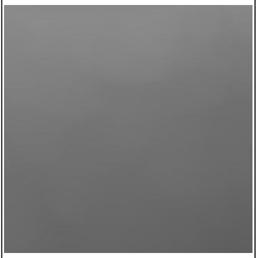
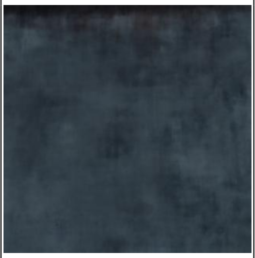
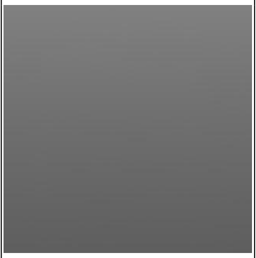
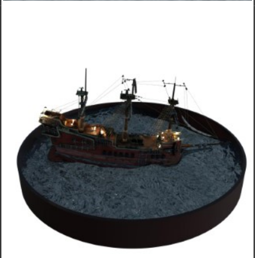
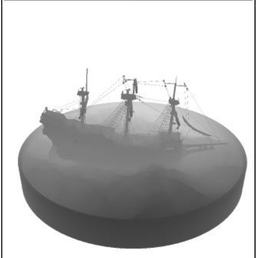
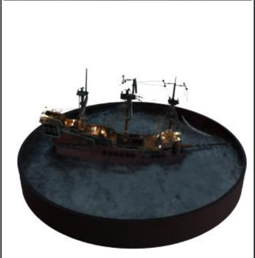
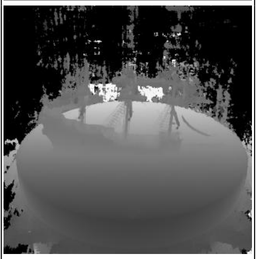
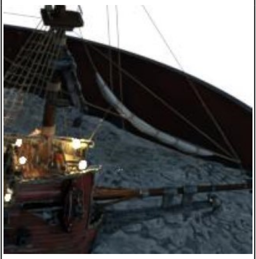
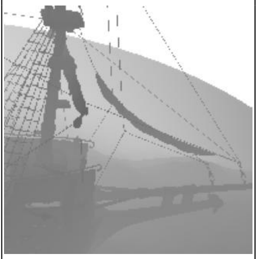
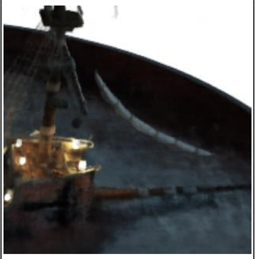
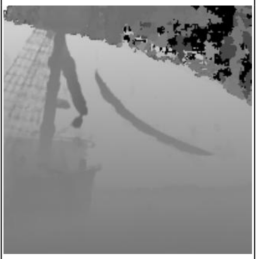
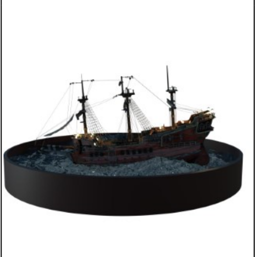
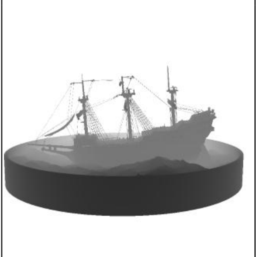
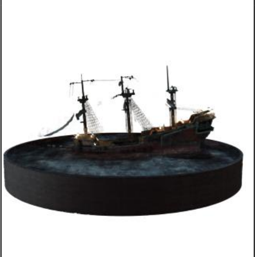
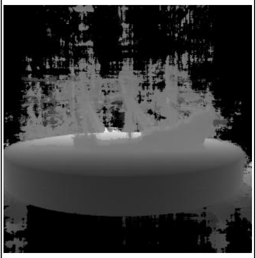
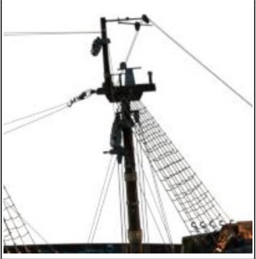
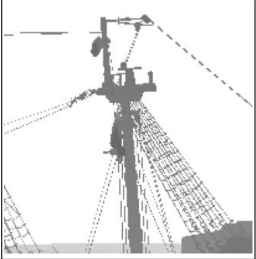
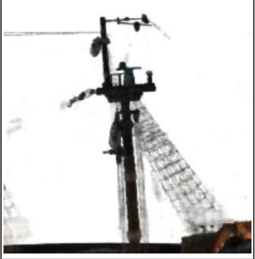
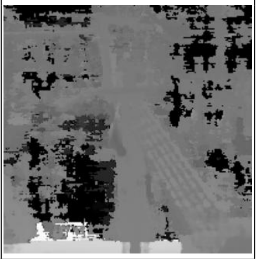
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.19: Depth-supervised mip-NeRF renderings of the synthetic dataset with the derived KL divergence loss function, $\lambda_d = 0.1$, 25000 iterations

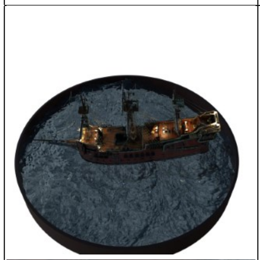
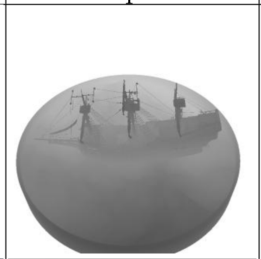
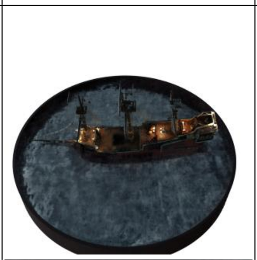
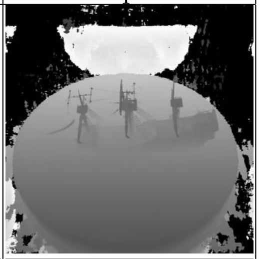
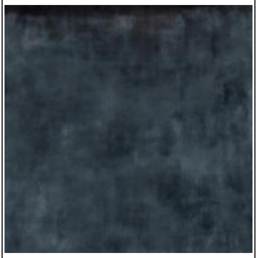
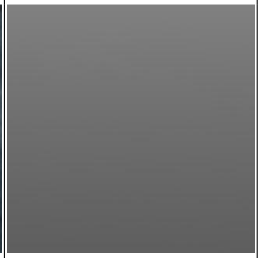
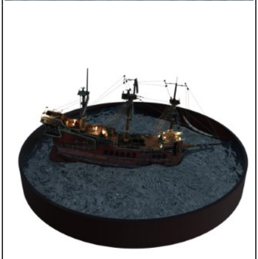
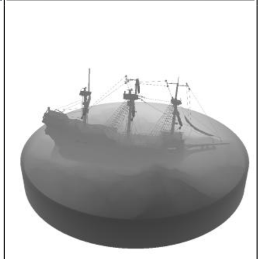
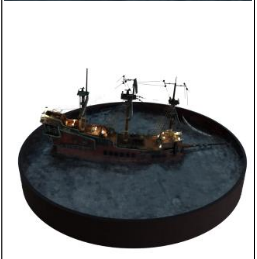
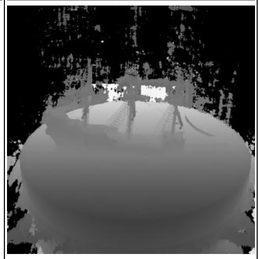
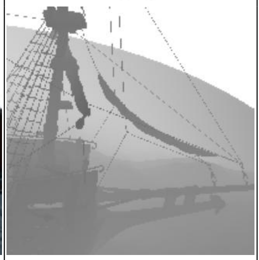
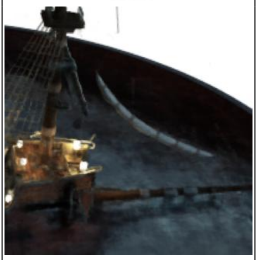
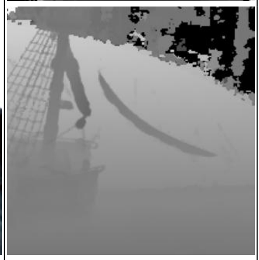
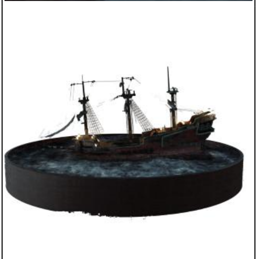
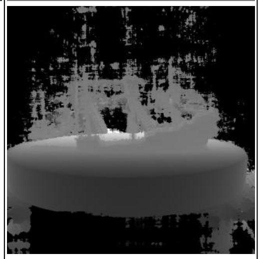
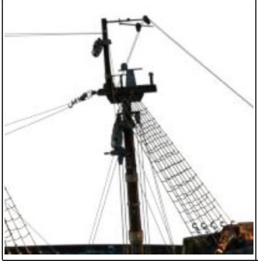
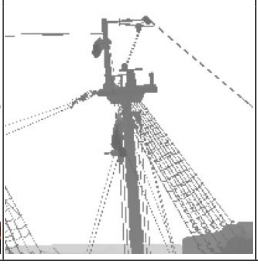
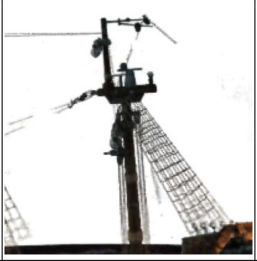
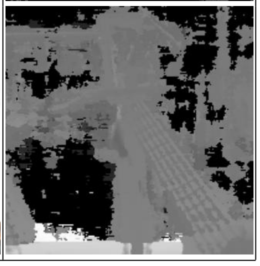
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.20: Depth-supervised mip-NeRF renderings of the synthetic dataset with the derived KL divergence loss function, $\lambda_d = 0.1$, 100000 iterations

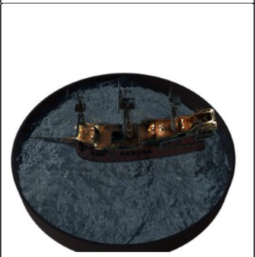
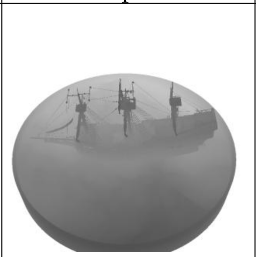
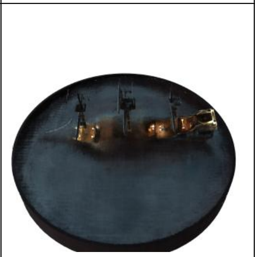
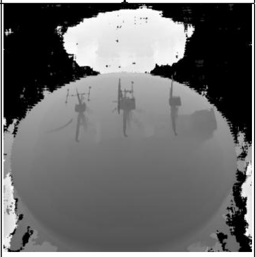
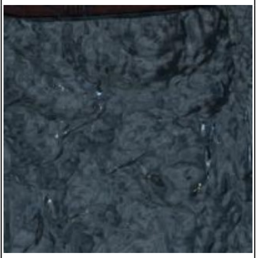
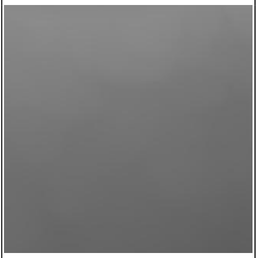
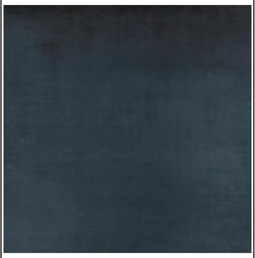
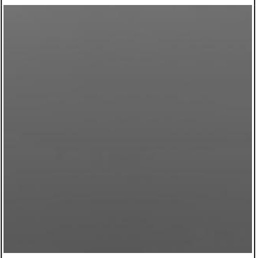
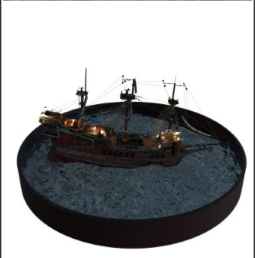
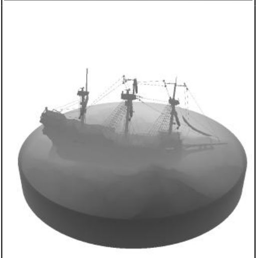
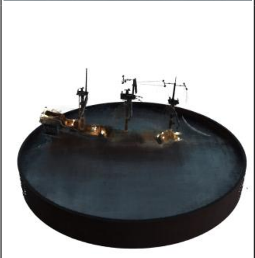
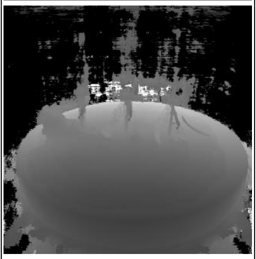
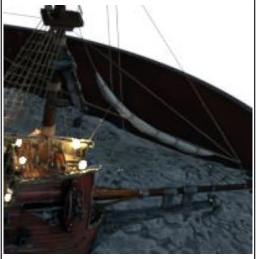
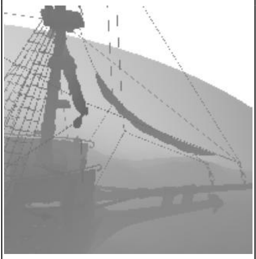
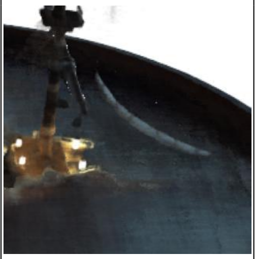
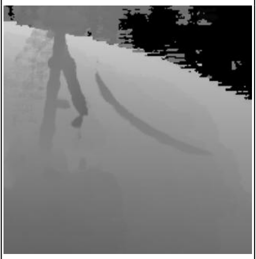
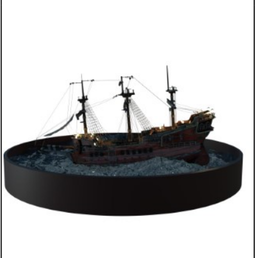
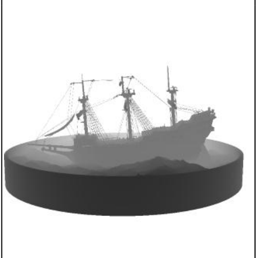
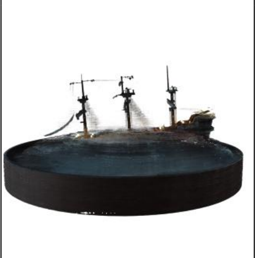
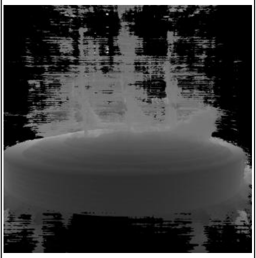
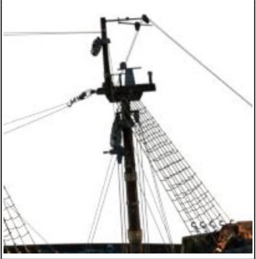
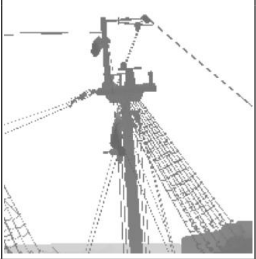
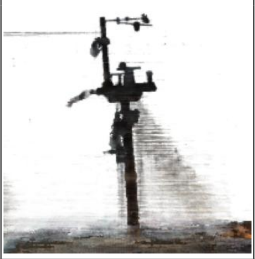
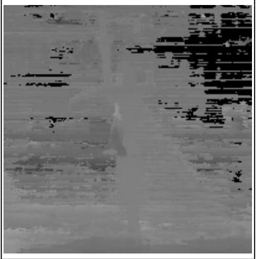
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.21: Depth-supervised mip-NeRF renderings of the synthetic dataset with the derived KL divergence loss function, $\lambda_d = 1.0$, 25000 iterations

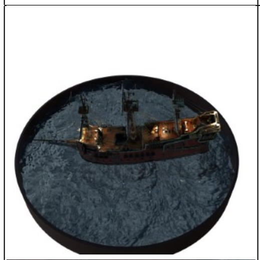
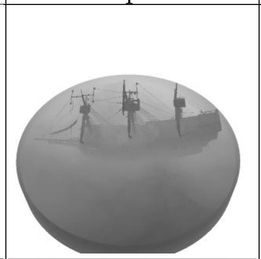
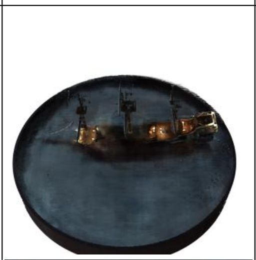
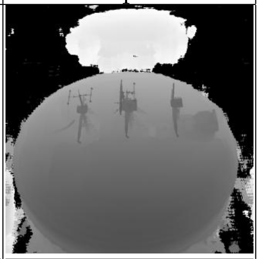
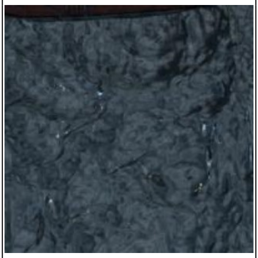
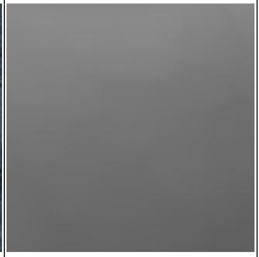
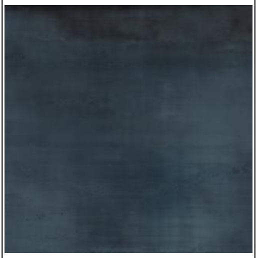
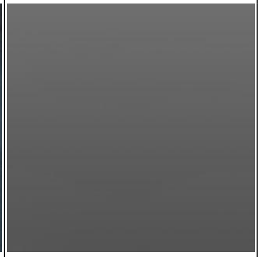
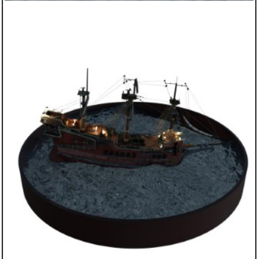
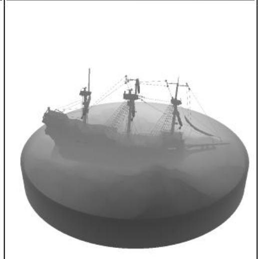
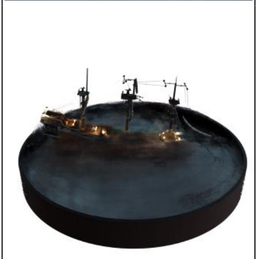
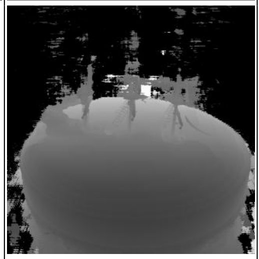
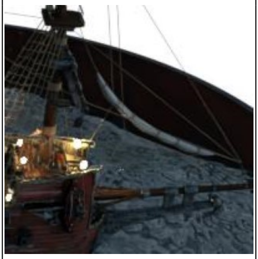
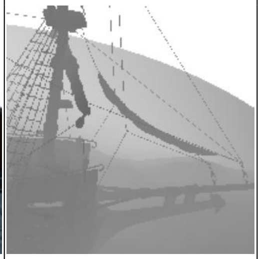
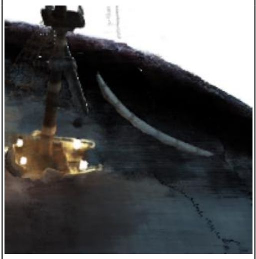
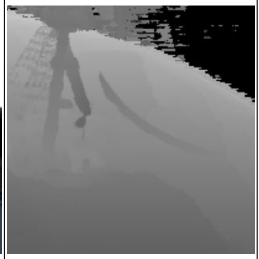
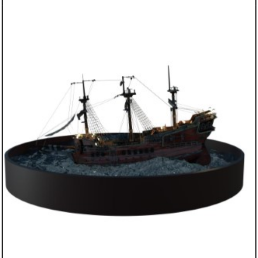
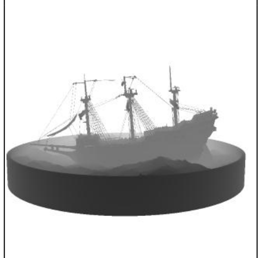
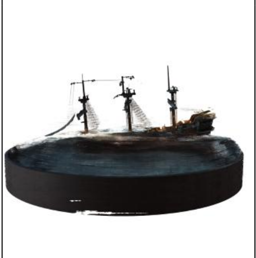
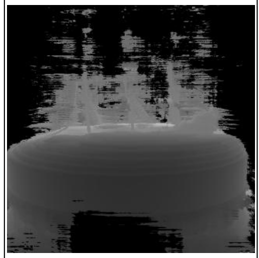
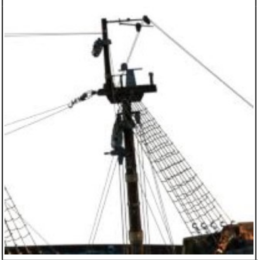
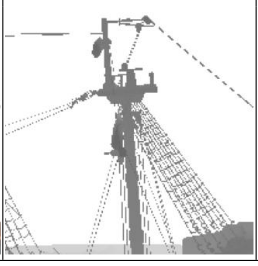
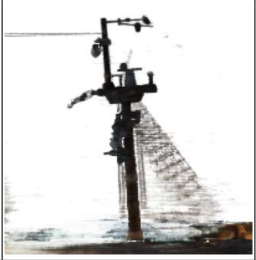
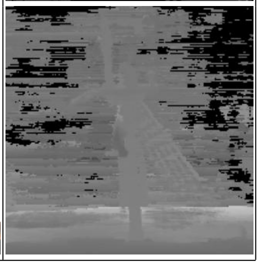
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.22: Depth-supervised mip-NeRF renderings of the synthetic dataset with the derived KL divergence loss function, $\lambda_d = 1.0$, 100000 iterations

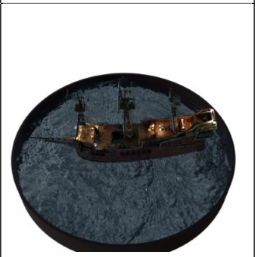
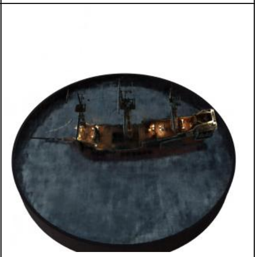
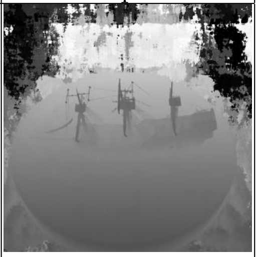
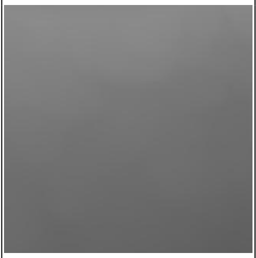
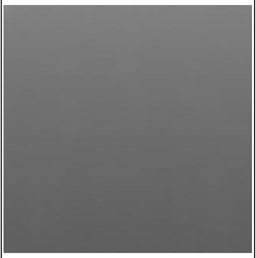
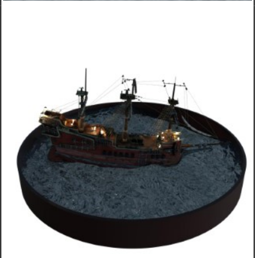
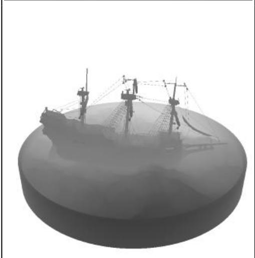
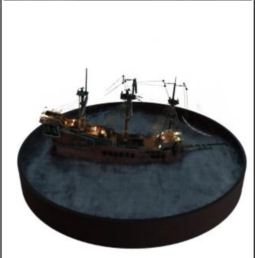
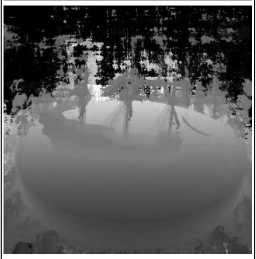
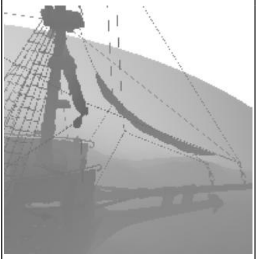
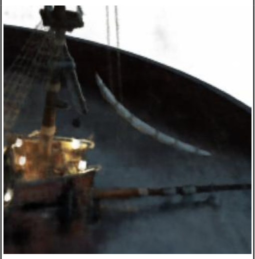
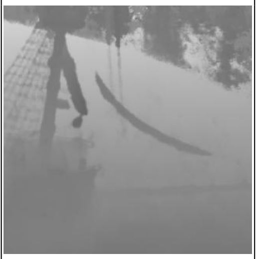
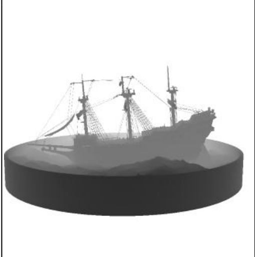
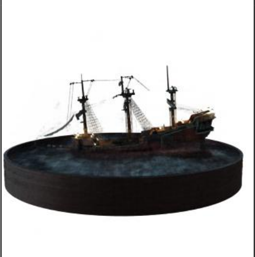
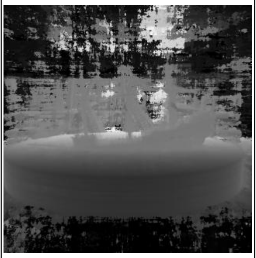
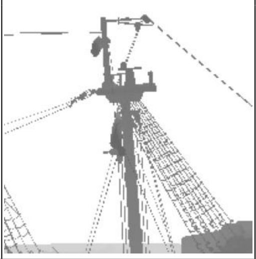
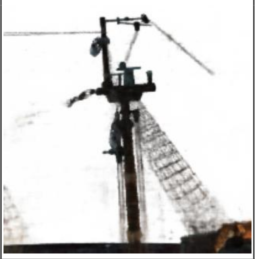
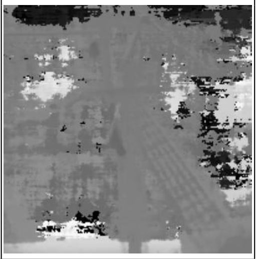
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.23: Depth-supervised mip-NeRF renderings of the synthetic dataset with the derived KL divergence loss function, $\lambda_d = 1.0$ with decay rate $\xi_d = 0.999$, 25000 iterations

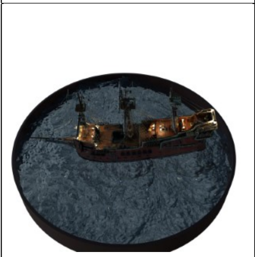
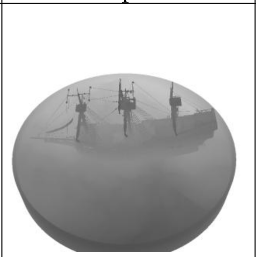
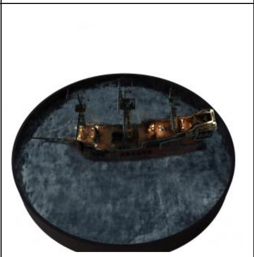
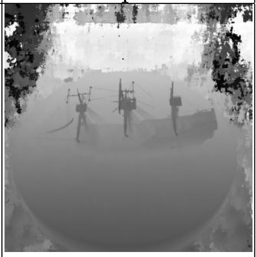
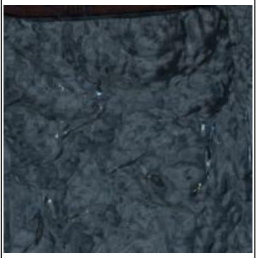
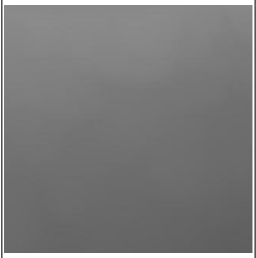
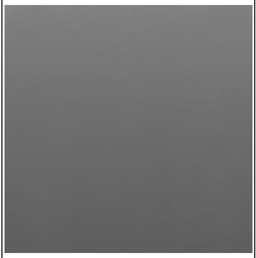
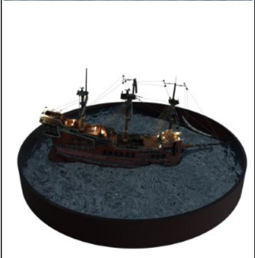
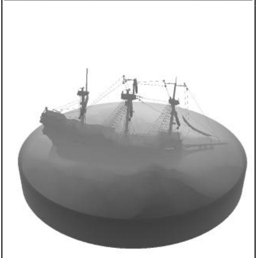
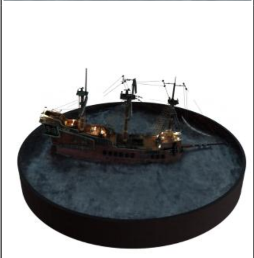
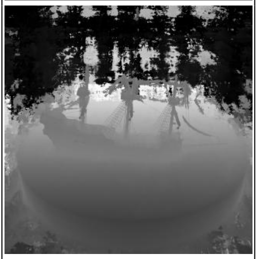
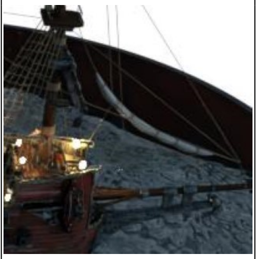
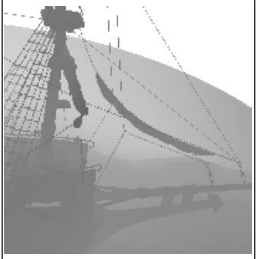
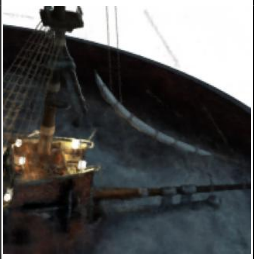
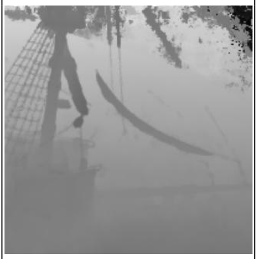
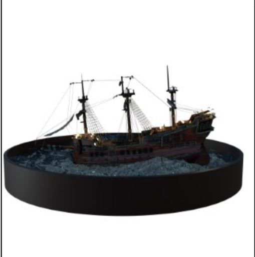
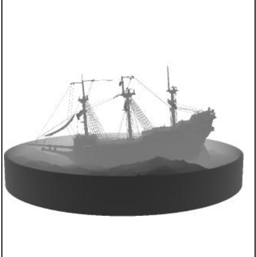
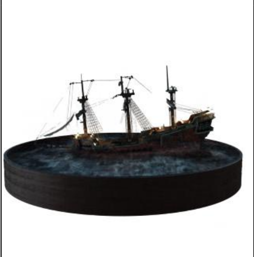
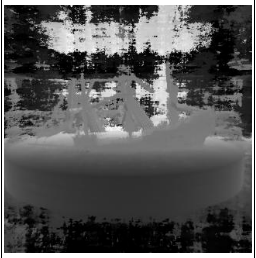
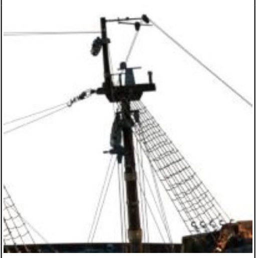
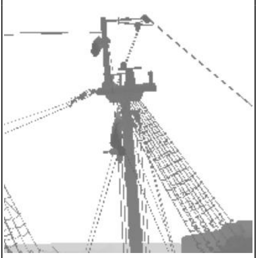
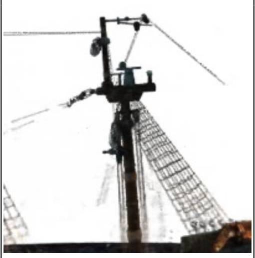
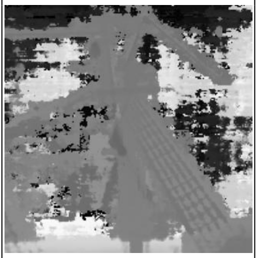
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.24: Depth-supervised mip-NeRF renderings of the synthetic dataset with the derived KL divergence loss function, $\lambda_d = 1.0$ with decay rate $\xi_d = 0.999$, 100000 iterations

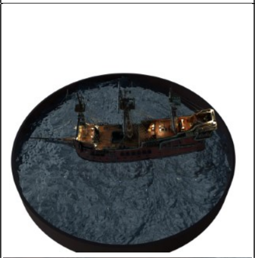
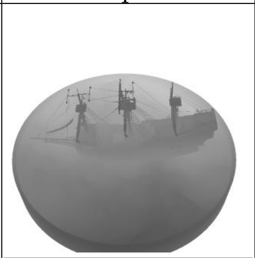
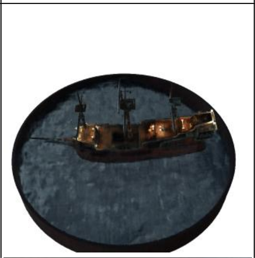
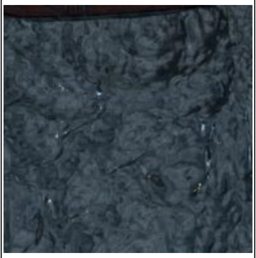
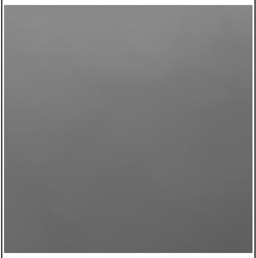
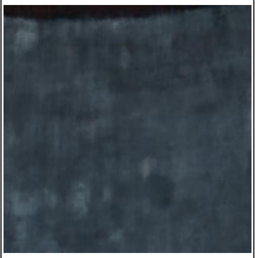
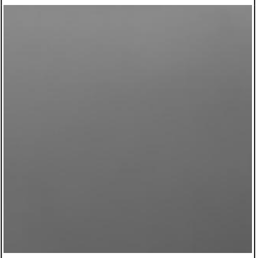
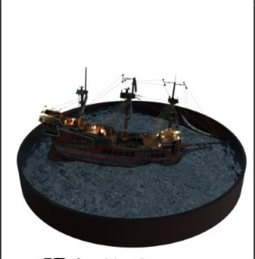
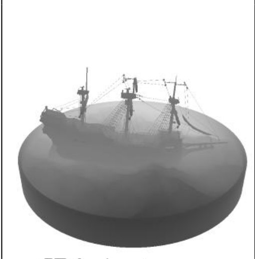
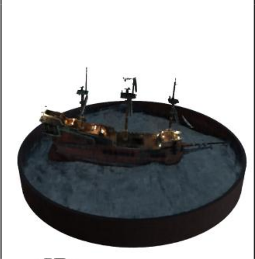
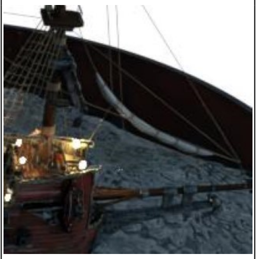
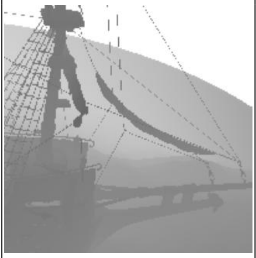
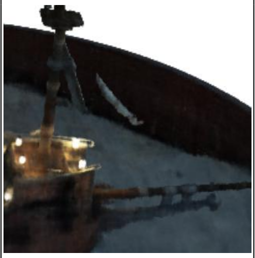
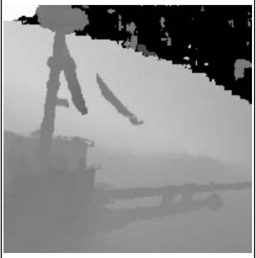
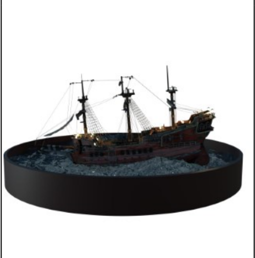
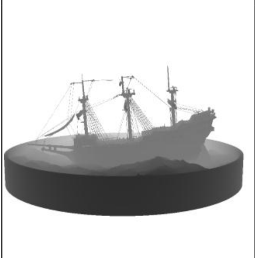
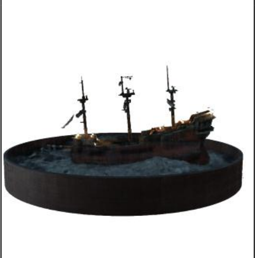
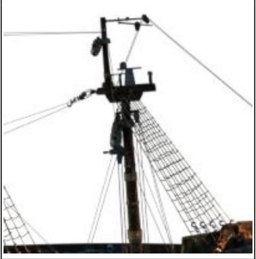
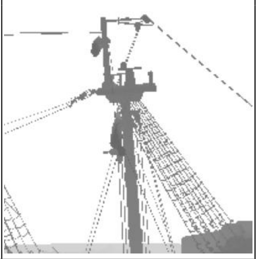
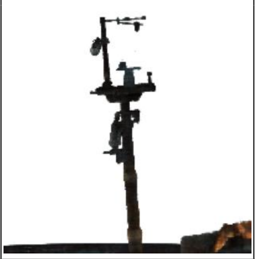
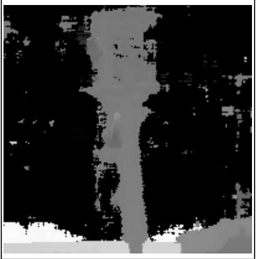
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.25: Depth-supervised mip-NeRF renderings of the synthetic dataset with the mean square loss function, $\lambda_d = 0.1$, 25000 iterations

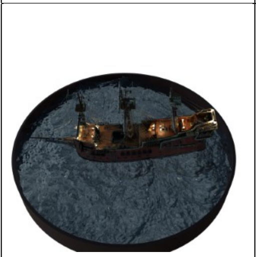
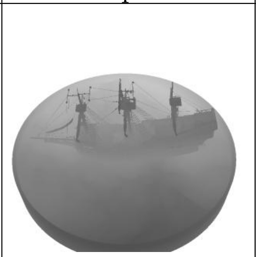
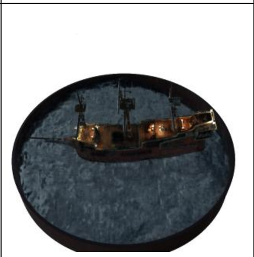
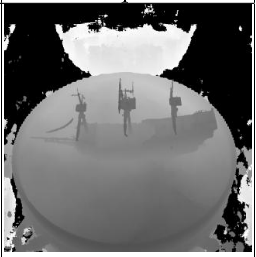
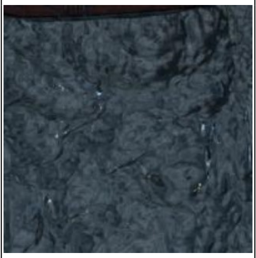
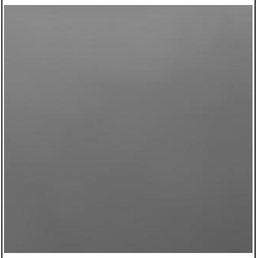
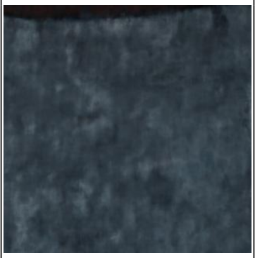
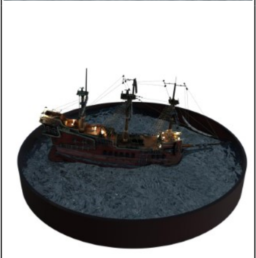
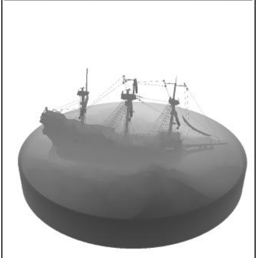
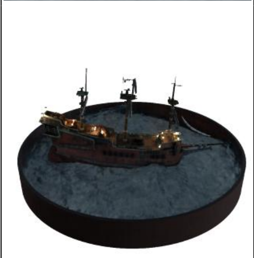
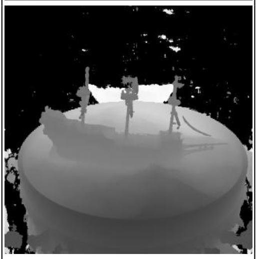
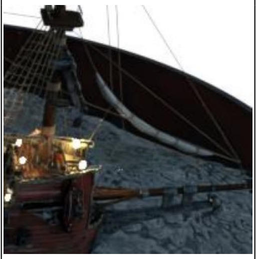
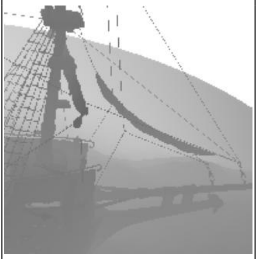
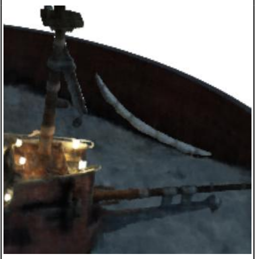
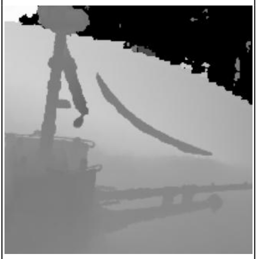
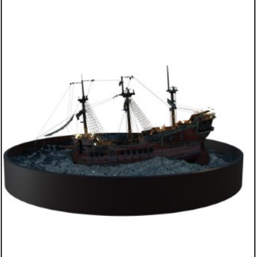
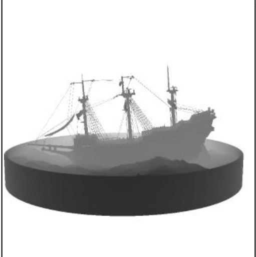
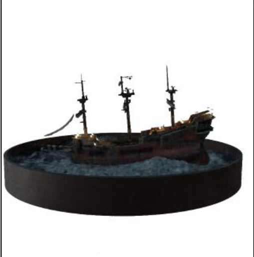
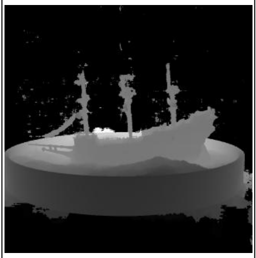
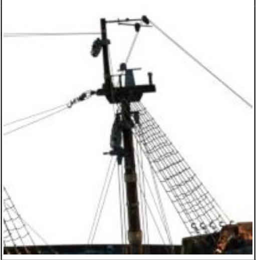
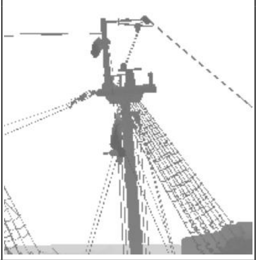
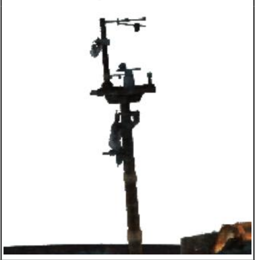
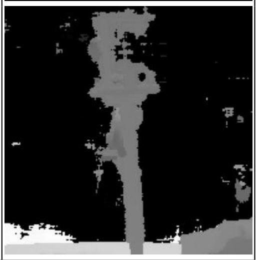
GT color	Recorded depth	Rendered color	Rendered depth
			
			
			
			
			
			

Table A.26: Depth-supervised mip-NeRF renderings of the synthetic dataset with the mean square loss function, $\lambda_d = 0.1$, 100000 iterations

Appendix B
List of Attachments

This chapter contains the list of all attachments that have been uploaded along with the electronic version of this thesis.

- **results.txt**
  A text file that contains a link to high-resolution versions of images that were presented in this thesis with depth maps preserved in their original format.
- **datasets.txt**
  A text file that contains a link to used datasets, which are stored on google drive.
- **checkpoints.txt**
  A text file that contains a link to checkpoints generated during the experiments, which are stored on google drive.
- **code.txt**
  A text file that contains a link to the GitHub repository with source code.