HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Master Thesis

in the field of

Media Informatics

Neural Radiance Fields in the context of the Industrial Metaverse

supervised by: Prof. Dr. Uwe Hahne

cosupervised by: Dr. Jakob Lindinger

submitted on: 10.10.2023

submitted by: Sabine Schleise
matriculation number: 2701169
Wilhelmstr. 4, 78120 Furtwangen
sabine.schleise@hs-furtwangen.de

Accompanying Website

As part of this thesis, an accompanying website has been developed to supplement the content presented in this document. The website hosts various videos and animations that provide a dynamic understanding of certain concepts which might be hard to grasp through text and static figures alone. Wherever applicable, interactive demos or simulations are provided, enabling users to engage with the content and experiment in a hands-on manner.

For those reading this in print or in a non-interactive format, the website can be accessed at `https://sabinecelina.github.io/masterthesis-nerf_mv/`. I encourage all readers to explore the website to gain a richer and more comprehensive understanding of the topics discussed in this document.

Abstract

With the advancement of virtual and augmented reality technologies, the term "Metaverse" is becoming increasingly familiar. The Metaverse represents a convergent virtual world for interaction, work, and entertainment. Especially in the industrial sector, the Metaverse opens up new possibilities for training, product development and collaboration. One of the key elements of the virtual world is the ability to display 3D scenes. Traditionally, triangle meshes with textures, point clouds, volumetric grids, and implicit surface functions have been the preferred methods for $3D$ scene representation because of their clear structure and fast GPU/CUDA compatibility (Kerbl et al., 2023; Tewari et al., 2020). However, a new method called Neural Radiance Fields has recently emerged (Mildenhall et al., 2020). Neural Radiance Fields (NeRF) leverages continuous scene representations, typically employing a Multi-Layer Perceptron (MLP) optimized through volumetric ray-marching for the synthesis of novel views of captured scenes. Instead of directly reconstructing the complete $3D$ scene geometry, NeRF generates a volumetric representation called a "radiance field," which is capable of creating color and density at every point within the relevant $3D$ space.

However, using existing framework to generate a neural radiance field from real world data requires developer expertise and considerable effort, which significantly limits their adoption in industry. Therefore, we introduce a NeRF trainer to bridge this gap. This NeRF trainer implementation reduces the complexity of creating neural radiance fields by providing users a user-friendly tool to construct their own 3D representations. We show the potential of the NeRF algorithm in generating high-fidelity $3D$ scenes from $2D$ images and how it can advance the Metaverse in the industrial context.

Zusammenfassung

Mit der Weiterentwicklung der Technologien für virtuelle und erweiterte Realität wird der Begriff "Metaverse" immer vertrauter. Das Metaverse stellt eine konvergente virtuelle Welt für Interaktion, Arbeit und Unterhaltung dar. Besonders im industriellen Bereich eröffnet das Metaverse neue Möglichkeiten für Training, Produktentwicklung und Zusammenarbeit. Eines der Schlüsselelemente der virtuellen Welt ist die Möglichkeit, $3D$-Szenen darzustellen. Traditionell waren Meshes mit Texturen, Punktwolken, volumetrische Gitter und implizite Oberflächenfunktionen aufgrund ihrer klaren Struktur und schnellen GPU/CUDA-Kompatibilität die bevorzugten Methoden zur Darstellung von $3D$-Szenen (Kerbl et al., 2023; Tewari et al., 2020). Vor kurzem wurde jedoch eine neue Methode namens Neural Radiance Fields vorgestellt (Mildenhall et al., 2020). NeRF nutzt kontinuierliche Szenendarstellungen, wobei typischerweise ein Multi-Layer Perceptron (MLP) eingesetzt wird, das durch volumetrisches Ray-Marching für die Synthese neuartiger Ansichten erfasster Szenen optimiert wird. Anstatt die komplette $3D$-Szenengeometrie direkt zu rekonstruieren, generiert NeRF eine volumetrische Darstellung, ein sogenanntes "Strahlungsfeld", das in der Lage ist, Farbe und Dichte an jedem Punkt im relevanten $3D$-Raum zu erzeugen.

Die Verwendung bestehender Frameworks zur Generierung eines neuronalen Strahlungsfeldes aus echten Daten erfordert jedoch das Fachwissen von Entwicklern und einen beträchtlichen Aufwand, was deren Einsatz in der Industrie erheblich einschränkt. Daher führen wir einen NeRF-Trainer ein, um diese Lücke zu schließen. Diese NeRF-Trainer-Implementierung reduziert die Komplexität der Erstellung neuronaler Strahlungsfelder, indem sie den Benutzern ein benutzerfreundliches Werkzeug zur Verfügung stellt, mit dem sie ihre eigenen $3D$-Darstellungen konstruieren können. Wir zeigen das Potenzial des NeRF-Algorithmus mit der Erzeugung von $3D$Szenen mit hoher Wiedergabetreue aus $2D$Bildern und wie er das Metaverse im industriellen Kontext voranbringen kann.

Contents

List of Figures

List of Tables

List of Code Listings

List of Abbreviations

GPU Graphics Processing Unit

NeRF Neural Radiance Fields

instant-ngp Instant Neural Graphics Primitives

MLP Multilayer Perceptron

GUI graphical user interface

CUDA Compute Unified Device Architecture

mip-NeRF A Multiscale Representation for Anti-Aliasing Neural Radiance Fields

SSIM structural similarity

VR Virtual Reality

AR Augmented Reality

XR Extended Reality

MR Mixed Reality

IPE integrated positional encoding

PE Positional Encoding

USD Universal Scene Description

SfM Structure-from-Motion

SIFT Scale Invariant Feature Transform

BA Bundle Adjustment

SDF Signed Distance Function

PyPI Python Package Index

CI continuous integration

CD continuous delivery

CG computer graphics

SSAN Signed Surface Approximation Network

TSDF Truncated Signed Distance Field

IOT Internet of Things

AI Artificial Intelligence

1. Introduction

In this thesis we investigate the ability of the NeRF algorithm to convert $2D$ images into $3D$ scenes, particularly in the context of the Industrial Metaverse. We introduce a user-friendly framework to enable users without a technical background to participate in the creation of a neural radiance field with their own data. The NeRF Trainer framework allows users to explore and experiment with scene representations that are derived from neural radiance fields.

1.1. Evolution of capturing the visual reality

The concept of recording visual reality deals with the process of representing the real world at a particular point in time. Since the earliest cave paintings, situations and experiences have been recorded for posterity. Throughout history, various techniques and technologies have been developed to provide more accurate representations (Reisig and Freytag, 2006).

The camera obscura laid the groundwork for advanced cameras with accurate representations of the real world. In 1850, the first stereophotographs were presented, making it possible for the first time to create a $3D$ impression in images. Stereography, which uses two slightly offset images, provides a sense of depth by presenting each eye with a slightly different perspective, mimicking the way our two eyes perceive depth in reality. Beyond stereography, another method called photogrammetry emerged to extract $3D$ data from $2D$ images. Derived from the Greek words "photos" (light), "gramma" (something written or drawn), and "metron" (measure), photogrammetry focuses on capturing and processing information from images. Its main objective is to determine the shape, size, and spatial positioning of objects, using multiple images from various viewpoints to create precise $3D$ reconstructions (Reisig and Freytag, 2006).

Photogrammetry is not without its challenges. One notable issue is the handling of transparent or reflective objects. These materials can lead to inaccurate reconstructions due to their unique light interaction properties. In addition, the process requires specialized software and considerable time, making it less accessible to those without technical expertise (Reisig and Freytag, 2006). Traditional photogrammetry focuses on reconstructing the geometric attributes of objects, neglecting the nuanced lighting and color information that radiance fields encapsulate. It is in this gap that the NeRF

algorithm has made a transformative difference (Mildenhall et al., 2020; Müller et al., 2022). Unlike traditional methods, NeRF generates a volumetric representation called a "radiance field". This field assigns color and density attributes to each point in $3D$ space, providing a more complete understanding of how light interacts with objects and scenes. This enhanced representation contributes to more realistic and visually appealing reconstructions. Currently, however, creating neural radiance fields is a complex process that requires developer expertise and specialized tools. This has resulted in their being accessible only to those with technical expertise. Given this barrier, this thesis presents an implementation of a NeRF framework. The aim of this framework is to make this advanced approach accessible to users regardless of their technical background. It allows individuals to engage with NeRF technology without being hindered by the complexities normally associated with its usage.

1.2. NeRF in the context of the Industrial Metaverse

The Industrial Metaverse represents the convergence of virtual and physical reality in industrial contexts, opening up new opportunities for the application of virtual technologies in manufacturing, product development, sales and training to create more efficient and collaborative workflows. In the Industrial Metaverse, companies can create virtual environments where employees, customers and partners can interact. These virtual environments can be digital representations of manufacturing facilities, virtual training environments, or virtual product development labs. Virtual Reality (VR) is a powerful virtual environment tool that offers remarkable levels of immersion by placing the user in a completely virtual environment. This immersive encounter actually gives the feeling of being in a different, tangible virtual setting. An important aspect of this concept is the creation of realistic $3D$ models, product environments and scenarios tailored to various industrial use cases.

However, as previously described, the current process for creating these complex $3D$ scenes is challenging. It requires modelling the scenes or photogrammetry with manual post-processing. To overcome this challenge, NeRF has emerged as a promising solution. Using neural network based algorithms, NeRF enables the generation of complex $3D$ scenes and objects from $2D$ images. This technique has the potential to improve the efficiency of converting $2D$ information into accurate and detailed $3D$ representations. While the theoretical potential of NeRF is considerable, its practical application in the industrial sector is still in its early stages. Currently, because of its novelty, its exploration is mostly limited to academics, developers and tech enthusiasts. This thesis aspires to address this oversight. The objective is not only to integrate NeRF into the industrial context, but also to thoroughly evaluate its possibilities. Furthermore, this thesis aims to explore the various applications and

implications of this innovative approach in the industrial domain. The aim of this thesis is to provide a comprehensive assessment of the potential role of NeRF in the advancement of industrial processes and methods, particularly in the context of the Industrial Metaverse.

1.3. Outline

Chapter 2 lays the theoretical foundation, touching on the elementary components of early vision to understand the concept of reality capture. We then go on to explain the workings of the NeRF algorithm, highlighting its subsequent advancements designed to address challenges and limitations. The focus of this chapter is on the instant-ngp algorithm, a recent improvement to the original NeRF algorithm, which is used in our thesis to generate NeRFs. Additionally, we discuss other research initiatives in the NeRF framework, highlighting the growing academic interest in this area.

Chapter 3 introduces existing frameworks that are capable of training, rendering and storing NeRF models. We evaluate the accessibility and intuitiveness of these frameworks and investigate how neural radiance fields can be created from user-generated data within these frameworks.

Then, in chapter 4, we explain the motivations for developing a custom framework. We dive deeper into the "NeRF-Trainer" implementation, which is accessible via a simple web application. This platform facilitates the training of neural radiance fields, requiring only the upload of user-generated data.

NeRF has a wide range of potential applications, especially in the context of the Industrial Metaverse. In Chapter 5, we explore the diverse applications of NeRF in the Industrial Metaverse, particularly where traditional approaches like photogrammetry fall short, such as in avatar creation, mesh generation or in rendering difficult material properties. However, we also identify limitations the algorithm has and present research that is attempting to overcome these challenges.

The final chapter 6 provides a concise summary and discussion of the thesis, as well as an outlook on potential future projects.

2. Basics

Computer Vision has made significant advances by using mathematical functions parameterized by the weights of a MLP to represent computer graphics primitives, which parameterize physical properties of scenes or objects across space and time (Tewari et al., 2020; Xie et al., 2021). This chapter explores the NeRF algorithm, an approach that uses MLPs to regress a $5D$ input coordinate to a volume density and view-dependent color output. By querying the neural network for that output information, the algorithm uses classical rendering techniques to synthesize novel views. Before going into more detail on this method, we pick up on the idea of capturing the reality by introducing the plenoptic function. Then, in section 2.2, we explore the fundamentals of the NeRF approach and how it leverages the power of neural networks to model volumetric scenes, enabling the generation of high-quality $3D$ representations from $2D$ images. The NeRF method creates the volumetric scene representation based on input images and camera poses, which can lead to undersampling and aliasing artifacts. Furthermore, the training process is slow, and rendering can take a long time (Rabby and Zhang, 2023). We will therefore turn our attention to the instant-ngp (Müller et al., 2022) algorithm, which forms the basis of this thesis. This algorithm adeptly addresses the issues of training efficiency and rendering speed. Finally, we will present different definitions of the Metaverse in order to identify the main challenges and potential of using the NeRF algorithm in the context of the Industrial Metaverse.

2.1. Capturing Reality: The Elements of Early Vision

The question of the basic elements of early vision deals with the fundamental substances that underlie visual perception. In this context, the focus is less on discrete objects such as simple edges and corners, but rather on the essential building blocks of the visual process. The first stage of visual processing comprises a series of parallel pathways, each dedicated to a specific visual property, such as motion, color, orientation, and binocular disparity. These fundamental attributes are regarded as measurable units of early vision, with the focus on how the visual system quantifies the characteristics of objects, rather than how it labels them. The primary goal of this approach is to systematically derive visual elements and elucidate their connection to the structure of visual information in the surrounding world. All critical visual measurements can be

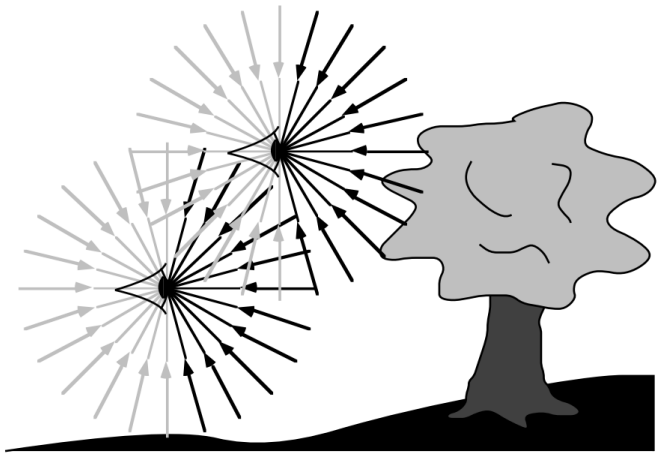


Figure 1.: The plenoptic function describes the information available to an observer at any point in space and time. This illustration depicts two schematic eyes, each with punctate pupils, capturing bundles of light rays. Notably, the plenoptic function encompasses rays originating from beyond the observer's viewpoint, which remain unseen (Adelson and Bergen, 1991).

characterized as local variations in one or more dimensions of a single function. This function describes the nature of incoming light to an observer and includes everything that has the potential for visual perception. This comprehensive function is called the "plenoptic function", derived from "plenus" meaning complete or full and "optical", which is relating to optics. The importance of the plenoptic function lies in its role as an intermediary between $3D$ physical objects and the resulting retinal images in the eye of the observer. Rather than directly imparting attributes to objects, the plenoptic function imparts a pattern of light rays to the surrounding space. Ultimately, the observer samples this function to shape their visual perception. The plenoptic function acts as a link between the outside world and the inner eye. Introduced by Adelson and Bergen (1991), the plenoptic function attempts to capture the entirety of our visual reality and the information it contains. It represents a conceptual function with a $7D$ input denoted as $P = P(V_x, V_y, V_z, \theta, \phi, \lambda, t)$, where V represents an idealized eye at every possible location, recording the intensity of the light ray passing through the center of the pupil at every possible angle (θ, ϕ), for each wavelength λ, at each time t. The plenoptic function aims to model our visual reality comprehensively, including the $3D$ structure of the scene, surface features, motion, and other important visual aspects. It serves as the foundation for visual perception, providing raw data from which the visual system constructs our perception of the world. Figure 1 illustrates the idea of the plenoptic function (Adelson and Bergen, 1991).

Early attempts to address the theoretical concept of the plenoptic function include *Light Field Rendering* (Levoy and Hanrahan, 1996) and *The Lumigraph* (Gortler et al., 1996). *Light Field Rendering* captures and represents all the light rays passing

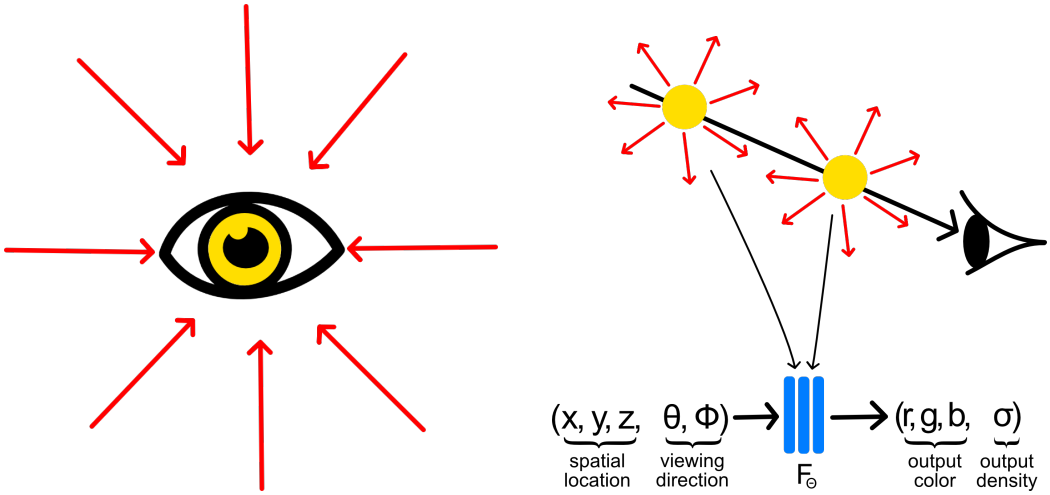
through a scene from various viewpoints and directions, providing a rich dataset that enables the reconstruction of different views and supports flexible rendering and post-processing. In contrast, *The Lumigraph* captures a set of images from a fixed set of viewpoints, allowing for the recreation of different views of the scene within the captured range. Both of these approaches represent significant advances in capturing and rendering the visual information contained within a scene, providing practical solutions for exploiting the vast amount of data encompassed by the plenoptic function. For further information about these two approaches, we refer to the paper by Levoy and Hanrahan, titled *Light Field Rendering* (Levoy and Hanrahan, 1996) and *The Lumigraph*, by Gortler et al. (Gortler et al., 1996), both presented at Siggraph in 1996.

Simplifying the plenoptic function by removing the dimensions of time and wavelength results in a static representation of the world. This simplification leads to a $5D$ function, $P = P(V_x, V_y, V_z, \phi, \theta)$, similar to the input for the network used in the NeRF algorithm. The plenoptic function models everything that comes into a single wavelength at any time and place. This is why, in theory, direct querying of this function would enable the rendering of diverse views from numerous vantage points. However, the NeRF algorithm takes a distinct approach. Instead of modeling the incoming visual data into a single eye as the plenoptic function does, it models the particles at every single point in space with their color and view-dependent radiance. This facilitates dynamic traversal through a scene, setting it apart from conventional light fields or prior methods. As a result, NeRF establishes a flexible representation, grounded in a robust underlying $3D$ structure. Each of these particles is further characterized by a function that takes into account $5D$ parameters and generates outputs for color and density, forming the core of NeRF's functionality (Kanawaza, 2023). Figure 2 visually contrasts the distinctions between the plenoptic function and the input employed by NeRF.

Through this comparison, it becomes evident that while the plenoptic function provides a comprehensive representation of visual information entering a single eye at a given moment, the NeRF approach diverges by considering each point in space along a ray (Kanawaza, 2023).

2.2. Neural Radiance Fields

NeRF is a method for synthesizing novel views of complex scenes by optimizing an underlying continuous volumetric scene function using a sparse set of 2D input images. Given a set of images capturing a static scene from multiple angles, along with their corresponding poses, the neural network learns to represent that scene in a way that allows new views to be synthesized. The NeRF algorithm represents a continuous scene



(a) Plenoptic Function (b) NeRF Function

Figure 2.: Illustration highlighting distinction between the plenoptic function and the approach employed by NeRF. While the plenoptic function aims to capture all visual information that could enter a single eye at any given moment and location, NeRF instead feeds the network with information about every point along a ray. This ray is illustrated in Figure 2b, where it is marked in black, leading into the eye, and each point along it is indicated in yellow (Kanawaza, 2023)

as a $5D$ vector-valued function whose input is a $3D$ spatial location $X = (x, y, z)$ and a $2D$ viewing direction $d = (\phi, \theta)$, and whose output is the emitted color $c = (r, g, b)$ depending on each direction at each point and an view-independent volume density σ that ranges from $[0, \infty)$. To approximate this continuous $5D$ scene representation, an MLP network F_Θ is utilized, where $F_\Theta : (x, d) \longrightarrow (c, \sigma)$. The weights Θ of the network are optimized to map each $5D$ input coordinate to its corresponding volume density and directional emitted color. Part a and b of Figure 3 provide an overview of the NeRF scene representation (Mildenhall et al., 2020). Part c and d will be explained later in this section.

2.2.1. Overview of the NeRF scene representation

The algorithm enables view synthesis through a three-step process, as illustrated in Figure 3. First, $5D$ coordinates are sampled along the camera rays and passed through an MLP to produce both color and volume density (steps a and b). Second, the output from the MLP is utilized to generate the individual pixels of an image using classical volume rendering techniques (step c). Finally, the differentiability of the rendering function is leveraged to optimize the weights of the MLP (step d). This is achieved by minimizing the loss function, which is simply defined as the total squared error

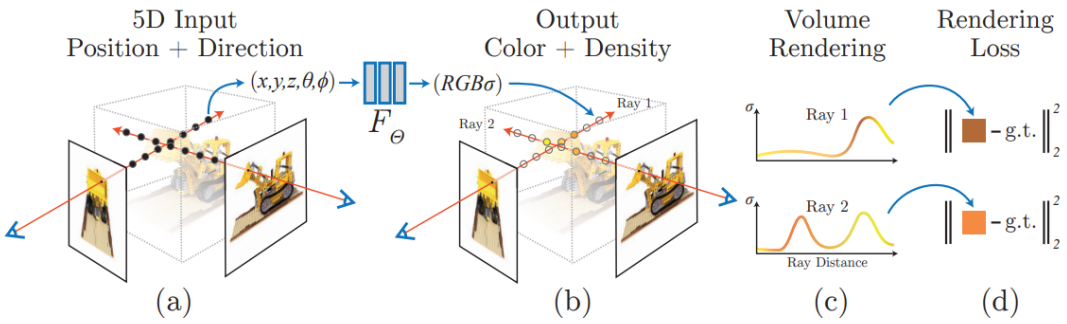


Figure 3.: An overview of the neural radiance field scene representation and differentiable rendering procedure. The synthesis of images is accomplished by sampling 5D coordinates, encompassing location and viewing direction, along camera rays (a). These sampled locations are then fed into a MLP to generate color and volume density values (b). By employing volume rendering techniques, these values are composited to create the final image (c). The rendering function is differentiable, enabling the optimization of the scene representation by minimizing the residual between the synthesized images and the observed ground truth images (d) (Mildenhall et al., 2020)

between the rendered and true pixel colors for both the coarse and fine rendering:

$$\mathcal{L} = \sum_{r \in \mathcal{R}} \left(\hat{C}_c(r) - C_{\text{gt}}(r) \right)^2 + \left(\hat{C}_f(r) - C_{\text{gt}}(r) \right)^2 \tag{2.1}$$

where $\mathcal{R}$ represents the set of rays in each batch and $C_{\text{gt}}(r)$, $\hat{C}_c(r)$ and $\hat{C}_f(r)$ denote the ground-truth, coarse volume predicted, and fine volume predicted pixel colors for ray r, respectively. The ray r passing through a pixel is determined based on the pixel's position and the camera's location. Due to the differentiability of the rendering function, the only input required to optimize this scene representation is a set of images with known camera poses (Mildenhall et al., 2020).

Detailed overview of the MLP

A scene is represented by using a fully-connected (non-convolutional) deep network, whose input is a single continuous $5D$ coordinate and whose output is the volume density and view-dependent emitted radiance at that spatial location.

The MLP, with an architecture of 8 layers and 256 neurons per layer, first takes the $3D$ spatial position coordinate as input and processes it through eight layers, returning two values: σ and a 256-dimensional feature vector. The feature vector is then concatenated with the viewing direction parameters $d = (\phi, \theta)$ and passed through another MLP layer, which produces the viewing direction dependent color $c = (r, g, b)$. Figure 4 illustrates the architecture of this MLP, highlighting the flow of information through the layers (Mildenhall et al., 2020). The variable $\gamma(x)$ in the Figure refers to the positional encoding explained in section 2.2.1, which helps

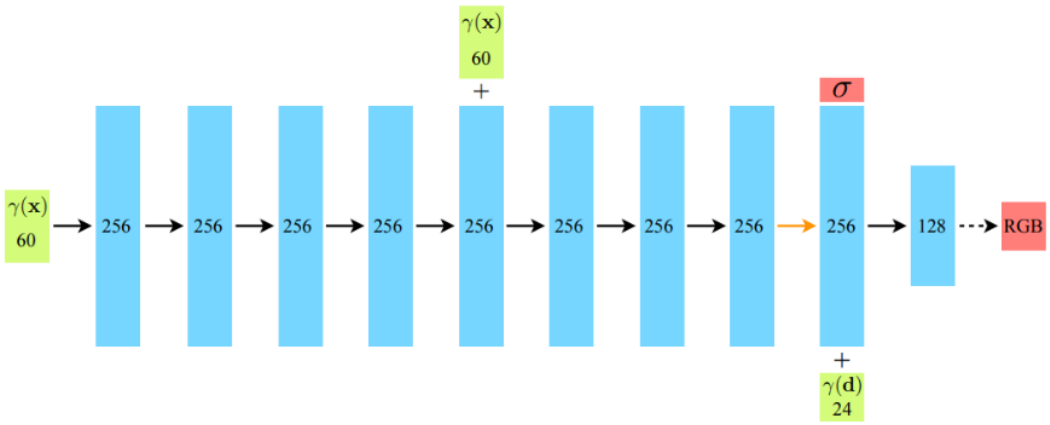


Figure 4.: The architecture of the MLP is depicted above. The green blocks represent the initial input vectors fed into the network, which undergo positional encoding denoted by $\gamma(x)$ as elaborated in section 2.2.1. The blue blocks illustrate the hidden layers and specify the number of neurons employed at each layer. Finally, the red blocks show the output vectors produced by the network. The MLP starts by taking the spatial location coordinate X as input. This input is processed through eight layers, resulting in the generation of σ and a 256-dimensional feature vector. The feature vector is then concatenated with the viewing direction parameters d, and this combined input is passed through an additional layer. The output of this layer represents the view-dependent color c (Mildenhall et al., 2020).

to process the input information effectively in the network. Once the neural network has optimized the weights to represent a scene, new views can be synthesized by rendering the neural radiance field from a particular viewpoint using classical rendering techniques.

Volume Rendering

With the output of the neural network, volume rendering is used to obtain the color $C(r)$ of any camera ray $r(t) = o + td$, with camera position o and viewing direction d using the classical rendering function represented as

$$C(r) = \int_{t_n}^{t_f} T(t) \cdot \sigma(r(t)) \cdot c(r(t), d) dt, \text{ where} \tag{2.2}$$
$$T(t) = \exp\left(-\int_{t_n}^{t} \sigma(r(s)), ds\right).$$

The rendering process, shown in the formula 2.2 involves integrating the emitted color $c(r(t), d)$, which depends on the $3D$ spatial location along the camera ray $r(t)$ and the viewing direction d, multiplied by the volume density $\sigma(r(t))$ of the scene at that location, over the range of distances along the ray from t_n to t_f.

This integral is attenuated by the factor $T(t)$, which represents the probability that the ray travels from t_n to t without hitting other particles, and accounts for the attenuation of light due to the density of the scene. The function $T(t)$ denotes the accumulated transmittance as the exponential of the integral of σ along the ray from t_n to t. Since σ is always non-negative, $T(t)$ can only decrease or stay the same as t increases along the ray. The final color $C(r)$ at a pixel $r(u, v)$ on the image plane is obtained by integrating the contributions of all points along the ray, each weighted by their emitted color, density, and attenuation factor, resulting in an accurate synthesis of novel views of complex scenes in NeRF. Rendering a view from the continuous neural radiance field requires estimating this integral $C(r)$ for a camera ray traced through each pixel of the desired virtual camera (Mildenhall et al., 2020).

The estimation of the continuous integral utilizes quadrature techniques. However, employing deterministic quadrature, typically used for rendering discrete voxel grids, would impose resolution constraints on the representation. This limitation would arise if the MLP were only evaluated at predetermined discrete locations. In contrast, the NeRF algorithm incorporates a stratified sampling technique. It involves dividing the interval $[t_n, t_f]$ into N evenly-spaced bins and drawing random samples uniformly from within each bin. This approach enables a more flexible and higher-resolution representation.

$$t_i \sim \mathsf{U}\left[t_n + \frac{i-1}{N}(t_f - t_n), t_n + \frac{i}{N}(t_f - t_n)\right] \tag{2.3}$$

Despite utilizing a discrete set of samples for integral estimation, stratified sampling facilitates the representation of a continuous scene. This is achieved by evaluating the MLP at continuous positions during the optimization process. The obtained samples are then used to estimate $C(r)$ using the quadrature rule outlined in Max's volume rendering review (Max, 1995).

$$\hat{C}(r) = \sum_{i=1}^{N} T_i \left(1 - \exp(-\sigma_i \delta_i)\right) c_i, \text{ where } T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right) \tag{2.4}$$

Here, $\delta_i = t_{i+1} - t_i$ represents the distance between adjacent samples. Calculating $\hat{C}(r)$ as a function of c_i and σ_i according to Equation (2.4) is trivially differentiable, which is beneficial for the optimization process. This calculation simplifies to traditional alpha compositing with alpha values $\alpha_i = 1 - \exp(-\sigma_i \delta_i)$ (Mildenhall et al., 2020; Tancik et al., 2020).

Positional Encoding

Allowing the F_θ network to operate directly on the $5D$ input coordinates often leads to poor rendering in terms of representing high-frequency variations in color and geometry, as noted in (Mildenhall et al., 2020). Input encodings have proven useful in recurrent architectures, particularly in attention components, where they help the neural network to identify the location it is currently processing (Müller et al., 2022; Vaswani et al., 2023). The NeRF algorithm encodes scalar positions as a multi-resolution sequence of $L \in \mathbb{N}$ sine and cosine functions as shown in the formula 2.5.

$$\gamma(x) = \Big(sin(2^0 x), sin(2^1 x), ..., sin(2^{L-1} x), cos(2^0 x), cos(2^1 x), .., cos(2^{L-1} x)\Big) \tag{2.5}$$

Here, L is a parameter representing the number of encoding functions used (Mildenhall et al., 2020; Müller et al., 2022). The NeRF algorithm applies this encoding to the $5D$ input coordinates independently. By mapping the input into a higher-dimensional space using high-frequency functions before passing it to the network, the algorithm achieves better fitting of data containing high-frequency variations (Mildenhall et al., 2020).

To conclude this section, Figure 5 visually represents the crucial aspects of view dependency and positional encoding within the context of NeRF.

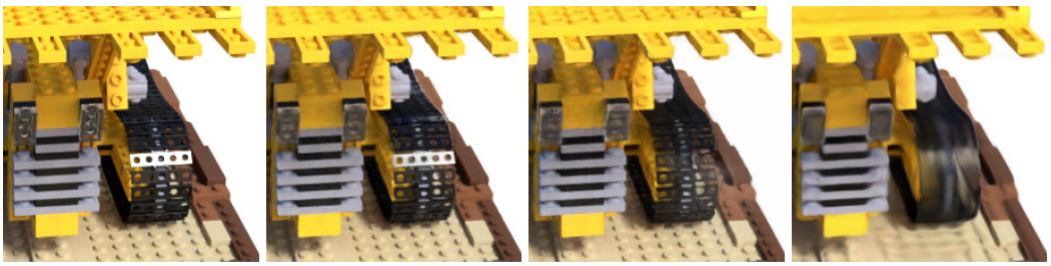


Figure 5.: This visualization demonstrates the advantages of incorporating view-dependent emitted radiance and utilizing high-frequency positional encoding in a full model. When view dependence is removed, the model fails to reproduce the specular reflection on the bulldozer tread. Likewise, the absence of positional encoding significantly diminishes the model's ability to represent intricate details of high-frequency geometry and texture, leading to an overly smoothed appearance (Mildenhall et al., 2020).

The first image (a) of Figure 5 shows the ground truth, providing a reference for the scene's actual appearance and spatial relationships. Image 5(b) demonstrates the output of the complete NeRF model, which incorporates both view dependency and positional encoding. This image showcases the model's remarkable ability to synthesize accurate views of the scene. By considering these components, NeRF successfully captures intricate details, resulting in visually appealing reconstructions.

Image 5(c) focuses on the impact of removing view dependency from the NeRF model. By neglecting variations in appearance caused by different viewpoints, the generated image may lack to reproduce the specular reflection. Lastly, Image 5(d) explores the implications of excluding positional encoding from the NeRF algorithm. Removing the positional encoding decreases the model's ability to represent high frequency geometry and texture, resulting in an oversmoothed appearance (Mildenhall et al., 2020).

2.2.2. Improvements of the NeRF algorithm

The NeRF approach has demonstrated its effectiveness in both synthetic data and real-world scenes. In synthetic settings, the consistency in viewing distance during training and testing makes it easier to achieve effective results. Moreover, NeRF outperforms other novel view synthesis algorithms such as Local Light Field Fusion (LLFF) (Mildenhall et al., 2019b) and Scene Representation Networks (SRN) (Sitzmann et al., 2020) when applied to real-world scenarios.

However, single-scene training alone requires a minimum of 12 hours, highlighting the computationally intensive nature of the process, which requires significant time and resources. In addition, the rendering process of a trained neural radiance field, which involves ray-marching through the scene, can be time consuming, posing a challenge for real-time or interactive rendering. Additionally, while NeRF excels in many areas, it still faces difficulties in accurately capturing the diversity and complexity inherent in real-world scenes, particularly concerning geometry, lighting, and unbounded conditions. To overcome these limitations, researchers have developed improvements to NeRF that aim to increase training efficiency, reduce rendering times, and enhance its ability to handle real-world scenes.

To provide a timeline of the improvements made to overcome these challenges, it's worth mentioning some key milestones. In 2020, NeRF was introduced, revolutionizing the field of novel view synthesis by using an MLP. Subsequently, the authors of NeRF released mip-NeRF in October 2021, which addresses the substantial limitations encountered when rendering images with varying resolutions or capturing scenes from different distances (Barron et al., 2021). NeRF also struggles to produce high-quality renderings for large unbounded scenes, where the camera can point in any direction and content can exist at any distance. In this context, conventional NeRF-like models often face challenges such as the generation of blurred or low-resolution renderings (resulting from the imbalance in detail and scale between near and far objects), slow training times, and potential artifacts arising from the inherent ambiguity of reconstructing a large scene from a limited set of images. To address these challenges, Barron et al. propose an extension of mip-NeRF that uses nonlinear scene parameterization, online distillation, and a novel distortion-based regularizer in June 2022. This model is called mip-NeRF 360 because it targets scenes in which the

camera rotates 360 degrees around a point (Barron et al., 2022).

One notable drawback of NeRF lies in its performance challenges. Training the model on a single high-end GPU demands a substantial time frame of one to two days, making it a significant investment in terms of time. Furthermore, the process of generating a single image using the trained model through inference also consumes several minutes. Addressing this concern, Müller et al. proposed a promising innovation named instant-ngp in July 2022 (Müller et al., 2022). This advancement introduced a grid-based technique that not only accelerated training but also expedited the rendering process. With the implementation of this algorithm, the convergence of the NeRF model can now be observed within 20 seconds. The instant-ngp algorithm is used in this work to generate neural radiance fields and is therefore explained in more detail in section 2.2.3.

Further advancements were made in 2023 with the unveiling of Zip-NeRF, effectively addressing challenges related to aliasing, scaling, and training time, resulting in significant overall performance improvements. The authors show how ideas from rendering and signal processing can be used to construct a technique that combines mip-NeRF 360 and grid-based models such as instant-ngp to yield error rates that are 8%-77% lower than either prior technique, and that trains 24x faster than mip-NeRF360. The latest release on Siggraph in August 2023 came with MeRF and BakedSDF, which promise further developments to push the idea of NeRF forward. For readers who wish to delve deeper, the appendix A includes detailed explanations of the algorithms Mip-NeRF, Mip-NeRF360, Zip-NeRF, MeRF and BakedSDF.

2.2.3. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding

An important element of the NeRF approach is, as described in section 2.2.1, the inclusion of positional encoding. While this technique is powerful for representing geometry, it introduces complexity into training and affects performance on Graphics Processing Unit (GPU)s where control flow and pointer chasing are costly (Müller et al., 2022). These challenges are addressed by the introduction of a new encoding method, proposed by Müller et al. in a paper titled "Instant Neural Graphics Primitives". The authors introduce a flexible approach for input encoding, enabling the utilization of a smaller network without compromising quality. This results in a reduction of floating-point operations and memory access, achieved by integrating a compact neural network with a multi-resolution hash table. The architecture benefits from the multi-resolution structure, allowing the network to effectively manage hash collisions and facilitating easy parallelization on modern GPUs. To maximize this parallelism, the entire system is implemented using fully-fused Compute Unified Device Architecture (CUDA) kernels, designed to minimize computational waste and memory bandwidth. The outcome is a remarkable speedup by several orders of magnitude,

enabling rapid training of high-quality neural graphics components within seconds. An example of the rapid training dynamics is provided in Figure 6, where we showcase the training progression that converges within 20 seconds. The scene was captured from a 180-degree perspective, utilizing 145 frames for training, and trained using a Laptop GPU A3000.

(a) Initial state at the beginning of training. (b) State after 20 seconds of training.

Figure 6.: Training of a neural radiance field using instant-ngp. The object was captured from a 180-degree perspective, utilizing 145 images for training. Impressively swift convergence is observed in the training process, yielding a usable scene within 20 seconds on a Laptop A3000 GPU. This visualization underscores the remarkable speed of the training procedure.

Additionally, rendering tasks are expedited, taking mere tens of milliseconds even at a resolution of 1920×1080. This new encoding approach not only demonstrates efficiency improvements for the NeRF algorithm but also for other constructs known as "neural graphics primitives". These are mathematical functions that utilize neural networks to parameterize appearance. For instance, the Signed Distance Function (SDF) is used to represent shapes in $3D$ space by measuring the shortest distance to its surface, while Gigapixel Image Approximation aims to provide high-resolution image reconstructions using neural network models, which learn the $2D$ to RGB mapping of image coordinates to colors (Müller et al., 2022).

The Multiresolution Grid Encoding

Given a fully connected neural network $m(y;\Phi)$, the focus lies in developing an encoding strategy for its inputs, denoted as $y = \gamma(x;\theta)$. This encoding scheme is designed to enhance the accuracy of approximations and expedite training across a diverse array of applications, all while ensuring that any resultant performance gains are achieved without introducing significant overhead. Within this neural network architecture, not only are the weight parameters Φ trainable, but also the encoding parameters θ. These parameters are organized into L levels, with each level accommodating up to T feature vectors, each with a dimensionality of F. To illustrate, Figure 7 outlines the sequential stages encompassed within the multiresolution hash encoding process.

Each level, illustrated as red and blue in this Figure, operates independently, functioning as a repository for feature vectors positioned at the vertices of a grid. The resolution of this grid is meticulously selected to align with a geometric progression that bridges the most coarsest and finest resolutions $[N_{\text{min}}, N_{\text{max}}]$. The resolution of each grid increases by a fixed factor relative to the previous level, although it's important to note that the growth factor is not limited to two. In actuality, the authors deduce the scaling based on the selection of three distinct hyperparameters:

$$
\begin{aligned}
L &:= \text{Number of Levels} \\
N_{\text{min}} &= \text{Coarsest Resolution} \\
N_{\text{max}} &= \text{Finest Resolution}
\end{aligned}
$$

and the grow factor computed with

$$
b := \exp\left(\frac{\ln N_{\text{max}} - \ln N_{\text{min}}}{L-1}\right).
$$

Therefore, the resolution of grid level l is $N_l := \lfloor N_{\text{min}} \cdot b^l \rfloor$, where $\lfloor \cdot \rfloor$ denotes the floor function that rounds down to the next lowest integer.

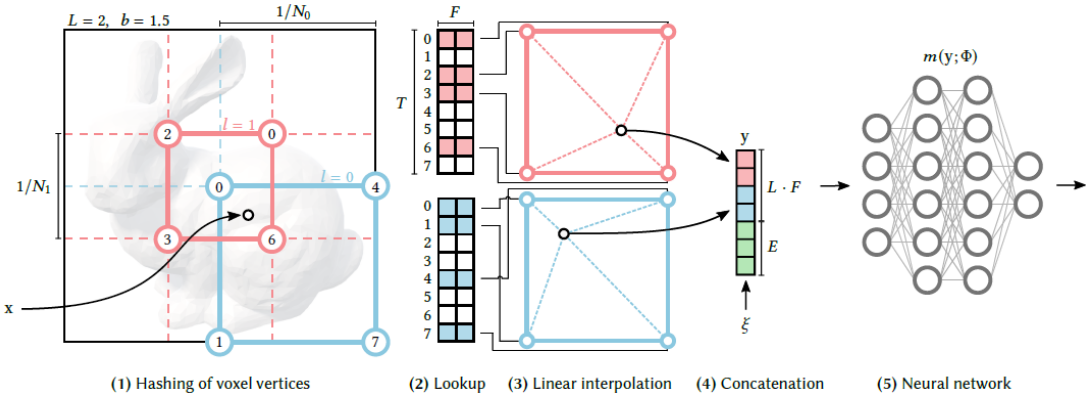


Figure 7.: Illustration of the multiresolution hash encoding in $2D$. (1) for a given input coordinate x, Müller et al. find the surrounding voxels at L resolution levels and assign indices to their corners by hashing their integer coordinates. (2) for all resulting corner indices, they look up the corresponding F-dimensional feature vectors from the hash tables θ_l and (3) linearly interpolate them according to the relative position of x within the respective l-th voxel. (4) the result of each level will be concatenated, as well as auxiliary inputs $\xi \in \mathbb{R}^E$, producing the encoded MLP input $y \in \mathbb{R}^{LF+E}$, which (5) is evaluated last. To train the encoding, loss gradients are backpropagated through the MLP (5), the concatenation (4), the linear interpolation (3), and then accumulated in the looked-up (2) feature vectors (Müller et al., 2022).

Following the establishment of the grid hierarchy, the next step involves defining the input transformation. For a given input coordinate x, the surrounding voxels are identified at L resolution levels. For each level l, the initial procedure entails

the scaling of x by N_l and rounding to the nearest integer, thereby identifying the corresponding cell that envelops x. The indices to the corners of these voxels are then assigned using a hashing technique based on their integer coordinates. This process is illustrated in (1) in Figure 7 (Müller et al., 2022; Slater, 2022).

Next, for each resulting corner index, the corresponding F-dimensional feature vectors are retrieved from the hash tables θ_l depicted in (2) in Figure 7. Each corner coordinate is hashed with the employed function $h(x) = \bigoplus_{i=1}^{d} x_i \cdot \pi_i \bmod T$, where $\oplus$ denotes bit-wise exclusive-or and π_i are large prime numbers. To enhance cache coherence, the authors set $\pi_1 = 1$, along with $\pi_2 = 2654435761$ and $\pi_3 = 805459861$. To map the hash to a slot in each level's hash table, we simply modulo by T. The hash is then used as an index into a hash table. Each grid level has a corresponding hash table described by two more hyperparameters:

$$T := \text{Hash Table Slot}$$
$$F := \text{Features per Slot}$$

Each slot contains F learnable parameters. Throughout the training process, gradients are backpropagated through the entire network, extending even to the hash table entries. This dynamic optimization refines the entries to establish an effective input encoding strategy. The authors do not explicitly handle hash function collisions through traditional methods such as probing, bucketing, or chaining. Instead, they rely on the neural network to learn how to disambiguate hash collisions, thereby avoiding control flow divergence, simplifying implementation complexity, and improving performance. However, this adaptability presents a challenge when scaling down the encoding to very small hash table sizes—eventually, a large number of grid points are assigned to each slot (Müller et al., 2022; Slater, 2022).

The feature vectors F_i are then linearly interpolated based on the relative position of x within the respective l-th voxel to establish a resultant value at $\mathbf{x}$ itself (3). The authors note that interpolating the discrete values is necessary for optimization with gradient descent: it makes the encoding function continuous.

At this point, we have mapped x to L distinct vectors, each with a length of $\mathbf{F}$ - one vector for every grid level l. The results from each level are concatenated, along with auxiliary inputs (such as the encoded view direction and spatial location in neural radiance fields) $\xi \in \mathbb{R}^E$ (4) to produce $y \in \mathbb{R}^{\mathbb{L}\mathbb{F}+\mathbb{E}}$, which is the encoded input $\gamma(x;\theta)$ to the MLP $m(y;\Phi)$.

The encoded result y is then fed into a fully connected basic neural network with ReLU activations, similar to a multilayer perceptron. Remarkably, this network can be compact; for example, the authors use two hidden layers, each with a dimension of 64. During training, loss gradients are backpropagated through the MLP, the

concatenation, and the linear interpolation. These gradients are then accumulated in the corresponding looked-up feature vectors, facilitating the training of the encoding process (Müller et al., 2022; Slater, 2022).

2.2.4. Importance of the NeRF algorithm

In recent times, NeRF models have attracted significant attention within the Computer Vision community. Numerous papers and preprints have surfaced on code aggregation platforms, and a substantial portion of these works have found their way into Computer Vision conferences. The original NeRF paper by Mildenhall et al. received more than 3000 citations and a growing interest with no signs of slowing down (*Neural Radiance Fields Mildenhall et. al.* n.d.). These research innovations have driven interests in a wide variety of disciplines in both academia and industry.

Roboticists have ventured into leveraging NeRFs for a spectrum of applications including manipulation, motion planning, simulation, and mapping. Moreover, NeRF methods have extended their influence to tomography applications, as well as the task of perceiving individuals within videos. The potential of NeRFs has reverberated across domains such as visual effects and gaming studios, wherein the technology is being explored for production and the creation of digital assets. Even news outlets have embraced NeRFs to craft narratives using novel formats. The horizon of potential applications stretches extensively, and the surge of interest has even given rise to startups dedicated to the deployment of this technology (Tancik et al., 2023).

We would like to point out some extension ideas that originated in the original NeRF algorithm, which have emerged, each addressing specific challenges and scenarios in Computer Vision. Some extensions focus on training NeRF models without known camera parameters (Wang et al., 2021; Meng et al., 2021; Yen-Chen et al., 2021; Heo et al., 2023), while others grapple with the challenges of managing unstructured sets of real-world photos (Martin-Brualla et al., 2021; Chen et al., 2022b; Jun-Seong et al., 2022). Further developments seek to tackle dynamic scenes characterized by intricate, non-static geometries (Pumarola et al., 2020; Tretschk et al., 2021; Weng et al., 2022; Li et al., 2023c). The realm of generative image synthesis has also benefited from NeRF-inspired extensions (Schwarz et al., 2020; Yu et al., 2020; Mildenhall et al., 2022). Lastly, there are extensions specifically tailored for novel view synthesis with LiDAR sensors (Tao et al., 2023; Zhang et al., 2023; Huang et al., 2023).

In conclusion, the NeRF algorithm has emerged as a cornerstone in the landscape of Computer Vision and beyond. Its ability to synthesize high-quality $3D$ reconstructions from $2D$ images has not only revolutionized our perception of scene representation but has also extended its influence across a multitude of disciplines. The algorithm's capacity to fuse data from various sources, including unconstrained photo collections and dynamic scenes with complex geometries, underscores its adaptability

and potential. The importance of NeRF resonates deeply in academia, where it has inspired new paths of research and exploration. Its integration into Computer Vision conferences and sustained citation growth are testament to its impact and continued relevance. The industrial and practical applications of NeRF are equally remarkable. The technology's adoption in industries such as robotics, entertainment, simulation, and generative image synthesis speaks to its potential. As NeRF continues to inspire extensions and adaptations that address an ever-widening spectrum of challenges, its importance becomes even more apparent. This algorithm has opened up novel possibilities for how we interact with, interpret, and create visual content.

2.3. The Industrial Metaverse

As mentioned in the introduction 1, NeRF has a wide range of potential applications, especially for the industry.NeRF can be used to generate high-fidelity virtual representations of physical objects. This can be used for virtual prototyping and testing, as well as for training and simulation.

In addition, the NeRF method can be used for remote collaboration and communication within the Industrial Metaverse. By leveraging NeRF's ability to generate high-quality $3D$ representations, remote teams can effectively visualize and discuss complex industrial assets, designs, or processes. This fosters better collaboration, efficient decision-making, and reduces the need for physical presence. Before we can discuss the idea of NeRFs in the Industrial Metaverse, it is important to look at the concept of the Metaverse.

The term 'Metaverse' traces its origins to the realms of science fiction, particularly a book by Stephenson. He didn't offer a rigid definition of the Metaverse, but he did describe a persistent virtual world that reached into, interacted with and influenced almost every aspect of human existence. It was a place of work and leisure, self-realization and physical exhaustion, art and commerce. At any one time, there were some 15 million human-controlled avatars on the Street, which Stephenson described as "the Broadway, the Champs Elysees of the Metaverse, but covering the entire surface of a virtual planet more than two and a half times the size of the Earth" (Ball, 2022).

The phenomenon of the Metaverse, given its recent emergence, faces a significant challenge in the absence of a universally accepted definition. This lack of consensus provides an opportunity to explore various attempts to define the Metaverse, highlighting its multifaceted nature and its potential for innovation. While different definitions continue to be proposed, there is a consensus in all ongoing debates that the Metaverse will be experienced in the form of a virtual world (Bitkom, 2022).

2.3.1. Defintions of the Metaverse

From NVIDIA's perspective, the Metaverse is seen as an extension of the internet's evolution. The internet started with a web interface that primarily relied on text-based interactions. Over time, it progressed to incorporate $2D$ images, multimedia content, and became increasingly accessible to a wider audience. The Metaverse is the internet in $3D$, a network of connected, persistent, virtual worlds. It will extend $2D$ web pages into $3D$ spaces and worlds (Caulfield, 2022). This transition allows for a more immersive and interactive online experience. In this context, the Metaverse's potential expands beyond gaming and entertainment, encompassing the construction of factories, development of cities, the building of bridges, and other challenging projects. NVIDIA's vision of the Metaverse was put together into a platform known as "Omniverse", a platform for connecting $3D$ worlds into a shared virtual universe, or Metaverse. Omniverse was introduced at the GTC conference, a global Artificial Intelligence (AI) conference for developer and IT experts, in 2019 and was later released as an open beta version in 2020. It is based on a scene description format called Universal Scene Description (USD), originally developed by Pixar. USD is an open source API and file framework for complex scene graphs is not only easily extensible but also simplifies the exchange of assets between industry software. This innovative foundation enables the construction and seamless integration of various elements within the Metaverse in real time. Leveraging the capabilities of USD enables unprecedented real-time collaboration, providing users with instant updates and notifications when their scene needs to be synchronized with changes made by other team members. This streamlined workflow facilitates efficient teamwork and ensures that everyone remains in sync within the dynamic Omniverse environment (NVIDIA, 2021; NVIDIA, 2023).

Another definition that will be examined in more detail below is that of Bitkom, Germany's digital industry association. In this definition, the Metaverse should not be understood as a virtual parallel world with no connection to the real world, but rather as a multiplicity of connections in both directions. On the one hand, real world objects will find their virtual counterparts, effectively mirroring their existence. On the other hand, Extended Reality can be used to seamlessly integrate virtual objects into the tangible environment, enabling interactive experiences with these virtual entities (Bitkom, 2022).

> With today's knowledge, the Metaverse can be seen as the next logical stage of the Internet, a three-dimensional Internet that is permanent and happens in real time (Bitkom, 2022).

The Metaverse demonstrates an increasing integration of virtual and real worlds. Extended Reality, which includes Augmented Reality, Mixed Reality and Virtual Reality

technologies, serves as a gateway to the virtual world and plays a crucial role in the experience of the Metaverse. The immersive nature of Extended Reality (XR) technologies allows users to fully engage with the Metaverse. VR in particular offers total immersion, allowing users to experience the Metaverse in a fully virtual $3D$ environment. It provides a full and immersive experience by transporting users into a digital realm. On the other hand, Augmented Reality (AR) and Mixed Reality (MR) broadcast virtual $3D$ objects into the real world, blurring the boundaries between the virtual and physical worlds. These technologies allow virtual objects to be integrated into our real world environment, serving as extensions or content within the Metaverse. An additional aspect of the Metaverse is the emergence of multidimensional user-generated content, particularly in form of $3D$ objects. These objects can be transferred, ported and traded within the context of the Metaverse, enabling dynamic interactions and experiences. Bitkom, Germany's digital industry association, concludes that the Metaverse combines elements of AR, MR and VR with the traditional Internet to present content to users according to their needs (Bitkom, 2022).

Matthew Ball's definition, an author known for his insightful analysis and writings on the Metaverse, expands on the concept by providing a comprehensive framework that captures its key attributes. According to Ball, the Metaverse is a "massively scaled and interoperable network of real-time rendered $3D$ virtual worlds that can be experienced synchronously and persistently by a virtually unlimited number of users with an individual sense of presence and with continuity of data such as identity, history, permissions, objects, communication and payments" (Ball, 2022). This description emphasizes the interconnected nature of the Metaverse, highlighting that it is not a solitary virtual environment but a network of multiple virtual worlds that coexist and interact with each other. In addition, Ball emphasizes that the Metaverse enables synchronous and persistent experiences for users. Synchronous refers to the real-time nature of interactions within the Metaverse, allowing users to engage with each other and the virtual environment simultaneously. Persistence implies that the Metaverse is not a temporary or fleeting experience but rather a continuous and ongoing digital realm that retains data and progress over time. The Metaverse aims to create a sense of embodiment and agency, allowing users to navigate and interact with the virtual environment as if they were physically present from a first person point of view. Continuity of data is another significant aspect of the Metaverse. This encompasses various elements such as identity, history, permissions, objects, communication, and payments. In the Metaverse, users should be able to carry their digital identity, personal history, and virtual possessions across different virtual worlds seamlessly. Moreover, they should be able to communicate and transact within the Metaverse, enabling social interactions, commerce, and other activities. By encompassing these elements, Matthew Ball's definition provides a holistic understanding

of the Metaverse. It emphasizes the scale, interconnectivity, real-time nature, persistence, presence, and continuity of data that make up this evolving concept. This comprehensive framework helps to shape the discourse around the Metaverse, fueling further exploration and innovation in this exciting digital frontier (Ball, 2022).

Given the different perspectives and definitions of the Metaverse, it is clear that although the Metaverse is a multifaceted concept, there are certain key aspects that are common to its various interpretations. For the purposes of this thesis, the Metaverse is defined as follows:

> The Metaverse is the next level of digitalization that connects (digital) ecosystems and extends the existing (mostly $2D$) internet with realistic $3D$ representations. It represents a persistent, shared, $3D$ virtual space that integrates real and virtual worlds, providing an interactive platform for a virtually unlimited number of users. It operates in real time, thereby enabling synchronous collaboration and communication. Drawing from Ball's definition, this virtual environment encompasses the continuity of data, such as individual identity, history, permissions, and objects, and supports various transactions including communication and payments.

This definition reflects the Metaverse's function as the next evolution of the Internet, a perspective acknowledged by Bitkom. Similar to NVIDIA's vision, it acknowledges the seamless integration of various elements of the real and virtual worlds and the creation of an immersive user experience. The Metaverse in this context enables high-fidelity representations of physical objects and processes, fostering enhanced collaboration and decision-making, particularly in industrial settings. Therefore, the Metaverse within the context of this thesis is a technologically advanced, interactive platform that is persistently evolving and blurring the boundaries between physical and virtual realities. This platform allows for both synchronous collaboration and individual exploration, with the potential to significantly alter the ways we interact, work, and innovate within the virtual industrial world.

Upon establishing a comprehensive understanding of the Metaverse, we can now delve into the concept of the Industrial Metaverse. Engaging the Chatbot "GPT-3" to define the Industrial Metaverse yields an insightful insight:

> The Industrial Metaverse is a digital simulation of the physical world, where physical and digital systems interact with each other. It utilizes digital twin technology to replicate physical systems, allowing for simulations and autonomous control of processes. The Industrial Metaverse allows businesses to create digital assets that generate expected income and value by connecting these isolated systems. It provides opportunities

> to unlock the potential of digital technologies and benefits all participants in the transformation (GPT-3, privat conversation, 08-30-2023).

According to the definition provided by GPT-3, the Industrial Metaverse is characterized as a digital emulation of the physical world, where the interplay between physical and digital systems takes center stage. Facilitated by digital twin technology, this phenomenon replicates tangible systems digitally, resulting in the capacity for simulations and the autonomous control of processes. The Industrial Metaverse serves as a platform for businesses to craft digital assets capable of generating anticipated value and income, achieved through the interconnection of previously disparate systems.

It's important to note, however, that the Industrial Metaverse is focused on specific industries, particularly the industrial sector. Siemens' delineation of the Industrial Metaverse underscores its purpose-driven nature, specifically designed to address real-world challenges and business needs. This specialized iteration recognizes the potential of the Metaverse to improve operational efficiency, optimize resource management and promote sustainable practices within the industrial landscape. Unlike the more general Metaverse, which spans a wide range of applications and domains, the Industrial Metaverse takes on a tailored role, addressing industry-specific challenges, streamlining industrial processes and facilitating data-driven decision making.Siemens further narrows down the concept, emphasizing its industry-specific applications (SIEMENS, 2023):

> While consumer metaverse applications are still evolving, the Industrial Metaverse focuses on purpose-driven use cases that address real-world challenges and business needs. The resource efficiencies enabled by Industrial Metaverse solutions can enhance business competitiveness and drive progress towards sustainability and resilience goals (SIEMENS, 2023).

By adopting Siemens' perspective, we highlight the Industrial Metaverse's specific role in utilizing digital technology to transform industries, emphasizing its focus on efficiency and sustainability.

2.3.2. The Current State of the Industrial Metaverse

The advent of Industry 4.0 is revolutionizing the way companies manufacture, improve and distribute their products. Manufacturers are integrating new technologies such as the Internet of Things, cloud computing and analytics, and AI and machine learning into their production facilities and throughout their operations. These smart factories are equipped with advanced sensors, embedded software and robotics that collect and analyze data and allow for better decision making. This digital technologies lead to increased automation, predictive maintenance, self-optimization of processes improvements and, above all, a new level of efficiency and responsiveness to customers

not previously possible. Developing smart factories provides an incredible opportunity for the manufacturing industry to enter the fourth industrial revolution. Analyzing the large amounts of big data collected from sensors on the factory floor ensures real-time visibility of manufacturing assets and can provide tools for performing predictive maintenance in order to minimize equipment downtime. Using high-tech **IoT!** (**IoT!**) devices in smart factories leads to higher productivity and improved quality. Replacing manual inspection business models with AI-powered visual insights reduces manufacturing errors and saves money and time. With minimal investment, quality control personnel can set up a smartphone connected to the cloud to monitor manufacturing processes from virtually anywhere. By applying machine learning algorithms, manufacturers can detect errors immediately, rather than at later stages when repair work is more expensive (IBM, 2023).

However, in the midst of this progress it's important to recognize that humans still play an important role. While the vision of Industry 4.0 emphasizes automated individualized mass production, its foundation and evolution are deeply rooted in human insight. The Metaverse shifts this focus towards user-centric experiences. To effectively navigate this transition, innovative solutions like $3D$ visualization and novel interfaces are needed to navigate this evolving landscape (Soerensen, 2023). As the industrial landscape continues to transform, the Metaverse stands out, highlighted by projects such as CUT (Connected Urban Twins) for cities like Hamburg and Leipzig (Reinecke et al., 2021). Its significance is further amplified with the introduction of a new award category named "Metaverse" in Time magazine's 'Best Innovation' award (Time, 2022).

A significant stride towards materializing the Industrial Metaverse has been unveiled through a partnership between Siemens and NVIDIA. By synergizing Siemens' prowess in industrial automation, software, and infrastructure with NVIDIA's AI and accelerated graphics expertise, this collaboration aims to construct an Industrial Metaverse. The partnership's initial phases envision an amalgamation of Siemens Xcelerator and NVIDIA Omniverse, creating a domain where physics-based digital models from Siemens merge with real-time AI capabilities from NVIDIA. This unified platform empowers companies to make faster, more confident decisions, fostering an era of unparalleled industrial innovation (NVIDIA, 2022). Exemplifying the tangible impact of this trajectory, even revered car manufacturer Mercedes-Benz has embraced the digital-first ethos, leveraging NVIDIA Omniverse to revolutionize production strategies, marking a decisive step towards manufacturing within the Metaverse (shapiro, 2023).

The notion of a 'Metaverse' has long been an aspiration of futurists, science fiction writers and tech enthusiasts. We may not yet have fully explored the depths of the Metaverse, but as Jensen Huang, CEO of NVIDIA, has observed:

> The metaverse will grow organically as the internet did — continuously and simultaneously across all industries, but exponentially, because of computing's compounding and network effects (Caulfield, 2022).

It's important to keep in mind that the concept of the Industrial Metaverse is still fluid and will change over the coming years.

2.3.3. The Metaverse and it's opportunities for the industry

The core technologies behind the Industrial Metaverse is the concept of digital twins. These are virtual representations of physical objects, person, processes, or even entire production lines. Digital twins serve multiple functionalities such as prototyping, testing, training and simulation, thereby enabling better control over manufacturing operation.

Prototyping Engineers can manipulate virtual models to try out different configurations, and refine the design before it reaches the physical manufacturing stage. This results in cost-saving and a more efficient design process.

Testing Virtual environments provided by the Industrial Metaverse can be leveraged for robust testing scenarios. Companies can use the digital twin technology to simulate real-world conditions under which the prototype will operate, enabling early identification and mitigation of potential issues.

Training The Industrial Metaverse serves as a platform for workforce training. Through the use of immersive simulations based on digital twins, employees can participate in realistic training exercises. This enhances skill development and readiness without the risks associated with real training environments.This methodology not only amplifies skill development and preparedness but also provides the benefit of repeated practice, which may be logistically challenging or costly in real-world settings.

KLM, a national airline in the netherlands, serves as a case in point, pioneering the use of VR in pilot training. As Sebastian Gerkens, Senior Instructor Embraer at *KLM*, articulates, "Virtual Reality makes training more accessible. It allows for on-demand, location-independent training sessions, freeing pilots from the constraints of traditional classrooms or simulators. It encourages exploration within a safe, virtual environment". This innovative approach to training yields financial efficiencies as well. By reducing reliance on external suppliers and offering more flexible scheduling options for pilots, *KLM* has identified a cost-effective strategy that could serve as a model for other industries (*KLM Cityhopper introduces Virtual Reality training for pilots* 2020).

Simulation Beyond prototyping and testing, digital twins can be used for complex simulations to model various scenarios, including supply chain events, workflow bottlenecks, or disaster response. These simulations contribute to more effective decision-making processes

Collaboration Another transformative aspect of the Industrial Metaverse is its capacity for fostering collaboration. Teams located in disparate geographical locations can engage in real-time cooperation through a shared virtual environment. This breaks down barriers and expedites project timelines.

Sales How does the Industrial Metaverse affect sales in a hardware-centric industry? The answer lies in a strategic shift to an application-driven perspective. While many hardware companies have deep product knowledge, there is significant potential in understanding the practical applications for which these products are designed. Typically, this application knowledge is held by customers and sales teams who understand the exact roles and functions these products perform. The true value of a product comes when it is directly aligned with a customer's specific requirements. The challenge, however, is that while customers are adept at outlining their application needs, they do not always identify the exact products that meet those needs (Soerensen, 2023).

The Industrial Metaverse provides an elegant solution to this gap. It curates a user-centric ecosystem that is conducive to both sales conversations and customer consultations. Within this space, users can not only articulate their challenges, but also have the unique advantage of visualizing their production facilities. This virtual replication helps to identify and optimize solutions as they can be tested and refined within the metaverse prior to real-world implementation. Rather than navigating the traditional product-first maze, the Metaverse introduces a dynamic platform where users can collaboratively design unique solutions. This paradigm allows for instant visual adaptations; for example, adjusting the dimensions of a conveyor belt no longer depends on static diagrams. The inherent immersion of the metaverse enhances solution demonstrations. Using augmented reality (AR), the fit of a proposed solution within a customer's workspace can be instantly visualized, facilitating quick compatibility checks and corrections for unforeseen spatial challenges (Soerensen, 2023).

In this metaverse-driven shift, hardware-based companies are on the threshold of redefining their approach to the industry. By adopting an user-centric strategy and leveraging immersive technologies, they can increase customer engagement, streamline project workflows, and provide instant insight into the effectiveness of their solutions, ushering in a new era of innovation in the industrial landscape (Soerensen, 2023).

3. Related Work

In the context of NeRF research, a variety of frameworks have emerged, each with the aim of facilitating the development and the appicability of NeRF methods in different domains. In this chapter we delve into a selection of notable frameworks that contribute to the ongoing development and enhancement of NeRF. Our primary objective is to gain a comprehensive understanding of the functionalities and offerings provided by these frameworks. By thoroughly testing and interacting, we uncover both the progress and challenges within the domain of generating neural radiance fields, particularly when dealing with user-generated data.

Unlike frameworks that re-implemented original NeRF methods, such as the NeRF-pytorch framework (Yen-Chen, 2020), transitioning the original NeRF algorithm from the Tensorflow codebase to PyTorch, or HashNeRF-Pytorch (Bhalgat, 2022), a dedicated PyTorch-based implementation of the instant-ngp algorithm, we shift our focus to frameworks that aim to achive different goals.

We are exploring frameworks that seek to integrate different NeRF methods into a modular framework to make NeRF methods accessible to a wider audience, including scientists, artists, photographers, hobbyists and journalists. We will also look at a framework designed specifically for researchers to make it easy for them to develop their own NeRF methods. Additionally, our research includes frameworks that have successfully re-implemented the instant-ngp algorithm, aiming for a user-friendly design that not only removes the dependence on a graphics card, but also does not require deep technical knowledge. Lastly, we discuss LumaAI, a start-up app that offers the unique capability to generate neural radiance fields without any coding skills, using just an iPhone equipped with a LiDAR scanner. We emphasize that, to date, no frameworks are explicitly designed for industrial applications, leading us to our own implementation. We further underscore this aspect in the final discussion.

3.1. Overview over Frameworks offering several NeRF methods

NeRF is a rapidly growing field of research with diverse applications in computer vision, graphics, robotics and many other areas. Recognizing its importance, developers are building frameworks for NeRF methods to make them easier to use. These frameworks eliminate the need to set up each NeRF algorithm separately, making research more efficient and user-friendly. An outstanding feature of these frameworks is their modular

structure, which allows researchers and developers to easily implement different NeRF methods and adapt them to their specific needs. The modular design means that new methods can be easily integrated into existing frameworks without having to rewrite encodings, network architectures or ray marching functions. This is particularly useful as the field of NeRF is constantly evolving, with new methods and approaches emerging regularly.

Of particular interest for deeper exploration is *Nerfstudio*, which currently holds the distinction of being the largest and most well-known framework in this realm. Before taking a closer look at *Nerfstudio*, we would like to briefly discuss other frameworks that have dealt with the unification of different NeRF methods.

ArcNeRF is a product of Tencent's Applied Research Center Lab and a customizable framework that brings together different NeRF-based techniques for novel view rendering and object extraction. The goal is to encapsulate NeRF methods in a modular system that promotes seamless component changes and experimental execution. The distinctiveness of this framework is rooted in its pronounced modular design, which provides a clear distinction between the different domains. Moreover, this design facilitates the integration of any newly crafted module without affecting the framework's overall integrity. The flexibility extends to the configurational level, enabling easy adjustments via configuration files. This approach assures that modifications in one domain don't inadvertently impact others. Another interesting approach is the subdivision of the model into a foreground and background model. Each of these is confined to modeling in specific areas defined by parameters like sphere volume and other geometrical constructs. This design is optimal for everyday captured videos, simultaneously delivering high-quality object meshes and background renderings (Yue Luo, 2022). However, the installation process is notably intricate, often giving rise to errors that pose challenges in terms of comprehension. Moreover, gaining proficiency in training neural radiance fields with user-generated data demands extensive research. Once the training process is successfully initiated, the progress can be observed in real-time through the implementation of the *Nerfstudio* web viewer. It's important to note that while the viewer displays the progress, certain interactive elements such as buttons remain unimplemented, as they are transmitted from *Nerfstudio*.

JNeRF presents a benchmarking framework based on Jittor, a high-performance deep learning framework with Python as frontend and CUDA/C++ as backend. The scope of *JNeRF* includes the original NeRF and the instant-ngp algorithm; it is worth noting that more implementations of NeRF methods are planned. The operational aspects of *JNeRF* include training from scratch in a defined scene, such as the Lego

scenario, testing a pre-trained model stored as a `pkl` file, rendering a demonstration video by specifying a particular camera path, or extracting a mesh with color information. When training *JNeRF* with custom datasets, one must first create the dataset format, which involves splitting the datasets into different training, validation and testing subsets, each carefully paired with a corresponding JSON file. These JSON files needs to contain important camera parameters for each image, facilitating a structured and systematic approach to training. Nonetheless, the installation process entails numerous requirements, and it is noteworthy that the framework lacks explicit instructions for data pre-processing or a graphical user interface (GUI). This makes it a less interesting framework for our purposes (Hu et al., 2020).

XRNeRF constitutes an open-source PyTorch-based codebase for NeRF, serving as an integral component within the OpenXRLab project. The framework is designed to facilitate various aspects of NeRF research and experimentation. *XRNeRF* is oriented towards providing a comprehensive toolbox for NeRF exploration and implementation. The framework encompasses a range of functionalities, including benchmarking, dataset handling, model building, training, testing, and iterative control mechanisms. A noteworthy feature is the support for a variety of scene-NeRF methods, such as NeRF (Mildenhall et al., 2020), Mip-NeRF (Barron et al., 2021), KiloNeRF (Reiser et al., 2021), and instant-ngp (Müller et al., 2022), among others. Additionally, *XRNeRF* caters to human-NeRF methods like NeuralBody (Peng et al., 2021b) and AniNeRF (Peng et al., 2021a), which underscores its versatility and applicability. *XRNeRF* requires compliance with certain requirements, but users also have the option of using a Docker container to facilitate the installation. The framework underscores its modular approach by systematically categorizing model components into distinct types. These include the 'network,' which consists of an embedder, MLP, and renderer that form the complete NeRF pipeline. The 'embedder' is responsible for converting point-position and view-direction data into embedded formats. The MLP utilizes the output from the embedder to generate raw data, such as RGB and density values at sampled positions, usually through fully connected layers. Finally, the 'renderer' processes this raw data to output RGB values at individual pixels. *XRNeRF* allows for the creation of new networks, enabling researchers to build and customize models efficiently while maintaining compatibility with the overall framework structure. Notably, the framework distinguishes between training and test modes, adjusting the maximum iterations accordingly. The versatility of *XRNeRF* is evident in its ability to train models from scratch on specific scenes, such as the Lego scenario, or test pre-trained models, all while accommodating data selection through command-line arguments. While the framework provides ample functionalities, it does not offer explicit guidance on data pre-processing or incorporate a graphical user interface. In summary,

XRNeRF is a versatile tool that provides researchers with a comprehensive toolkit for NeRF research and includes a wide range of functionalities, model components and scene implementations (Contributors, 2022).

NeRF-Factory is proving to be a robust framework in the field of NeRF, providing PyTorch re-implementations of several established variants, including NeRF (Mildenhall et al., 2020), NeRF++ (Zhang et al., 2020), DVGO (Sun et al., 2022), Plenoxels (Fridovich-Keil and Yu et al., 2022), Mip-NeRF 360 (Barron et al., 2022), and Ref-NeRF (Verbin et al., 2022). It also encapsulates seven of the popular NeRF datasets. While its portfolio is extensive, it is pertinent to highlight the absence of the specific instant-ngp implementation that is central to this thesis. The design ethos behind NeRF-Factory emphasises modularity, ensuring a seamless integration experience for its users. At its core, *NeRF-Factory* is optimized to facilitate training of neural radiance fields on pre-established datasets. This positions it as an invaluable asset for researchers and practitioners keen on exploring NeRF advancements, enabling a comparative analysis of diverse algorithms (Yoonwoo et al., 2022).

NerfAcc emerges as a PyTorch toolbox tailored for accelerating both training and inference processes within the context of NeRF. The toolbox predominantly centers on optimizing the sampling procedure within the volumetric rendering pipeline of radiance fields - a foundational aspect applicable and seamlessly integrable across a multitude of NeRF implementations. Notably, *NerfAcc* stands as a universal and plug-and-play solution compatible with most NeRF methodologies. This toolbox brings forth notable acceleration benefits with minimal requisite modifications to existing codebases, effectively expediting the training of diverse NeRF models presented in recent research papers. With a pure Python interface, *NerfAcc* further offers the advantage of flexible APIs, enhancing usability and adaptability (Li et al., 2023a).

Kaolin Wisp NVIDIA *Kaolin Wisp* was designed as a dynamic, research-focused library for neural fields to help researchers address the challenges of this discipline. The platform is built on the foundational capabilities of the Kaolin library, which includes a broader and more robust suite of components for $3D$ deep learning research. The library consists of modular building blocks that facilitate the construction of complex neural fields, and an interactive application for training and visualizing these neural fields. This framework is easily extensible for research purposes and features a modular pipeline where each component can be easily swapped out to provide a plug-and-play configuration for standard training. Its array of features encompasses datasets, image input/output, mesh manipulation, and ray-related utilities. Notably, Wisp also offers fundamental components like differentiable renderers and data structures

(e.g., octrees, hash grids, triplanar features) that prove instrumental in constructing intricate neural fields. *Kaolin Wisp* is paired with an interactive renderer that supports flexible rendering of neural primitives pipelines like NeRF. Additionally, it incorporates debugging visualization tools, interactive rendering and training capabilities, logging functionalities, as well as trainer classes. Figure 8 provides an overview about the building blocks from Kaolin.

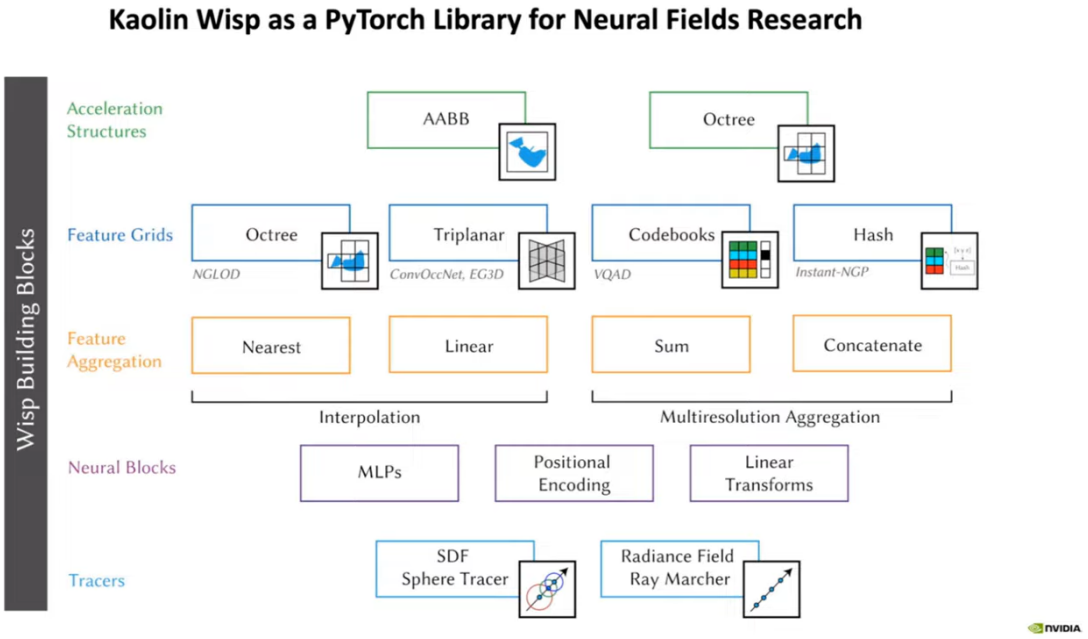


Figure 8.: NVIDIA Kaolin Wisp architecture and building blocks. Each pipeline component can be easily swapped out to provide a plug-and-play configuration for standard training (Takikawa et al., 2022)

The goal of *Kaolin Wisp* is to address limitations by offering modular building blocks that can be flexibly combined, akin to constructing with LEGO bricks. These Python-based modules enhance the accessibility of neural field research, allowing researchers to easily mix and match components according to their specific requirements (Takikawa et al., 2022).

NVIDIA *Kaolin Wisp* offers an extensive range of utility functions for neural field research. However, it may not align perfectly with the specific objectives of this thesis project. While *Kaolin Wisp* does provide valuable tools for researchers exploring various aspects of neural fields, it may not directly address the unique challenges and requirements highlighted in this thesis (Takikawa et al., 2022).

3.2. Nerfstudio - A Modular Framework for NeRF Development

Nerfstudio serves as a flexible and comprehensive framework for NeRF development. Its goal is to streamline various NeRF techniques into reusable, modular components,

providing real-time visualization of neural radiance field scenes. With a set of user-friendly controls, *Nerfstudio* facilitates an end-to-end workflow for generating neural radiance fields from popular NeRF datasets and user-generated data. The design of the library simplifies the implementation of NeRF by dividing each element into modular units, enhancing interpretability and user experience. We will not delve into detailed explanations of these components in this chapter; interested readers are referred to the original paper for a more thorough understanding (Tancik et al., 2023). Figure 9 provides an overview of the pipeline process and the diverse possibilities within the *Nerfstudio* framework.

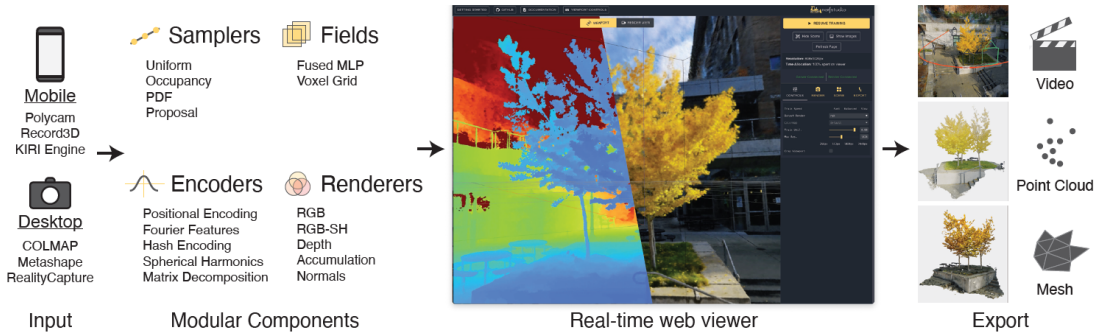


Figure 9.: *Nerfstudio* Framework Overview: *Nerfstudio* is a Python framework designed specifically for developing NeRF applications. It offers a range of features including support for various input data pipelines, a modular architecture based on core NeRF components, integration with a real-time web viewer, and versatile export capabilities. The primary objective of *Nerfstudio* is to streamline the development process of customized NeRF methods, facilitate real-world data processing, and enable seamless interaction with reconstructions (Tancik et al., 2023).

Nerfstudio is a highly interesting platform for the generation of neural radiance fields for researchers, developers, artists and more due to its four key components.

Firstly, *Nerfstudio's* user experience mirrors a plug-and-play model, allowing users to effortlessly incorporate NeRF into their projects. The development team is dedicated to providing extensive educational resources for all levels of NeRF expertise. Recognizing potential challenges for beginners and experienced users alike, *Nerfstudio* offers a wealth of tutorials, comprehensive documentation, and additional support materials.

Secondly, *Nerfstudio* supports images and videos from different camera types and mobile capture applications such as *Polycam*, *Record3D*, and *KIRI Engine*. It also accepts outputs from popular photogrammetry software like *RealityCapture* and *Metashape*. This integration eliminates the need for time-consuming structure-from-motion tools such as COLMAP.

Thirdly, *Nerfstudio's* real-time viewer, as described by the authors, has been inspired by the real-time rendering during training as presented in instant-ngp (Müller et al.,

2022). To address the difficulties associated with deploying local computing resources in remote settings, the authors developed a publicly accessible website that utilizes a ReactJS-based web viewer. The viewer is versatile, catering to users employing both local and remote GPUs. To simplify the use of remote compute, users only need to forward a port locally via SSH. Once training begins, the web interface provides real-time rendering of the neural radiance field, facilitating user interactions such as panning, zooming, and rotating around the scene during the optimization process and for the converged model. The viewer leverages ThreeJS for the implementation of camera controls and UI, allowing for the overlay of $3D$ assets like images, splines, and cropping boxes on the NeRF renderings. This enables users to intuitively compare the performance from different viewpoints. To maintain a steady frame rate and prevent lags in the user experience during fast camera movements, rendering resolution is adjusted, a strategy employed by Instant NGP (Müller et al., 2022). This approach also optimizes resource allocation, diverting more towards rendering in the viewer and reducing training time. The viewer is endowed with several features, including switching between different model outputs (such as RGB, depth, normals, semantics), creating custom camera paths with position and focal length interpolation, $3D$ visualization of captured training images, and crop and export options for point clouds and meshes. Navigation within the scene is facilitated through mouse and keyboard controls. The user interface of *Nerfstudio's* web viewer is depicted in Figure 10 (Tancik et al., 2023).

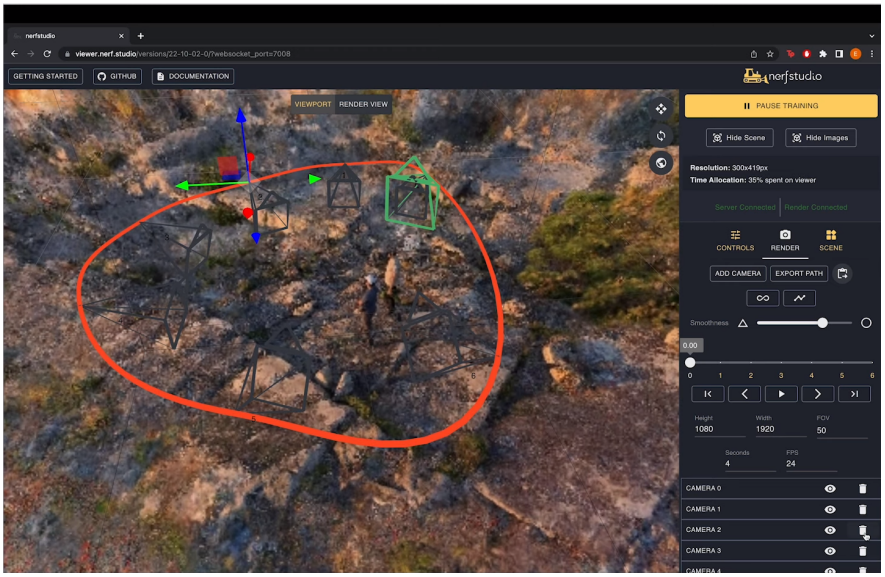


Figure 10.: The web viewer interface of *Nerfstudio*, providing real-time rendering and interactive exploration of the scene during training or model evaluation (Tancik et al., 2023).

Finally, *Nerfstudio* enhances its functionality by supporting various export methods, and the framework allows for easy incorporation of new export techniques as needed.

The export interface, along with some of the supported formats, is demonstrated in Figure 11. These formats include point clouds, a truncated signed distance function (TSDF) to generate a mesh, and poisson surface reconstruction (Kazhdan et al., 2006). The texturing process of the mesh involves dense sampling of the texture image, using barycentric interpolation for identifying corresponding $3D$ point locations, and rendering short rays near the surface along the normals to obtain RGB values.

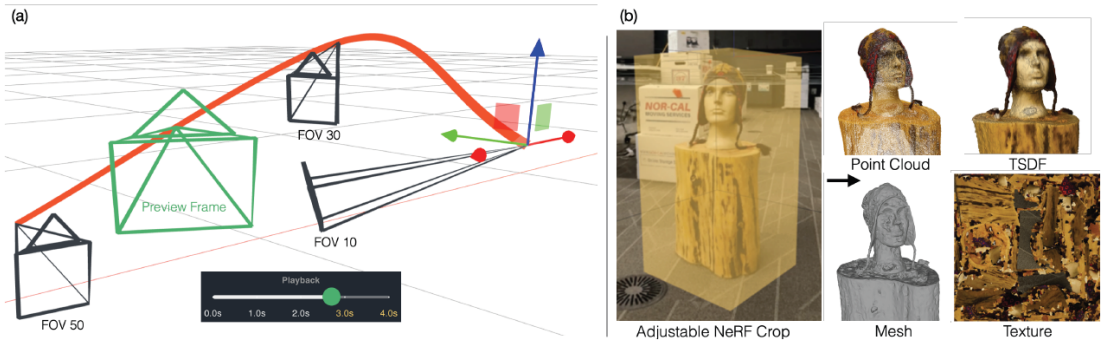


Figure 11.: Exporting videos and geometry with *Nerfstudio*. The interactive camera trajectory editor on the left facilitates the animation of poses, FOVs, and speed for rendering videos of NeRF's outputs. The cropping interface in the viewer and the resulting export formats, such as point clouds, TSDFs, and textured meshes, are presented on the right (Tancik et al., 2023).

Getting started with *Nerfstudio* can be a challenge. Although the framework is very powerful, there are barriers to installing the framework that prevent potential users, especially those with limited technical knowledge, from using this framework.

Firstly, the installation process is far from being user-friendly. It is not as simple as downloading an executable file, but requires the use of Python, CUDA and PyTorch. These three installations are interdependent and their complex installation challenges are discussed in Chapter 4.1.3 and 4.1.4. Not only does this affect the initial setup, but the entire installation process requires a significant amount of technical knowledge and coding skills that may be impossible for a layperson to master. Consequently, this complexity may limit the attractiveness of the framework and restrict its user base to those with the technical skills to manage the process.

Secondly, *Nerfstudio* is built on a command line interface, users cannot simply upload their data via a web viewer and specify the type of training, but have to execute the commands via a command line interface. The process starts with the execution of the command `ns-process-data {video,images,polycam,record3d} --data {DATA_PATH} --output-dir {PROCESSED_DATA_DIR}`, which processes the input data types such as videos, images, Record3D, etc. from the user, to produce the training input format. Subsequently, the command `ns-train {options}` is then used to start training a NeRF algorithm, for example `ns-train instant-ngp {options}`. The option `ns-train --help` reveals additional options, some are shown in Figure 12. This shows the complexity

and versatility of the training options.

Figure 12.: The figure illustrates the range of options available for training the instant-ngp algorithm in *Nerfstudio*, obtained via the `ns-train instant-ngp --help` command. The multitude of options highlights the complexity and flexibility inherent to the *Nerfstudio* platform.

Finally, the training process offers user-friendly features. One of them is the possibility to monitor the training progress via a web viewer. This visualization can be easily interpreted even by non-technical users. It provides an engaging way to observe how the neural radiance fields are created and refined. In addition, after completing the training, users have the option of creating a camera path in the web viewer that can be used for possible video rendering. However, the subsequent stages in the workflow — namely rendering the video or exporting a mesh — necessitate a return to the command line interface. Nonetheless, these commands can be pre-defined and entered through the web viewer, providing a degree of ease. The following code snippet illustrates the sequence of steps in which a point cloud (line 1), a mesh (line 2) and the rendering of a predefined path (line 3) are executed.

```
1 ns-export pointcloud --load-config instant-ngp/2023-07-17_120630/config.yml --output-
    dir exports/pcd/ --num-points 1000000 --remove-outliers True --normal-method
    open3d --use-bounding-box True --bounding-box-min -1 -1 -1 --bounding-box-max 1 1
     1
2 ns-export poisson --load-config instant-ngp/2023-07-17_120630/config.yml --output-dir
     exports/mesh/ --target-num-faces 50000 --num-pixels-per-side 2048 --normal-
    method open3d --num-points 1000000 --remove-outliers True --use-bounding-box True
     --bounding-box-min -1 -1 -1 --bounding-box-max 1 1 1
3 ns-render camera-path --load-config instant-ngp/2023-07-17_120630/config.yml --camera
    -path-filename ./camera_paths/2023-07-17_120630.json --output-path renders
    /./2023-07-17_120630.mp4
```

Code Listing 1: Executing export commands in *Nerfstudio*

In conclusion, *Nerfstudio*, with its comprehensive capabilities, stands out as an essential resource for those committed to keeping pace with advances in NeRF technology. Its extensive features, while demonstrating its power and versatility, can be overwhelming, especially for users who are not well versed in the field. The complexity of its setup, combined with a command line interface that can be daunting for some, highlights a clear gap in user accessibility. Errors and problems occur during installation and execution that users have to fix themselves, which is a daunting task for non-programmers. This not only affects the user experience, but also adds a layer of complexity. While it offers immense potential for experienced researchers and enthusiasts, there's an obvious need for a more intuitive, straightforward alternative for a wider audience. This gap motivates the exploration and development of more a user-centric framework in the field of NeRF technology.

3.3. Torch-NGP - A Pytorch CUDA Extension

In our search for frameworks that could be used to generate neural radiance fields from user-generated data, we also took the approach of looking for alternative implementations that do not rely on specific graphics cards. While this approach may result in slower training and rendering processes, it is consistent with the goal of making the algorithm accessible to a wider audience, including users who do not have high-end graphics cards.

Torch-NGP offers various PyTorch re-implementations of NeRF technologies. Included in its portfolio are PyTorch realisations of the Signed Distance Function (SDF) and NeRF components found in instant-ngp (Müller et al., 2022), TensoRF (Tensorial Radiance Fields, Chen et al., 2022a), CCNeRF (Compressible-composable NeRF via Rank Residual Decomposition, Tang et al., 2022a), and D-NeRF (Neural Radiance Fields for Dynamic Scenes, Pumarola et al., 2020). The framework thus offers not only the algorithm instant-ngp, but also the possibility to prepare self-generated datasets for training input (Tang, 2022; Tang et al., 2022b) and a simple graphical user interface for training/visualizing neural radiance fields (Tang, 2022).

To generate training input from user-generated data, it is imperative to process the data into an appropriate format. This entails creating a JSON file that encapsulates essential camera parameters alongside their corresponding images. In order to effectively prepare and train with self-generated data, the user has to follow these steps:

- Data Acquisition: The procedure starts by capturing visual data, which can take the form of a video or a series of images. The objective is to capture the target scene from a variety of perspectives.

- Data Organization: The captured data should be subsequently organized within a designated repository. For videos, storage occurs in the directory `./data/custom/video.mp4`, while images are stored in the directory `./data/custom/images/` as `*.jpg` files.

- Preprocessing Execution: The execution of a provided script is necessary to facilitate the transformation of the acquired data. Successful execution of the script necessitates the prior installation of both FFMPEG and COLMAP. Upon the execution of the preprocessing script, a `transform.json` file is generated, which encapsulates essential information extracted from the COLMAP process and serves as the fundamental input for subsequent stages of training.

The code snippet 2 shows the commands that need to be executed to get from the user input to the training input.

```
# For video data:
python scripts/colmap2nerf.py --video ./path/to/video.mp4 --run_colmap
# For image data:
python scripts/colmap2nerf.py --images ./path/to/images/ --run_colmap
# For dynamic scenes (suitable for D-NeRF settings):
python scripts/colmap2nerf.py --video ./path/to/video.mp4 --run_colmap --dynamic
```

Code Listing 2: Prepare user-generated data in Torch-NGP

This comprehensive methodology ensures the seamless conversion of user-generated data into a suitable format and thus forms the basis for the subsequent training.

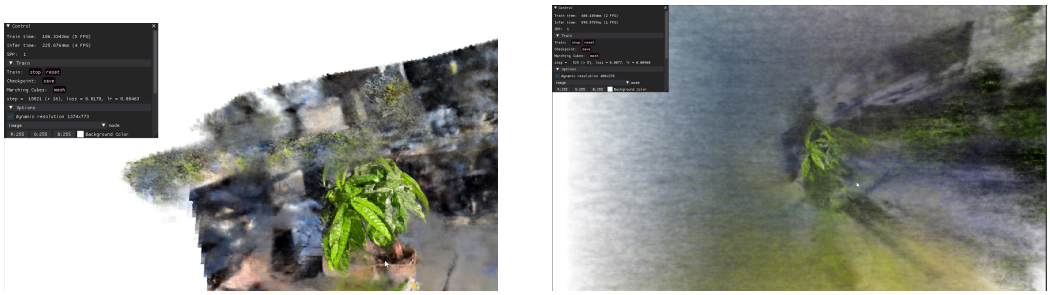
Torch-NGP provides flexibility in choosing the backbone, when training a neural radiance field with the instant-ngp algorithm. The following command-line instructions showcase the different backbone options available. These range from the traditional PyTorch-based Ray Marching method to options tailored for the default COLMAP dataset settings (like –bound 2 and –scale 0.33) and even those leveraging CUDA as the backend (Tang, 2022).

```
# Train with different backbones
python main_nerf.py data/fox --workspace trial_nerf # Standard mode with 32-bit
  floating point precision (fp32)
python main_nerf.py data/fox --workspace trial_nerf --fp16 # Enhanced speed with
  16-bit floating point precision (fp16) using PyTorch's automatic mixed precision
  (AMP)
python main_nerf.py data/fox --workspace trial_nerf --fp16 --ff # Incorporating
  FFMLP in 16-bit floating point precision (fp16) mode
python main_nerf.py data/fox --workspace trial_nerf --fp16 --tcnn # Utilizing the
  official tinycudann's encoder & MLP in 16-bit floating point precision (fp16)
  mode
```

Code Listing 3: Commands in Torch-NGP to train a neural radiance field with the instant-ngp algorithm

Such a diverse set of command options enables users to fine-tune their training strategies to their specific needs, optimizing the performance of the NeRF models they work

with. Figure 13 displays two training results after 5 minutes of training with different backbones. The figure's left side illustrates the progress with a CUDA-backed. In contrast, the right side showcases results without CUDA. A key takeaway from this visual comparison is the substantial speed reduction when opting for PyTorch over CUDA. Even with CUDA's capabilities, the results remain a step behind what the original instant-ngp framework can be seen in 6.



(a) Training progression after 5 minutes with CUDA support

(b) Training progression after 5 minutes without CUDA support

Figure 13.: A visual representation of neural radiance field training using the *torch-ngp* framework. The training data contains 288 images, captured over a 180-degree field of view. The left side 13a details results after 5 minutes of training with CUDA Backend, while the right side 13b does the same for a PyTorch-only session.

Highlighting its user-centric design, *Torch-NGP* includes an intuitive graphical user interface that simplifies the process of training neural radiance fields. By simply incorporating the `--gui` flag, users gain effortless access to the GUI, allowing them to start, stop, and save training sessions with ease. Additionally, the process of mesh exporting is optimized through the integration of the Marching Cubes algorithm (Tang, 2022).

In conclusion, *Torch-NGP* is an excellent framework for users looking for an easy way to generate neural radiance field. However, it is important to recognize that the training process within *Torch-NGP* is significantly slower compared to the original instant-ngp algorithm. Consequently, opting for the original instant-ngp algorithm is the more logical choice in terms of training efficiency (Tang, 2022).

3.4. Luma AI

Launched in mid-2022, *Luma AI* marked its entry into the NeRF domain with the introduction of a private beta. Today, this pioneering application is available on the App Store for users with an iPhone 11 or newer. *Luma AI* boasts a unique feature set that empowers users to craft high-fidelity photo-realistic $3D$ assets and expansive environments swiftly using the NeRF algorithm as basis. Their underlying motivation is eloquently captured in their observation:

> We live in a three dimensional world. Our perception and memories are

> of three dimensional beings. Yet there is no way to capture and revisit those moments like you are there. All we have to show for it are these 2D slices we call photos. It's time for that to change (Yu and Jain, 2023).

Luma AI's vision is not merely to record memories in $3D$ but to revolutionize how products are presented, memories are relayed, and spaces are explored digitally. With the goal of transcending the limitations of traditional $2D$ media, *Luma* aims to establish a vibrant $3D$ mixed reality. Their belief is that with the latest breakthroughs in neural rendering, deep learning, and computational capabilities, achieving photorealistic $3D$ capture is now within reach (Yu and Jain, 2023).

When starting the *Luma AI* app, users will find a user-friendly interface. They have the option of capturing either entire scenes or specific objects. When choosing to capture an object, the app seamlessly guides the user through a systematic process. Figure 14 describes this guiding. It starts with the user delimiting the target object, whereupon an adjustable bounding box appears around the object. This bounding box not only shows the boundaries, but also helps optimize the camera movement. The app then provides explicit guidance on camera movement with a real-time feedback system that highlights the captured camera angles, ensuring thorough data collection while minimizing redundancies. The user circles the selected object three times at different heights using the guide to ensure comprehensive data collection, which serves for optimal NeRF training. After data acquisition, the application moves to the crucial phase of training the neural radiation field. The final image in Figure 14 represents this phase and shows the training progress.

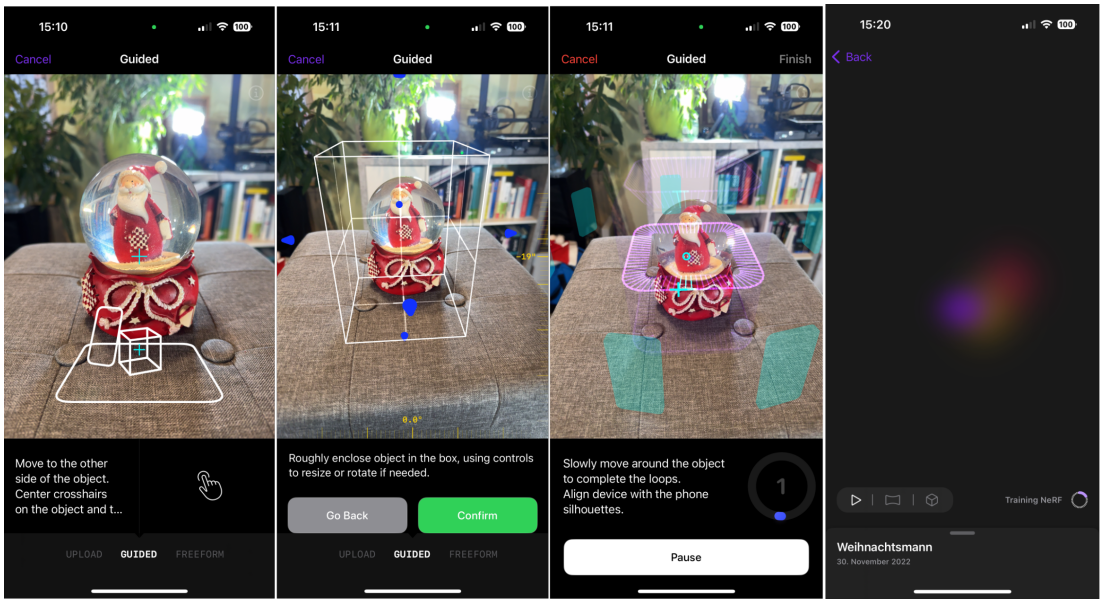


Figure 14.: The systematic capture process within Luma AI: (first) Users designate the object, initiating the bounding box (second) adjustable to match the object's dimensions. Subsequent steps guide the user on camera movement, showcasing captured angles (third). The neural radiance field's training status is represented in illustration four.

Figure 15 shows the result, where the user can view a high-quality, $3D$ neural radiance field representation of the captured object.

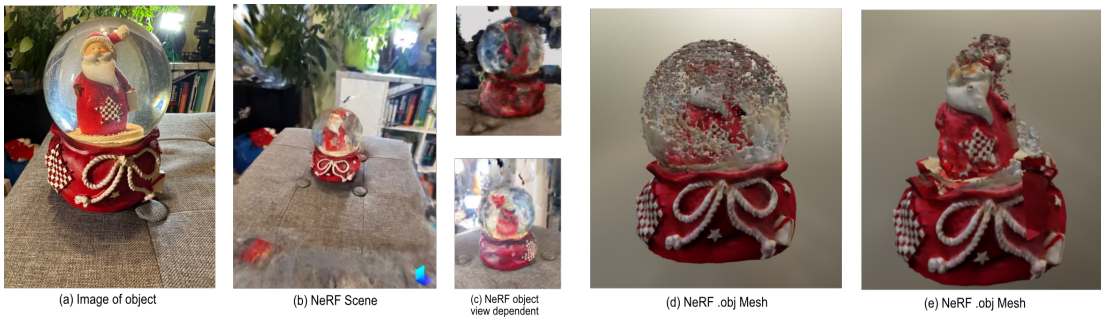


Figure 15.: The comprehensive visualization of the neural radiance field's training outcome for the Santa Claus object. Figure (a) is a reference photo of the object. (b) is the neural radiance field scene. (c) is an illustration of the view dependency, highlighting the significance of this aspect with the NeRF algorithm. (d)shows the extracted mesh showcasing the Santa Claus encapsulated within the transparent sphere. (e) highglights the Mesh without the surrounding sphere, emphasizing NeRF's capability to manage transparency; even with the glass removed, the central object remains intact, validating the efficient rendering of the glass alongside the contained object.

A noteworthy aspect, already mentioned in the introduction, is the competence of NeRF in dealing with transparency. As can be seen in Figure 15 (e), Santa Claus remains well defined even without the surrounding glass, indicating an effective representation of both the glass and the figure within it. Furthermore, illustration (c)

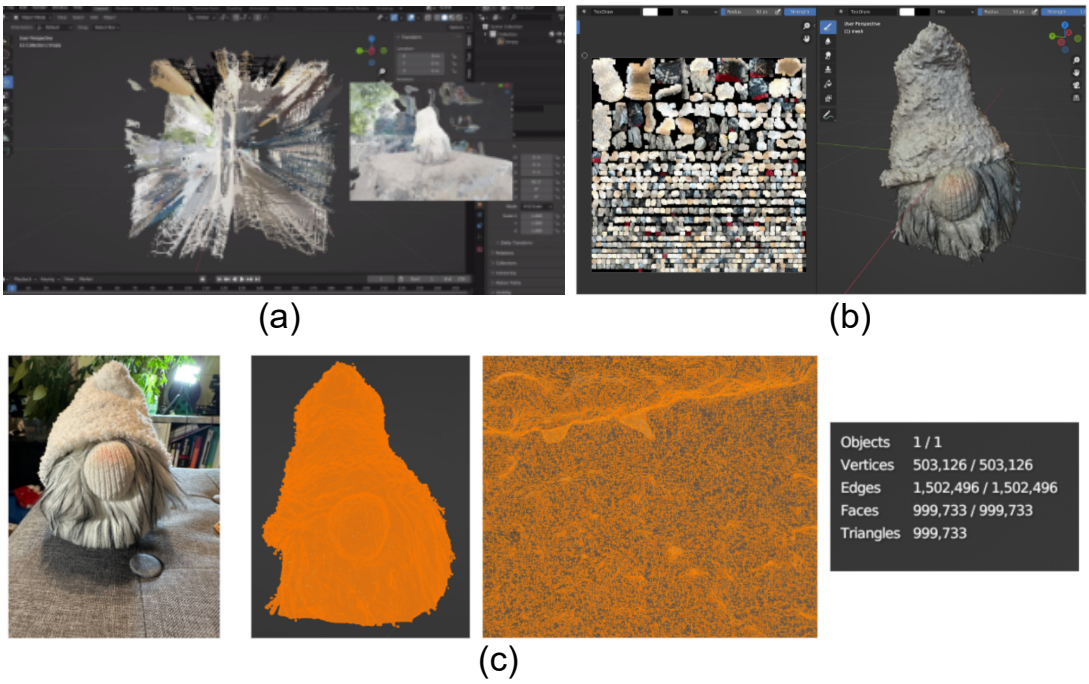


Figure 16.: A showcase of LUMA AI's export features and inherent challenges: (a) Point cloud depiction requiring post-process object extraction, (b) UV-Map of the exported texture, and (c) High-density mesh, highlighting rendering performance concerns.

shows the clear influence of view dependence on glass rendering.

LUMA AI facilitates a variety of export options for the trained radiance field. Users can export scenes as point clouds in PLY files, as objects with varying polygonal details as OBJ files, or even utilize an integration plugin for Unreal Engine, underscoring its potential for gaming and simulations. An additional feature lets users chart camera paths to generate videos.

However, while these features seem promising, they come with their set of challenges in *LUMA AI*. Exporting as a point cloud, for instance, can yield a scene with millions of points, demanding post-processing for object delineation, as depicted in Figure 16(a). The derived meshes, especially from dense point clouds, can contain millions of faces, which can be detrimental to real-time rendering performance, as shown in Figure 16(c). Moreover, the neural radiance field doesn't always produce ideal results when depicting larger scenes, as shown in Figure 17. Aliasing artifacts become evident, and it often becomes challenging to distinctly perceive the scene from various viewpoints.

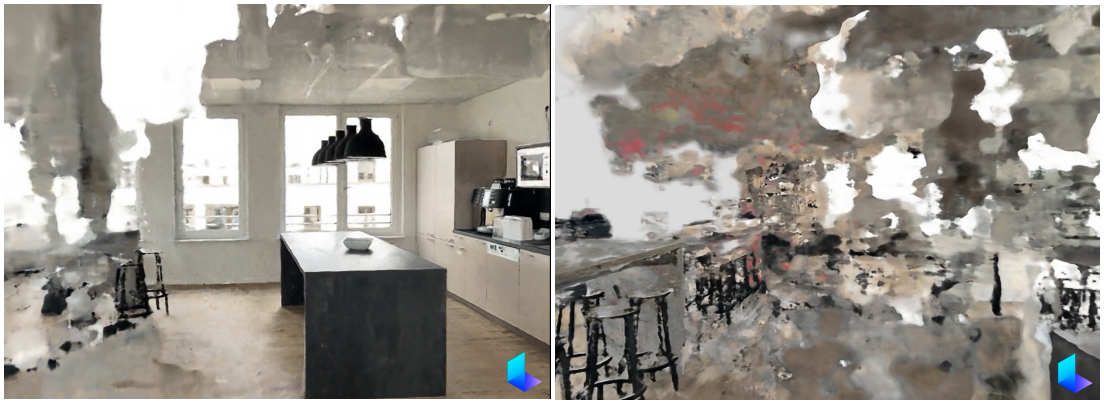


Figure 17.: *LUMA AI's* challenges with rendering large scenes: The left image presents the most accurately rendered view of a scene, while the right image reveals less-than-optimal renderings, marked by artifacts and overall challenges in discerning the scene. This juxtaposition highlights *LUMA AI's* constraints when dealing with large-scale environments.

In the context of utilizing LiDAR sensors for creating neural radiance fields, challenges arises when encountering minor recording errors. These errors can lead to complexities in accurately comprehending the captured scene within the *Luma AI* app. As illustrated in Figure 18, these subtle recording inaccuracies can result in difficulties during scene interpretation and reconstruction.

Figure 18.: *LUMA AI's* challenges: The left image presents the most accurately rendered view of a scene, while the right image reveals less-than-optimal renderings. This image underscores that even minor recording inaccuracies can lead to significant distortions.

LUMA AI has rapidly established itself as an innovative NeRF application since its launch in mid-2022, enabling users to create high-quality, photorealistic $3D$ assets and environments in a short time. Its user-friendly design steers individuals seamlessly through a systematic capture process, aimed at deriving optimal NeRF models. However, despite the impressive suite of functionalities it offers, *LUMA AI* is not without its limitations, particularly evident when tackling expansive scenes or exporting intricate models. Recent endeavors, such as the creation of a music video leveraging

LUMA AI's capabilities, further highlight its potential in reshaping multimedia experiences. Notably, the original developer of NeRF, Matthew Tancik, has become an integral part of this emerging start-up. With heavyweights like NVIDIA backing the vision, it sees a promising foundation forming for *LUMA AI* (Yu and Jain, 2023).

> "Luma is one of the first to bring to market new 3D NeRF capture technology, enabling non-technical users to create higher quality 3D renders than has previously been possible. The team has consistently demonstrated the ability to rapidly productize new research, and their long-term vision of democratizing access to 3D workflow tools for consumers has the potential to drastically lower the skills barrier to create & edit in 3D (Mohamed Siddeek, NVIDIA (NVentures))."

3.5. The instant-ngp Framework

Following the release of NVIDIA's research on neural graphics primitives (Müller et al., 2022), NVIDIA has not only made the algorithm available as open-source code but has also introduced a comprehensive framework encompassing all necessary components, thereby offering an almost complete end-to-end solution. This framework offers pre-installations for popular NVIDIA graphics cards, a feature-rich GUI, and provides scripts accompanied by thorough explanations for preparing user-generated input data for the NeRF training. In the subsequent sections, we will delve deeper into the array of features and functionalities presented by instant-ngp.

NVIDIA provides pre-built installations for Windows users. People working on Linux platforms, who require special Python bindings, or who have GPUs such as Hopper, Volta or Maxwell iterations, will need to manually build instant-ngp. The instant-ngp framework is equipped with an interactive GUI, shown in Figure 19, which includes a wide range of features, each designed to enhance user interaction and functionality.

Figure 19.: Illustration of the *instant-ngp* framework. This framework is armed with an interactive GUI that boasts a diverse array of features, all thoughtfully designed to amplify user interaction and enhance functionality. On the right side of this image is a GUI where users can make changes, save and load "snapshots", export a mesh and many more features e.g. debug options. The GUI on the bottom left contains a camera path editor that allows users to create custom videos and make adjustments to the camera for rendering (NVlabs, 2022).

These features include comprehensive controls that enable users to engage with neural graphics primitives in an interactive manner. Additionally, the framework offers a Virtual Reality and Augmented Reality mode, facilitating the visualization of neural graphics primitives through the use of virtual/augmented-reality headsets. Users also have the capability to save and load "snapshots," simplifying the sharing of graphics primitives online. The GUI further incorporates a camera path editor, empowering users to create customized videos, and supports seamless conversion between NeRF and Mesh representations. Notably, the framework provides tools for optimizing camera pose and lens parameters, ensuring optimal visual outcomes. In addition to these features, the instant-ngp GUI encompasses a host of other functionalities, all of which contribute to an enriched user experience while expanding the framework's overall capabilities. Figure 19 illustrates the instant-ngp framework (NVlabs, 2022). The GUI functions used in this thesis, such as the mesh export, are discussed in chapter 5.

In order to use instant-ngp on custom data, users should first establish a Python environment that includes the necessary libraries, as listed in the GitHub repository. The data, whether videos, images, or depth information, must then be prepared for training. This involves computing the camera parameters and storing them in a JSON format. To streamline this process, developers have crafted a Python script, `colmap2nerf.py`. This tool enables users to process video files or image sequences using the open-source COLMAP structure, which then outputs a JSON file. The script

needs specific arguments as inputs, such as the path to the video file and others. Certain arguments, such as `aabb_scale`, explained in further detail in this thesis, are crucial for achieving high-quality neural radiance fields but may not be intuitively understood. The following code snippet provides an example on how to execute the script. It is crucial for users to comprehend these arguments and configure them appropriately to achieve the desired results.

```
python [path-to-instant-ngp]/scripts/colmap2nerf.py --video_in <video filename> --
  video_fps 2 --run_colmap --aabb_scale 32
```

Code Listing 4: Execute the colmap2nerf.py script from instant-ngp

Given the challenges involved, particularly for users without a technical background, the process of creating neural radiance fields using the instant-ngp framework can be difficult and time-consuming. Users must navigate through the intricacies of installation, data preparation, parameter selection for the JSON file generation, and subsequent network training. Furthermore, the possession of an NVIDIA GPU is a prerequisite, which may not be readily available to all users. These factors significantly hinder the accessibility of this framework. It is also important to note that the installation process and requirements of the instant-ngp framework are also dependent on the user's operating system and the versions of the required dependencies. This adds an additional layer of complexity, as users need to ensure compatibility with their specific setup.

Following the successful initialization and preparation of the necessary parameters, one is poised to commence the NeRF model training. This can be initiated from the instant-ngp directory. Alternatively, for those who prefer a more direct graphical interaction, the designated .exe executable provides a user interface wherein one can easily integrate both the image datasets and the transforms.json configuration:

```
instant-ngp$ ./instant-ngp [path to training data folder containing transforms.json]
```

The scripts, `colmap2nerf.py` and `record3d2nerf.py`, operate under the premise that training images predominantly orient themselves towards a common focal point, which the scripts designate at the coordinate origin. This focal point is discerned by computing a weighted mean of the nearest points of intersection among rays emanating from the central pixel across all training image pairs. Operationally, this suggests an optimal performance when the training images are strategically captured with an inward orientation towards the primary object, even if they don't encapsulate a comprehensive 360-degree view. It should be noted that any backdrop behind the primary object remains subject to reconstruction, especially if the *aabb_scale* parameter surpasses a unitary value, as previously elaborated.

Figure 20 shows the training process using instant-ngp. For the purpose of debugging, cameras and the unit cube representing a scale of 1 have been mapped.

Figure 20.: This illustration shows the training in instant-ngp. Within the debugging visualization context, both the cameras and the unit cube—characterized by a scale of 1 - are incorporated (NVlabs, 2022).

3.6. Discussion

The instant-ngp framework stands out for its extensive features and the best implementation of the instant-ngp algorithm with CUDA. Alternative implementations such as *Torch-NGP* train more slowly and do not offer the same feature set. The primary goal of this thesis is to design a framework that allows users to efficiently train neural radiance fields with the instant-ngp algorithm, which is why *Nerfstudio* and similar frameworks are too extensive for the context of this thesis.

Throughout our research, we have taken an industrial perspective into account, acknowledging factors like the availability of administrator rights, which might not be universally accessible in an industrial environment, and the requirement of possessing an NVIDIA graphics card with RTX capabilities. We've also considered the scenario where users may lack coding expertise or familiarity with command-line interfaces. This exploration underscores the observation that existing frameworks, especially in an industrial context, do not offer a straightforward and efficient solution for generating neural radiance fields without the specific prerequisites mentioned above. Moreover, numerous current frameworks assume the availability of training input and do not extensively address this facet. Although methods to generate input from videos do exist, these steps often involve manual execution of commands or the use of external tools such as Colmap or FFMPEG. Given these intricacies, our decision was to develop a customized framework. Notably, our framework drew inspiration from the explored frameworks. For instance, *ArcNeRF* contributed insights into logger design and configuration files, while the approach taken in *instant-ngp* provided the foundation for transitioning from video to input generation. In the next chapter (4), we take a closer look at the specific requirements that led us to embark on our unique implementation journey.

4. Implementation

As highlighted in chapter 3, the instant-ngp framework emerges as the most efficient and optimal solution for generating neural radiance fields with the multiresolution hash encoding. Along with its core functionalities, the framework furnishes auxiliary scripts that enable the transformation of self-captured data into a format suitable for training. To manually install the framework from NVIDIA's GitHub, users must need certain hardware and software requirements, including an NVIDIA GPU with CUDA support, CUDA version 11.6 or higher, Python, CMake, and a compatible C++ compiler. Although NVIDIA offers pre-built installations for common graphics cards, users will still need to convert user-generated data by running the auxiliary scripts with additional arguments as input. Industrial users face significant challenges in installing and using the framework. Many do not have the necessary administrative rights to install or run the framework, and additionally they can find themselves lost in the complexity, not just in comprehending the python commands but also in choosing the right parameters as input. Given these challenges and the complexity of data preparation, we saw the need for a custom pipeline that would automatically create a neural radiance field from user-generated data. This pipeline is tailored to convert diverse data formats - such as videos, zipped image collections, and r3d captures from LiDAR-equipped iPhones - into neural radiance fields.

In our exploration and installation of the academic projects presented in Chapter 3, it became evident that many of them are linked to certain prerequisites. Some projects require specific software and package versions, e.g. Python 3.7 or PyTorch 1.10+, others can only be installed on Ubuntu, while other installation procedures depend on the operating system used. A key priority for us was therefore to ensure that this project was compatible across different operating systems and software and package dependencies. To this end, we used tools such as Poetry and Docker to help us orchestrate dependencies and maintain a consistent environment. This allows the entire project to be easily installed locally without having to address specific requirements.

Another main objective of our project was to provide users a platform without diving into complicated installation or programming tasks. With this in mind, we developed a front-end React application where users can easily upload data, adjust parameters, and download the resulting neural radiance field as a `.ingp` file, which can be used within the instant-ngp framework's GUI (Müller et al., 2022), enabling further exploration. The back-end, built in Python, can run on any server equipped

with an NVIDIA RTX GPU, eliminating the need for a local graphics card. This combination of a user-friendly interface and a seperate back-end makes it simpler for users to create high-quality neural radiance fields. Figure 21 showcases the "SICK - NeRF" webpage, which has been designed with user experience in mind.

In this chapter, we detail the methodologies and tools employed in our project. We begin with an overview of how we achieved system-independence and package management, specifically using Docker and Poetry. We then discuss key aspects on how our project works, followed by a detailed explanation of each pipeline stage, from image extraction to the training with instant-ngp. Following these, we delve deeper into developed of the web-based interface using Flask and React. The chapter concludes with a summary of the key points discussed.

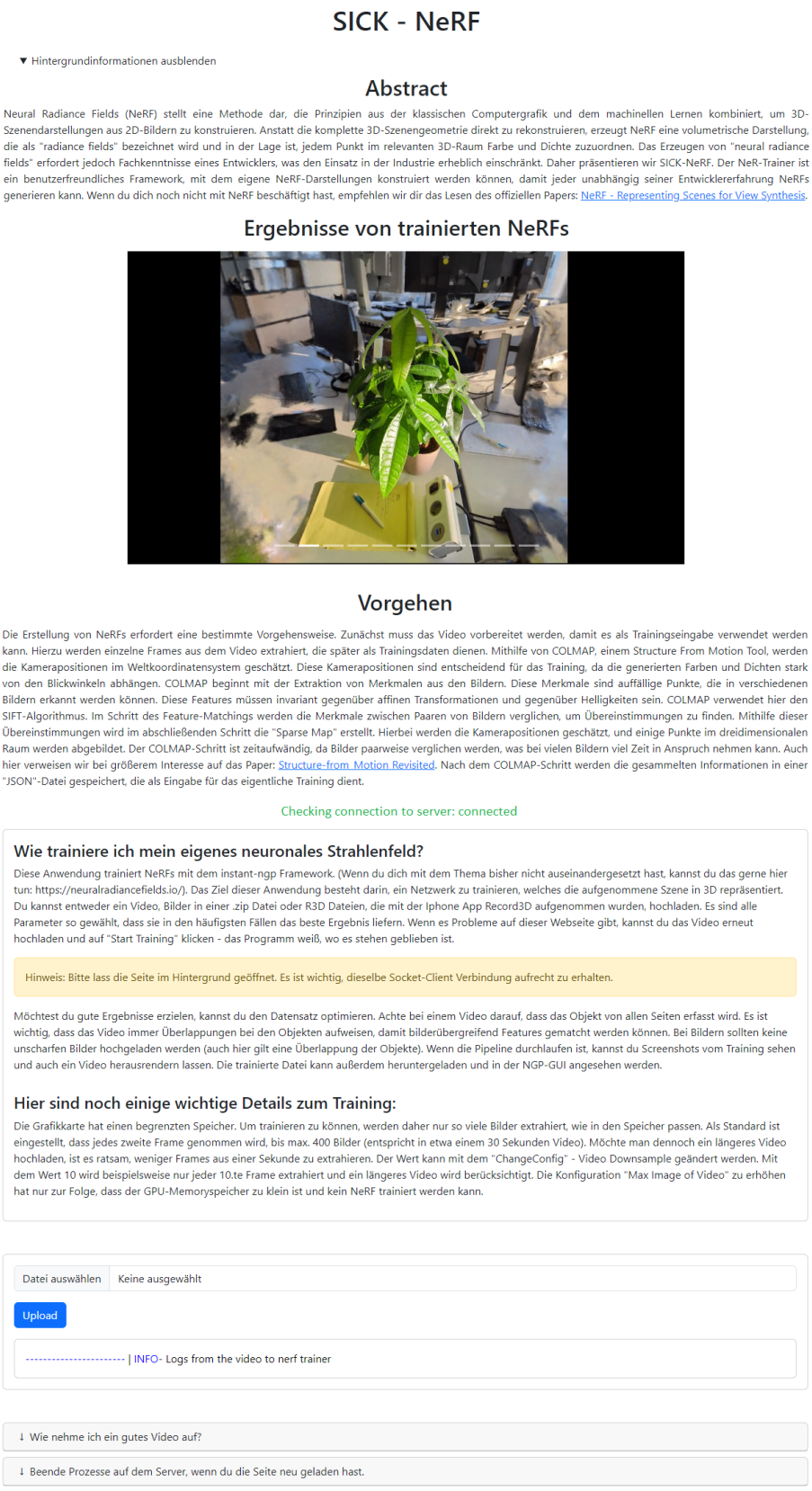


Figure 21.: The "SICK - NeRF" website first presents an abstract, followed by a video slider with results of trained neural radiance fields. Below is an information section describing how the pipeline works. The entire background section can be collapsed to focus on the main section. The main part of this page consists of the container box with an upload form that can be used to upload data and then train the neural radiance fields. Further down, expandable information accordions provide additional insights.

4.1. The system-independent and package-managed Installation

We wanted to ensure that the project could be installed on different systems without having to resolve dependencies. This section deals with the implementation of a system-independent and package-managed project for the custom pipeline with Poetry and Docker. Section 4.1.1 goes into the use of Poetry and highlights the benefits it offers in terms of package management. Furthermore, the integration of Docker, a containerization platform, improves the portability and consistency of the custom pipeline installation. Section 4.1.2 explores Docker and explains its role in achieving an independent system installation. By combining the capabilities of Poetry and Docker, the custom pipeline ensures a seamless and reliable installation process, regardless of the user's operating system and dependency versions.

4.1.1. Poetry

Efficient packaging and effective dependency management are important components in software development, contributing significantly to creating robust and maintainable projects. In the field of Python development, a variety of tools have been introduced to meet these needs, with Anaconda and Poetry being two popular choices, each offering unique strengths and areas of application. Anaconda, particularly favored in data science, simplifies access to a host of scientific libraries and packages, such as SciPy, NumPy, and TensorFlow. It is also well-known for its robust environment management, enabling users to create isolated environments for different projects to prevent dependency conflicts. Moreover, Anaconda provides a user-friendly graphical interface through the Anaconda Navigator, making it accessible for developers less comfortable with command-line operations (Anaconda, 2023). However, for this particular project, we chose Poetry, a command-line tool to address the dependency challenges specifically in Python development. The primary focus of Poetry is to provide a comprehensive and user-friendly experience, prioritizing simplicity, ease of use, and consistency. With Poetry, we can easily define project dependencies, manage virtual environments, build and package projects, and even publish them to the Python Package Index (poetry, 2023).

A typical Poetry project is structured with key directories and files essential for its functionality. The root directory, named `nerf-industrial-meverse` in this instance, serves as the project's primary identifier. Central to a Poetry-managed project is the `pyproject.toml` file, which dictates dependency management, build information, and other metadata. Accompanying this is the `README.md`, providing documentation or instructions pertinent to the project. The directory `nerf_industrial_meverse` likely contains the core modules and functionalities of the project. Its presence, marked by an `init.py` file, designates it as a Python package, facilitating the organization of related modules and allowing them to be easily imported. Similarly, the `tests` directory, also initialized with an `init.py` file, is conventionally used to house unit tests, integration

tests, or other testing scripts to validate the project's functionalities (poetry, 2023).

```
nerfindustrialmetaverse
├── pyproject.toml
├── README.md
├── nerf_industrial_meverse
│   └── __init__.py
└── tests
    └── __init__.py
```

Diving deeper into the core elements, the `pyproject.toml` file stands out as an instrumental component. It acts as an orchestrator for the project and its dependencies. This file not only contains important configuration details but also serves as a central hub for managing various aspects of the project (poetry, 2023). In particular, it's able to navigate complex dependency structures and automatically resolve any conflicts that may arise, making the development process more seamless than with alternatives such as Anaconda. The file starts with metadata information about the project, as shown below:

```
[tool.poetry]
name = "nerf_industrial_metaverse"
version = "0.1.0"
description = ""
authors = ["Sabine Schleise <sabine.schleise@sick.de>"]
readme = "README.md"
packages = [{ include = "nerf_industrial_metaverse" }]
```

Code Listing 5: Metadata in Poetry

To integrate the required packages, we can use the command `poetry add numpy`. This not only updates the `pyproject.toml` but also installs the packages into the virtual environment. Similar to Anaconda, Poetry promotes the creation of isolated environments for different projects. We have used this encapsulated setup to ensure that our project and its companions remain separate from the central Python setup on the system, giving us a clear overview of the active packages (poetry, 2023). The subsequent code sample 6 illustrates how dependencies are declared within the project.

```
[tool.poetry.dependencies]
python = "^3.9.13"
ffmpeg-python = "^0.2.0"
scipy = "1.9.3"
opencv-python-headless = "^4.7.0.72"
```

Code Listing 6: Poetry's dependency management

By registering packages in the `pyproject.toml`, we can specify exact version numbers or define flexible version ranges. This approach facilitates smooth updates and ensures compatibility. In the `[tool.poetry.dependencies]` section, each dependency is matched with a version or a version constraint expressed using semantic versioning notation. The ^ symbol indicates that updates within the specified major version are allowed. For instance, ^3.9.13 permits any version within the 3.x.x range, restricting upgrades

to the major version. While the `pyproject.toml` indicates desired versions or version ranges, the actual versions installed might vary based on these constraints and external package updates.

Packages and their respective versions can also be directly appended to the `pyproject.toml` file and then installed by using the `poetry install` command for installation. We can also just add the package and it's version to the `pyproject.toml` file and then install them with `poetry install`. Upon completion, Poetry generates a `poetry.lock` file, which captures the exact versions of all project dependencies used during development, acting as a reference for all future installations, ensuring that identical versions are installed and eliminating unexpected changes due to dependency upgrades. By including the `poetry.lock` file in version control and sharing it with collaborators, developers can achieve consistent and reliable installations across various environments (poetry, 2023).

Managing dependencies in this way provides several benefits. It ensures that the project is built on the intended versions of its dependencies, reducing the risk of compatibility issues. It also allows for controlled updates, as developers can choose to update dependencies within the specified version ranges when desired, while avoiding unexpected or incompatible upgrades.

In our project, we took also advantage of Poetry's capability to define and execute custom scripts. This feature, as exemplified in code snippet 7, is housed within the `[tool.poetry.scripts]` section of our `pyproject.toml` file. Each script we defined, for instance, `ngp-colmap`, is mapped to a particular function within our project. Specifically, executing `ngp-colmap` will trigger the `nerf_industrial_metaverse.tools.run_poses:entrypoint` function we implemented.

```
[tool.poetry.scripts]
ngp-extract="nerf_industrial_metaverse.tools.extract_video:entrypoint"
ngp-colmap="nerf_industrial_metaverse.tools.run_poses:entrypoint"
ngp-json="nerf_industrial_metaverse.tools.export_json:entrypoint"
ngp-iphone="nerf_industrial_metaverse.tools.record3d_to_json:entrypoint"
ngp-train="nerf_industrial_metaverse.scripts_ngp.run:entrypoint"
ngp-trainvideo="nerf_industrial_metaverse.tools.video_to_training:entrypoint"
```

Code Listing 7: Customized commands for executing the pipeline steps

Running these customized scripts can either be done with the command `poetry run [script-name]` or, if the project has been installed with `pip install -e .`, by running `ngp-extract`. We integrated these user-defined commands for instance in our python scripts using the subprocess package. The `subprocess.check_output()` function allows for running a command and capturing its standard output. The provided code snippet 8 illustrates the whole pipeline process implemented in Python, which can be executed with the custom script `ngp-trainvideo`.

```
  if not valid_key_in_cfgs(cfgs, 'iphone'):
    if not osp.isdir(osp.join(scene_dir, "images")):
```

```
      extract_video_cmd = ["ngp-extract", "--configs", args.configs]
      extract_video_output = subprocess.check_output(
      extract_video_cmd, universal_newlines=True)
    if not osp.exists(osp.join(scene_dir, "colmap_output.txt")):
      run_colmap_cmd = ["ngp-colmap", "--configs", args.configs]
      run_colmap_output = subprocess.check_output(
      run_colmap_cmd, universal_newlines=True)
    if not osp.exists(osp.join(scene_dir, cfgs.json.name)):
      logger.add_log('--- EXPORT TRANSFORM.JSON ---')
      export_json_cmd = ["ngp-json", "--configs", args.configs]
      export_json_output = subprocess.check_output(
      export_json_cmd, universal_newlines=True)
  else:
    if not osp.exists(osp.join(scene_dir, "metadata")):
      exit()
    if not osp.exists(osp.join(scene_dir, cfgs.json.name)):
      export_json_cmd = ["ngp-iphone", "--configs", args.configs]
      export_json_output = subprocess.check_output(
      export_json_cmd, universal_newlines=True)
if osp.exists(osp.join(scene_dir, cfgs.json.name)):
  train_ngp_cmd = ["ngp-train", "--configs", args.configs]
  train_ngp_output = subprocess.check_output(train_ngp_cmd, stdin=None, stderr=None,
    universal_newlines=True)E ---')
```

Code Listing 8: Converting input data to input for the training

In summary, we used Poetry as the solution to manage the complexity of the project and its dependencies. With the `pyproject.toml` file, we achieved a centralized setup that gave us the precision we needed to handle dependencies and their respective versions. In addition, Poetry's ability to detect and resolve conflicts proved invaluable. We further improved our workflow by taking advantage of Poetry's ability to create virtual environments, which strengthened the stability and reproducibility of our project by isolating it from potential system-level Python installations. Poetry also paved the way for us to efficiently customize and execute specific scripts. In contrast to tools like Anaconda, we found Poetry to be a more cohesive and streamlined choice for our project management needs, while still being lightweight and easy to use (poetry, 2023).

Built-in Package Publishing While we did not utilize this particular feature in our project, it's worth noting that Poetry isn't just for dependency management - it also excels in packaging and distribution. Poetry offers a comprehensive solution for building and publishing packages to the Python Package Index (PyPI) with a single, easy-to-use command: `poetry publish project-demo`. This feature provides a unified and streamlined interface for managing the entire lifecycle of a package. Furthermore, from pip version 10.0 onwards (pip developers, n.d.), users can install the project locally using the command `pip install -e .`. This action leverages the `pyproject.toml` file to build the project, and it sets the stage for executing custom scripts described in the subsequent section.

4.1.2. Docker

Docker is a platform that uses open source technology to facilitate the automated deployment and management of applications using lightweight, isolated containers. Each of these containers acts as a self-contained unit, housing both the application and its dependencies. One of the primary purposes of these containers is to ensure a consistent behavior across various environments. By design, Docker containers offer isolation, meaning each application operates in its unique sandboxed environment, free from interference by other applications or the underlying system. Containers also can be easily moved between different environments, such as development, testing, and production, without the concern of compatibility issues. When a container runs, it uses a dedicated and isolated file system, which is derived from a container image. This image is encapsulating everything required for the application to function - dependencies, configurations, scripts, binaries, and more (Docker, 2023).

To set up such an image, with just our needs in it, we created a `Dockerfile`, which serves as a blueprint for building images. The `Dockerfile` starts with a specification of the base image that will be used as the foundation. For example, `FROM node:18-alpine` specifies that the base image for the Docker container will be derived from the "node" image with version 18, specifically using the Alpine Linux distribution. This base image provides a minimal and lightweight environment (Docker, 2023). In this project, we used the NVIDIA CUDA 11.8 development image, specifically `nvidia/cuda:11.8.0-devel-ubuntu22.04`, as base image. This image provides a pre-configured environment with CUDA version 11.8. The `devel` tag indicates that the image includes additional development packages and libraries, making it suitable for software development and compilation tasks involving CUDA. Moreover, the image is based on the Ubuntu 22.04 operating system, ensuring compatibility and familiarity with the Ubuntu environment for developers. This specific combination of CUDA version, development packages, and Ubuntu OS was selected based on the project requirements and the need for a CUDA-enabled development environment.

Once the base image is set, Docker allows developers to take further steps in refining their image. Within the `Dockerfile`, certain keywords like `ARG`, `ENV`, `RUN`, and more, empower developers to make additional installations and configurations, thereby customizing the container to their liking (Docker, 2023). The `ARG` keyword enables the definition of variables that can be passed as arguments during the Docker build process. This allows developers to specify dynamic values that can be utilized during container creation. The `ENV` keyword facilitates the setting of environment variables within the container. These variables can be accessed by the application running inside the container, providing a convenient way to configure its behavior. The provided code snippet 9, as indicated in snippet 9, demonstrates the initial setup and configuration

in our `Dockerfile`.

```
FROM nvidia/cuda:11.8.0-devel-ubuntu22.04

ARG COLMAP_VERSION=dev
ENV CERES_SOLVER_VERSION=2.0.0
ARG CUDA_ARCHITECTURES=89
ENV PATH=/usr/local/cuda/bin:$PATH
```

Code Listing 9: Building a Docker Image

The `RUN` keyword is used to execute specific commands within the container during the build process. This enables developers to perform installations, set up dependencies, and carry out various configuration tasks to ensure the container has all the necessary components for running the desired application. Each `RUN` command is executed independently and in sequence, the && operator can be used to connect them (Docker, 2023). The next code snippet (10) demonstrates a `RUN` command using Ubuntu as the base system. This command updates the system's package index and then installs the package `python-is-python3`. Once installed, this package ensures that the python command within the container will point to Python 3, streamlining any subsequent Python-related operations.

```
  RUN apt-get update && apt-get install -y python-is-python3
```

Code Listing 10: RUN command in Docker

In addition to these keywords, Docker also provides other essential keywords such as `ADD`, `COPY`, and `ENTRYPOINT`. The `ADD` and `COPY` keywords allow developers to transfer files and directories from the host machine into the container, facilitating the inclusion of additional resources. The `ENTRYPOINT` keyword specifies the default command that will be executed when the container is run, allowing developers to define the primary executable or script for the application (Docker, 2023). The code snippet 11 shows the `Dockerfile` we created for this specific project. For brevity, we have excluded a few lines. The full `Dockerfile` is available as an attachment.

```
FROM nvidia/cuda:11.8.0-devel-ubuntu22.04

ARG COLMAP_VERSION=dev
ARG CUDA_ARCHITECTURES=89
ENV PATH=/usr/local/cuda/bin:$PATH

# Prevent stop building ubuntu at time zone selection.
ENV DEBIAN_FRONTEND=noninteractive

# Prepare and empty machine for building.
RUN apt-get update && apt-get install -y \
python-is-python3 \
python3-pip \
ffmpeg \
git \
cmake \
ninja-build \
build-essential \
```

```
[...] \
libceres-dev \

# Install latest Node.js version
RUN mkdir temp && cd temp && wget https://nodejs.org/dist/v16.16.0/node-v16.16.0-
    linux-x64.tar.xz && \
tar -xf node-v16.16.0-linux-x64.tar.xz --strip-components=1 -C /usr/local
ENV PATH="/usr/local/bin:${PATH}"
RUN rm -r temp

RUN echo "Installing COLMAP ver. ${COLMAP_VERSION}..."
RUN git clone https://github.com/colmap/colmap.git
RUN cd colmap && \
git reset --hard ${COLMAP_VERSION} && \
mkdir build && \
cd build && \
cmake .. -GNinja -DCMAKE_CUDA_ARCHITECTURES={CUDA_ARCHITECTURES} && \
ninja && \
ninja install && \
cd .. && rm -rf colmap

# Build instant-nerf wihout gui
RUN echo "Installing Instant-NeRF"
RUN  git clone --recursive https://github.com/nvlabs/instant-ngp
RUN cd instant-ngp && \
cmake -DNGP_BUILD_WITH_GUI=off ./ -B ./build && \
cmake --build build --config RelWithDebInfo -j 16

ADD . nerf-industrial-metaverse
RUN cp instant-ngp/build/pyngp.cpython* /nerf-industrial-metaverse
RUN cd nerf-industrial-metaverse && pip install -e .
```

Code Listing 11: Dockerfile used for this project

After defining the base image and initializing the required variables, we proceed to install essential packages within the container. This is followed by the integration of the latest version of `node.js`, essential for our front-end application. Next, the COLMAP repository is cloned, and through a sequence of commands, we build and install it with the utilization of the ninja build system. Special attention is given to the compilation of instant-ngp, excluding the GUI. The decision to exclude the GUI is rooted in the inherent complexities and challenges associated with running graphical interfaces within Docker containers. Docker, by design, emphasizes the isolation of containerized applications from their host system. As a result, GUI applications, which require direct interaction with the host's X-Server, encounter barriers. Moreover, allowing such interaction raises security concerns, as the X-Server lacks protective measures against applications accessing it. Following the build of instant-ngp, we migrate our entire project to the "nerf-industrial-metaverse" directory within the Docker image. The final touch involves employing pip to install the project, ensuring that our customizes scripts from "nerf-industrial-metaverse" can be executed. With this `Dockerfile`, we've established a container environment precisely tailored to our project's requirements.

After we successfully configured and created the `Dockerfile`, we can build an im-

age using the `docker build` command. Specially the `docker build nerf_metaverse -f Dockerfile .` instructs Docker to build an image named *"nerf_metaverse"* based on the instructions specified in the `Dockerfile`.

Once the image is successfully built, we initiate the container with `docker run`, more precisely, we use `docker run --gpus all --rm --it name_of_container` to run our container. The `--gpus all` flag ensures that the container uses all available GPUs on the system. The `--rm` flag tells Docker to remove the container when it is finished, optimizing resource management. The `--it` flag stands for "interactive terminal", which allows users to interact directly with the container via the terminal.

Docker containers serve as self-contained environments that replicate the exact specifications defined in their corresponding `Dockerfile`. Upon initialization, a Docker container provides a tailored setting that includes the pre-configured tools and utilities. However, it's crucial to understand the inherent characteristics of these containers. Once a Docker container is instantiated, it provides users with an environment equipped with all the tools and utilities described in the `Dockerfile`. This ensures immediate and easy access to these tools without the need for further setup. On the other hand, any tools or utilities not specified during image creation will remain unavailable. Users can install additional tools within an active container. However, due to the isolated nature of Docker containers, all modifications within the container, whether they be configurations, installations, or file changes, are transient. This means that upon termination or restart of the container, these modifications are discarded, returning the container to its original state as defined by the image. For lasting changes to be integrated, they must be incorporated into the `Dockerfile` and the Docker image must subsequently be rebuilt.

4.1.3. CUDA

Poetry and Docker helped us to create a robust and maintainable project. However, the instant-ngp algorithm's dependence on CUDA, a low-level programming language developed by NVIDIA, to optimize GPU efficiency, can introduce complex challenges that are not easily resolved. While Docker is designed to encapsulate and isolate its environment from the host, GPU acceleration requires more direct interaction with the hardware. To utilize CUDA within a Docker container, the NVIDIA Container Toolkit must be installed on the hosting system. This toolkit serves as a bridge between Docker's isolated environment and the direct hardware access needed for GPU acceleration, making it an essential component of GPU-based containerized solutions. One notable feature of the NVIDIA Container Toolkit is its adaptability. NVIDIA ensures backward compatibility, allowing older containers relying on previous GPU driver versions to function seamlessly even after host system GPU driver updates. This guarantees flexibility and reliability in deployment scenarios. Additionally, the

toolkit simplifies GPU usage in multi-GPU systems. By default, a Docker container has access to all available NVIDIA GPUs on the host. However, with the toolkit, it's possible to restrict a container to specific GPUs, ensuring efficient resource allocation and segregation in more complex scenarios. To install the NVIDIA Toolkit, we provide a script called `setup-ubuntu.sh`.

Throughout this project, various laptop GPU models were utilized. The first one, known as "Ampere A3000," is specifically designed with a CUDA compute capability of 86. The second GPU, "GeForce 4090 Ti RTX," boasts a superior compute capability of 89. Additionally, the server serving as the back-end is equipped with an A100 40 GB GPU, featuring a compute capability of 80. The compute capability is important since it determines the general architecture features as well as the specifics of the installed GPU. This metric becomes especially relevant when executing parallel computations or employing specific CUDA features. It also ensures that the appropriate binaries are utilized, optimizing performance and minimizing potential compatibility issues. For example, any mismatch in the compute capability specification in the `Dockerfile`, in line `ARG CUDA_ARCHITECTURES=89`, can cause the graphics card not to be recognized, resulting in errors such as "illegal memory access". Troubleshooting such errors can be difficult as they are too general to be directly related to computational performance. Unfortunately, there's no automated mechanism in place to detect and adjust the compatibility. This underlines the significance of accurately setting compute capability values during deployment and development stages. To ascertain the compute capability of a specific graphics card, you can refer to the dedicated Wikipedia page on CUDA. This page offers a comprehensive list of all graphics cards and their compute capability. It is essential to consult this list to ensure compatibility and maximize the efficiency of GPU-intensive tasks.

4.1.4. Installation of Pytorch with CUDA and Tiny-Cuda-NN

Some of the frameworks discussed in chapter 3 included CUDA, PyTorch and/or tiny-CUDA-NN in their projects. Installing these components presented a number of challenges, which are discussed in more detail in this section.

To use CUDA, it's essential to first install the compatible CUDA version for the used graphics card. The list of CUDA versions supported by various graphics cards can be referenced on the Wikipedia page about CUDA. In parallel, it is crucial to ensure that the graphics driver aligns with the intended CUDA version. This compatibility information is available on on NVIDIA's official driver download page. In our specific case, the graphics card were compatible with CUDA version 12, but it's worth noting that it's possible to downgrade to an earlier version if necessary.

Following the establishment of the CUDA environment, the installation of PyTorch is the subsequent step. PyTorch is a comprehensive framework for developing deep

learning models and is written in Python, making it accessible and user-friendly for machine learning developers. A direct PyTorch installation via `pip install torch` defaults to the latest version without CUDA support. It is therefore recommended to install a PyTorch binary that supports the CUDA version of the graphics card. Configuration guidelines and recommendations are provided on the official PyTorch website. For example, to install PyTorch 2.0.1 with CUDA 11.8, use the following command:

```
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/
  whl/cu118
## or
python3 -m pip install torch==2.0.1+cu118 torchvision==0.15.2+cu118 --extra-index-
  url https://download.pytorch.org/whl/cu118
```

Code Listing 12: PyTorch Installation with CUDA

We can check for successful installation with the following Python commands, which will return `true` if CUDA is available:

```
import torch
torch.cuda.is_available()
```

Code Listing 13: Verify Pytorch Installation

At this point I would like to refer to the Stack Overflow question page, where the user Jodac, who qualifies as a research scientist, has written a detailed and clear instruction.

The installation of the framework Tiny-CUDA-NN is also non-trivial. This framework is a concise, independent system proficient in training and querying neural networks. It features a rapid "fully fused" multi-layer perceptron, a flexible multiresolution hash encoding, and supports a broad spectrum of input encodings, losses, and optimizers. Tiny-CUDA-NN offers a PyTorch extension, enabling users to utilize its swift MLPs and input encodings within a Python environment. However, if PyTorch is already installed, Tiny-CUDA-NN can only be successfully installed if the PyTorch installation has the correct CUDA version. If the basic PyTorch installation was done via pip, the installation of Tiny-CUDA-NN will fail as it relies on the CUDA compute capability, which needs to be specified as an environment variable.

```
ENV TCNN_CUDA_ARCHITECTURES=${CUDA_ARCHITECTURES}
RUN python3.10 -m pip install git+https://github.com/NVlabs/tiny-cuda-nn.git@v1.6#
  subdirectory=bindings/torch
```

With accurate graphics card specifications and adherence to the correct installation processes, including the selection of the appropriate CUDA version, both PyTorch and tiny-CUDA-NN can be installed without complications.

4.2. Ways to run the projcet

In this section, we explore the multifaceted implementation of our project tailored for varied user expertise. For developers, the project can be initiated using the provided

`Dockerfile` or by fetching the pre-built Docker image from a Hub. Upon activation with the command `docker run --gpus all --rm -it nerf_industrial_metaverse -v path/to/video/folder:/nerf_industrial_metaverse/videos`, users can operate via the Docker Container's command-line interface. This approach was particularly useful during the pipeline implementation phase.

In addition to the command line, users can also launch the web application within the Docker container by running the `npm start` command in the `client` folder and `flask run` in the `server` folder. Once launched, the entire application is accessible via `localhost:3000`, creating a seamless experience. This execution method allows users to create a container on a server equipped with a high-performance GPU and still conduct training through the web application.

We've also structured our web application into two distinct components: the back-end and the front-end. This separation allows us to host the back-end on a server equipped with an NVIDIA GPU, while the front-end can smoothly operate on a standard server, readily accessible via a simple URL. This approach offers significant advantages. Users can seamlessly explore the world of neural radiance fields without the need for complicated installations or specialized graphics hardware. In addition, the web application's user-friendly design encourages users to experiment with the algorithm, providing a more accessible alternative to complex installations, command line interfaces or script execution. With robust server-client interactions and a detailed logging system in place, we guarantee transparency and active user involvement at every step. This project implementation ensures that users, regardless of their technical prowess, can access and benefit from our solution.

4.3. Key aspects of this project

In this section, we explore the multifaceted implementation aspects of our project tailored for varied user expertise.

4.3.1. The pipeline evolution

First, we set up the pipeline inside our Docker container and initiated it with the command `docker run --gpus all --rm -it nerf_industrial_metaverse -v path/to/video/folder:/nerf_industrial_metaverse/videos`. With the `-v` flag we have mounted a directory `(path/to/video/folder)` from the host machine into the container `(/nerf_industrial_metaverse/videos)`. This ensures that data can be accessed and modified both in the container and on the host system. Within this container environment, we maintain full control. We run this container on a server with an NVIDIA 4090 Ti GPU, which allows us to modify the YAML configurations and traverse the pipeline processes without impacting the local machine.

Following the effective implementation of our pipeline, we proceeded to implement a web application within the same Docker container. To activate this web application, we deploy the `npm start` command in the `client` directory and the `flask run` command within the `server` directory. Once operational, the entire application is readily available at `localhost:3000`.

After verifying the full functionality of the web application, we split the back-end and front-end into two separate GitLab projects. This separation allows the back-end to be hosted on a GPU-accelerated server, while the front-end is optimised to run on a conventional server and is easily accessible via a simple URL. By adopting a continuous integration and continuous delivery (CI/CD) framework, we've ensured that any changes made and pushed to the GitLab project trigger a rebuild and republication of the project. Our web application has multiple benefits. Users can seamlessly navigate the realm of radiation fields without having to navigate complicated installations or rely on specialized graphics systems. The application's user-friendly design encourages user exploration and serves as an accessible counterpart to more complicated installations and command-line utilities. With robust server-client interactions and a detailed logging system, we ensure a transparent process and consistent user involvement from start to finish. This careful project architecture emphasizes a user-centric approach, making our solution versatile and accessible to a diverse user base.

4.3.2. Project Organization

Developed in Python, the project is organized using modular scripts, each tailored for a specific function. This structure ensures optimal code organization, scalability, and ease of maintenance. As the project evolves, this modular setup facilitates seamless feature integration. Central to the implementation are several key directories, each reflecting a distinct functional aspect.

1. Common Directory: A foundational directory, *common* houses utility functions like the `config_util` for loading and writing YAML, `logger_utils`, `image_utils`, and more

2. Scripts_ngp Directory: In this directory we have adapted the instant-ngp scripts to run the training and modified them to meet the unique requirements of this project

3. Tools Directory: This directory contain python scripts responsible for individual tasks in the pipeline. These range from extracting frames from videos, running COLMAP commands, and processing data into a suitable format for training

4. Viewer directory: The *viewer* directory acts as a hub for interfacing with users, and is divided into two main subdirectories:

a) Client: This component is responsible for rendering the web pages the end-users interact with.

b) Server: The server component manages the back-end operations, processing user requests, and handling data appropriately.

Figure 22 shows the detailed folder structure and a complete script overview.

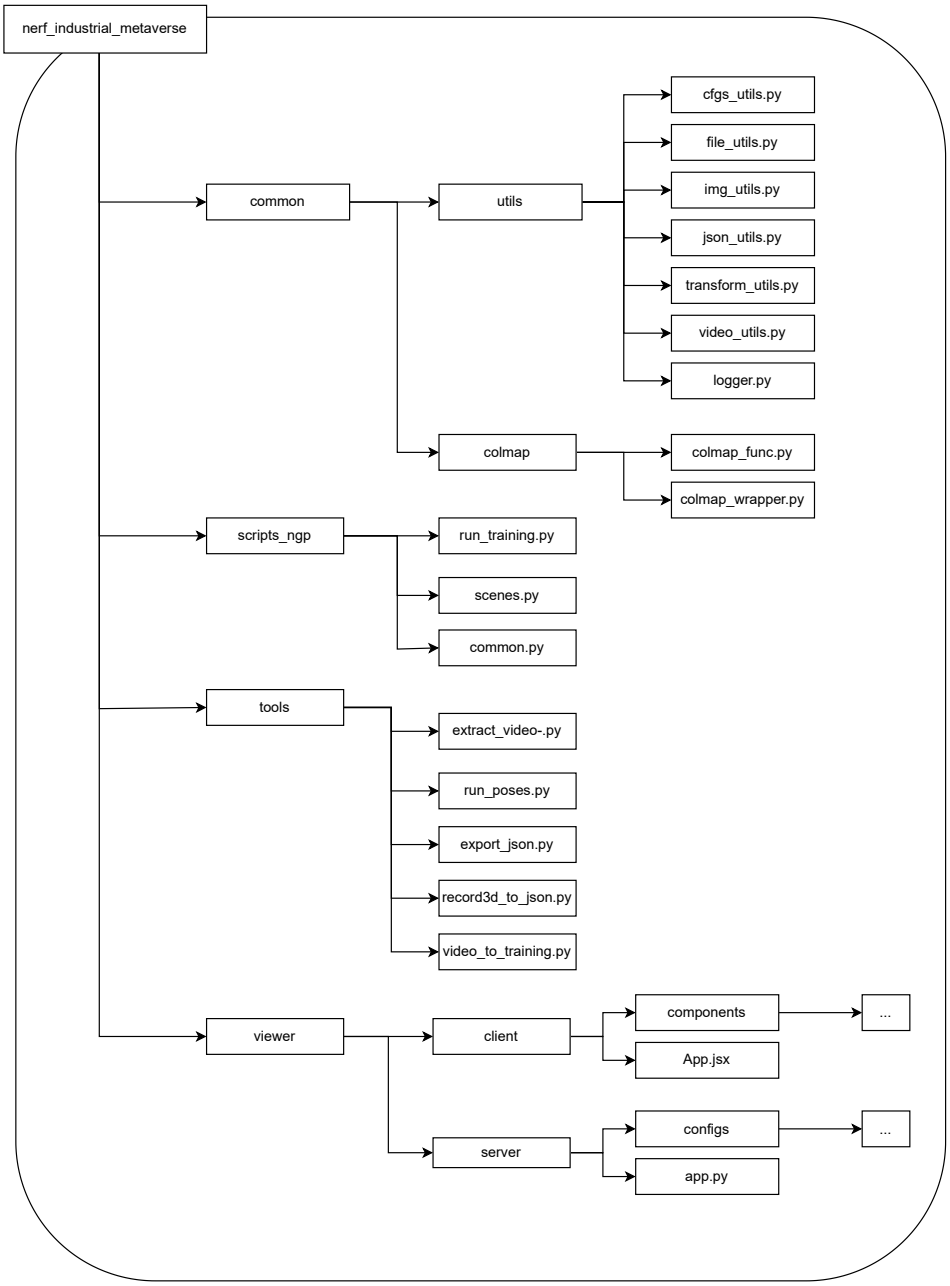


Figure 22.: The project consists of four main components: (1) *common* - containing auxiliary colmap scripts and utilities, (2) *scripts_ngp* - customized scripts from the instant-ngp framework, (3) *tools* - essential pipeline steps like video extraction and JSON file export, and (4) *viewer* - includes server and client scripts.

4.3.3. YAML as Configuration

To simplify parameter changes and configurations, we utilized a YAML configuration file. Users operating within the container can directly modify this YAML file. On the front-end, we've designed a user-friendly interface: users can adjust key configurations via input fields.. If no custom configurations are supplied, the system reverts to the settings from the `default.yaml` file. YAML is particularly apt for such configurations due to its human-readable nature. Its syntax is clear and more concise compared to other formats like JSON or XML, which facilitates users to organize and customize configurations intuitively (döt Net et al., 2021). Here's an illustrative snippet from our configuration file, highlighting its structured layout that allows users to easily adjust parameters such as the `match_type` of COLMAP:

```
data:
colmap:
match_type: sequential_matcher
video_downsample: '3'
video_path: ./uploads/VID_Buro.mp4
```

Code Listing 14: Example YAML Configuration used in this project

Beyond its readability, YAML provides flexibility with lists, key-value pairs, and nested structures for a layered data representation. Its streamlined syntax reduces excessive punctuation, ensuring clean and concise files. This enhances both readability and maintainability. Furthermore, editing YAML is straightforward; users only require a basic text editor, eliminating the need for specialized software. Its compatibility with version control systems like Git also facilitates team collaboration and a controlled development workflow (döt Net et al., 2021).

4.3.4. Real Time Logging

Going through the pipeline, especially during the COLMAP step, can be notably time-intensive. Each of the COLMAP steps demands considerable processing power and time, given the complexity of the operations involved. Therefore, providing continuous feedback to the user is important to keep them informed about ongoing background processes. To address this, we employed a logger, which continually logging steps to inform the user about the current stage and its results. The user is kept well-informed about the configurations made and the storage locations of the outputs. As illustrated in code snippet 15, the logger chronicles every significant action, providing a real-time update on the current stage, its results, and other pertinent details like configurations and output storage locations.

```
2023-08-02 12:44:30.097 | INFO | - Start to run COLMAP and estimate cam_poses...
  Scene dir: data/Waki_Showroom
2023-08-02 12:44:30.097 | INFO | - Need to run construct from colmap...
2023-08-02 12:44:30.098 | INFO | - Starting Feature Extraction...
```

```
2023-08-02 12:44:30.233 | INFO | - Features extracted...
2023-08-02 12:44:30.233 | INFO | - Starting Feature Matching...
2023-08-02 12:44:30.871 | INFO | - Features matched...
2023-08-02 12:44:30.871 | INFO | - Starting create sparse map...
```

Code Listing 15: Logger output detailing the steps and progress

The logger not only records activity on the command line, but also on the web page with an established socket connection. Each step's status and outcomes are clearly displayed to the user, offering a transparent and interactive experience.

4.4. Implementation of the pipeline steps

To train a neural radiance field with our self-recorded data, certain pre-processing steps are necessary to obtain the appropriate training input. In scenarios with video data, we start by extracting individual frames and estimating camera parameters. For this task, we use COLMAP, a Structure from Motion tool. After COLMAP has completed its processing, we convert the resulting data into a JSON format. This JSON file, together with the associated images, forms our training input. Another efficient method we have implemented is the R3D format from the Record3D app. The app uses the power of ARKit technology and is tailored for devices such as the iPhone 12 and its successors. This app generates recordings with key camera parameters, which are then converted directly into the required JSON format, bypassing the need for COLMAP and allowing for a smoother transition into the training phase. With the suitable trainingsformat and the configurations for our training, we can then train the neural radiance field.

4.4.1. Extract Images from Video

The script named `extract_video.py` is dedicated to frame extraction from video data, subsequently saving them as individual image files. The input to this function is important and will therefore outlines as follows:

1. video path: Source path to the input video
2. destination folder: Destination directory for the saved image
3. video downsampling factor: Determines the frequency of frame extraction, allowing users to skip over frames. A factor of 1 would extract every frame, 2 every second frame, and so forth.
4. maximum frames: Optional parameter to limit the number of extracted frames image extension: Image file format, either .png or .jpg

The `video downsampling` and the `maximum frames` variables are adjustable parameters within the `YAML` Configuration. Depending on the graphics card in use, memory limitations might be encountered when processing large training inputs. Adjusting these two parameters allows for optimization based on the capabilities of the GPU. For instance, with large videos, increasing the downsampling factor ensures that the entire video is processed without overwhelming the GPU memory. Conversely, for shorter videos, the downsampling factor can be decreased until the specified maximum number of frames is reached. This flexibility ensures efficient processing while maximizing the utility of available hardware resources.

Upon calling the function, it checks if the video file exists, opens the video using OpenCV's `VideoCapture` class, and iterates through the frames. Each frame is then saved with a filename in the format "xxxxx.jpg," with the x's representing a sequence of digits padded with zeros. The function continues extracting and saving frames until it reaches the maximum frame count (if specified) or processes all frames in the video. Finally, it releases the video capture object to free up resources.

4.4.2. Estimate camera poses from images with COLMAP

Using the images in the `data/video_name` folder, we calibrate the camera to derive poses for each image. Camera calibration extracts key parameters that define the camera's geometric properties. Intrinsic parameters, such as focal length, principal point coordinates, and lens distortion coefficients, detail the camera's internal configurations. Conversely, extrinsic parameters, like rotation and translation matrices for each image, indicate the camera's spatial orientation and position relative to a scene.Structure-from-Motion (SfM) is a technique used for camera calibration, where $3D$ structures are reconstructed from sequences of $2D$ images captured from various angles and positions. Through various algorithms, this method determines the $3D$ geometry of a scene as well as the corresponding camera positions.

In our research, we used COLMAP, an open-source tool that embodies the algorithms of the SfM method. By executing the command `ngp-colmap`, inspired by the code in (Mildenhall et al., 2019a; Müller et al., 2022), a set of COLMAP commands is activated. These commands represent different stages of the process, ranging from feature extraction to bundle matching. In the following sections we elaborate on these algorithms to clarify how SfM calibrates the camera.

The initial phase involves performing a correspondence search to detect scene overlaps within the input images, denoted as $I = \{I_i \mid i = 1..N_I\}$. This search aims to identify the projections of identical points that appear in multiple overlapping images. The outcome of this process consists of a collection of image pairs, $\overline{C}$, which have been geometrically verified, along with a graph that represents the image projections for each identified point.

Using the `colmap feature extractor` command, COLAMP processes each image I_i and obtains a set of local features $F_i = \{(x_j, f_j) \mid j = 1..N_{F_i}\}$ specific to that image. Here, N_{F_i} represents the total number of local features detected in image I_i. These features are carefully designed to maintain invariance under both radiometric and geometric variations, enabling SfM to accurately and uniquely recognize them across multiple images. Among the various feature extraction options available, Scale Invariant Feature Transform (SIFT) stands out as the gold standard in terms of robustness and will be utilized in this context (Schönberger and Frahm, 2016). This command can be fine-tuned with various parameters to enhance the output. For instance, when working with a singular camera, the Camera Model can be preset to *OPENCV*. The following code, referenced as 16, succinctly demonstrates this feature extraction procedure, including the tailored parameters:

```
feature_extractor_args = [
"colmap",
'feature_extractor',
'--database_path',
os.path.join(scene_dir, 'database.db'),
'--image_path',
os.path.join(scene_dir, 'images'),
'--ImageReader.camera_model=OPENCV',
'--SiftExtraction.use_gpu=true',
'--SiftExtraction.estimate_affine_shape=true',
'--SiftExtraction.domain_size_pooling=true',
'--ImageReader.single_camera=1',
'--SiftExtraction.gpu_index=0'
]
logger.add_log('Starting Feature Extraction...')
run_subprocess(feature_extractor_args, logger, logfile)
logger.add_log('    Features extracted...')
```

Code Listing 16: The command `colmap feature_extractor` with its input options

Once features have been extracted from a set of images using techniques like SIFT, the next step in SfM is feature matching, executed using the command `colmap feature_matching`. At its core, this process aims to identify shared features across different images to deduce which images depict overlapping parts of a scene. A straightforward approach to this involves testing every possible image pair for scene overlap. This is achieved by searching for feature correspondences, where each feature in image I_a is compared to every feature in image I_b using a similarity metric that compares their appearances f_j. However, this naive approach has a computational complexity of $O(N_I \cdot N_{F_i}^2)$, making it impractical for large image collections. To address this challenge, various approaches have been developed to make the matching process scalable and efficient. The ultimate output of this stage is a set of potentially overlapping image pairs $C = \{\{I_a, I_b\} \mid I_a, I_b \in I, a < b\}$, along with their associated feature correspondences $M_{ab} \in F_a \times F_b$. These correspondences are essential for establishing the relationships between different images and are crucial for subsequent

$3D$ reconstruction and scene understanding (Schönberger and Frahm, 2016).

Notably, in our setup, the default matching type is set to the `sequential matcher`. This mode proves useful when the images are captured sequentially, as is the case with video camera footage. As consecutive frames typically have visual overlap, there's no need to exhaustively match all image pairs. Instead, the consecutively captured images are matched against each other. By modifying the YAML file, one can switch the matcher type, for instance, to `exhaustive`, which might be more suited for standalone images.

```
feature_matcher_args = [
colmap_binary,
match_type,
'--database_path',
os.path.join(scene_dir, 'database.db'),
'--SiftMatching.use_gpu=true',
'--SiftMatching.guided_matching=true',
'--SiftMatching.gpu_index=0',
]
logger.add_log('Starting Feature Matching...')
run_subprocess(feature_matcher_args, logger, logfile)
logger.add_log('    Features matched...')
```

Code Listing 17: The command `colmap feature_matching` with its additional input

The matching stage also involves the verification of potentially overlapping image pairs $\overline{C}$. As matching is based solely on appearance, it cannot guarantee that the corresponding features actually correspond to the same scene points. To address this, SfM attempts to estimate transformations that map feature points between images using projective geometry. Depending on the movements of the camera and the nature of the scene, different mathematical models can be used to describe the relationship between features in different images. For example, a homography H is used to describe the transformation of a purely rotating or moving camera capturing a planar scene. Epipolar geometry, on the other hand, is used to describe the relation for a moving camera through the essential matrix E (calibrated) or the fundamental matrix F (uncalibrated). To verify the matches, SfM attempts to find a valid transformation that maps a sufficient number of features between the images. This process is essential to establish the geometric accuracy of the matched pairs. However, since the correspondences from matching may be contaminated with outliers, robust estimation techniques like RANSAC are employed to handle the outliers effectively. The output of this stage is a set of geometrically verified image pairs $\overline{C}$, their associated inlier correspondences $\overline{M}_{ab}$ and optionally a description of their geometric relation G_{ab}. COLMAP creates with the `feature matching` command a so-called scene graph with images as nodes and verified pairs of images as edges (Schönberger and Frahm, 2016).

This scene graph works as input to the reconstruction stage. The main objective

of this stage is to deduce pose estimates $P = \{P_c \in SE(3) \mid c = 1...N_P\}$ for previously registered images and to specify the reconstructed spatial scene structure as a set of points $X = \{X_k \in \mathbb{R}^{\not\#} \mid k = 1..N_X\}$. To achieve this, we utilize the `colmap mapper` command. The process begins with the creation of a basic model derived from a carefully selected two-view reconstruction. After obtaining a metric reconstruction, new images can be registered to the refine the existing model using the Perspective-n-Point (PnP) problem, which involves feature correspondences with already triangulated points in registered images ($2D$-$3D$ correspondences). In case of calibrated cameras, the PnP problem's pose estimation often employs RANSAC and a minimal pose solver, whereas uncalibrated cameras may use various minimal solvers or sampling-based approaches. For a newly registered image, COLAMP observes existing scene points and expands the scene's coverage by adding new points to the set of points X through triangulation. To triangulate and incorporate a new scene point X_k into X, COLMAP requires registration of at least one additional image that offers a different viewpoint of the new scene. Triangulation is a critical step in SfM as it enhances the stability of the existing model through redundancy and enables the registration of new images by providing additional $2D$-$3D$ correspondences. Image registration and triangulation are separate procedures, but their outcomes are highly correlated, as uncertainties in camera pose affect triangulated points and vice versa. Further, additional triangulations may enhance the initial camera pose through increased redundancy.

By default, COLMAP keeps the principal point constant during the reconstruction, as principal point estimation is an ill-posed problem in general. Once all images are reconstructed, the problem is most often constrained enough that you can try to refine the principal point in global bundle adjustment, especially when sharing intrinsic parameters between multiple images. After constructing a sparse model of the scene, we apply Bundle Adjustment to fine-tune the principal point by minimizing the "reprojection error" E. This error essentially measures the difference between the observed positions of points in an image and the projected positions of these points based on our current understanding of camera parameters and point locations. Mathematically, this error can be represented as:

$$E = \sum_j p_j \left(||\pi(P_C, X_k) - x_j||_2^2 \right)$$

The task of minimizing projection error is tackled by several algorithms within the field, two of which are noteworthy. The Levenberg-Marquardt algorithm, a commonly employed approach, is widely recognized for its effectiveness. Additionally, the Schur complement trick is another strategy motivated by the unique parameter structure encountered in Bundle Adjustment (BA) problems. This technique involves a two-

step process: firstly, solving the reduced camera system, followed by the subsequent update of points via back-substitution, as detailed in the work by Schoenberger in (Schönberger and Frahm, 2016).

To provide insight into the COLMAP procedure, we showcase the results using the COLMAP GUI. Figure 23 offers a snapshot of this presentation. where, the sparse scene reconstruction is represented as a set of points, and the reconstructed cameras are highlighted in red.

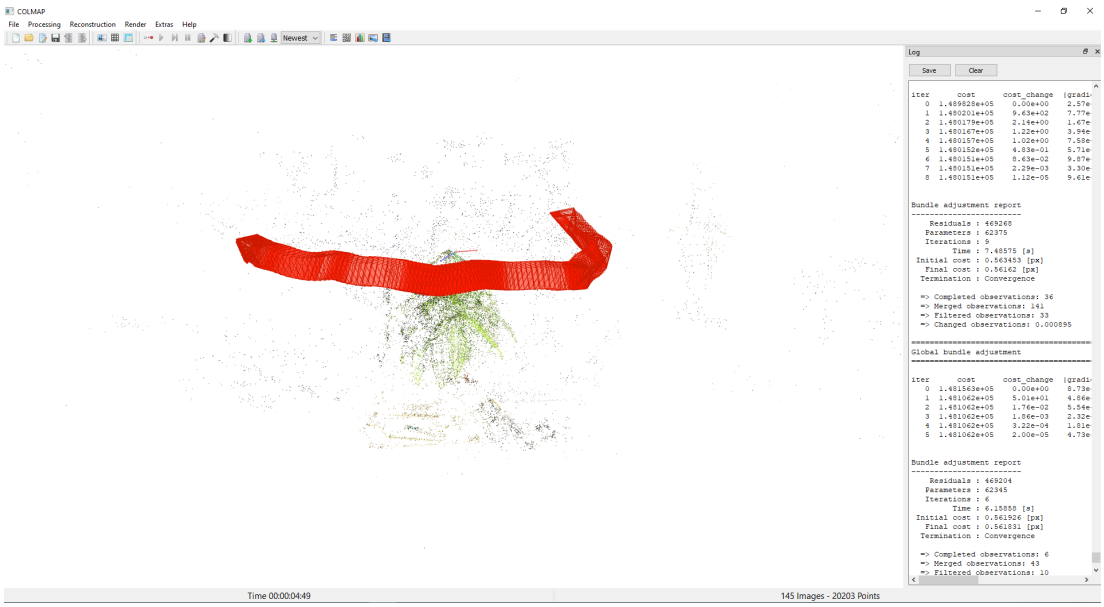


Figure 23.: COLMAP Result after the Scene Reconstruction. The red symbols represent the reconstructed cameras, and the scene is representedwith the estimated $3D$ points.

After the successful execution of COLMAP, it generates three essential text files: *cameras.txt*, *images.txt*, and *points3D.txt*.

cameras.txt: This file contains information about each camera that captured the images. Specifically, it outlines the intrinsic parameters of the cameras, which include details like the focal length, principal point, and distortion coefficients. These parameters provide insights into the internal configurations of the camera.

- *cameras.txt*: This file contains the intrinsic parameters of all reconstructed cameras in the dataset
- *images.txt*: This file contains the pose and keypoints of all reconstructed images
- *points3D.txt*: This file contains the information of all reconstructed 3D points

For the subsequent processing steps, we specifically use the *cameras.txt* and *images.txt* files.

4.4.3. Generating the JSON Input

After we have already secured a set of images designated for our training input, our next objective is to construct a JSON file, in which we encapsulate the intrinsic camera parameters and, for each existing image, also the corresponding extrinsic camera parameters. To generate this file, we execute our customizes `export_json.py` script. The resulting JSON file is structured as follows 18.

```
{
  "camera_angle_x": 0.7599288285742521, // The camera angles of view in radians
  "camera_angle_y": 1.2336458746664418, //
  "fl_x": 1072.0, //  The focal lengths in pixels for the x and y axes
  "fl_y": 1068.0,
  "cx": 553.0, // The principal points (x,y) in pixels, representing the image center
     coordinates
  "cy": 969.0,
  "w": 1080, //  The image width w and height h in pixels
  "h": 1920, //
  "k1": 0.0312, // Distortion parameters (k) used for radial distortion correction
  "k2": 0,
  "k3": 0,
  "k4": 0,
  "p1": 0, // Distortion parameters (p) used for tangential distortion correction
  "p2": 0,
  "aabb_scale": 4,
  "frames": [] // A section where per-frame extrinsic parameters are specified
}
```

Code Listing 18: Input file for training neural radiance fields with instant-ngp

The parameters `camera_angle_x` and `camera_angle_y` denote the horizontal and vertical fields of view (FoV) of the camera, respectively. In essence, these angles define the angular span that the camera captures, representing how much of the scene is visible to the lens. The fields `fl_x` and `fl_y` denote the focal length for the x and y axes. The focal length `fl_x` and `fl_y` specify the focal length for the x and y axes. The focal length gives information about the angle of view (how much of the scene is captured). Every camera sensor possesses a central point, often referred to as the 'principal point.' In our configuration, this is denoted by `cx` and `cy` parameters, representing the image center in pixel coordinates. The resolution of the image is captured by `w` and `h`, indicating its width and height in pixels.c A common issue in photography is radial distortion, where images get distorted, especially at the edges. The parameters `k1` and `k2`, `k2`, `k3`, `k4`, are coefficients of a polynomial that helps in rectifying this distortion. Similarly, tangential distortions can be described using `p1` and `p2`. The flag `is_fisheye` is indicative of the camera lens model. If set to true, it denotes that the camera employs a fisheye lens, characterized by an wide hemispherical view. Such a detailed calibration allows the transformation from $3D$ world points to $2D$ image points.

Extrinsic camera parameters provide information about the position and orientation

of a camera in a $3D$ space. In our case, these parameters are represented in the form of a transformation matrix, denoted as $M = [R|t]$. Here, R, represents a 3x3 rotation matrix, and t represents a 3x1 translation vector. To explain further, the rotation matrix R adjusts the camera's coordinate system to the world's standard coordinates. In parallel, the translation vector t moves the camera's origin to coincide with that of the world. Given the context of homogeneous coordinates, the transformation matrix M contains an additional component that expands it into a 4x4 matrix. The advantage of using a 4x4 matrix in homogeneous coordinates lies in its efficiency: it consolidates rotation and translation into a single matrix multiplication, thus streamlining the transformation process. Below is a representative example of how the transformation matrix is structured within the JSON file.

```
{
  // ...
  "frames": [
  {
    "file_path": "images/frame_00001.jpeg",
    "transform_matrix": [
    [1.0, 0.0, 0.0, 0.0],
    [0.0, 1.0, 0.0, 0.0],
    [0.0, 0.0, 1.0, 0.0],
    [0.0, 0.0, 0.0, 1.0]
    ]
  }
```

Code Listing 19: Example of extrinsic parameters stored in the JSON file

In this structure, the field `frames` defines an array, where each entry corresponds to a particular frame. Within each frame, the `file_path` provides the path to the image associated with the frame. Following the file path, we have the `transformation_matrix`, which captures the extrinsic parameters.

To address the absence of documentation on the transformation matrix in instant-ngp, we designed a simple `transform.json` file, which allows for specific and explicit changes to the parameters, providing a clear view of how the transformation matrix is defined in instant-ngp. In this JSON structure, we integrated frames accompanied by black images, simultaneously varying the parameters of the transformation matrix. Figure 24 depicts the coordinate system alongside the cameras we instituted via our JSON in instant-ngp. The distinct axes are color-coded for clarity: blue signifies the forward (z-axis) direction, red symbolizes the side (x-axis) perspective, and green indicates the upward (y-axis) orientation. We strategically positioned cameras at every vertex of this coordinate system, marking their respective translation matrices. For instance, the primary camera, centrally placed at the coordinate $(0, 0, 0, 1)$, remains

with no translation and has the transformation matrix:

$$\text{transform\_matrix} = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Adjustments to the first translation value result in camera translation along the z-axis. Modifying the second translation value causes the camera to traverse the x-axis, and setting the translation to $(0, 0, 1, 1)$ moves the camera vertically along the y-axis.

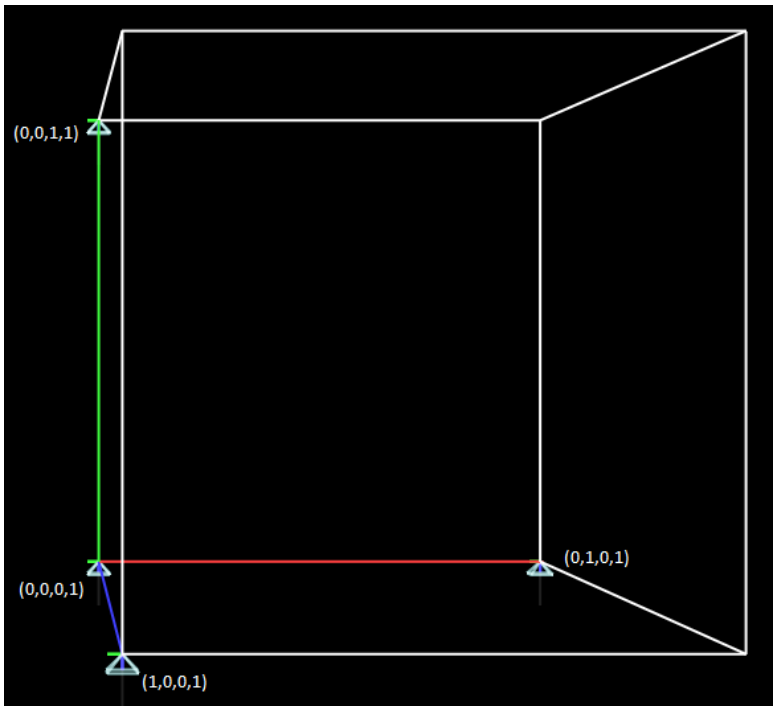


Figure 24.: The coordinate system as applied in instant-ngp. The blue, red, and green axes represent the z, x, and y dimensions, respectively. Cameras are strategically positioned at each system corner, with their corresponding translation matrices annotated. The central camera, situated at $(0, 0, 0, 1)$, remains stationary. Adjusting the matrix's first value moves the camera along the z-axis, the second value adjustment translates it across the x-axis, and setting it to $(0, 0, 1, 1)$ elevates the camera along the y-axis.

The transformation matrix can therefore be represented as:

$$\text{transformation\_matrix} = \begin{bmatrix} 0 & 0 & 0 & Z \\ 0 & 0 & 0 & X \\ 0 & +0 & 0 & Y \\ 0.0 & 0.0 & 0.0 & 1 \end{bmatrix}$$

We can gain further insight into this transformation matrix by examining the instant-ngp

script, which implemented how the COLMAP input is converted into the matrix. Additionally, we intend to investigate the rotations as part of our comprehensive analysis.

The *images.txt* file offers details on the pose and keypoints of every reconstructed image within the dataset, with each image being described over two lines.

```
# Image list with two lines of data per image:
# IMAGE_ID, QW, QX, QY, QZ, TX, TY, TZ, CAMERA_ID, NAME
# POINTS2D[] as (X, Y, POINT3D_ID)
# Number of images: 2, mean observations per image: 2
1 0.851773 0.0165051 0.503764 -0.142941 -0.737434 1.02973 3.74354 1 P1180141.JPG
2362.39 248.498 58396 1784.7 268.254 59027 1784.7 268.254 -1
2 0.851773 0.0165051 0.503764 -0.142941 -0.737434 1.02973 3.74354 1 P1180142.JPG
1190.83 663.957 23056 1258.77 640.354 59070
```

The pose of the reconstructed image is represented through a quaternion (Q_W, Q_X, Q_Y, Q_Z) and a translation vector (T_X, T_Y, T_Z). This quaternion adheres to the Hamilton convention, a standard used by the Eigen library. The camera's local coordinate system is oriented such that the X-axis points rightward, the Y-axis downward, and the Z-axis forward, all from the perspective of the image (Schönberger and Frahm, 2016).

The adaptive code, sourced from *instant-ngp*, initiates its operation by loading the `images.txt` file and subsequently transforms this input into a matrix. Below is a snippet illustrating this transformation:

```
qvec = np.array(tuple(map(float, elems[1:5])))
tvec = np.array(tuple(map(float, elems[5:8])))
R = qvec2rotmat(-qvec)
t = tvec.reshape([3, 1])
m = np.concatenate([np.concatenate([R, t], 1), bottom], 0)
c2w = np.linalg.inv(m)
c2w[0:3, [1, 2]] *= -1  # flip the y and z axis
c2w = c2w[[1, 0, 2, 3], :]
c2w[2, :] *= -1  # flip whole world upside down
```

Code Listing 20: Converting COLMAP's extrinsics to the transformation matrix

To understand the given Python code more deeply, let's break down its operation mathematically:

- A quaternion $q = [q_w, q_x, q_y, q_z]$ is extracted from the input and is converted to a 3x3 rotation matrix R using the given `qvec2rotmat` function. The matrix R is represented as:

$$R = \begin{bmatrix} 1 - 2q_y^2 - 2q_z^2 & 2q_xq_y - 2q_wq_z & 2q_xq_z + 2q_wq_y \\ 2q_xq_y + 2q_wq_z & 1 - 2q_x^2 - 2q_z^2 & 2q_yq_z - 2q_wq_x \\ 2q_xq_z - 2q_wq_y & 2q_yq_z + 2q_wq_x & 1 - 2q_x^2 - 2q_y^2 \end{bmatrix}$$

- This rotation matrix R is then concatenated with the translation vector t to form a 4x4 transformation matrix M:

$$M = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$$

- Next, in line 5 in 20, the matrix is inverted to convert the camera-to-world transformation into a world-to-camera transformation.
- Certain coordinate system transformations are applied:
  - The y and z axes are flipped (line 7).
  - The rows of the x and y axes are swapped (line 8).
  - The entire orientation of the world is reversed (line 9).
- After these adjustments, the final camera-to-world matrix $C2W$ is represented as:

$$C2W_{transformed} = \begin{bmatrix} +X_0 & +Y_0 & +Z_0 & Z \\ +X_1 & +Y_1 & +Z_1 & X \\ +X_2 & +Y_2 & +Z_2 & Y \\ 0.0 & 0.0 & 0.0 & 1 \end{bmatrix}$$

The operations and their corresponding mathematical representations elucidate the transformation process of COLMAP's extrinsics into a matrix format, which defines the camera's position in $3D$ space.

In our in-depth analysis, we determined the coordinate system of the transformation matrix. Given its non-conventional nature, we found an issue discussed on the GitHub page of instant-ngp concerning this matrix. Alex Evans, one of the paper's authors, clarified the situation:

> We apologize for the shift in the coordinate system. Internally, NGP employs an entirely 0-1 bounding box, as exhibited in the GUI, with cameras oriented towards the positive z-direction. This convention was an early choice. Nonetheless, we aimed for compatibility with the original NeRF datasets, which define the origin at 0, scale the cameras to a distance approximately three units from the origin, and follow a distinct convention for orientation. Therefore, the snippet you encountered is essentially a translation from the original NeRF paper conventions to NGP's conventions. Over time, the original NeRF format of transforms.json became the primary method for data integration into NGP, which might seem confusing due to the lack of explicit discussion on the mapping (Alex Evans).

In summary, the design remains as intended. If required, users can modify the scale and offset components of the transformation by including additional parameters such as 'offset', 'scale' and 'aabb' in the JSON. For now, however, the coordinate inversion is hard-coded and cannot be changed (Alex Evans).

4.4.4. Training Neural Radiance Fields with instant-ngp

After generating the `transform.json` file, we initiate the neural radiance field training. The pipeline automatically triggers the modified script from instant-ngp using the `ngp-train` command.

The training configurations are initially loaded from the configuration YAML file, derived from the instant-ngp `run.y` parameters. These adaptable configurations are presented in code snippet 21:

```
train: # Run instant neural graphics primitives with additional configuration & output options
  scene: True            # The scene to load. Can be the scene's name or a full path to the training data. Can be NeRF dataset, a *.obj/*.stl mesh for training a SDF, an image, or a *.nvdb volume.
  n_steps: 150000               # Number of steps to train for before quitting
  network: ""                   # Path to the network config. configs/base.json. Uses the scene's default if unspecified
  #load_snapshot: ""            # Load this snapshot before training. recommended extension: .ingp/.msgpack
  # nerf_compatibility: True    # Matches parameters with original NeRF. Can cause slowness and worse results on some scenes, but helps with high PSNR on synthetic scenes
  #test_transforms: ""          # Path to a nerf style transforms json from which we will compute PSNR.
  near_distance: -1             # Set the distance from the camera at which training rays start for nerf. <0 means use ngp default
  exposure: 0.0                 # Controls the brightness of the image. Positive numbers increase brightness, negative numbers decrease it
  screenshot_transforms: ./screenshots/screenshot.json     # Path to a nerf style transforms.json from which to save screenshots.
  screenshot_frames: 0     # Which frame(s) to take screenshots of
  screenshot_dir: ./data/VID_20230615_101612/screenshots           # Which directory to output screenshots to
  screenshot_spp: 16           # type=int: Number of samples per pixel in screenshots
  width: 1920
  height: 1080
  video_camera_smoothing: True # Applies additional smoothing to the camera trajectory with the caveat that the endpoint of the camera path may not be reached
  video_fps: 30                # Number of frames per second
  video_n_seconds: 5           # Number of seconds the rendered video should be long.
  video_render_range: -1, -1  # Limit output to frames between START_FRAME and END_FRAME (inclusive)
  video_spp: 8                 # Number of samples per pixel. A larger number means less noise, but slower rendering
  #video_output: VID_20230615_101612.mp4

  #save_mesh: ""                # Output a marching-cubes based mesh from the NeRF
```

```
    or SDF model. Supports OBJ and PLY format.
marching_cubes_res: 256       # Sets the resolution for the marching cubes grid.
sharpen: 0                    # Set amount of sharpening applied to NeRF training
    images. Range 0.0 to 1.0.
```

Code Listing 21: Configurations for training

The default configuration ensures that a neural radiance field is trained for `n_steps` iterations before saving under the `save_snapshot` tag. With the `screenshot_transform` tag, screenshot images are rendered and saved, leveraging the `screenshot.json` file. This file was created concurrently with the primary JSON file and contains the intrinsic parameters and two slightly modified transformation matrices from the first and last frames of the JSON file. The small changes in the transformation matrix highlight the creation of novel views and serve as evidence of successful training.

After the training concludes, the pipeline automatically displays the screenshots on the front-end, offering a visual validation of the successful training. On this interface, users can either opt to render videos following a pre-defined camera path or download the trained `.ingp` file. Additionally, users can export a mesh, enabling them to utilize the generated $3D$ geometry. This possibilities require the addition of particular keys, followed by the re-initiation of the `ngp-train` with the updated YAML parameters.

```
keys_to_remove_list = [
['train.save_snapshot'],
['train.save_mesh'],
['train.screenshot_transforms', 'train.screenshot_dir']
]
new_config = {
  'train.video_output': configurations['video_output'],
  'train.load_snapshot': configurations['snapshot'],
  'train.video_camera_path': "configs/base_cam.json"
}
```

Code Listing 22: keys for rendering a video from a trained snaphot

Essentially, this process entails the elimination of specific tags and the integration of new ones, which ensures the `run-ngp.py` script aligns with the updated configurations. Figure 25 shows the web page after a successful training.

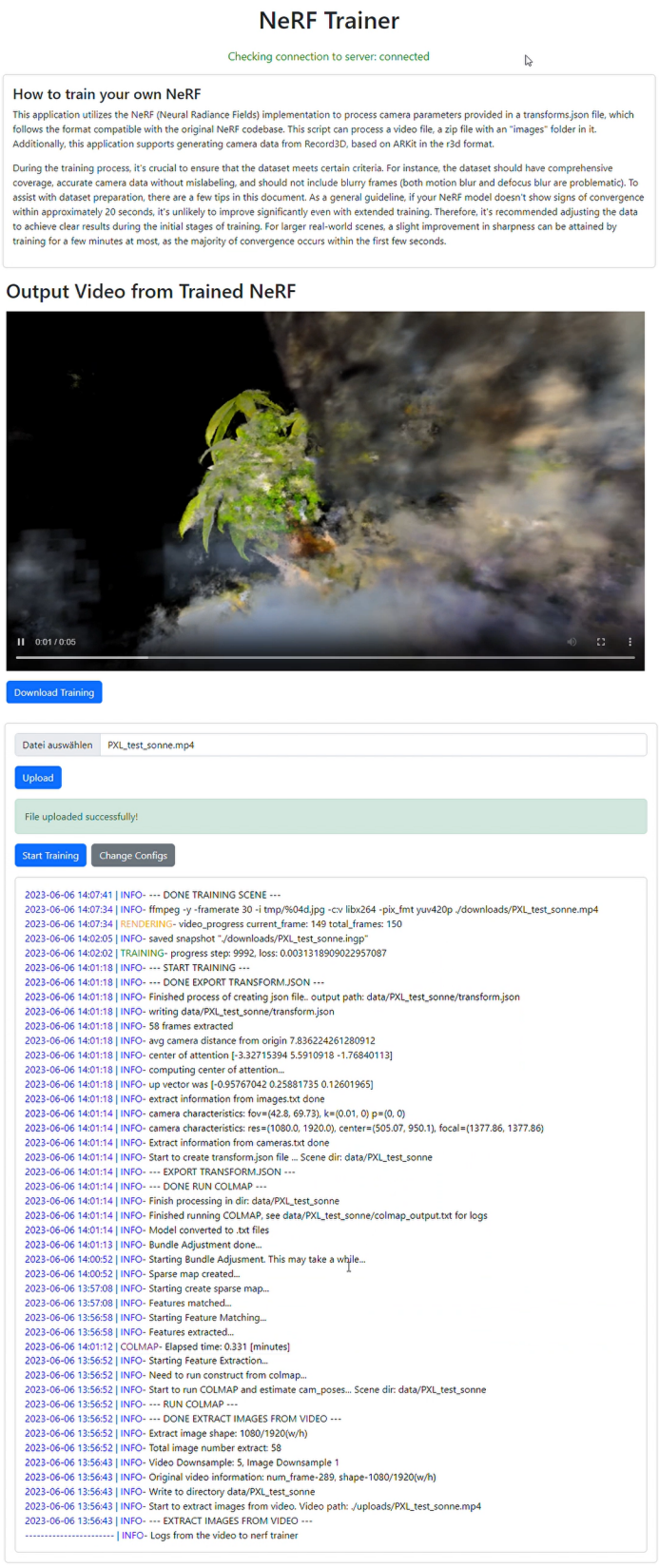


Figure 25.: This figure presents the user interface on the webpage, encompassing backend logs. Through a socket connection, users receive ongoing updates, ensuring they remain updated about the training's progression and outcome. Such transparent feedback and live interactions elevate the user experience, making the entire training journey more intuitive.

4.5. The webpage for the pipeline

After implementing our pipeline, we wanted to move away from the Docker Development and towards developing a web application. The main motivation behind this move was to provide developers and non-developers with a more intuitive interface for training neural radiance fields, eliminating the need for installations, initiating containers, or navigating command-line tools. Offering a web-based solution simplifies these processes, ensuring a smoother experience for all users. Given that our original pipeline was developed in Python, it was a logical choice to continue using Python for the back-end. As a result, we opted for the Flask framework for back-end operations and paired it with React to create an interactive front-end experience.

4.5.1. Flask Application

Flask is a lightweight yet powerful web framework for Python that is particularly suitable for developing small to medium sized web applications. It provides the flexibility needed to create a variety of backend services, including REST APIs, web pages and more. In this section we will look at the implementation in Flask in more detail.

In our Flask application, the core structure and functionalities are encapsulated within a central file named `app.py`. This file not only hosts the essential configurations but also forms the backbone of our Flask setup. The configurations employed in our application are depicted in Figure 23.

```
from flask import Flask, request, jsonify, session, g, send_file
from werkzeug.utils import secure_filename
from flask_cors import CORS
from flask_socketio import SocketIO

app = Flask(name)

CORS(app, resources={
  r"/*": {"origins": ""}})

socketio = SocketIO(app, cors_allowed_origins="*", async_mode='threading')

SECRET_KEY = secrets.token_hex(16)
app.secret_key = SECRET_KEY

if name == 'main':
socketio.run(app, host='0.0.0.0', port=5000, debug=True)
```

Code Listing 23: Configuration of the Flask application in `app.py`

We begin by importing necessary modules and functionalities. The Flask class is indispensable for setting up any Flask application. Additional imports like `request`, `jsonify`, `session`, `g`, and `send_file` offer functionalities for handling client requests, JSON formatting, managing user sessions, a special object for temporary storage during a request, and sending files as responses, respectively. The `secure_filename`

function from the `werkzeug` library aids in ensuring that the uploaded files have secure names, a security measure preventing malicious uploads. The `CORS` function from `flask_cors` package is pivotal in handling Cross-Origin Resource Sharing (CORS), allowing the front-end and back-end to communicate without any restrictions, which is especially significant if they are hosted on different domains or ports. Furthermore, to enable to real-time communications, we employ `SocketIO` from the `flask_socketio` library. This ensures bidirectional communication between the web clients and servers, a requirement for many modern web applications. The line `app = Flask(__name__)` initiates our Flask app. For security, we have generated a secret key for our application, which ensures the confidentiality of user sessions and prevents potential tampering. This is established using the `secrets.token_hex(16)` function, which produces a random token that we set as the `app.secret_key`. Lastly, the `if __name__ == '__main__'` block signifies that if this script is run directly, the Flask app will start in debug mode on the specified host and port. The use of `socketio.run` instead of the conventional `app.run` allows for the integration of Flask with `Socket.IO's` functionalities.

Flask's route system stands out due to its intuitive nature. Using the `@app.route()` decorator, functions are easily mapped to specific URLs. The following code snippet is an example of this route system in Flask:

```
@app.route('/upload', methods=['POST'])
def upload():
  if 'file' not in request.files:
    return jsonify({'status': 'error', 'message': 'No file uploaded'}), 400
  file = request.files['file']
  if file and allowed_file(file.filename):
    os.makedirs(UPLOAD_FOLDER, exist_ok=True)
    filename = secure_filename(file.filename)
    file.save(osp.join(UPLOAD_FOLDER, filename))
    session['filename'] = get_filename_without_extension(
    filename)
    session['file_type'] = get_file_type(filename)
    set_configs_path(replace_config(DATA_FOLDER, session['filename']))
    update_yaml_with_session()
    return jsonify({'status': 'success', 'message': 'File uploaded successfully!'})
  return jsonify({'status': 'error', 'message': 'Invalid file type'}), 400
```

Code Listing 24: Route System in Flask

The route `@app.route('/upload', methods=['POST'])` defines the endpoint `/upload` and specifies that it accepts only `POST` requests. This route essentially acts as a gateway for users to upload files. Inside the upload function, there's a preliminary check to ascertain if a file is included in the incoming request using the condition `if 'file' not in request.files`: If no file is detected, an error response is generated. The line `filename = secure_filename(file.filename)` employs Flask's `secure_filename` function. This step is crucial as it ensures that the uploaded filename is safe to use with the filesystem. We save this file to a predefined `UPLOAD_FOLDER` using the `file.save()` function. We used session variable, which is Flask's way of maintaining stateful information between re-

quests. Here, the `filename` and its `filetype` are stored in the session for subsequent use. Towards the end, configurations associated with the uploaded file undergo updates. This is achieved through functions such as `set_configs_path`, which ensures the correct path is set to `data/scene_name/default.yaml`. Additionally, the `update_yaml_with_session` function is utilized to modify various YAML configurations, including the scene's name and specific locations for saving and loading. These operations underline the application's capability to adjust to user-specific configurations, thereby enhancing the system's dynamism and adaptability.

In this implementation, we established specific routes to interact with the front-end. For instance, the route `@app.route('/start-training', methods=['POST'])` initiates the pipeline when the "Start Training" button is clicked. Similarly, the route `@app.route('/update_yaml_config', methods=['POST'])` updates the YAML configuration based on changes made in the front-end. We also defined routes for transmitting screenshots as well as the rendered video, as exemplified in the schemas `@app.route('/api/images/path:image_path')` and `@app.route('/api/images')`.

Concluding this implementation, we've seamlessly integrated essential functionalities of the pipeline into a back-end web application, enabling a streamlined workflow for training neural radiance fields with uploaded data. Moreover, the platform provides visual feedback, equipping users with optional utilities such as downloading data, rendering videos, and exporting meshes.

To host the project on a back-end server, we implemented a continuous integration (CI) and a continuous delivery (CD) pipeline. This ensured that any change or update to our repository would trigger a new build of the back-end docker file. Such a mechanism guaranteed that the latest version of our application was always built and ready for deployment. We used GitLab's CI/CD offerings for this project. Our construction and deployment processes were delineated in a `.gitlab-ci.yml` file. The following code snippet, denoted by 25, illustrates part of our CI setup:

```
variables:
  ... [shortened for clarity]
stages:
  publish

publish:
  tags:
    [...]
  stage: publish
  image: ${IMAGE_PUBLISH}
  script: |
    echo ${ARTIFACTORY_TOKEN} | docker login -u=${ARTIFACTORY_USER} --password-stdin
    ${REGISTRY}
      docker build --file ${DOCKERFILE_PATH} -t ${REGISTRY}/${CI_PROJECT_NAME}:latest
     .
      docker push ${REGISTRY}/${CI_PROJECT_NAME}:latest
```

Code Listing 25: Deployment configuration of our back-end

The "script" section of this configuration outlines the sequential process: first, it logs into our private Docker registry using credentials. Following that, it builds the Docker image using our specified `Dockerfile`. Lastly, the newly built image is pushed to the registry. This structured and automated flow ensures that our back-end image is consistently up-to-date. In our configuration, our `Dockerfile` ends with the `ENTRYPOINT [ "python3", "app.py" ]`, which ensures that every time the container is initiated, the Flask application `app.py` script runs using the Python3 interpreter.

After ensuring the Continuous Integration (CI) of our project via GitLab CI, the subsequent step was to set up a Continuous Delivery (CD) pipeline. The primary goal was to ensure that our back-end server always operated the most recent version of our application. To accomplish this, we used Kubernetes, a powerful orchestration tool that automates the deployment, scaling, and management of containerized applications. Kubernetes groups containers that make up an application into logical units, called 'pods', for easy management and discovery. In our case, after building the Docker image of our application, this image is pulled onto a Kubernetes cluster. Once on Kubernetes, the platform ensures that the application is running as defined, automatically replacing containers that fail and killing those that don't respond to health checks. This not only guarantees high availability but also effective load balancing, ensuring optimal distribution of network traffic to maintain a seamless user experience.

4.5.2. React Application

React, developed and maintained by Facebook, is a premier open-source JavaScript library primarily utilized for constructing user interface (UI) components for web applications. Central to React's architecture are "components", self-contained blocks of code that detail the rendering and behavior of UI elements. React encourages developers to break down web interfaces into smaller, reusable components, fostering a more modular and easily maintainable codebase. Such a philosophy simplifies development, particularly in expansive applications. The structure of a standard React application generally consists of a series of nested components, with parent components holding and managing data for their child components (Source, 2023). Stylesheets, utilities, and configuration files complement these components. In the context of our project, the React application's structure is outlined as:

```
src
└── components
    ├── UploadForm.jsx
    ├── HandleTraining.jsx
    ├── ConfigInputField.jsx
    ├── ConfigsChanger.jsx
    ├── LogViewer.jsx
    ├── RenderVideo.jsx
    ├── VideoSlider.jsx
    └── ImageSlider.jsx
```

```
    ├── CheckConnection.jsx
    ├── ExportMesh.jsx
    ├── videos
    │   └── RenderedVideosFromTrainedSnaphots
├── App.jsx
├── App.css
├── index.js
└── [...]
```

To gain a deeper understanding of how this structure supports our React application, let's delve into the purpose and significance of each file and directory:

- src: The primary directory where all our React code resides.
  - components: Here are individual JSX files, each representing a distinct UI component. The `UploadForm.jsx` handles the functionality of uploading data. `HandleTraining.jsx` is responsible for the training processes. `ConfigInputField.jsx` and `ConfigsChanger.jsx` allow users to input and modify configurations, respectively. `LogViewer.jsx` provides the logger implementation `VideoSlider.jsx`, and `ImageSlider.jsx` are dedicated to video and image rendering and display functionalities. `CheckConnection.jsx` ensures the backend connection is active and responsive. `ExportMesh.jsx` and `RenderVideo.jsx` handles the mesh exporting and video render functionality. The videos sub-folder stores rendered videos resulting from trained radiance fields, offering a repository for output visualization with a slider implemente in `VideoSlider.jsx`.
  - App.jsx: This file serves as the main component. It imports and integrates various other components from the components directory, establishing the overall layout and flow of the application.
  - App.css: This stylesheet is linked to App.jsx. While React components handle functionality, accompanying stylesheets like App.css manage the style, ensuring a coherent visual experience.
  - ... : Placeholder for various other essential files. These include routing configurations, global state management setups, utility functions, and other necessary integrations.

In React, components can be integrated with `html`, as shown in the provided code snippet. The rendered view of the component `UploadForm`, as shown in figure 26, includes also the `LogViewer` because it was integrated into the `UploadForm`

```
<div className="card my-5">
  <div className="card-body">
    <UploadForm />
  </div>
```

```
</div>
```

Code Listing 26: Integrating React components

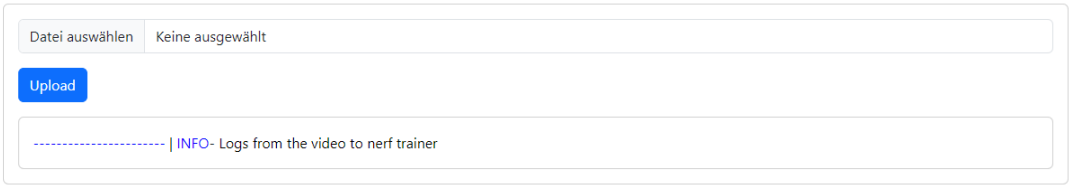


Figure 26.: Rendered view of the `UploadForm.jsx` component

By breaking down the `UploadForm` component, we aim to offer a clear view of our React-based implementation. Serving primarily as an interface for users to upload files, this component adeptly manages upload, status, and subsequent actions following the upload.

Component Initialization and Dependencies The component starts by importing essential React hooks and associated components. The `useState` hook, an integral part of React's functionality, facilitates state management within functional components.

```
import React, { useState } from 'react';
import LogViewer from './LogViewer';
import HandleTraining from './HandleTraining';
```

Code Listing 27: Initializing the UploadForm component

State Management The component's local state comprises variables that maintain the selected file, record upload status, and determine the visibility of specific UI elements.

```
const UploadForm = () => {
  const [selectedFile, setSelectedFile] = useState(null);
  const [uploadStatus, setUploadStatus] = useState('');
  const [showButtons, setShowButtons] = useState(false);
```

User Interactions and Networking Within the component, the `handleFileChange` function captures user-selected files. In contrast, the `handleUpload` function orchestrates the file upload. This function checks for a selected file, constructs the necessary `FormData` object, and initiates a `POST` request to the /upload endpoint.

```
const handleFileChange = (event) => {...};
const handleUpload = () => {
  if (!selectedFile) {
    setUploadStatus('No file selected.');
    return;
```

```
  }
  const formData = new FormData();
  fetch('URL-to-backend/upload', {
    method: 'POST',
    body: formData,
    },
  })
  .then((response) => {
    if (response.ok) {
      setUploadStatus('File uploaded successfully!');
      setShowButtons(true);
    } else {
      setUploadStatus('Upload failed.');
    }
  })
  .catch((error) => {
    setUploadStatus('Upload failed.');
    console.error(error);
  });
```

Code Listing 28: handle Upload in React

Rendering and UI Feedback The rendered output of the component is a collection of input fields, buttons and other nested components. In particular, the `HandleTraining` component becomes visible after a successful upload. At the same time, the `LogViewer` component is rendered, which processes the LOG entries.

```
  return (
  <div>
  <input type="file" className="form-control mb-3" onChange={handleFileChange} />
  <button className="btn btn-primary" onClick={handleUpload}>
  Upload
  </button>
  {uploadStatus && (
    <div className={`mt-3 alert ${uploadStatus === 'File uploaded successfully!' ? '
    alert-success' : 'alert-danger'}`} role="alert">
    {uploadStatus}
    </div>
    )}
  {showButtons && (
    <HandleTraining />
    )}
  <LogViewer />
  </div>
  );
};

export default UploadForm;
```

Code Listing 29: Rendering Element in React

Every component is architected to encapsulate its specific responsibility while seamlessly integrating with the larger system.

In this application, we maintaini real-time data flow between the server and client. We achieved this by using WebSockets. The provided code snippet showcases how

a React application establishes a WebSocket connection using the `socket.io-client` library.

```
1  import { io } from 'socket.io-client';
2
3  export const socket = io('URL-to-backend:5000', {
4    cors: {
5      origin: "URL-to-frontend:PORT",
6      methods: ["GET", "POST"],
7      credentials: true,
8      transports: ['websocket', 'polling'],
9    },
10   allowEIO3: true,
11   reconnection: true,
12   reconnectionAttempts: 10,
13   reconnectionDelay: 10000,
14   reconnectionDelayMax: 50000
15 });
```

Code Listing 30: Socket Connection in React

The connection is established using the `io` function. This function requires the server's URI and an optional configuration object. The configuration object defines various parameters to ensure the connection is secure and robust:

- cors: Defines the Cross-Origin Resource Sharing (CORS) configurations.
- origin: Specifies the client's origin, allowing requests from them
- methods: Lists the HTTP methods permitted.
- credentials: Indicates whether the browser should send cookies with requests.
- transports: Determines the transport protocols. Here, both 'websocket' and 'polling' are used, implying the connection will first try `WebSockets` and fallback to polling if necessary.
- allowEIO3: This ensures compatibility with both Engine.IO v3 and v4 protocols
- reconnection - reconnectionDelayMax (line 11-14 in 30): One notable aspect of this configuration is the emphasis on reconnection. Given that certain intensive operations, such as those performed by COLMAP, can monopolize server resources, there's potential for intermittent socket connection interruptions. To address this, the configuration is tailored to ensure that the client attempts to reconnect if the connection is lost, thereby reducing the risk of losing data communications due to temporary server unavailability

The script uses the `socket.io-client` library to manage this connection, ensuring real-time data communication while incorporating several robustness measures to handle connection interruptions.

We developed the front-end application in isolation from the back-end within a GitLab project. Once our development was complete, we also used GitLab's CI/CD pipeline, where each pushed change triggers an automatic rebuild and deployment process. This modular approach allows the front-end to be hosted on a server regardless of its GPU capabilities. In situations where the connection between the server and the client may be interrupted or non-existent, the design ensures that users will still be able to access the platform, familiarize themselves with the content and view the results of the trained neural radiance field videos. This ensures an uninterrupted user experience and consistent access to information.

4.6. Conclusion

In conclusion, the architecture detailed in this chapter signifies a comprehensive, end-to-end implementation that encapsulates the entire data flow – from initial data input to the training of neural radiance fields. This implementation, comprising system-independent installations, streamlined package management, efficient data upload processes, training mechanisms, and real-time monitoring, is brought together seamlessly through an intuitive web interface. Notably, this pipeline represents the first of its kind, pioneering an approach that integrates the initial data input through to the training of Neural Radiance Fields. By amalgamating diverse tools such as Poetry, Docker, CUDA, and front-end technologies like Flask and React, we have crafted a robust system that not only ensures efficiency but also greatly enhances user accessibility and experience.

5. Neural Radiance Fields in the Context of the Metaverse

The introduction of the NeRF algorithm marked a significant technological innovation. In general, neural scene rendering is a rapidly growing field with the potential to revolutionize the way we generate $3D$ content. Despite the comprehensive research highlighted in section 2.2.4, defining practical applications for this innovation has proven challenging. This chapter addresses these challenges by focusing on the potential applications of NeRF within the Industrial Metaverse. As we analyze the current state of the art, it is important to keep in mind throughout this chapter that it is not about how perfect the result is, but about the potential of those results.

In this chapter, we first address the idea of capturing reality in an industrial setting. We then will examine complicated scenarios where traditional methods such as photogrammetry fail, but where NeRF provides excellent alternative. We will then simulate two scenarios in the context of the Industrial Metaverse to determine the efficiency of NeRF in these scenarios. We round out this chapter by highlighting the specific limitations of NeRF that make it difficult to apply in practice and provide an overview of ongoing research efforts aimed at addressing these very challenges.

5.1. Capturing Reality at a single moment

The NeRF algorithm represents a static scene as a volumetric representation that captures intricate details, depth information, and light interactions. With the idea of an accurate representation of the real environment in a virtual environment, we can explore the possible use cases, especially in capturing the reality at specific times. Equally important to mention, but outside the scope of our current context, is the ability to continuously capture and update dynamic real world scenarios. This ensures near real-time synchronization with dynamic changes in the physical environment. Figure 27 visually represents the use cases of capturing scenes at a given point in time. One of the universal benefits, which affect all use cases, is the ability to continue operation without interruption, whether for production lines, machines or other processes. Moreover, users can immerse themselves in these virtual environments without having to be physically present. The illustration underscores following multifaced use cases of precise reality capture, although we would like to note that these options relate specifically to capturing reality with real-world conditions, whereas the options we mentioned in the Industrial Metaverse chapter 2.3.3 relate to post-modelling reality.

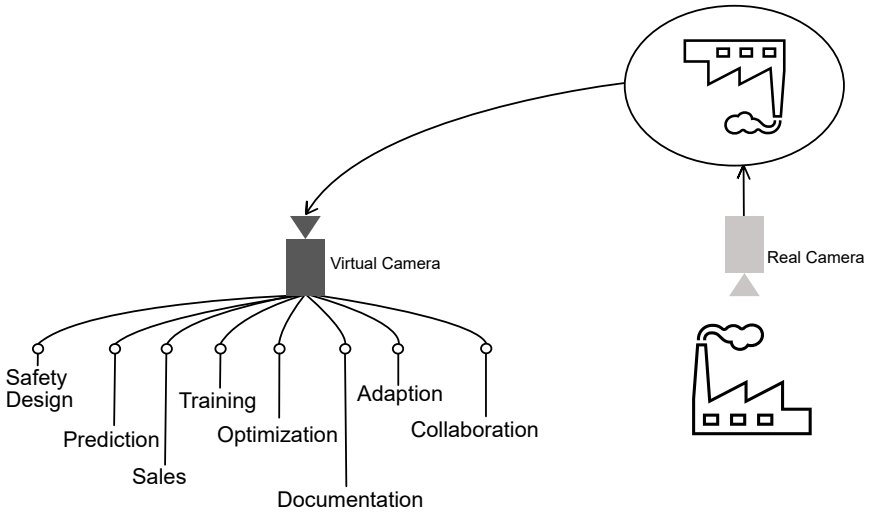


Figure 27.: Illustration of the possible applications of a Reality Capture within the Industrial Metaverse. By capturing the current static state of a scene, we have several possibilities to view it from our virtual camera and we do not need to be on site to understand the full scope of the scene.

- Safety Design: Until now, safety engineers have to be physically on site to consider an accurate safety design, as factors such as solar radiation, distances and other real attributes have to be taken into account. With NeRF, safety concepts can now be pre-formulated in a virtual domain. NeRF represents not only the geometry, but also the visual impression of a scene or environment. This includes effects such as light reflections or glare that cannot be derived directly from the geometry. This allows all potential risks and standards to be met to be considered proactively.

- Optimization: The precision in recreating the real world in a virtual space provides opportunities for process refinement. Distances, spatial relationships and complicated environmental factors are faithfully reproduced, making virtual optimization reliable. Before changes are made in the real world, this digital twin can be adjusted and fine-tuned in the virtual world to operate under optimal conditions. This approach ensures that any changes made to the actual environment are robust, reducing potential risks and delivering more effective results.

- Training: Accurate scene capture facilitates enables trainees to immerse themselves in virtual scenarios that closely mimic real-world conditions. This eliminates several challenges associated with traditional training methods. For example, there is no longer a need to shut down machinery or equipment, which can lead to downtime and lost productivity. It also eliminates the need for training participants to be on-site, providing flexibility in time and location. One of

the key benefits of this virtual approach is that training participants have the option to complete the training sessions multiple times. Under real-world conditions, repetition can be costly, time-consuming, or simply impractical due to operational constraints. In a virtual space, however, trainees can replay difficult scenarios, practice techniques, and refine their skills as many times as necessary to ensure they are well equipped and confident when confronted with the actual task.

- Sales: In the business world, many companies rely mainly on facts and figures recorded in Excel spreadsheets or in writing. This traditional method can often be abstract and difficult for customers to understand. However, when a sales representative presents a visualized solution, the customer receives a clearer and more impressive representation of the offer. This not only facilitates the customer's understanding, but also creates a more compelling sales argument. Such a visual offer makes it easier to convince customers of the quality and relevance of a product or service. Particularly when demonstrating to the customer that the solution is tailored to their environment, NeRF provides as optimal solution.

- Adaptation: Within highly detailed virtual environments, real-world concepts, practices, or products can be effectively adapted into the digital domain. This precise mirroring of reality streamlines the process of integrating physical elements into virtual spaces. For instance, when designing intricate architectural structures, digital representation allows for accurate recreation and modification of every facet of the design. This ensures consistent transitions between design phases and real-world implementation. In this context, adaptation in virtual environments provides a straightforward and efficient method for bridging the gap between conceptualization and realization.

- Collaboration: Virtual environments enable people to work together in real time, regardless of geographic boundaries. This can be valuable not only in a training process, but also when learning different scenarios, such as teaching specific machines. By integrating virtual environments, team members can work on projects together in a shared virtual space, such as when designing a safety concept.

- Documentation: Product and application details can be effortlessly visualized in a virtual environment to support a traditional written method. Consider the purchase of a complex large machine. Typically, the customer receives an introductory training session on how to set up the machine and a text-based manual at the time of purchase. However, due to long delivery times, the

details of the initial training may be forgotten by the time the machine arrives and needs to be set up - often several months later. Interpreting a text-heavy manual can then become a daunting task. Instead of relying solely on one-off instructions and subsequent written documentation, companies can offer their customers a virtual set-up solution. This interactive platform can be accessed at any time, allowing users to repeat the setup process in a virtual environment.

Even though the above applications show promising possibilities, it is important to note that these ideas are primarily based on brainstorming the idea of capturing reality. Although NeRF has showcased remarkable proficiency in producing high-quality $3D$ scene representations, the algorithm, in its current form, presents certain constraints that may challenge the realization of these concepts. This chapter will delve deeper into these limitations and the ongoing research addressing them. The practical viability of these ideas awaits further exploration and validation.

5.2. Results of neural radiance fields in capturing reality

Before delving into complex scenes and scenarios, we want to show results of how well and fast the instant-ngp algorithm works in accurately generating $3D$ representations of static scenes. In these experiments, we used shorter videos - roughly 30 seconds long, comprising 200-300 extracted frames -as the basis for our models.The COLMAP computation for these experiments required approximately 2 hours, while the training phase was impressively completed in less than 2 minutes. The rendered images were chosen to approximate, but not exactly match, the input views. Results of the experiment are displayed in Figure 28.

5.3. NeRF and it's ability to handle complex scene conditions

A distinctive strength of NeRF lies in its capability to handle complex material properties, which has historically been a challenge in computer graphics and $3D$ reconstruction. We conducted experiments with a selection of challenging objects to evaluate the capability of NeRF in faithfully and realistically representing them.

5.3.1. Generate avatars with NeRF

Human skin, with its inherent subsurface scattering where light penetrates and scatters within, brings its own set of complexities. This phenomenon, important for creating lifelike human representations, has always been a challenge to capture authentically. Furthermore, the dynamic nature of clothing, with its folds, wrinkles, and drapes, poses another layer of complexity for traditional computer graphics (CG), as does the task of realistically depicting the intricacies of the human face, from the

Figure 28.: The figure illustrates NeRF's prowess in generating $3D$ representations of static scenes. Derived from shorter videos, the COLMAP computations for these scenes lasted approximately 2 hours, with an impressive training duration of less than 2 minutes. The displayed renderings are intended to be close approximations, but not exact replicas, of the original input views. Despite minor discrepancies, the outcomes are highly commendable.

Figure 29.: This Figure presents a neural radiance field of a human avatar captured with Record3D. Remarkably, the entire process of data preparation and model training took a mere 2 minutes. While there are some areas that present challenges, the overall representation of the avatar as a neural radiance field is commendably accurate and realistic. It's also worth noting that this avatar is a static representation; it lacks a rigid body or other physical components, limiting its range of motion and interaction.

lips to the eyes and the subtleties of expression. We tested how the NeRF algorithm handles this complexity with the idea to generate avatars that can be used for remote collaboration and communication, as well as for training and interactive experiences within the Industrial Metaverse. By using NeRF to generate avatars, it is possible to create virtual experiences that authentically replicate human presence. Figure 29 displays a human neural radiance field captured with Record3D. This method allowed us to prepare the data and train this model within 2 minutes, which is quite impressive. To this end, it's important to note that while the neural radiance field can beautifully capture the details and lighting of this human, what it essentially provides is a static representation without any inherent capability to simulate rigid body dynamics or animation. This presents a limitation when trying to replicate the real-time movements and dynamics of human figures.

While the instant-ngp algorithm excels at capturing static scenes, its limitations become evident when dealing with the natural movements and dynamics of human subjects. Even minimal motion can introduce noticeable distortions in the resulting $3D$ representation, as illustrated in Figure 30.

To meet the specific requirements of the Industrial Metaverse, where nuanced human interactions may be integral, specialized NeRF algorithms are essential. For instance, *Neural Body* can reconstruct a moving human from a single-camera video, while *AniNeRF* specializes in creating animatable human models from multi-view videos. Recent work by Gafni et al. even extends the capabilities to model both the appearance and dynamics of the human face. Currently, it is necessary to use specific NeRF algorithms for each use case, tailored to the required scenarios.

Figure 30.: This illustration underscores the challenge posed by the minimal motion and dynamics of humans. While NeRF is adept at capturing static scenes, even subtle movements can introduce significant distortions to the representation. Despite these intricacies, the resultant avatar in the neural radiance field remains notably accurate and lifelike.

Figure 31.: This figure illustrates an attempt to create a neural radiance field of a hand using Record3D. Notably, influences from varying light conditions posed challenges to the modeling process. The result, while showcasing the potential of NeRF, also underlines the intricacies and difficulties inherent in accurately capturing and representing hands.

5.3.2. Generate hands with NeRF

Generating $3D$ representations of hands from images is inherently challenging due to the intricate anatomy and vast range of possible hand poses. Each hand possesses unique features, from varying finger lengths to distinct knuckle shapes. Furthermore, hands can adopt numerous poses, and during many of these poses, fingers or parts of the hand can overlap or hide other parts, leading to self-occlusion. This occlusion makes it difficult to reconstruct the $3D$ representation from $2D$ images as not all parts of the hand are always visible. Adding to this, hands often display fine details, such as wrinkles and fingernails, that are important for a realistic representation but can be challenging to capture from images. Interaction with other objects, skin deformation during movement, and changing lighting conditions further complicate the modeling process. In Industrial Metaverse, $3D$ models of human hands can be used to perform safety assessments. For example, the intrusion of a human hand into a production line can be simulated to evaluate the effectiveness of existing safety protocols and mechanisms, and also to show the customer what happens when an employee's hand gets into the equipment.

Figure 31 illustrates our result of training a neural radiance field with the instant-ngp algorithm of a hand.

As previously noted, the generated neural radiance field merely constitutes a volumetric scene function; it lacks physical properties and a rigid body structure. Consequently, this generated radiance field currently has no capability for interaction or dynamic behavior.

5.3.3. Generating large scenes with NeRF

The instant-ngp scripts operate under the assumption that all training images are oriented towards a common focal point, which is positioned at the origin. The determination of this focal point is achieved by taking a weighted average of the nearest points of convergence between the rays passing through the central pixel of all pairs

of training images. This inherently implies that optimal results are obtained when the training images are captured with their focus inwards towards the object, although a complete 360-degree view isn't mandatory (NVlabs, 2022). However, driven by the urge to push the boundaries of this tool, we embarked on an exploratory mission to determine its efficacy in capturing larger scenes.

One of the primary challenges of capturing more extensive scenes lies in the diversity of elements and the variance in lighting and shadows over a broader area. An individual object can be controlled, isolated, and often positioned under ideal lighting conditions to ensure the best capture. In contrast, an entire scene introduces complexities: dynamic lighting, overlapping objects, and a vast range of textures and materials that might not always be optimally positioned for the capturing equipment.

For our experiment, we trained two larger scenes. The data preparation process, involving COLMAP, was notably time-consuming, taking approximately 12 hours. However, the actual training phase was surprisingly swift, concluding in just 5 minutes. This rapid training period showcases the efficiency of the instant-ngp algorithm once it has the necessary data. In Figure 32, we captured a room adorned with information boards. Initially, the depiction showcases the entirety of the neural radiance field within its *aabb_scale*. As we progress through each frame, the viewer is drawn closer to the information panels, revealing the distinct lighting conditions of the scene. The neural radiance field aptly captures the vastness of this space, though there are noticeable artifacts and a discernible lack of focus on the information panels. Our aim was to virtualize this room to place the user in a virtual customer center, allowing them to explore the products on display in a VR environment.

In our pursuit to explore the versatility of the instant-ngp algorithm, we ventured into capturing another large setting: an office space furnished with two distinct desk blocks. The results, showcasing the intricate nuances of the scene, are presented in Figure 33. While the inherent challenges of capturing such a multifaceted environment were evident, the final renderings capture the essence of the space is impressive.

Figure 32.: This figure shows renderings of a large scene. The first image shows the entire scene within its bounding box. Subsequent renderings dive deeper into this radiation field and show each captured information platform. The overall result turned out very well except for the artifacts.

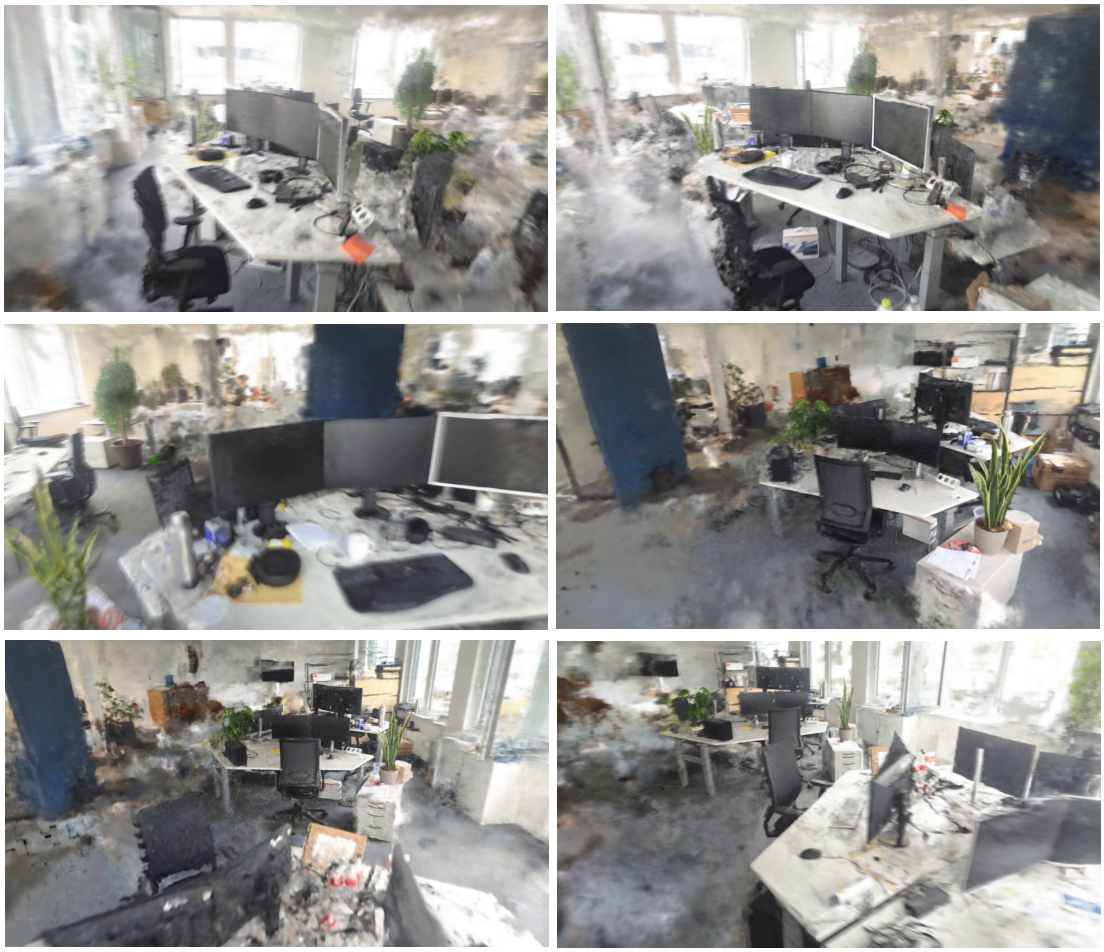


Figure 33.: This figure showcases renderings from an office setting with two distinct desk blocks. The initial four images highlight the first desk block, while the remaining two focus on the second. Despite the presence of artifacts, the results are commendable.

5.4. Generate scenarios for a sales setting

In chapter 2, we presented the transition from a product-centric model to an application-centric model. Instead of digging through static images and product descriptions on a company's websites, users create neural radiance fields of their environments and applications. A virtual representation of their environment enables deeper integration and testing of products in their actual operational context, ensuring the selection of the most appropriate solutions. Beyond just integration, users can also use these neural radiation fields to create virtual training modules, experiment with new strategies, and fine-tune their applications. Customers not only have the possibility to generate a neural radiance field from their application, but also from a product they want a solution for. In this way, sales managers can virtually place the product in a digital twin and assess its performance and suitability in different environments to create the most optimal solution. Additionally, AR technology can be used to visualize how

these product solution fit into the customers environment.

In the sections that follow, we will undertake a practical exploration, testing the generation of neural radiance fields in various customer environments and for a range of products.

5.4.1. Generate neural radiance fields from customers environment

Customers can convert their applications into a neural radiance field, allowing them to embed and evaluate potential product integrations within their unique environment. This provides a tangible way to visualize how these products would function and integrate within their specific contexts.

In a pilot test, we transformed a production facility into a neural radiance field using footage captured with a smartphone. The rendered images, illustrated in Figure 34, demonstrate NeRF's aptitude in capturing spatial intricacies and attributes of the facility with high fidelity.

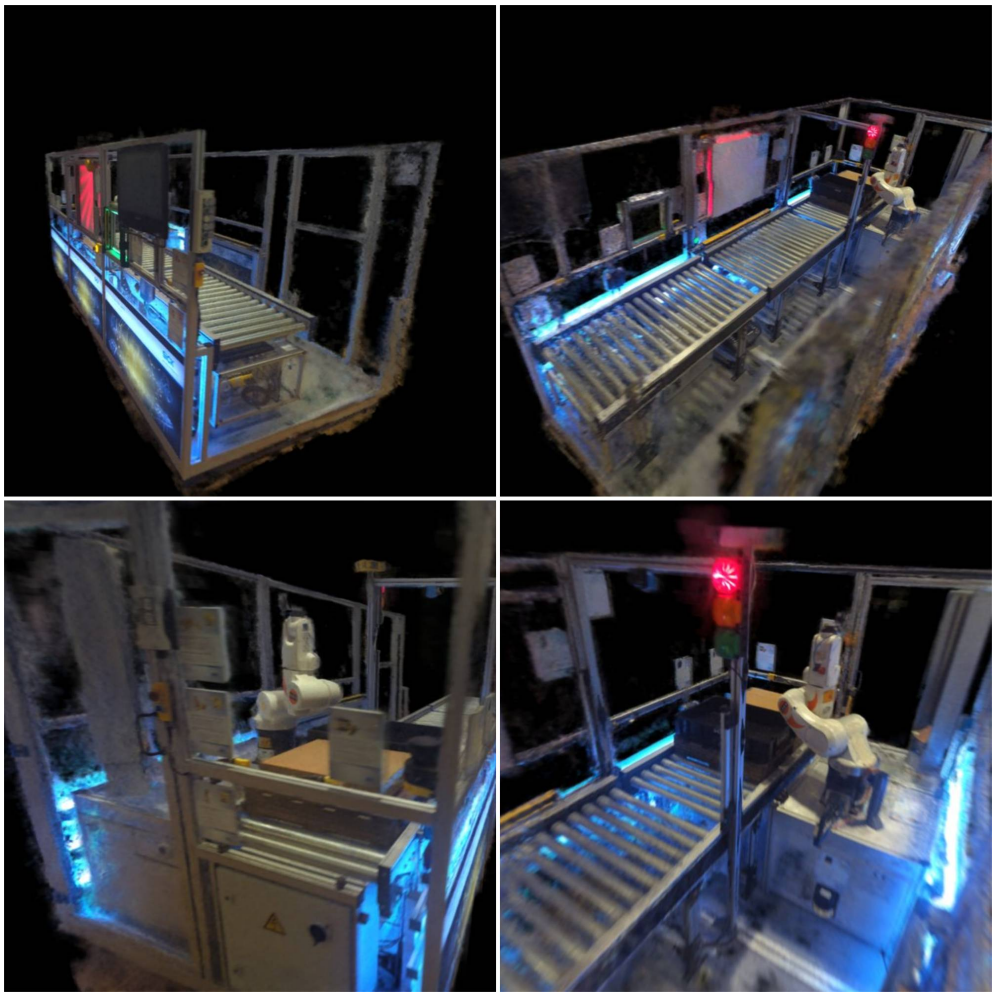


Figure 34.: Rendered views from the neural radiance field of a production application. The renderings accurately capture the spatial details of the scene.

For immersive visualization, the instant-ngp framework supports integrating the product's application into VR/AR environments. Upon connecting our headset to Steam VR, we initiate the VR mode using the "Connect to VR/AR headset" button. This immersive VR setting equips users with multiple navigation and interaction tools, as delineated by (NVlabs, 2022). They can seamlessly traverse the environment, modify the camera's vantage point, rotate, zoom and even erase points from the neural radiance field. Figure 35 shows the GUI of the framework, in which the rendered VR views are displayed.

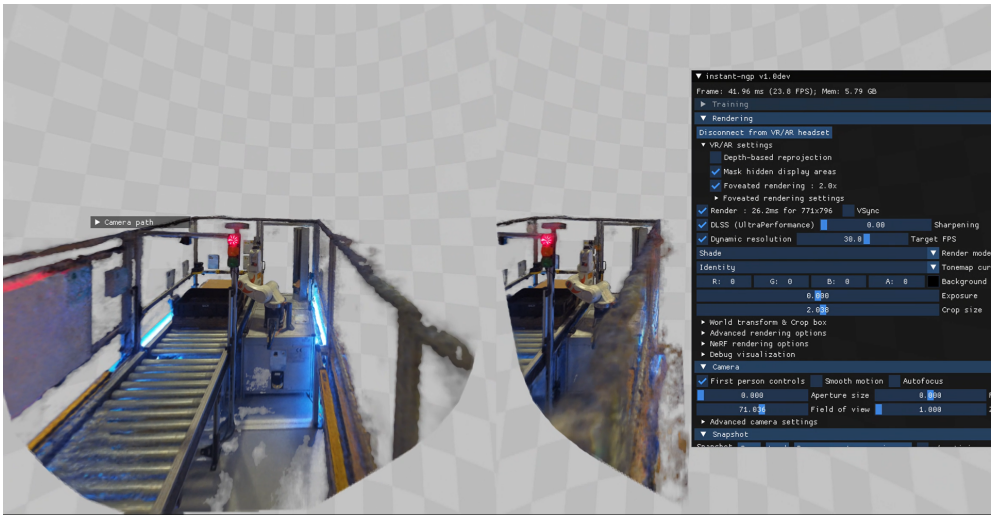


Figure 35.: Screenshot of the instant-ngp application rendering the views currently displayed in the VR headset.

The current version of instant-ngp primarily provides a viewing platform for neural radiance fields. Although it allows users to visualize results, adjust camera positions for video rendering, and navigate through scenes, it falls short in offering tools to modify these radiance fields. This lack of editability prevents us from our idea of embedding products in a neural radiance field.

Despite this challenge, we were determined to test this approach, and to use the resources currently available in the framework. Therefore, we turn our attention to the rendering capabilities provided by instant-ngp. instant-ngp offers considerable flexibility in the choice of camera settings for image rendering. This means that users can adjust various parameters related to camera placement, orientation, field of view and focal length to create images from different perspectives.

In an illustrative use case, consider a customer owning a production facility with various packages in transit. This customer is keen on testing different cameras to read the barcodes present on these packages. To facilitate this, a company specializing in these cameras offers a unique application. Within this application, customers can train a neural radiance field of their production environment. Once this radiance field is established, they can virtually position cameras at different vantage points

and view the resultant rendered images from these camera perspectives. Given that the camera configurations provided by the company are predefined, the customer can quickly gauge the effectiveness of each camera setup. This simulation and assessment process equips the customer with the tools needed to select the camera that best aligns with their production environment's requirements, ensuring critical details are consistently and accurately captured. To visually encapsulate this concept, Figure 36 showcases how different camera positions and their respective field of view settings influence the rendered images.

Figure 36.: This illustration displays the rendering results from two distinct camera positions (top and bottom) along with variations in camera settings (left and right).

This particular use case, showcased within the framework of instant-ngp, not only demonstrates the capabilities at hand but also signifies the potential for broader adaptability, allowing companies to tailor it to their unique contexts and requirements. While the visualizations in this scenario were generated via the GUI, upcoming iterations could contemplate the inclusion of a virtual reality interface. Such an advancement would amplify the spatial rendering and interactive editing capabilities. It would create an immersive milieu where users can be virtually ensconced within their application landscape, positioning and adjusting cameras as they deem fit. The transition from a graphical interface to a virtual one underscores the evolution of user interaction, elevating the experiential aspect of the entire process.

5.4.2. Generate neural radiance fields from customers product

With NeRF, customers can create an accurate digital twin of their products that can then be used for diverse applications. For example, once a product is digitally represented, it can be seamlessly integrated into another digital environment, such as a simulated production line or virtual showroom. Furthermore, such integration facilitates proactive troubleshooting, like discerning why a product isn't scanning correctly, product optimization, such as repositioning a barcode for optimal scanning, and the creation of interactive customer experiences (showcasing the efficiency of a scanner in real-time scenarios with their digitized products). To demonstrate the feasibility and accuracy of this approach, we conducted an experiment. Figure 37 highlights the precision with which both transparency and intricate radiance properties have been captured. The neural radiance field effectively incorporates the different reflections perceived from different perspectives, resulting in a rich and accurate representation.

Figure 37.: This figure showcases a captured neural radiance field of a object with transparent elements. The neural radiance field effectively captures and represents the radiance properties with varying reflections from different angles.

While we are able to save these results, their integration into digital twins presents a challenge. An important prerequisite for the integration of neural radiance field is the ability to render the radiance field separately from the digital twin scene, a feature that is not implemented in any engine (e.g. Omniverse). We therefore explored the possibility of extracting a mesh from this neural radiance field in order to place it in the context of a digital twin.

5.4.3. Generate meshes in instant-ngp

The result obtained from the neural radiance field, as depicted in Figure 37, is undeniably impressive. It stands as a compelling demonstration of the $3D$ scene representation derived from $2D$ images through the framework of instant-ngp. instant-ngp provides the functionality to extract a mesh from the radiance field, utilizing the Marching Cubes algorithm for the extraction process.

The Marching Cubes algorithm, developed by Lorensen and Cline in 1987 (Lorensen and Cline, 1987), is a method used to create triangle models of constant density surfaces from $3D$ volume elements or polygons. It takes volumetric data, represented

as a $3D$ grid of voxels, each containing information about the density of the $3D$ scene. By selecting a desired surface density value, the algorithm identifies the voxels that cross this threshold, distinguishing between the inside and outside of the object. The $3D$ grid is divided into small cubes, usually with eight voxels each. These cubes represent local regions of the volume and hold density values at their eight corners. Since each cube has eight vertices and two states (inside and outside), there are only 256 ways a surface can intersect the cube. The algorithm enumerates these cases and creates a table to look up surface-edge intersections, based on the labeling of a cube's vertices. This table contains the edges intersected for each case. For every cube, the algorithm determines which edges are intersected by the isosurface (where the density crosses the threshold). Then, based on the intersected edges, a unique 4-bit index is generated to represent the isosurface configuration within the cube. Using a pre-computed lookup table, the algorithm maps this 4-bit index to a set of triangles that approximate the isosurface within the cube. These triangles are defined by interpolating the positions of the intersected edges. The triangles generated for each cube are combined to form the final $3D$ mesh representing the isosurface of the object (Lorensen and Cline, 1987).

instant-ngp also provides parameters to optimize the resulting mesh. These parameters are *Laplacian Smoothing*, *Density Push*, and *Inflate*. *Laplacian Smoothing* is a technique used to improve the quality of a $3D$ mesh by adjusting the positions of its vertices. It works by moving each vertex toward the average position of its neighboring vertices. This process helps to reduce sharp angles and irregularities in the mesh, resulting in a smoother and more aesthetically pleasing surface. The amount of smoothing applied can be controlled by adjusting the *Laplacian Smoothing* parameter. Increasing this value will apply more smoothing, while decreasing it will retain more of the original mesh details (Sorkine, 2005). *Density push* is a parameter that allows users to control the expansion or contraction of the $3D$ mesh based on the underlying density information. It works by pushing the vertices of the mesh away from or towards the regions with higher or lower density values, respectively. This can be useful in cases where the original mesh may not align perfectly with the desired density surface. By adjusting the *Density Push* parameter, users can fine-tune the level of mesh deformation to better match the density distribution. The *Inflate* parameter is used to adjust the overall size or thickness of the $3D$ mesh. Increasing the inflation value will expand the mesh, while decreasing it will shrink it. This parameter can be helpful in adjusting the thickness of the generated surface to achieve the desired visual appearance or physical properties. By providing users with control over these mesh optimization parameters, the instant-ngp framework enables the creation of more refined and customized $3D$ meshes, tailored to specific requirements or visual preferences.

In Figure 38 we provide an illustrative example of a mesh output, specifically one derived using the default settings provided by the framework. This particular figure is a composite of three distinct images, each emphasizing a different aspect of the process. The first image portrays the original neural radiance field, giving viewers an understanding of the raw data from which the mesh is derived. The second image offers a visualization of the mesh, with an emphasis on vertex normals. This provides insight into the mesh's geometric features, highlighting how surfaces are oriented in three-dimensional space. The third image within the figure displays the mesh, but with a focus on vertex colors. This showcases the color information associated with each vertex, allowing for a comprehensive view of both the mesh's structure and its appearance.

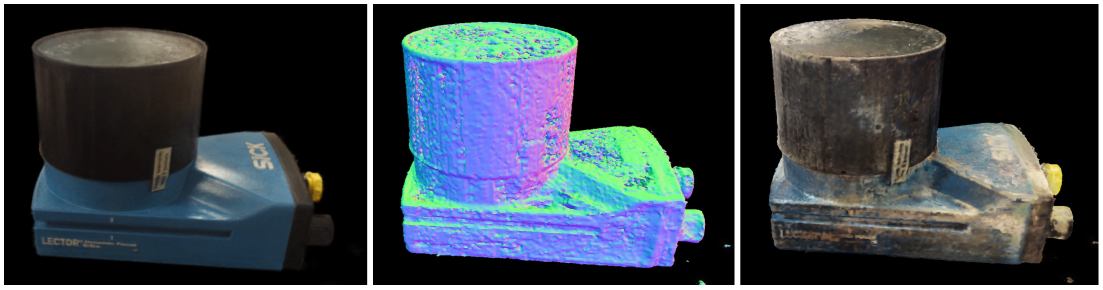


Figure 38.: Exported mesh with default values. The first image represents the neural radiance field, the second image shows the mesh with vertex normals, and the third image displays the mesh with vertex colors.

To further enhance the mesh's quality and usability, manual optimizations have been applied, as depicted in Figure 39. On the left side, we illustrate the mesh without any manual editing and on the right side, the mesh has undergone small improvements using Blender modifier, resulting in a better overall appearance.

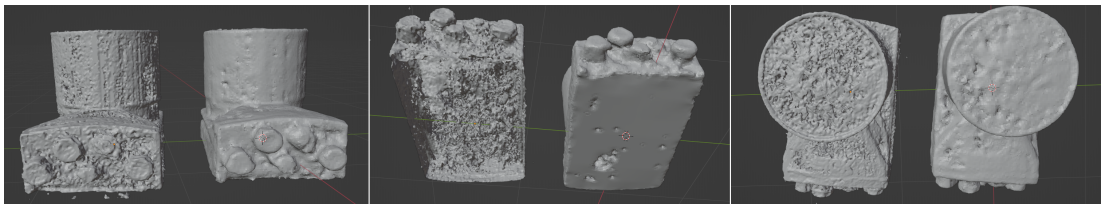


Figure 39.: Optimized exported mesh with manual editing. The left side shows the original Mesh Export without any manual editing, while on the right side, Blender edits have been applied to achieve a better result. Although the mesh may not be perfect, it provides valuable insights into the structure of the subject.

The extracted mesh loses the radiance field properties that it initially had. While materials within a radiance field are distinctly identifiable, exporting to mesh only generates a framework without associated material data. This means that transparent objects aren't extracted with their transparency intact. However, if the primary goal

is to integrate just the object's shape from the radiance field into a digital twin, this method proves suitable. Furthermore, it is faster than traditional photogrammetry techniques. The manual enhancements also significantly elevate the visual fidelity and adaptability of the mesh for its application within the digital twin environment.

instant-ngp is primarily centered around the concept of neural radiance fields. In our quest for improved results, we've integrated the user-friendly and artist-oriented framework known as *Nerfstudio*. *Nerfstudio* offers diverse export methods tailored to the needs of creators and artists, facilitating the generation of point clouds or meshes for further processing and seamless integration into downstream tools such as game engines. Figure 40 provides an illustration of an exported point cloud.

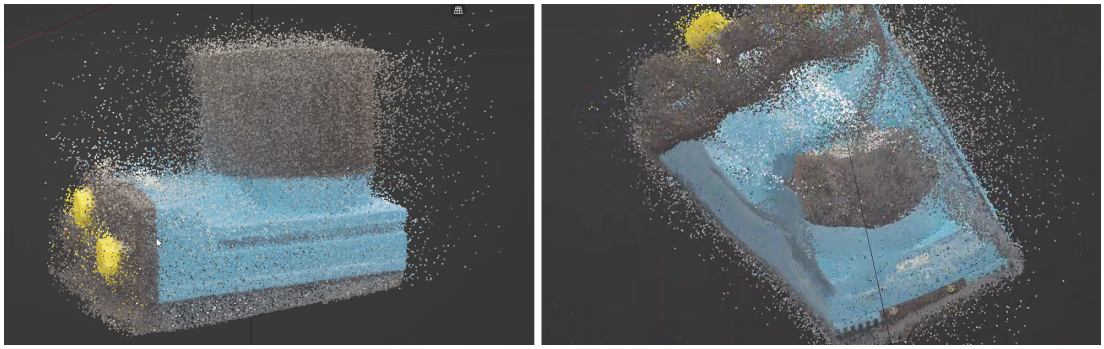


Figure 40.: Point cloud export from Nerfstudio showcasing the challenges in mesh extraction. The presence of outliers and the absence of ground points make the process non-trivial. Despite the obstacles, the point cloud provides valuable insights for further analysis and potential improvements in mesh extraction.

Examining the point cloud reveals the reasons why a mesh export is not straightforward. The presence of many outliers can complicate the process of mesh extraction, as these irregular data points can disrupt the smooth surface representation necessary for creating a coherent mesh. Additionally, the absence of points on the ground rules out the possibility of accurately connecting the mesh in this region.

Figure 41 shows a mesh generated with *Nerfstudio*, but it is obvious that the results do not improve on the quality achieved with the instant-ngp version. Mesh generation is computationally intensive, resulting in meshes with a high number of triangles, making their use in other programs like Unity challenging. Manual post-processing becomes essential if one aims to create a useful mesh despite these limitations. The lack of a standardized NeRF training file format adds further complexity, as it hinders seamless transitions between different applications without re-running the entire process. This difficulty also affects the ability to easily load neural radiance field for further editing within a web viewer.

Consequently, we can conclude that the NeRF algorithm may not be the optimal choice for extracting meshes. Currently, without generating a mesh, the created neural radiance field cannot be integrated into other platforms like *Unity*, *Omniverse*,

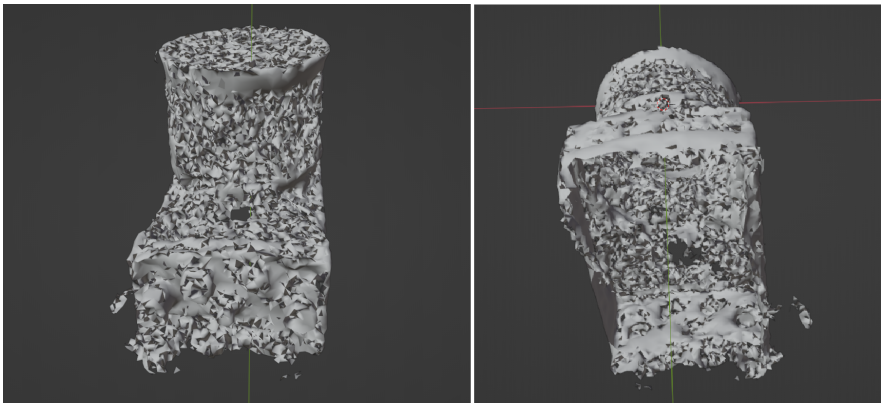


Figure 41.: Generated Mesh from *Nerfstudio* with the Poison Surface Reconstruction. As we can observe, the generated mesh does not improve on the quality achieved with the instant-ngp version

or *Unreal Engine* due to the absence of an API for these applications.

5.5. Limitations of the NeRF Algorithm

Although NeRF has taken a major role for its innovative approach in generating $3D$ scene representations, the nature of the algorithm is not without limitations that hinder its effective application in the context of the Industrial Metaverse. In the absence of a comprehensive overview of these limitations, this section aims to thoroughly examine the inherent challenges posed by the algorithm.

At its core, NeRF utilizes a *volumetric* representation as a dense field of radiance. Therefore, any modifications to the scene necessitate adjustments to this radiance field, a process that is both computationally demanding and time-intensive (Rabby and Zhang, 2023). Moreover, the neural network is trained on a specific dataset of images with its unique lightning conditions; consequently, its ability to generalize scenes captured differently are limited (Rabby and Zhang, 2023).

The architecture of NeRF models the emitted radiance but does not explicitly account for the material properties such as reflectance, roughness, or transparency. These properties are important for a realistic rendering of objects, especially when light interacts with them in complex ways. Since NeRF entangles material and lighting information in its view-dependent radiance, it is not straightforward to alter the lighting conditions without retraining the model. In traditional $3D$ modeling, the material and lighting are separate parameters, allowing users to easily change them without affecting the other. In NeRF, however, since these are combined into a single model, any attempt to modify the lighting would also inadvertently affect the material properties, making accurate relighting a challenge.

The algorithm faces difficulties in training large-scale scenes while maintaining detail and quality. Inherent to its design, instant-ngp targets object-level rather than

scene-level representation. The attempt to create several NeRFs of a scene and to combine them afterwards also fails. This would require the development of a new rendering algorithm, which must carefully evaluate which neural radiance field goes into saturation first (i.e., which has a higher integrated density) depending on the viewing direction, and then output the corresponding color.

Various frameworks, such as instant-ngp and *Nerfstudio*, employ proprietary file formats for storing neural radiance fields, complicating data interchange and standardization. These platforms also have unique data preparation requirements, mandating specific procedures like COLMAP application in each case. As NeRF evolves, new algorithms alter rendering techniques, constraining trained NeRF models to their specific rendering environment.

The integration of a neural radiance field into frameworks like Unity, Unreal or Omniverse presents another barrier. NeRF relies on unique rendering functions, demanding external engines to adopt this specific mechanism. The volumetric representation also complicates embedding into scenes with disparate lighting conditions.

Lastly, generating mesh from NeRF presents challenges. While the algorithm excels in capturing lighting conditions, it may not be optimal for producing mesh representations, as we saw in section 5.4.3.

These limitations constrain the applicability of NeRF in the context of the Industrial Metaverse. However, as mentioned in the introduction of this chapter, our aim is to explore the existing potential. To further this, we will now look at research efforts to address these limitations.

5.5.1. Editing neural radiance fields

The editing of virtual $3D$ representations is an important function within the Industrial Metaverse. By modifying these virtual environments, we can simulate different configurations to optimize production lines. For instance, altering the positions of manufacturing machines or adjusting the dimensions of a conveyor belt can be done to achieve more efficient workflows. Additionally, the utility of this feature extends to collaborative settings. Virtual reality-based remote collaboration allows multiple stakeholders to make and view changes in real-time. Furthermore, these virtual spaces can be customized to facilitate a variety of training programs, thereby enhancing skill development and task proficiency.

One research effort that focuses on editing NeRFs is *NeRFShop*. The goal is to create an interactive NeRF editing solution that operates on entire scenes and offers free-form direct manipulation capabilities, encompassing deformation, translation, scaling, rotation, and object duplication, all within the volumetric representation. Notably, this method is build on the instant-ngp algorithm (Jambon et al., 2023).

The idea is to allow users to interactively select and deform objects through cage-

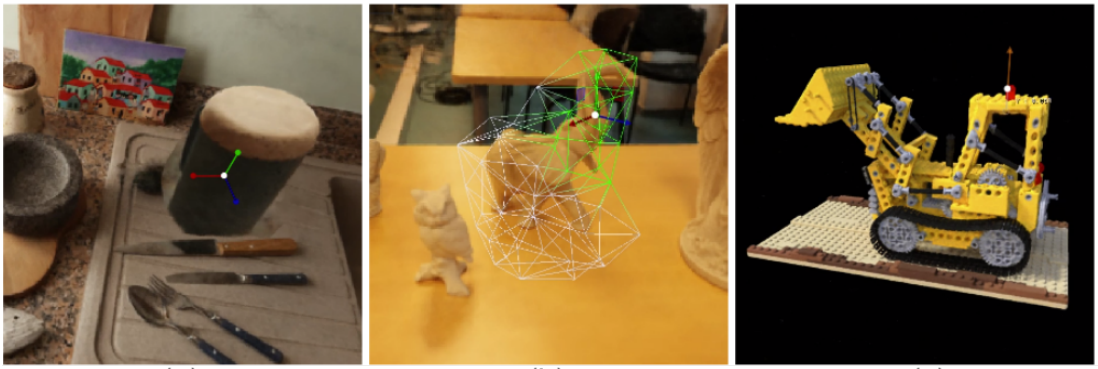


Figure 42.: This figure demonstrates the manipulation capabilities within a scene. (a) displays a cup accompanied by a coordinate system, allowing for movement of the cup in the x, y, z directions. It was already moved from the left side in x-direction. (b) highlights the cage-based system. Within this cage, specific points can be selected for editing. In the Lego representation (c) the roof is translated in the y direction, illustrating the possibility of translation manipulation (Jambon et al., 2023).

based transformations. The framework provides fine scribble-based user control for the selection of regions or objects to edit, semi-automatic cage creation, and interactive volumetric manipulation of scene content due to the GPU-friendly two-level interpolation scheme. Further, the authors introduce a preliminary approach that reduces potential resulting artifacts of these transformations with a volumetric membrane interpolation technique inspired by Poisson image editing and provide a process that "distills" the edits into a standalone NeRF representation (Jambon et al., 2023). Figure 42 showcases the capabilities of scene editing using the described approach.

Another approach, *NeuralEditor*, enables neural radiance fields natively editable for general shape editing tasks. Their key insight is leveraging an explicit point cloud representation to form NeRFs. This stems from the understanding that NeRF rendering can be seen as "plotting" the related $3D$ point cloud onto a $2D$ image plane. In pursuit of this, *NeuralEditor* innovatively uses a rendering strategy rooted in deterministic integration within K-D tree-steered density-adaptive voxels. This approach not only offers superior rendering quality but also optimizes to yield accurate point clouds. Shape modifications are then carried out by mapping relevant points between these point clouds. Comprehensive assessments indicate that *NeuralEditor* sets new benchmarks in tasks like shape deformation and scene morphing. Importantly, it can execute both immediate zero-shot inferences and offer the possibility for refining edits in the scene (Chen et al., 2023).

5.5.2. Shape and Reflectance

Zhang et al. address the complex task of extracting both the shape and varying reflectance properties of an object based on multi-view images captured under an un-

defined lighting condition. Their methodology, termed Neural Radiance Factorization (NeRFactor), succeeds in rendering new object views under diverse environmental lighting, as well as enabling material property adjustments. The key of *NeRFactor* lies in its ability to transition from the volumetric form of an object, as represented by NeRF, to a more malleable surface-based model. During this transition, the system also simultaneously refines the object's geometry and calculates its spatially-variable reflectance and lighting conditions.

Specifically, *NeRFactor* employs unsupervised techniques to recover fields of surface normals, light visibility, albedo, and Bidirectional Reflectance Distribution Functions (BRDFs). It achieves this using a combination of a re-rendering loss metric, elementary smoothness priors, and a data-driven BRDF model derived from real-world measurements. This explicit modeling of light visibility enables *NeRFactor* to distinguish between shadows and albedo, facilitating the generation of either soft or hard shadows under various lighting conditions.

NeRFactor demonstrates robust performance in creating convincing $3D$ models for free-viewpoint relighting, even in less-than-ideal capture setups, and excels in both synthetic and real-world scenes. Comparative evaluations indicate that NeRFactor surpasses both traditional and deep learning-based approaches across multiple tasks (Zhang et al., 2021).

5.5.3. Generalization to novel scenes

// TODO

5.5.4. Integration in Engines

// TODO

5.5.5. NeRF on Large Scenes

Block-NeRF is an evolved form of NeRF, tailored to depict expansive environments. In particular, the developers highlight the significance of breaking down a large scene, like that of city-scale environments spanning numerous blocks, into distinct NeRFs for individual training. This strategic decomposition not only separates the rendering time from the overall size of the scene but also allows rendering of vast expanses and supports updates on a per-block basis. To ensure NeRF's resilience against data collected over extended periods and varying environmental conditions, several architectural modifications were integrated. These include the incorporation of appearance embeddings, the refinement of learned poses, and adjustable exposure for every unique NeRF. Additionally, a technique for synchronizing appearances between neighboring NeRFs has been introduced, ensuring a fluid amalgamation of adjacent blocks. By

harnessing the power of 2.8 million images, a grid of *Block-NeRFs* was constructed, marking the creation of the most expansive neural scene depiction till date, showcasing a comprehensive neighborhood in San Francisco (Tancik et al., 2022).

Figure 43.: The illustration showcases the capability of *Block-NeRF* to execute large-scale scene reconstructions by leveraging multiple compact NeRFs, each optimized for memory. During the inference process, *Block-NeRF* adeptly integrates renderings from pertinent NeRFs tailored for the specific zone. This image highlights a reconstruction of the Alamo Square neighborhood in San Francisco, derived from data aggregated over a span of 3 months. A noteworthy aspect of *Block-NeRF* is its capacity to refresh distinct environment blocks without necessitating retraining on the entire landscape, evident from the developmental progress on the right (Tancik et al., 2022).

5.5.6. Mesh Generation with Neural Rendering

In addressing the challenge of mesh generation from neural radiance fields, a solution named *NeRFMeshing* (Rakotosaona et al., 2023). *NeRFMeshing* introduces an advanced pipeline that adeptly extracts geometrically consistent meshes from trained NeRF models. This approach, while being time-efficient, yields meshes that are geometrically precise and equipped with neural colors, facilitating real-time rendering on standard hardware. Central to this is the Signed Surface Approximation Network (SSAN), a post-processing component of the NeRF pipeline, calibrated to accurately represent both surface and appearance. SSAN works by estimating a Truncated Signed Distance Field (TSDF) in conjunction with a feature appearance field. The outcome is the extraction of a $3D$triangle mesh that represents the scene, which is subsequently rendered using an appearance network to generate view-dependent color gradients (Rakotosaona et al., 2023). When benchmarked against other techniques, *NeRFMeshing* stands out in multiple facets. It is compatible with diverse NeRF architectures, making the assimilation of future advancements seamless. The technique is adept at managing unbounded scenarios and intricate, non-lambertian surfaces. Furthermore, *NeRFMeshing* preserves the high-resolution details characteristic of neural radiance fields, encompassing view-dependent nuances and reflections, solidifying its viability for real-time novel view synthesis (Rakotosaona et al., 2023).

The goal of Neural Radiance Fields is indeed to capture the radiance field of a

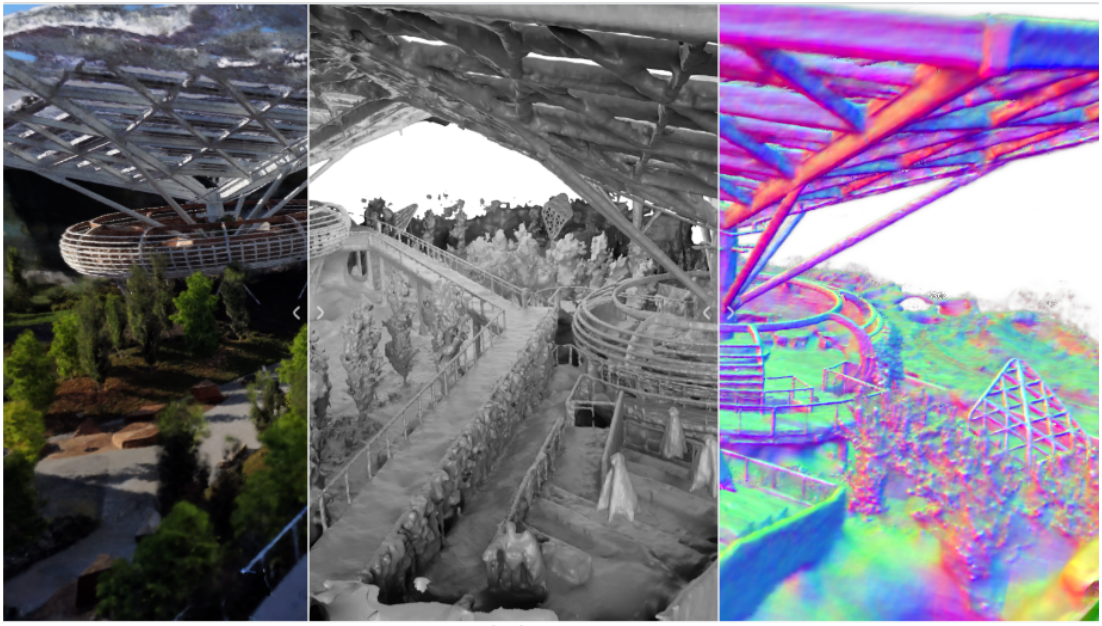


Figure 44.: Results of *Neuralangelo*: (a) RGB View Synthesis, (b) Surface Reconstruction, (c) Surface Normals. These visualizations illustrate the capabilities of *Neuralangelo* in generating realistic views, reconstructing surfaces, and depicting surface normals. These abilities hold promising prospects for highly accurate $3D$ model creation from image and video data (Li et al., 2023b).

specific scene. However, the extraction to meshes from these radiance fields currently leads to the loss of this valuable property. While NeRF results may look visually appealing, the generated meshes lack structure and detail. In response to these challenges, ongoing research focuses on novel approaches to mesh generation with other neural rendering techniques. *Neuralangelo* (Li et al., 2023b) introduces a framework for high-fidelity $3D$ surface reconstruction from RGB video captures using the signed distance function. This algorithm utilizes the multi-resolution hash encoding from instant-ngp to represent positions. The encoded features are then fed into a neural signed distance function representing the underlying $3D$ scene. Additionally, a color MLP is employed to synthesize images using neural surface rendering. *Neuralangelo* effectively recovers dense scene structures of both object-centric captures and large-scale indoor/outdoor scenes with extremely high fidelity, enabling detailed large-scale reconstruction from RGB videos (Li et al., 2023b).

Figure 44 illustrates the results of the *Neuralangelo* algorithm. Three main aspects can be seen here: (a) Result of RGB view synthesis, where realistic color views are generated from the captured data, (b) Visualization of the reconstructed surface derived from the given data, (c) Surface normals, which provide important information about the orientation and structure of the reconstructed surface. These visualizations illustrate the ability to recover dense scene structures with accuracy (Li et al., 2023b).

5.6. Discussion

NeRFs high fidelity in capturing and rendering $3D$ objects could revolutionize how industrial assets are visualized within the metaverse. Traditional methods often rely on $3D$ models that may not capture the full complexity and nuance of industrial machinery. NeRF, with its ability to create detailed, view-dependent renderings, could elevate the realism of these digital twins to an unprecedented level, thereby enhancing the reliability of simulations and virtual inspections.

A detailed and realistic $3D$ representation would allow engineers, designers, and operators to interact with industrial systems virtually as if they were physically present. This is particularly beneficial for cross-disciplinary teams spread across different geographical locations, providing a unified, real-time platform for interaction.

One significant limitation is NeRFs inherently static nature. Industrial environments are dynamic, with machinery and conditions constantly changing. NeRFs static models could pose challenges in representing real-time modifications, thereby restricting its effectiveness for certain real-time industrial applications.

Currently, various NeRF developments are tailored for specific tasks, lacking a one-size-fits-all solution. Each new NeRF iteration tends to alter not just the rendering capabilities but also the function, input, or output parameters. This lack of uniformity creates challenges for widespread industrial adoption, as it hinders the development of a unified platform that can handle multiple tasks without requiring specialized modifications.

While NeRF holds immense potential for enriching the Industrial Metaverse, its current limitations - ranging from its static nature, task-specific designs, and computational requirements - need to be addressed for full-scale industrial application.

6. Conclusions and Outlook

In summary, the proposed framework based on the instant-ngp algorithm provides a comprehensive and user-friendly end-to-end solution for Neural Radiance Fields. Its modular architecture, divided into a frontend and a backend, facilitates easy data preparation, training, and result retrieval. The web-based interface further enhances accessibility, while the backend's compatibility with NVIDIA graphics cards utilizing CUDA allows for optimized training. More specifically, we presented the following content in this thesis: In Chapter 2, we first provided detailed technical background on the Neural Radiance Field algorithm which forms the basis of this research. Building on this, we then introduced improvements of NeRF, including a detailed explanation of the **instabt-ngp!** (**instabt-ngp!**) algorithm, which was used for this thesis.

Then, in Chapter 3 ... xxx

6.1. Outlook

This thesis was written all about NeRF and components of Neural Rendering. The newest

Bibliography

Neural radiance fields mildenhall et. al. [Accessed: 2023-08-28]. (n.d.).

Klm cityhopper introduces virtual reality training for pilots [Accessed: 2023-09-18]. (2020).

Adelson, E. H., & Bergen, J. R. (1991). The plenoptic function and the elements of early vision. In *Computational models of visual processing* (pp. 3–20).

Anaconda, I. (2023). Anaconda.

Ball, M. (2022). *The metaverse: And how it will revolutionize everything*. Liveright Publishing Corporation, a division of W.W. Norton; amp; Company.

Barron, J. T., Mildenhall, B., Tancik, M., Hedman, P., Martin-Brualla, R., & Srinivasan, P. P. (2021). Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields. *ICCV*.

Barron, J. T., Mildenhall, B., Verbin, D., Srinivasan, P. P., & Hedman, P. (2022). Mip-nerf 360: Unbounded anti-aliased neural radiance fields. *CVPR*.

Barron, J. T., Mildenhall, B., Verbin, D., Srinivasan, P. P., & Hedman, P. (2023). Zip-nerf: Anti-aliased grid-based neural radiance fields. *arXiv*.

Barron, J. (2021, March). Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields.

Bhalgat, Y. (2022). Hashnerf-pytorch.

Bitkom, e. (2022). A guidebook to the metaverse: Leitfaden 2022.

Caulfield, B. (2022).

Chen, A., Xu, Z., Geiger, A., Yu, J., & Su, H. (2022a). Tensorf: Tensorial radiance fields. *European Conference on Computer Vision (ECCV)*.

Chen, X., Zhang, Q., Li, X., Chen, Y., Feng, Y., Wang, X., & Wang, J. (2022b). Hallucinated neural radiance fields in the wild. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 12943–12952.

Chen, J.-K., Lyu, J., & Wang, Y.-X. (2023). NeuralEditor: Editing neural radiance fields via manipulating point clouds. *CVPR*.

Contributors, X. (2022). Openxrlab neural radiance field toolbox and benchmark.

döt Net, I., Müller, T., Antoniou, P., Aro, E., & Smith, T. (2021).

Docker. (2023).

Fridovich-Keil and Yu, Tancik, M., Chen, Q., Recht, B., & Kanazawa, A. (2022). Plenoxels: Radiance fields without neural networks. *CVPR*.

Gafni, G., Thies, J., Zollhöfer, M., & Nießner, M. (2021). Dynamic neural radiance fields for monocular 4d facial avatar reconstruction. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 8649–8658.

Gortler, S. J., Grzeszczuk, R., Szeliski, R., & Cohen, M. F. (1996). The lumigraph. *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, 43–54.

Heo, H., Kim, T., Lee, J., Lee, J., Kim, S., Kim, H. J., & Kim, J.-H. (2023). Robust camera pose refinement for multi-resolution hash encoding.

Hu, S.-M., Liang, D., Yang, G.-Y., Yang, G.-W., & Zhou, W.-Y. (2020). Jittor: A novel deep learning framework with meta-operators and unified graph execution. *Science China Information Sciences*, *63*(222103), 1–21.

Huang, S., Gojcic, Z., Wang, Z., Williams, F., Kasten, Y., Fidler, S., Schindler, K., & Litany, O. (2023). Neural lidar fields for novel view synthesis. *Proceedings of the IEEE/CVF International Conference on Computer Vision*.

IBM. (2023). How industry 4.0 technologies are changing manufacturing.

Jambon, C., Kerbl, B., Kopanas, G., Diolatzis, S., Leimkühler, T., & Drettakis, G. (2023). Nerfshop: Interactive editing of neural radiance fields". *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, *6*(1).

Jun-Seong, K., Yu-Ji, K., Ye-Bin, M., & Oh, T.-H. (2022). Hdr-plenoxels: Self-calibrating high dynamic range radiance fields. *ECCV*.

Kanawaza, A. (2023). Nerfstudio: A modular framework for neural radiance field development.

Kazhdan, M., Bolitho, M., & Hoppe, H. (2006). Poisson Surface Reconstruction. In A. Sheffer & K. Polthier (Eds.), *Symposium on geometry processing*. The Eurographics Association.

Kerbl, B., Kopanas, G., Leimkühler, T., & Drettakis, G. (2023). 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, *42*(4).

Levoy, M., & Hanrahan, P. (1996). Light field rendering, 31–42.

Li, R., Gao, H., Tancik, M., & Kanazawa, A. (2023a). Nerfacc: Efficient sampling accelerates nerfs. *arXiv preprint arXiv:2305.04966*.

Li, Z., Müller, T., Evans, A., Taylor, R. H., Unberath, M., Liu, M.-Y., & Lin, C.-H. (2023b). Neuralangelo: High-fidelity neural surface reconstruction. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Li, Z., Wang, Q., Cole, F., Tucker, R., & Snavely, N. (2023c). Dynibar: Neural dynamic image-based rendering. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Lorensen, W. E., & Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, *21*(4), 163–169.

Martin-Brualla, R., Radwan, N., Sajjadi, M. S. M., Barron, J. T., Dosovitskiy, A., & Duckworth, D. (2021). NeRF in the wild: Neural radiance fields for unconstrained photo collections. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 7206–7215.

Max, N. (1995). Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, *1*(2), 99–108.

Meng, Q., Chen, A., Luo, H., Wu, M., Su, H., Xu, L., He, X., & Yu, J. (2021). Gnerf: Gan-based neural radiance field without posed camera. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, 6331–6341.

Mildenhall, B., Srinivasan, P. P., Cayon, R. O., Kalantari, N. K., Ramamoorthi, R., Ng, R., & Kar, A. (2019a). Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *CoRR*, *abs/1905.00889*.

Mildenhall, B., Srinivasan, P. P., Ortiz-Cayon, R., Kalantari, N. K., Ramamoorthi, R., Ng, R., & Kar, A. (2019b). Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics (TOG)*.

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:2003.08934 [cs]].

Mildenhall, B., Hedman, P., Martin-Brualla, R., Srinivasan, P. P., & Barron, J. T. (2022). NeRF in the dark: High dynamic range view synthesis from noisy raw images. *CVPR*.

Müller, T., Evans, A., Schied, C., & Keller, A. (2022). Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, *41*(4), 102:1–102:15.

NVIDIA. (2021).

NVIDIA. (2022).

NVIDIA. (2023).

NVlabs. (2022). Nvlabs/instant-ngp: Instant neural graphics primitives: Lightning fast nerf and more.

pip developers, T. (n.d.). Pyproject.toml.

Peng, S., Dong, J., Wang, Q., Zhang, S., Shuai, Q., Zhou, X., & Bao, H. (2021a). Animatable neural radiance fields for modeling dynamic human bodies. *ICCV*.

Peng, S., Zhang, Y., Xu, Y., Wang, Q., Shuai, Q., Bao, H., & Zhou, X. (2021b). Neural body: Implicit neural representations with structured latent codes for novel view synthesis of dynamic humans. *CVPR*.

poetry. (2023). Python packaging and dependency management made easy.

Pumarola, A., Corona, E., Pons-Moll, G., & Moreno-Noguer, F. (2020). D-NeRF: Neural Radiance Fields for Dynamic Scenes. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

Rabby, A. S. A., & Zhang, C. (2023). Beyondpixels: A comprehensive review of the evolution of neural radiance fields.

Rakotosaona, M.-J., Manhardt, F., Arroyo, D. M., Niemeyer, M., Kundu, A., & Tombari, F. (2023). Nerfmeshing: Distilling neural radiance fields into geometrically-accurate 3d meshes.

Reisig, W., & Freytag, J.-C. (2006). *Informatik - aktuelle themen im historischen kontext*. Springer Berlin Heidelberg.

Reinecke, D. N., Hinzen, L., Mühlpfort, M., & Wafa, D. H. A. E. (2021). Urban digital twins for urban development of the future.

Reiser, C., Peng, S., Liao, Y., & Geiger, A. (2021). Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. *International Conference on Computer Vision (ICCV)*.

Reiser, C., Szeliski, R., Verbin, D., Srinivasan, P. P., Mildenhall, B., Geiger, A., Barron, J. T., & Hedman, P. (2023). Merf: Memory-efficient radiance fields for real-time view synthesis in unbounded scenes. *SIGGRAPH*.

Schönberger, J. L., & Frahm, J.-M. (2016). Structure-from-motion revisited. *Conference on Computer Vision and Pattern Recognition (CVPR)*.

Schwarz, K., Liao, Y., Niemeyer, M., & Geiger, A. (2020). GRAF: generative radiance fields for 3d-aware image synthesis. *CoRR*, *abs/2007.02442*.

shapiro, D. (2023).

SIEMENS. (2023). The emergent industrial metaverse.

Sitzmann, V., Zollhöfer, M., & Wetzstein, G. (2020). Scene representation networks: Continuous 3d-structure-aware neural scene representations.

Slater, M. (2022). Exploring neural graphics primitives.

Soerensen, A. (2023). Das industrielle metaverse und die chancen fuer sick.

Sorkine, O. (2005). Laplacian Mesh Processing. In Y. Chrysanthou & M. Magnor (Eds.), *Eurographics 2005 - state of the art reports*. The Eurographics Association.

Source, M. O. (2023).

Sun, C., Sun, M., & Chen, H. (2022). Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. *CVPR*.

Takikawa, T., Perel, O., Tsang, C. F., Loop, C., Litalien, J., Tremblay, J., Fidler, S., & Shugrina, M. (2022). Kaolin wisp: A pytorch library and engine for neural fields research.

Tancik, M., Srinivasan, P. P., Mildenhall, B., Fridovich-Keil, S., Raghavan, N., Singhal, U., Ramamoorthi, R., Barron, J. T., & Ng, R. (2020). Fourier features let networks learn high frequency functions in low dimensional domains.

Tancik, M., Casser, V., Yan, X., Pradhan, S., Mildenhall, B., Srinivasan, P., Barron, J. T., & Kretzschmar, H. (2022). Block-NeRF: Scalable large scene neural view synthesis. *arXiv*.

Tang, J., Chen, X., Wang, J., & Zeng, G. (2022a). Compressible-composable nerf via rank-residual decomposition.

Tang, J., Chen, X., Wang, J., & Zeng, G. (2022b). Compressible-composable nerf via rank-residual decomposition. *arXiv preprint arXiv:2205.14870*.

Tancik, M., Weber, E., Ng, E., Li, R., Yi, B., Wang, T., Kristoffersen, A., Austin, J., Salahi, K., Ahuja, A., Mcallister, D., Kerr, J., & Kanazawa, A. (2023). Nerfstudio: A modular framework for neural radiance field development. *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Proceedings*.

Tang, J. (2022). Torch-ngp: A pytorch implementation of instant-ngp [https://github.com/ashawkey/torch-ngp].

Tao, T., Gao, L., Wang, G., Lao, Y., Chen, P., Zhao, H., Hao, D., Liang, X., Salzmann, M., & Yu, K. (2023). Lidar-nerf: Novel lidar view synthesis via neural radiance fields.

Tewari, A., Fried, O., Thies, J., Sitzmann, V., Lombardi, S., Sunkavalli, K., Martin-Brualla, R., Simon, T., Saragih, J. M., Nießner, M., Pandey, R., Fanello, S. R., Wetzstein, G., Zhu, J., Theobalt, C., Agrawala, M., Shechtman, E., Goldman, D. B., & Zollhöfer, M. (2020). State of the art on neural rendering. *CoRR*, *abs/2004.03805*.

Time. (2022).

Tretschk, E., Tewari, A., Golyanik, V., Zollhöfer, M., Lassner, C., & Theobalt, C. (2021). Non-rigid neural radiance fields: Reconstruction and novel view synthesis of a dynamic scene from monocular video. *IEEE International Conference on Computer Vision (ICCV)*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023). Attention is all you need.

Verbin, D., Hedman, P., Mildenhall, B., Zickler, T., Barron, J. T., & Srinivasan, P. P. (2022). Ref-NeRF: Structured view-dependent appearance for neural radiance fields. *CVPR*.

Wang, Z., Bovik, A., Sheikh, H., & Simoncelli, E. (2004). Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, *13*(4), 600–612.

Wang, Z., Wu, S., Xie, W., Chen, M., & Prisacariu, V. A. (2021). NeRF−−: Neural radiance fields without known camera parameters. *arXiv preprint arXiv:2102.07064*.

Weng, C.-Y., Curless, B., Srinivasan, P. P., Barron, J. T., & Kemelmacher-Shlizerman, I. (2022). HumanNeRF: Free-viewpoint rendering of moving people from monocular video. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 16210–16220.

Williams, L. (1983). Pyramidal parametrics. *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, 1–11.

Xie, Y., Takikawa, T., Saito, S., Litany, O., Yan, S., Khan, N., Tombari, F., Tompkin, J., Sitzmann, V., & Sridhar, S. (2021). Neural fields in visual computing and beyond. *CoRR*, *abs/2111.11426*.

Yariv, L., Hedman, P., Reiser, C., Verbin, D., Srinivasan, P. P., Szeliski, R., Barron, J. T., & Mildenhall, B. (2023). Bakedsdf: Meshing neural sdfs for real-time view synthesis. *arXiv*.

Yen-Chen, L., Florence, P., Barron, J. T., Rodriguez, A., Isola, P., & Lin, T.-Y. (2021). Inerf: Inverting neural radiance fields for pose estimation. *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 1323–1330.

Yen-Chen, L. (2020). Nerf-pytorch.

Yoonwoo, J., Seungjoo, S., & Kibaek, P. (2022). Kakaobrain/nerf-factory: An awesome pytorch nerf library.

Yu, A., Ye, V., Tancik, M., & Kanazawa, A. (2020). Pixelnerf: Neural radiance fields from one or few images. *CoRR*, *abs/2012.02190*.

Yu, A., & Jain, A. (2023). Luma ai - join us.

Yue Luo, Y.-P. C. (2022). Arcnerf: Nerf-based object/scene rendering and extraction framework.

Zhang, K., Riegler, G., Snavely, N., & Koltun, V. (2020). Nerf++: Analyzing and improving neural radiance fields. *CoRR*, *abs/2010.07492*.

Zhang, X., Srinivasan, P. P., Deng, B., Debevec, P., Freeman, W. T., & Barron, J. T. (2021). NeRFactor: Neural factorization of shape and reflectance under an unknown illumination. *ACM Transactions on Graphics*, *40*(6), 1–18.

Zhang, J., Zhang, F., Kuang, S., & Zhang, L. (2023). Nerf-lidar: Generating realistic lidar point clouds with neural radiance fields.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Furtwangen, 10.10.2023 Sabine Schleise

A. Improvements of the original NeRF method

A.1. A Multiscale Representation for Anti-Aliasing Neural Radiance Fields

This section addresses the substantial limitations encountered when rendering images with varying resolutions or capturing scenes from different distances, as depicted in Figure 45. While the renderings produced with NeRF are visually impressive at full resolution and achieve a high structural similarity score (structural similarity (SSIM)) Wang et al. (2004), which serves as a quantitative measure of the perceptual quality and similarity between the rendered image and the ground truth, problems arise when the renderings are viewed from a distance or at different resolutions. These problems manifest themselves in the form of jagged artifacts or aliasing, resulting in a significant drop in the SSIM score. The following video 46 provides a visual demonstration of these problems, where the ground truth is displayed on the right side and the jagged artifacts are observed in the NeRF renderings on the left side, which were generated using training inputs at the same distance, labeled as *Single Scale Training* in the video. Trying to address this problem by training neural radiance fields on multi-resolution data have not provided a satisfactory solution. While there is a slight improvement in performance at low resolutions, the quality worsens at higher resolutions, which is undesirable. In particular, the NeRF renderings exhibit excessive blurring in close-up views and exhibit aliasing artifacts in distant views. To overcome these challenge, the authors Barron et al. of the original NeRF proposed the mip-NeRF algorithm, a multiscale representation for anti-aliasing Neural Radiance Fields. This approach aims to reduce the aliasing and scale-related issues encountered with rendering the original neural radiance field.

mip-NeRF (multum in parvo NeRF as in "mipmap" Barron et al. (2021)) is inspired by the mipmapping Williams (1983) technique commonly used in computer graphics rendering pipelines to mitigate aliasing artifacts. Mipmapping involves generating a series of scaled-down versions, known as mipmaps, of a signal such as an image or texture map. The selection of an appropriate mipmap scale for a ray is based on the projection of the pixel footprint onto the intersected geometry. However, implementing this technique directly for neural volumetric representations like NeRF can be computationally intensive, as it requires numerous MLP evaluations to render a single ray and considerable time to reconstruct a scene Barron et al. (2021).

Figure 45.: In the original NeRF paper Mildenhall et al. (2020), the main results were on scenes of objects floating in space, with a hemisphere of cameras around each object pointing inwards, visualized as camera frustums. Notably, all cameras are placed equidistant from the object, allowing NeRF to perform view synthesis without scale or aliasing considerations. However, the introduction of additional cameras placed at varying distances from the object reveals the limitations of NeRF as a single-scale model attempting to address a multi-scale problem Barron (2021, March).

The proposed solution, mip-NeRF, extends the capabilities of NeRF by simultaneously representing the prefiltered radiance field across a continuous range of scales as shown in figure 47. In mip-NeRF, a $3D$ Gaussian is used as input to define the region over which the radiance field should be integrated. Renderings of prefiltered pixels are obtained by querying mip-NeRF at intervals along a cone, utilizing Gaussians to approximate the conical frustums corresponding to each pixel. To encode a $3D$ position and its surrounding Gaussian region, a new encoding called integrated positional encoding (IPE) is introduced. IPE is an extension of NeRF's Positional Encoding (PE) and enables the compact representation of a region in space, rather than just a single point. While PE maps a single point in space to a feature vector, IPE considers Gaussian regions of space, allowing the network to reason about sampling and aliasing. This enables a natural way to input a region of space as a query to a coordinate-based neural network. As the region widens, higher frequency features automatically diminish, providing lower-frequency inputs to the network. Conversely, as the region narrows, these features converge to the original positional encoding. By utilizing IPE, NeRF is trained to generate anti-aliased renderings. Instead of casting infinitesimal rays through each pixel, mip-NeRF casts a cone. For each queried point along a ray, the associated 3D conical frustum is considered. Different cameras viewing the same point can result in distinct conical frustums. To incorporate this information into the NeRF network, a multivariate Gaussian is fitted to the conical frustum, and the IPE is employed to create the input feature vector for the network.

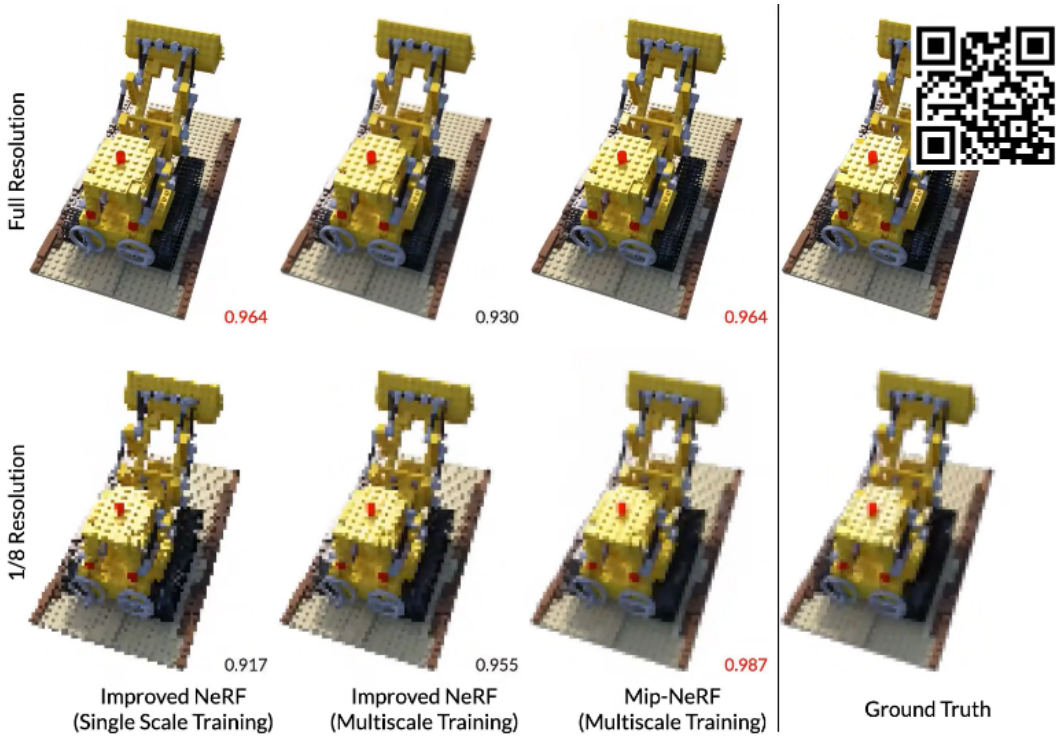


Figure 46.: The video showcases the challenges faced by NeRF in rendering scenes. On the right side, the ground truth images are displayed. The first left side shows the results of training a neural radiance field with single-scale training, while the next image demonstrates the results after applying multiscale training. These renderings exhibit jagged artifacts and aliasing when viewed at different resolutions. Finally, next to the ground truth, the results from the new approach mipnerf are displayed, which aims to address these challenges and improve the rendering quality.

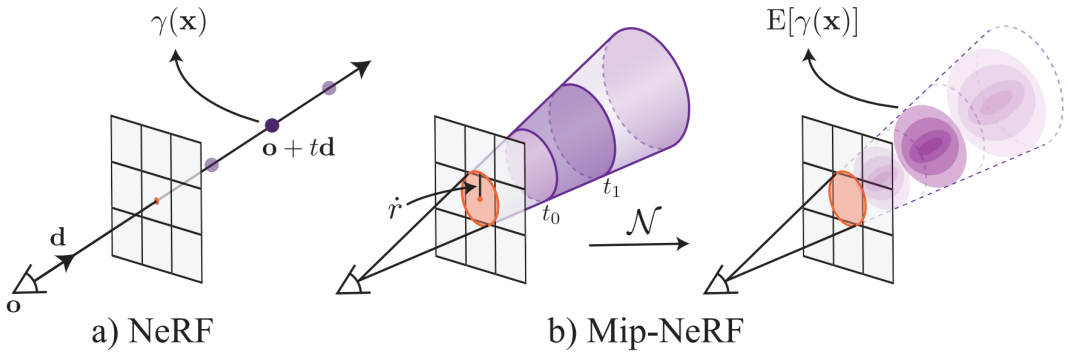


Figure 47.: NeRF (a) employs point sampling along rays traced from the camera's center of projection through each pixel. These points are then encoded using a PE)(γ), generating the corresponding feature $\gamma(x)$. In contrast, Mip-NeRF (b) operates by reasoning about the $3D$ conical frustum associated with a camera pixel. These conical frustums are featurized using an IPE method. The IPE approximates the frustum with a multivariate Gaussian and computes the closed-form integral $E[\gamma(x)]$ over the positional encodings of the coordinates within the Gaussian Barron et al. (2021).

In summary,mip-NeRF renders anti-aliased conical frustums instead of rays, reducing aliasing artifacts and improving the ability to represent fine details. It demonstrates a rendering speed that is 7% faster and a model size that is half that of NeRF. Moreover, mip-NeRF achieves a 17% reduction in average error rates on the dataset used with NeRF and a significant 60% reduction on a challenging multiscale variant of the dataset Barron et al. (2021).

A.2. Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields

NeRF also struggles to produce high-quality renderings for large unbounded scenes, where the camera can point in any direction and content can exist at any distance. In this context, conventional NeRF-like models often face challenges such as the generation of blurred or low-resolution renderings (resulting from the imbalance in detail and scale between near and far objects), slow training times, and potential artefacts arising from the inherent ambiguity of reconstructing a large scene from a limited set of images. To address these challenges, the authors Barron et al. (2022) propose an extension of mip-NeRF that uses nonlinear scene parameterisation, online distillation, and a novel distortion-based regulariser. This model "is called "mip-NeRF 360" because it targets scenes in which the camera rotates 360 degrees around a point Barron et al. (2022).

The authors Barron et al. address these issues by introducing several key enhancements to mip-NeRF. Firstly, they employ a nonlinear scene parameterization to handle the representation of unbounded scenes, as mip-NeRF inherently requires a bounded domain. This is achieved by wrapping the Gaussian distributions used in mip-NeRF into a non-Euclidean space using a technique reminiscent of an extended Kalman filter. This allows for the representation of scenes that extend indefinitely and will be explained in detail. Furthermore, the authors tackle the challenge of handling highly detailed large scenes. Simply increasing the size of the neural network underlying NeRF leads to slow training times. To address this, they employ a two-stage optimization process. They train a small proposal MLP to approximate the geometry predicted by a larger NeRF MLP. This approach enables faster training while still capturing the intricate details present in the scene. Lastly, the inherent ambiguity of 3D reconstruction in larger scenes often results in artifacts. To mitigate this, the authors introduce a novel distortion-based regularizer specifically designed for mip-NeRF ray intervals. This regularizer helps improve the overall quality of the rendered images by addressing the ambiguity and reducing artifacts. By incorporating these enhancements, mip-NeRF 360 offers improved capabilities for rendering large unbounded scenes, addressing issues related to representation, training efficiency, and artifact reduction Barron et al. (2022).

In mip-NeRF, the parameterization of unbounded 360-degree scenes in an infinite Euclidean space presents a challenge of imposing a bounded domain on the 3D scene coordinates. To address this, a Kalman-like approach is employed, utilizing Gaussians for parameterization. The scene parameterization is visualized in Figure 48. The contract operator, represented by the function contract$(\cdot)$, maps the coordinates onto a ball of radius 2. Points within a radius of 1 remain unaffected, while points outside this range are contracted towards the origin. Mathematically, the contract$(\cdot)$ operation can be defined as

$$\text{contract}(x) = \begin{cases} x, & ||x|| \leq 1 \\ \left(2 - \frac{1}{||x||}\right)\left(\frac{x}{||x||}\right), & ||x|| > 1. \end{cases} \tag{A.1}$$

This contract$(\cdot)$ operation is applied to the Gaussians of mip-NeRF in the Euclidean 3D space, resembling the behavior of a Kalman filter. The resulting contracted Gaussians, represented by red ellipses in the visualization, ensure that their centers lie within a ball of radius 2. By combining the contract$(\cdot)$ design with linearly spaced ray intervals based on disparity, rays cast from a camera positioned at the origin of the scene exhibit equidistant intervals within the orange region, as demonstrated in 48. This parameterization technique enables the effective representation of unbounded scenes while maintaining a bounded domain for the 3D coordinates in mip-NeRF Barron et al. (2022).

For further details and in-depth explanations of the efficiency considerations and the mitigation of ambiguity in mip-NeRF 360, please refer to the original paper Barron et al. (2022).

A.3. Zip-NeRF: Combining Grid-Based Techniques and mip-NeRF 360

NeRF training can benefit from the use of grid-based representations in mapping spatial coordinates to colors and volumetric density. However, these grid-based approaches often lack an explicit understanding of scale, leading to aliasing issues such as jaggies or missing scene content. To address this challenge, the concept of anti-aliasing was previously introduced in mip-NeRF 360, which operates by reasoning about sub-volumes along a cone instead of individual points along a ray. However, this approach is not natively compatible with current grid-based techniques. In this paper, the authors Barron et al. present Zip-NeRF, a novel technique that combines the benefits of mip-NeRF 360 and grid-based models like Instant NGP. Zip-NeRF leverages ideas from rendering and signal processing to achieve superior performance compared to previous methods. Specifically, Zip-NeRF achieves error rates that are 8% to 77% lower than either prior technique and offers a training speed that is 24

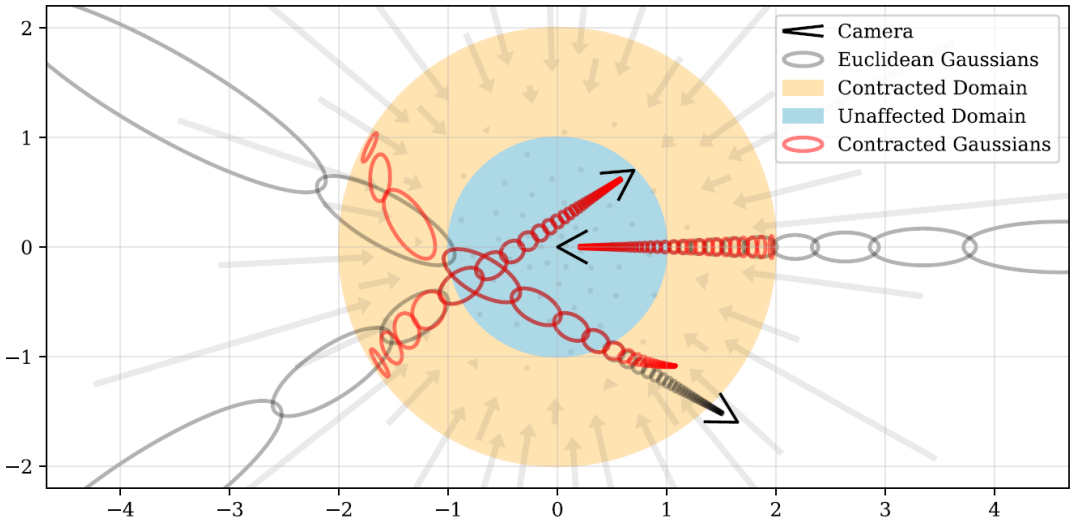


Figure 48.: $2D$ visualization of the scene parameterization. The contract$(\cdot)$ operator maps coordinates onto a ball of radius 2 (orange), where points within a radius of 1 (blue) remain unaffected. The contract$(\cdot)$ operation is applied to the Gaussians of mip-NeRF in Euclidean 3D space (gray ellipses), similar to a Kalman filter. The contracted Gaussians (red ellipses) guarantee that their centers lie within a ball of radius 2. The combination of the contract$(\cdot)$ design and the linear spacing of ray intervals based on disparity ensures equidistant intervals for rays cast from a camera at the origin of the scene, as demonstrated in this visualization Barron et al. (2022).

times faster than mip-NeRF 360. The combination between these methods yields promising results, as showcased in Video 51.

One of the key components of Zip-NeRF is the use of multisampling to approximate the average NGP feature over a conical frustum. The authors construct a 6-sample pattern that precisely matches the first and second moments of the frustum. During training, they randomly rotate and flip each pattern, while during rendering, they deterministically flip and rotate each adjacent pattern by 30 degrees as shown in figure 49. This multisampling technique enhances the accuracy of Zip-NeRF's predictions and reduces aliasing artifacts. Another important aspect addressed by Zip-NeRF is the artifact known as z-aliasing, which arises from the proposal network used for resampling points along rays in mip-NeRF 360. Z-aliasing manifests as foreground content intermittently appearing and disappearing as the camera moves towards or away from the scene. This artifact occurs when the initial set of samples from the proposal network is not dense enough to capture thin structures, resulting in missed content that cannot be recovered by subsequent rounds of sampling. To mitigate z-aliasing, Zip-NeRF introduces improvements to the proposal network supervision, resulting in a prefiltered proposal output that preserves foreground objects consistently across frames.

For more detailed information and comprehensive results regarding Zip-NeRF, we

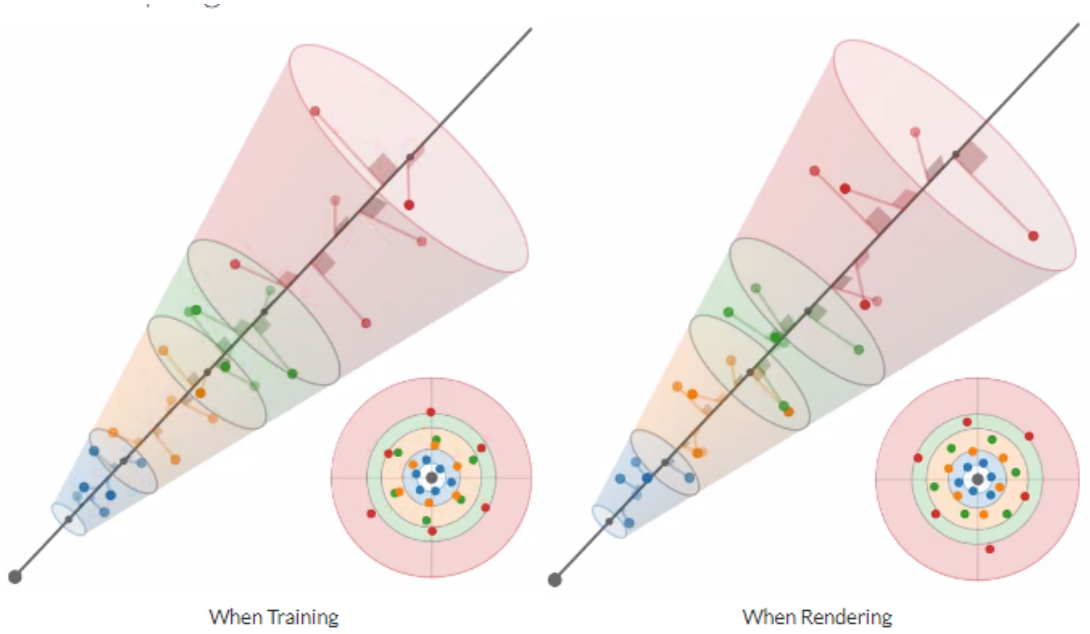


Figure 49.:]
Approximating the average NGP feature over a conical frustum using multisampling. A 6-sample pattern is constructed to precisely match the frustum's first and second moments. During training, patterns are randomly rotated and flipped (along the ray axis), while during rendering, adjacent patterns are deterministically flipped and rotated by 30 degrees Barron et al. (2023).

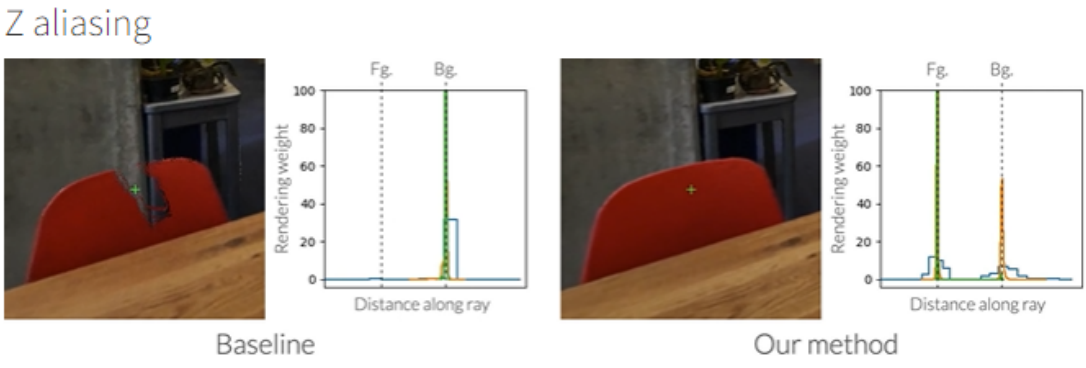


Figure 50.: The improvements to proposal network supervision result in a prefiltered proposal output that preserves the foreground object for all frames in this sequence. The plots above depict samples along a ray for three rounds of resampling (blue, orange, and green lines), with the y axis showing rendering weight (how much each interval contributes to the final rendered color), as a normalized probability density. Barron et al. (2023).

Figure 51.: Teaser of a Zip-NeRF rendered video

refer interested readers to the original paper Barron et al. (2023).

A.4. MeRF and BakedSDF

Existing representations of radiance fields suffer from either high computational demands, making real-time rendering impractical, or excessive memory requirements, limiting scalability for large scenes. Recognizing these challenges, the authors of NeRF have recently focused on overcoming these limitations. In their latest publications, they have made significant efforts to strike a balance between computational efficiency and memory usage, resulting in more practical and scalable implementations of radiance fields.

One notable contribution is the introduction of MERF, a memory-efficient Radiance Field representation that enables real-time rendering of large-scale scenes in a browser. MERF achieves this by reducing the memory consumption of previous sparse volumetric radiance fields through a combination of a sparse feature grid and high-resolution 2D feature planes. Additionally, to support large-scale unbounded scenes, the authors propose a novel contraction function that maps scene coordinates into a bounded volume while still facilitating efficient ray-box intersection. Moreover, the authors have designed a lossless procedure to bake the parameterization used during training into a model. This approach ensures that the resulting model achieves real-time rendering capabilities while preserving the photorealistic view synthesis quality characteristic of a volumetric radiance field Reiser et al. (2023).

In addition to the advancements in MERF, the authors also present another method called "BakedSDF" that addresses the reconstruction of high-quality meshes for large

unbounded real-world scenes, facilitating photorealistic novel view synthesis. The authors optimize a hybrid neural volume-surface representation designed to have well-behaved level sets that accurately correspond to the surfaces within the scene. This representation is then baked into a high-quality triangle mesh, which is augmented with a fast and straightforward view-dependent appearance model based on spherical Gaussians. The authors further optimize this baked representation to faithfully reproduce the captured viewpoints, enhancing the fidelity of the model. By leveraging accelerated polygon rasterization pipelines on commodity hardware, the approach enables real-time view synthesis. Notably, this method outperforms previous scene representations in terms of accuracy, speed, and power consumption, making it a superior choice for real-time rendering. Additionally, the high-quality meshes generated by the approach offer numerous possibilities for applications such as appearance editing and physical simulation Yariv et al. (2023).

By integrating the capabilities of BakedSDF and MERF, the authors have made substantial progress in achieving realistic and scalable rendering of radiance fields for various applications.