Continual Learning for NeRF in Scenes Obtained from Drones

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
TECHNICAL UNIVERSITY OF CRETE

A thesis submitted in fulfillment of the requirements for the undergraduate degree in Electrical and Computer Engineering.

Author: Nikolaos Angelidis
Supervisor: Vasilis Samoladas

Department of Electrical and Computer Engineering

Greece

2025

Abstract

In recent years, climate change has increased the frequency and severity of natural disasters such as earthquakes, floods, and wildfires. In the immediate aftermath of such events, fast and accurate 3D reconstructions of affected areas can support critical decision making for emergency response and recovery. Neural Radiance Fields (NeRFs) have emerged as a powerful solution for novel-view synthesis and 3D reconstruction from sparse imagery. However, traditional NeRF pipelines assume static scenes and require full retraining whenever updates occur, making them unsuitable for dynamic, real-world environments.

In this thesis, we explore continual learning approaches to extend NeRF models with the ability to incrementally update scene reconstructions without catastrophic forgetting. We evaluate two frameworks, Nerfstudio and CL-NeRF, where custom tooling is developed to support continual learning scenarios. While Nerfstudio showed promise, technical limitations led us to focus primarily on CLNeRF, which we extended and adapted to suit a variety of experimental conditions. We assess performance across a diverse set of scene changes, including object additions/removals, lighting variations, occlusions, and with a variety of types of input datasets. Our findings demonstrate that continual learning methods can significantly reduce training time, maintain reconstruction quality, and handle complex scene dynamics. These results lay the groundwork for deploying systems using NeRFs in time-sensitive scenarios such as mapping of a post-disaster scene using drone imagery.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Vasilis Samoladas, for his guidance and insightful feedback throughout this project. Moreover, my thanks also go to George Anestis, whose generous help and constructive suggestions improved the quality of my work. Finally, I am grateful to my friends and family, for their moral support and for keeping my spirits high during the toughest phases of this journey. Their encouragement and support was consequential for the completion of my work.
Thank you all for making this possible.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Motivation

Over the past decades, extreme weather events and natural disasters, have grown both more frequent and more severe, driven largely by climate change. In 2024 alone, a powerful earthquake in Japan claimed hundreds of lives and left many missing [12], catastrophic floods in the United Arab Emirates disrupted critical infrastructure such as the Dubai Metro and Dubai International Airport [13], and Greece faced one of its most devastating wildfires. Between 2017 and 2024, over 450,000 acres of forest—approximately 37% of Attica's forests were lost to wildfires[4]. In the immediate aftermath of such events, first responders and disaster-relief teams would greatly appreciate accurate and up-to-date 3D maps of the affected areas, in order to plan safe approaches, allocate resources, and coordinate evacuations.

Neural Radiance Fields (NeRFs)[6] have emerged as a powerful way to reconstruct 3D scenes from a few images. However, most NeRF pipelines assume a static scene and must be re-trained from scratch whenever the environment changes, making them impractical in these time sensitive scenarios. In this work, we utilize continual learning pipelines that preserve a "before" NeRF model of a scene. When new imagery is captured following scene changes, the existing model is incrementally updated rather than retrained from scratch. This approach yields improvements in both time training efficiency and reconstruction quality.

1.2 Our results

We conducted our experiments using two frameworks, Nerfstudio [11] and CLNeRF [15]. Aspects of the Nerfstudio framework were modified in order to enable continual learning functionality. However, due to persistent limitations, not much experimentation was conducted with the Nerfstudio framework and we ultimately transitioned to the more modular and flexible CLNeRF framework. Building upon prior work, we adapted and extended the CLNeRF methodology to suit our custom scenarios. These approaches not only accelerated 3D reconstruction compared to standard NeRF training, but also preserved the historical context of the scene and enhanced the visual quality of unchanged regions.

Our experiments covered a diverse set of scenarios designed to thoroughly evaluate and push the limits of both frameworks. These include object additions and removals, lighting changes, partial occlusions of the scene, and varying task volumes to examine the performance of the algorithms between different types of input datasets. Our results show that, both continual learning frameworks can consistently integrate new data into the scene with minimal degradation of previously learned content and overall the practicality of continual learning in dynamic 3D environments, highlighting robustness across complex update scenarios.

Motivated by the potential applications of continual learning in post-disaster scenarios, we aimed to adapt our methods to work with drone footage. Such capabilities could prove valuable in situations where rapid and robust updates to 3D scene reconstructions are needed. However, due to the absence of corresponding ”after” footage for our drone data, we instead created custom scenarios to simulate scene changes and extensively test the limits of the frameworks.

Chapter 2

Related Work

Reconstructing 3D scenes from images has long been studied in computer vision and graphics. In this chapter, we first review classical multi-view stereo (MVS) and Structure-from-Motion (SfM) approaches, which serve as foundational techniques for recovering geometry and camera motion from image sequences. We then examine the role of neural networks and how they have enhanced these traditional pipelines, enabling more robust and accurate reconstructions. This is done through the recent breakthrough of Neural Radiance Fields (NeRFs) and extend to Continual Learning NeRF models that allow for incrementally adapting to scene changes without retraining from scratch. This progression from traditional photogrammetry to neural rendering highlights the evolution of 3D reconstruction methods and sets the stage for our contributions.

2.1 Classical 3D Reconstruction Techniques

In this section, we examine how the integration of Structure-from-Motion (SfM) [9] and Multi-View Stereo (MVS) [10] have become a foundational technique in the photogrammetry and computer-vision field for reconstructing three-dimensional scenes from images. We then describe how our project leverages COLMAP, a state-of-the-art implementation of these techniques, to produce a fully automated, end-to-end reconstruction pipeline that serves as input for Neural Radiance Fields (NeRF).

2.1.1 Structure-from-Motion (SfM)

Structure-from-Motion (SfM) refers to the process of jointly estimating camera parameters (extrinsics and intrinsics) and a sparse 3D point cloud from a set of overlapping images of a scene. Firstly, feature extraction is executed, where the SfM tool detects local keypoints in each image and computes appearance descriptors that match across views despite differences in viewpoint, scale, and illumination. Secondly, feature matching compares these descriptors across image pairs and identifies correspondences among these images. Finally, bundle adjustment is applied to refine all camera extrinsics, intrinsics and 3D point positions by opting to minimize the reprojection error. This is done by projecting scene points to the image space and with the use of a loss function clear away any outliers. For more information on SfM refer to [9].

2.1.2 Multi-View Stereo (MVS)

Having obtained the calibrated camera poses and a sparse point cloud from Structure-from-Motion (SfM), the Multi-View Stereo (MVS) algorithm can estimate the depth and geometry, creating a 3D representation of the scene from the captured images. In classical MVS, a per-pixel depth hypothesis is generated for each image by sampling a range of depths along the viewing ray and projecting small image patches into neighboring views to measure photometric consistency. By iteratively refining these depth estimates, the algorithm converges toward accurate depth maps even in textured regions. Once reliable depth maps are obtained, points whose depth and normal estimates disagree across several views are discarded, while inlier points are merged to produce a high-density point cloud or mesh. At this point this system has produced a detailed 3D reconstruction of the captured scene that can be used for various applications, such as 3D mapping, virtual reality, and robotics. The performance of the MVS algorithm can significantly be improved by leveraging neural networks to learn depth estimation and 3D reconstruction from large datasets. These methods are able to handle complex scenes and texture-less regions more effectively than traditional approaches. For more information on MVS refer to [10].

2.1.3 COLMAP

In our work, the execution of the complete Structure-from-Motion (SfM) and Multi-View Stereo (MVS) pipeline is done by an open-source reconstruction framework called COLMAP. COLMAP supports image collections

and stores all their intermediate data, such as detected features, matches, camera parameters, and sparse 3D points, in a SQLite database, allowing for efficient querying and iterative refinement at each stage. Once the sparse model is built from Structure-from-Motion, COLMAP builds a high-density point cloud through fusing depth maps that are generated by a Patch Match based stereo algorithm. COLMAP offers several practical advantages that make it well-suited for data preparation in 3D reconstruction workflows. Its GPU-accelerated SIFT feature extraction and matching enable efficient processing, while adaptive local and global bundle adjustment algorithms enhance the reconstruction's accuracy. These capabilities collectively make COLMAP an ideal choice for generating fast, high-quality, and consistent reconstructions to pass as input into our neural network.

2.2 Neural Networks

In this section, we will briefly explain the fundamental concepts of artificial neural networks, covering the basic model (weights, biases), the role of layers, and how a network learns via forward propagation, loss computation, and backpropagation. Stating the importance of nonlinear activation functions, how training proceeds through gradient-based optimization. Next, we will focus on the specifications of Multi-Layer Perceptrons (MLPs), a modern feedforward neural network consisting of fully connected neurons with nonlinear activation functions, organized in layers, notable for being able to distinguish data that is not linearly separable. We explicitly explain MLPs here because they form the core function-approximators used in Neural Radiance Fields (NeRFs), and understanding their structure and training dynamics is essential for grasping how NeRFs model 3D scenes.

2.2.1 What is a Neural Network

Neurals Networks (NN) are computational models inspired by the human brain's network of neurons. A Neural Network is composed of layers of connected **neurons**, each neuron receives inputs from other neurons, processes them, and transmits its output to subsequent neurons. This information is transmitted between neurons with connections that are regulated attributes called weights and biases. The performance of the neural network depends on the refinement of these attributes in order to gradually improve. Each neural networks consists of three types of layers. The input layer, receives raw data with each input neuron representing one feature. Next up, the hidden layers perform core computations where there might optionally be one

or several layers of this type on the network. Completely, the output layer is the final layer of the network that produces the networks prediction based on the processed information. This layered structure allows neural networks to model complex functions that map the input data to output predictions. In essence, a neural network is a function approximator, through adjusting its weights, it can learn to approximate relationships in data, enabling tasks like classification, regression, and pattern recognition.

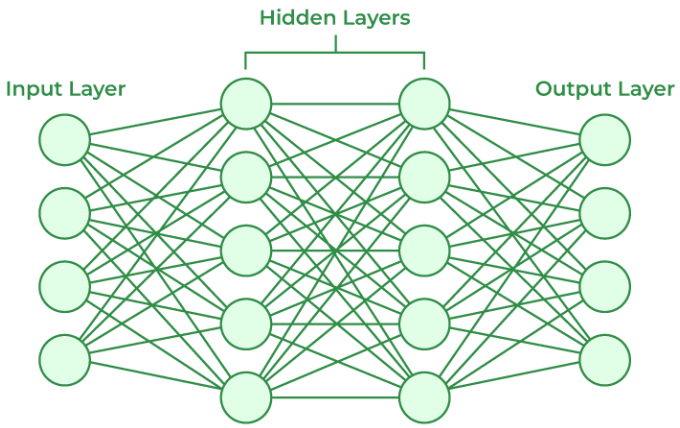


Figure 2.1: Basic Neural Network Structure (source: downloaded from [3]).

2.2.2 How they work

Neural networks are capable of learning and identifying patterns directly from data without predefined rules. This is done with constant updating during training of crucial attributes of the neurons that compose the neural network, weights and biases. During each training step, the network first computes a loss, this value measures the difference between the predicted output and the ground truth. Common choices used as **loss functions** are the mean squared error for regression tasks or cross-entropy loss for classification tasks. It then performs **backpropagation**, calculating the gradient of the loss with respect to every weight and bias to determine how each parameter contributed to the error. Finally, the weights and biases are updated using an optimization algorithm like **stochastic gradient descent** (SGD). The weights are adjusted in the opposite direction of the gradient to minimize the loss, with the size adjustment being determined by the **learning rate**.

Neural networks tend to employ a hierarchical feature learning structure. Early layers specialize in detecting simple, low-level features such as edges or corners, while deeper layers progressively capture more abstract and complex patterns, like object components or semantic groupings. This distributed and layered processing enables neural networks to approximate highly nonlinear and complex functions, provided they have sufficient capacity and access to diverse training data.

2.2.3 Training and Evaluation

During training, the goal of the neural network is to minimize the loss at each training step. In practice, training a neural network involves processing data from the original input dataset in **mini-batches** over multiple **epochs**, a complete pass through the training dataset. At every epoch the neural networks achieves minimization of loss by backpropagation combined with optimization algorithms. Backpropagation computes the gradient of the loss with respect to each weight in the network by propagating the error backward from the output layer to earlier layers. The weights are then adjusted slightly in the direction that reduces the loss (the gradient descent step) until the network converges or the loss stops improving.

A common issue that neural network models face is overfitting, this is where the model memorizes training examples rather than generalizing, this means that although the model's performance on the training dataset may be excellent, its accuracy on new, unseen data declines. So it is of high importance to select appropriate **hyperparameters**, such as learning rate, batch size, number of hidden layers, number of epochs and activation functions for the composition of a solid model, as these settings directly shape the training dynamics and convergence behavior of the model. The right choices can mean the difference between a well-trained model and one that underperforms or even fails to converge. Finally, to evaluate the network's performance, unseen validation and test splits from the original dataset are reserved to measure performance metrics such as loss, accuracy and also training time to ensure the model generalizes beyond its training data.

2.2.4 Multi-Layer Perceptron (MLP)

A Multi-Layer Perceptron (MLP) is one of the fundamental architectures of neural networks. It is essentially a feed-forward neural network with one or more hidden layers of fully-connected neurons (each neuron in one

layer is connected to every neuron in the next layer). The network being "multi-layer" indicates that it has at least one hidden layer between the input and output. The "single-layer" perceptron (with no hidden layer) can only solve linearly separable problems, but a multilayer perceptron with sufficient neurons and at least one hidden layer can approximate more complex functions. MLPs use nonlinear activation functions in their hidden layers so that the network can learn nonlinear relationships. This use of nonlinear activation functions is essential when working with images. Real world images are rich with complex structures, edges, shadows, lighting variations, occlusions, and object boundaries, all of which involve nonlinear relationships between pixels and scene attributes. By incorporating nonlinearity at each layer, MLPs are able to learn and model these intricate relationships effectively. In the context of 3D scene representations, MLPs are often used as the function approximators for implicit models (as we will see with NeRF), due to their ability to represent continuous functions in space.

2.3 Neural Radiance Fields (NeRFs)

In recent years, Neural Radiance Fields (NeRFs) [6] have emerged as one of the most popular techniques for novel-view synthesis from only a sparse set of input images of a scene. In the following section, we provide a detailed overview of the network's architecture, underlying principles and explain their innovative implementation.

2.3.1 Network Architecture

A Neural Radiance Field (NeRF) is a fully-connected neural network, specifically, a multi-layer perceptron (MLP) that maps a 5D input, that consists of a spatial coordinate (x, y, z) and a viewing direction (θ, ϕ), to a density σ and a color $c = (R, G, B)$. The network begins by applying a positional encoding, a technique used to map low dimensional input coordinates into a higher dimensional space using a series of Fourier basis functions, enabling the MLP to represent high frequency variations in the scene, $\gamma(x)$ ($\gamma(\cdot)$, positional encoding function) to the input spatial coordinate x, then processes this through eight fully-connected layers of 256 ReLU units each, with a skip connection that concatenates $\gamma(x)$ into the input of the fifth layer. From the fifth layer's activations, a linear output branch predicts the volume density σ and a 256 dimensional feature vector. This feature vector is concatenated with the positional encoding $\gamma(d)$ of the viewing direction d and passed through one more 128 unit ReLU layer. Finally, a sigmoid-activated linear

layer maps this to the RGB color emitted at x in direction d. During the training of the network, at each step a batch of 4096 rays is processed to balance gradient stability and memory efficiency. All network parameters are optimized using the Adam optimizer with an initial learning rate of $5 \cdot 10^4$, which decays exponentially to $5 \cdot 10^5$ over the course of training.

The volumetric density σ is predicted solely from the spatial coordinate $\mathbf{x} = (x, y, z)$, and then its output is concatenated with the viewing direction $\mathbf{d} = (\theta, \phi)$ and fed to the second part of the network to predict the color $\mathbf{c} = (R, G, B)$. This is the case because, the presence of an object on the scene at a given point does not depend on the viewpoint but only on the location. In contrast, the emitted color c is treated as a function of both the spatial coordinate and the viewing direction, thereby capturing view-dependent appearance effects for different viewpoints.

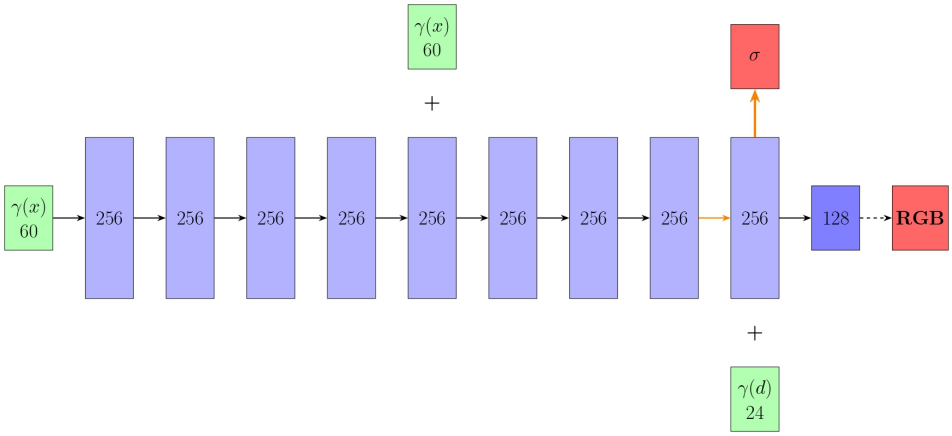


Figure 2.2: NeRF MLP architecture: positional encoding, eight 256-unit ReLU layers with a skip connection, density branch, directional encoding, a 128-unit ReLU layer, and sigmoid-activated RGB output. Inspired by Figure 7 from [6].

2.3.2 Hierarchical Volume Sampling

During the training process, there are two networks that are being optimized simultaneously, a **coarse** and **fine** network pair in order to learn both a rough and a high-fidelity representation of the scene. First, the "coarse" network makes N_c predictions at uniformly spaced points along each ray. It then uses these predictions to compute volumetric weights and render a rough

pixel color, in order to output a refined sampling distribution for the ”fine” network to place additional N_f samples where the scene's geometry and appearance change most. From this distribution using inverse transform sampling, our ”fine” network is evaluated at the union of the first and second set of samples, and computes the final rendered color of the ray. Both the coarse and fine renderings incur mean-squared-error losses against ground-truth pixels, encouraging the coarse network to guide sampling effectively and the fine network to produce highly detailed reconstructions.

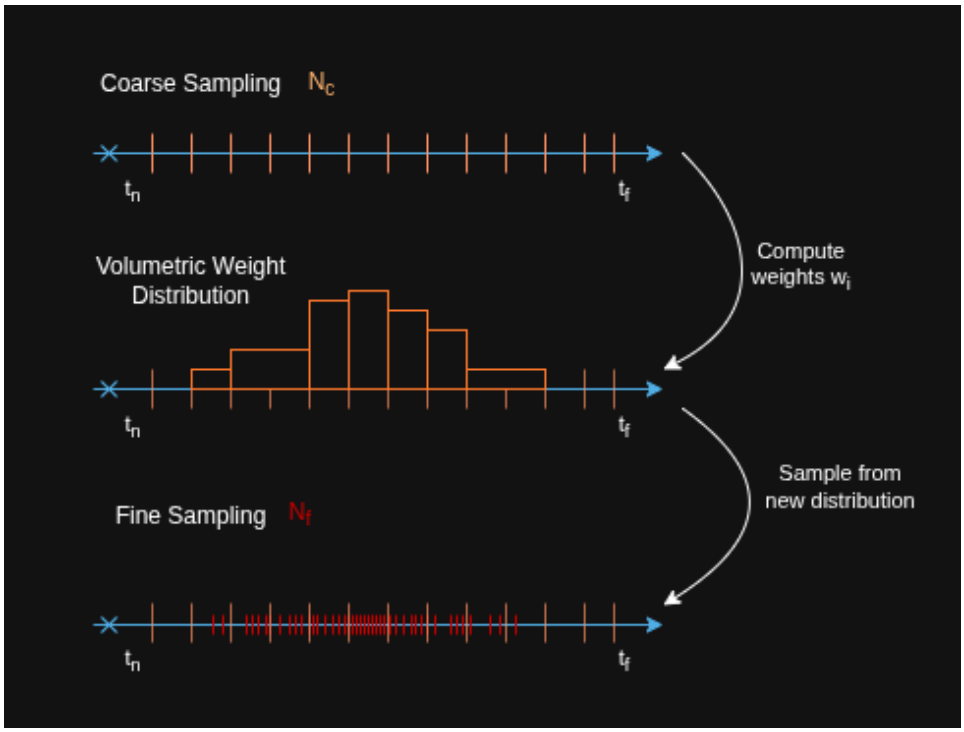


Figure 2.3: Hierarchical Sampling Process: the coarse network samples N_c points uniformly along the entire ray to compute a volumetric weight distribution, then the fine network draws N_f additional samples from this distribution, focusing more where the geometry and appearance of the scene changes the most, in order to produce the final, high fidelity pixel color. Inspired by [2].

2.3.3 Volumetric Rendering

NeRF synthesizes novel views by applying volumetric rendering to a learned radiance field. The radiance field is defined by the neural network's output, volume density σ and RGB color $\mathbf{c}$. To render an image, rays are cast from a given camera pose into the scene. Along each ray, the network samples multiple 3D points, and the color contributions from these points are accumulated according to their densities using classical volume rendering equations [5]. This process determines the final pixel color, where contributions from occluded or dense regions are appropriately attenuated.

How it works

A ray is represented as: $\mathbf{r}(t) = o + t\mathbf{d}$ (o = ray origin, d = direction, t being distance along the ray), the expected color $C(r)$ is given by an integral of the form

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt$$

where t_n and t_f are the ray's near and far bounds, where the ray enters and exits the scene respectively. $T(t) = exp(-\int_{t_n}^{t} \sigma(\mathbf{r}(t))ds)$ is the accumulated transmittance, the probability that the ray travels from t_n to t without hitting any opaque material. Thus, the integral is approximated by sampling a set of discrete points along the ray (as discussed in 2.3.2). In implementation, for sample i at depth t_i with density σ_i and color c_i, the contribution to the pixel color is computed as:

$$T_i = exp\Big(-\sum_{j=1}^{i-1} \sigma_j \delta_i\Big), \quad C(r) \approx \sum_{i=1}^{N} T_i \Big(1 - exp(-\sigma_i \delta_i)\Big) c_i.$$

where δ_i is the distance between sample i and $i-1$ along the ray. The term $\alpha_i = (1 - \exp(-\sigma_i \delta_i))$ can be thought of as the opacity of the segment at i, and T_i is the transmittance up to that segment as mentioned before. This formulation is often called alpha compositing and it accumulates colors from front to back until the ray is fully opaque. This way occlusion is naturally handled, meaning, if an object has high density, $T(t)$ beyond it drops near zero, so little to no color from behind it is contributed to the final rendered pixel color for the current ray.

This way the NeRF model avoids having to explicitly classify surfaces or determine a single geometry and semi-transparent or thin structures can be represented as needed by appropriate σ values. It also inherently handles

view-dependent effects (with $c(r(t), d)$ depending on direction d). This is important for modeling shiny or reflective materials, signifying that the NeRF model can learn to effectively emit different color depending on viewpoint. In summary, volumetric rendering in NeRF turns the neural field into actual images, bridging the gap between the continuous scene representation and pixel observations.

2.3.4 Training Objective and Loss

Training a NeRF model means adjusting the network's parameters so that the rendered images match as close as possible the observed images from the training dataset. Therefore, in order to execute the refinement of the network's parameters the training loss is defined on rendered pixels vs. ground truth pixels. The loss function is a straightforward pixel-wise Mean Squared Error (MSE). For each ray (pixel) in a training image, the color $C_{\text{pred}}(r)$ predicted by the NeRF (via volumetric rendering as in 2.3.3) is compared to the ground truth pixel color $C_{\text{gt}}(r)$. The error $|C_{\text{pred}} - C_{\text{gt}}|^2$ is computed, and the loss is the average of these over all sampled rays in all training images.

Formally, $L = \sum_{r \in \mathcal{R}} |C_{\text{pred}}(r) - C_{\text{gt}}(r)|^2$, where $\mathcal{R}$ is the set of all rays. This simple photometric reconstruction loss drives the network to correctly predict densities and colors such that rendered views match the real images. Both the coarse and fine networks (2.3.2) in NeRF are trained simultaneously using this loss. In addition to the fine model's output, the coarse model's rendered output is also compared to the true pixel color but with a lower weight. This means the coarse network also learns to approximate the scene, which helps guide the fine network as discussed previously. The combined loss encourages the scene representation to explain all views consistently.

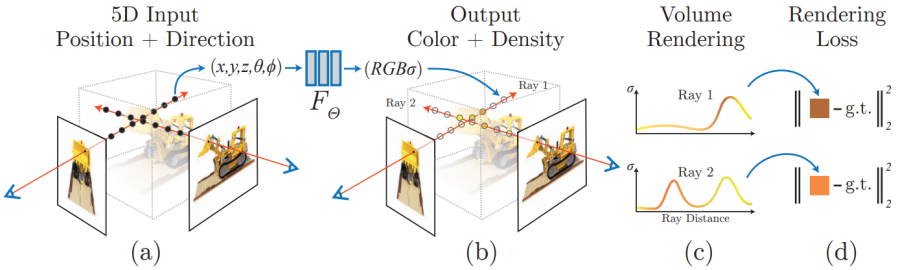


Figure 2.4: Overview of the Neural Radiance Fields (NeRF) Rendering Pipeline: (a) Rays are cast from the camera into the 3D scene, each sampled point along a ray is represented by a 5D input — 3D position (x, y, z) and 2D viewing direction (θ, ϕ). (b) These inputs are processed by a neural network, which outputs RGB color and volume density (σ) for each sampled point. (c) Volume rendering is applied to accumulate colors and densities along each ray to produce final pixel colors. (d) The rendered output is compared to ground truth images using the Mean Squared Error (MSE). Figure 2 from [6].

2.4 Continual Learning NeRF

While traditional NeRF models can achieve impressive reconstructions, they typically assume a fixed set of training images captured from a static scene. Continual learning for Neural Radiance Fields (NeRFs) refers to approaches that enable NeRFs to be trained incrementally as new data comes in, without forgetting the previously learned scene content. This is of higher importance for scenarios like mapping, where a robot or drone may continually capture new images of an environment as it changes over time. In our case study, where the scene has been deformed due to natural disasters, training from scratch on the combined data is not optimal and just updating a pre trained scene would prove beneficial.

Catastrophic forgetting is a major problem where the network forgets previously learned aspects of the scene when being trained on new data. Several recent works tackle this challenge with various approaches. For example, memory-efficient incremental learning for NeRF (MEIL-NeRF) [1] uses a generative replay strategy to update a NeRF as new images arrive, focusing on minimizing memory growth. Another line of work, CL-NeRF with task-specific adaptors [14], introduces lightweight model adaptations and knowledge distillation to retain old knowledge. There are also instant

continual learning frameworks for NeRF that leverage hybrid representations to quickly integrate new scene data while using replay-based training to prevent forgetting [8].

Within this landscape of research, CLNeRF (Continual Learning NeRF) [15] is a method that combines a replay-based continual learning scheme with a high-performance NeRF architecture. CLNeRF introduces generative replay into NeRF training and leverages the Instant Neural Graphics Primitives (NGP) framework [7]. In essence, CLNeRF can handle complex scene changes over time, and achieves performance on par with a conventional NeRF trained on all data at once, all while using a compact model and without storing large amounts of past data. In the following, we detail the network architecture of CLNeRF and explain how its training scheme works to mitigate catastrophic forgetting.

2.4.1 Network Architecture

CLNeRF adopts the Instant NGP architecture as its backbone, which is an optimized NeRF implementation using a multi-resolution hash grid encoding and small MLP decoders. This backbone provides both efficient training and rendering, forming a strong base for the model's training. On top of this, CLNeRF expands the network by adding trainable embedding vectors that serve as learned context parameters for each time step of the scene. In particular, two types of embeddings are used. An appearance embedding e_a which is intended to capture changes in scene appearance (lighting, weather, color variations across time) and a geometry embedding e_g that handles structural changes in the scene (objects added or removed between scans).

These embeddings are inserted into the NGP network's MLP decoder modules to condition the radiance field on the specific time of captured data. Concretely, after the hash-grid encoder produces a spatial feature vector f for a sampled 3D point, the density decoder takes f along with the geometry code e_g to predict the volume density σ at that point, and the color decoder takes f with the viewing direction d and appearance code e_a to predict the RGB color c. By design, e_g influences which parts of space are filled or empty, enabling the network to represent different geometry configurations, whereas e_a influences only the color output, accounting for illumination or texture changes without altering geometry. Another significant architectural consideration is handling transient objects or changes that should not be permanently learned. CLNeRF makes use of segmentation masks to exclude transient or moving objects, such as a person walking through a scene, from the training loss. This prevents the network from overfitting to one-time

anomalies in a particular scan.

In summary, the architecture of CLNeRF is a NeRF model conditioned on timestep specific embeddings, enabling it to represent a evolving scene without growing the model significantly and without separate per version networks.

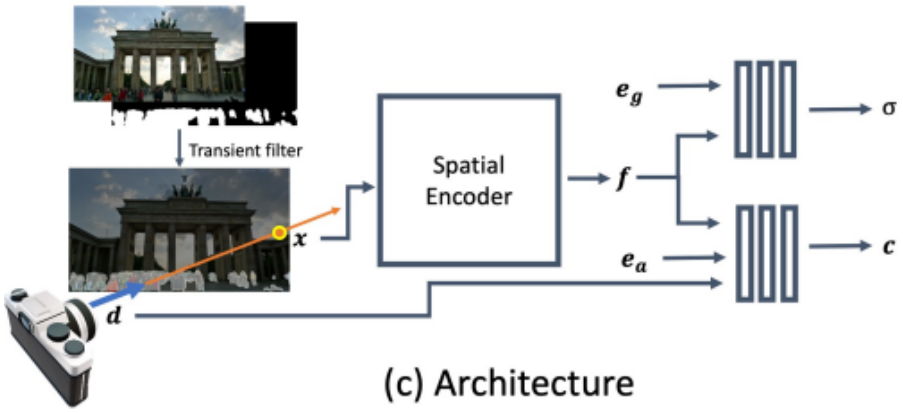


Figure 2.5: CLNeRF Network Architecture: To handle transient or of low importance moving object, segmentation masks are used to filter transient objects. Apply appearance embeddings e_a and geometry embeddings e_g to handle scene changes at different time steps. Part (c) of Figure 2 from [15].

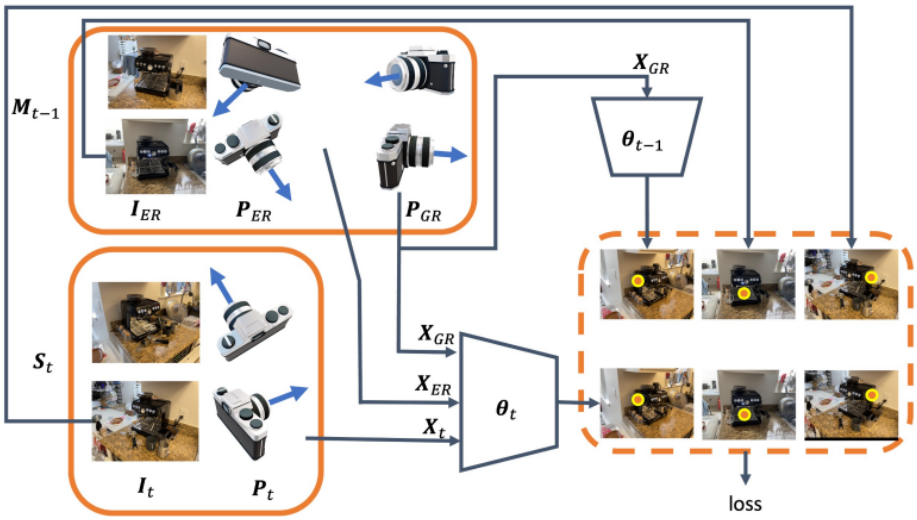
2.4.2 How they work

CLNeRF is trained sequentially on a stream of data corresponding to different time steps of a scene, for example, consecutive scans of a site captured at different times that can mean change in geometry and lighting.

Let Θ_{t-1} be the model after learning up to time $t-1$ (including its learned embeddings for past scans), and S_t be the new set of images at time t. The goal is to learn the new scan's content from S_t while retaining the ability to render all previous scans with high fidelity. This is implemented by a replay-based strategy, combination of experience replay and generative replay. At each training iteration on S_t, the system assembles a mixed batch of rays drawn from the **new** images, a small buffer of **past** images and **synthetic** replay data generated from the previous model Θ_{t-1}. The rays of new images use the actual ground-truth pixel colors as supervision signals. For generative replay, however, the training uses rendered color values from

the old model Θ_{t-1} that were seen in earlier scans, and uses those as target observations for the current model Θ_t.

During training, CLNeRF optimizes a composite loss over the batch that includes both new and replay data. In effect, the objective is to minimize the error $L = \sum_X |C(X) - \hat{C}(X \mid \Theta_t)|^2$ across all sampled rays, where: $X \in X_{\text{new}} \cup X_{\text{ER}} \cup X_{\text{GR}}$, $C(X)$ is either the ground truth color for new images or the pseudo ground-truth for generative replay rays colors and $\hat{C}(X|\Theta_t)$ is the color rendered by the current model. After sufficient training on S_t with this replay-enhanced objective, the model parameters and the new embeddings for time t are finalized as Θ_t.



(a) Model update

Figure 2.6: Overview of the update process of the CLNeRF model. Part (a) of Figure 2 from [15].

The replay buffer is also updated, if a fixed small memory budget for images is allowed, CLNeRF will retain at most **K** past images using reservoir sampling (randomized algorithm used to maintain a representative sample of K items from a data stream) to refresh the buffer with some of the latest observations. In memory constrained scenarios, where no past images are stored at all ($K = 0$), the system relies purely on generative replay using stored camera parameters.

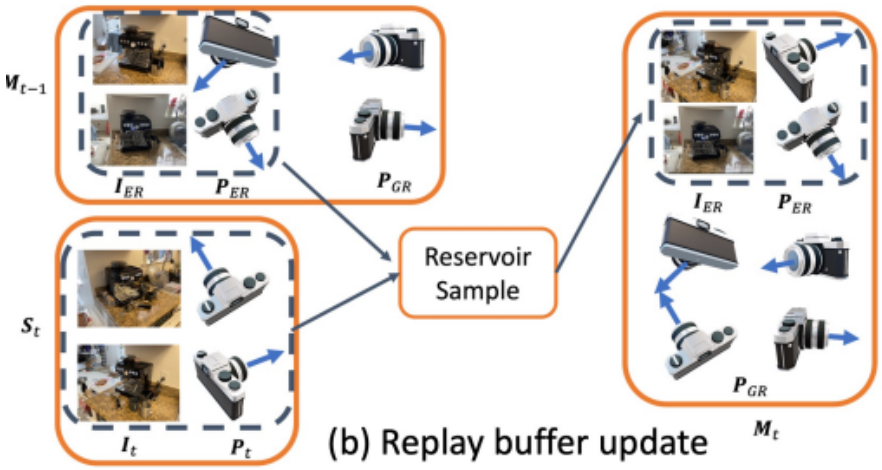


Figure 2.7: Overview of the replay buffer update mechanism in CLNeRF. At time step t, the current scan S_t and the memory buffer from the previous step M_{t1} are combined via reservoir sampling. The resulting updated memory M_t includes a subset of prior data and newly observed samples, while maintaining a fixed memory budget. Part (b) of Figure 2 from [15].

In summary, CLNeRF's design consists of a single neural radiance field with scan-specific embeddings, trained using a combination of newly acquired real data and previously rendered replay data. This architecture effectively mitigates catastrophic forgetting by maintaining a consistent memory of past observations, allowing the model to incrementally learn new information without overwriting previously learned content. Furthermore, this design enables the continual learning pipeline to scale efficiently over time, requiring minimal memory growth. As a result, CLNeRF is well-suited for dynamic environments where frequent scene updates are expected, making it a practical solution for applications like post-disaster mapping.

Chapter 3

Methodology

In this chapter, we will provide an overview of the core tools and frameworks utilized throughout this research, detailing both the high-level pipeline and the technical specifications of each model's implementation. The end-to-end workflow of each framework is outlined, beginning with data capture and preparation, and proceeding through to model training and evaluation. First, we will present the structure and capabilities of the Nerfstudio framework, followed by a description of the modifications and extensions introduced to support the specific requirements of continual learning that were made. Then we will analyze Continual Learning NeRF (CLNeRF) model and explain its integration, data preparation specifications and training protocols in order to achieve the desired result.

3.1 Nerfstudio

Nerfstudio[11] is an open-source, modular framework designed to streamline the development, training, and evaluation of Neural Radiance Fields (NeRF) and related 3D scene representations. It provides a user-friendly, extensible pipeline that integrates all stages of a NeRF workflow, from data preprocessing to model training and interactive visualization, which played a vital role in choosing it over other NeRF training pipelines. Moreover, Nerfstudio was a valuable choice due to its convenient and easily customizable key modules that it offers like the DataManager, Viewer, Training Manager and more.

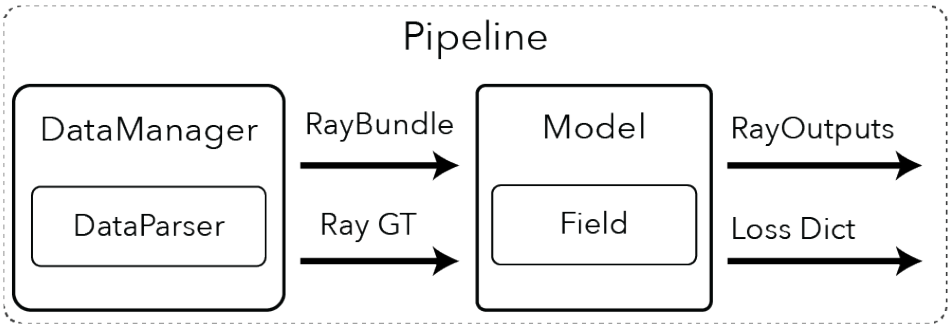


Figure 3.1: Nerfstudio Default Pipeline High-level Nerfstudio pipeline: the DataManager (via its DataParser) loads images and metadata to produce RayBundles and GT (ground-truth) rays, which the Model's neural field consumes to emit rendered RayOutputs and a Loss dictionary. (source: downloaded from [11]).

3.1.1 Nerfstudio Framework Overview

Scene Capture and Preprocessing Pipeline

To begin, obtaining input data for most scenarios can be effectively achieved by using a standard video recorded with a smartphone camera. A longer video duration typically results in a higher number of extracted frames, which in turn provides more diverse viewpoints of the scene. This increased viewpoint density enables the NeRF model to learn a more complete and geometrically consistent reconstruction. To ensure optimal training results, the captured frames should be free of motion blur and avoid too bright or too dark areas that may obscure structural details. Consistent lighting throughout the video is also recommended to minimize photometric inconsistencies that can degrade model performance. By adhering to these practices during data collection, the resulting capture of the scene provides a robust foundation for the Structure-from-Motion (SfM) system to generate the required input data for the NeRF model to be trained on.

In the case of the Nerfstudio framework, data preprocessing is streamlined with a single CLI command, the user can transform a captured video or a collection of images of a scene into a fully processed dataset. This process includes frame extraction via ffmpeg (for video input), camera pose estimation via COLMAP, and the generation of necessary metadata. The resulting output is formatted to be directly compatible with any of the supported models within the Nerfstudio ecosystem, enabling efficient and

reproducible training pipelines.

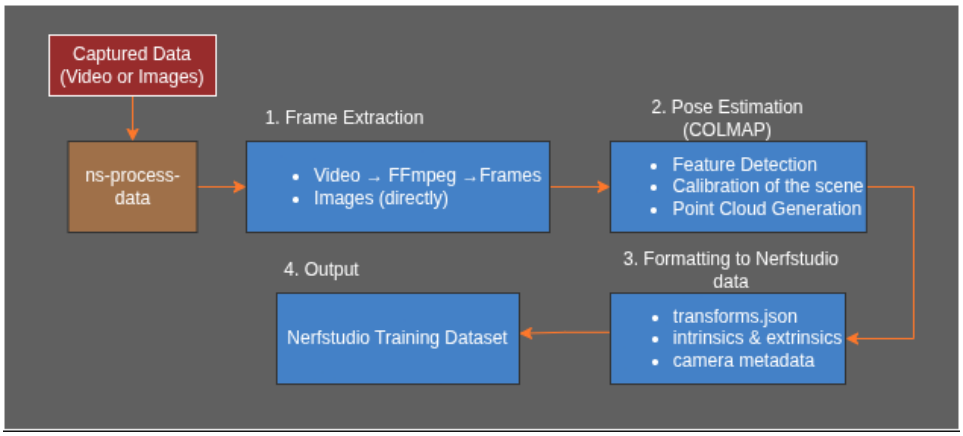


Figure 3.2: Workflow of Input Data Preprocessing on Nerfstudio.

Standard Training Workflow

Using Nerfstudio, the training process of a model is straightforward via the `ns-train` of the Command-Line Interface (CLI) tool. In order to train typical NeRF model, the **nerfacto** method is used, it implements NeRF with the many recent advancements for speedup and quality improvements. By providing the processed data directory, typically the output of the command `ns-process-data`, the framework begins the training loop, periodically saving model checkpoints that capture both optimizer states and the current weights. Simultaneously, the Nerfstudio built-in web viewer is launched, to enable optional real time visualization of the model as it is being trained where also the user can freely move around to capture never before seen viewpoints. After the training process is completed, through Nerfstudio viewer the user can render custom frames, videos and export the model in a point cloud (.ply file) or Mesh. These outputs can then be loaded into external tools for further editing or integration into other pipelines.

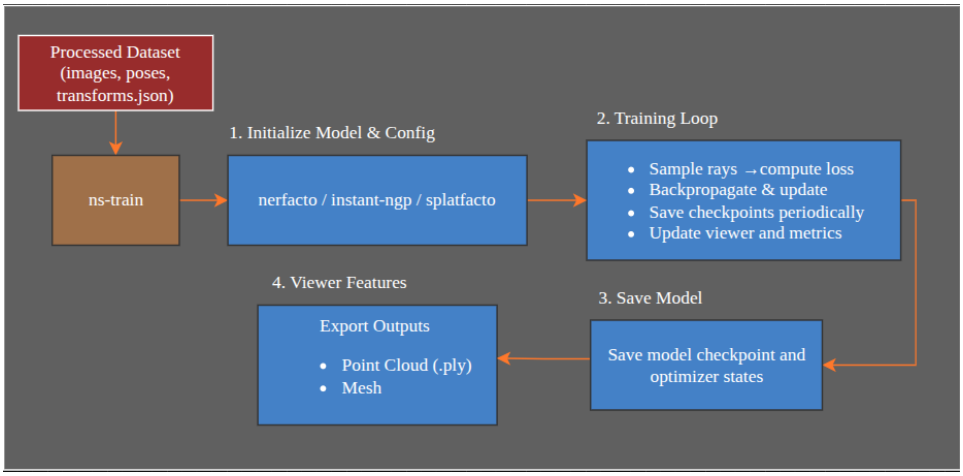


Figure 3.3: Workflow for the `ns-train` command in Nerfstudio. Starting from the processed dataset, the pipeline initializes the chosen training method and enters the training loop. Training progress can be visualized in real time and through the viewer the trained model can be exported.

3.1.2 Nerfstudio Workflow for Continual Learning

In this section, we will explain in greater detail how the Nerfstudio framework was utilized to support our continual learning experiments, outlining the complete workflow and analyze each command's specifications from data capture and preprocessing to model training and final result evaluation through the viewer.

Data Capture and Preprocessing

Data Capture. For continual learning, we acquire comprehensive coverage of each scene by capturing as many viewpoints as possible, ensuring that key subjects remain in view and under consistent lighting. In our experiments, footage was recorded in landscape orientation at 1080 p and 30 FPS. This was done for both base and changed scenes in our continual learning scenarios.

Preprocessing. We then invoke Nerfstudio's `ns-process-data` CLI, which uses `FFmpeg` to extract frames (for video inputs) and runs COLMAP for camera calibration and pose estimation. The result is a Nerfstudio compatible dataset comprising RGB images, camera intrinsics and extrinsics, and associated scene metadata, ready for training (see Figure 3.2).

In most Continual Learning scenarios, we begin with a foundation model trained on a `base` dataset that represents the scene in its original, unaltered state. This base scene serves as the reference point prior to any environmental or structural changes. Captured after modifications to the scene is then incrementally integrated through continual learning mechanisms. For the preprocessing of the `base` dataset, we employ the `ns-process-data` command in Nerfstudio to extract frames, estimate camera poses using COLMAP, and generate the structured output required for initial model training, as illustrated below.

```
$ ns-process-data video --data {DATA_PATH} --output-dir
↪  {PROCESSED_DATA_DIR}
```

Listing 1: Command for processing video input with Nerfstudio

```
$ ns-process-data images --data {DATA_PATH} --output-dir
↪  {PROCESSED_DATA_DIR}
```

Listing 2: Command for processing images input with Nerfstudio

For the generation of input data corresponding to the newly added training input, it is essential that the merging of the `base` and `continued` datasets occurs during the SfM stage of preprocessing. This ensures that all frames are aligned within a common 3D coordinate system. Since the `base` and `added` datasets are captured at different time steps and contain changes in the scene, we manually execute COLMAP commands to jointly register both the original and new frames in a unified reconstruction. Once this process is complete, the resulting COLMAP output is converted into a format expected by Nerfstudio's train command using the appropriate tooling. During training, we configure the `transforms.json` file, a file which controls which frames will be used for a specific training process, to include only the subset of images corresponding to the `continued` (added) data, thereby ensuring that the continual learning model is updated exclusively with new information while preserving the integrity of the foundation model.

Training Process

For the training of the base scene, we utilize the `ns-train` command provided by the Nerfstudio CLI. As input, we supply the preprocessed dataset generated through `ns-process-data`. The training proceeds using the `nerfacto` method, which implements NeRF with the many recent advancements for speedup and quality improvements. This results in a fully trained model that captures the geometric and appearance based properties of the original scene. Upon completion, the Nerfstudio viewer allows for interactive exploration of the reconstructed 3D space, enabling custom camera pose rendering and even video rendering from a trajectory built from a collection of custom poses. This model will serve as our foundation model that can be incrementally updated with updated scene data.

```
$ ns-train nerfacto --data {PROCESSED_DATA_DIR}
```

Listing 3: Command for training base scene with Nerfstudio

```
$ ns-viewer --load-config {TRAINED_MODEL_CONFIG.YML_FILE}
```

Listing 4: Command for opening viewer for model with Nerfstudio

Once the dataset for the `added` scene content has been processed and merged with the `base` dataset, ensuring that all images are registered within a common coordinate system, we proceed to incrementally train the existing foundation model. A key observation during this continual learning phase is that relatively few iterations are required for the model to incorporate the newly introduced changes in the scene. In our experiments, we found that approximately one-quarter of the original training steps were sufficient for the model to converge on the updated geometry and appearance, leading to significant time savings compared to retraining from scratch. We will talk more about these improvements on Section 4.1. The output of this process is again a refined 3D representation of the scene that includes the added elements. This updated model can once again be visualized through the Nerfstudio viewer, allowing for free navigation and the rendering of novel views that include the new content.

```
$ ns-train nerfacto --data {PROCESSED_DATA_DIR } --load-dir
↪  {TRAINED_MODEL_NERFSTUDIO_MODELS_FILE}
```

Listing 5: Command for incrementally training the continued model with Nerfstudio

Result Evaluation

After training the base model using the Nerfstudio framework with the `ns-train` command and the `nerfacto` training method, as illustrated in

3, we utilize the Nerfstudio viewer to inspect and interact with the resulting scene reconstruction. This is done via the `ns-viewer` command by providing the configuration file corresponding to the final model, as shown in 4.

The viewer serves as a valuable tool for qualitative inspection and result visualization. It supports rendering individual frames, generating videos along user-defined camera trajectories via keypoints, displaying estimated depth maps, and offering various interactive features for scene exploration. Examples of these capabilities are presented on Figure 3.6.

3.1.3 Example of Continual Learning Workflow on Nerfstudio

To demonstrate the continual learning workflow in practice, we use the `counter_add` experiment's dataset. In this scenario, the base scene consists of several objects arranged on a countertop. The continued scene introduces a new element (a mug) that was not present during the initial capture. In this section, we present the full end-to-end workflow, including the sequence of commands executed and the evaluation of the resulting models, to illustrate how continual learning was applied and assessed using the Nerfstudio framework.

Data Processing

To begin the initial state of our dataset, contains two video captures of the scene one before (`normal_split_base.mp4`) and one after (`normal_split_after.mp4`) the object was added to the scene.

```
# initial setup
data/counter_demonstation
|-- videos
    |-- normal_split_added.mp4
    |-- normal_split_base.mp4

# result
data/counter_demonstation
|-- merged # merged colmap
|   |-- database.db
|   |-- images
|   |-- sparse
|-- outputs
|   |-- ns_counter_demonstration # final merged dataset
|   |-- p1_ns_base               # processed base dataset
|   |-- p2_ns_added              # processed added dataset
|-- videos # original videos
    |-- normal_split_added.mp4
    |-- normal_split_base.mp4
```

Listing 6: Initial and Resulting Setup: **before** preprocessing we have just the two captured videos of the scene, **after** we have three processed outputs that contain the train ready datasets. We also have the colmap output for the merged dataset (before transforming to Nerfstudio format) since we executed it manually.

With the use of Nerfstudio `ns-process-data` command for videos, we generated the desired colmap outputs needed for training. Command usage is shown here 1.

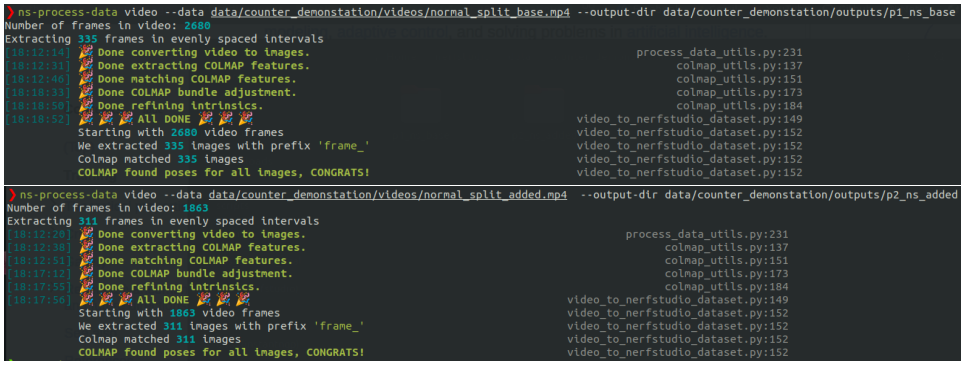


Figure 3.4: Nerfstudio `ns-process-data` command output for both `base` and `added` datasets.

Training

Next up, we train the base NeRF model using the `ns-train` command (Listing 3). Figure 3.5 illustrates the three key stages: the exact CLI invocation, the live training log output, and the final logging as the training is completed.

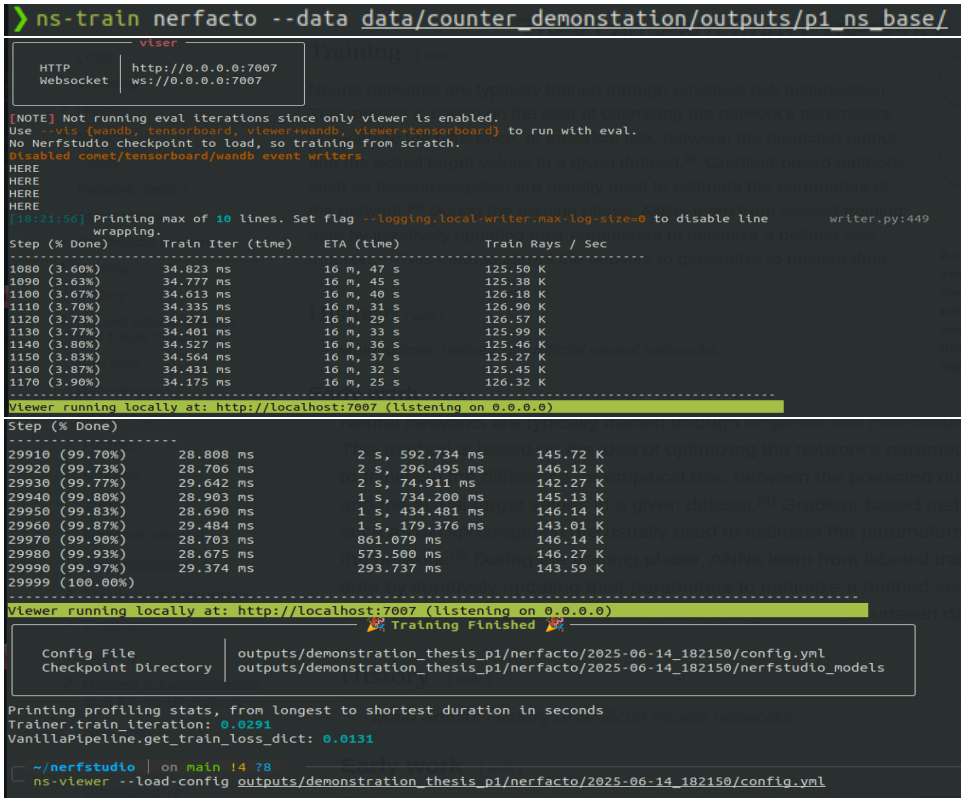


Figure 3.5: Nerfstudio `ns-train` command output for model training with the `base` dataset. **First** image displays the exact command that was executed, **second** image showing training information, **third** image shows the final output when training is completed.

Once training completes, we launch the Nerfstudio viewer, as shown in 4, to inspect the reconstructed scene.

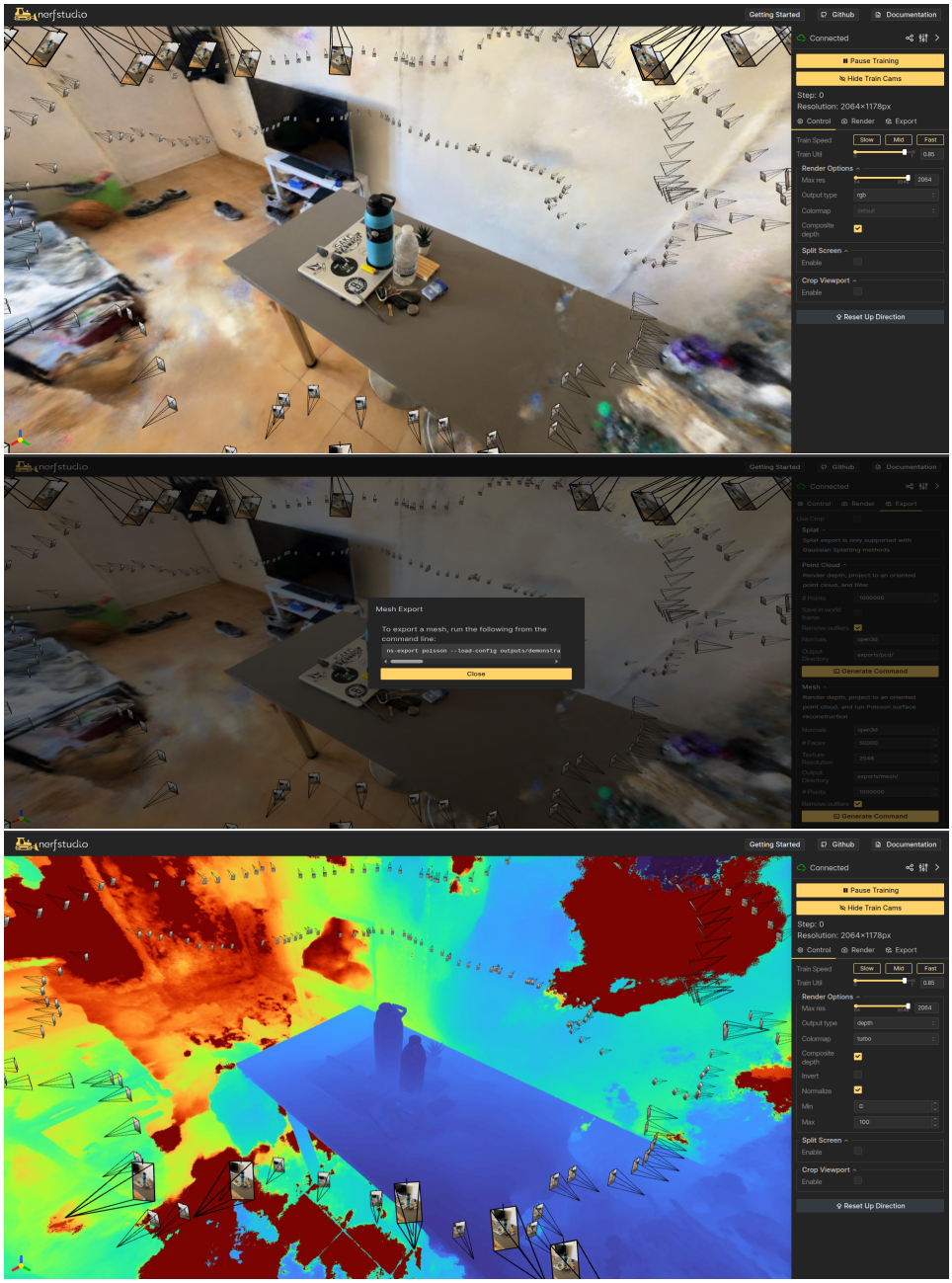


Figure 3.6: Nerfstudio viewer. **Top:** rendered RGB preview from current viewpoint. **Center:** custom export command for the trained model. **Bottom:** depth map overlay for the current viewpoint.

At this stage, in order to train the model with the added data, it is necessary to preprocess the newly captured scene together with the base scene so that all frames are aligned within a common 3D coordinate system, as previously discussed. This alignment is achieved by manually executing COLMAP commands to perform joint Structure-from-Motion reconstruction across both datasets. Once the COLMAP output is generated, it is converted into the format expected by Nerfstudio using the `ns-process-data` command, thereby preparing the data for subsequent training with `ns-train`.

```
# create joint database
$ colmap database_creator
↪  --database_path data/counter_demonstation/merged/database.db

# feature extraction of all images
$ colmap feature_extractor
↪  --database_path data/counter_demonstation/merged/database.db
↪  --image_path data/counter_demonstation/merged/images/
↪  --ImageReader.single_camera 1 --SiftExtraction.use_gpu 1

# feature matching between all images
$ colmap exhaustive_matcher
↪  --database_path data/counter_demonstation/merged/database.db
↪  --SiftMatching.use_gpu 1

# mapping of all images
$ colmap mapper
↪  --database_path data/counter_demonstation/merged/database.db
↪  --image_path data/counter_demonstation/merged/images/
↪  --output_path data/counter_demonstation/merged/sparse/
↪  --Mapper.ba_global_function_tolerance=1e-6

# transform into nerfstudio train-ready data
$ ns-process-data images
↪  --data data/counter_demonstation/merged/images/
↪  --output-dir
↪  data/counter_demonstation/outputs/ns_counter_demonstration
↪  --colmap-model-path
↪  data/counter_demonstation/merged/sparse/0/
↪  --skip-colmap
```

Listing 7: Colmap Commands for Added Dataset Preprocessing and Transformation to Nerfstudio expected format

Before executing the `ns-train` command for continual training, it is necessary to manually modify the `transforms.json` file that was generated during the `ns-process-data` preprocessing command. This file contains metadata about all frames in the joint dataset, including camera poses and file paths. To ensure that the model is trained exclusively on the newly added scene content, we filter this file to retain only the frames corresponding

to the `added` dataset. A constraint we had to face when using the Nerfstudio framework requires that the number of input frames during continual training matches the number used in the initial base training. To satisfy this requirement, we duplicate selected frames from the `added` set until the total number aligns with that of the `base` dataset. Finally, we launch the training process using the `ns-train` command, including the `--load-dir` flag to initialize the model with the weights of the previously trained foundation model, thereby enabling the network to continue training from the saved checkpoint.

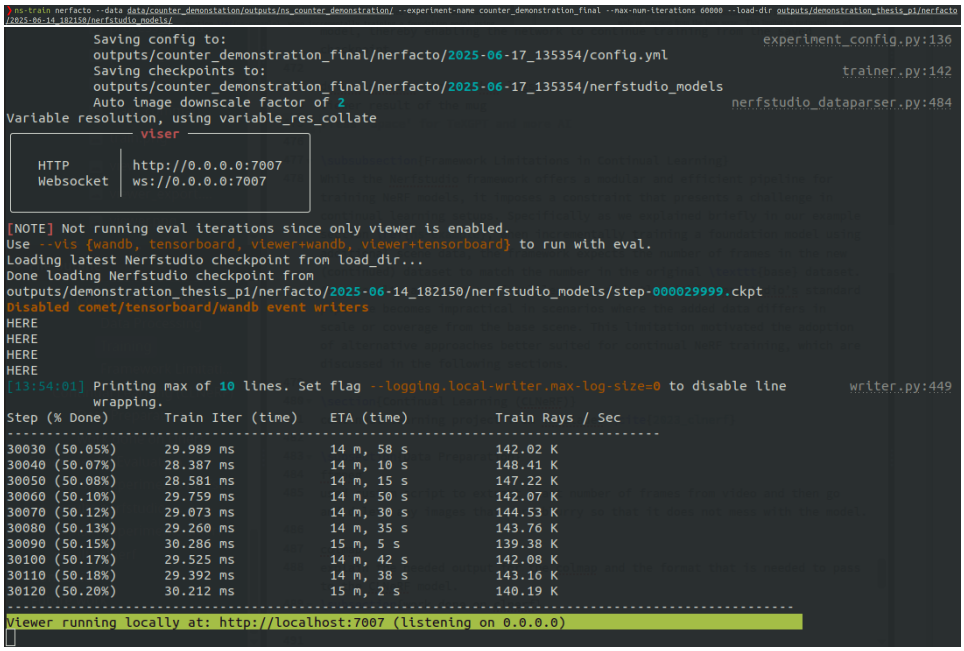


Figure 3.7: Nerfstudio `ns-train` command used to incrementally train the model. First image displays the exact command that was executed, second image showing training information, also showing that the base model was loaded successfully.

At this point, we launch the Nerfstudio viewer using the updated configuration corresponding to the continually trained model. As visualized through the viewer, the model successfully incorporates the newly introduced object from the `added` dataset, demonstrating that it has effectively learned and integrated the new scene content.

Figure 3.8: Nerfstudio viewer result, showing that new object (mug) was successfully learned by the model and added to the scene.

3.1.4 Scene Reconstruction from Drone Data with Nerfstudio

This example demonstrates the Nerfstudio framework's capability to effectively train on drone-captured data, `drz` dataset, successfully reconstructing a complex outdoor scene with high fidelity. Using 200 input images and 30,000 training iterations, the model produced detailed and accurate geometry, showcasing the framework's robustness and adaptability to real-world aerial datasets.

Figure 3.9: Renders of Drone Captured Footage using the Nerfstudio Framework

Framework Limitations in Continual Learning on Nerfstudio

While the Nerfstudio framework offers a modular and efficient pipeline for training NeRF models, it imposes a constraint that presents a challenge in continual learning setups. Specifically as we explained briefly in our example Section 3.1.3, when incrementally training a foundation model using additional scene data, the framework expects the number of frames in the new (continued) dataset to match the number in the original `base` dataset. As a result, applying continual learning directly within Nerfstudio's standard pipeline becomes impractical in scenarios where the added data differs in scale or coverage from the base scene. This limitation motivated the adoption of alternative approaches better suited for continual NeRF training, which are discussed in the following sections.

3.2 Continual Learning NeRF (CLNeRF)

In this section, we present an in-depth analysis of the continual learning pipeline employed in our experiments, based on the CLNeRF framework developed by Intel Labs [15]. We begin by explaining the overall structure and workflow of the CLNeRF model, highlighting its ability to incrementally learn scene updates without catastrophic forgetting, while also explaining the necessary adaptations and modifications made to align the original implementation with the requirements of our experimental setup. This includes the use of custom COLMAP processing steps for dataset alignment, ensuring consistent camera coordinate systems across time-separated captures. We detail how the dataset structure and metadata were reformatted to meet the input requirements of CLNeRF's training pipeline. For qualitative evaluation, a custom interactive renderer was developed that allows users to explore the reconstructed 3D scene. This viewer supports free camera navigation and enables rendering of novel views from arbitrary poses, providing a convenient way to verify that scene updates have been effectively learned and integrated into the final representation.

3.2.1 Overview of the CLNeRF Framework

The continual learning workflow using the CLNeRF framework begins with capturing the scene data. This can be done either though individual image frames or video recordings, similar to Nerfstudio. Regardless of the method, the key requirement is to have complete coverage of the scene, without motion blur and good consistent lighting throughout the capture of the scene

in order to obtain a high-quality reconstruction. Next, the need preprocessing of the obtained training data is executed. As with Nerfstudio, all data must be aligned within a common coordinate system. However, a major distinction in CLNeRF lies in its task-based segmentation, that splits tasks based on image directories. Each different directory of images corresponds to a separate capture at a distinct time step. For instance, if the dataset contains three directories of input images, the training is divided into three tasks, one for the base scene and two for subsequent updates.

During the training process the users can specify which task, time step, they want the model to be trained on. The framework automatically selects the appropriate dataset based on the provided arguments and configures the training accordingly. The training then proceeds as detailed in Section 2.4, continuously giving feedback to the user, by displaying epoch and iteration counters, current loss values, and additional evaluation metrics to track the model's performance. A significant component of the CLNeRF model training process is that, if the current task is not the final one, the system renders and stores frames corresponding to the poses of the training dataset's frames from the newly trained model. These frames are used to populate the replay buffer that supports continual learning in subsequent tasks.

Finally, after training is complete, the trained model can be examined using a custom viewer. This tool allows for free camera movement around the 3D world and rendering from arbitrary poses, allowing for qualitative evaluation and to verify that model updated correctly representation of the scene.

3.2.2 Dataset Preparation and COLMAP Customization

If the scene is captured as a video, in order to prepare the dataset for structure-from-motion (SfM) to be executed, a custom script is employed that utilizes the tool `FFmpeg` to extract a precise number of frames from the input video. After extraction, any frames that appear blurry or with bad lighting are manually removed to avoid degrading model performance. As a best practice, it's advisable to initially extract more frames than needed, allowing flexibility to discard low-quality images while still retaining a sufficient number for training. If the input consists of individual images rather than video, the frame extraction step is skipped and the dataset can be passed directly to `COLMAP`.

To prepare the input data in the format required by the CLNeRF framework, we first process all images from all tasks simultaneously using `COLMAP`, similar to the Nerfstudio framework. This ensures that all tasks

are registered within a shared 3D coordinate system. Each task's images must be placed in separate directories, as the framework relies on directory structure to segment the training tasks. To enforce a specific chronological training order, these directories should be named in alphanumeric sequence, since the framework assigns task IDs based on their directory order. The final output of this preprocessing step includes a COLMAP-generated database containing image correspondences, a sparse reconstruction folder with camera parameters, image poses, and 3D points and the associated image files.

```
# initial setup
current_experiment
|-- images
|    |-- task_1
|        |-- frame_0001.png
|        |-- .
|        |-- frame_x.png
|    |-- task_2
|        |-- frame_0001.png
|        |-- .
|        |-- frame_x.png

# resulting setup
current_experiment
|-- database.db
|-- images
|    |-- task_1
|        |-- frame_0001.png
|        |-- .
|        |-- frame_x.png
|    |-- task_2
|        |-- frame_0001.png
|        |-- .
|        |-- frame_x.png
|-- sparse
|    |-- 0
|-- poses_bounds.npy # optional file that stores near and far depth
↪  bounds (not used in our experiments)
```

Listing 8: Initial directory setup of an experiment and what the CLNeRF framework expects after the data preprocessing is completed.

```
# create joint database
$ colmap database_creator --database_path database.db

# feature extraction of all images
$ colmap feature_extractor
↪  --database_path database.db
↪  --image_path images/
↪  --ImageReader.single_camera 1
↪  --SiftExtraction.use_gpu 1

# feature matching between all images
$ colmap exhaustive_matcher
↪  --database_path database.db
↪  --SiftMatching.use_gpu 1

# make colmap output directory
mkdir sparse

# mapping of all images
$ colmap mapper
↪  --database_path database.db
↪  --image_path images/
↪  --output_path sparse/
↪  --Mapper.ba_global_function_tolerance=1e-6
```

Listing 9: Colmap Commands for Dataset Preprocessing that generate the desired output.

3.2.3 Training Workflow

With the data preprocessing complete, we proceed to model training using a custom script. This script shown below was used to launch the training pipeline with specific configurations tailored to the experiment. To initiate training for the base scene (the first task), we set `task_curr` to 0, while `task_number` is set to the total number of tasks defined for the continual learning sequence.

```
#!/usr/bin/env bash
scene_name=current_experiment   # scene name
export DATA_DIR=/data/$scene_name/

task_curr=0             # task id that is being trained
task_number=2           # total tasks
rep=5                   # replay buffer size
epochs=10               # epochs number during training
batch_size=4096         # size of batches
downsample=1.0          # downsampling during rendering
dim_a=64                # dimension of apperance embeddings
dim_g=16                # dimension of geometry embeddings
scale=8.0               # scene near/far scale
lr=1e-2                 # learning rate
num_gpus=1              # number of gpus to use

experiment_name=${scene_name}_s${scale}_lr${lr}_dima${dim_a}
↪  _dimg${dim_g}_r${rep}_e${epochs}_b${batch_size}_d${downsample}_gpu${num_gpus}
echo Experiment name : $experiment_name

python train_ngpgv2_CLNerf.py \
        --root_dir $DATA_DIR \
        --dataset_name colmap_ngpa_CLNerf \
        --exp_name $experiment_name \
        --rep_size $rep \
        --num_epochs $epochs \
        --task_curr $task_curr \
        --task_number $task_number \
        --batch_size $batch_size \
        --num_gpus $num_gpus \
        --lr $lr \
        --eval_lpips \
        --dim_a $dim_a \
        --dim_g $dim_g \
        --scale $scale \
        --downsample ${downsample} \
        --vocab_size ${task_number} \
        --no_save_test # dont save test video.

echo Training done.
```

Listing 10: Script to launch a CLNeRF training on the captured scene. This scripts allows for modification to all key hyperparameters and constructs a descriptive experiment name automatically.

After training is completed, if the current task is not the final one, the replay buffer will be populated with rendered images generated from the dataset's camera poses for the next task. Additionally, a model checkpoint is saved, and a log file is produced containing valuable training metrics such as loss and PSNR recorded throughout the training process at each training

step. Using the saved model checkpoint, we can load the trained model into the custom renderer to perform a qualitative evaluation, allowing us to render novel views from arbitrary camera poses and verify the quality of the reconstruction, which will be further explained on Section 3.2.4.

3.2.4 Adaptations and Extensions to the Original Code

Code change

Unlike the original implementation by the Intel team, which reserved every 8th image from the dataset for validation and used these poses to evaluate the trained model and generate a rendered video, our approach avoids having a `test_dataset` entirely. Instead, we train on the full dataset to maximize data utilization. Since we have developed a custom renderer with the ability to interactively explore the scene and render from arbitrary poses, a separate validation set is unnecessary for our workflow.

CLNeRF Framework Renderer

An essential component for qualitatively assessing the training process and verifying the success of the training of the model was the development of a custom renderer. This tool allows for free navigation within the scene and rendering from arbitrary camera poses, providing flexible visual inspection of the model's outputs. This capability was not available in the original CLNeRF framework, so we implemented it from scratch to better support evaluation for our experiments and try to reproduce the workings of the Nerfstudio viewer.

Firstly, the renderer parses command-line options that the user provided such as the data path, model checkpoint and other optional parameters being the width and height and the output directory of the generated frames. After loading the checkpoint weights, it creates the output directory and computes an initial camera pose where the navigator will start from. Starting the rendering session, on-screen controls are displayed that allow the user to freely navigate the scene and move to custom camera poses where they can render frames from the current viewpoint. Finally, it provides a feature where users can store their current camera pose to a file, which can later be used for rendering through a separate script without relying on the interactive navigator. This enables consistent rendering from identical viewpoints across both the 'before' and 'after' models, allowing for direct side-by-side comparisons and more effective evaluation of scene changes.

More information about the execution of the rendering workflow is explained in Appendix A, more specifically in the Listing 11 for the navigating renderer and in the Listing 12 for the pre-calculated pose renderer.

Chapter 4

Experimental Evaluation

In this section, we present and analyze our experimental workflow utilizing both the Nerfstudio and CLNeRF frameworks to evaluate the effectiveness of the continual learning techniques that were applied. We outline the specific steps taken in each experiment, the configurations applied, and the objectives we aimed to achieve, testing the limits and adaptability of each algorithm under various conditions. Furthermore, we highlight the challenges encountered during implementation, describe the solutions developed to overcome them, and provide qualitative insights into the results produced by each training pipeline.

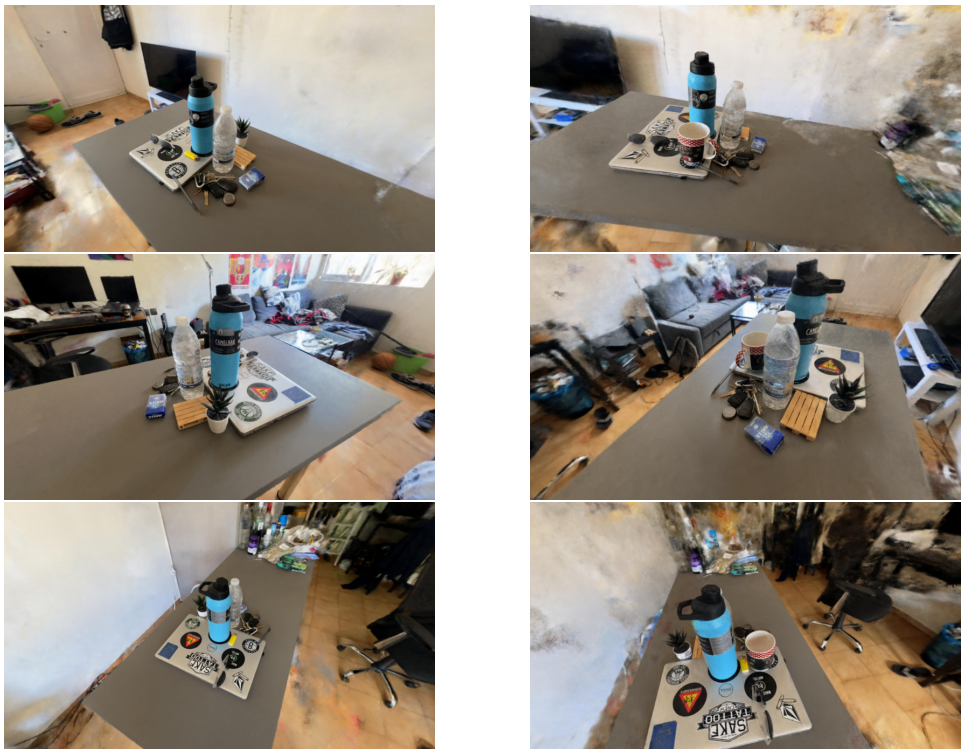
4.1 Nerfstudio: Experiments and results

It is important to note that only limited experimentation was conducted using the Nerfstudio framework due to the key limitation we mentioned, the requirement for the base and added scenes to contain the same number of input images. As a result, we carried out some experiments within this environment. The "`counter_add`" and the "`counter_rm`" scene. We also examined how different amounts of iterations on the continued training affect the quality of the resulting model.

4.1.1 counter_add

In this experiment, we first trained the model solely on the `base` dataset to generate the foundation model, afterwards training with the `added` dataset was done in order to implement continual learning. For the subsequent training with the added data, we modified the `transforms.json` file as described

in Section 3.1.3 to ensure compatibility with the Nerfstudio framework. Specifically, the base model was originally trained for 30,000 iterations which took approximately 15 minutes.



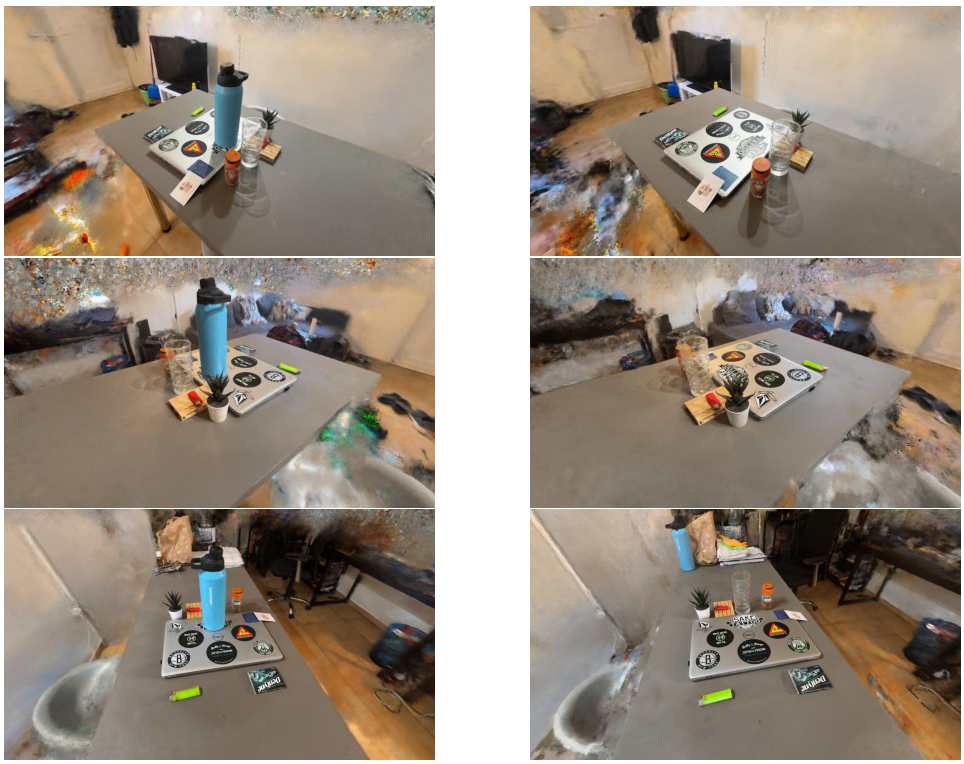
(a) Base model. (b) Updated model.

Figure 4.1: Comparison of scene reconstructions before and after continual learning for 30k iterations using the Nerfstudio framework. Each row displays renderings from a specific camera pose. **Left column (a)**: the base model before any updates. **Right column (b)**: the model after additional training with the updated scene. The visual differences demonstrate the model's successful integration of new scene elements through continual learning and correct addition of the object to the scene, with only a bit of visual impairment to the background parts of the scene.

4.1.2 counter_rm

In this experiment, we first trained a foundation model using the `base` dataset. Subsequently, we introduced the `added` dataset to continue training

under a continual learning setup with the Nerfstudio Framework. As in the previous experiment, the `transforms.json` file was adjusted to enable compatibility with the Nerfstudio framework for continual training. The base model was trained for 30,000 iterations, taking approximately 16 minutes, while the added model was trained for an additional 30,000 iterations over 14 minutes.

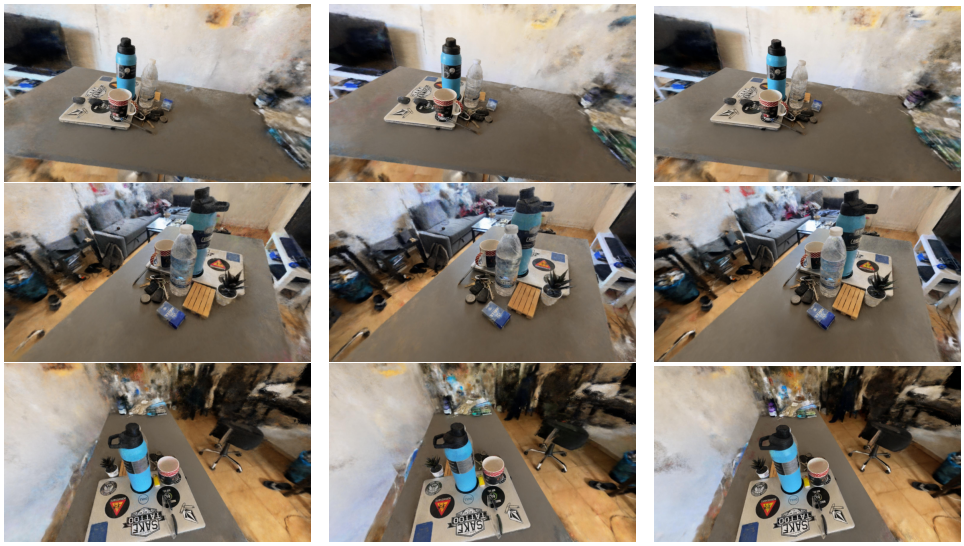


(a) Base model. (b) Updated model.

Figure 4.2: Comparison of scene reconstructions before and after continual learning for 30k iterations using the Nerfstudio framework. **Left column (a)**: the base model before any updates. **Right column (b)**: the model after additional training with the updated scene. The model is shown to have successfully integrated the changes of the new scene through continual learning and has correctly removed the object (flask) while observing minor degrading in the background of the scene.

4.1.3 Evaluating Iteration Count for Added Scene Training

To evaluate the effectiveness of continual learning within the Nerfstudio framework, three additional training tasks were conducted using the updated scene data, each with a different number of iterations: 7,500, 15,000 and 30,000. This setup allowed us to evaluate how well the model adapts to newly introduced data and how training iterations impact the quality of the learned scene modifications.



(a) 7.5k Iterations. (b) 15k Iterations. (c) 30k Iterations.

Figure 4.3: Evaluation of training efficiency for continual learning with the Nerfstudio framework. The figure compares results from models trained with **7.5k**, **15k**, and **30k** iterations. Despite fewer iterations, the 15k and even 7.5k models maintain comparable visual quality, demonstrating that the desired scene updates can be learned in significantly less time, highlighting the efficiency gains of continual learning.

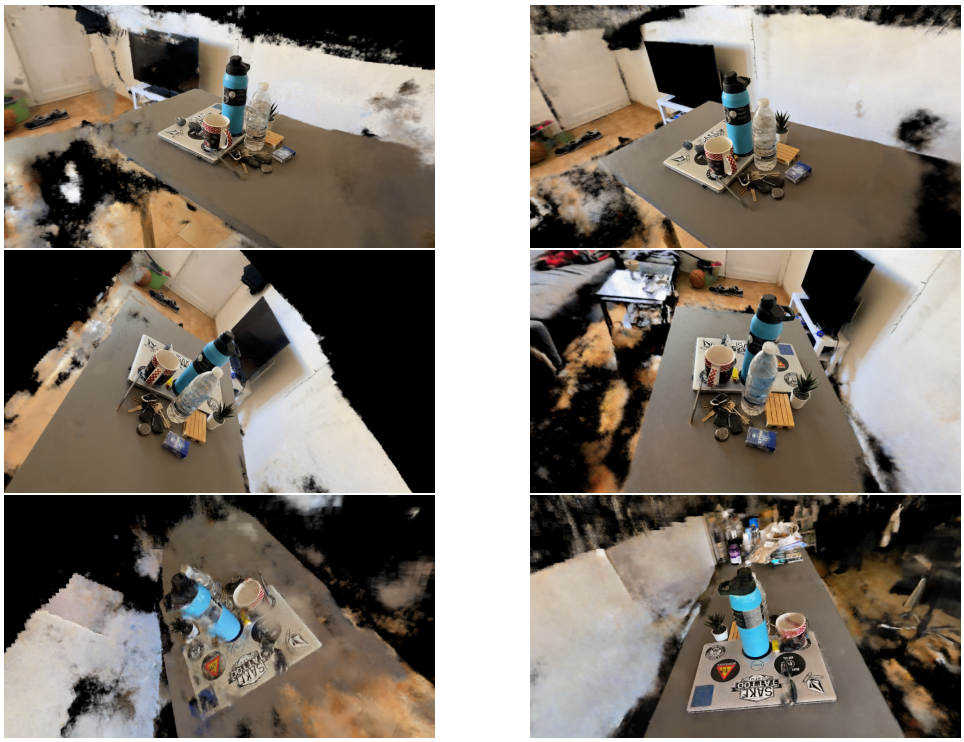
The results demonstrated that the model was able to successfully learn the scene changes even with significantly fewer training iterations than the training needed for the base model. This highlights the efficiency of continual learning in reducing training time without compromising reconstruction quality.

4.2 CLNeRF: Experiments and results

In this section, we present the experiments conducted using the CLNeRF framework, detailing the specifications applied during setup, the training durations, and the qualitative evaluation methods used to assess model performance. We include renders from each experiment that visually demonstrate how and why the CLNeRF pipeline successfully adapted to scene changes, highlighting the strengths and limitations of continual learning in 3D scene reconstruction. The scene's dataset, referred to as `counter`, served as the primary testbed for our experiments. Many different versions of the `counter` dataset were used, but in a similar setup each time. This dataset provided us with a broader range of continual learning scenarios, allowing to assess the framework's robustness across various challenges such as object additions, removals, lighting changes, and occlusions of the scene.

4.2.1 From Scratch Training vs. Continual Learning

In this experiment, we trained two models using the newly acquired data. The first was trained from scratch using only the `added` dataset, while the second continued training from the previously trained base model (trained with the `base` dataset) using the same additional data. This comparison aimed to highlight the qualitative benefits of continual learning. Even though there were no significant gains in training time, the continual learning approach consistently produced higher quality reconstructions particularly from specific camera poses.



(a) Only Added model.

(b) Continual Learning model.

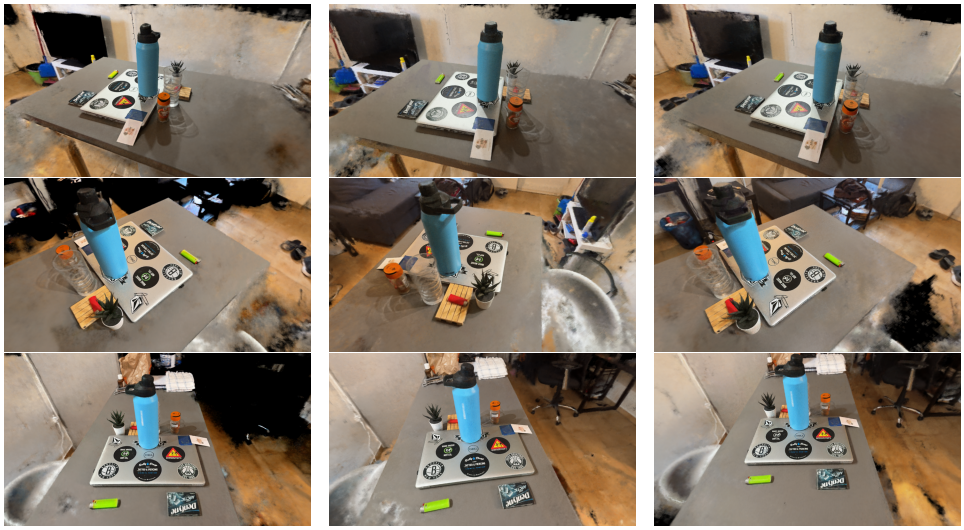
Figure 4.4: The left column shows results from the model trained solely on the `added` dataset, while the right column displays results from the model trained using continual learning (base + added data). Although both models perform similarly from certain viewpoints, notable differences emerge in more challenging perspectives. In the third row, the continual learning model maintains better fidelity and fewer artifacts. The model that trained only with the `added` dataset **(left column (a))** took for 8 minutes. The total training time for the combined base and added model **(right column (b))** was 31 minutes (23 minutes for the base model and 8 minutes for the additional training). However, since the base model would have been trained beforehand regardless, our primary focus is on the added training time.

Experiment Conclusion

Conclusively, this experiment demonstrates the qualitative advantages of continual learning over training a model solely on newly captured data. While both approaches can produce comparable reconstructions from certain viewpoints, the continual learning model consistently yields more accurate results. The final row in the comparison Figure 4.4 highlights this clearly, with the `added-only` model struggling to maintain fidelity and exhibiting noticeable reconstruction errors from certain viewpoints, whereas the model that went was trained incrementally preserves visual quality. This reinforces the value of continual learning for efficiency and more consistent scene reconstructions across diverse viewpoints.

4.2.2 Impact of Base Dataset Size on Reconstruction Quality

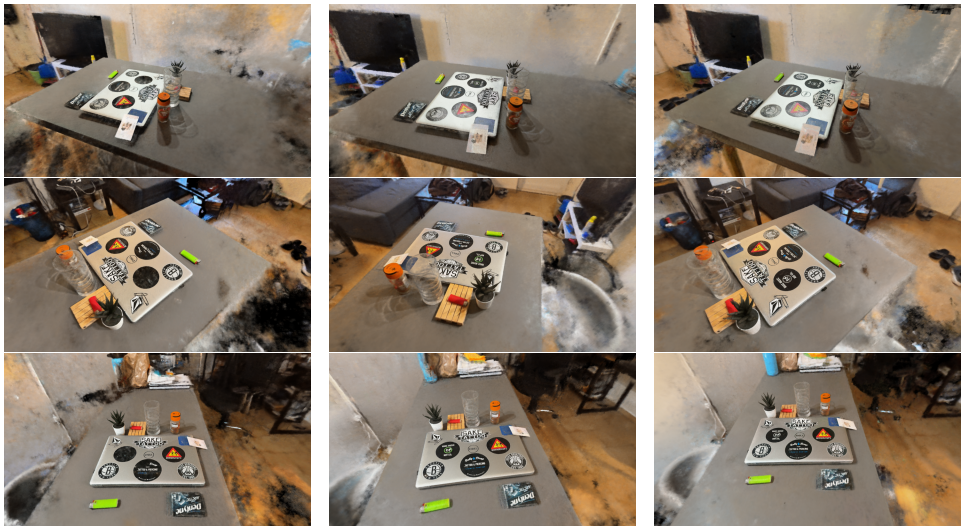
In this experiment, we wanted to investigate how the quantity of training data for the base scene affects the final reconstruction quality and the reliability of the structure-from-motion (SfM) pipeline. Specifically, we compare models trained with 100, 200, and 300 frames extracted from the same `base` input video, `counter_rm` dataset, to evaluate whether increasing the number of input images enhances both COLMAP's performance and if it leads to fewer artifacts in the final CLNeRF reconstruction. Since COLMAP is an approximate method, the hypothesis is that more frames may yield better camera alignment and point cloud accuracy, which in turn could improve the fidelity of the resulting base model.



(a) 100 Frames Model. (b) 200 Frames Model. (c) 300 Frames Model.

Figure 4.5: Qualitative comparison of reconstructed scenes using different numbers of training images for the base model: 100 (left column), 200 (middle column), and 300 (right column) frames. The main foreground objects are reconstructed with similar visual quality. Slight differences are noticeable in the background details.

Next, we proceed by training each model using an `added` dataset consisting of the same number of frames across all cases, the `added` dataset in this case consists of 50 frames extracted from the captured video where an object is removed from the base scene, this scenario is explained further in Section 4.2.4. Having the same number of frames across all cases allows us to isolate and evaluate how the base model training influences the final reconstruction outcome after continual learning.



(a) 100 Frames Model. (b) 200 Frames Model. (c) 300 Frames Model.

Figure 4.6: Qualitative comparison of reconstructed scenes using different numbers of training images for the added models: 100 (left column), 200 (middle column), and 300 (right column) frames. We observe that the main objects in the final scene reconstructions are consistently well-preserved across all cases, and the object removal has been successfully achieved in each. While the overall scene quality remains high regardless of the `base` dataset size, models trained with a higher number of base frames exhibit slightly improved background detail and consistency as in the base models.

Experiment Conclusion

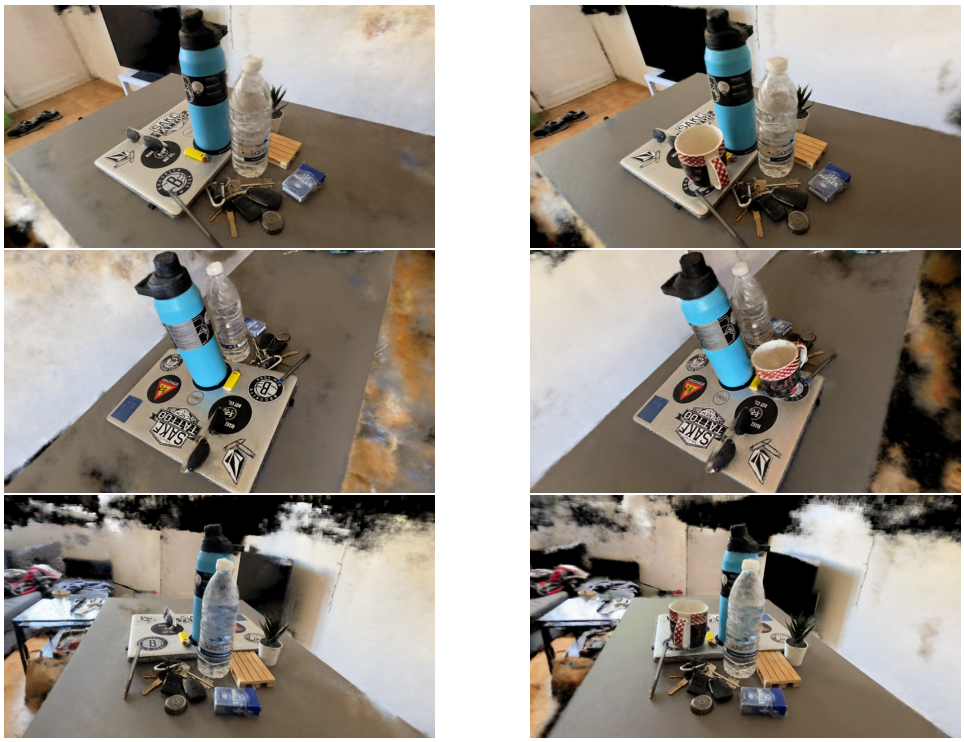
The training times for the different `base` dataset sizes show a trend of increasing computational cost with no clear improvements in visual quality. Specifically, as the number of input frames increases from 100 to 200 to 300, the COLMAP preprocessing time grows from 5 to 12 to 23 minutes respectively, and base model training time rises from 30 to 45 to 68 minutes, since more frames need to be rendered for the next training task. Despite this substantial increase in time (more than double), the resulting scene quality remains largely unchanged. Only slight gains are observed in background detail and consistency. This suggests that while additional training data can improve reconstruction quality in subtle ways, the benefits diminish relative to the cost of training times. The added model training time remained constant across all cases (10 minutes), further emphasizing that the bottleneck

in efficiency lies in the base model reconstruction stage.

4.2.3 Data and Task Volume changes

The aim of this experiment was to investigate how varying the number of training tasks and the number of training images per task, while keeping the total number of images constant across all configurations, affects reconstruction quality and overall training time but finally having the same number for the whole dataset. Additionally, we set out to experimentally determine the minimum number of new input frames needed in the `added` dataset to reliably capture and reflect scene changes in the continually updated model for future experiments. This allowed us to evaluate how different training structures influence the model's learning efficiency. To ensure consistent comparisons, we saved a fixed set of camera poses across all training tasks and rendered three images for the models, before (base), intermediate (if there was one) and after (added) continual training. These configurations were tested under three scenarios: 4 tasks with 10 images each, 2 tasks with 20 images each, and a single task with 40 images. This setup enabled a comprehensive evaluation of how training structure influences performance and learning efficiency.

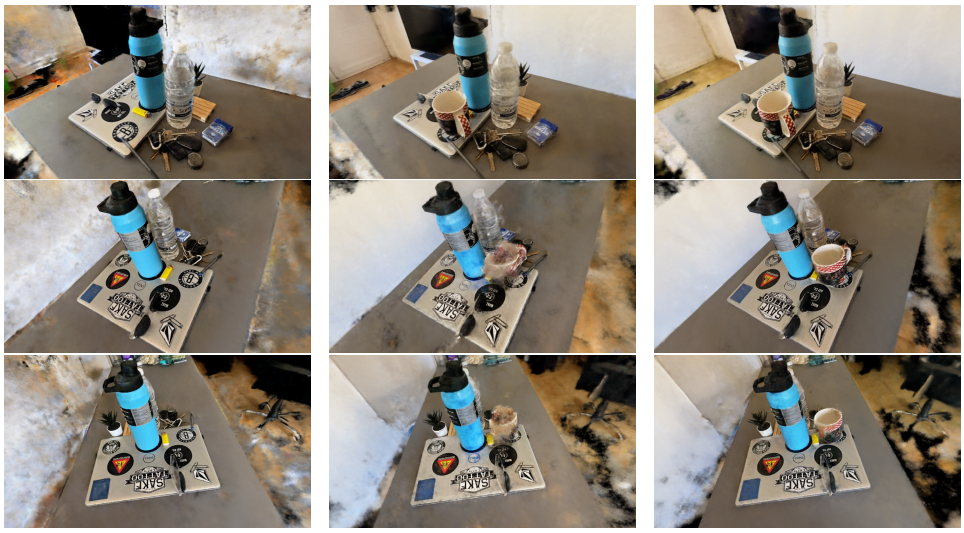
1 Training Task of 40 Frames



(a) Base model. (b) Updated model.

Figure 4.7: The model successfully integrated the newly added object into the scene. The primary elements remain accurately reconstructed, with only minimal degradation in background quality, indicating effective continual learning with minimal compromise to the original scene representation. The total training time up to final model was 31 minutes, 23 for the **base (a)** model and 8 **added (b)** one.

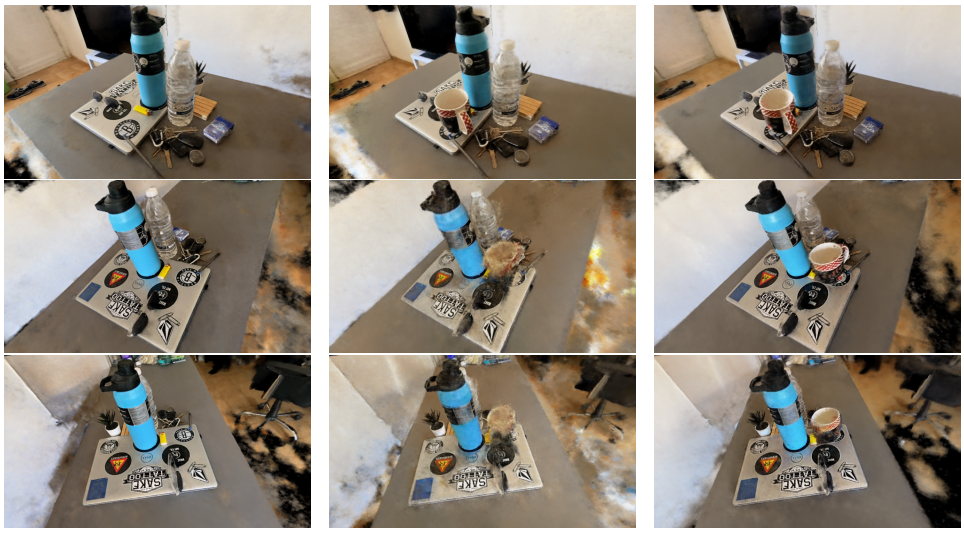
2 Training Task of 20 Frames



(a) Base Model. (b) Intermediate Model. (c) Final Model.

Figure 4.8: The model successfully learned the introduction of the newly added object into the scene. Where the main objects are accurately reconstructed, but this time the background quality seems to have increased. The total training time up to final model was 63 minutes, 28 for the **base model (a)**, 27 for the **intermediate (b)** one and 8 **added (c)** one.

4 Training Task of 10 Frames



(a) Base Model. (b) Intermediate Model. (c) Final Model.

Figure 4.9: The results presented above illustrate the model's progression through three stages: **base (a)**, **intermediate (b)**, and **final (c)**. While the visual quality differences between the final models are relatively subtle, the cumulative training time for this particular experiment was significantly higher. Specifically, the final model required 113 minutes to complete training, with the base model and the four continual learning tasks taking approximately 24, 27, 27, 28, and 7 minutes, respectively. This highlights the substantial time overhead introduced by multiple incremental training stages, despite limited observable improvement in output quality.

Experiment Line Conclusion

This line of experiments that took place highlight that the number of training tasks has a significant impact on total training time of the final model. The training durations for the three experiments increased significantly, from 32 minutes for `counter_1_40`, to 63 minutes for `counter_2_20`, and up to 113 minutes for `counter_4_10`, almost doubling each time. Despite this significant rise in training time, no substantial improvements in scene representation were observed across the experiments. This time overhead occurs because, when the model is training for an intermediate task and more tasks

are expected to follow, the framework must render and store frames from the current training dataset into a replay buffer to support future learning steps. This additional rendering step introduces overhead. Conclusively, training the same total number of images in a single task eliminates the need for intermediate replay buffer updates and substantially reduces the total training time, while not compromising final model quality.

Importantly, the `counter_4_10` experiment allowed us to identify the lower bound of input frames required to achieve satisfactory reconstruction quality. We observed that when the base model was trained with fewer than 30 frames, the results were suboptimal, failing to accurately capture the updated scene. In contrast, models trained with 30 to 40 frames consistently produced reliable reconstructions, successfully integrating the new content. This insight provided a useful baseline for subsequent experiments, indicating that more than 30 frames should be used for the `added` dataset to ensure high-quality results in our continual learning scenarios.

4.2.4 Object Addition and Removal

In this set of experiments, we aimed to evaluate the CLNeRF model's ability to handle dynamic changes, specifically, the addition and removal of objects. These changes challenge the model not only geometrically, as the 3D structure of the scene is altered, but also photometrically, due to variations in appearance such as color and texture. We investigated how effectively the model can integrate newly introduced elements or accurately update the scene by removing previously present ones, and whether it can preserve or reveal occluded details as a result of these modifications.

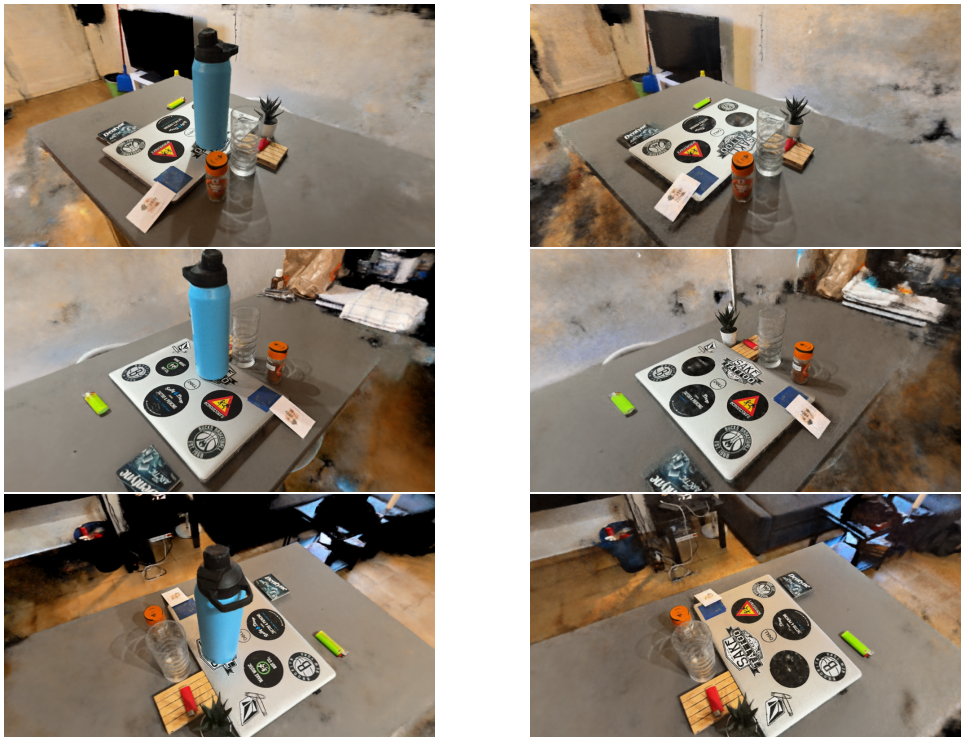
counter_add

The addition of a new object to the scene was explored in Section 4.2.3, where we demonstrated that the CLNeRF framework successfully learned the updated scene through continual training, as shown in Figure 4.7, alongside also investigating how the varying number of tasks affects the model's performance and what the lower bound of added training frames should be to have a competent resulting model.

counter_rm

In this experiment, we aimed to evaluate whether the continual learning algorithm could successfully update the scene when a change involves the

removal of an object rather than the addition of one. This scenario is particularly meaningful as it not only tests the model's ability to adapt to geometric alterations but also challenges its capacity to handle photometric changes. The removed object featured strong, vivid colors, and its absence affected surrounding transparent and reflective elements in the scene.

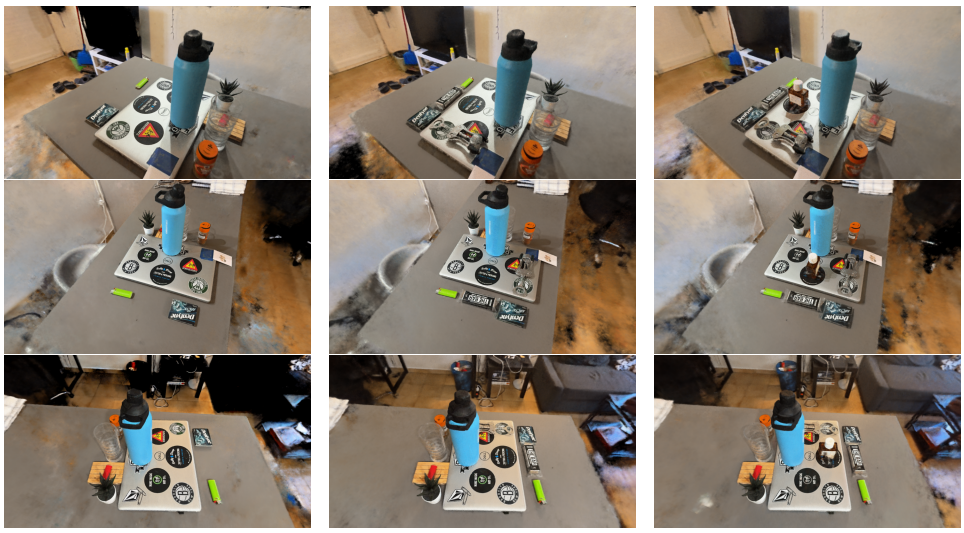


(a) Base model. (b) Updated model.

Figure 4.10: The figure above clearly demonstrates that the CLNeRF model successfully adapted to the removal of the object (flask), accurately updating the scene based on the new input data. The model not only was able to update the model for the object removal, but also revealed and reconstructed details that were previously occluded, in this case, the text on the sticker that was hidden behind the object in the base scene. The total training time to reach the final model was 40 minutes: 31 minutes for the **base (a)** model and 9 minutes for the **added (b)** model.

counter_tt

The goal of this experiment was to test the CLNeRF framework's ability to handle sequential scene updates by adding new objects at two distinct time steps. After training the base model on the `base` scene dataset, we introduced a wine opener and a protein bar as the **first** update. Subsequently, a new object (cologne bottle) was added in a later training task. This setup allowed us to assess how well the model can incrementally learn and integrate **multiple** updates without overwriting previously learned information. It also served to test the framework's robustness in managing both geometric and visual consistency over time as the scene evolves.



(a) Base Model. (b) First Model Update. (c) Final Model Update.

Figure 4.11: The base model was first trained on the initial scene, followed by two continual learning tasks introducing new objects (wine opener and protein bar, then cologne bottle). This experiment demonstrates the model's ability to incrementally incorporate multiple scene updates while preserving prior knowledge and maintaining geometric and visual consistency across time.

Experiment Line Conclusion

In this line of experiments it is demonstrated that the CLNeRF algorithm can successfully adapt to both geometric and appearance-based changes in a

dynamic scene. From a geometric standpoint, the model effectively learned to integrate the addition of new objects and accurately removed objects that no longer existed, updating the spatial layout of the scene accordingly. Simultaneously, the algorithm was able to capture secondary effects caused by these geometric changes, such as alterations in lighting, reflections, and occlusions of other details which directly impacted the appearance of surrounding objects. This ability is critical in continual learning scenarios, where the goal is not only to reflect structural modifications but also to ensure consistent and accurate representation of the scene's evolving visual context.

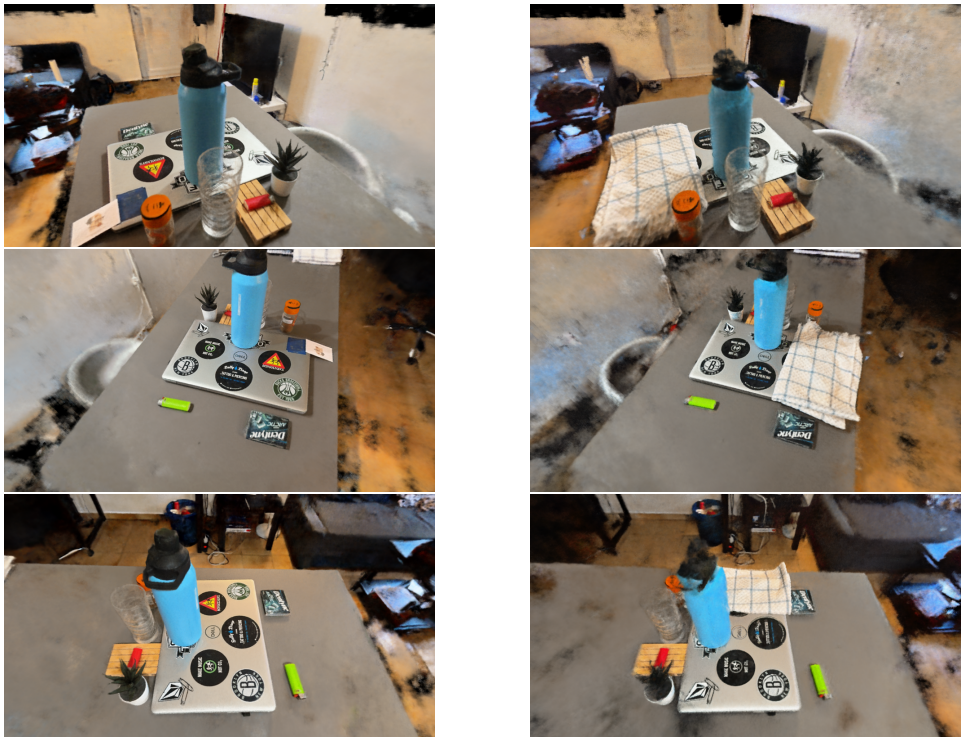
4.2.5 Impact of Partial Occlusions

In this set of experiments, we investigated how partial occlusions in the `added` dataset affect the performance and stability of the continual learning process. Specifically, we progressively occluded different portions of the scene using objects with varying textures and colors. The goal was to assess the model's ability to adapt to altered visual information and to evaluate whether such occlusions lead to catastrophic forgetting of the original scene content. By varying the appearance of the occlusions across experiments, we aimed to test the robustness of the CLNeRF framework under increasingly challenging visibility conditions.

As displayed in later sections, in the experiment `counter_bland` involving a bland-colored towel, partial occlusion of the scene led to unexpected artifacts in the resulting reconstruction, more specifically to the cap of the flask. To investigate this behavior further, we conducted a series of experiments using a **shirt** with a neutral color to progressively increase the level of scene coverage. This allowed us to better understand how incremental occlusion with low-texture surfaces impacts the model's ability to retain and integrate scene information.

counter_bland

In this experiment, we evaluated the algorithm's ability to adapt to partial occlusions by introducing a low-texture object covering part of the scene, a white towel with blue stripes. The goal was to test how such occlusions affect the model's retain ability and update scene information.

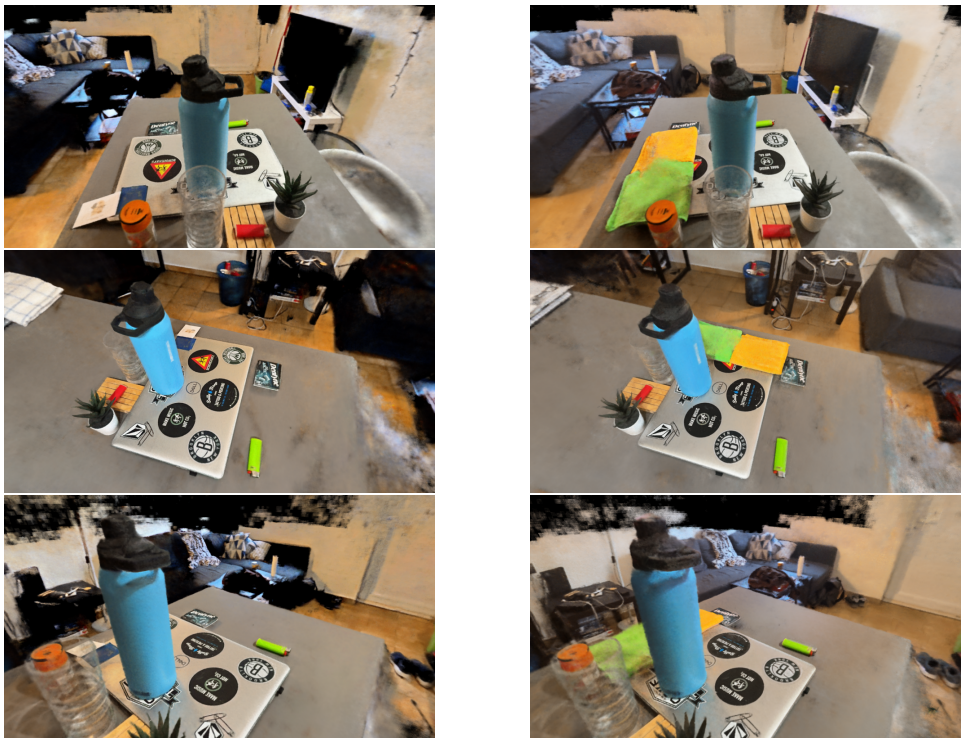


(a) Base model. (b) Updated model.

Figure 4.12: The figure above illustrates the model's ability to successfully integrate the towel into the scene. However, some quality degradations are noticeable, the blue flask's cap, which was accurately reconstructed in the base model, appears to be distorted in the updated scene. This observation prompted further investigation into the model's behavior under partial occlusion and its potential impact on previously learned geometry. The total training time to reach the final model was 37 minutes: 28 minutes for the **base (a)** model and 9 minutes for the **updated (b)** model.

counter_extra

In this experiment, we introduced two smaller towels that occluded the same part of the scene as in the previous `counter_bland` experiment. However, unlike the earlier setup where the occlusion was caused by a towel with soft tones, these new towels featured vivid green and orange color. The goal was to investigate how the model responds to occlusions that introduce new, visually distinct color information into the scene and how it affects the adaptation capabilities of the CLNeRF framework.



(a) Base model. (b) Updated model.

Figure 4.13: The figure above shows how the model successfully adapted to the addition of the two vividly colored towels in the scene. Aside from some minor color blending observed between the two towels in the first row of rendered frames, the model demonstrates strong reconstruction performance, accurately capturing the scene's updated geometry and appearance. Indicating that the CLNeRF framework can effectively integrate new, visually distinct elements without compromising overall scene quality. The total training time to reach the final model was 41 minutes: 33 minutes for the **base (a)** model and 8 minutes for the **updated (b)** model.

4.2.6 Shirt Occlusion Experiments

The shirt occlusion experiments were designed to systematically analyze how increasing levels of occlusion percentage of the original scene. The occlusion is achieved with the introduction of neutral colored shirt. By incrementally covering more of the scene across three stages (`low`, `mid`, and `high` coverage), we aimed to evaluate how occlusion severity impacts both the quality of reconstruction and the risk of catastrophic forgetting. These experiments also served to test the model's resilience in preserving the visibility of objects from the base scene that became partially or fully obscured over time. Together, they provide insight into how the framework handles real-world scenarios where parts of a scene may be temporarily blocked or altered.

counter_shirt_low

In this experiment, we introduced a low-level occlusion to the base scene, aiming to cover only a small portion of the original setup. Specifically, the shirt was positioned to partially obscure the white book that had vividly colored lighters placed on top. This setup allowed us to observe how minor visual obstructions influence the model's ability to retain previously learned scene details.

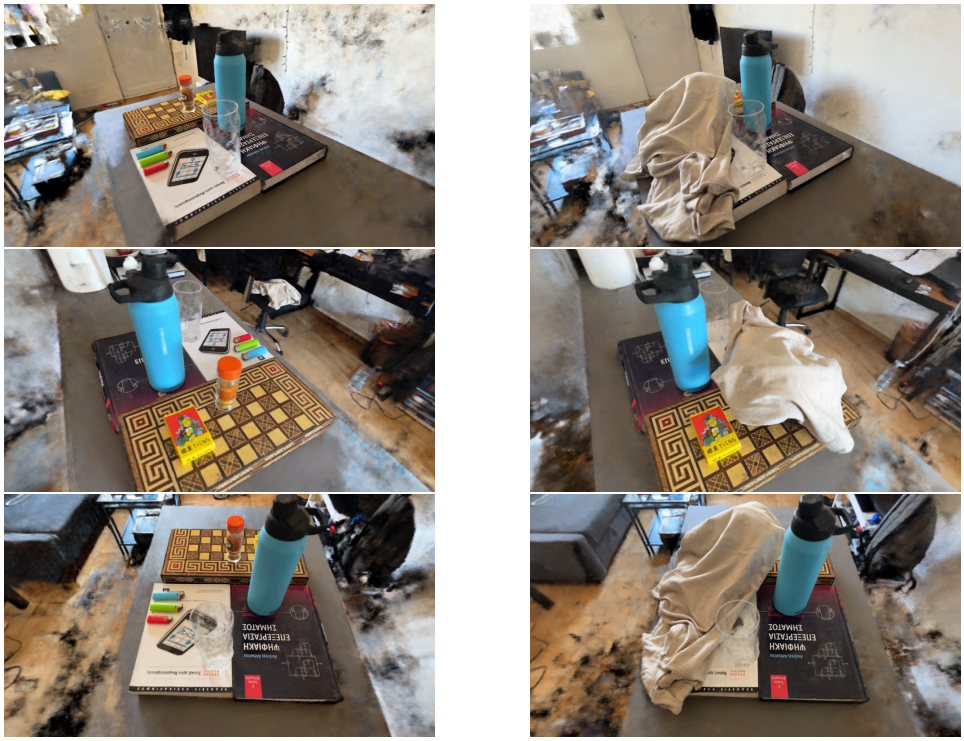
(a) Base model.

(b) Updated model.

Figure 4.14: The figure demonstrates that the model successfully learned the addition of the shirt to the scene during continual training. Moreover, the quality the background in the final model also shows improvement, indicating that the additional training not only incorporated the new occlusion but also refined the overall scene reconstruction. The total training time to reach the final model was 37 minutes: 30 minutes for the **base (a)** model and 7 minutes for the **updated (b)** model.

counter_shirt_mid

In a similar fashion to the experiment before, we introduced a mid-level occlusion to the base scene, aiming to cover a slightly bigger percentage of the original setup. Specifically, the shirt was positioned to obscure again the white book with the lighters but also a part of the backgammon board and also the pepper pot that was on top of it. This setup allowed us to observe how bigger visual obstructions impact the model's performance.



(a) Base model.

(b) Updated model.

Figure 4.15: The results once again demonstrate that the model can effectively update based on new data, even when a larger portion of the original scene is altered. It successfully learns the new colors introduced by the occluding object while adapting to the removal of previous feature, such as the brightly colored cap of the pepper pot. The total training time to reach the final model was 35 minutes: 27 minutes for the **base (a)** model and 8 minutes for the **updated (b)** model.

counter_shirt_high

In this experiment, we significantly increased the occlusion level by covering a large portion of the base scene with a shirt, obscuring multiple key objects and textures, leaving only the flask and the playing cards uncovered from the original scene. By introducing high coverage with a single, textured object, we again aimed to challenge the model's capacity to learn new geometry and color distributions without suffering from catastrophic forgetting.

(a) Base model. (b) Updated model.

Figure 4.16: From the above figure, we observe that the model is once again able to robustly reconstruct the updated scene, successfully integrating the added shirt despite it obscuring a significant portion of the original setup. Importantly, the consistency and integrity of the remaining scene elements are preserved, demonstrating the model's resilience in handling occlusions without degrading the quality of previously learned content. The total training time to reach the final model was 36 minutes: 28 minutes for the **base (a)** model and 8 minutes for the **updated (b)** model.

Experiment Line Conclusion

The experiments in this section provide valuable insight into the robustness and limitations of the CLNeRF framework when dealing with partial occlusions. By introducing occlusions of varying size but having a similar texture, and color, we evaluated the model's ability to incorporate new visual information while preserving previously learned scene content. Our results indicate that the CLNeRF model is generally capable of adapting to many kinds of occlusion coverage percentages. In most cases, the model successfully integrates newly introduced geometry and visual features while maintaining strong consistency in unaffected parts of the scene.

In a scene where newly added objects showed high vividness in `counter_extra`, the changes were integrated with high visual fidelity. On the other hand, the low-texture and bland objects like in `counter_bland` introduced minor degradation in certain previously stable scene regions, which made us experiment further with a similar kind of setup. The `shirt` occlusion experiments further demonstrated that the model can handle increasingly severe occlusion levels, including scenarios where the majority of the scene is hidden. Even under high coverage, the model retained its ability to reconstruct the newly occluded content with minimal artifacts and no evidence of catastrophic forgetting. These experiments help to validate the framework's utility in dynamic settings where visibility may be compromised or altered over time.

4.2.7 Scene Lighting Changes

In this final experiment, the `added` dataset did not introduce or remove any objects from the scene. Instead, the only change was a shift in the lighting conditions, by adding a new light source. The goal of this experiment was to evaluate how well the CLNeRF framework can adapt to changes in illumination, including variations in shadow placement and the altered appearance of transparent or reflective objects caused by the new lighting.

counter dark

(a) Base model. (b) Updated model.

Figure 4.17: The model successfully learns and reconstructs the updated lighting conditions in the scene. Noticeably, it improves the appearance of elements like the bottle cap, which was not well reconstructed in the base model. Shadows and other lighting-dependent details are accurately captured, and the background is correctly adjusted to reflect the changes in the altered floor and wall coloration. Importantly, the objects within the scene retain their structural and textural integrity, adapting smoothly to both the shadows cast upon them and the direct effects of the new light source. The total training time to reach the final model was 37 minutes: 29 minutes for the **base (a)** model and 8 minutes for the **updated (b)** model.

Experiment Conclusion

This experiment demonstrated the CLNeRF framework's ability to handle non-geometric scene changes by successfully adapting to altered lighting conditions. Despite no additions or removals of objects, the model effectively captured the resulting variations in shadows, reflections, and color tones. The model update was able to maintain the structural consistency of existing elements while adjusting the global illumination effects across the scene. These results highlight the model's robustness in learning appearance-based scene updates, reinforcing its applicability in dynamic real-world environments where lighting variations are common.

Chapter 5

Conclusion

In this thesis, we investigated the application of continual learning to Neural Radiance Fields (NeRFs) for dynamic 3D scene reconstruction, particularly in the context of rapid response scenarios such as natural disasters. Traditional NeRF pipelines are limited by their need to be retrained from scratch upon any scene change, making them impractical for real-world environments where they may undergo changes. Our goal was to overcome these limitations and demonstrate the feasibility of incremental learning within NeRF-based systems. We began by evaluating Nerfstudio, modifying its pipeline to accommodate continual learning. However, structural limitations constrained its usability. This led us to focus primarily on CLNeRF, a more flexible framework that we customized extensively. Through a series of carefully designed experiments, we assessed the robustness and scalability of continual learning in a range of challenging scenarios, including object additions and removals, lighting variations, occlusions, and varying dataset sizes and structures. Our results confirm that continual learning that the models preserve and sometimes improve the previously learned scene elements while also adapting to changes. Even under severe occlusions or complex appearance changes, the CLNeRF framework maintained high reconstruction fidelity, demonstrating resilience against catastrophic forgetting. Although limited by the lack of drone footage of an "after" scene, our simulated datasets successfully captured the core challenges of continual scene updates.

Ultimately, this work highlights the potential of continual learning for enabling real-time, adaptive 3D reconstruction in dynamic environments. With further improvements in model scalability, viewer interactivity, and hardware support, these methods could be deployed in practical applications, where quick and reliable scene understanding is vital.

Chapter 6

Future work

Based on the insights gained throughout this project, several promising directions for future development have been identified:

6.1 Multi-GPU Support for CLNeRF

We aim to extend the CLNeRF framework to support training and rendering using multiple GPUs. Although preliminary attempts were made during our experiments, the implementation was inconsistent and unstable. A properly integrated multi-GPU pipeline could prove very useful since it can significantly reduce training and rendering times.

6.2 3D Viewer for CLNeRF Models

Another important enhancement would be the development of a real-time 3D viewer tailored for CLNeRF. The proposed solution involves loading two instances of the trained model the user wants to view. The first instance of the model will render low-resolution frames during user navigation through the scene, and the other for generating high-quality frames when the camera becomes stationary, indicating the user wants to inspect the specific viewpoint they are currently in. Additionally, a ”before-and-after” toggle could be useful feature while developing a viewer of models for the CLNeRF framework. By simultaneously loading the base and updated models, the users could compare the scene's state across time from identical camera poses, a useful tool for monitoring the results of the training and providing valuable information about the scene's updates without having to reload models each time.

6.3 Solve Compatibility Issues of Nerfstudio with Unequal Dataset Sizes

Currently, the Nerfstudio framework assumes equal numbers of frames between `base` and `added` datasets, posing a major limitation for continual learning. We propose to address this by modifying the way base model checkpoints are loaded and how dataset metadata is processed. By getting around the dataset size requirement during continual training, we can make Nerfstudio more flexible and compatible with real-world scenarios where the number of new frames often differs from the original.

Bibliography

[1] Jaeyoung Chung et al. 'MEIL-NeRF: Memory-Efficient Incremental Learning of Neural Radiance Fields'. In: *arXiv preprint arXiv:2212.08328* (2022).

[2] Luc Frachon. *Neural Radiance Fields from Scratch (Part II): From Theory to Practice*. Medium. Apr. 2023. URL: https://medium.com/@luc.frachon/neural-radiance-fields-from-scratch-part-ii-from-theory-to-practice-66a622e28998 (visited on 12/06/2025).

[3] GeeksforGeeks. *Neural Networks – A Beginner's Guide*. Accessed: 2025-06-07. 2025. URL: https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/.

[4] Κ. Λαγουβάρδος and Β. Κοτρώνη and Θ. Μ. Γιάνναρος and Γ. Κύρος. *Meteo: Το 37% των δασών της Αττικής κάπκε τα 8 τελευταία χρόνια*. Greek. 2024. URL: https://meteo.gr/article_view.cfm?entryID=3355 (visited on 22/05/2025).

[5] N. Max. 'Optical models for direct volume rendering'. In: *IEEE Transactions on Visualization and Computer Graphics* 1.2 (1995), pp. 99–108. DOI: 10.1109/2945.468400.

[6] Ben Mildenhall et al. 'NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis'. In: *ECCV*. 2020.

[7] Thomas Müller et al. 'Instant Neural Graphics Primitives with a Multiresolution Hash Encoding'. In: *ACM Trans. Graph.* 41.4 (July 2022), 102:1–102:15. DOI: 10.1145/3528223.3530127. URL: https://doi.org/10.1145/3528223.3530127.

[8] Ryan Po et al. *Instant Continual Learning of Neural Radiance Fields*. 2023. arXiv: 2309.01811 [cs.CV]. URL: https://arxiv.org/abs/2309.01811.

[9] Johannes Lutz Schönberger and Jan-Michael Frahm. 'Structure-from-Motion Revisited'. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[10] Johannes Lutz Schönberger et al. 'Pixelwise View Selection for Unstructured Multi-View Stereo'. In: *European Conference on Computer Vision (ECCV)*. 2016.

[11] Matthew Tancik et al. 'Nerfstudio: A Modular Framework for Neural Radiance Field Development'. In: *ACM SIGGRAPH 2023 Conference Proceedings*. SIGGRAPH '23. 2023. URL: https://docs.nerf.studio/index.html.

[12] Wikipedia contributors. *2024 Noto earthquake*. Accessed: 2025-05-22. 2024. URL: https://en.wikipedia.org/wiki/2024_Noto_earthquake.

[13] Wikipedia contributors. *2024 United Arab Emirates floods*. Accessed: 2025-05-22. 2024. URL: https://en.wikipedia.org/wiki/2024_United_Arab_Emirates_floods.

[14] Xiuzhe Wu et al. 'CL-NeRF: Continual Learning of Neural Radiance Fields for Evolving Scene Representation'. In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh et al. Vol. 36. Curran Associates, Inc., 2023, pp. 34426–34438. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/6c7154e394e24c69409256ccf8bf0804-Paper-Conference.pdf.

[15] Matthias Müller Zhipeng Cai. 'CLNeRF: Continual Learning Meets NeRF'. In: *ICCV*. 2023.

Appendices

Appendix A

CLNeRF Renderer Workflow

The rendering workflow begins by launching the navigator renderer, `test_checkpoint_keyboard.py`, which allows the user to freely explore the trained scene and generate frames from any desired viewpoint. During this interactive session, users can also store specific camera poses. These poses are saved to a `.npy` file upon exiting the navigator. The file contains `numpy` arrays representing the camera's rotation matrix and transformation vector at each saved pose.

```
python test_checkpoint_keyboard.py --root_dir {CURRENT_DATA_DIR} --exp_name
↪  {EXPERIMENT_NAME}  --weight_path {MODEL_CHECKPOINT_WEIGHTS} --task_curr
↪  {CURRENT_TASK_ID} (--width {CUSTOM_FRAME_WIDTH} --height {CUSTOM_FRAME_HEIGHT}
↪  --base_output_dir {CUSTOM_OUTPUT_DIR})
```

Listing 11: Command to launch the CLNeRF renderer to generate frames from custom poses and save poses to a .npy file. The `width`, `height` and `base_output_dir` arguments are optional.

Once a set of poses has been recorded, the user can optinally run the custom made rendering script, `test_checkpoint_custom_poses.py`, to just generate the renderings directly from these predefined viewpoints through the `poses_path` argument without needing to reopen the navigator.

```
python test_checkpoint_custom_poses.py --root_dir {CURRENT_DATA_DIR} --exp_name
↪  {EXPERIMENT_NAME} --weight_path {MODEL_CHECKPOINT_WEIGHTS} --task_curr
↪  {CURRENT_TASK_ID} --poses_path {PRECALCULATED_POSES_FILEPATH} (--width
↪  {CUSTOM_FRAME_WIDTH} --height {CUSTOM_FRAME_HEIGHT} --base_output_dir
↪  {CUSTOM_OUTPUT_DIR})
```

Listing 12: Command to launch the renderer of CLNeRF to generate frames from precalculated custom poses. The `width`, `height` and `base_output_dir` arguments are again optional.

This rendering workflow was used throughout our CLNeRF experimental evaluations. It ensured that we were able to render from identical viewpoints at each stage of the model's training, enabling consistent side-by-side visual comparisons to assess reconstruction quality across time. This significantly improved both the quality and efficiency of the rendering process, as it eliminated the need to manually navigate to the exact same viewpoint for each render.