Studying the Geometric Convergence Behaviour of Neural Radiance Fields for Improving Training Time

---

Master's Thesis of Berk Kıvılcım

A thesis submitted in fulfillment of the requirements for the Master Thesis

at the Department of Civil Engineering, Geo and Environmental Sciences

Institute of Photogrammetry and Remote Sensing (IPF)

Reviewer: Dr. Martin Weinmann

Second Reviewer: Prof. Dr. Stefan Hinz

Supervisor: Assist. Prof. Dr. Michael Weinmann

Second Supervisor: M.Sc. Dennis Haitz

Submission Date

02.05.2024

Abstract

This study presents an innovative enhancement to Neural Radiance Fields (NeRF) leveraging the Nerfacto method, focusing on voxels and their analytical evaluation to dynamically stop the training process based on geometrical convergence gains from iterations and mask the pixels where the depth information of rays intersect with converged voxel regions during the training. Unlike traditional NeRF approaches that rely on a predetermined number of iterations, presented methodology introduces an adaptive strategy that terminates training when overall 3D geometric convergence becomes sufficient, thereby conserving time and computational resources. Because, each different structure or scenes' 3D reconstruction process will require varying amounts of iterations depending on the context. The main idea is, by dividing the scene box into grids at designated iteration steps and generating voxels to store density values, a method is devised to assess training progress by comparing the differences between sequentially created voxels. In addition, effects of decreasing the number of the samples during the training process according to that overall geometric convergence is investigated with respect to time and quality. Moreover, in each batch, the intersections between the samples that represent where the objects located and the voxels were calculated. Based on this, a decision was made on whether to exclude those rays from the training process in subsequent iterations. This masking approach aims to focus on unlearned areas, enabling it to prevent some amount of divergence in training.

Keywords: Neural Radiance Fields; NeRF; Deep Learning; Computer Vision; Nerfstudio; 3D Reconstruction; Training Time Improvement

Contents

1. Introduction

Fast and accurate 3D reconstruction is longstanding problem which is fundamental to many applications such as navigation, robotics, game and animation industry, medical diagnoses and so on [18]. A recent breakthrough in the field of novel view synthesis and 3D reconstruction is the Neural Radiance Fields (NeRF) [3]. NeRF employs deep learning to represents 3D scenes based on the over-fitting of a scene to observations like photographs [1]. Due to NeRF's ability to produce high-quality results, this capability is valuable in many of those fields [20]. Since the introduction of NeRF technology, various NeRF-based models have been developed to operate faster and yield more accurate results [7]. Tremendous amount of these methods are based on voxel grids to provide improvements for accelerated convergence [8], faster renderings [9, 10, 11, 16], better sampling [4], better learning with limited number of inputs [12], better results with moving objects [13], 3D segmentation [14]. Therefore, using the voxel grids in NeRF is not a new task and the usefulness of voxels has been proven in many studies. However, whether or not voxels are used, current NeRF research has not sufficiently focused on tasks such as early training termination or pixel masking during the training.

The main objective in this study is to create a reasonable accuracy-time tradeoff by terminating the training early, once the 3D-models reach a low geometric convergence gains from iterations. When an adequate convergence is satisfied, further training iterations provides low benefits. This approach aims to avoids these further training iterations by stopping the training, thereby significantly reducing total training time.

Additionally, another aspect that is implemented in this study involves masking pixels that represent sufficiently converged objects. The goal is to stabilize the model's random stochastic learning process and divergence rate. This approach seeks to refine the efficiency of the training process by focusing on areas requiring further learning, thereby potentially enhancing the overall performance and accuracy of the model.

Furthermore, during the learning phase of the model, certain iteration steps were observed and interpreted to understand which feature structures were learned earlier and which ones were acquired later in the training process. This examination aimed to gain insights into the learning dynamics, providing valuable information on learning to assist to define proper parameter selections for determining voxel creation steps. Throughout these processes, the implementation builds upon the Nerfacto method [26], as it is considered as one of the state-of-the-art approach which integrates ideas from multiple research papers such as MipNeRF [24] and Instant-NGP [23].

Besides, the effects of reducing the sample size during the training process by using the informations from voxel is tested to see trade off between time and quality since a study indicates, with a larger number of samples the rendering should get better but becomes computationally prohibitive [2]. The detailed explanation of the code implementation is elaborated upon in the appendix.

2. Related Work

A recent study updated in early 2024, provides a comprehensive overview of previous NeRF research and advancements [7]. In addition to that, a Github page provides a categorized bibliography about NeRF papers and surveys [29]. It is clearly seen that, utilizing voxel approaches is a widespread method and has been employed in numerous articles [4, 8, 9, 10, 11, 12, 13, 14, 16]. However, as previously mentioned, methods developed with NeRF have been used for various purposes but have lacked focus yet on issues such as early termination of training or pixel masking according to geometric convergence. In this study, an innovative approach is presented that leverage voxels to overcome these unaddressed issues by achieving good results with short-term training.

Unlike some of the previous studies, a hierarchical voxel structure such as an octree is not implemented. Because, the presented methodology is based on comparison of the two different voxels from different iteration steps to do convergence analysis and these voxels should remain in the same structure for a fast and proper comparison.

A Previous study have leveraged voxels to achieve super-fast convergence which directly enhance training time [8]. In contrast, this study involves indirect improvement on the training time by stopping the training once sufficient convergence is achieved.

Although some studies have used the term "early stopping" in their publications [9, 17] but the context in which they use this term differs from ours. Our study employs "early stopping" specifically to terminate training before a selected number of iterations is reached. According to previous study [9], the neural network is used not for predicting RGB values but for estimating spherical harmonic values, and training is stopped early before full convergence is achieved. Subsequently, a voxel is produced within a hierarchical structure that holds spherical harmonics at leaf nodes by using this early-stopped trained model and a threshold value. This voxel then utilizes spherical harmonic values to predict RGB values for specific ray directions, facilitating rapid rendering. Additionally, octree data structure that represents voxel is optimized through fine-tuning of the neural network. Hence, the training is not halted based on convergence, and the voxel still requires further optimization with fine-tuning in this study.

Another article [17], uses the term "early stopping" to indicate that if a sample point should proceed through the next level of recursive network to continue on training or should give an output result from current level of detail stage. The early stopping mechanism continues until all sample points, including those with the finest details, drop below a certain level of uncertainty. This means that if even a small region fails to achieve sufficient convergence, the training persists until the entire model converges down to the finest detail. In contrast, our method involves performing a geometric convergence analysis on the overall 3D constructed voxel model, and if the model achieves good geometric convergence generally, the training process is terminated. This approach eliminates the need to wait until all the finest details are converged.

In that recursive network strategy [17], sample points with different complexities are using different levels of detail in the network. To decide whether a point should proceed to the next level of detail layer and continue its training, the amount of uncertainty at that point is examined. This uncertainty is quantified using a loss function value calculated from the difference between the ground truth pixel and the generated synthetic pixel. Points that represent simple regions produce low uncertainty values compared to others, so they are processed using the initial stages of the recursive network and give output instead of progressing to later stages. Conversely, sample points belonging to more complex scenes maintain high uncertainty, which results in these points being trained for longer durations in later layers of the recursive network. This method achieves more photorealistic views for complex viewpoints. Therefore, the aim of this approach is similar to what is desired to achieve with pixel masking method by excluding some pixels at an early stage of training. However, this recursive approach involves a clustering algorithm such as k-means. The introduced methodology in this study offers a masking method for local individual pixels while early stop mechanism depends on overall geometric convergence.

3. Methodology

Before explaining the methodological details, some fundamental principles about the NeRF is explained to provide an overview, especially for the concepts such as "densities" since the methodology of this study uses voxels based on density values of specific samples or other concepts like "depth calculations" which is used by ray exclusion method. Since the methodology of this study is based on Nerfacto method, the pipeline structure of Nerfacto is illustrated in Figure 1.

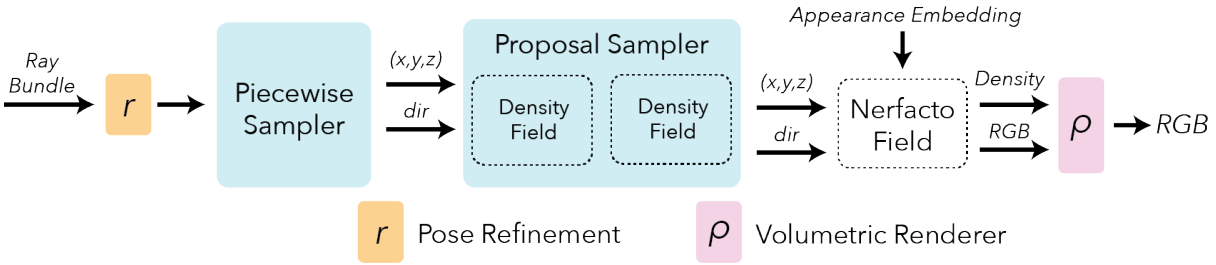

Figure 1: *An overview pipeline for Nerfacto [32]. Piecewise and Proposal Samplers are explained in detail in section 3.4, Nerfacto Field demonstrates the deep neural network, Volumetric Renderer is used to compute RGB values of synthetic images explained in section 3.1*

3.1 Important Fundamental Principles of NeRF and Nerfacto

The primary objective of NeRF is integrating the ray tracing and volume rendering techniques with neural network approaches which is trained by the difference between the ground truth images and synthesized images. This involves ray generations from pixels and point sampling along these rays. These sample points have XYZ coordinates in 3D Space and each ray has direction orientation values. Those established XYZ values and directions are used as inputs to a neural network that predicts the density and RGB values of these sample points. The RGB values represent the color of that point and the density value signifies the opacity of that point. For instance, if a point is associated with a location in vacant space, its expected that the density value should be minimal and it will not be perceptible. However, if a point is situated on an object, its density value will be high which means the point RGB values will be more visible. These RGB and density values from samples along a ray are used to compute a synthetic RGB value for the pixel that constructs that ray. The loss function is then calculated based on the difference between these synthetically generated pixels and the ground truth pixels, allowing the model to backpropagate for minimizing the difference between reference images and synthesized images [1]. The illustration of a ray, sample points and image pixel given in Figure 2.

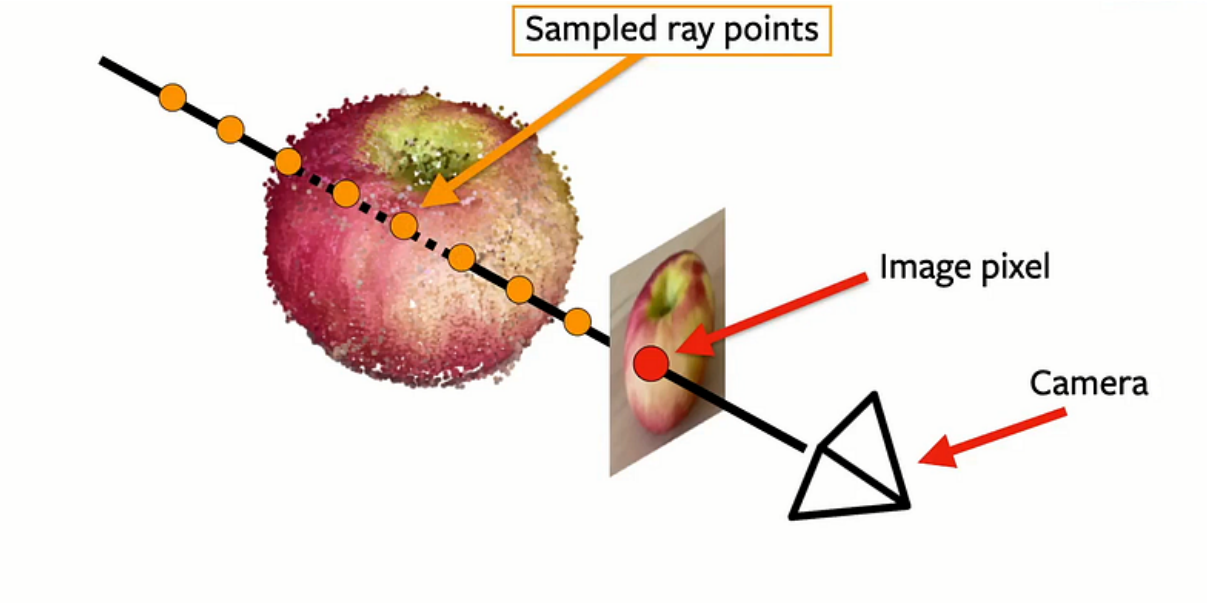

Figure 2: *A visual illustration of the relations between image pixel, ray, samples on ray and 3D reconstructed object [31]*

It is crucial to note that the density values are predicted by only using XYZ values as an input for MLP. Therefore, it is also sufficient to use only XYZ values to produce voxels that represent density fields. However, the RGB values are predicted by using directions and positions together as an input. The significant benefit of this approach can model situations where a point at the same 3D coordinate appears differently when viewed from different directions. Besides, the density values still directly influence the synthetic (rendered) image results. That means the RGB and density output values together have an impact on the pixel colors of the rendered image. Therefore, the learning procedures of MLP with back-propagation aims to achieve the minimum amount of loss with the most accurate RGB and density output values.

For RGB image and depth generations, the core logic of alpha composition (alpha blending) is utilized during the rendering process. Alpha composition is used in visual content creation to combine two or more color values based on transparency information. For example: given two color values C_1 and C_2 and their respective alpha values α_1 and α_2, the resulting color value C_{out} can be calculated as $C_{\text{out}} = \alpha_1 \cdot C_1 + \alpha_2 \cdot C_2$. In conclusion, alpha values indicate transparency probabilities. Alpha value 1 means the pixel completely covers the pixel behind it. With respect to NeRF, alpha values are determined by using density and sampling distance values while color expressed with RGB values. The pixel color values in rendered image calculated by;

$C_{\text{out}} = \sum_{i=1}^{N} \alpha_{\text{i}} \cdot C_{\text{i}}$ where N indicates total number of samples on a ray and $\alpha_{\text{i}} = 1 - exp(-\sigma_i \cdot \delta_i)$; where σ_i represent density and δ_i represent sampling distance.

Besides, the Nerfacto model allows us to use another different method for depth rendering. It is possible to use the "expected" method which is similar to how the RGB images rendered that explained above or the "median" method which the depth value is set to distance of specific sample where the accumulated weights reaches 0.5. Weights are normalized version of density values along a ray. Summation of all the weights of samples along a ray gives 1. Since the median method is initially defined one for Nerfacto, this study sticks to this median depth calculation method and also the ray exclusion algorithm based on this depth information of median method.

3.2 Voxel Creation

Training with NeRF methods requires exterior orientation parameters, which includes the camera's position and orientation information. Given that Nerfstudio allows to download various reference datasets with known camera poses [28], preliminary examinations were primarily conducted on these datasets. By using the near and far plane parameters and those exterior orientations, Nerfstudio first generates rays then creates samples along those rays. Consequently, each sample has their own XYZ coordinates. By using these points Nerfstudio constructs an axis aligned bounding box. To create that box, minimum and maximum XYZ coordinates are used. The coordinate system of this scene-box ranges from -1 to 1 in each of the XYZ directions. For instance, the point with the lowest X coordinate corresponds to the -1 value on the X-axis within the scene box, while the point with the highest X coordinate corresponds to the +1 value. Consequently, the central coordinate of the scene-box is established at (0,0,0).

For constructing the voxel grid, firstly an uniform sampling is employed within the range of -1 to 1 to generate XYZ coordinates of samples based on the "specified voxel resolution+1". For instance, with the voxel resolution of 300, 301^3 uniformly distributed points are created in scene box. Then those points are shifted with a half voxel element size in each directions to derive center coordinate points of the each voxel elements. After this shifting process some points located outside of the scene box. Those points are eliminated and the final point sampling size become 300^3 where those points represent the voxel elements center coordinates. This process resulted in a total of 27 million samples with resolution of 300.

Feeding all these coordinates into the model to obtain density values for each of these points only took less than a second, even though for that 27 million points. However, utilizing higher resolutions posed memory challenger on the RTX 3090 GPU. Therefore, further experiment are conducted with the voxel resolution of 250. Additionally, a resolution of 128 still yielded satisfactory voxel outcomes, but resolutions below 128 began to exhibit noticeably coarser results. Some other studies used the voxel resolution as 384 [4]. After all these uniformly distributed point creation steps and density calculation of each point, a numpy array [35] is created. Whole process of creating the voxel fields are illustrated in the Figure 3.

For the voxel data visualization task, a linear stretching normalization method is implemented to define the transparency values during the visualization. That means, the voxel with the highest density will be totally visible while the density with the lowest value can not be perceptible. However, firstly a top 1 percentage of the highest density values setted to next highest values. Because during the training, some voxel elements' density values are constantly increased for some datasets such as "storefront data" which fools the normalization. The visualization of some results with different voxel resolutions given in Figure 4. The gain from resolutions higher than 128 loses its significance but still using the higher resolution is recommended due to low time consuming of voxel creation steps and also resolution directly influence the ray exclusion method.

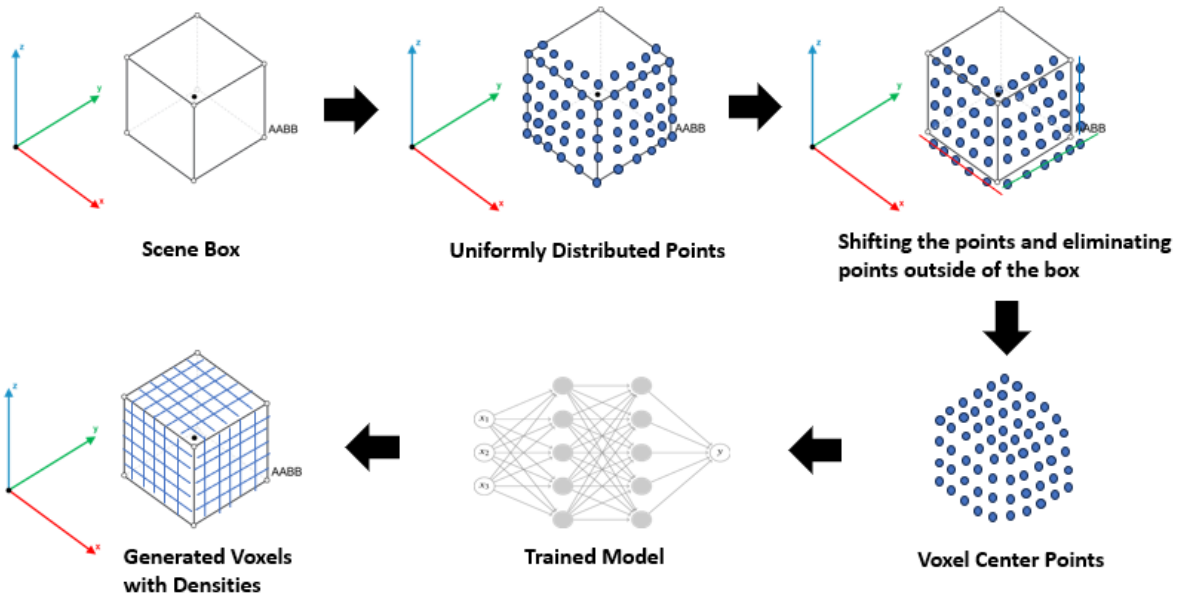

Figure 3: *The process of creating the voxel fields by using uniformly distributed samples in a scene box*

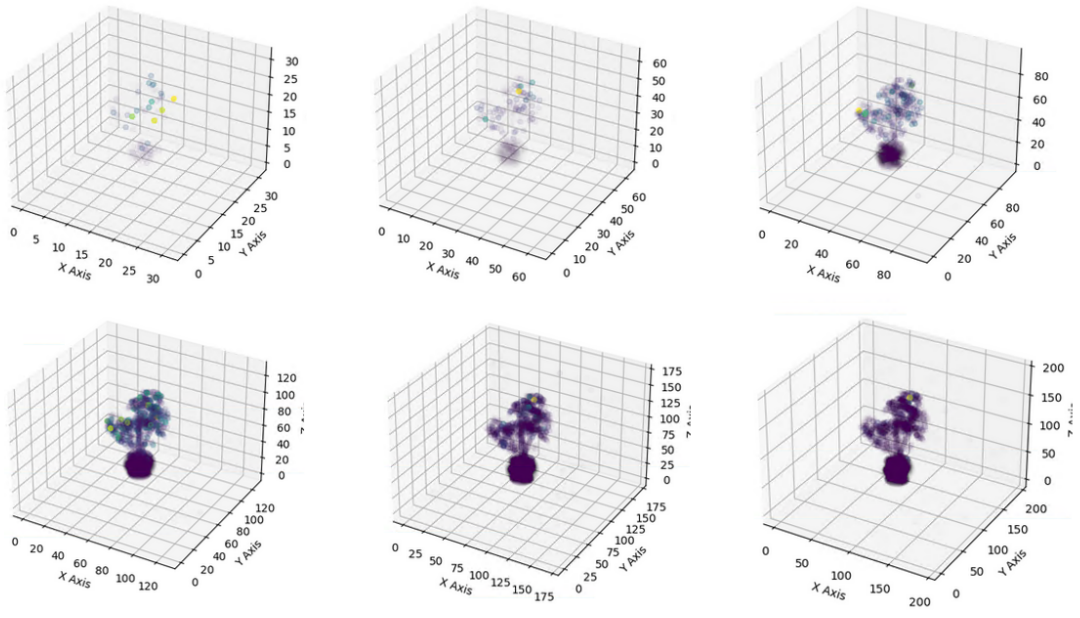

Figure 4: *From top, left to right, the resolutions are 32, 64, 96 and on the bottom 128, 170, 195*

3.3 Analysing Convergence and Implementing An Early Stop Mechanism

For terminating the whole training process, the overall geometric convergence gains from iterations should be analysed. Therefore, the methodology is based on the differences between generated voxels. The first task is finding the most optimal voxel creation steps for this task.

3.3.1 Decision of voxel creation steps

Given the method's focus on the differences between voxels, it is anticipated that a certain degree of differences between two sequential voxels if convergence gains from the iterations is still high. However, the amount of change expected as minimal for consecutive steps (such as between step 5 and 6) and it is not expected to see much total progress in learning during the very first iterations. That is one of the main reason why a first step for the voxel creation is decided. Following this, a sampling is performed between the initial voxel creation step and the defined final iteration step. In this context, a logarithmic sampling is opted, rationalized by the premise that the huge amount of convergence occurs in the early phases, thus necessitating denser sampling initially and gradually decreasing the frequency for optimal outcomes. For instance, it is recommended to start at step 100. The Figure 5 supports this proposal. In that figure, it is obvious that firstly an overall color over all structure changed in point cloud while it remains the cubic structure of it. The coarse structures become visible after the step of 100.

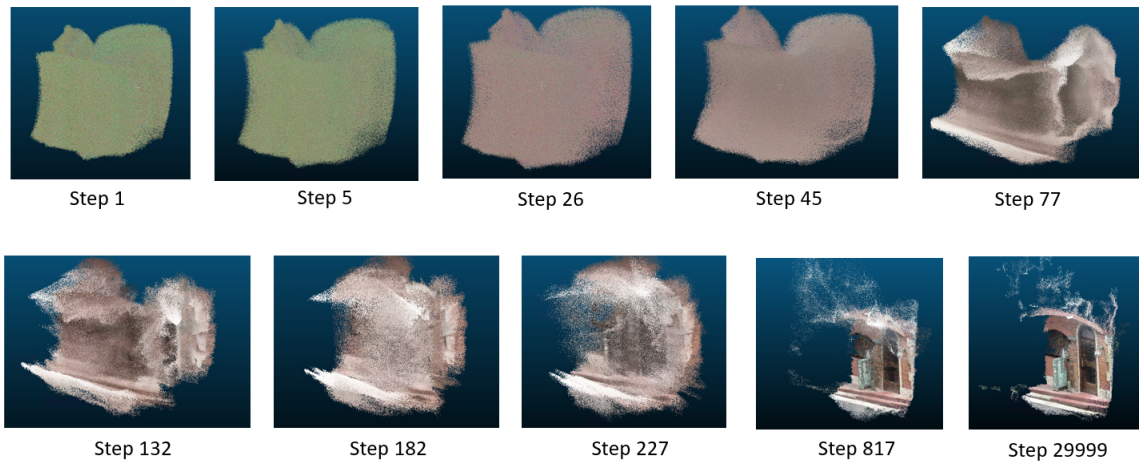

Figure 5: *Storefront dataset point cloud results at certain steps*

Same phenomena occurs with the other datasets such as poster, ficus, lego. Firstly color over whole image is changed. When the training process reach the step of 100, coarse and thick structures starts to become distinctive and details evolves to finer scale in further steps. This phenomena can be seen in Figure 6.

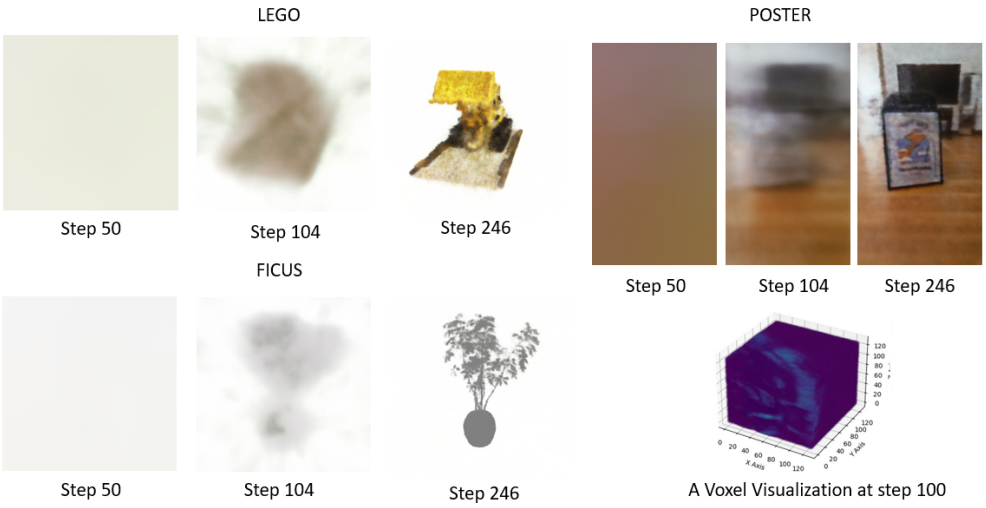

Figure 6: *Lego, ficus and poster rendered RGB results and a voxel visualization at step of 100*

3.3.2 Voxel Difference

In this study, initial voxel creation step selected as 100, with the final iteration numbered at 30.000, and chose to generate a total of 20 voxel samples. After applying a logarithmic sampling, the steps of where the voxels were produced as follows: [100, 135, 182, 246, 332, 448, 605, 817, 1103, 1488, 2009, 2712, 3661, 4942, 6671, 9005, 12156, 16409, 22150, 29999]. For instance, after the second voxel was produced at step 135, analyzing the differences began. Initially, to filter the outliers, the top %1 of voxels with the highest values adjusted to replace them with the next highest value and then normalized these to scale between 0 and 1. Subsequently, the absolute difference between the densities of step 135 and step 100 is examined. This process continued by comparing the absolute differences between successive steps, such as between steps 182 and 135, and so forth. This approach allowed us to observe and quantify the progression and significant changes in voxel densities throughout the training process. In Figure 7, it is clearly seen that the outer part of the ficus data learned between the steps of 135 and 100, while the inner part is learned between 182-135. The convergence precission between the steps of 448 and 332 decreases and becomes more random around the object, while the randomness increased between the steps of 29999-22150. For the illustration method in Figure 7, only the top highest %0.1 of values visualized in difference voxels.

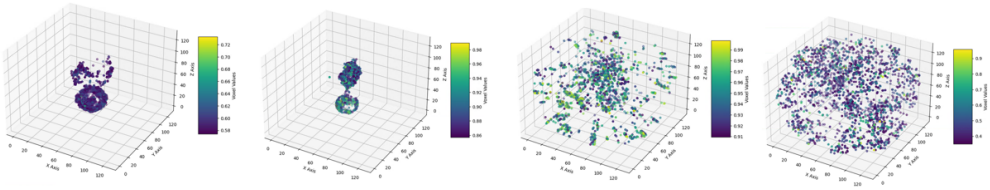

Figure 7: *Visualization of the differences between voxels, sequentially from the left: steps 135-100, 182-135, 448-332, and 29999-22150 (with voxel resolution of 128).*

The most significant finding is that, across various data types (lego, ficus, poster, library, storefront), as the training progresses, the histogram of differences between voxels begins to cumulate around the value of 0.0. This phenomenon used for early training termination. The Figure 8 represent this behaviour with histograms.

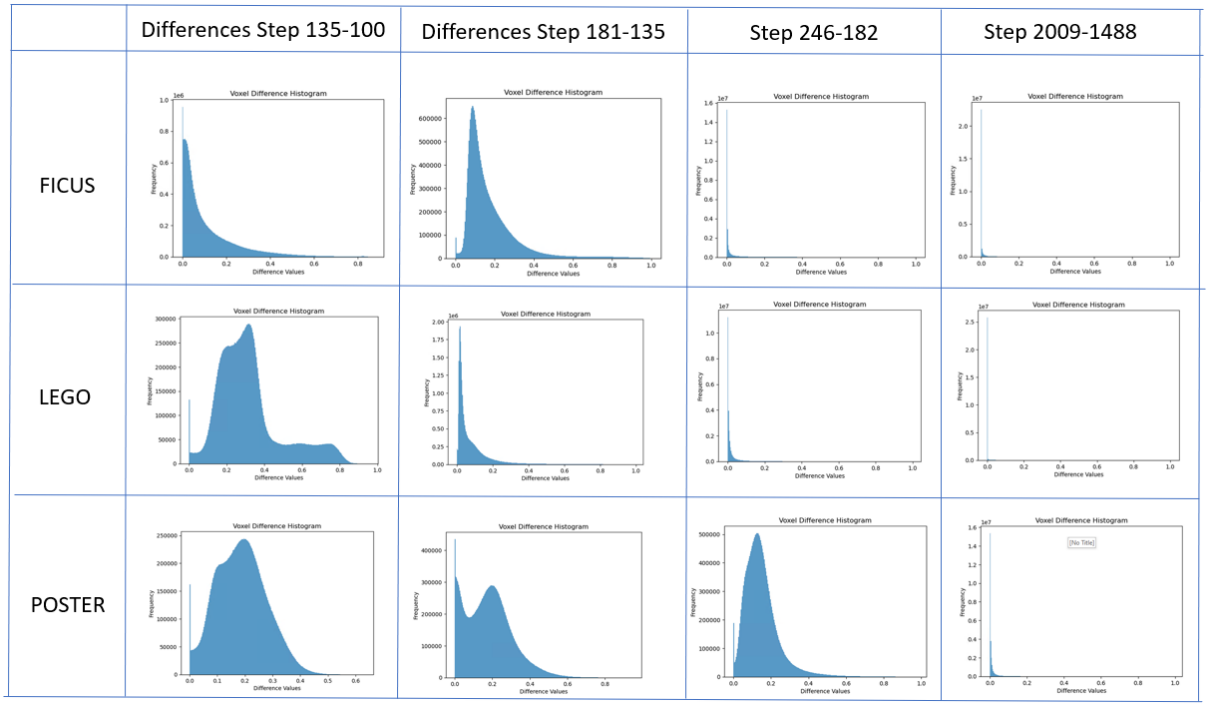

Figure 8: *Histograms of voxel differences at certain steps with voxel resolution of 300*

3.3.3 Early Stop

As training progresses, the histogram of differences between voxels starts to accumulate towards 0.0. Due to this behaviour, it is decided that if the median value of the histogram drops below 0.0001, the training is terminated. The median is used here because random points with high differences values still may exist, making the average a poor measure for overall assessment. The threshold of 0.0001 is determined as an optimal value through trial and error; it can be lowered for more precise results or raised for quicker, albeit coarser, outcomes. In the training, the poster data reached this condition by step 12.156, while the library dataset did so by step 6.671, lego data varied, mostly stopping as step 2.009 or sometimes at 1.488, ficus data by step 2.712, and the storefront data did not meet this condition yet by the final iteration of 30.000 steps. Moreover, the early stop iteration results remained same whether the voxel resolution is set at 128 or 300. However, in the ray exclusion method, resolution can affect the precision of the ray masking which is based on the voxel data's resolution. For this reason, and because voxel creation is a quick process, it's recommended to use the highest possible voxel resolution as allowed by the computer's performance and memory capacity.

3.4 Testing to Decrease the Sample Sizes During Training Process

During the sampling stage, initially, piecewise sampling with N samples is implemented Piecewise sampling increases the step size between points in distant regions of the scene and ensures a denser formation in closer regions. After the RGB and density value determinations of those coarse samples according to piecewise sampling, a probability density function is constructed by using the density values. Based on this function, a finer sampling is performed on specific regions to predict RGB and density values of finer scale. The goal is to achieve more accurate models by performing higher samplings in regions with objects. In Nerfacto proposal network sampler is used. The proposal network sampler initialized with an uniform or piecewise sampler then the PDF sampler is iteratively used according to the hyperparameter of proposal iterations. Therefore it is possible to think of piecewise sampler as a coarse sampling and a proposal sampler as a fine sampling. The initial N_{coarse} is 256, and for the fine sampling iteration number is 2 with the N_{fine} is 96 for the first iteration and N_{fine} is 48 for the rest. Those 48 samples per ray are also used for the ray exclusion algorithm since they represent the latest outputs of PDF function which precisely focus on where the object is.

To understand whether reducing the number of samples would be a reasonable strategy, some experiments are conducted. Starting the training with low sampling numbers resulted in poor outcomes as expected but without any visible time gain. Another approach is, beginning with high sampling numbers and reducing them during the training process. Since the highest rate of convergence occurs in the initial steps, it is logical to keep the sampling numbers high initially and use a function that decreases the sampling numbers. The function of $N_{\text{initial}} \cdot \log(1 + 100 \cdot histogram\ mean)/\log(1 + 100)$ used, which maintains high sampling for mean values of difference voxel histogram between 0.1 and 1 but decreases rapidly between 0 and 0.1. For example, the coarse sampling started at 256, and when mean value dropped to 0.18, the sampling rate reduced to 166. The sampling number fell to 15 when mean reached 0.0002. Applying this idea to both fine and coarse sampling, poor results are observed. This was due to the significant information contained in fine samplings even in later training steps. As another experiment, only the coarse sampling numbers reduced throughout the training. The results were not drastically poor but did slightly worsen the training scenario. In that scenario the training for the poster dataset did not end at step 12156 but instead at 16409, which was contrary to the computation time performance expectations. Besides, influence of reducing the sample numbers was almost no impact and negligible regarding to training speed. The graph of used formula for sample reducing and some visual results illustrated in Figure 9.

Figure 9: *Right: the graph of used formula for decreasing the sample numbers, Mid: Decreasing the number of coarse and fine samples over the training, Right: Decreasing the number of coarse samples over the training*

Hence, using the highest possible number of samples appears to be the most sensible strategy. However, significantly increasing the sample size, leads to memory issues just like the stage of voxel creation. Thus, employing to the highest level of sampling that memory constraints allow seems to be the best approach.

3.5 Ray Exclusion with Pixel Masking

The next task is to determine which voxel contains the object that the pixel sees. For each pixel, in other words rays because rays are generated from pixels, median depth method used which is described in section 3.1 to calculate depth information of pixels. The weights of 48 samples obtained from the last iteration of proposal network for each ray, as well as the XYZ coordinates of those samples within the scene box's coordinate system in range of [-1,1] are used. The weights are normalized version of density values. The total summation of those weights gives the result of 1.

For instance, if batch size selected as 4096, then in each iteration 4096 rays are used from 4096 pixels. This gives a tensor dimensions of (4096,48,1) for the weights and (4096,48,3) for the coordinates. The cumulative summation of weights are calculated on each ray. A specific sample where the cumulative summation reaches 0.5 is represent where the object of that pixel sees located in scene box. Since this specific sample represent the location of the object, the operations performed based on this sample, thus creating a tensor of dimensions (4096,1,3) that contains only this sample and its scene box coordinates.

Given that the scene box spans the range [-1,1] and each voxel is uniformly divided, each voxel's size is set as $(1-(-1)/voxel\ resolution)$. This allows to quickly determine which voxel contains the sample, converting the scene box coordinate system into a voxel coordinate system. At this point, the sample's tensor size remains (4096,1,3) but the coordinate system is now in terms of voxel coordinates along the XYZ axes. For example, if the voxel resolution is 300, the range is no longer between [-1,1] but [0,299].

At this point, it is known that which voxel position contains the object that a ray passes through. If the difference value between two voxels at this specific location is below 0.0001, it is assumed that ray is looking at a converged area and excluded from further training steps. To do this, camera indices and pixel coordinates of the ray's corresponding pixel used for masking. The mask that created in the beginning of the training has a size of (number of cameras, height, width). Since it is know for the camera indices and pixel coordinates, the value of 1 changed to 0 for this specific locations in the masking array. These steps are performed for each ray in each iteration.

The illustration of the explained idea above here is given in Figure 10.

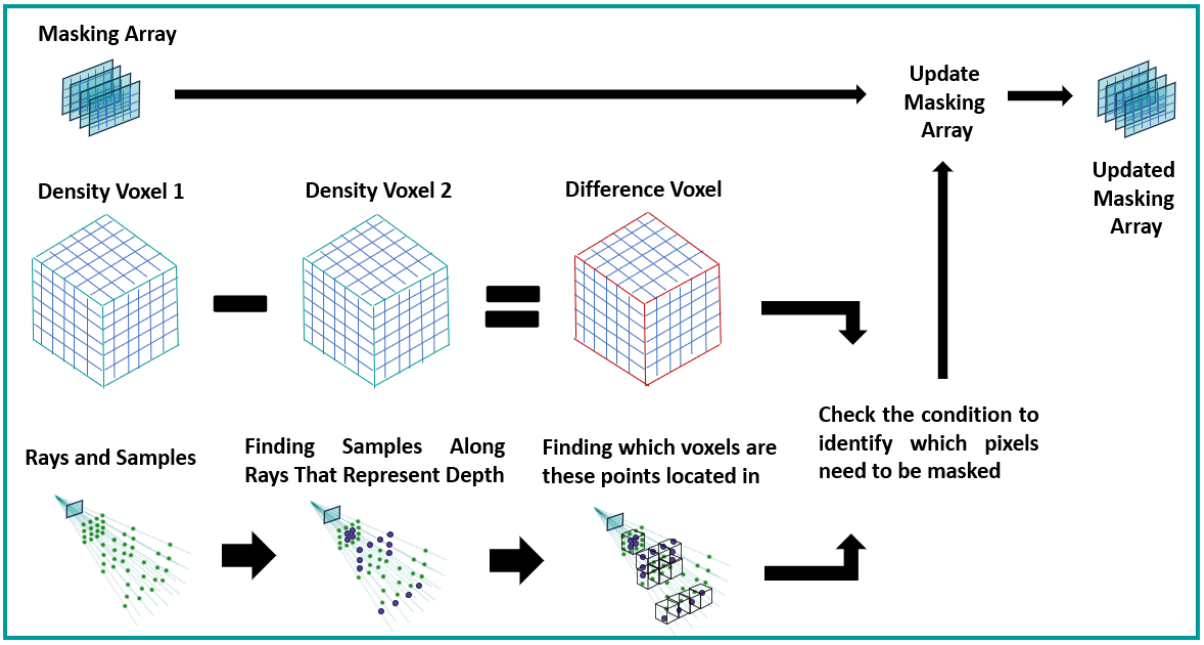

Figure 10: *Workflow illustration of the ray masking principles that uses samples on rays, difference voxel and masking array*

3.6 Configuration Details

The configuration values used in this study are identical to those specified in the "adding new method" section of Nerfstudio [27]. All of the config parameters can be seen in [30]. However, it is important to note that these values differ from the initial configuration parameters of Nerfacto [33]. This distinction is crucial for those looking to compare our work with Nerfacto, as the results obtained with configuration values of [30] have shown to be superior to those of initial Nerfacto. Among the differing configurations are the learning rate for the camera optimizer, as well as the final learning rate and maximum steps for both the camera optimizer and fields in exponential decay.

Exponential decay reduces the learning rate over the course of training, allowing for quick convergence initially while reducing the risk of divergence and deviation from the global minimum in later steps. The effect of decaying learning rate is to improve the learning of complex patterns, and the effect of an initially large rate is to avoid memorization of noisy data [15]. The final learning rate represents the learning rate to be reached after all of decaying process, and max steps parameter in the scheduler indicate the total number of steps over which this exponential decay occurs. In the configuration parameters compared to Nerfacto, the max steps values used for field and camera optimization are lower, resulting in a faster rate of learning rate decay. The formula of decayed learning rate is; $lr = exp(log(lr_{\text{initial}}) \cdot (1 - t) + log(lr_{\text{final}}) \cdot t)$ where $t = (current\ step)/(max\ step)$.

4. Results

This section is divided into three parts. Firstly, the effectiveness of early stopping approach has been evaluated by examining some quantitative and qualitative results. Then, in the following section, the effectiveness of the masking method has been examined especially for long-term training. Finally, the computational overhead of the functions have been analyzed by comparing the learning rate per second values.

4.1 Comparison Between Full Training and Early Terminated Training

In Figures of 11, 12, 13, 14, the qualitative results from where the training terminated at certain steps and full training results (30k iterations) are compared with each other for an object centric scene like poster dataset or outdoor scene like library or blender datasets like lego and ficus. The results of storefront dataset is not included here since it does not meet the 0.0001 median value threshold condition before the 30k iteration but the median value decreased to 0.0002 at step of 30k. This means the behaviour of median value converges to zero in time but the convergence rate is just slower than the other datasets. According to the experiments, the blender datas are terminated earlier compared to real scene datas. For instance, lego and ficus trainings terminated at steps of 2009 and 2712. While poster and library data trainings terminated at step of 12156 and 6671. It is clearly seen that, the most important coarse structures are learned for lego and ficus data but results are not sharp as full training. On the other hand, there is almost no difference seen between full training and early stop for poster and library data according to rendered RGB images and point clouds.

For the quantitative comparisons, PSNR (Peak-Signal-Noise-Ratio), SSIM (Structural Similarity Index) and LPIPS (Learned perceptual Image Patch Similarity) are used. Also, those experiments conducted for several times to ensure their reliabilities. Since the results of several experiments for a same setup almost identical to each other, only the one result from each unique setups are given in the Table 1.

PSNR represents how two images (prediction and ground truth) are similar to each other and expressed in dB. Values over 40 dB can be considered very good and below 20 dB are normally considered unsatisfactory [25]. Higher PSNR means better quality. PSNR is simple to calculate but PSNR does not correctly represent human visual perception quality [19]. The PSNR values of early terminated results could be higher than the results of full trained models and some quantitative experiments also shows that in Table 1. However, this situation should not be the case with a reliable metric.

SSIM [19] measures structural similarities between two images by considering illumination, contrast and structure. Those functions are created by using means, variances and covariances between two images. SSIM provides a metric which is close to human eye perception that makes it an important metric. The SSIM value of 1 indicates two images are exactly the same. 0 means no similarity and -1 means perfect anti-correlation. According to Table 1, SSIM provides more reliable insight but still early stop results may be higher or equal to full training results which could be misleading.

There is another metric known as Learned Perceptual Image Patch Similarity (LPIPS) specifically designed to evaluate the similarities like human visual perception through deep learning techniques to overcome shallowness of PSNR or SSIM [5]. LPIPS divides the images into patches and measures the similarity between those patches with learned weights. If a LPIPS value is closer to 0 two images are more similar to each other. The higher the LPIPS value means less similarity. According to results in Table 1, LPIPS tends to give better results with further training steps. Since better results are expected with higher step numbers and it is also qualitatively observed in Figures 11, 12, 13, 14, the LPIPS method considered as the most reliable among the other metric methods, and further experimentation is mainly based on this metric.

In conclusions according to both qualitative and quantitative results, the early stopped method gives satisfactory results while saving a lot of time by skipping unnecessary training steps. Only the ficus dataset's LPIPS and PSNR values look unsatisfactory but it is seen that in Figure 14, the constructed object part looks satisfactory in both point cloud and synthetic RGB images yet some white colored outlier points around the object exist in point cloud. Consequently, it is approved that the early stop method with a threshold value of 0.0001 works for various kind of content.

Figure 11: *Upper row: Rendered images and point cloud from steps of 12156 where the training is terminated, Bottom row: Qualitative results from full training of 29999 iterations*

Figure 12: *Left Sides: Rendered images and point cloud from steps of 6671 where the training is terminated, Right sides: rendered images and point cloud from full training of 29999 iterations*

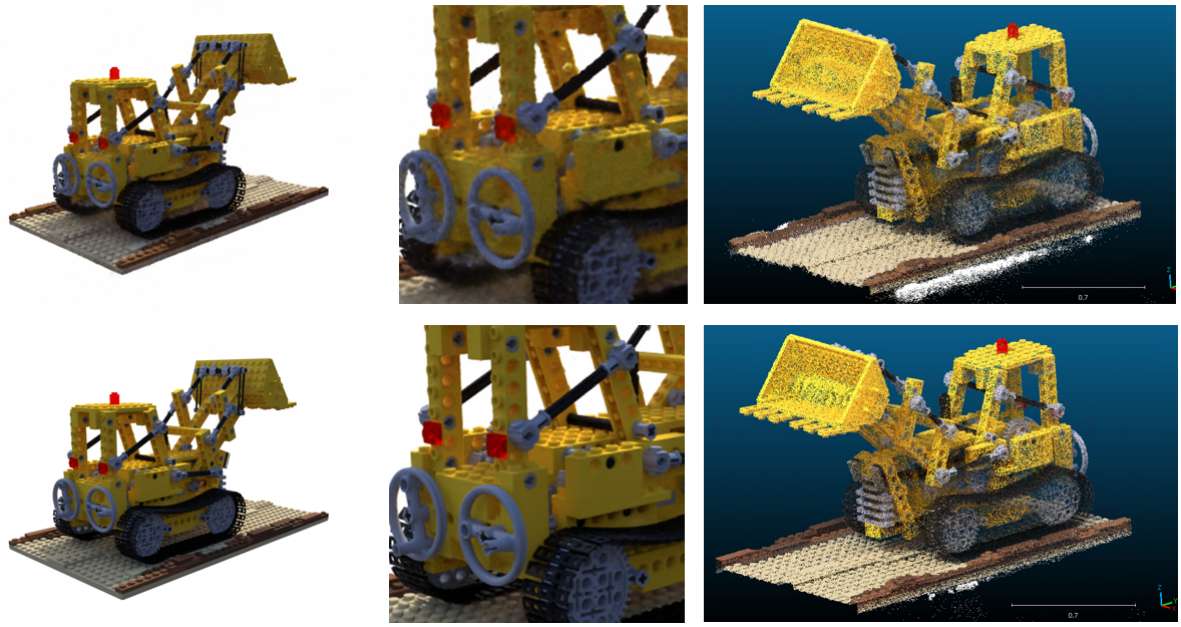

Figure 13: *Upper row: Rendered images and point cloud from steps of 2009 where the training is terminated, Bottom row: rendered images and point cloud from full training of 29999 iterations*

Figure 14: *Upper row: Rendered images and point cloud from steps of 2712 where the training is terminated, Bottom row: rendered images and point cloud from full training of 29999 iterations*

Table 1: *Quantitative Evaluations of Metrics. The first column is Nerfacto with initial configs, the middle column represents orginal Nerfacto method but with the configs of [30], last column represents the results with early stopping mechanism and configs of [30]*

Datasets	Nerfacto (30k)	Nerfacto (30k) [30]	Early stop
Poster			Step 12156
PSNR	21.30 dB	22.91 dB	23.30 dB
SSIM	0.84	0.88	0.89
LPIPS	0.21	0.17	0.19
Library			Step 6671
PSNR	25.08 dB	26.65 dB	25.97 dB
SSIM	0.81	0.88	0.83
LPIPS	0.15	0.10	0.15
Lego			Step 2009
PSNR	22.41 dB	22.45 dB	24.76 dB
SSIM	0.91	0.90	0.89
LPIPS	0.09	0.09	0.10
Ficus			Step 2712
PSNR	14.23 dB	21.49 dB	16.56 dB
SSIM	0.85	0.90	0.83
LPIPS	0.23	0.12	0.28

4.2 Results with Ray Masking

During the training phase, masking specific pixels had a negligible effect for short term early stopped trainings. One reason for that is, the algorithm only checks at several pixels equal to batch size in each iteration and selects among those pixels to eliminate. For instance, with a batch size of 4096, even if every pixel met the condition and masked, in the 12156-step training for poster data, the algorithm may mask at most 49.7 million pixels. This number is likely to be much lower since the condition starts being met mostly in later stages of training also. In example, if the training stopped at step 10k and the total removed number of rays is checked, it is observed that only 12.8 million pixels have been masked. However, the dataset contains more than 200 images with size of 540x960, there are total of more than 100 million pixels exist. Therefore, it is expected that the pixel masking method will be more effective in longer trainings rather than prematurely terminated ones. According to both qualitative and quantitative results, no definitive and visible differences were observed between methods using pixel masking and original methods. The quantitative results given in Table 2.

First, the efficiency of masking method in longer training sessions are tested with voxel resolution of 250. The experiments were conducted with the number of iterations set to 300k. During these extended training sessions, the issue of divergence began to emerge for the poster and library data. However, the pixel masking method provides more stability and manages to mitigate that issue to some extent for the poster data. Qualitative results of the poster dataset given in Figure 15.

Figure 15: *Upper row is from first experiment, Bottom row is from another second experiment, Left Column: Rendered image from iteration step of 300k with ray masking, Mid: ground truth images, Right: Rendered image from iteration step of 300k with without any masking*

For the library dataset, sometimes the masking method provides good results but sometimes the original method outperforms it which means it is still not reliable for the library dataset. One of the main reasons might be that library data still includes some high frequency features and also some regions where the depth is unclear such as the sky. Since the masking method is based on the ideas of depth information and voxels. It is expected to fail in high frequency regions because voxels only provide a coarse approximation.

Visual results for the storefront dataset also not included due to the absence of divergence indications during the 300k training. The reason might be the storefront dataset is constructed with more than 1000 highly overlapped images while the library dataset includes 398 images where some of the regions with lack of overlaps exist and poster dataset only includes 226 images.

On the other side, the pixel masking method gives some inferior results especially for the high frequency blender datasets such as leaf regions in ficus data or tracks of excavator for lego dataset. It is also expected that since the masking method is based on voxels which is an approximation and to achieve good results on high frequency data, higher voxel resolutions might be needed. The Figure 16 shows some results of ficus and lego.

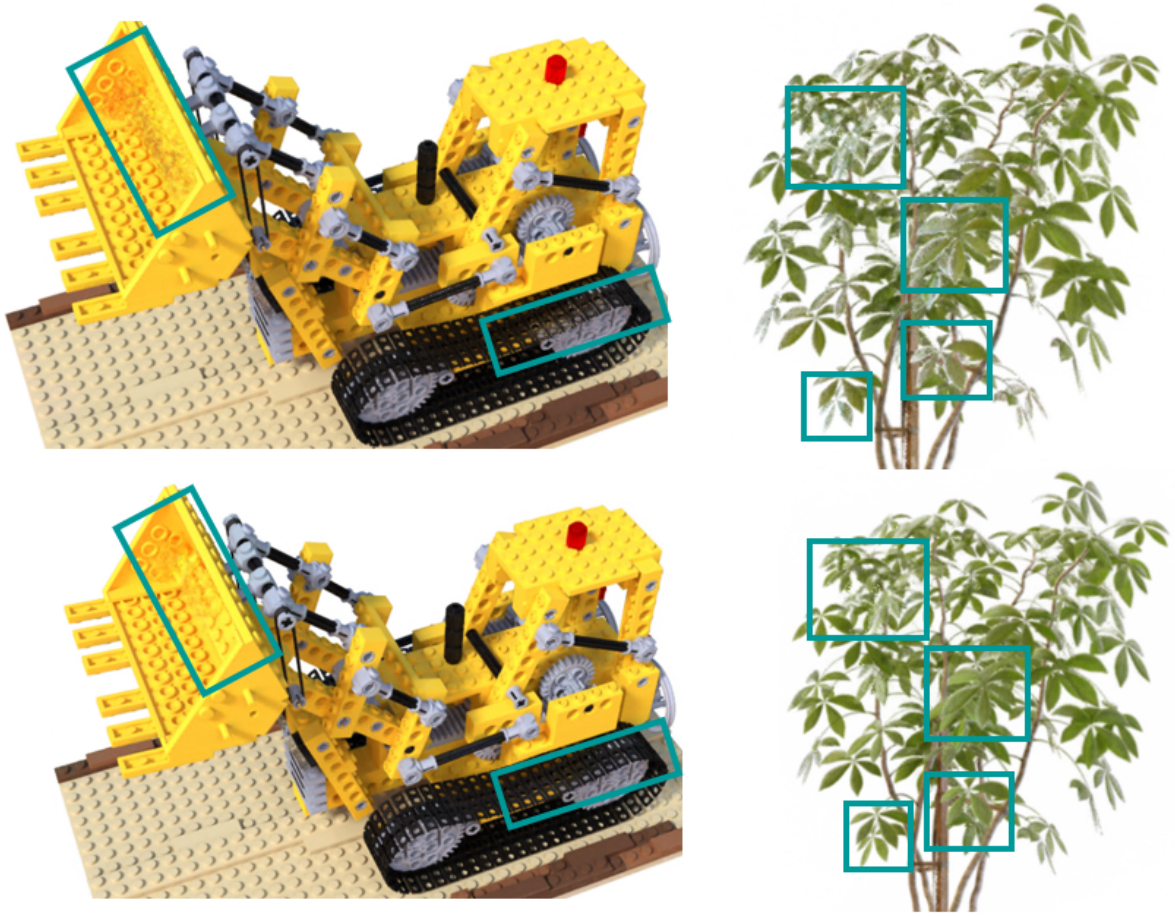

Figure 16: *Top: Rendered image from iteration step of 300k with ray (pixel) masking algorithm, Bottom: Rendered image from iteration step of 300k with without any masking*

Another experiment involved masking a 3x3 patch around each masked pixel. The main purpose of this is to accelerate the pixel exclusion processes with compromising some details. Still no significant visible changes were observed in the results, but LPIPS values for the poster and ficus datasets slightly decreased that can be seen in Table 2.

Table 2: *Quantitative Metric Evaluations of Masking Method for different datasets and setups. The unit of PSNR given in dB.*

Datasets	Early Stop	Early Stop + Masking	Masking with 3x3 Patches	300k Training	300k Training with Masking
Poster					
PSNR	23.30	22.90	22.81	21.79	21.59
SSIM	0.89	0.88	0.88	0.88	0.88
LPIPS	0.191	0.190	0.196	0.165	0.175
Library					
PSNR	25.97	25.95	25.98	26.59	26.52
SSIM	0.83	0.83	0.84	0.89	0.89
LPIPS	0.153	0.154	0.154	0.089	0.087
Lego					
PSNR	24.76	24.95	24.97	20.03	21.15
SSIM	0.89	0.89	0.89	0.87	0.88
LPIPS	0.10	0.10	0.10	0.13	0.11
Ficus					
PSNR	16.56	16.27	16.40	20.66	20.32
SSIM	0.83	0.83	0.83	0.87	0.88
LPIPS	0.28	0.28	0.30	0.15	0.14

4.3 Execution time

Table 3 represents the total number of training rays that processed per second with different setups [36]. Higher that value means faster the training. For example, the column of "with voxel and ray interaction function" means; the voxel-ray interaction is analysed and the masking array is updated but that masking is not considered in training. Column of "with masking" indicates; those mask updating and masking applied together using pixel sampler module of Nerfstudio which is highly affected by total number of training images in dataset. For instance, the storefront dataset's training speed decreased significantly to 7k.

Table 3: *The table represents execution speed comparisons with RTX 3090 GPU. The values given in this table shows the total number of training rays that is processed per second [36]*

Datasets	Only early stop	With voxel and ray interaction function	With masking
Poster	100k	75k	63k
Library	100k	75k	55k
Storefront	100k	60k	**7k**
Lego	125k	95k	79k
Ficus	125k	100k	82k

5. Discussion

This study follows three different approaches that use voxels to improve training time. The first approach aims to achieve improvement indirectly by terminating training early, while the masking method is designed to achieve better convergence. Finally, reducing the sampling size seeks to decrease the computational overhead.

Upon examining the voxels, it has been observed that creating voxels is time efficient but memory intensive. Thus, it is most reasonable to use the highest voxel resolution possible, depending on available memory.

It is observed that reducing the resolution of voxels below to 128 starts affecting the effectiveness of early stop mechanisms. For example, the poster dataset training terminated at step 12156 with voxel resolution of 250. On the other hand, using voxel resolution lower than 128 led to training being stopped at different iterations instead of 12156. However, at resolutions of 128 or above, training consistently stopped at the same iteration step of 12156 for poster dataset despite using varying voxel resolutions higher than 128. This indicated that early stop mechanism stabilized at resolution of 128 or higher. This recommended minimum voxel resolution of 128 works stable with all of the tested library, poster, storefront, lego and ficus datasets but the most reliable solution is to use the highest voxel resolution possible as previously mentioned.

Since the early stop mechanism relies on the difference between two voxels, these voxels are normalized. However, before the normalization, an outlier adjustment in density values is necessary due to divergence issue. For instance, the voxel generated at step 2009 for the storefront dataset, density median value is 0.78 while the maximum density value is 4653. At step 12156, the density median value is 0.30 and the maximum density value is 240546. By step 29999, the median value is 0.14 while the maximum value rises to 7309515. As training progresses, very few voxel in a small region converge to infinity in density values. As seen in Figure 17, without this outlier adjustment, the visualization of a voxel shows that density values in a small region diverge to infinity with further iteration steps, making the normalization of all remaining voxels ineffective. At step 29999, all remaining voxels become transparent and imperceptible as the diverging voxel normalization value close to 1 while rest of the others normalization close to 0. Also seen in Figure 17, if the voxel's top %1 values are replaced with the most next highest value, the normalizations performed well and the structures of the storefront data reveals as it should.

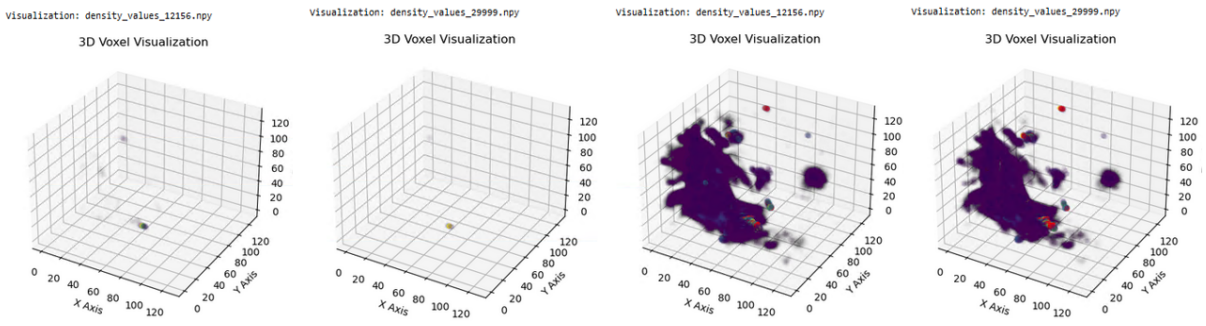

Figure 17: *First and second images illustrate the voxel visualization of storefront data produced at steps of 12156 and 29999 without any outlier adjustment. Third and fourth images represent the same setups but with outlier adjustments. The adjusted outlier voxel regions are also visualized as red in third and fourth images*

After these outlier adjustments and normalization processes, the differences between two consecutively generated voxels are calculated with a simple subtraction operation. Also, using more complex 3D feature structures like linearity or planarity is considered before the implementations but calculations of these structures bring additional computational costs. To do a 3D structure tensor analysis via using Python, a previous study introduced a package for this purposes [22]. Moreover, as neighboring voxels with low density values that represents vacant space may still exhibit linear or planar behaviour, that means filtering out these low density regions would also be necessary to use 3D features and for filtering with global density thresholds usually requires more empirical investigations [21]. Since subtraction between voxels provides satisfactory analysing results and is straightforward computationally, using the more complex 3D features deemed unnecessary and ineffective.

Nevertheless, an important detail needs to be considered is the selection of the voxel creation steps to use this introduced subtraction idea properly. Since the differences between two consecutively generated voxels are evaluated, comparing two closely spaced iterations can be risky. For instance, if the first voxel is generated at iteration 1 and the second voxel at iteration 2, the similarity between these voxels will be high, making their difference insufficiently informative. Thus, as explained in *section 3.3.1*, selection of the first voxel generation step recommended as 100. Additionally, a logarithmic sampling approach, starting densely and becoming more sparse has yielded good results. For example, the difference between steps 100 and 135 shows significant difference despite only 35 iteration steps passed, indicating that the model is converging rapidly at this stage. However, at later stages such as between steps 22150 and 29999, the difference is extremely small despite 7850 iteration steps passed that means convergence speed is significantly decreased and the gains from each iteration is extremely low. This highlights the effectiveness of early stop mechanism and underscores the importance of selecting proper voxel generation steps.

After the difference between two voxels is calculated, if the median value of the difference voxel falls below 0.0001, training is stopped. This global density mean threshold value determined with empirical trial and error investigations. Using the median provides more reliability compared to the mean. Because the difference voxels range normalized to 0 to 1 and the chosen threshold of 0.0001 can be considered relatively low which means just some few outliers with high values close to 1 can easily influence the early stop decision with mean metric. As a result, the mean metric may not always provide a consistent decrease over the training. However, the median metric provides a steady decline towards zero over the training which demonstrates a consistent information for geometric convergence.

Another finding about this 0.0001 median threshold is, the training stops much earlier for blender datasets compared to real world scenes. This situation resulted in minimal but noticeable differences between the early-stopped and fully trained models for blender datasets. On the other hand, in real world scenes such as poster and library datasets, no noticeable differences are observed between early stopped training and full training. Additionally, for the ficus dataset, the LPIPS and PSNR values do not entirely look satisfactory, though qualitative results appear to show the coarse and important structures constructed successfully. In this context, different threshold values for Blender and real world scenes can be considered separately with further research to find more optimal threshold values.

Besides, some previous studies indicates that, LPIPS metrics that match with human similarity judgments are susceptible to such imperceptible adversarial perturbations [6]. Even the one pixel change fools the metric a lot which is shown in Figure 18. Also, this interpretability challenges of NeRF is still a well known issue and indicated in the previous review papers [20].

Figure 18: *Robustness experiments of LPIPS from recent surveys [6]*

The masking method did not result in faster convergence contrary to expectations. By using the masking method, the steps of when the model early stopped remained the same. Additionally, as the masking method adds more computation due to voxel-ray intersection calculations. Besides to that, the masking process increases the memory and computational consumption according to the number of training images which makes the masking method unsuitable for improving the training time. Furthermore, since the masking method only shows its effectiveness at later stages of training, its combination with the early stop mechanism is impractical. Nonetheless, it is observed that the masking method stabilizes training in the long-term, decreasing the divergence for the poster dataset. However, this benefit diminishes with more complex and high-frequency datasets such as in the leaves of the ficus dataset and in ambiguous depth areas like the sky region in library dataset. This indicates the importance of proper depth calculation methods and higher voxel resolutions.

In addition, point cloud extraction during the training process with the masking method is not possible yet without further optimizations with original Nerfstudio codes. Because, using the masking method requires modifications within the datamanager's .next_train() function, including an additional input parameter "mask". On the other hand, initial "generate_point_cloud() function of Nerfstudio calls the .next_train() function without a mask input parameter to extract point clouds [34].

Another approach which is experimented is reducing the number of samples based on the convergence information from voxels. However, this approach did not yield satisfactory results. As mentioned in *section 3.4*, reducing the number of samples used for piecewise sampling led to minimal accuracy loss but also caused the model to converge more slowly. As a result, the training of the poster data set terminated at the step 16409 instead of 12156 which is not ideal for improving the training time. Additionally, reducing the number of samples did not result in a noticeable speed increase. Besides, even processing tens of millions of samples to obtain density outputs for voxel creation with a neural network only took a few milliseconds that support the statement of the ineffectiveness of reducing the number of samples regarding processing time. Therefore, reducing the number of samples is also not considered as a viable approach to achieve improvement in training time.

6. Conclusion

In this study, voxel techniques were initially employed to prematurely terminate the training process. Voxels provide a coarse approximation, which has proven to be highly effective in overall reconstruction convergence analyses at voxel resolutions minimum of 128 or higher. Importantly, this approach has not resulted in any noticeable performance decline in terms of training speed.

The proposed method has produced satisfactory results for all tested training dataset, despite using a single same threshold and same voxel creation steps for all of them. The rendered images and point clouds obtained from the prematurely terminated training were not significantly different from those of full training, yet numerous unnecessary iterations were skipped. This approach resulted in a training time reduction of up to 15 times, depending on the content of the scene.

Additionally, based on the analyses with voxels, a strategy to reduce sampling numbers throughout the training has been tested. While reducing both coarse and fine sampling numbers yielded poor results, decreasing only the coarse sampling numbers did not lead to significant effects on the outcomes, although there was a slight decrease in quality. However, since almost no time savings were achieved in either case but also convergence speed is decreased, this method has been completely disregarded in subsequent experiments.

Another aspect tested was the ray exclusion (pixel masking) method using the voxel data. However, the masking method adds more computation and it causes negative effects to improve training time. Moreover, this masking method did not contribute to increase convergence speed nor increase the quality in short-term training. Nevertheless, it has been discussed that this method could be beneficial in preventing divergence in long-term training scenarios with less complex and less detailed content, such as poster data.

Based on all these findings, it has been determined that the current most reasonable outcomes in the quality-time consumption trade-off are achieved by implementing the presented early stop mechanism, without decreasing any of the number of samples, and without applying the masking method with current available voxel resolutions.

In conclusion, the developed early stop methodology in this study is simple and effective, can be easily integrated into other NeRF researches, and yields good results even across different datasets due to its adaptive nature. This method also could be highly useful for any real-time NeRF applications where the training needs to be conducted and completed swiftly. Furthermore, voxel differences that represent geometrical convergence changes might help for future researches that study convergence while the masking method might be helpful for divergence studies.

For future researches, different threshold values and voxel creation steps could be examined separately for blender and real world scene datasets to make the early stopping method more effective. Additionally, it remains an open question as to what ranges of quantitative metrics from early-stopped models will be considered acceptable. For the outlier adjustment of difference voxels, the top %1 density values were optimized but to do this outlier optimization more complex and interpretable methods can be applied. Furthermore, as mentioned in the discussion section, the explanation for the divergence of some density voxel regions towards to infinity, particularly in the storefront dataset, remains unclear. Regarding to the masking method, the practicality of it still questionable but with more powerful hardware that can allows to use higher voxel resolutions or using additional data sources such as from active sensors that can guide to better depth calculations, the masking method could be more practical for solving divergence issues in long-term trainings by overcoming the current limitations with these advancements.

Bibliography

Reference Papers

[1] Ben Mildenhall et al. “Nerf: Representing scenes as neural radiance fields for view synthesis”. In: *Communications of the ACM* 65.1 (2021), pp. 99–106.

[2] Relja Arandjelović and Andrew Zisserman. “Nerf in detail: Learning to sample for view synthesis”. In: *arXiv preprint arXiv:2106.05264* (2021).

[3] Albert Ge. *Importance Sampling of Views in NeRF*.

[4] Tao Hu et al. “Efficientnerf efficient neural radiance fields”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 12902–12911.

[5] Richard Zhang et al. “The unreasonable effectiveness of deep features as a perceptual metric”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 586–595.

[6] Abhijay Ghildyal and Feng Liu. “Attacking perceptual similarity metrics”. In: *arXiv preprint arXiv:2305.08840* (2023).

[7] Ansh Mittal. “Neural Radiance Fields: Past, Present, and Future”. In: *arXiv preprint arXiv:2304.10050* (2023).

[8] Cheng Sun, Min Sun, and Hwann-Tzong Chen. “Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 5459–5469.

[9] Alex Yu et al. “Plenoctrees for real-time rendering of neural radiance fields”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 5752–5761.

[10] Sen Wang et al. “VoxNeRF: Bridging voxel representation and neural radiance fields for enhanced indoor view synthesis”. In: *arXiv preprint arXiv:2311.05289* (2023).

[11] Lingjie Liu et al. “Neural sparse voxel fields”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 15651–15663.

[12] Abdullah Hamdi, Bernard Ghanem, and Matthias Nießner. “Sparf: Large-scale learning of 3d sparse radiance fields from few input images”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2023, pp. 2930–2940.

[13] Feng Wang et al. “Mixed neural voxels for fast multi-view video synthesis”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2023, pp. 19706–19716.

[14] Jiazhong Cen et al. “Segment anything in 3d with nerfs”. In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 25971–25990.

[15] Kaichao You et al. “How does learning rate decay help modern neural networks?” In: *arXiv preprint arXiv:1908.01878* (2019).

[16] Stephan J Garbin et al. “Fastnerf: High-fidelity neural rendering at 200fps”. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 14346–14355.

[17] Guo-Wei Yang et al. “Recursive-nerf: An efficient and dynamically growing nerf”. In: *IEEE Transactions on Visualization and Computer Graphics* (2022).

[18] Xian-Feng Han, Hamid Laga, and Mohammed Bennamoun. “Image-based 3D object reconstruction: State-of-the-art and trends in the deep learning era”. In: *IEEE transactions on pattern analysis and machine intelligence* 43.5 (2019), pp. 1578–1604.

[19] Zhou Wang et al. “Image quality assessment: from error visibility to structural similarity”. In: *IEEE transactions on image processing* 13.4 (2004), pp. 600–612.

[20] AKM Rabby and Chengcui Zhang. “Beyondpixels: A comprehensive review of the evolution of neural radiance fields”. In: *arXiv preprint arXiv:2306.03000* (2023).

[21] Miriam Jäger and Boris Jutzi. “3D Density-Gradient based Edge Detection on Neural Radiance Fields (NeRFs) for Geometric Reconstruction”. In: *arXiv preprint arXiv:2309.14800* (2023).

[22] N Jeppesen et al. “Quantifying effects of manufacturing methods on fiber orientation in unidirectional composites using structure tensor analysis”. In: *Composites Part A: Applied Science and Manufacturing* 149 (2021), p. 106541.

[23] Thomas Müller et al. “Instant neural graphics primitives with a multiresolution hash encoding”. In: *ACM Transactions on Graphics (ToG)* 41.4 (2022), pp. 1–15.

[24] Jonathan T Barron et al. “Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 5855–5864.

[25] David R. Bull and Fan Zhang. “Chapter 4 - Digital picture formats and representations”. In: *Intelligent Image and Video Compression (Second Edition)*.

Ed. by David R. Bull and Fan Zhang. Second Edition. Oxford: Academic Press, 2021, pp. 107–142. ISBN: 978-0-12-820353-8.

[26] Matthew Tancik et al. “Nerfstudio: A modular framework for neural radiance field development”. In: *ACM SIGGRAPH 2023 Conference Proceedings*. 2023, pp. 1–12.

Other Reference Sources

[27] *Adding New Method for Nerfstudio*. https://docs.nerf.studio/developer_guides/new_methods.html.

[28] *Available Nerfstudio Datasets*. https://github.com/nerfstudio-project/nerfstudio/blob/f31f3bba12841955102f3f3846ee9f855f4a6878/nerfstudio/scripts/downloads/download_data.py.

[29] *Categorized NeRF References*. https://github.com/awesome-NeRF/awesome-NeRF?tab=readme-ov-file.

[30] *Configs of Method Template*. https://github.com/nerfstudio-project/nerfstudio-method-template/blob/main/method_template/template_config.py.

[31] *Main Principles of NeRF*. https://github.com/facebookresearch/pytorch3d.

[32] *Nerfacto*. https://docs.nerf.studio/nerfology/methods/nerfacto.html.

[33] *Nerfacto Initial Configs*. https://github.com/autonomousvision/sdfstudio/blob/master/nerfstudio/configs/method_configs.py.

[34] *Nerfstudio module for exporting the results*. https://github.com/nerfstudio-project/nerfstudio/blob/main/nerfstudio/exporter/exporter_utils.py.

[35] *Numpy Array Formatting*. https://numpy.org/doc/stable/reference/generated/numpy.array.html.

[36] *Training interface details*. https://docs.nerf.studio/quickstart/first_nerf.html.

Appendix

Code Implementation Details

The implementations are primarily structured around two different code segments (`Pipeline` and `Model` modules), although minor modifications have also been made to our method's data manager module and some original Nerfstudio modules. Since our pipeline module manages the workflow sequence, the code introduction section started with this module and transition to introducing the other modules as when their turns arise in pipeline. All of the implemented modules can be reached from github link: *https://github.com/BerkKivilcim/Studying-the-Geometric-Convergence-Behaviour-of-Neural-Radiance-Fields-for-Improving-Training-Time.*

The `Config Class` and the `__init__` function of the pipeline are not elaborated upon in this section, as they have undergone only minor modifications, with no substantial changes. Briefly, the `Config Class` defines some hyperparameters used in our method, while the `__init__` function initializes the necessary variables and employs the `super()` method to enable the use of all the functionalities from the previously established `VanillaPipeline` within this section. To truly understand the workflow, we will look at the `"get_train_loss_dict()"` function in our pipeline.

A function described in *section 3.3.1* is invoked just for one time at the start of the training to decide at which steps voxels will be created. Additionally, it also creates an initial masking array composed entirely of ones based on the total number of training images and the width and height sizes of the images. It assumes that all images have the same width and height values, which is a reasonable assumption if all images are obtained within the same camera. The related code part given below.

```
def get_train_loss_dict(self, step: int):
    if step == 0:
        first_step = self.model.config.first_step #first step
        num_partitions= self.model.config.num_partitions #number of partitions
        last_step = self.model.config.max_num_iterations_in_pipeline
        self.samples = self.incremental_sampling(first_step, num_partitions, last_step)
        num_images = len(self.datamanager.train_dataset.cameras)  # Number of images
        height = self.datamanager.train_dataset.cameras[0].height  #Height of first image
        width = self.datamanager.train_dataset.cameras[0].width # Width of first image
        self.config.mask_pixel = torch.ones((num_images, height, width), dtype=torch.int)
```

The implementation of function that decides voxel generation steps, given in next page. In that code, the range between the first and the last step is divided logarithmically. For example, if the hyperparameter for number of voxel samples is set to 20, the code divides the range into 19 parts. The last iteration step is used as the 20th voxel production step, resulting in a total of 20 voxel samples being produced. However, in the logarithmic method, especially in the initial steps, consecutive steps may result in the same value. For instance, if the first step is chosen as 1, the second step might also be 1. If this situation occurs, adding +1 to one of the consecutive values resolves the issue. The result of this function is an array such as *[100, 135,..., 22150, 29999]*.

```
def incremental_sampling(self, first_step, num_partitions, last_step):
    steps_to_distribute = last_step - first_step
    # divide the data range logaritmicaly
    remaining_samples = np.logspace(np.log10(first_step), np.log10(steps_to_distribute), num=
        num_partitions-1, endpoint=False, base=10.0) #base=10 means in the base of log10
    remaining_samples = np.round(remaining_samples).astype(int)
    # makes every step unique
    for i in range(1, len(remaining_samples)):
        if remaining_samples[i] <= remaining_samples[i-1]:
            remaining_samples[i] = remaining_samples[i-1] + 1
    # add last step
    samples = np.append(remaining_samples, last_step-1)
    return samples
```

At this point, our pipeline calls a function from the datamanager to create ray bundles and batches, so firstly, the change made in the datamanager is introduced in the code below. Here, an additional input `"mask"` is defined within `"next_iteration()"` function. If the user opts for masked training, this mask is initially switched to CUDA and created as a tensor then it is engraved into the `"image_batch"` variable. If the image batch contains a mask element, it is automatically applied by Nerfstudio's pixel sampler module. If the user chooses unmasked training, the datamanager operates as it did initially. The modified section of datamanager code is provided below.

It is important to note that, due to this masking process, pointcloud extraction during the training is no longer available since we used an additional input parameter. The code will give an error of `reconstructionDataManager.next_train() missing 1 required positional argument:'mask'` while trying to extract point cloud during the training since `generate_point_cloud()` uses `.next_train(0)` inside of it. Therefore, it is not possible to use point cloud extraction and masking principles together yet. If the user wants to extract a point cloud, he/she needs to remove these masking inputs from both next_train function and where the next_train function is called in pipeline.

```
def next_train(self, step: int, mask) -> Tuple[RayBundle, Dict]:
    """Returns the next batch of data from the train dataloader."""
    self.train_count += 1
    image_batch = next(self.iter_train_image_dataloader)

    if self.config.implement_masking is True:
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        mask = torch.tensor(mask, device=device, dtype=torch.int)
        image_batch["mask"] = mask.unsqueeze(-1)

    assert self.train_pixel_sampler is not None
    assert isinstance(image_batch, dict)
    batch = self.train_pixel_sampler.sample(image_batch)
    ray_indices = batch["indices"]
    ray_bundle = self.train_ray_generator(ray_indices)
    return ray_bundle, batch
```

After creating the ray bundles via using the data manager, the first task is to execute the first training steps in pipeline module with the code part given below here.

```
    ray_bundle, batch = self.datamanager.next_train(step, self.config.mask_pixel)
    model_outputs = self.model(ray_bundle)
    metrics_dict = self.model.get_metrics_dict(model_outputs, batch)
    loss_dict = self.model.get_loss_dict(model_outputs, batch, metrics_dict)
```

Now that our first model training step has been completed, we can initialize to processes such as extraction of rendered RGB images or creating the voxels. To do this, the function of `"generate_and_save_point_cloud()"` is called, which uses an array that contains information of which steps the voxel or extractions will be processed.

```
    self.generate_and_save_point_cloud(step, output_directory, self.samples)
```

The point cloud extraction code implementation part given below here.

```
def generate_and_save_point_cloud(self, step: int, output_directory: Path, samples):
    if step  in samples:
        if self.config.export_point_cloud_per_sample is True:
            pcd = generate_point_cloud(
                pipeline=self,
                num_points=1000000,
                remove_outliers=True,
                estimate_normals=True,
                reorient_normals=True,
                rgb_output_name="rgb",
                depth_output_name="depth",  #depth, expected_depth, custom_depth_render
                normal_output_name=None,
                use_bounding_box=True,
                bounding_box_min=(-1, -1, -1.5),
                bounding_box_max=(1, 1, 1.5),
                crop_obb=None,
                std_ratio=10.0,)  # Outlier removal threshold
            output_path = output_directory / f"point_cloud_{step}.ply"
            o3d.io.write_point_cloud(str(output_path), pcd)
```

Creating synthetic images is computationally expensive. Approximately it takes 1 seconds per image. Therefore, instead of rendering all the images, we select only a few of them for rendering. Here, we use another hyperparameter called `"sample_percentage_train_image_rendering"`. For instance, if this parameter is set to 0.05, only %5 of the images are rendered. By using these parameters, an appropriate step size is established. This approach also allows us to compare results consistently, as the same selected images are rendered after each training session. For the poster dataset, after the image with index 0 is rendered, the next rendered image index is not 1 but the 20, and then it continues to the 40th, 60th, up to 200th image.

After creating RGB and depth images using the `"render_trajectory()"` function, all images were gathered into a single large list. Using a second for loop, we separated the RGB and depth images, processed each one individually, and then extracted. Our RGB values were between 0 and 1. Therefore, these values are multiplied by 255 and extracted in *uint8* format to a specified folder path, using image numbers for labeling.

However, handling depth images was more complex because their values were not confined to the *0 - 1* range; they could exceed *1*. Consequently, a normalization method was initially applied. In areas where the depth was unclear (such as the sky or windows), depth values could be very high, potentially distorting the normalized visualization. Therefore, a logarithmic function was first applied to these depth values, followed by linear stretching to bring them into the *0 - 1* range. Subsequently, these values were multiplied by 255 and exported in *uint8* format.

The rendered image extraction code implementation part given below here.

```
    if self.config.export_rendered_image_per_sample is True:
        total_images = len(self.datamanager.train_dataset.cameras)
        sample_percentage = self.config.sample_percentage_train_image_rendering #uniform
            sampling rate for selecting the training images for rendering
        num_samples = int(total_images * sample_percentage)
        step_size = total_images // num_samples
        selected_indices = list(range(0, total_images, step_size))

        for i in selected_indices:
            rgb_images, depth_images = render_trajectory(
                pipeline=self,
                cameras=self.datamanager.train_dataset.cameras[i],
                rgb_output_name="rgb",
                depth_output_name="depth", #depth, expected_depth, custom_depth_render
                rendered_resolution_scaling_factor=1.0,
                disable_distortion=True,
                return_rgba_images=True
            )

            for j, (rgb_img, depth_img) in enumerate(zip(rgb_images, depth_images)): #checking
                 each validation (or training, depends on the cameras) images
                output_directory_rgb = Path("E:/Berk/code/reconstruction/outputs/outputs_rgb/"
                    f"image_{i}")
                output_directory_depth = Path("E:/Berk/code/reconstruction/outputs/
                    outputs_depth/"f"image_{i}")

                #creates the folders
                os.makedirs(output_directory_rgb, exist_ok=True)
                os.makedirs(output_directory_depth, exist_ok=True)

                rgb_img = (rgb_img * 255).astype(np.uint8) #converting to uint8. Without them
                    the results looks bad. Probably previous format is float32

                log_data = np.log10(depth_img)
                log_min, log_max = log_data.min(), log_data.max()
                normalized_data = (log_data - log_min) / (log_max - log_min)
                depth_final_img = (normalized_data *255).astype(np.uint8)

                o3d_rgb = o3d.geometry.Image(rgb_img) #without them it does not worko3d
                o3d_depth = o3d.geometry.Image(depth_final_img)

                rgb_path = output_directory_rgb / f"rgb_{step}_{i}.png"
                depth_path = output_directory_depth / f"depth_{step}_{i}.png"

                o3d.io.write_image(str(rgb_path), o3d_rgb)
                o3d.io.write_image(str(depth_path), o3d_depth)
```

In addition to all of them, voxel creation function (named as "voxel_density_values()" which is located inside to our model code module) is called in this extraction function, right after the rendered image extraction. So, it allows to extract the latest results before the training termination, besides those voxels are generated at the same iteration steps where the extraction happens.

```
    voxel_density = reconstructionModel.voxel_density_values(self._model)
    if self.config.export_voxel_arrays_per_sample is True:
        output_directory = Path("E:/Berk/code/reconstruction/outputs/outputs_density_voxels")
        output_directory.mkdir(parents=True, exist_ok=True)
        output_path = output_directory / f"density_values_{step}.npy"
        np.save(output_path, voxel_density)
    self.calculate_voxel_differences(voxel_density)
```

The principles of voxel generation are explained in *section 3.2*. The code implementation part is given below here. The function is located inside the model module since variables like `scene_box` exist inside the model module.

```
def voxel_density_values(self):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    reso = self.config.voxel_resolution + 1
    scene_aabb = self.scene_box.aabb.flatten()
    aabb_min, aabb_max = torch.tensor(scene_aabb[:3], device=device, dtype=torch.float32),
        torch.tensor(scene_aabb[3:], device=device, dtype=torch.float32)
    # Adjusts for voxel center calculation. Creates coordinates for each voxel's center
    x = torch.linspace(aabb_min[0], aabb_max[0], steps=reso, device=device)[:-1] + (aabb_max
        [0] - aabb_min[0]) / (reso * 2)
    y = torch.linspace(aabb_min[1], aabb_max[1], steps=reso, device=device)[:-1] + (aabb_max
        [1] - aabb_min[1]) / (reso * 2)
    z = torch.linspace(aabb_min[2], aabb_max[2], steps=reso, device=device)[:-1] + (aabb_max
        [2] - aabb_min[2]) / (reso * 2)
    grid_x, grid_y, grid_z = torch.meshgrid(x, y, z, indexing="ij")
    voxel_centers = torch.stack((grid_x, grid_y, grid_z), dim=-1).reshape(-1, 3)
    # Puts the voxel center points into model and finds density values for each voxel center
    densities_tensor = self.field.density_fn(voxel_centers).reshape(reso-1, reso-1, reso-1)
    # Convert densities tensor to a numpy array
        densities = densities_tensor.detach().cpu().numpy()
    return densities
```

After the voxel generation, the `"calculated_voxel_differences()"` function is called by input of the recently generated voxel. This function first processes our voxels by adjusting outliers that have very high density values then normalizes those density values between 0 and 1. If a recently generated voxel is the first one produced, the median variable is set to infinity, allowing the training to continue regardless of the selected median threshold value. Next, our previous voxel variable represented by `"prev_voxel_density"` is defined as the most recently produced voxel. If the generated voxel is not the first one, the difference between previous voxel variable and the newly produced voxel is calculated then the median value of this difference is analyzed. If the median value is below the predetermined threshold, `"stop_training"` variable turns to True. If it is True, the training automatically terminates; if not, the previous voxel value is set as the most recently produced voxel again. In addition, the difference voxel variable is updated and stored by this function, which is used in the masking method later on.

```
    def calculate_voxel_differences(self, current_voxel_density):
        #Set highest top %1 values to next highest values then normalize the voxel
        top_percent_value_current = np.percentile(current_voxel_density, 99)
        current_voxel_density[current_voxel_density > top_percent_value_current] =
            top_percent_value_current
        current_voxel_density_normalized = (current_voxel_density - np.min(
            current_voxel_density)) / (np.max(current_voxel_density) - np.min(
            current_voxel_density))
        if self.prev_voxel_density is not None:
            diff = np.abs(current_voxel_density_normalized - self.prev_voxel_density)
            median = np.median(diff)
            self.mean = np.mean(diff) #Used in coarse sampling decreasing function
            self.model.config.difference_voxel = diff #Used in Masking Method
            if median < self.model.config.converge_threshold:
                self.stop_training = True #Updates parameter for early stop
        else:
            median = np.inf
            self.mean = 1
        self.prev_voxel_density = current_voxel_density_normalized
```

It is extremely crucial to mention that some original Nerfstudio codes are also optimized. One of them is applied inside the *nerfstudio » engine » trainer.py*. The training process is managed from this code. Therefore, our early termination code is engraved inside the `"def train()"` function of *trainer.py*. The code part given below here is inserted to the line of between 284 to 285, just right after the `self._update_viewer_state()`. If the user decides to use an early stop mechanism and if our `stop_training` variable turns to True, the training is stopped by using `break`.

```
        self._update_viewer_state(step)
###############################################################################################
        if self.pipeline.stop_training is True and self.pipeline.config.early_stop is True:
            break
###############################################################################################
```

In addition to that, the next optimizations are implemented in both *"nerfstudio > cameras > cameras.py"* and *"nerfstudio > cameras > rays.py"* codes. Because our masking array is based on number of images, image height and image width, which means the camera indices and image pixel coordinates are necessary to implement this masking idea. Therefore, those pixel coordinates are engraved into our ray bundle. Consequently, the pixel coordinates of the each ray now have been included in ray bundle.

For the *nerfstudio » cameras » cameras.py*, the code optimization implemented on line of 882 where variable of `coords` is added. The code part is given in below here.

```
        return RayBundle(
            origins=origins,
            directions=directions,
            pixel_area=pixel_area,
            camera_indices=camera_indices,
            times=times,
            metadata=metadata,
            coords=coords
        )
```

For the *nerfstudio » cameras » rays.py*, the code optimization implemented on line of 212 where variable of `coords` is added. The code part is given in below here.

```
class RayBundle(TensorDataclass):
    """A bundle of ray parameters."""
    # TODO(ethan): make sure the sizes with ... are correct
    origins: Float[Tensor, "*batch 3"]
    """Ray origins (XYZ)"""
    directions: Float[Tensor, "*batch 3"]
    """Unit ray direction vector"""
    pixel_area: Float[Tensor, "*batch 1"]
    """Projected area of pixel a distance 1 away from origin"""
    camera_indices: Optional[Int[Tensor, "*batch 1"]] = None
    """Camera indices"""
    nears: Optional[Float[Tensor, "*batch 1"]] = None
    """Distance along ray to start sampling"""
    fars: Optional[Float[Tensor, "*batch 1"]] = None
    """Rays Distance along ray to stop sampling"""
    metadata: Dict[str, Shaped[Tensor, "num_rays latent_dims"]] = field(default_factory=dict)
    """Additional metadata or data needed for interpolation, will mimic shape of rays"""
    times: Optional[Float[Tensor, "*batch 1"]] = None
    """Times at which rays are sampled"""
    coords: Optional[torch.Tensor] = None
```

For the next task, if the user chooses to train the model with using a masking method that implements ray exclusion ideas and if a difference voxel is already created, our code calls a function named as "map_rays_to_voxels()" that calculates which pixels will be masked. This function is located in the model module with the same reason why the voxel creation function is located in the model module. This function, which assesses the voxel-ray interaction to identify the rays or in other terms the pixels to be masked, is called after the voxel differences created. For example, if the first voxel produced at the 100th step and second voxel at the 135th step, that means the differences between the voxels will be calculated at step 135. So, the function of "map_rays_to_voxels()" will be called after the 136th and rest of the later steps. The details about the working principles are explained in *section 3.5*. The code part given below is used to update masking array.

```
        if step > self.samples[1] + 1 and self.datamanager.config.implement_masking is True:
            self.config.mask_pixel = reconstructionModel.map_rays_to_voxels(self._model,
                model_outputs, ray_bundle, self.config.mask_pixel)
```

The code implementation of the function "map_rays_to_voxels()" given in the below and in the next pages. It uses model outputs, ray bundle and previous mask as inputs.

The model output includes information about samples produced both from piecewise sampling (for example, 256 samples) and from PDF (such as 96 and 48). However, since we are interested in the samples produced in the last iteration of the proposal sampler, we only use the last index of the attribute that contains the sample information inside the model output. Also, only the last index of the weights_list is used for the same reason. However, this sample information does not include scene box coordinates of each sample point. Therefore, these scene box coordinates for each sample are computed by using "frustums.get_position()" function.

As a small note, as previously mentioned, since weights are the normalized form of density values, for nerfstudio datasets (such as poster, library) the accumulated weights total of samples on a ray always 1. However, for blender datasets, these values may not always exactly be 1, but they are close to 1 and never exceed it.

```
def map_rays_to_voxels(self, outputs, ray_bundle, mask_pixel):
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        voxel_resolution = self.config.voxel_resolution
        #Find the x,y,z coordinates (in the coordinate system of scene_box) of each sample on
            a ray. Index of [-1] is used because of considering the final iteration samples
            from the proposal network
        sample_coordinates = outputs['ray_samples_list'][-1].frustums.get_positions() #Size of
             (4096,48,3) >> (Batch size(number of rays), Number of samples on a ray, xyz
            coordinates)
        sample_weights = outputs['weights_list'][-1]
```

Continuing with the explanation of the code's operation, initially a tensor with a size of (4096, 48, 1) was created with `cumsum()` function of torch, which contains cumulative weights for each points instead of individual weights. Points that exceeded the value of 0.5 were labeled as 1, and those that did not were labeled as 0. Subsequently, the position of the first point on the ray that exceeded the value of 0.5 was identified by using `argmax()` function of torch. Because if several points share the same highest value (which is 1 in our case) argmax gives the first encountered one's index. A tensor named `"selected_positions"`, with dimension of (4096, 1, 3) was then created, indicating the XYZ position of that point within the scene box coordinate system.

```
        cumulative_weights = torch.cumsum(sample_weights, dim=1)
        mask = cumulative_weights >= 0.5
        int_mask = mask.long()
        first_pass_indices = torch.argmax(int_mask, dim=1, keepdim=True)
        selected_positions = torch.gather(sample_coordinates, 1, first_pass_indices.expand(-1,
             -1, 3))
```

Subsequently, the size of the each voxel element was calculated. To perform this calculation, the total length of a voxel (always 2, since the voxel range is between -1 and 1) was divided by the voxel resolution number. Then, using this size information, the scene box coordinates of our `"selected_positions"` first shifted to range from [-1,1] to [0,2] then transformed to a voxel coordinate system from [0,2] to [0, *voxel resolution*] by dividing the voxel size. After that, our tensor with a dimension of (4096, 1, 3) was recreated that contain the voxel coordinates of selected samples.

```
        scene_aabb = self.scene_box.aabb.flatten()
        aabb_min, aabb_max = torch.tensor(scene_aabb[:3], device=device, dtype=torch.float32),
             torch.tensor(scene_aabb[3:], device=device, dtype=torch.float32)
        voxel_size = (aabb_max - aabb_min) / voxel_resolution  # The range is [-1, 1], so
            total should be length is 2
        voxel_indices = ((selected_positions + 1.0) / voxel_size).long()  # Normalize to [0,
            voxel_resolution]
        voxel_indices = torch.clamp(voxel_indices, 0, voxel_resolution - 1)
```

In the next task, using these voxel coordinates, a mask tensor consisting of True and False values with a size of (4096,1) was created by checking which voxels at these coordinates in the "difference voxel" were greater than or smaller than selected median threshold value. Using this mask tensor, it was possible to determine which rays among the 4096 should have masked by using their camera index and camera pixel coordinates. Consequently, a tensor of size (*number of excluded pixels*, 3), representing (*camera index, pixel height coordinate, pixel width coordinate*) was created.

Another important point is that since the rays are generated from the center of the pixels, the pixel coordinates would take half-integer values, such as (400.5, 200.5). For this reason, we adjusted these coordinates to integer values like (400, 200) by removing the fractional parts with using `".floor().long()"`.

```
        voxel_x, voxel_y, voxel_z = voxel_indices[:, 0, :].T
        camera_indices = ray_bundle.camera_indices
        pixel_coords = ray_bundle.coords.floor().long()

        #Move the difference voxels into GPU for fast implementation
        difference_voxel_tensor = torch.tensor(self.config.difference_voxel, device=device,
            dtype=torch.float32)

        mask = difference_voxel_tensor[voxel_x, voxel_y, voxel_z] < self.config.
            converge_threshold #find the voxel coordinates where the value of voxel below the
            threshhold

        #Finds the pixel informations (camera index and pixel_x, pixel_y) which is not see any
             voxel region with above a threshhold
        filtered_camera_indices = camera_indices[mask]
        filtered_pixel_coords = pixel_coords[mask]
        result = torch.cat([filtered_camera_indices, filtered_pixel_coords], dim=1) #(x,3) >>
            (x is number of pixels, 3 for (camera index, pixel_x, pixel_y))
```

We have also added an offset mechanism for those who want to apply the ray masking function with patches. For example, if the user do not wish to use patches, `"offset = torch.tensor([0, 0], device=device)"` should be used. To use a 3x3 patch, `"offset = torch.tensor([-1, 0, 1], [-1, 0, 1], device=device)"` and `"repeated_camera_indices = result[:,0].repeat_interleave(9)"` should be used. The reason for including the `"repeat_interleave()"` function is to ensure that after shifting the camera pixel coordinates with the offset, the indices of these coordinates match the camera index of the central location of the patch. For instance, if 100 rays are to be excluded without an offset, using a 3x3 patch would result in a total of 900 pixels being excluded. When patches are applied at the edges, they might extend beyond the image boundaries. Therefore, we ensure that the masked patch pixel coordinates always remain within the range of *(0, height-1)* and *(0, width-1)*.

In the end, the result of this function provides an updated array for masking.

```
        num_images, height, width = mask_pixel.shape
        offsets = torch.tensor([[0],[0]], device=device) #Defines patches for 1x1, 3x3...
        y_offsets, x_offsets = torch.meshgrid(offsets[0], offsets[1], indexing='ij')
        y_offsets, x_offsets = y_offsets.flatten(), x_offsets.flatten()
        result_y, result_x = result[:, 1], result[:, 2]
        result_y, result_x = result_y[:, None], result_x[:, None]
        neighbor_y_indices = result_y + y_offsets
        neighbor_x_indices = result_x + x_offsets

        repeated_camera_indices = result[:,0].repeat_interleave(1) #Do not forget to fix here,
             9 element for 3x3 patch
        flat_neighbor_y_indices = neighbor_y_indices.view(-1)
        flat_neighbor_x_indices = neighbor_x_indices.view(-1)
        # Ensure indices are within the image dimensions and clamp if necessary
        neighbor_y_indices = torch.clamp(flat_neighbor_y_indices, 0, height-1)
        neighbor_x_indices = torch.clamp(flat_neighbor_x_indices, 0, width-1)

        mask_pixel = torch.tensor(mask_pixel, device=device)
        mask_pixel[repeated_camera_indices, neighbor_y_indices, neighbor_x_indices] = 0
        return mask_pixel
```

The code implementation related to the reduction of sampling is discussed previously in *section 3.4* is provided below. While details about the general logic are covered in the methodology section, here we provide some details about the implementation. After the masking process is completed, if the user chooses to reduce the coarse sampling numbers, the "train_step()" function called in each iteration, if we are at the same step where voxels are produced, it calls the function to compute new reduced sample size.

```
    if self.config.decrease_coarse_sampling is True:
        #logaritmically decreases coarse sampling according to voxel means
        #Decreasing rate is significantly increased if voxel mean getting closer to zero, but
            decreasing rate is still remains low if mean value is higher than 0.1
        self.train_step(step, self.samples)
```

```
def train_step(self, step: int, samples):
    if step in samples:
        new_num_prop_samples, new_num_nerf_samples = self.calculate_new_num_samples(step)
        self.model.update_sampler_config(new_num_prop_samples, new_num_nerf_samples)
def calculate_new_num_samples(self, step:int) ->int:
    new_initial_num_samples = round(self.initial_num_samples * np.log(1 + 100 * self.mean) /
        np.log(1 + 100)) + 1
    return (new_initial_num_samples, 96), 48
```

The challenging part is being able to update the reduced sampling during the training phase. This is because our sampling values are used inside the proposal sampler within inside the model module, and the parameters of the model module are compiled once at the beginning and then remain fixed. To solve this problem, we have encapsulated the proposal sampler within a function. This function is called when the code is compiled for the first time, and thereafter, it is only called again when the sampling numbers need to be changed, thus updating the proposal network.

```
class reconstructionModel(NerfactoModel):
    """Template Model."""
    config: reconstructionModelConfig
    def populate_modules(self):
        super().populate_modules()
        self.renderer_depth = CustomDepthRenderer(method=self.config.depth_method)
        self.create_samplers() #When the training starts, defines the Sampler with initial
            parameters

    def create_samplers(self): #Updates or Initialize the Sampler with Parameters
        self.proposal_sampler = ProposalNetworkSampler(
            num_nerf_samples_per_ray=self.config.num_nerf_samples_per_ray,
            num_proposal_samples_per_ray=self.config.num_proposal_samples_per_ray,
            num_proposal_network_iterations=self.config.num_proposal_iterations,
            single_jitter=self.config.use_single_jitter,
            update_sched=lambda step: 1,
            initial_sampler=None,
        )
    def update_sampler_config(self, new_num_prop_samples, new_num_nerf_samples): #Updates
        Sampler Parameters
        self.config.num_proposal_samples_per_ray = new_num_prop_samples
        self.config.num_nerf_samples_per_ray = new_num_nerf_samples
        self.create_samplers() #Calls the sampler again with updated parameters
```

After completing all these procedures, the process moves to the next iteration, and "get_train_loss_dict()" is called again, allowing the training to continue with the next batch and updated parameters.

General Workflow Diagram

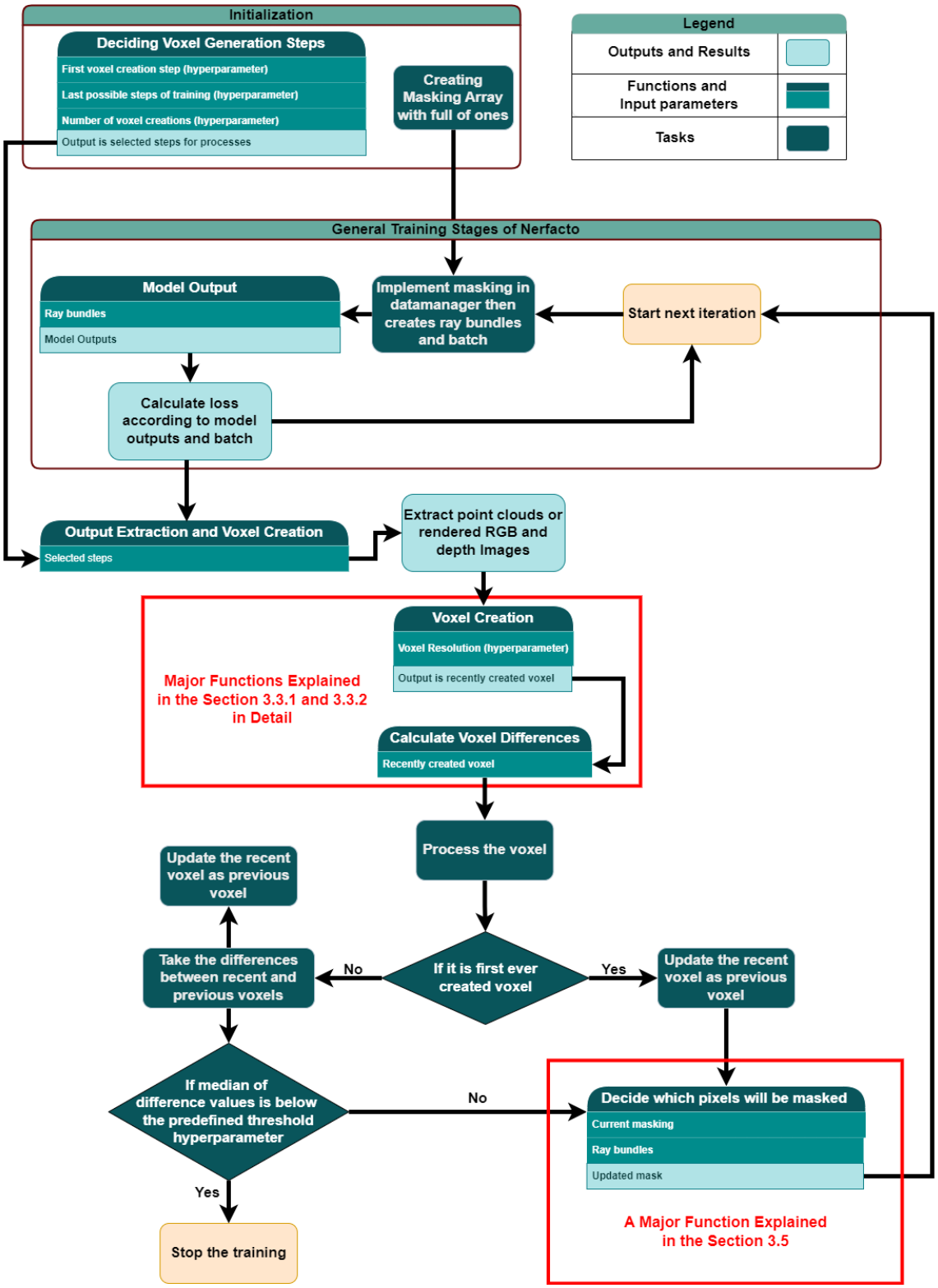

Figure 19: *A workflow diagram to briefly demonstrates the important steps*

Declaration of Authorship

I, Berk KIVILCIM, declare that this thesis titled, *"Studying the Geometric Convergence Behaviour of Neural Radiance Fields for Improving Training Time"* and the work presented in it are my own. I confirm that:

I hereby recognize KIT's 'Guidelines for Ethical Principles' and the rules applicable at KIT for 'Safeguarding Good Scientific Practice'.

I recognize that in case of deliberate scientific misconduct, the opportunity of re-submission is lost and the elaboration will not be approved.

In accordance with the 'Studies and Examination Regulations of Karlsruhe Institute of Technology (KIT) for the Master's Program of Remote Sensing and Geoinformatics' the examination commitee can exclude student who repeatedly violate these policies.

_____

Signature

_____

Date