Alma Mater Studiorum · Università di Bologna

Scuola di Ingegneria e Architettura

Dipartimento di Informatica - Scienza e Ingegneria (DISI)

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea

in

Computer Vision and Image Processing

Improving the Convergence Speed of NeRFs with Depth Supervision and Weight Initialization

Relatore:
Prof. Luigi Di Stefano
Correlatori:
PhD. Daniele De Gregorio
PhD. Luca De Luigi

Candidato:
Damiano Bolognini

Anno Accademico: 2020/2021

A tutte le persone che mi hanno accompagnato durante questi anni universitari, che mi sono state sempre vicine, nonostante le difficoltà. In particolare ai miei genitori, senza di voi questo sogno non si sarebbe potuto realizzare.

Grazie

Abstract

Neural rendering is a new and developing field where computer graphics and deep learning techniques are combined to generate photo-realistic images using deep neural networks.

In particular, Neural Radiance Fields (NeRF) is able to synthesise novel views of a scene with unprecedented quality by fitting a Multi-Layer Perceptron (MLP) to RGB images. However, training this network requires plenty of time and computation even on modern GPUs, making this new technology hardly employable on practical specialized applications.

In this project, we show that employing the known depth of the scene as an additional supervision during the training, and starting from pre-trained weights of other scene with similar setups, instead of from scratch, leads to a convergence speed 3 to 5 time faster.

Sommario

Il neural rendering è un settore emergente che combina concetti di computer graphics e di deep learning per generare immagini fotorealistiche tramite reti neurali.

In particolare, NeRF, acronimo di Neural Radiance Fields, è in grado di sintetizzare nuove viste di una scena con qualità mai vista prima, usando un Multi-Layer Perceptron (MLP) allenato su immagini RGB. Tuttavia, allenare questa rete richiede molto tempo e computazione anche sulle GPU più moderne, rendendo questa nuova tecnologia difficilmente utilizzabile in applicazioni pratiche specializzate.

In questo progetto, dimostriamo che sfruttare la profondità nota della scena come supervisione aggiuntiva durante l'allenamento, e che partire da pesi preallenati di altre scene con una struttura simile, invece che da zero, porta a una velocità di convergenza da 3 a 5 volte più veloce.

Contents

Introduction

Rendering photo-realistic images of a real-world scene has always been one of the hot topic of computer graphics and computer vision. Over the years, several methods have been developed to estimate the physical properties of a real-world scene from its observations, such as images and videos, necessary to achieve photo-realistic synthesis on novel views. However, this estimation task, known as inverse rendering, is extremely challenging and it often fails to reach the wanted image quality.

In contrast, in the past few years, a new and innovative field has emerged: the neural rendering, which combines classical computer graphics techniques with recent advancements in deep learning. In particular, Neural Radiance Fields (NeRF) is an unprecedented and disruptive technology able to represent real-world scenes with photo-realistic image quality with just few megabytes.

The main idea behind it is to employ a deep neural network, a Multi-Layer Percepetron (MLP), to synthesise novel views images of a real-world or synthetic scene by fitting on its observation, such as RGB images and videos.

The results are stunning, but this new technology is far away from being used for everyday user applications, since training and interrogating the network requires a lot of time and computation, which are never enough. The aim of this work is to find a way to speed up the training phase of this technology.

For our project we have combined two ideas: the depth supervision and the weight initialization: the former consists in using the known depth of the scene to help the network understand the undergoing geometry, while the latter

consist in training the network starting from pre-trained weights of different scenes with similar setups to accelerate the training phase.

We show that this combination leads to a convergence speed 3 up to 5 times faster, with a better trade-off between training time and image quality, over different synthetic scenes of Lego models.

Our results have been used by the Eyecan s.r.l. company, which believes that deep learning will be the center of the industry of the future, to take part to the annual OpenCV Spatial AI Contest.

Outline

This work is divided in five Chapters. Firstly in Chapter 1, we will explain in details the fundamentals of neural rendering necessary to understand NeRF, which will be described in Chapter 2, along with few related papers concerning the acceleration of training and rendering time.

Then, in Chapter 3, we present our NeRF model based on weight initialization and depth supervision techniques to boost the convergence speed of the network; the qualitative and quantitative results of our experiments are reported in Chapter 4.

Finally, in Chapter 5, we reported the conclusions and possible future improvements which could be made.

Chapter 1

Introduction to Neural Rendering

Computer graphics has born in the fifties, and since then it has been developed and used for several tasks such as video games, phone and computer displays, art and many specialized tasks.

One of its main topic has always been synthesising controllable and photo-realistic images and videos. In the last decades, the main approach to this problem has been trying to model real cameras, materials and global illumination. These models are based on physics laws, therefore, to render a scene, all the physical parameters must be known, which it is not possible or feasible in certain applications.

To overcome these models' complexity, several methods, based on nontrivial mathematical approximations, have been formalized to simulate the real-world models, such as triangular meshes and heuristics. However, even with these techniques, rendering novel views of a known objects remains challenging, leading to a low realism.

A much harder and related problem is rendering photo-realistic novel views of a real-world object, where the physical parameters are not known, and must

be estimated, through inverse rendering [1], from observations of the scene, such as images and videos, which is often challenging.

Therefore the following question arises, is there another way to render photo-realistic images and videos of a real-world scene? The answer is yes, and it is called *"Neural Rendering"*, which has been defined by Tewari et al. as:

"[...] a rapidly emerging field which allows the compact representation of scenes, and rendering can be learned from existing observations by utilizing neural networks. The main idea of neural rendering is to combine insights from classical (physics-based) computer graphics and recent advances in deep learning. Similar to classical computer graphics, the goal of neural rendering is to generate photo-realistic imagery in a controllable way. This, for example, includes novel viewpoint synthesis, relighting, deformation of the scene, and compositing." [2]

In this way, the neural network approximates internally all the physical parameters for the rendering, without the need of knowing them directly. Therefore, the problem is reduced to find the best network architecture for the given task, which can be challenging, but in less than four years there have been many improvements, and now neural rendering can be used for novel view synthesis, relighting, pose estimation, generative models and many other specialized tasks.

It is a new and developing field, one of the first publications which gave the name to neural rendering was written in 2018 [3], but only in 2020, with the article *"NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis"* by Mildenhall et al. [4], the neural rendering gained more and more interest, so much that this phenomenon has been defined as "The NeRF Explosion"[5].

The main concept behind neural rendering is the disentanglement of the camera capturing process and the scene representation during the training of the neural network. The disentanglement gives consistency while rendering new images with the network, but to achieve it, it is necessary to simulate the camera acquisition process of the scene.

There are several image formation models from computer graphics, there-

fore, in this chapter, we will describe how we can describe a scene (surfaces and volumes), render it (the image formation models) and finally which neural network is the most used for neural rendering.

1.1 Scene Representations

In computer graphics, many different ways have been defined to describe a scene. The most known is without doubt triangle meshes, but also voxel grids, point clouds, implicit or parametric surfaces are commonly used. There are two categories of representations: surface and volumetric, Figure 1.1.1.

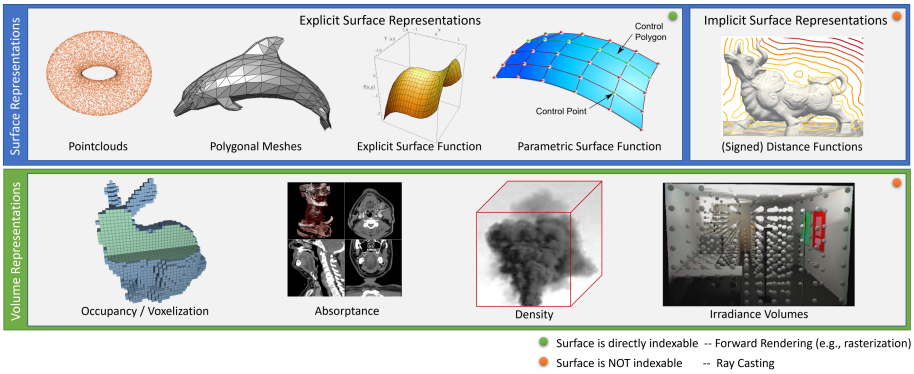


Figure 1.1.1: Different ways to represent a scene [2].

Both have their pros and cons (it depends on the task), in general, volumetric representations can be used to describe surfaces too, but not vice-versa. Additionally, volumetric representations can store several properties of the scene such as density, radiance, occupancy, along with the classical features such as colors. Surface representations are instead more "limited", because they can store properties only of the surface of the object they are describing. These representations can be discretized or continuous, the latters are particularly used in neural rendering, because they provide an analytic gradient, therefore they lend themselves well to neural networks.

1.1.1 Surface Representations

A surface can be described in two ways, explicitly and implicitly.

In the former way, the surface is described by an explicit function, $f_{explicit}(x, y)$ in the Euclidean space, an example could be the height maps $z = f(x, y)$. This representation can be useful, because it is easy to explicitly calculate tangents surfaces, normals and curvature, which are used for rendering.

In the latter way instead, the surface is described by an implicit function, $f_{implicit}(x, y, z) = 0$ in the Euclidean space, an example could be the Signed Distance Fields (SDF), where it is easy to check on which side of the surface a point is located just by looking at the sign of the function in the given point, but it is not easy to enumerate points on the surface and to piece-wise join them maintaining continuity and smoothness.

In general, the function can be any function, for easy ones, such as planes, a linear function is sufficient, but for complex shapes more advanced techniques are needed, such as Taylor series.

The most common ways to approximate the surface function are point clouds and meshes.

A point cloud discretize a surface into a set of sample points of the Euclidean space (x, y, z), they are usually obtained from a 3D scan of a real world object. Other than their positions, each point can store additional attributes such as normals (oriented point cloud) and colors. Since a point cloud is a set of elements of the Euclidean space, they can approximate volumes too, by storing additional attributes such as density and opacity.

A polygonal mesh instead is a linear piece-wise approximation of a surface. It is the most common way in computer graphics to approximate a surface, in particular triangle meshes are considered a standard, and rendering pipelines are able to process billions of triangles in seconds. Each vertex of a triangle can store attributes such as normals and colors, but a common strategy is to employ 2D texture maps to store them. The coordinates of the texture are attached to the vertices of the mesh, in this way, texture coordinates can be computed for

any point of the mesh using barycentric interpolation, and the attributes of the point can be obtained from the texture with bilinear interpolation. Textures are included in the standard graphics pipeline, making their usage easy and fast.

1.1.2 Volume Representations

Similarly to surfaces, volumes can be described by a function too, in this case it describes properties in the entire space: $f_{vol}(x, y, z)$, and the most common way to approximate it is a voxel grid.

A voxel is the equivalent of a pixel in a three-dimensional space, but other than colors, a voxel can store more attributes, such as occupancy, density and transparency. A voxel grid is a collection of voxels, it is usually obtained by a CT scan of a real world object and it is used as a discretized description of a volume. The final appearance of the scene is obtained by interpolating the attributes of the voxels with a rendering technique called volume rendering, which will be discussed later in this chapter, Section 1.2.5.

Voxel grids are a powerful way to store volume descriptions, but they require a large storage, thus they are difficult to share on a normal user bandwidth.

1.2 Image Formation

We have seen how we can describe and store the 3D geometry of a scene, now we will see how we can generate images from these representations through rendering. The best rendering method is to trace every light particle in the scene, but for obviously reason this is impossible, it would take an enormous amount of computation and time, therefore several algorithms have been researched to obtain an image in a reasonable amount of time without loosing too much image quality. In particular, three families of algorithms are used in rendering:

- rasterization: the scene representation is projected into the image plane;
- ray casting: it simulates the camera acquisition process by casting rays into the scene;

- ray tracing: it is the evolution of ray casting employing more advanced simulations to obtain more realistic results.

Rendering methods are based on the pinhole camera model, where the basic intercept theorem describe how a 3D world point is projected into the 2D image plane. This projection is described by a non-injective function, therefore it is not easily invertible.

The pinhole model is described by a parameter matrix for the projection, called intrinsic matrix $\mathbf{K}$; it contains the focal lens normalized by pixel size $\mathbf{f} = [\alpha_x, \alpha_y]$, the axis skew γ, which is usually zero, and the center point $\mathbf{c} = [c_x, c_y]$:

$$\mathbf{K} = \begin{bmatrix} \alpha_x & \gamma & c_x & 0 \\ 0 & \alpha_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

From the homogeneous coordinates of a scene point $\mathbf{p}' = [x, y, z, 1]$, we can find its projection in pixel $\mathbf{q}' = [u, v, 1]$ with the intercept theorem:

$$\mathbf{q}' = \mathbf{K} \cdot \mathbf{p}' \tag{1.2.1}$$

The center of the projection is at the coordinate origin and the camera is axis-aligned. For arbitrary camera positions, an homogeneous 4×4 roto-translation matrix can be used, which is called extrinsic matrix, or pose:

$$\mathbf{E} = \begin{bmatrix} \mathbf{R}_{3\times3} & \mathbf{t}_{3\times1} \\ \mathbf{0}_{1\times3} & 1 \end{bmatrix}$$

where $\mathbf{R}$ is a rotation matrix and $\mathbf{t}$ is a translation vector, such that $\mathbf{p_c} = \mathbf{R} \cdot \mathbf{p_w} + \mathbf{t}$, where $\mathbf{p_w}$ is the point in world coordinates and $\mathbf{p_c}$ in camera coordinates.

In computer vision this model is commonly used, and it is referred to as "world-to-cam" mapping, while in computer graphics the inverse "cam-to-world" mapping is more frequent. In the "world-to-cam" mapping the final full projection of a real world point $\mathbf{p_w}$ to its corresponding pixel $\mathbf{q_p}$ becomes:

$$\mathbf{q_p}' = \mathbf{K} \cdot \mathbf{E} \cdot \mathbf{p_w}'. \tag{1.2.2}$$

which is without doubt extremely fast and efficient since it is just a sequence of matrix multiplications.

To correctly model the camera, it is necessary to model the lens too, because they add distortion effects to the projection function. Modeling the lens is trivial, because a camera calibration is needed and they are modeled through polynomials up to degree five, therefore they are not easily invertible. Anyway there are modern approaches for camera calibration which use many more parameters to achieve a better accuracy and make it invertible and differentiable [6].

1.2.1 Rasterization

Rasterization is usually faster than the ray-based techniques, and it is the rendering method used by all graphics cards. The main idea behind rasterization is to project the geometric primitives of the scene, which are usually meshes, into the image plane, Figure 1.2.1. It is usually faster because it focuses only

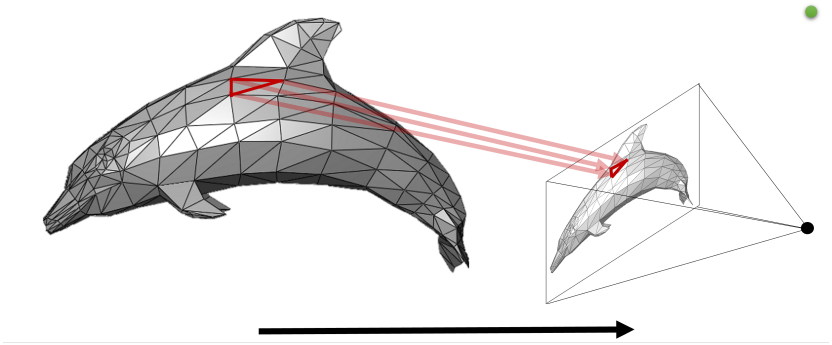


Figure 1.2.1: In rasterization the scene primitives are projected into the image plane [2].

on the primitives, therefore the empty areas are automatically ignored, while in the ray-based methods all the areas are checked. In addition, contiguous pixels are usually occupied by the same primitive, therefore it is possible to reduce redundant computations. The pixels' color is given by the primitive, and different

approaches can be used.

Since it is faster, it is generally used when interactive rendering is required, i.e. video games and simulations, but usually the image quality is worse than the ray-based techniques and it is less versatile because it is based on several image assumptions.

1.2.2 Ray Casting

Ray casting is the easiest ray-based rendering technique, which instead of starting from the primitives to obtain the final image, it starts from the pixels. Ray casting simulates the camera acquisition process by casting rays from the camera center into the scene passing through the pixels, usually one ray per pixel, Figure 1.2.2. Each ray is then sampled, and for each sample the corresponding radiance, which is the quantity of light coming from the ray direction, is computed. Finally, the contributions of each sample are accumulated into the final color value. Ray casting assumes that the rays follow a straight path without

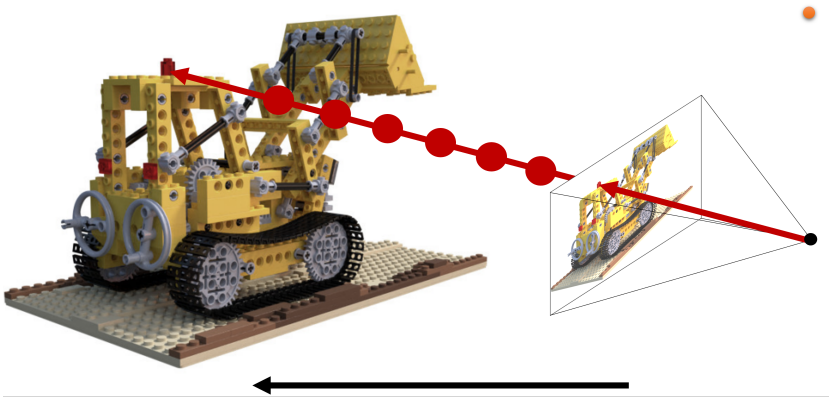


Figure 1.2.2: In ray casting the image is generated by casting and sampling rays [2].

bouncing, therefore the light effects usually are not sophisticated as other ray-based rendering techniques, but it is faster and it was commonly used in early nineties video games, such as Wolfestein and Doom.

1.2.3 Ray Tracing

Ray tracing is a more sophisticated ray-based rendering method, it aims to simulate the flow of light particles. It is the method that achieve the better quality, but it is also the slower because a larger number of rays is needed.

In general, for each pixel several rays are shot, and when a ray hits an object, other rays from the intersection are shot, such as shadow rays to see the direct light interaction, refraction rays if the object is refractive, reflected rays if the object is reflective and so on .

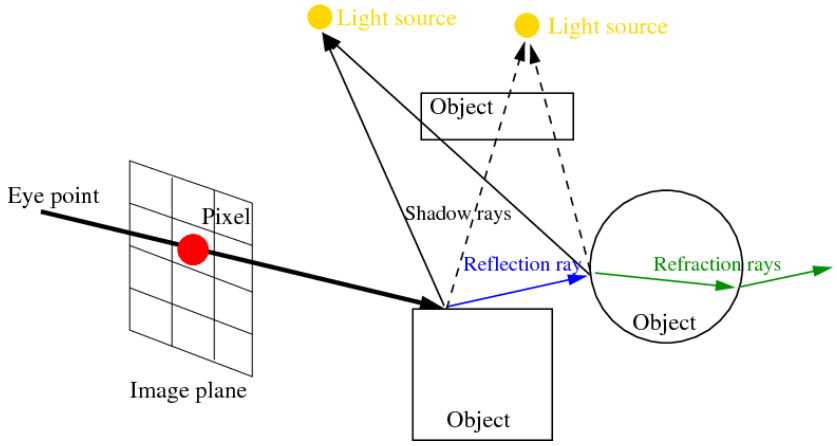


Figure 1.2.3: In ray tracing the image is generated by casting rays and combing their contributions [7].

Finally, the contributions of each ray are combined to obtain the final pixel color. The bounces of the rays allow to obtain a photo-realistic light interaction, but it comes with a cost, casting more rays means more computation.

Since it is computational demanding, the number of bounces is usually limited, for example in path tracing only a single ray, or none, is fired at each intersection.

Ray tracing is a brute force method, thus it is rarely used for real-time applications since it is too slow, but in the last few years there have been several attempt, especially in the video games industry, to make it real-time, with

promising results.

1.2.4 Surface Rendering

Polygonal Meshes The most common way to render a polygonal mesh is through rasterization. First of all, the polygons are divided into triangles, if it is not already a triangle mesh, then each vertex of the mesh is transformed into camera coordinates by using the extrinsic matrix **E**. The points that are outside the view frustum or have a wrong normal orientation are not considered, because they are not visible from the camera, and this reduce the amount of faces to process. After the culling, the projections of the vertices are found by using the Equation 1.2.2.

Finally, the pixels of each triangle are found from the pixel locations of its vertices, this can be done with different algorithms, such as the Bresenham algorithm; for the pixels final color there are different techniques too, for example linear interpolation of the vertices colors. To respect the right depth order of the triangles, the depths can be stored in a z-buffer, in this way only the visible triangles are rendered to screen.

The ray-based rendering techniques can be used with polygonal meshes too, the main idea is to find the intersection of the rays with the faces. Using ray tracing leads to better results, but rasterization remains faster and it is still the most used method for real-time applications.

Point Clouds Point clouds are usually converted into other surface representations such as polygonal meshes and NURBS surfaces. The conversion can be done in several ways, some of them create a network of triangles from the points of the point cloud, such as the Delaunay triangulation, others convert the point cloud into a volumetric distance field and then the implicit surface is obtained through a marching cubes algorithm.

1.2.5 Volume Rendering

We have previous said that voxel grids can be rendered through volume rendering. Volume rendering is based on ray casting, and the scene is seen not as a collection of hard surfaces, such as polygonal meshes, but as a continuous field of volume densities. Each 3D point x in the scene has its density $\sigma(x)$, which can be seen as the differential probability of a ray terminating at an infinitesimal particle at location x, and color $c(x, d)$, where d is the viewing direction.

This information are usually given by voxel grids, where each voxel has its density and color, used for volume rendering. As we said before, to render an image, rays are shot from the optical center into the scene, one ray per pixel. The final color of a pixel, with its corresponding camera ray $r(t) = o + td$, with o the origin and d its direction, with near and far bounds t_n and t_f, is given by the integral:

$$\begin{aligned} C(r) &= \int_{t_n}^{t_f} T(t)\ \sigma(r(t))\ c(r(t), d)\ dt; \\ \text{where } T(t) &= \exp\left(-\int_{t_n}^{t} \sigma((s))\ ds \right). \end{aligned} \tag{1.2.3}$$

The function $T(t)$ indicates the accumulated transmittance, which is the probability of the ray to not hit anything in $[t_n, t]$.

The continuous integral is then discretized with the quadrature rule[8]:

$$\begin{aligned} \hat{C}(r) &= \sum_{i\ =\ 1}^{N} T_i(1 - \exp(-\sigma_i\delta_i))c_i; \\ \text{where } \ Ti &= \exp\left(-\sum_{j=1}^{i-1} \sigma_j\ \delta_j \right). \end{aligned} \tag{1.2.4}$$

where $\sigma_i = t_{i+1} - t_i$ is the distance between two adjacent samples.

This function is differentiable and it can be seen as traditional alpha blending with $\alpha = 1 - \exp(-\sigma_i\delta_i)$.

1.3 MLP: Multi-Layer Percepetron

Earlier we said that surfaces and volumes can be described by a function, which can be approximated in different ways. In neural rendering, the main idea is to approximate this function with a neural network, in particular with the Multi-Layer Perceptron (MLP), which is know to behave as an Universal Function Approximator [9].

The multi-layer perceptron is one of the oldest and simplest fully-connected neural network, it has at least three layers: the input layer, the hidden layer and the output layer, and each node is a perceptron [9]. Except for the input layer, the nodes are activated by a nonlinear activation function, usually the Rectified Linear Unit (ReLU).

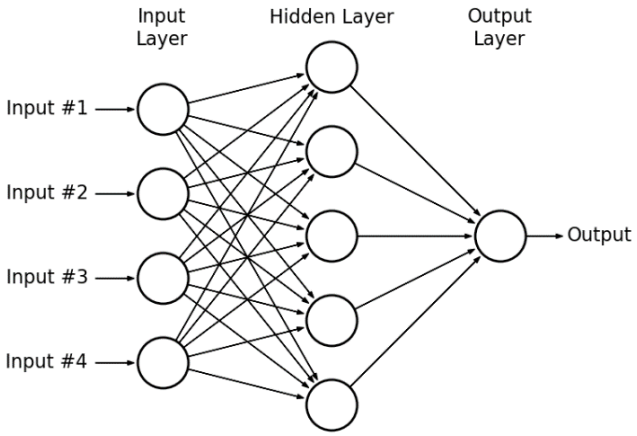


Figure 1.3.1: The Multi-Layer Perceptron.

MLPs work like every deep neural network: first there is the feed forward phase, where from a given input the network computes a certain output through a series of weighted sums. Then, the weights of the networks are updated in the backpropagation phase in the opposite direction of the gradient, given by the loss function, which is usually the mean squared error between the output of the network and the expected value.

In our use case, the MLP takes as input the coordinates of a 3D point in the

space and the viewing direction, and gives as output some values corresponding to that point, such as colors and density. This type of networks is known as *coordinate based neural network*, and this scene representation is called *coordinate based scene representation* [2].

To obtain the best from the network, a procedure, called positional encoding, is performed, where the input is mapped to a higher dimensional space using high frequency functions, such as goniometric functions. This is needed because the MLP performs poorly at representing high-frequency variation in color and geometry, and positional encoding enables better fitting of data that contains high frequency variation [10].

Chapter 2

Neural Radiance Fields

In the previous chapter we have described the fundamentals of neural rendering, now in this chapter we will see how they are used to generated photo-realistic images of a real world scene. The most relevant article on the subject is without doubt: *"NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis"* [4]; since its publication, several articles have been published each week, and this phenomenon has been called *"The NeRF explosion"* [5]. Even if neural radiance fields are extremely powerful tools for many different specialized applications, they are very slow both at training and inference time, making them unfeasible for real time application.

In this chapter we will first explain what is NeRF, then we will focus on few articles containing good ideas on how we could speed it up.

2.1 NeRF

NeRF stands for Neural Radiance Fields:

- neural: because it is a neural network;
- radiance: because the network describes the radiance of the scene;

- fields: because the network is able to return the radiance of any point of the scene, i.e. it is a continuous representation, not discrete.

In other words, NeRF is a neural network able to perform impressive photo-realistic novel views synthesis of a real-world or synthetic scene (view synthesis results are best viewed as videos, so we recommend to visit the official NeRF website: https://www.matthewtancik.com/nerf to see them).

The main idea behind NeRF is simple: train a neural network over a dataset of images of a scene taken from multiple point of views, and once that the network is trained, it is able to generate images of the scene from novel views, Figure 2.1.1.

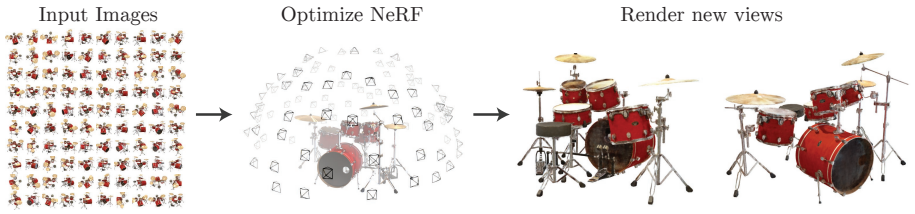


Figure 2.1.1: From a dataset of images of the scene, NeRF is able, once trained, to generate novel views of the scene with photo-realistic quality [4].

Obviously, the images of the dataset should be taken uniformly all around the scene, otherwise the network will learn correctly only the views that has seen the most.

2.1.1 Scene Representation

In the last chapter, we have said that an MLP can approximate the scene representation function. In NeRF, the scene is represented by a 5D vector-valued function, whose input is a 3D point of the scene $x = (\mathbf{x}, \mathbf{y}, \mathbf{z})$ and a 2D viewing direction $d = (\boldsymbol{\theta}, \boldsymbol{\phi})$ (which in practice is a 3D Cartesian unit vector) and whose output is the emitted color $c = (\mathbf{r}, \mathbf{g}, \mathbf{b})$ and volume density σ of the point. In other words, this continuous function, given a point of a scene

and from where it is seen, gives as output the color of the input point and its density.

The function is approximated using an MLP F_Θ, where its weights Θ encode the scene representation, each scene will have its set of weights, while the network structure remains unchanged.

To render an image from the MLP, a ray for each pixel is shot from the camera center into the scene, then the ray is sampled and each sample is given as input to the network. The final color of the pixel is computed by combining the outputs of the network with volume rendering, Section 1.2.5. Therefore, the pose and the camera parameters of each image are needed to cast the rays, and if they are not available they can still be retrieved using Structure-From-Motion (SFM) solvers like COLMAP.

To train the network, the procedure is similar, given the final color of the pixel, the rendering loss is simply the mean squared error of the output color $\hat{C}(r)$ and the ground truth value $C(r)$, Figure 2.1.2:

$$\mathcal{L} = \sum_{r \in R} \|\hat{C}(r) - C(r)\|_2^2 \tag{2.1.1}$$

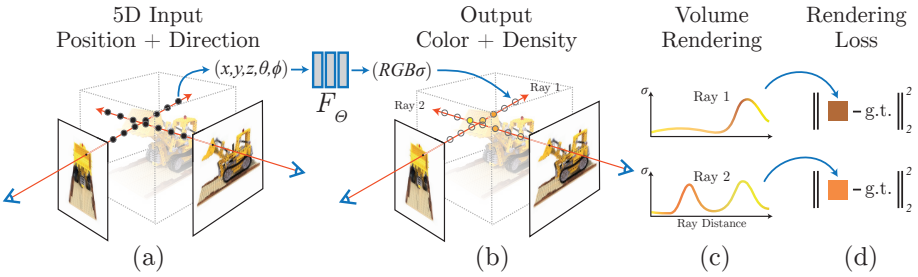


Figure 2.1.2: The camera rays are sampled (a), the samples are fed to the network to obtain their colors and densities (b), volume rendering composes the colors and densities of the camera ray to obtain the final pixel color (c). Since the rendering function is differentiable, the scene is optimized by minimizing the mean squared error of the output color and ground truth [4].

The multi-view consistency is enforced by the network structure, Figure

2.1.3: the volume density σ depends only on the spatial location x, while the color c depends on both location and viewing direction d. To achieve this, the network first takes as input only the 3D location x and processes it with 8 fully-connected layers, using ReLU activation and 256 channels each, to obtain the volume density σ and a 256-dimensional feature vector. The feature vector is concatenated with the viewing direction d and is given as input to the last fully-connected layer, still using ReLU activation and 128 channels. Finally, the point color c is obtained with a sigmoid activation. The main idea is to make the network focus more on the density (with more layers), and only at the end the viewing direction is used to obtain the color, in this way the MLP is able to represent reflections more effectively.

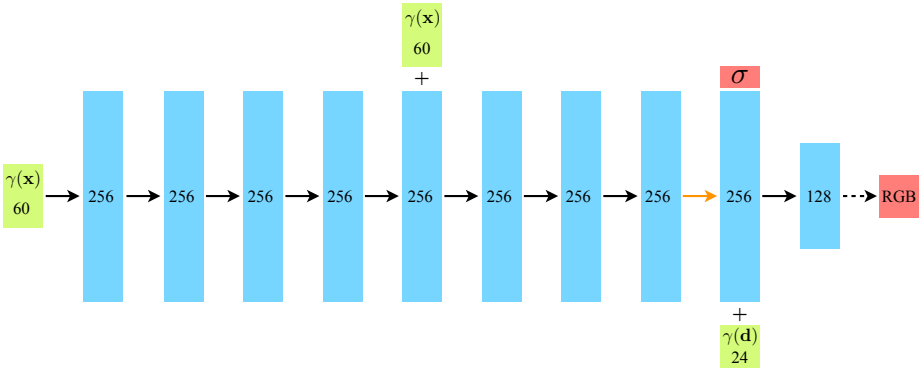


Figure 2.1.3: The network architecture. Input vectors are shown in green, intermediate hidden layers are shown in blue, output vectors are shown in red, and the number inside each block signifies the vector's dimension. All layers are standard fully-connected layers, black arrows indicate layers with ReLU activation, orange arrows indicate layers with no activation, dashed black arrows indicate layers with sigmoid activation, and "+" denotes vector concatenation. The $\gamma(\cdot)$ indicates positional encoding [4].

According to Zhang et al. [11], NeRF is overparameterized: the function takes as input five parameters (three for the position and two for the viewing direction) to describe the radiance fields, therefore there are five degrees of

freedom, but radiance fields can be described by just four parameters [12] [13], basically because the radiance is constant when the ray travels in free space and it changes only on surfaces, which are two dimensional, therefore just four parameters are sufficient. In NeRF we have the illusion that the radiance changes when the ray travels in free space, consequently we have an additional degree of freedom along the ray which is redundant and illusory. In theory, NeRF should be able to render correctly only the training images, since there are infinite "wrong" solutions which explains them, thanks to the extra degree of freedom, but NeRF is able to always choose the "right" solution, and the novel views are coherent with the scene, so why does it work?

The reason behind this has to be searched inside the network architecture: we have said that the MLP takes as input only the position, and just at the end, for just one layer, it takes as input the viewing direction; in this way, the majority of the layers depends only on three parameters: the network is starved, reducing the probability of overfitting.

2.1.2 Volume Rendering

As we said in Section 1.2.5, we can use volume rendering to render the final color of a pixel given the samples along its corresponding ray.

In NeRF, the samples are not a fixed discrete set of locations, because it would limit the representation's resolution, but instead, the ray's bounds $[t_n, t_f]$ are partitioned into N evenly-spaced bins, and then a random sample is taken from each bin:

$$t_i \sim \mathcal{U}\left[t_n + \frac{i-1}{N}(t_f - t_n),\ t_n + \frac{i}{N}(t_f - t_n)\right]. \tag{2.1.2}$$

This discrete set of samples is then fed to the network, and the outputs are used to compute the final pixel color with the Equation 1.2.4.

2.1.3 Optimizations

To achieve high resolution in complex scenes, the core components we have described are not sufficient, in particular two optimization are needed: the first one is positional encoding, already cited in Section 1.3, while the second one is hierarchical sampling.

Positional Encoding Without positional encoding, the network is not able to learn high frequency variations in color and geometry, because deep networks are predisposed to learn lower frequency functions [14]. Therefore, the input is mapped into a higher dimensional space using high frequency functions. The mapping is done with the function $\gamma(\cdot)$, which maps the input from $\mathbb{R}$ to $\mathbb{R}^{2L}$, where L is a tunable hyper-parameter:

$$\gamma(p) = \left(\sin\left(2^0\pi p\right), \cos\left(2^0\pi p\right), \ldots, \sin\left(2^{L-1}\pi p\right), \cos\left(2^{L-1}\pi p\right)\right) \tag{2.1.3}$$

the positional encoding is applied to both point x and viewing direction d.

Tuning L is fundamental, if it is too low the network underfit, leading to oversmoothed interpolation, if it is too high the network overfit, leading to noisy interpolation.

In the original NeRF implementation, the high frequency functions for the positional encoding are simply sines and cosines, but better results can be achieved by using Random Fourier Features instead, as explained in the article *"Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains"* by Tancik et al.[10].

Hierarchical Sampling We said that the samples along the ray are taken randomly from evenly-spaced bins, but this sampling technique can be improved with the hierarchical sampling. The main idea is to take more samples in the regions where the density is higher, because those are the samples that have a bigger influence on the final rendering, and it is where the visible content is present.

Therefore, two networks are used, one "coarse" and one "fine". The "coarse" network evaluates the samples taken with the sampling technique we have already described, to have an idea on how the density is distributed along the ray. Then, given the output of the "coarse" network, more samples are taken in the regions where the density seems to be higher.

To achieve this, the alpha composited color from the coarse network, Equation 1.2.4, is rewritten as a weighted sum of all sampled colors c_i along the ray:

$$\hat{C}_c(r) = \sum_{i=1}^{N_c} w_i c_i, \;\; w_i = T_i(1 - \exp(-\sigma_i \delta_i)). \tag{2.1.4}$$

A piecewise-constant PDF along the ray is obtained by normalizing this weights as $\widehat{w_i} = \frac{w_i}{\sum_{j=1}^{N_c} w_i}$.

The additional samples are taken from this distribution using inverse transform sampling. Finally the "fine" network $\hat{C}_f$ is evaluated at all the sampled points to obtain the final color.

The loss is given by the mean squared error between the true color and the rendered one of both "fine" $\hat{C}_f$ and "coarse" $\hat{C}_c$ networks:

$$\mathcal{L} = \sum_{r \in R} \left[\| \hat{C}_c(r) - C(r) \|_2^2 + \| \hat{C}_f(r) - C(r) \|_2^2 \right] \tag{2.1.5}$$

where R is the set of rays in each batch and $C(r)$ is the ground truth. Even if the final color is given only by the "fine" network, the loss of the "coarse" one is minimized too, in this way its weight distribution can be used to choose the samples for the "fine" network.

2.1.4 Pros and Cons of NeRF

NeRF is without doubt a powerful way to represent a scene for several reasons:

- it allows novel views synthesis with photo-realistic quality;
- the scene representation, i.e. the network weights, requires little storage, only 5 megabytes per scene;

- it does not need complex hardware, a single NVIDIA V100 GPU is sufficient;
- the images of the dataset can be taken by a normal cell phone.

Even if NeRF achieves incredible results, several improvements are needed before it could be used for practical applications:

- it can render only static scenes;
- it takes up to two days to correctly train the network on a NVIDIA V100 GPU;
- it takes up to twenty seconds to render a frame on a NVIDIA V100 GPU, therefore it cannot be used for real-time applications;
- a large number of training images is needed to avoid artifacts on novel views;
- it is not possible to change the scene properties, such as the light, because the radiance is strictly bonded to the image conditions.

Several improvements have already been made to overcome these limitations, and now neural radiance fields can be used for many different specialized tasks[2].

2.2 KiloNeRF

One of the main advantages of NeRF is that the scene representation requires little storage, therefore it can be easily streamed on a normal user bandwidth, but the inference time remains slow, therefore it cannot be used by the user for real-time applications.

To overcome this problem, Reiser et al., in the article *"KiloNeRF: Speeding up Neural Radiance Fields with Thousands of Tiny MLPs"* [15], proposed to increase the rendering speed of NeRF by using a large number of independent

small networks, where each one of them represents only a fraction of the scene, instead of a single wide and deep MLP, Figure 2.2.1.

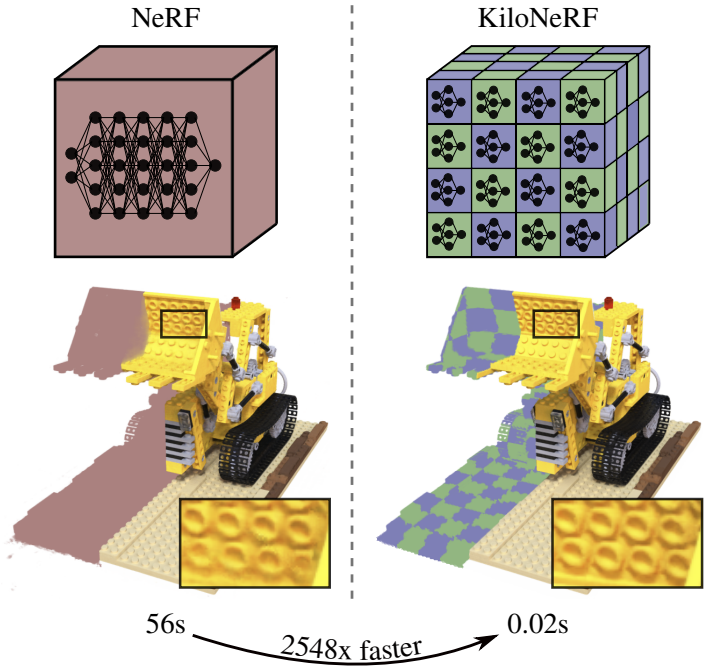


Figure 2.2.1: Instead of using a single wide and deep MLP, the scene is represented by thousands of tiny MLPs. This allows to render a scene three orders of magnitude faster [15].

This technique allows to have the same image quality of the original NeRF model, while the rendering time is three orders of magnitude faster, allowing real time rendering. The downside is that the training phase is longer because a three-stage training strategy is used: first a regular NeRF is trained, then the tiny MLPs are trained to match the outputs of the regular NeRF, finally the tiny MLPs are fine tuned on the original training images. This teacher-students model is much slower to train, but the advantages can be seen at rendering time.

2.2.1 The Network Architecture

In KiloNeRF, the scene is assumed to be contained inside an axis aligned bounding box, with b_{min} and b_{max} its minimum and maximum bounds. The scene is subdivided into an uniform grid of resolution $r = (r_x, r_y, r_z)$, $(16, 16, 16)$ at most, and a tiny MLP, with its own set of weights $\Theta(i)$, is assigned to each cell of the grid, with index $i = (i_x, i_y, i_z)$.

Given a 3D point x to evaluate, the corresponding tiny MLP index is obtained through spatial binning $g(\cdot)$:

$$g(x) = \lfloor (x - b_{min})/((b_{max} - b_{min})/r) \rfloor \tag{2.2.1}$$

then the color and density of the point, with viewing direction d, is computed by its corresponding tiny MLP:

$$(c, \sigma) = f_{\Theta(g(x))}(x, d) \tag{2.2.2}$$

The tiny MLP architecture is a downscaled version of the original NeRF MLP architecture, Figure 2.2.2: the network first takes as input only the 3D location x and process it with 2 fully-connected layers, using ReLU activation and 32 channels each, to obtain the volume density σ and a feature vector. The feature vector is concatenated with the viewing direction d and is given as input to the last fully-connected layer, still using ReLU activation and 32 channels. Finally, the point color c is obtained with a sigmoid activation.

2.2.2 The Teacher-Students Model

The teacher-students model is used in KiloNeRF because training the tiny MLPs from scratch leads to artifacts, so even if it is faster at rendering time, the image quality is worse. The three-stage training pipeline is composed of the following phases:

- pretrain phase: a regular NeRF is trained on the scene, the network model is the same we have described in Section 2.1, with the exception that the hierarchical-sampling is not performed;

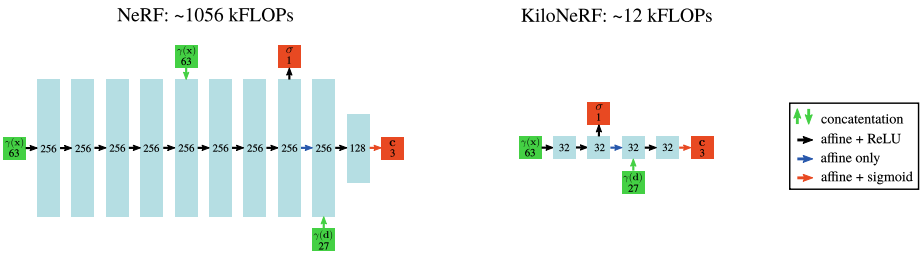


Figure 2.2.2: The tiny MLP architecture is a downscaled version of the original NeRF's architecture. In this way, a forward pass requires only 1/87th of the floating point operations (FLOPs) of the original architecture [15].

- distillation phase: the knowledge of the teacher is distilled into the students;
- finetune phase: the students are fine tuned on the training images.

To distill the knowledge of the teacher into the students, for each of the tiny networks, random samples with random viewing directions are taken from its corresponding grid cell, and the tiny network is trained to match the outputs of the teacher. Therefore, during the distillation phase, no volume rendering or training images are used.

Finally, the students are trained over the training images in the finetune phase, this is necessary because otherwise the rendering quality of KiloNeRF would be upper bounded by the rendering quality of the regular NeRF. All this pipeline can be seen as a way to provide a strong and solid weight initialization to KiloNeRF.

2.2.3 The Sampling Technique

Using thousands of tiny MLPs improves the rendering speed, but it is not enough to have a real-time rendering, therefore two sampling technique are used at rendering time to avoid unnecessary extra computation: empty ray skipping (ESS) and early ray termination (ERT), Figure 2.2.3. Hierarchical sampling is not performed in KiloNeRF because it is costlier than ESS.

Empty Space Skipping The main idea of ESS is to avoid querying the network in empty space. A second uniform grid with higher resolution, called occupancy grid, is generated, and each cell has a binary value to indicate if the cell is empty or not. The occupancy grid is generated by dividing each cell of the previous grid into a 3x3x3 subgrid, then the teacher network is evaluated inside each sub-cell to mark it as occupied or empty: a cell is marked as occupied if any of the evaluated densities is above a threshold.

During the rendering, only the samples which are inside a cell marked as occupied are evaluated, reducing the computation time.

Early Ray Termination The main idea of ERT is to avoid computing the last samples of a ray if the transmittance falls below a threshold during volume rendering. If the transmittance has a peak and then becomes close to zero along the ray, it means that the ray intercepted something and then went into an empty space, therefore computing more samples can be avoided.

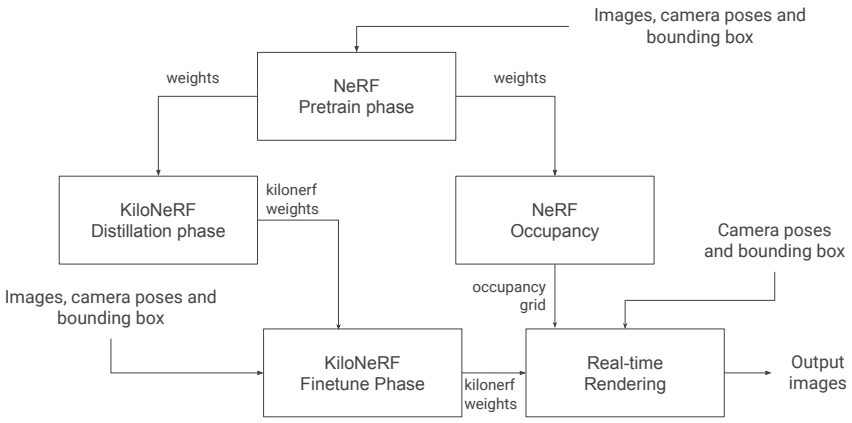


Figure 2.2.3: During the pretrain phase the regular NeRF is trained, then it is used to train the tiny MLPs in the distillation phase and to generate the occupancy grid, finally the tiny MLPs are fine tuned during the finetune phase and they can be used to perform real time rendering.

2.2.4 Pros and Cons of KiloNeRF

KiloNeRF is an evolution of NeRF to achieve real-time rendering, but it comes with a cost, the training is much slower: on an NVIDIA GTX 1080 Ti, the pretrain phase takes 2 days, the distillation phase 8 hours, computing the occupancy grid 4 hours, and the finetune phase 17 hours, for a total of more than 3 days, against the 1-2 days of the regular NeRF.

2.3 DS-NeRF

We have seen how we can speed up the rendering time with KiloNeRF, but what about the training time?

Deng et al. proposed DS-NeRF (Depth-Supervised NeRF) [16], where the main idea is to add depth supervision to the regular NeRF to make it converge faster with fewer training images. The depth supervision helps NeRF to learn the scene geometry, avoiding overfitting, but ground truth values for the depth are rarely available.

The depth could be provided by depth sensors, or can be estimated with stereo-cameras. Another solution is to use Structure-From-Motion solvers like COLMAP, since they are already used to estimate the camera parameters necessary to train NeRF, because they provide a sparse 3D point cloud of keypoints of the scene and the re-projection errors between detected 2D keypoints and projected 3D points, which can be used for depth supervision.

The main idea behind DS-NeRF is to compute the estimated depths of sparse 3D points with COLMAP and then use them in an additional loss which ensures that the depth obtained from the network is close to the observed one. In this way, the training is 2-6x faster, it requires fewer images without loosing in image quality.

2.3.1 The Depth Loss

COLMAP provides for each image i, its camera pose P_i and intrinsics K_i, in addition, it also estimates 3D keypoints $X : x_1, x_2, ... \in \mathbb{R}^3$ across multiple views, and their scene occlusions, in this way it explicitly identifies the subsets of keypoints visible from a particular camera i: $X_i \subset X$. The depth of the keypoints in camera j is obtained by reprojecting them using P_j before projecting the result onto its unit camera axis $[0, 0, 1]$.

The depth estimated by the network is obtained with volume rendering by just integrating the density σ of the samples along the ray. Given a ray $r(t) = o + td$, with o its origin and d its direction, with near and far bounds t_n and t_f, the depth is computed by the integral:

$$\hat{D}(r) = \int_{t_n}^{t_f} T(t)\sigma(t)tdt \tag{2.3.1}$$

where $T(t)$ is the same of Equation 1.2.3.

While the RGB supervision can be used everywhere, the depth supervision can be used only for the 3D keypoints estimated by the SFM. For each keypoint i, a ray is shot from the camera j: $r_{ij}(t) = o_j + \mathcal{F}_j(x_i - o_j)t$, which samples points lying between camera j's center of projection o_j and keypoint x_i; $\mathcal{F}_j$ is a function which scales the ray to have lenght 1 along j's camera axis.

Finally, the depth supervision loss is given by the mean squared error between the depth estimated by the network and the one estimated by the SFM, multiplied by the confidence w_i weight of the keypoint x_i:

$$\mathcal{L}_{Depth} = \sum_{x_i \in X_j} w_i \| \hat{D}(r_{ij}) - (P_j x_i) \cdot [0, 0, 1] \|_2^2 \tag{2.3.2}$$

The confidence weights are needed because SFM could produce spurious correspondences and/or unreliable keypoints. Therefore, the keypoints x_i are weighted according to their reprojection error e_{ij} estimated by the SFM, which is the distance in pixels between projected image coordinates $K_j P_j x_i$ and the detected keypoint in 2D. The confidence weight of a particular keypoint x_i is then computed using the its total reprojection error $e_i = \sum_j e_{ij}$:

$$w_i = \exp\left(-\left(\frac{e_i}{\overline{e}}\right)^2\right) \tag{2.3.3}$$

where $\overline{e}$ is the average absolute error over all keypoints in a scene.

The final training loss is given by the sum of the color loss, Equation 2.1.1, and the depth supervision loss:

$$\mathcal{L} \;=\; \mathcal{L}_{Color} + \lambda_D \mathcal{L}_{Depth} \tag{2.3.4}$$

where λ_D is an hyper-parameter to balance the two losses.

2.3.2 Pros and Cons of DS-NeRF

DS-NeRF offers an easy and effective way to speed up the training time and to reduce the number of training images. The only con of DS-NeRF is that it works uniquely with Structure-From-Motion solvers, which are useless if we already have the camera parameters and the depth.

However, the depth supervision loss leads to better performances and it can be implemented in different ways, with or without COLMAP.

2.4 MetaNeRF

Another interesting solution to speed up the training phase of NeRF has been proposed by Tancik et al. in the article *"Learned Initializations for Optimizing Coordinate-Based Neural Representations"* [17], where the main idea is to employ meta-learning techniques to learn the initial weight parameters for the network, for this reason their work is usually referred to as MetaNeRF.

The training phase of a neural network is usually slow because it has to solve an optimization problem which takes a large number of steps of gradient descent, but it has been proven that meta-learning can reduce the number of steps, reducing the training time [18].

The main concept behind meta-learning is to *"learn how to learn"*: using learned values for the initial weights, instead of standard random initialization, enables a faster convergence and better generalization.

2.4.1 The Meta-Learning Method

The two meta-learning algorithms employed are Model-Agnostic Meta Learning (MAML) [19] and Reptile [20], since they are easy to implement and use common standard gradient-based optimization, such as SGD (Stochastic Gradient Descent) and Adam [21].

Given a dataset of observations of signals T from the distribution $\mathcal{T}$, the task is to find the initial weights Θ_0^* which leads to the lowest final loss $L(\Theta_m)$ when optimizing a network f_Θ to represent a new unseen signal from the distribution $\mathcal{T}$:

$$\Theta_0^* = \arg\min_{\Theta_0} E_{T\sim\mathcal{T}}[L(\Theta_m(\Theta_0, T))] \tag{2.4.1}$$

Given the task T, computing the weights $\Theta_m(\Theta_0, T)$ requires m optimization steps, which are collectively referred to as inner loop. Normally, the initial weights Θ_0 are randomly chosen, but with MAML and Reptile algorithms, an outer loop of meta-learning is wrapped around the inner loop, in order to find a better initialization. Then, each outer loop samples a signal T_j from $\mathcal{T}$ and applies an update rule.

MAML applies the update rule:

$$\Theta_0^{j+1} = \Theta_0^j - \beta\nabla_\Theta L(\Theta_m(\Theta, T_j))|_{\Theta=\Theta_0^j} \tag{2.4.2}$$

while Reptile applies a rule which does not require calculating second-order gradients:

$$\Theta_0^{j+1} = \Theta_0^j - \beta\left(\Theta_m\left(\Theta_0^j, T_j\right) - \Theta_0^j\right) \tag{2.4.3}$$

Given a fixed number of iterations, MAML is able to find a better initialization than Reptile but it is more memory-intensive, therefore, Reptile can be unrolled for more inner loop steps in the same amount of time, which means it can focus over more different observations, leading to a better generalization.

In summary, this method is articulated over two phases:

- meta-learning phase: where MAML or Reptile are used with a training dataset of example tasks to optimize initial network weights for that class of signals;

- test-time optimization: where standard gradient-based optimization is used to fit the weights of a network to observations of a previously unseen signal from the same class.

MetaNeRF uses a simplified NeRF model which does not feed in the viewing direction and does not use the hierarchical sampling, while the MLP architecture remains the same of the regular NeRF. Furthermore, meta-learning techniques can be used not only for view synthesis, but for every coordinate-based networks such as image regression and CT reconstruction [17].

2.4.2 Pros and Cons of MetaNeRF

The main advantage of MetaNeRF is that the same set of meta-learned weights can be used to recover new objects of the same class with much less iterations and with better generalization. In addition, MAML and Reptile algorithms are easy to implement since they require just to add the outer loop, so they can be layered on top of existing implementations with few lines of code. The biggest limitation is that it requires a dataset of considerable size of example signals from the target distribution to obtain valuable initial weights, which it is hardly available for the majority of the use cases.

Chapter 3

NeRF Model and Experiments

In the previous chapter, we have seen what is NeRF and few related works on how the training and rendering time could be improved; now we will combine the above ideas to obtain a NeRF model less computationally demanding as possible during the training phase.

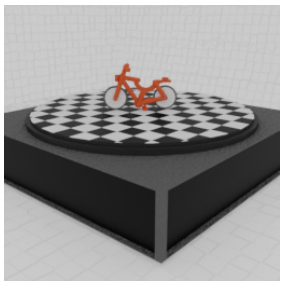
Firstly, we will describe the setup for our experiments, including the training data and network model; then, we will address the experiments that we have performed.

3.1 The Lego Dataset

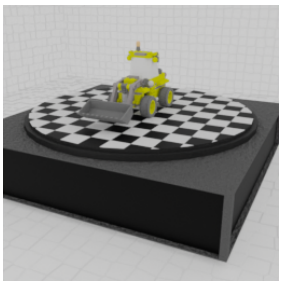
For our experiments, we have generated a synthetic dataset of eight scenes using Blender and BlenderProc[22], Figure 3.1.1.

For all the scenes we used the same setup: an object above a stand located inside a box. We have chosen to use this setup because in this way we are sure that the results of our experiments do not depend on the geometry of the scene, but just on the method applied.

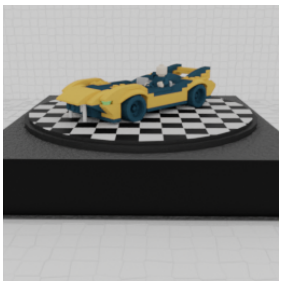
For each scene we have generated its corresponding bounding box, the intrinsics and 200 RGB images, each one with its pose and depth map, taken from different point of views from a dome located over the stand. The image resolution is low: 256x256; but in this way we have been able to perform more tests in less time, since we are interested in the convergence speed of our experiments, not the image quality.



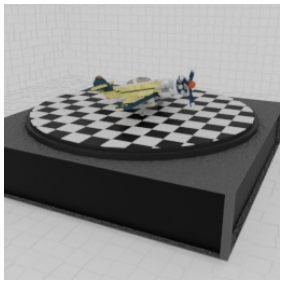
(a) Lego Bike



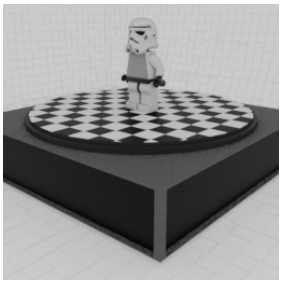
(b) Lego Bulldozer



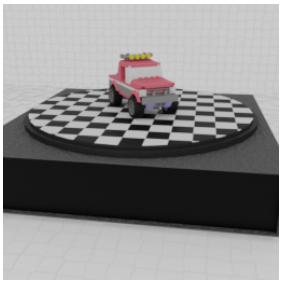
(c) Lego Car



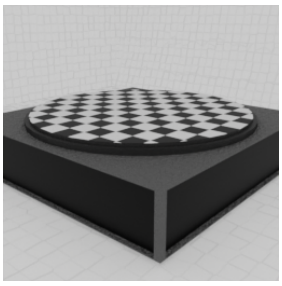
(d) Lego Plane



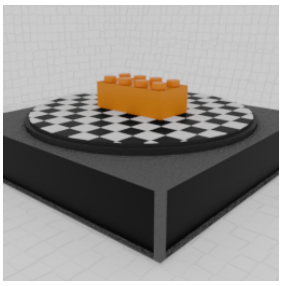
(e) Lego Trooper



(f) Lego Truck



(g) Empty scene



(h) Lego brick

Figure 3.1.1: The eight scenes used in our experiments.

3.2 The Model

Since we want to reduce both training and rendering time, for our model we have chosen to start from the work proposed by Reiser et al. in KiloNeRF [15], Section 2.2, since it already offers an efficient and effectively way to achieve real-time rendering.

We focused mainly on trying to accelerate the pretrain phase, which is the main bottleneck of all the pipeline, where a regular NeRF model, without hierarchical sampling, is trained for then be used as teacher in the distillation phase.

Our network architecture is thus the one described in Section 2.2.1, with an exception: we have modified the loss function to consider the depth too during the training, similarly to DS-NeRF[16], Section 2.3.

However, DS-NeRF has access to just a small set of pixels with known depth: the ones estimated by COLMAP, therefore the depth supervision is not employed everywhere, while in our model, since we have generated our dataset with the depth maps, Section 3.1, we can use a simplified depth loss function, which is the mean squared error between the depth estimated by the network $\hat{D}(r)$, Equation 2.3.1, and the ground truth value $D(r)$:

$$\mathcal{L}_{Depth} = \|\hat{D}(r) - D(r)\|_2^2 \tag{3.2.1}$$

The final loss function is given by the Equation 2.3.4, with $\lambda_D = 1$.

Additionally, our model has been incorporated inside an automated pipeline to automatically train several NeRFs of different scenes in succession, in this way it is easy to generate a dataset of NeRFs without human intervention. Each NeRF has been trained for 300k iterations, 256 rays per iteration and 384 samples per ray, for about five hours on a Nvidia GeForce RTX 2080 Ti, an Intel i9-9900K CPU and 64GB RAM.

3.3 Experiments

To find the best way to speed up the convergence speed, we have proceeded with the following experiments: first we have trained the NeRFs with a different number of training images without depth supervision, to find how big the training dataset should be and to have the baseline results.

Then, we have trained the same scenes with depth supervision, using the optimal number of training images, to find out how much the convergence speed is improved.

Finally, we have trained the scenes, with depth supervision, starting from pre-trained weights of different scenes, to see how it influences the training.

3.3.1 Number of Images

NeRF requires a large set of training images from different point of views to correctly learn the geometry of the scene, otherwise it overfits and the novel views present several artifacts. The training images must be taken uniformly around the scene, otherwise the network will be able to correctly synthesise just the novel views around the training images it has seen the most.

For this experiment, we have trained six different scenes: the Lego Bike, Figure 3.1.1a, the Lego Car, Figure 3.1.1c, the Lego Bulldozer, Figure 3.1.1b, the Lego Plane, Figure 3.1.1d, the Lego Truck, Figure 3.1.1f and the Lego Trooper, Figure 3.1.1e.

Each scene has been trained with 20, 50 and 100 images. Our goal is to find the number of images necessary to train correctly our NeRFs, the less the better.

3.3.2 Depth Supervision

We have seen in DS-NeRF[16], Section 2.3, that the depth supervision helps the network to understand the geometry of the scene, leading to a faster convergence speed. We want to check if employing the depth supervision to all the pixels of

the scene, and not just a subset, leads to similar results.

For this experiment, we have trained the same six scenes of the experiment described in Section 3.3.1, with 100 RGB images, including their depth maps for the depth supervision.

We expect that the scenes trained with depth supervision should converge faster with a better understanding of the geometry of the scene.

3.3.3 Weight Initialization

We have seen that meta-learning is able to accelerate the convergence speed of NeRF [17], Section 2.4, but it can be used only if several scenes of different objects belonging to the same class are available.

For our experiment we have tried something similar yet different: we have trained from scratch, with depth supervision, the NeRFs of the Empty, Figure 3.1.1g and the Lego Brick, Figure 3.1.1h, scenes; then we have trained the six scenes of the previous experiments starting from their pretrained weights.

The main idea is that the networks should converge faster since it already starts from learned weights of scenes with similar setups.

Chapter 4

Results

For each of our experiments, we report in this chapter the qualitative and quantitative results for each scene, along with the related plots.

During the training phases of the networks, a test set composed of ten novel views, not employed at training time, is evaluated every ten thousands iterations.

At each evaluation, the RGB image, with its corresponding depth map, is generated for each test view, and different metrics are computed to measure the gap between the ground truth and the images generated.

We have employed three different metrics to establish the quality of the RGB images generated by the networks with respect to the ground truth: the "Peak Signal-to-Noise Ratio" (PSNR), the higher the better, the "Structural Similarity Index Measure" (SSIM) [23], the higher the better, and the "Learned Perceptual Image Patch Similarity" (LPIPS) [24], the lower the better.

We have reported just the final quantitative and qualitative results, while the intermediate ones have been employed to generate the plots.

4.1 Number of Images

The quantitative results of the number of images experiment are reported in Table 4.1, while the qualitative results and the related plots of the different scenes are shown in Figures 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5 and 4.1.6.

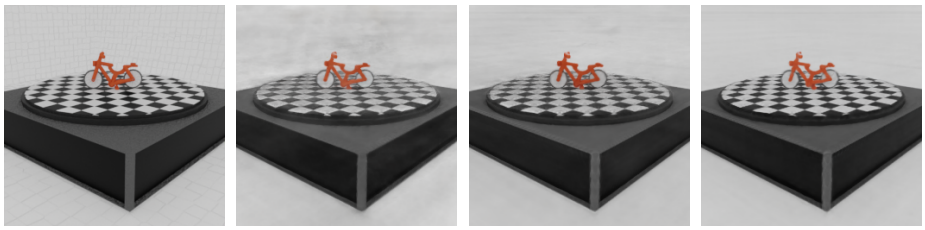
As we can see from the reported results, with just 20 images the network overfits and struggles to correctly understand the geometry of scene leading to several artifacts while rendering novel views never seen before.

However, between 50 and 100 images the results are similar, the network does not overfits and it is able to correctly render the novel views. With 100 images, rather than 50, the results are slightly better in general, but the gap between the results with 50 and 100 images is not as wide as the one between 20 and 50 images.

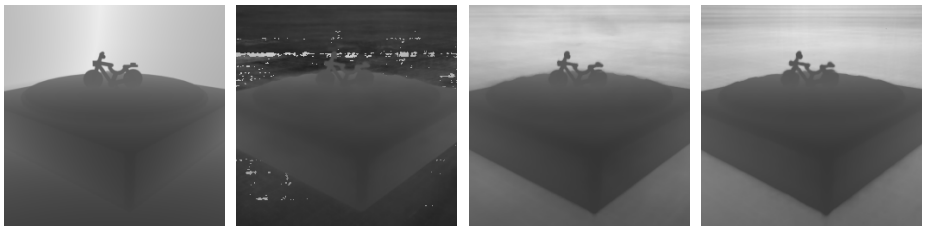
In conclusion, we have discovered that, for our setup, at least 50 images are needed to correctly train NeRF; but for the following experiments we will use 100 images, since it is the number of images which achieved the best results.

PSNR ↑			
Scene	20 Images	50 Images	100 Images
Lego Bike	25.44	30.68	**35.05**
Lego Bulldozer	23.23	31.90	**35.02**
Lego Car	24.19	**32.58**	32.20
Lego Plane	24.85	30.70	**32.60**
Lego Trooper	22.98	31.05	**33.19**
Lego Truck	23.47	31.69	**32.46**
LPIPS ↓			
Scene	20 Images	50 Images	100 Images
Lego Bike	0.07771	0.03245	**0.02377**
Lego Bulldozer	0.08185	0.02573	**0.01910**
Lego Car	0.07558	**0.02720**	0.02907
Lego Plane	0.07071	0.03190	**0.02756**
Lego Trooper	0.13830	0.03158	**0.02487**
Lego Truck	0.06781	**0.03253**	0.03421
SSIM ↑			
Scene	20 Images	50 Images	100 Images
Lego Bike	0.8376	0.9068	**0.9350**
Lego Bulldozer	0.8201	0.9234	**0.9448**
Lego Car	0.8084	0.9221	**0.9244**
Lego Plane	0.8397	0.9155	**0.9291**
Lego Trooper	0.7871	0.9122	**0.9338**
Lego Truck	0.8383	0.9148	**0.9172**

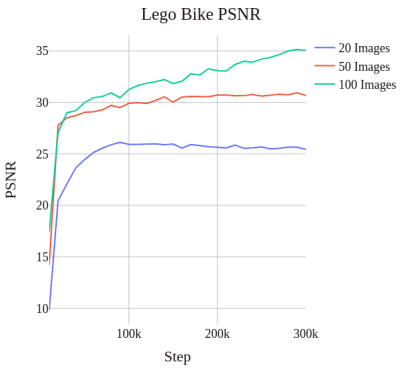
Table 4.1: Quantitative results of the number of images experiment for the number of images experiment.

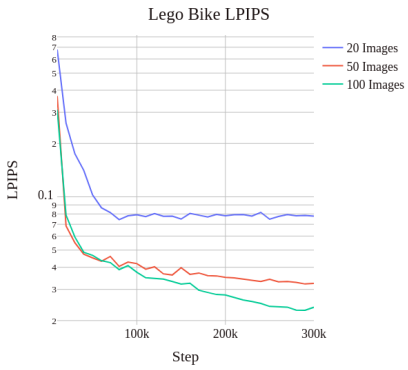


(a) GT (b) 20 Images (c) 50 Images (d) 100 Images



(e) GT (f) 20 Images (g) 50 Images (h) 100 Images





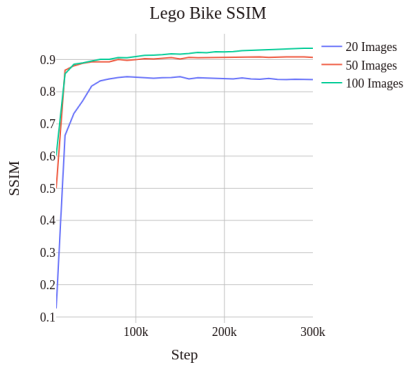
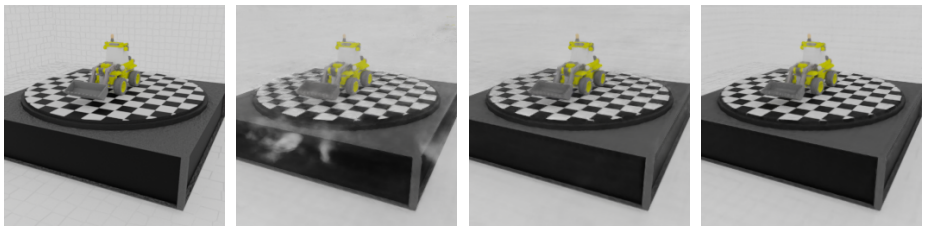
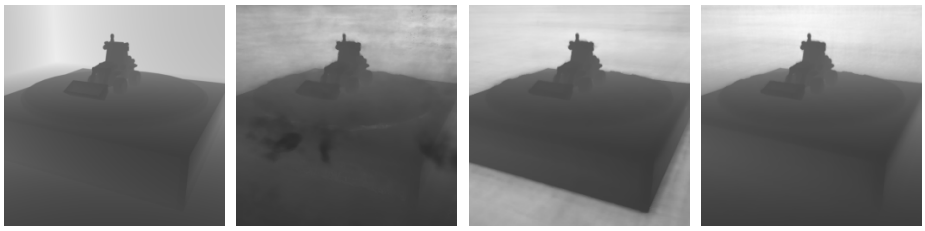


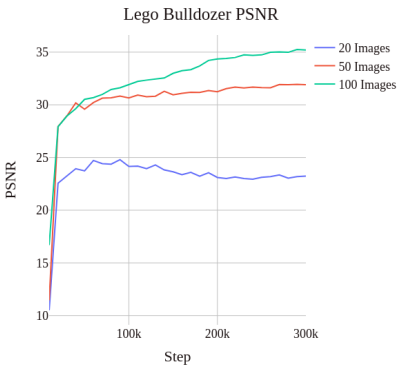
Figure 4.1.1: Qualitative results and related plots for the Lego Bike scene for the number of images experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.

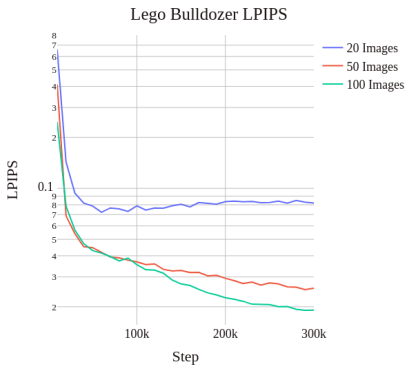


(a) GT (b) 20 Images (c) 50 Images (d) 100 Images



(e) GT (f) 20 Images (g) 50 Images (h) 100 Images





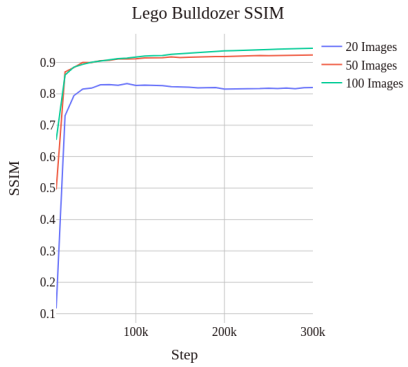
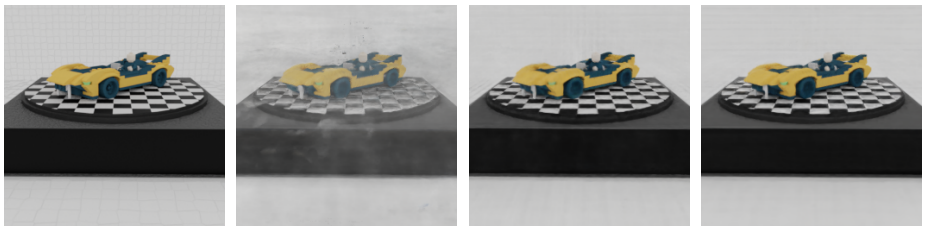
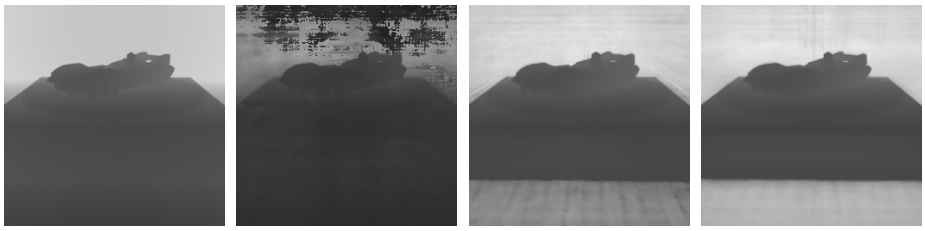


Figure 4.1.2: Qualitative results and related plots for the Lego Bulldozer scene for the number of images experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.



(a) GT (b) 20 Images (c) 50 Images (d) 100 Images



(e) GT (f) 20 Images (g) 50 Images (h) 100 Images

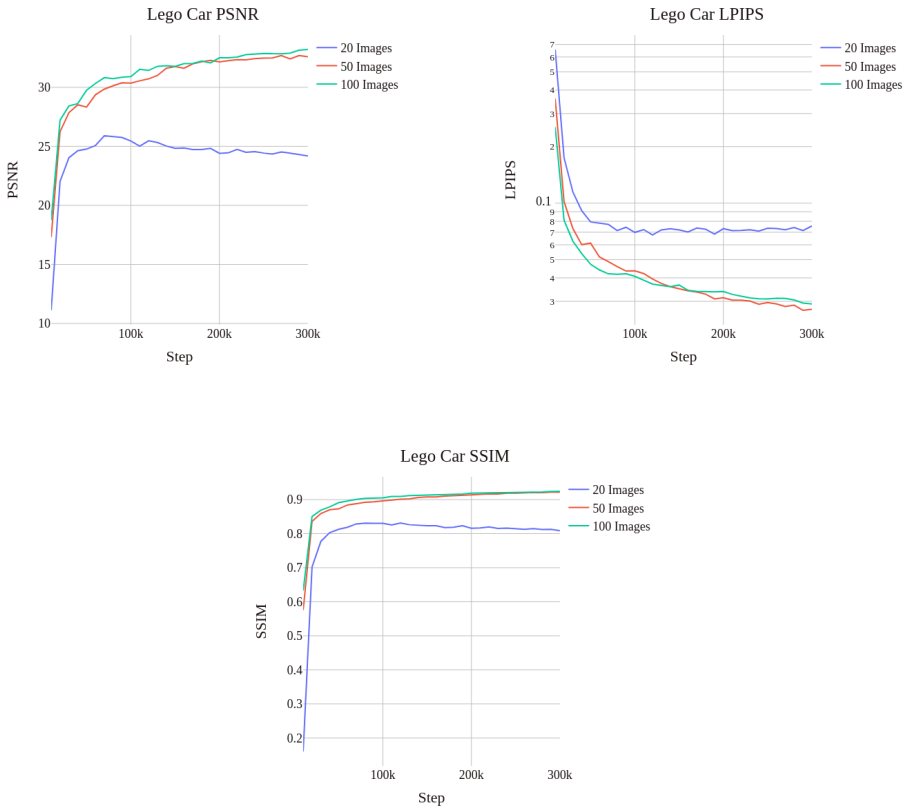
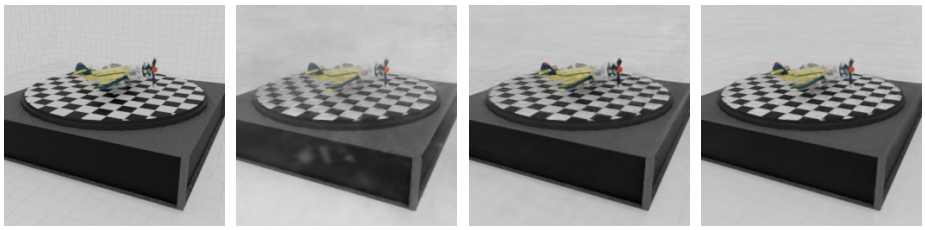
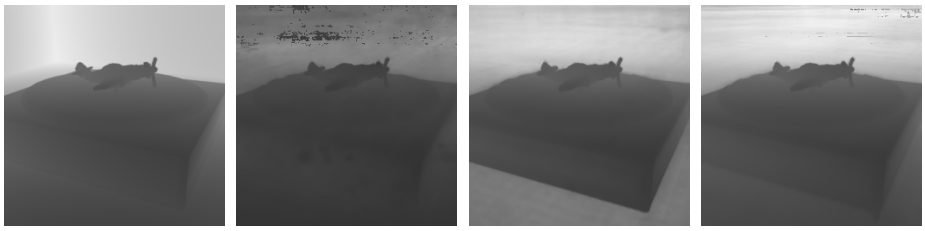


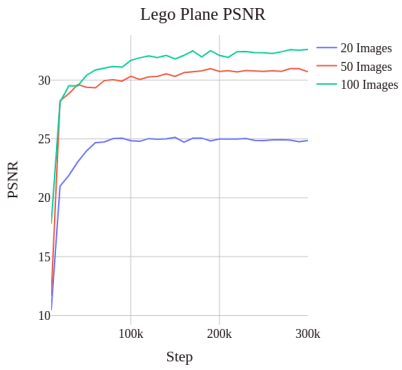
Figure 4.1.3: Qualitative results and related plots for the Lego Car scene for the number of images experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.

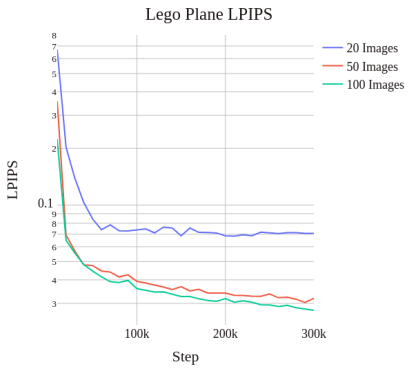


(a) GT (b) 20 Images (c) 50 Images (d) 100 Images



(e) GT (f) 20 Images (g) 50 Images (h) 100 Images





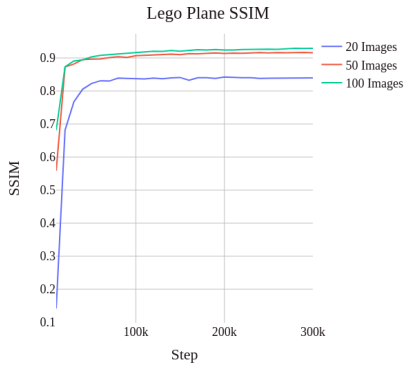
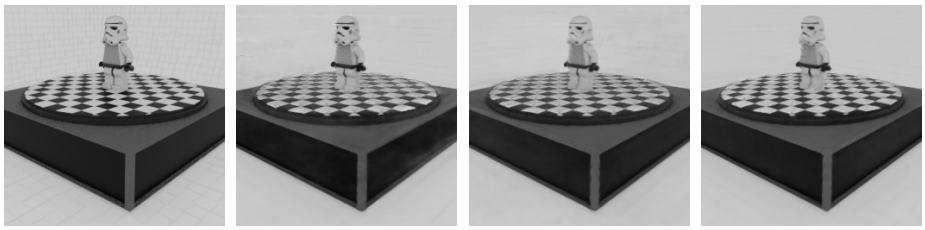
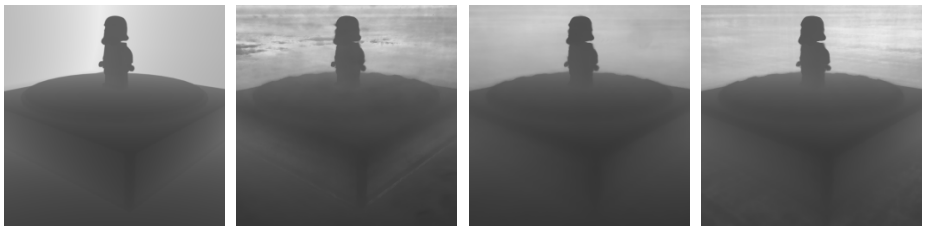


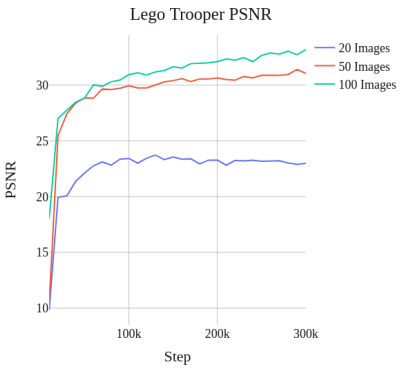
Figure 4.1.4: Qualitative results and related plots for the Lego Plane scene for the number of images experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.

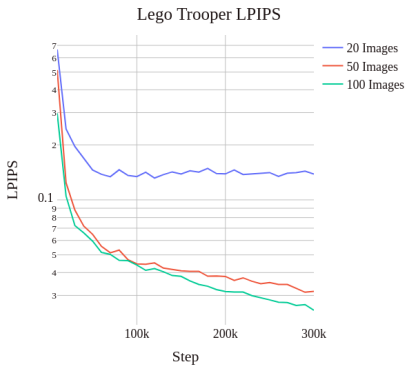


(a) GT (b) 20 Images (c) 50 Images (d) 100 Images



(e) GT (f) 20 Images (g) 50 Images (h) 100 Images





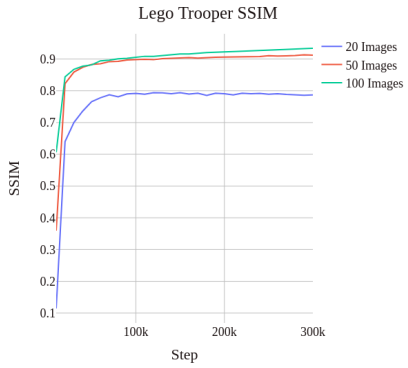
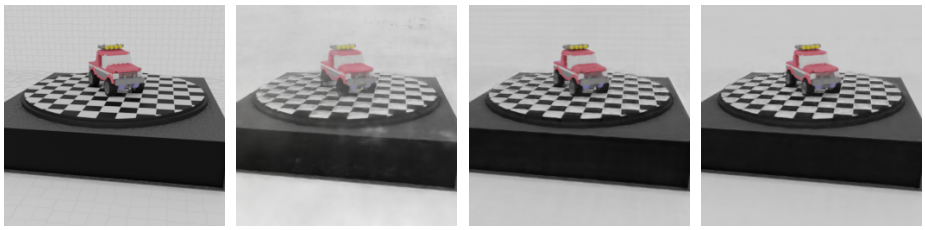
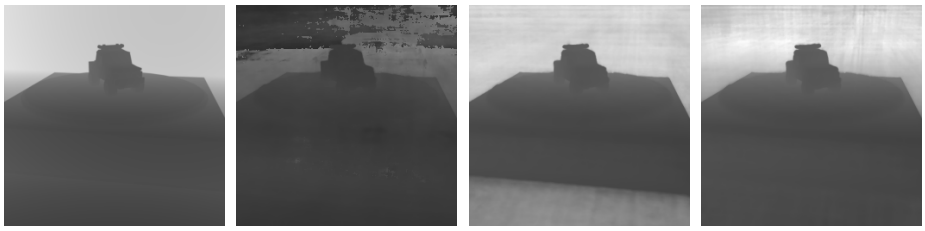


Figure 4.1.5: Qualitative results and related plots for the Lego Trooper scene for the number of images experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.



(a) GT (b) 20 Images (c) 50 Images (d) 100 Images



(e) GT (f) 20 Images (g) 50 Images (h) 100 Images

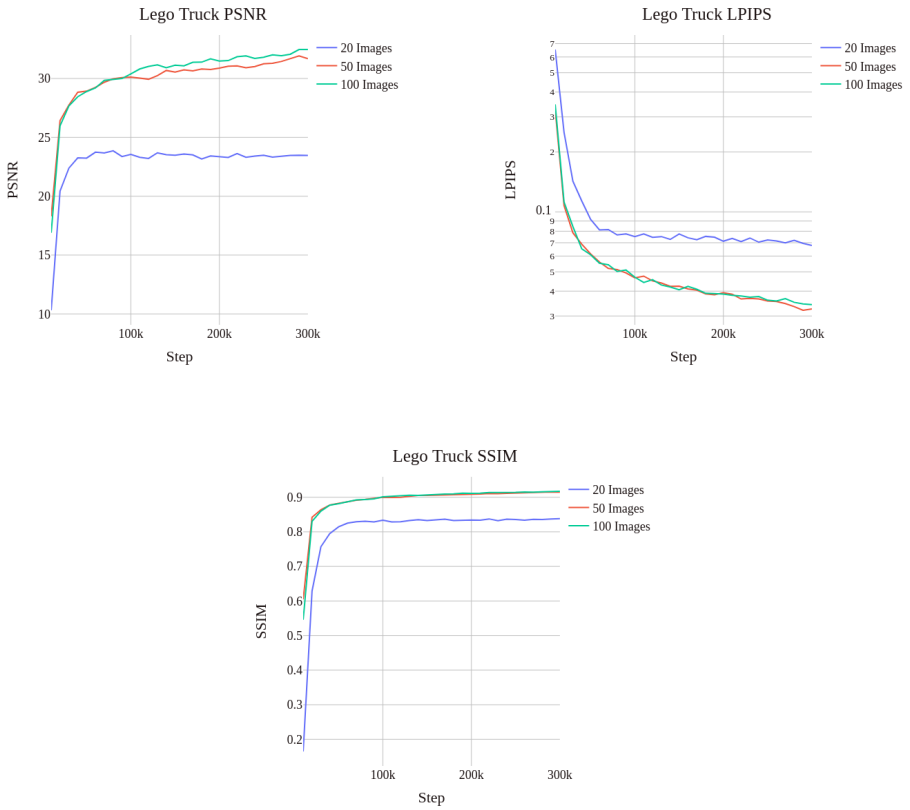


Figure 4.1.6: Qualitative results and related plots for the Lego Truck scene for the number of images experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.

4.2 Depth Supervision

The quantitative results of the depth supervision (DS) experiment are reported in Table 4.2, while the qualitative ones, and the related plots, are reported in Figures 4.2.1, 4.2.2, 4.2.3, 4.2.4, 4.2.5 and 4.2.6.

We can see that the scenes trained with depth supervision achieve in general a slightly better quality than the ones trained from scratch.

The depth maps without depth supervision are completely wrong and present several artifacts, which means that the network is not able to retrieve the undergoing geometry of the scene, in particular on the background and the floor, resulting in a longer training to reach the same quality.

In conclusion, we have discovered that employing the depth supervision leads to a slightly faster convergence speed, as reported in the plots, therefore, we will use the depth supervision also for our next and final experiment.

PSNR ↑		
Scene	Without DS	With DS
Lego Bike	35.05	**35.50**
Lego Bulldozer	**35.02**	34.87
Lego Car	32.20	**33.19**
Lego Plane	32.60	**35.08**
Lego Trooper	33.19	**33.77**
Lego Truck	32.46	**33.40**
LPIPS ↓		
Scene	Without DS	With DS
Lego Bike	0.02377	**0.02024**
Lego Bulldozer	**0.01910**	0.02116
Lego Car	0.02907	**0.02563**
Lego Plane	0.02756	**0.02212**
Lego Trooper	**0.02487**	0.02552
Lego Truck	0.03421	**0.02773**
SSIM ↑		
Scene	Without DS	With DS
Lego Bike	0.9350	**0.9430**
Lego Bulldozer	**0.9448**	0.9411
Lego Car	0.9244	**0.9361**
Lego Plane	0.9291	**0.9418**
Lego Trooper	0.9338	**0.9346**
Lego Truck	0.9172	**0.9339**

Table 4.2: Quantitative results of the depth supervision experiment

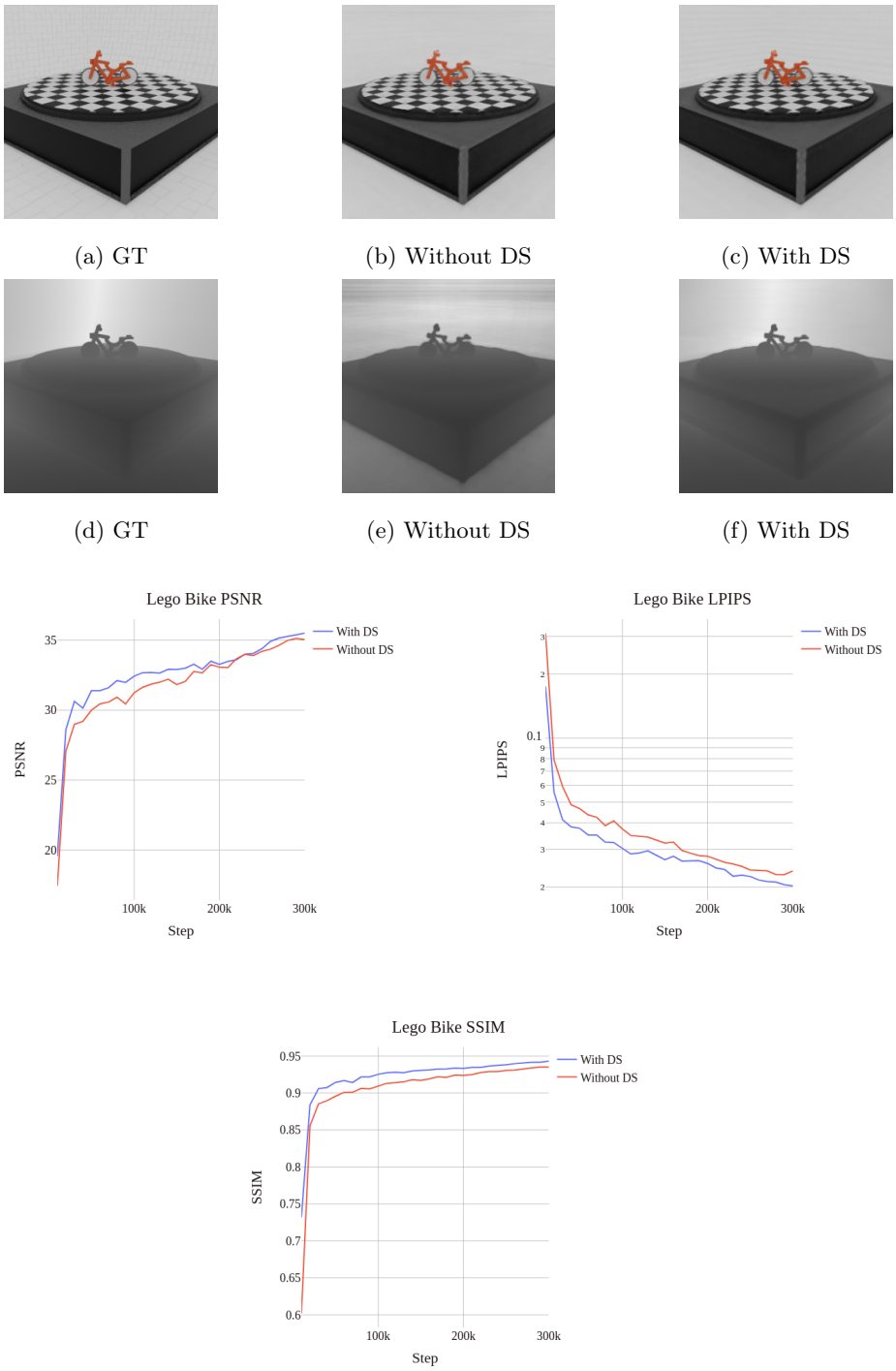
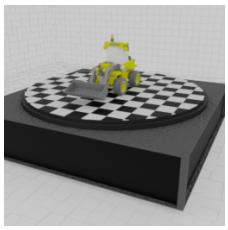
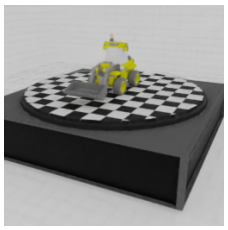


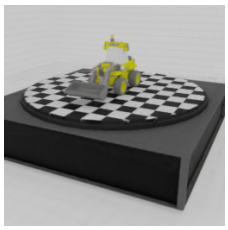
Figure 4.2.1: Qualitative results and related plots for the Lego Bike scene for the depth supervision experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.



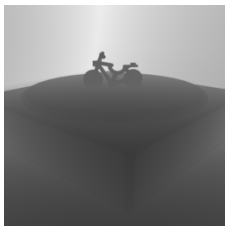
(a) GT



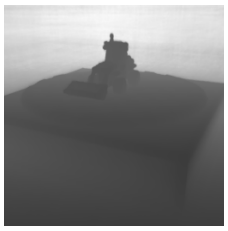
(b) Without DS



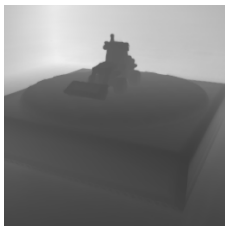
(c) With DS



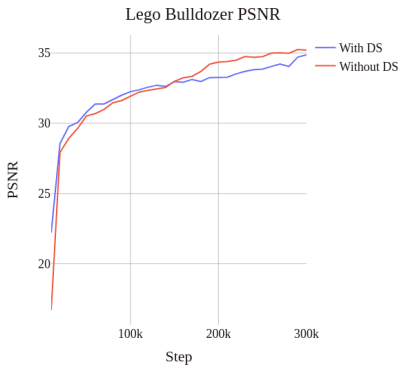
(d) GT

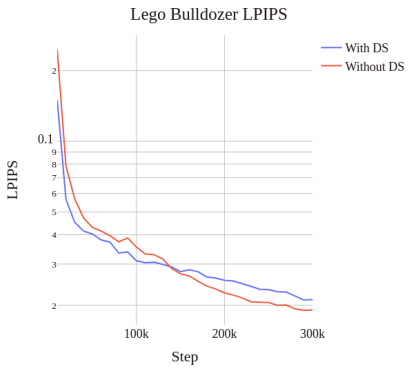


(e) Without DS



(f) With DS





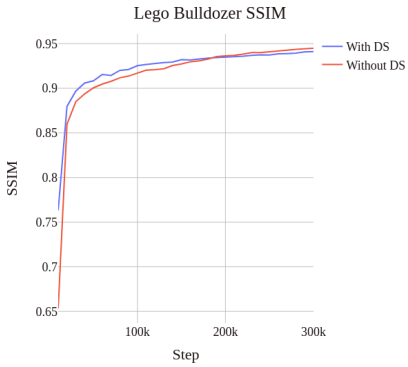
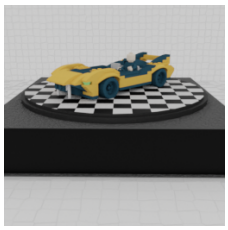


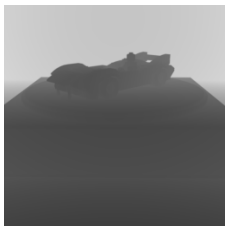
Figure 4.2.2: Qualitative results and related plots for the Lego Bulldozer scene for the depth supervision experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.



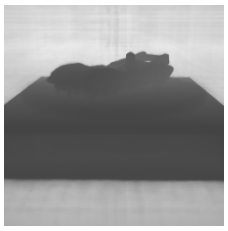
(a) GT

(b) Without DS

(c) With DS

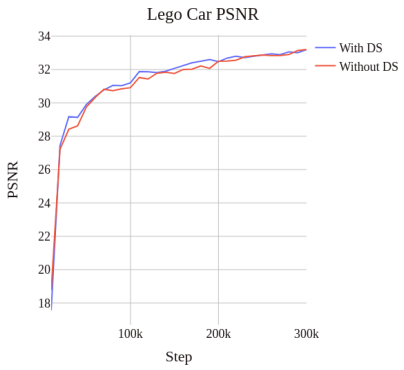


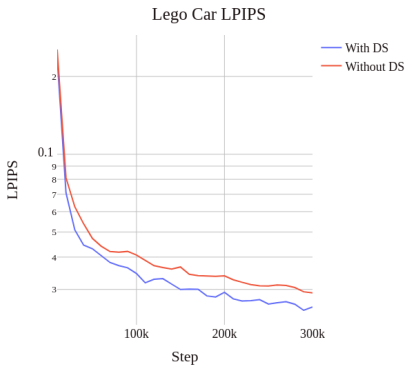
(d) GT



(e) Without DS

(f) With DS





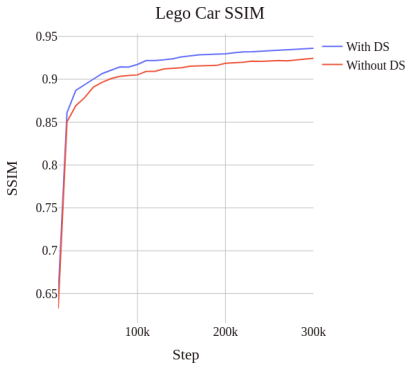
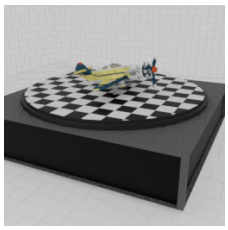
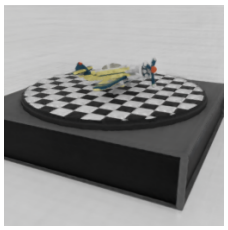


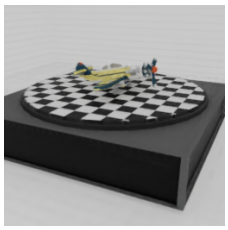
Figure 4.2.3: Qualitative results and related plots for the Lego Car scene for the depth supervision experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.



(a) GT



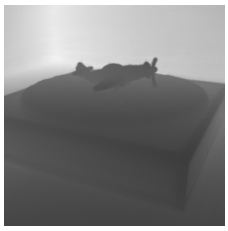
(b) Without DS



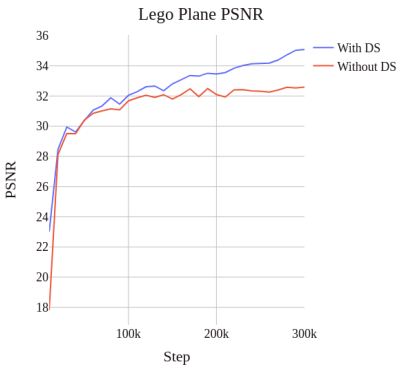
(c) With DS

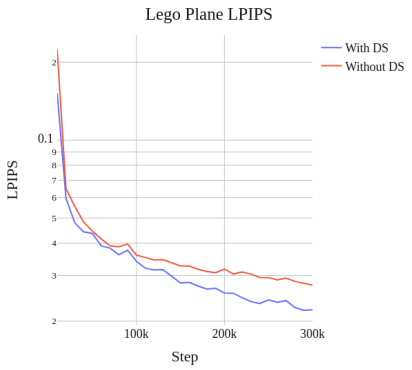
(d) GT

(e) Without DS



(f) With DS





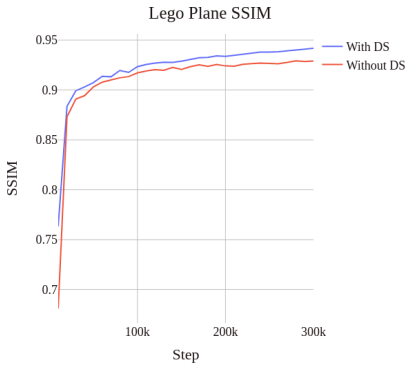
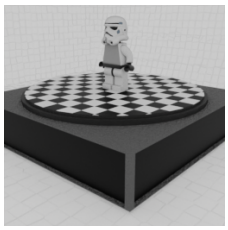


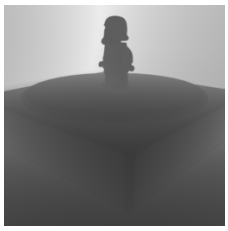
Figure 4.2.4: Qualitative results and related plots for the Lego Plane scene for the depth supervision experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.



(a) GT

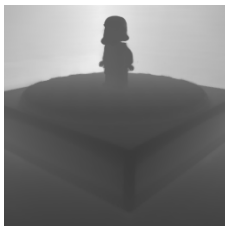
(b) Without DS

(c) With DS

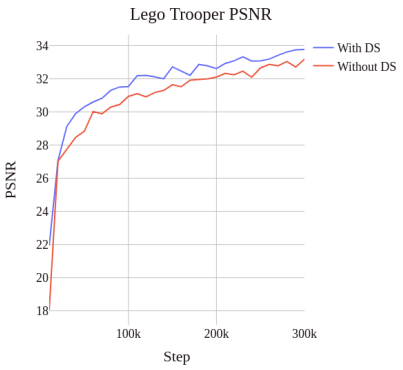


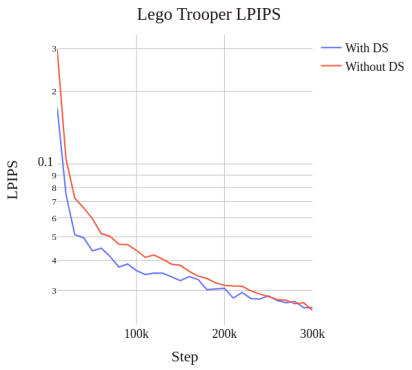
(d) GT

(e) Without DS



(f) With DS





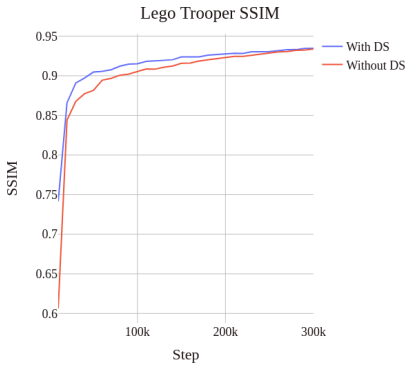
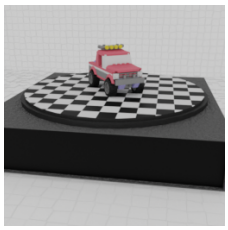
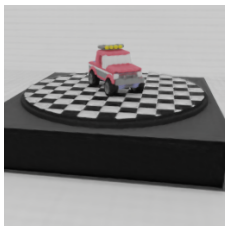


Figure 4.2.5: Qualitative results and related plots for the Lego Trooper scene for the depth supervision experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.

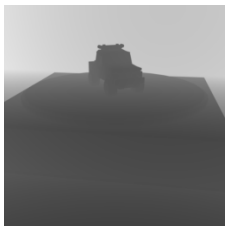


(a) GT

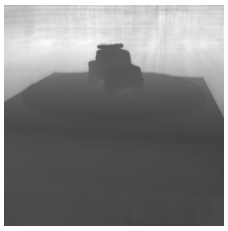
(b) Without DS



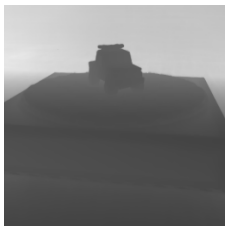
(c) With DS



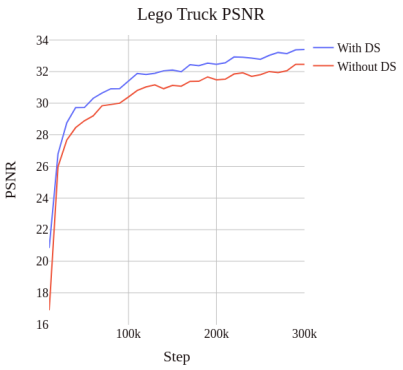
(d) GT

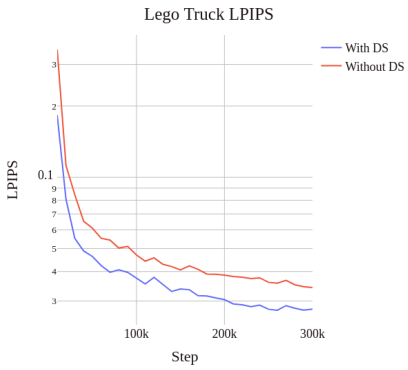


(e) Without DS



(f) With DS





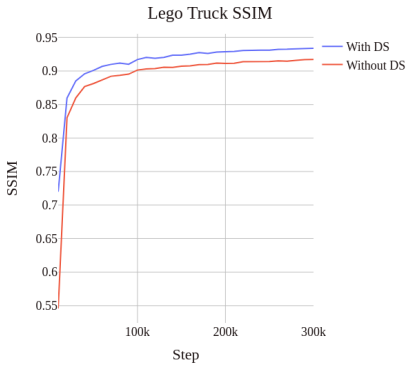


Figure 4.2.6: Qualitative results and related plots for the Lego Truck scene for the depth supervision experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.

4.3 Weight Initialization

The scenes for this experiment have been trained starting from the pre-trained weights of the Lego Brick and Empty scene. These two scenes have been trained just like the scenes of the depth supervision experiment, and their final results are reported in Table 4.3. We can see that the results are in line with the ones previously reported.

PSNR ↑	
Scene	With DS
Lego Brick	36.08
Empty	35.21
LPIPS ↓	
Scene	With DS
Lego Brick	0.01826
Empty	0.02119
SSIM ↑	
Scene	With DS
Lego Brick	0.9438
Empty	0.9409

.

Table 4.3: Quantitative results of the Lego Brick and Empty scenes trained with depth supervision.

The quantitative results of the weight initialization (WI) experiment are reported in Table 4.4, while the qualitative ones, and the related plots, are reported in Figures 4.3.1, 4.3.2, 4.3.3, 4.3.4, 4.3.5 and 4.3.6.

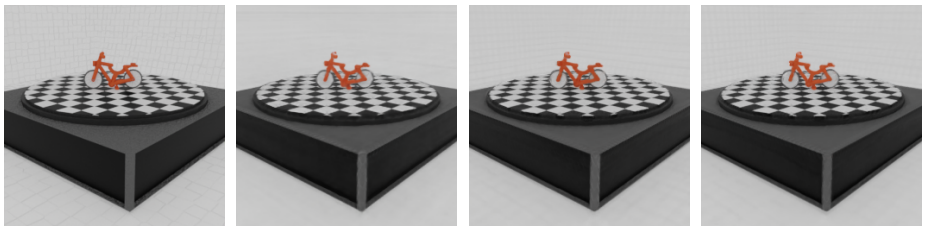
The results are impressive, from the plots we can see that the scenes trained with weight initialization reach the same results of the previous experiments, but 3 to 5 times faster (30 minutes against 5 hours at the best results). Furthermore, we can see from the table that the final results achieve better outcomes too; in particular, the scenes trained from the Lego Brick perform slightly better,

probably because the network has already learned that the density is higher in that region.

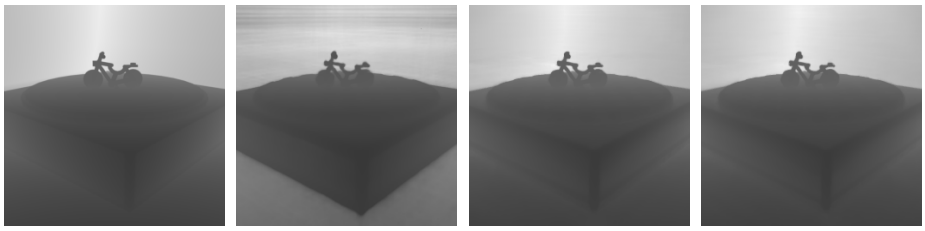
In conclusion, starting from initialized weights of different scenes with a similar setup increase considerably the convergence speed on new scenes, reaching a higher quality for the same number of iterations.

PSNR ↑				
Scene	Without DS	With DS	WI from Empty	WI from Brick
Lego Bike	35.05	35.50	36.58	**36.93**
Lego Bulldozer	35.02	34.87	35.69	**36.18**
Lego Car	32.20	33.19	34.84	**35.26**
Lego Plane	32.60	35.08	35.45	**36.03**
Lego Trooper	33.19	33.77	36.58	**36.88**
Lego Truck	32.46	33.40	36.18	**36.51**
LPIPS ↓				
Scene	Without DS	With DS	WI from Empty	WI from Brick
Lego Bike	0.02377	0.02024	0.01572	**0.01480**
Lego Bulldozer	0.01910	0.02116	0.01697	**0.01567**
Lego Car	0.02907	0.02563	0.02107	**0.01889**
Lego Plane	0.02756	0.02212	0.01882	**0.01725**
Lego Trooper	0.02487	0.02552	0.01588	**0.01517**
Lego Truck	0.03421	0.02773	0.01728	**0.01618**
SSIM ↑				
Scene	Without DS	With DS	WI from Empty	WI from Brick
Lego Bike	0.9350	0.9430	0.9524	**0.9538**
Lego Bulldozer	0.9448	0.9411	0.9488	**0.9521**
Lego Car	0.9244	0.9361	0.9395	**0.9437**
Lego Plane	0.9291	0.9418	0.9485	**0.9507**
Lego Trooper	0.9338	0.9346	0.9508	**0.9524**
Lego Truck	0.9172	0.9339	0.9499	**0.9512**

Table 4.4: Quantitative results of the weight initialization experiment



(a) GT (b) Without DS (c) WI from Empty (d) WI from Brick



(e) GT (f) Without DS (g) WI from Empty (h) WI from Brick

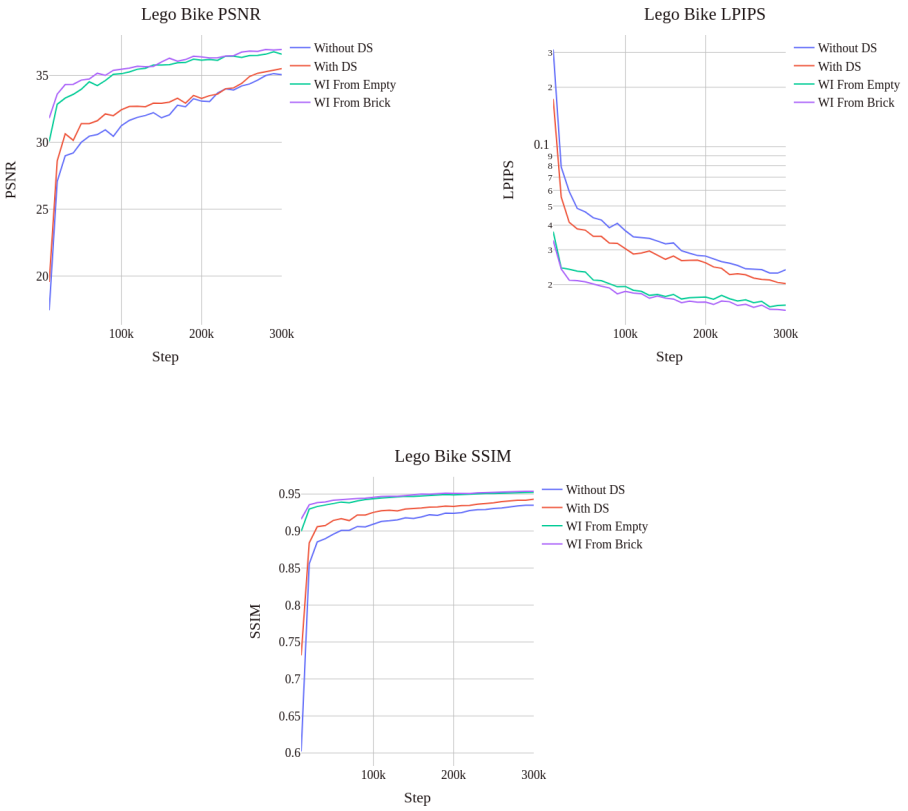
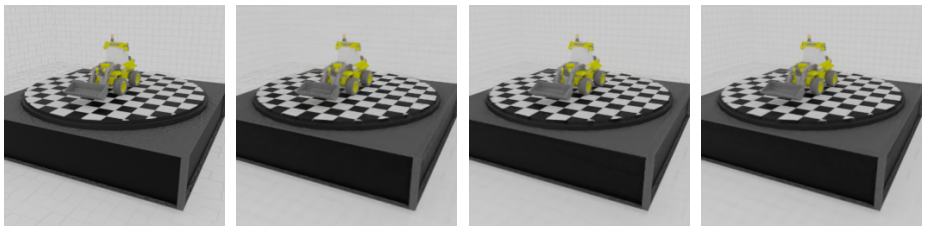
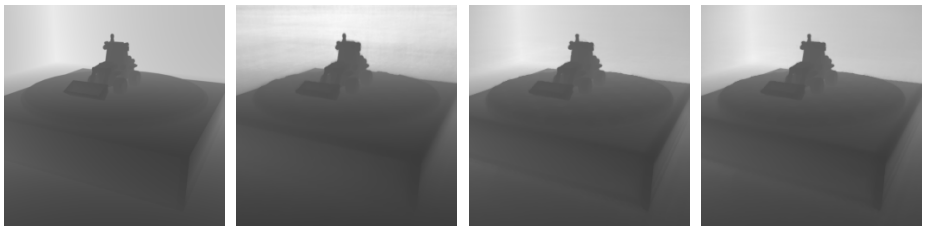


Figure 4.3.1: Qualitative results and related plots for the Lego Bike scene for the weight initialization experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.



(a) GT (b) Without DS (c) WI from Empty (d) WI from Brick



(e) GT (f) Without DS (g) WI from Empty (h) WI from Brick

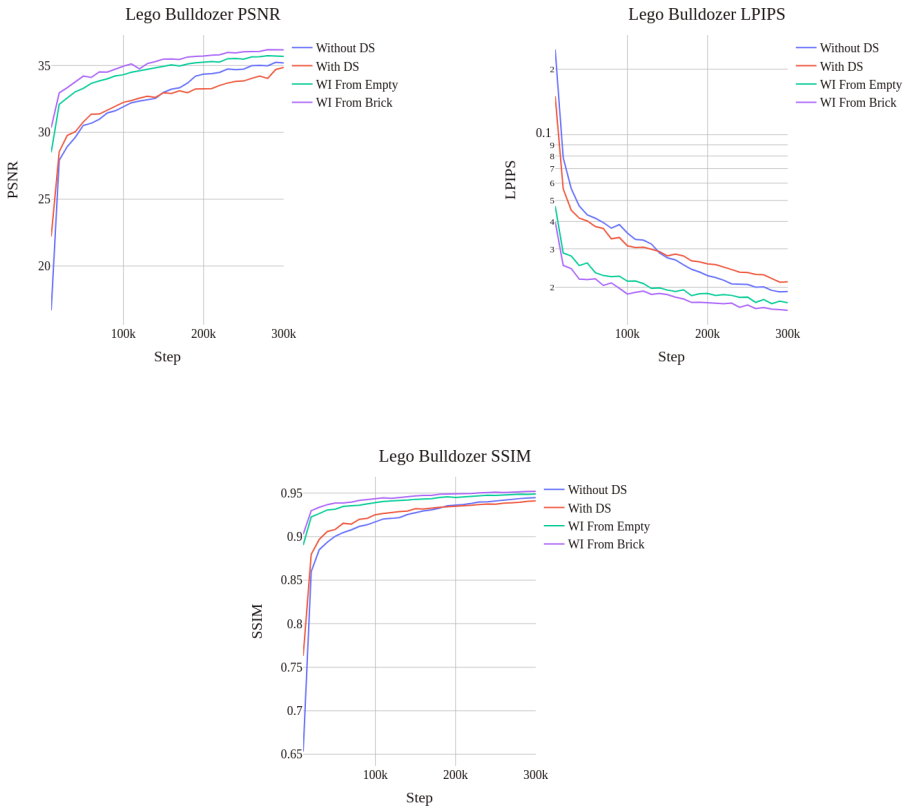
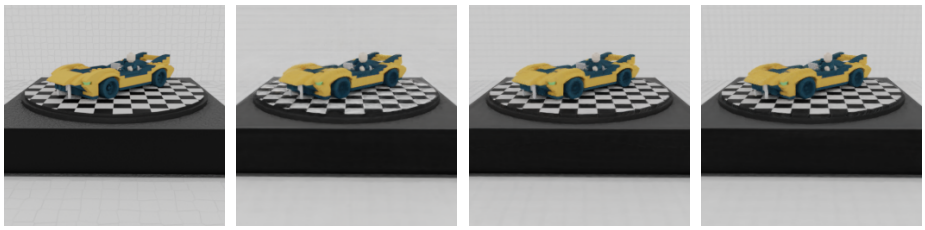
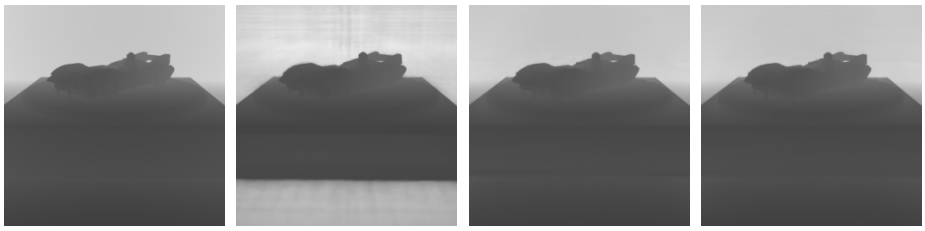


Figure 4.3.2: Qualitative results and related plots for the Lego Bulldozer scene for the weight initialization experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.



(a) GT (b) Without DS (c) WI from Empty (d) WI from Brick



(e) GT (f) Without DS (g) WI from Empty (h) WI from Brick

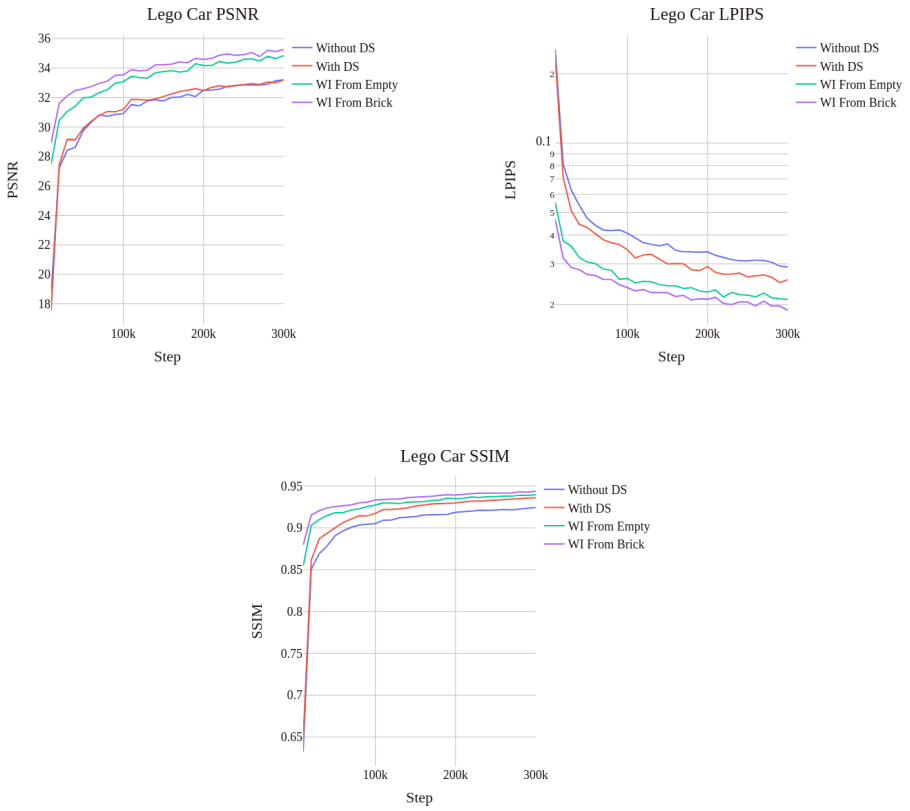
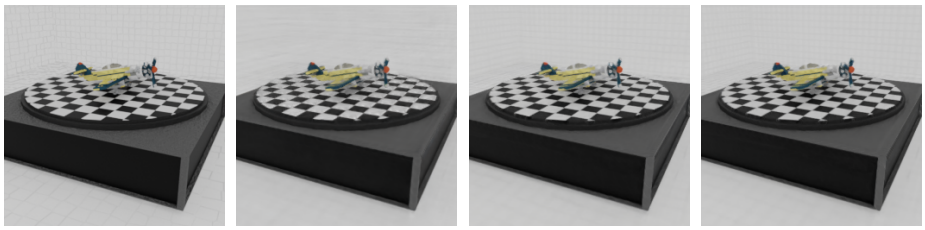
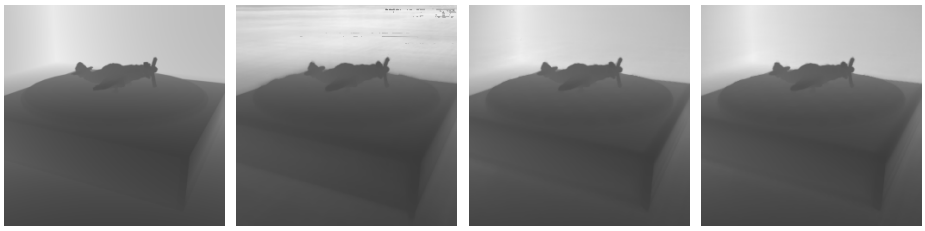


Figure 4.3.3: Qualitative results and related plots for the Lego Car scene for the weight initialization experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.



(a) GT (b) Without DS (c) WI from Empty (d) WI from Brick



(e) GT (f) Without DS (g) WI from Empty (h) WI from Brick

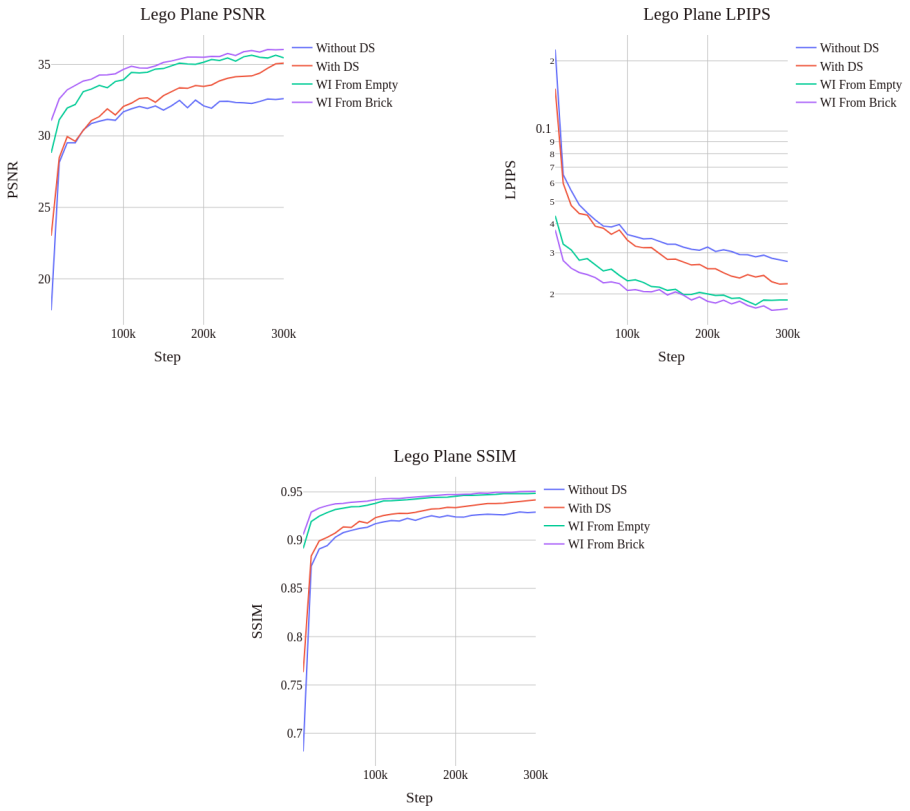
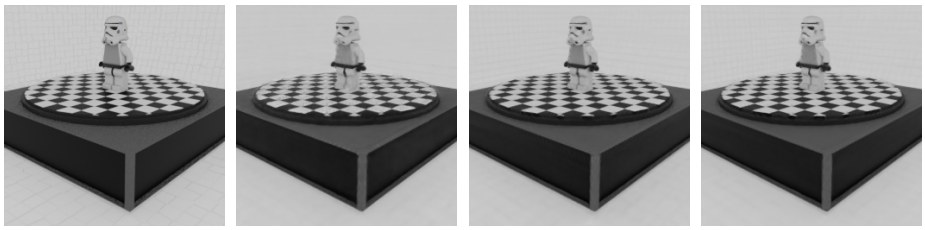
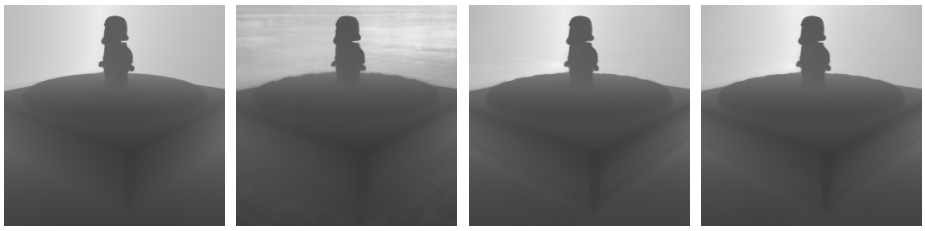


Figure 4.3.4: Qualitative results and related plots for the Lego Plane scene for the weight initialization experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.



(a) GT (b) Without DS (c) WI from Empty (d) WI from Brick



(e) GT (f) Without DS (g) WI from Empty (h) WI from Brick

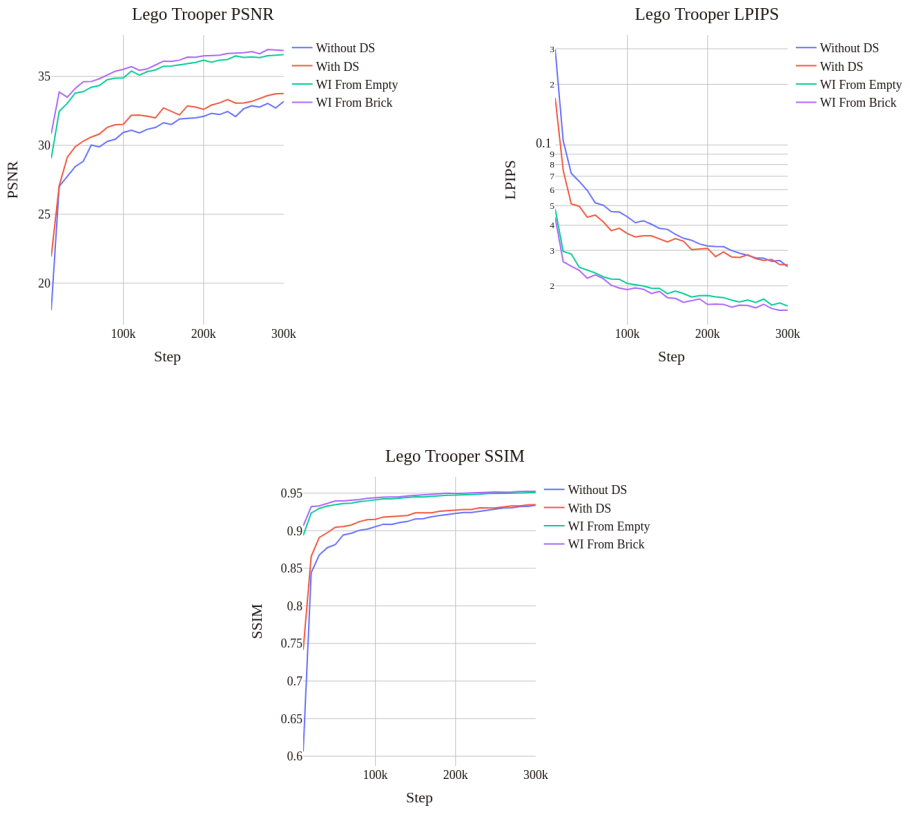
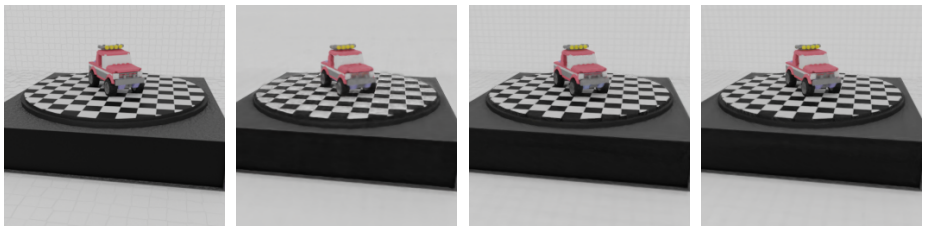
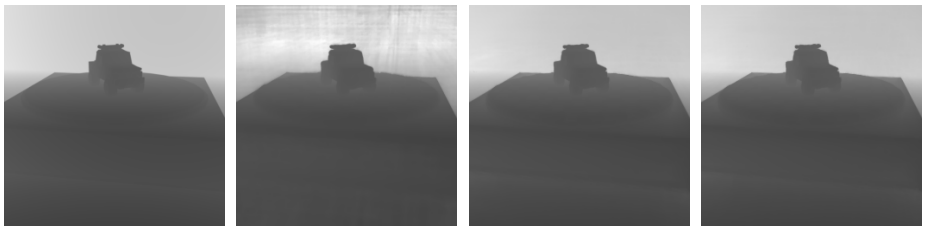


Figure 4.3.5: Qualitative results and related plots for the Lego Trooper scene for the weight initialization experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.



(a) GT (b) Without DS (c) WI from Empty (d) WI from Brick



(e) GT (f) Without DS (g) WI from Empty (h) WI from Brick

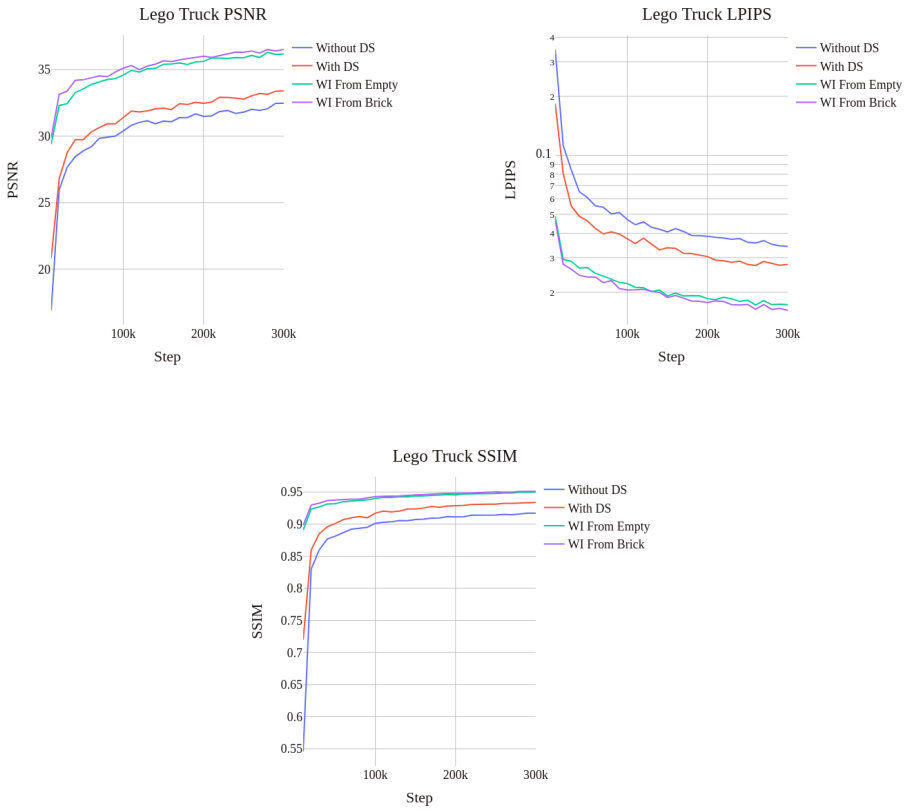


Figure 4.3.6: Qualitative results and related plots for the Lego Truck scene for the weight initialization experiment. First row: GT and generated novel views from the same pose of the GT. Second row: GT depth map and generated depth maps from the same pose of the GT. Third and fourth rows: plots of the different metrics on the test set.

Chapter 5

Conclusion and Future Works

Neural Radiance Fields will be probably deeply used in the future, since they are able to store with photo-realistic quality both synthetic and real-world scenes just with few megabytes, much less than the storage needed to store their training images.

However, training a NeRF requires time and computation, which are never enough; therefore, in this project we have proposed a solution to obtain a training pipeline less computationally needing as possible, from the training images to the final result. Our tests showed how many images are needed to correctly train a NeRF, in this way we can save time while generating the dataset of the scenes, and how we can speed up the training time by using weight initialization and depth supervision techniques. Instead of focusing on custom code to achieve the best from the hardware, we have found general ideas which can work under any hardware architecture, therefore more generally applicable.

We have shown that depth supervision is a cheap and easy way to boost the training speed, but it can be performed only if the ground truth of the depth is available, which could be given by stereo cameras, depth sensors and so on.

In addition, we have shown that it is not necessary to start from scratch each time we want to create a NeRF, instead, we can just train one NeRF and use it as starting point for all the others, especially if we want to create a dataset of

NeRFs of different objects starting from similar setups.

However, there is still a lot of work to do: even with our solution, training a NeRF remains a slow procedure which require both custom code and ideas like ours to make NeRF an every-day user technology.

For example, DONeRF [25] by Neff et al. employs a "depth oracle network" to guide sample placement, in this way less samples are necessary along the rays and the computation time is reduced. Alternatively, AutoInt [26] by Lindell et al. computes the volume rendering integral in a more efficient way, improving the trade-off between rendering speed and image quality.

All these ideas, and more, could be combined in one NeRF model, able to train and render NeRFs in just few minutes or seconds, making this technology suitable for modern day applications.

List of Figures

List of Tables

Bibliography

[1] Gustavo Patow and Xavier Pueyo, «A survey of inverse rendering problems», in *Computer graphics forum*, Wiley Online Library, vol. 22, 2003, pp. 663–687.

[2] Ayush Tewari, Justus Thies, Ben Mildenhall, *et al.*, «Advances in neural rendering», *arXiv preprint arXiv:2111.05849*, 2021.

[3] SM Ali Eslami, Danilo Jimenez Rezende, Frederic Besse, *et al.*, «Neural scene representation and rendering», *Science*, vol. 360, no. 6394, pp. 1204–1210, 2018.

[4] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng, «Nerf: Representing scenes as neural radiance fields for view synthesis», in *European conference on computer vision*, Springer, 2020, pp. 405–421.

[5] Frank Dellaert and Lin Yen-Chen, «Neural volume rendering: Nerf and beyond», *arXiv preprint arXiv:2101.05204*, 2020.

[6] Thomas Schops, Viktor Larsson, Marc Pollefeys, and Torsten Sattler, «Why having 10,000 parameters in your camera model is better than twelve», in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 2535–2544.

[7] Gerd Marmitt, «Interactive volume ray tracing», 2008.

[8] Nelson Max, «Optical models for direct volume rendering», *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99–108, 1995.

[9] Kurt Hornik, Maxwell Stinchcombe, and Halbert White, «Multilayer feedforward networks are universal approximators», *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[10] Matthew Tancik, Pratul Srinivasan, Ben Mildenhall, *et al.*, «Fourier features let networks learn high frequency functions in low dimensional domains», *Advances in Neural Information Processing Systems*, vol. 33, pp. 7537–7547, 2020.

[11] Kai Zhang, Gernot Riegler, Noah Snavely, and Vladlen Koltun, «Nerf++: Analyzing and improving neural radiance fields», *arXiv preprint arXiv:2010.07492*, 2020.

[12] Marc Levoy and Pat Hanrahan, «Light field rendering», in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 31–42.

[13] Steven J Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F Cohen, «The lumigraph», in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 43–54.

[14] Nasim Rahaman, Aristide Baratin, Devansh Arpit, *et al.*, «On the spectral bias of neural networks», in *International Conference on Machine Learning*, PMLR, 2019, pp. 5301–5310.

[15] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger, «Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps», in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 14 335–14 345.

[16] Kangle Deng, Andrew Liu, Jun-Yan Zhu, and Deva Ramanan, «Depth-supervised nerf: Fewer views and faster training for free», *arXiv preprint arXiv:2107.02791*, 2021.

[17] Matthew Tancik, Ben Mildenhall, Terrance Wang, *et al.*, «Learned initializations for optimizing coordinate-based neural representations», in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 2846–2855.

[18] Vincent Sitzmann, Eric Chan, Richard Tucker, Noah Snavely, and Gordon Wetzstein, «Metasdf: Meta-learning signed distance functions», *Advances in Neural Information Processing Systems*, vol. 33, pp. 10 136–10 147, 2020.

[19] Chelsea Finn, Pieter Abbeel, and Sergey Levine, «Model-agnostic meta-learning for fast adaptation of deep networks», in *International conference on machine learning*, PMLR, 2017, pp. 1126–1135.

[20] Alex Nichol, Joshua Achiam, and John Schulman, «On first-order meta-learning algorithms», *arXiv preprint arXiv:1803.02999*, 2018.

[21] Diederik P Kingma and Jimmy Ba, «Adam: A method for stochastic optimization», *arXiv preprint arXiv:1412.6980*, 2014.

[22] Maximilian Denninger, Martin Sundermeyer, Dominik Winkelbauer, *et al.*, «Blenderproc», *arXiv preprint arXiv:1911.01911*, 2019.

[23] Jim Nilsson and Tomas Akenine-Möller, «Understanding ssim», *arXiv preprint arXiv:2006.13846*, 2020.

[24] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang, «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.

[25] Thomas Neff, Pascal Stadlbauer, Mathias Parger, *et al.*, «Donerf: Towards real-time rendering of compact neural radiance fields using depth oracle networks», in *Computer Graphics Forum*, Wiley Online Library, vol. 40, 2021, pp. 45–59.

[26] David B Lindell, Julien NP Martel, and Gordon Wetzstein, «Autoint: Automatic integration for fast neural volume rendering», in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 14 556–14 565.