Rheinische
Friedrich-Wilhelms-
Universität Bonn

Institute for Computer Science
Department VI
Autonomous Intelligent Systems

Rheinische
Friedrich-Wilhelms-Universität Bonn

Bachelor Thesis

Novel View Synthesis from Single and Multiple Cameras

Author:
Andre Rochow

First Examiner:
Prof. Dr. Sven Behnke

Second Examiner:
Dr. Michael Weinmann

Date: April 30, 2021

Declaration

I hereby declare that I am the sole author of this thesis and that none other than the specified sources and aids have been used. Passages and figures quoted from other works have been marked with appropriate mention of the source.

Bonn, 30.04.2021 [illegible]

Place, Date Signature

Abstract

Any efficient and realistic Avatar system requires the ability that the operator can look through the Avatar's eyes. For a realistic remote experience any movement of the operator's head should result in an identical movement of the Avatar's head. However, latencies and limitations in degrees of freedom are problematic. To tackle this issue we have developed view synthesis methods with decreasing reliance on depth: From a first approach that estimates depth monocularly to a synthesis method with no depth estimation at all. Our first approach processes single input images without given depth. We introduce a pipeline that (i) estimates depth and (ii) renders the image at a novel view point, which is the input for a CNN that (iii) produces the output image. We use a fully differentiable point cloud renderer to backpropagate gradients to the estimated depth map. Compared to state-of-the-art, we achieve better results while being more than twice as fast. Our second approach, called FaDIV-Syn addresses the problem that view synthesis methods are often limited by their depth estimation stage, where incorrect depth predictions can lead to large projection errors. This often leads to a state where the quality of depth estimation is the bottleneck in performance of the whole pipeline. Hence we present a multi-source view synthesis network that is totally independent to depth features and can further process high resolution images in real-time. Multiple source frames are warped into the target frame for a range of assumed depth planes. The resulting tensor representation is fed into a U-Net-like CNN with gated convolutions, which directly produces the novel output view. We therefore side-step explicit depth estimation, which improves efficiency and performance on transparent, reflective, and feature-less scene parts. FaDIV-Syn can handle both interpolation and extrapolation tasks and outperforms state-of-the-art extrapolation methods on the large-scale RealEstate10k dataset. In contrast to comparable methods, it is capable of real-time operation due to its lightweight architecture. We further demonstrate data efficiency of FaDIV-Syn by training from fewer examples as well as its generalization to higher resolutions and arbitrary depth ranges under severe depth discretization.

Contents

1 Introduction

Novel view synthesis aims to estimate images from novel viewpoints of a scene captured from one or more reference views. It has applications in virtual reality, 3D movies, or any other field where free choice of viewpoint is desired. The task is challenging as scene geometry and surface properties are not given, but have to be inferred from the available views. Additionally, viewpoint changes induce both occlusions, where foreground objects occlude previously visible backgrounds, and disocclusions, which uncover previously invisible backgrounds. In the latter case, the disoccluded content has to be guessed from its context. The view synthesis task is related to Multi-View Stereo (MVS), which aims to recover scene geometry only, and image inpainting, which tries to fill missing pixels with realistic content.

This thesis is motivated by avatar scenarios, in which a human operator teleoperates a remote robot. For convincing immersion, the operator wears VR glasses to perceive the world from the robot's point of view. Often there is a large distance between the avatar and the operator. A given head movement can cause a number of latency sources. These could be for example: (i) network latency or (ii) joint speed limitations of the avatar's head. There may also be joint limits on the side of the robot. That means the robot does not have the possibility to execute the transmitted movement, because it would come to collisions or mechanical restrictions. If the above problems are not counteracted, this will result in strange-looking effects in the images seen, which will significantly limit the VR experience. Since this thesis is motivated by real world scenarios, we train and evaluate all proposed methods on the large RealEstate10k dataset [1].

Our first approach processes on single images and builds upon SynSin [2]. The inference pipeline consists of three different stages: (1) depth estimation, (2) projecting and rendering, and (3) refining. As we do not require ground truth depth, we can use the RealEstate10k dataset for both training and evaluation. Similar to Wiles et al. [2], we build a fully differentiable pipeline. Like recent approaches [3, 4, 5, 6] we train our depth predictor self-supervised. Often a spatial transformer network [7] is used to reproject a second view, which provides an implicit quality measure of the estimated depth. The CNN is therefore trained to predict depth that minimizes a photometric reprojection error.

The resulting single view approach is related to monocular depth estimation approaches, because the renderer assumes depth which is then used to render the scene from a different perspective. This depth image must be predicted by a CNN from a single image without other information such as sparse pointclouds. Our main contribution is to show how monocular depth estimation techniques can improve the depth prediction stage, and further the optimization of end-to-end losses for the disparity network. In comparison, SynSin [2] only uses end-to-end losses, which are backpropagated through the entire pipeline. We achieve state-of-the-art performance with an pipeline that is more than twice as fast and less complex than SynSin.

In our second approach (FaDIV-Syn), we focus on the problem setting of real-time view interpolation and extrapolation from two RGB images, which show the scene from roughly the same direction—as, for example, when captured from a stereo camera. We note, though, that our method is applicable to more input views.

Many stereo [8, 9, 10, 11] and view synthesis methods [1, 12, 13] use Plane Sweep Volumes (PSV) [14], which warp the input views on a range of planes defined in the target camera and thus pre-transform the input data under the assumption of a range of discrete depths. Usually, a disparity or depth map is estimated from the PSV, which is then used to project the input views into the target frame correctly [13] or to generate representations such as multi-plane images [1, 12]. However, this approach creates a bottleneck: Imprecise or wrong depth estimates, which occur especially on uniform, transparent, or reflective surfaces, will result in loss of information and lead to failures later on in the synthesis pipeline.

Our proposed method is related to Image-Based Rendering approaches, but forgoes the geometry estimation step by operating on the PSV directly to compute the output RGB image, without computing explicit depth. To this end, we learn an RGB generator network which processes the PSV to directly synthesize the novel view and equip it with operations such as group- and gated convolutions [15, 16], which are suitable for detecting layer-wise correspondences, masking of irrelevant areas, and blending. Unlike most Image-Based Rendering (IBR) approaches, FaDIV-Syn has no explicit blending or inpainting stage, which allows distribution of the blending and inpainting operations throughout the learned network at appropriate abstraction levels. Only a single forward pass is required for this second approach. In summary, our contributions include (1) a real-time view synthesis network operating on plane sweep volumes, and (2) a detailed evaluation on the large-scale RealEstate10k dataset [1], where we outperform comparable methods in accuracy and runtime.

2 Fundamentals

This section describes the methods, techniques and losses used in this thesis.

2.1 Convolutions

2.1.1 Vanilla Convolutions

A neural network that processes images can fulfill different tasks such as classification, segmentation, or novel view synthesis. In any of these tasks a neural network needs some kind of feature extraction, which the network either produces in a self-supervised manner or traditionally includes hand-craftet features. Convolutional neural networks have become very popular in machine learning based image processing algorithms. Preprocessing an input image with convolutions allows a neural network to extract features that project the input image into a different (and often higher dimensional) feature-space. This representation can then be used to fulfill the certain task more efficiently.

In deep learning the network's feature extraction is seperated into multiple convolutional layers $L_1, ..., L_n$, where L_1 processes the raw input image and $L_i, i \in (2, ..., n)$ gets the output of L_{i-1} as input. A convolution basicly consists of a weight tensor W_f with a kernelsize $k = (k_w, k_h)$. Further, a common convolutional filter has the depth of the number of input feature maps (or channels) C and calculates a pixel x, y in the lth output feature map as:

$$W_f^l \circledast I_{x,y} := \sum_{i=-k'_w}^{k'_w} \sum_{j=-k'_h}^{k'_h} W^l_{k'_w+i,k'_h+j} \cdot I_{x+i,y+j}, \tag{2.1}$$

where $\circledast$ is the convolution operator, $k' = (k'_w, k'_h) = (\frac{k_w-1}{2}, \frac{k_h-1}{2})$, $W_f^l \in \mathbb{R}^{k_w \times k_h \times C}$ denotes the weight tensor of the convolutional filter for the lth output feature map and I are the input feature maps.

2.1.2 Gated Convolutions

Since the weights of a vanilla convolutional layer are fixed, once a network is trained, the layer can not adapt its behaviour regarding to global or local context.

Hence the network learns to work around this limitation by filtering out abstract features in multiple convolutional layers. Often activation functions such as Sigmoid or ReLU are used for bringing in non-linear behaviour.

Yu et al. [16] use gated convolutions as proposed by Dauphin et al. [15], in order to allow a convolutional layer to adapt itself to local (or global) features. Adaptive behaviours have also been observed at human neurons [17]. Gated convolutions have shown promising results in image inpainting [16]. Instead of estimating $W_f \circledast I$ (see Section 2.1.1), a gating layer calculates the non-linear output

$$GC(W_f, W_g, I) = \sigma_1(W_f \circledast I) \odot \sigma_2(W_g \circledast I), \tag{2.2}$$

where $\odot$ denotes the elementwise multiplication, $\circledast$ is the convolution operator, W_g is the weight tensor of the gating feature extraction and σ is an activation function. Note that σ_2 is the Sigmoid function in [16], but we also allow different activation functions such as ReLU in our approaches. The formula shows that the gating feature extraction can explicitly modify the output features.

2.2 Geometries and Rendering

All proposed methods in this section assume a pinhole camera model [18] with a given camera matrix K and camera pose ${}^{C}T_{W}$ for each frame, where W is the world coordinate system.

2.2.1 Homography Warping

In the following we assume that all pixels of an image I are on the same plane P. If we consider two cameras C_a and C_b looking at the points of a plane $p_i \in P$, then we have two representations P^a, P^b of the same plane P. With a known transformation ${}^{a}T_{b}$ and plane distance d, we can define the inverse homography matrix [19] ${}^{a}H_{b}$ that projects each pixel p_i^b to the corresponding pixel p_i^a in camera C_a. We do so, by calculating

$$p_i^a = \frac{z_i^b}{z_i^a} K_a \cdot {}^{a}H_{b} \cdot K_b^{-1} \cdot p_i^b, \tag{2.3}$$

where K_a, K_b are the corresponding camera matrices and z_i^a, z_i^b denote the z-coordinates in cameras C_a, C_b. The inverse homography matrix is defined as:

$${}^{a}H_{b} := R - \frac{tn^T}{d}, \tag{2.4}$$

where n is the normal vector of the plane, d is the plane distance and R, t is the rotation, translation of ${}^{a}T_{b}$, which is the transformation matrix from camera a to b. Following this, we can compute a coordinate grid to sample the planar Image I_b at the viewpoint of I_a. Note that the computation can be vectorized and processed over multi-dimensional tensors simultaneously on a GPU.

2.2.2 Spatial Projection

We consider two cameras C_a, C_b, which obtain images I_a, I_b and a depth map Z_a that corresponds to the pixels of I_a. For convenience we define a function π that transforms from homogenous to Cartesian coordinates. In the following we describe the steps to reproject the pixels $p_i^b \in I_b$ into camera C_a:

1. Backproject each coordinate grid pixel $g_i^a = (x_i^a \quad y_i^a \quad 1)^T$ with given depth z_i^a into a spatial point s_i^a, where
$$s_i^a = K^{-1} g_i^a z_i^a. \tag{2.5}$$
2. Calculate
$$\tilde{g}_i^{a \mapsto b} = (x' \quad y' \quad z)^T = K\pi({}^{a}T_{b}\,(s_i^{a\ T} \quad 1)^T) \tag{2.6}$$
to transform and project this point into the new camera C_b, where ${}^{a}T_{b} = {}^{b}T_{w}{}^{-1}\,{}^{a}T_{w}$ is the homogenous transformation matrix from camera C_a to C_b and K is the camera matrix with shape 3×3. The projected grid location is then given by
$$g_i^{a \mapsto b} = (x_i^b \quad y_i^b)^T = (\frac{x'}{z} \quad \frac{y'}{z})^T = \pi(\tilde{g}_i^{a \mapsto b}). \tag{2.7}$$
3. Reprojection: With the resulting coordinate grid $G_{a \mapsto b} = \{g_i^{a \mapsto b}\}$ we can sample the corresponding pixel of I_b for each pixel in I_a.

Steps 1 to 3 can be executed completely vectorized. However, if we want to project from image I_a to camera C_b, we need to invert the sampling operator. In this case, due to discretization and occlusion effects, more than one pixel may fall on the same grid location. A possible solution is z-buffering, which only accepts the pixel with the smallest z-coordinate in camera C_b. Z-buffering requires to iterate through the coordinate grid $G_{a \mapsto b}$ once.

2.2.3 Spatial Transformer Networks

Spatial Transformer Networks, introduced by Jaderberg et al. [7], were designed to spatially manipulate data in images. This manipulation is done in a completely differentiable way. A Spatial Transformer Network consists of a localization network, that estimates the parameters θ for the transformation, a grid generator, which creates the coordinate grid G, and a sampler, which performs the transformation with respect to the estimated coordinate grid G. Throughout this thesis we use Spatial Transformer Networks for two purposes:

1. To spatially project images into different camera views based on given transformations and depths. Here our grid generator follows steps 1-2 in Section 2.2.2.

2. To warp planes using the inverse homography. Applying the inverse homography (see Section 2.2.1) immediately gives us the coordinate grid to sample a textured plane P_b in a different camera C_a.

Since all transformations are given, we do not require a localization network.

2.2.4 Differentiable Pointcloud Renderer

We use the pointcloud renderer PR, which was introduced by Wiles et. al. [2] and implemented as open source in PyTorch3D [20]. Their neural pointcloud renderer allows to implement techniques such as z-buffering and splatting in a completely differentiable way.

With splatting we can spread the information of a projected pixel in a predefined neighbourhood with radius r, which leads to denser projected images. Each target grid pixel keeps a z-buffer with a fixed number of maximum stored values K, where $K = 1$ represents a hard z-buffer. For values $K \geq 2$ each target grid pixel is influenced by the K nearest points. The degree of influence of a surrounding projected pixel is proportional to the eucledean distance $d_2(\cdot\ , \cdot)$ to the grid location. According to these distances, each z-buffer is sorted and accumulated using a blending hyperparameter $\gamma \in [0, 1]$. For further implementation details we refer to Wiles et al. [2].

2.3 Losses

2.3.1 L1 and Mean Squared Error

The L_1 loss and the Mean Squared Error (MSE) are the two most common losses in pixelwise image comparison. For two images I_1, I_2 with size $N \times M$ we define:

$$L_1(I_1, I_2) := \sum_{n=0}^{N} \sum_{m=0}^{M} \mid I_1(n,m) - I_2(n,m) \mid, \tag{2.8}$$

$$MSE(I_1, I_2) := \frac{1}{NM} \sum_{n=0}^{N} \sum_{m=0}^{M} (I_1(n,m) - I_2(n,m))^2. \tag{2.9}$$

2.3.2 Perceptual Loss

The Perceptual loss, as used by Shih et al. [21], aims on estimating an index for the perceptual similarity of two images. It is motivated to give a quality measure closer to human perception. We therefore use a VGG-19 Network [22] Ψ, pretrained on ImageNet [23], to hierarchically compare the activations in different layers of both the generated and the ground truth image. Due to this learned convolutional preprocessing we no longer have a direct pixel-to-pixel comparison. Furthermore, the pixel-wise influence decreases more the deeper we are in the layers. We define the Perceptual loss as:

$$\mathcal{L}_{perceptual}(I_1, I_2) := \sum_{l}^{L} \frac{w_l}{N_{\Psi_l}} |\Psi_l(I_1) - \Psi_l(I_2)|, \tag{2.10}$$

where $\Psi_l(\bullet)$ is the activation of the lth layer, w_l is the weighting of the lth layer and N_{Ψ_l} are the number of elements in the lth layer.

If we spatially project images in novel view synthesis, slight errors must be expected due to noisy or incorrect depth. Hence a pixelwise loss can lead to blurry effects (especially in edge regions) if used for training. Adding the Perceptual loss to the training can counteract this problem and thus lead to higher quality images.

2.3.3 Image Quality Metrics

We use different metrics to evaluate the performance of our proposed methods.

1. **Peak Signal-To-Noise Ratio (PSNR)**
   The peak signal-to-noise ratio is commonly used to estimate the quality of

generated novel views. It is defined as:

$$PSNR := 10 \cdot log_{10}(\frac{MAX_I}{\sqrt{MSE}}) = 20 \cdot log_{10}(MAX_I) - 10 \cdot log_{10}(MSE). \quad (2.11)$$

Since we process normalized images with values in range $[0, 1]$, we set $MAX_I = 1$ and simplify the equation to

$$PSNR := -10 \cdot log_{10}(MSE). \quad (2.12)$$

In general *PSNR* measures the ratio between the maximum possible value and the noise which affects the quality. In terms of generating novel views we therefore interpret lack of performance as noise in images.

2. **Structural Similarity Metric**
In order to estimate the image quality we also utilize the Structural Similarity Metric (SSIM) as proposed by Zhou Wang et al. [24]. Different to pixelwise losses as L_1, *PSNR*, or *MSE* it compares window cutouts with size $N \times N$ to estimate a quality factor $SSIM(I_1, I_2) \in [0, 1]$. In this bachelor thesis we have chosen a window size $N = 11$ to be comparable with others. The SSIM loss is formulated as:

$$SSIM(x, y) := \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2\sigma_y^2 + C_2)}, \quad (2.13)$$

where x, y are two corresponding image cutouts with size 11×11, μ is the mean value, σ is the standart deviation and σ_{xy} is the covariance. Note that C_1 and C_2 are small constant values for numerical stability. In evaluation we adapt our *SSIM* implementations to SynSin [2] or 3D-Photo [21] for a fair comparison.

3. **Learned Perceptual Image Patch Similarity (*LPIPS*)**
The LPIPS metric, introduced by Zhang et. al. [25], is similar to the Perceptual loss as described in Section 2.3.2 and is motivated to give a quality measure closer to human perception. The *LPIPS* metric is defined as:

$$LPIPS(I_1, I_2) := \sum_l \frac{1}{H_l W_l} \sum_{h,w} ||w_l \odot (y^1_{l,h,w} - y^2_{l,h,w})||_2^2, \quad (2.14)$$

where l is the layer, w_l is the layer weighting and y^l_{hw} denotes the activations at position (h, w) in layer l. The activations y are estimated by a pretrained CNN. Often a Alex-Net [26] or a VGG-16 network [22] is used. For a fair comparison, we adapt to the implementations of either SynSin [2] or 3D-Photo [21].

2.3.4 GAN Losses

A Generative Adversarial Network (GAN), first introduced by Goodfellow et al. [27], basically consists of a generator network G and a discriminator network D, that aims on detecting generated images or areas. The generator tries to outsmart the discriminator, while the discriminator tries to recognize the generated images of the generator. Hence both networks influence each other in training time. In order for the discriminator to learn recognizing generated images, it is simultaneously trained with real images. In this thesis we build upon the implementation of a multiscale discriminator from Siarohin et al. [28], which is similar to the implementation of Isola et al. [29].

There are different ways to generate GAN losses [30, 27, 31]. In the classical approach [27] D and G play a minimax max game with the value function $V(D, G)$:

$$min_G max_D \; V(D,G) = \mathbb{E}_{x \sim p_{data}}[log \; D(x)] + \mathbb{E}_{z \sim p_z}[log(1 - D(G(z)))], \tag{2.15}$$

where p_{data} is a pool of real images and p_z is a pool of input values for the generator. The higher the output $D(x)$, the more certain the discriminator is that x is real. While D tries to minimize $D(G(z))$, G tries to maximize $D(G(z))$, since generator G targets to create realistic output.

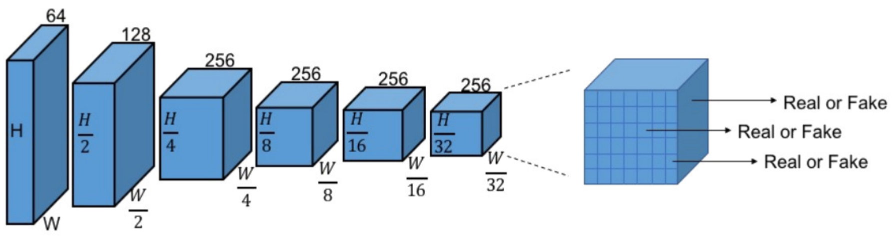


Figure 2.1: A typical GAN architecture extracted from [16].

In this thesis we have chosen to implement a Least-Square-GAN (LSGAN), which is more robust against vanishing gradient problems and further generates higher quality images than regular GANs [31]. For a given input z and a corresponding ground truth image I_z, the discriminator tries to assign $D(G(z)) = 0$ (fake) and $D(I_z) = 1$ (real). Our LSGAN loss is seperated into a discriminator and a generator part:

$$\mathcal{L}^D_{GAN}(z, I_z) = \sum_s ((D(I^s_z) - 1)^2 + (D(G(z)^s) - 0)^2), \tag{2.16}$$

$$\mathcal{L}^{G}_{GAN}(I) = \sum_{s} (D(G(z)^{s}) - 1)^{2}, \tag{2.17}$$

where s is the certain scale of the image. Note that $\mathcal{L}_{GAN}$ is averaged over the number of output features accumulated across all scales. For further implementation details we refer to Mao et al. [31].

2.4 Augmentation

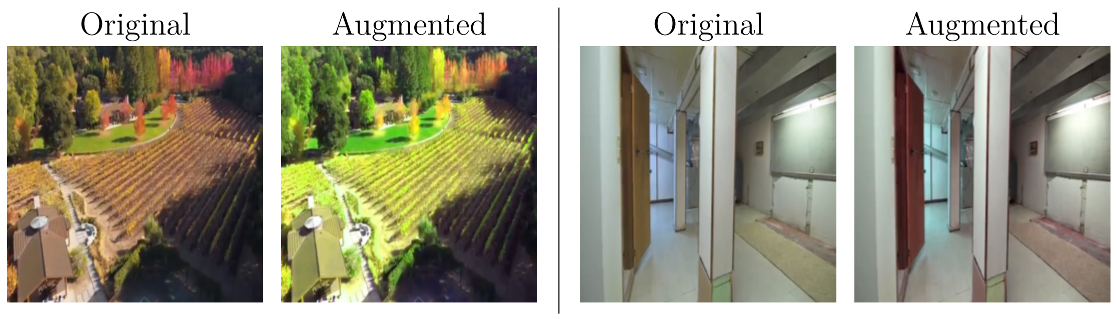


Figure 2.2: RealEstate10K [1] images augmented using color jitter (see Section 2.4).

When training a neural network overfitting problems often occur. This can be caused by a too small dataset or a network with too many parameters. To achieve a higher generalization on unseen data, we use image augmentation techniques, whereby given images in the training set are slightly modified to represent a new unseen image with similar properties. However, it is important that the augmentation does not change the data such a way that the network is no longer able to perform the intended task. For example, if we project an image spatially and use this projection as the basis for a refinement, we should not distort edges or rotate images.

Therefore, it is important that the augmentation methods are (1) transformation-invariant and (2) can always be applied to both the source images and the target image in the same way. In this work we limit ourselves to the color jitter augmentation (see Figure 2.2), which was introduced by He et al. [32] for the reimplementation of ResNet. Color jitter manipulates the following values:

1. brightness $\rightarrow$ [0.8,1.2]
2. contrast $\rightarrow$ [0.8,1.2]
3. saturation $\rightarrow$ [0.8,1.2]
4. hue $\rightarrow$ [-0.1,0.1]

The jittering values are uniformly chosen within our predefined interval[1]. They are applied equally to the entire image, as well as to corresponding input and ground truth images.

2.5 Batch Normalization

Batch normalization, introduced by Ioffe and Szegedy [33], has found wide application in deep learning approaches. A Batch Normalization layer (BN layer) tackles the problem that the input distribution of a layer changes during training, since the parameters of the previous layer change [33]. Neural networks often struggle learning this, and training can be much slower. If this problem is not addressed, a very small learning rate is often necessary in training. A BN layer normalizes the activations of a n-dimensional input x_i in a minibatch $\mathcal{B} = \{x_i, i \in [0, ..., b]\}$ of a layer as follows:

$$y_i = \gamma \hat{x}_i + \beta \tag{2.18}$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}, \tag{2.19}$$

where ϵ is a small constant for numerical stability, $\sigma_{\mathcal{B}}^2$ is the variance, $\mu_{\mathcal{B}}$ is the mean activation across the mini-batch and γ, β are parameters that need to be learned. γ and β allow to scale and shift the normalized activations $\hat{x}_i$. Note that after normalization the distribution of values of $\hat{x}$ have the expected value 0 and a variance of 1, if all elements $x_i \in x$ in the minibatch are sampled from the same distribution [33].

Besides a faster and more stable training, batch normalization has also shown regularizing effects. During inference, a BN layer uses the mean statistics $\tilde{\mu}_{\mathcal{B}} := \mathbb{E}(\mu_{\mathcal{B}})$ and $\tilde{\sigma}_{\mathcal{B}}^2 := \frac{m}{m-1}\mathbb{E}(\sigma_{\mathcal{B}}^2)$ accumulated during training with m mini batches.

[1]We use the PyTorch color jitter implementation: `https://pytorch.org/vision/stable/transforms.html#torchvision.transforms.ColorJitter`

3 Related Work

Novel view synthesis has been a long-standing problem in computer vision. There are already a number of approaches that differ mainly in the initial situation. For example, there are approaches that work on single or multiple images, consider depth as given or not given or require certain forms of annotation like segmentations. Most approaches at least assume given camera extrinsics and intrinsics. In general, a variety of approaches have already emerged, differing in the way they are implemented.

Image Based Rendering (IBR). In contrast to classical rendering of 3D scenes using textured geometry, IBR methods aim to render novel views by combining input images in the target pose [34, 35, 36, 37, 38, 13, 39, 40, 41, 42, 43]. To be able to project the input images correctly, IBR methods still require geometry, often in the form of depth maps, which are either available or estimated.

Recent approaches use blending to combine the images [34, 35, 36, 42]. Hedman et al. [35] learn the blending operation end-to-end. Going further, Riegler and Koltun [42] use a recurrent blending decoder in order to deal with a varying number of input images. In later work [43] heuristic input image selection is replaced with a fully-differentiable synthesis block. Penner and Zhang [34] introduce soft visibility volumes, which encode occlusion probabilities and thus avoid early decisions, but require a larger number of input views to compute. Kalantari, Wang, and Ramamoorthi [13] synthesize novel views in light field datasets. A lightfield is a camera matrix with $N \times M$ many cameras, which all have the same orientation and camera intrinsic. Thus, for example, an 9×9 lightfield consists of 81 different cameras. They use the corner cameras to predict depth in the target view, which is then used to warp the input views. Nguyen et al. [39] introduce RGBD-Net, which first estimates depth using a multi-scale PSV, warps the input images into the target frame, performs explicit blending, and refines the warped image using a depth-aware network.

Our FaDIV-Syn approach also works directly on the input images, but does not compute or require depth explicitly. Blending is learned implicitly by the network, together with detection and inpainting of extrapolated/disoccluded regions.

Geometry-Based Approaches. Recent approaches [44, 45, 2, 46] use depth features to spatially project pixel information and refine these projections to an output view. Wiles et al. [2] and Chen, Song, and Hilliges [47] process on single input images, estimating monocular depth in an end-to-end fashion. Wiles et al. [2] implement a differentiable point cloud renderer that allows z-buffering and splatting. Chen, Song, and Hilliges [47] predict depth in the target view using a transforming auto-encoder, that explicitly transforms latent code before entering the decoder. Similarly, Olszewski et al. [48] learn implicit voxel representations and transform encoded representations explicitly. Srinivasan et al. [46] predict RGB-D light fields from a single RGB image. They estimate precise scene geometry, render it to the target frame, and predict occluded rays using a second CNN.

Extreme View [45] predicts depth probabilities along camera rays in multiple input images and unites them in the target camera pose. They discretize the number of possible depth values and therefore reduce the depth estimation problem to a classification problem. A more recent approach [49] learns novel view synthesis without target view supervision by performing two synthesis steps, initially to an arbitrary target pose and from there to a pose where ground truth is available.

In contrast to these methods, FaDIV-Syn does not feature an explicit geometry representation. We argue that explicit geometry—besides requiring more effort to compute—forces early resolution of ambiguities, which can lead to loss of information.

Monocular Depth Estimation Almost all single view synthesis approaches require some kind of monocular depth estimation (if no given depth is used). Here it is problematic that monocular depth estimation is an ill-posed problem. However, using deep-learning methods, it has been possible in the past to estimate fairly good depths from monocular images [3, 4, 5, 6, 50].

Godard et al. [3] train a disparity network on estimating depth self supervised. They use a Spatial Transformer Network [7] in order to reproject the target image into the source images. Since their pipeline is completely differentiable, the disparity network can then be trained self-supervised using photometric losses only.

Nevertheless networks struggle to determine a correct scale. This problem can be countered by predicting scale invariant depth as Eigen, Puhrsch, and Fergus [51] propose. In combination with preprocessed keypoint depth, Tucker and Snavely [52] then estimate a scale factor for the whole predicted depth map.

Our single view approach estimates depth monocularly as well, but does not assume given keypoint depth to determine the correct scale.

GAN Based Approaches Recently, GAN-based approaches [27] have shown great progress in generating images [29, 53, 54, 2]. Yu et al. [16] use GAN losses in order to train a model, that can fill large inpainting areas. Image inpainting deals with the problem of filling in missing areas of the image. In most novel view synthesis approaches, disocclusions must be reconstructed in some form. Often the pixel neighbourhood is not sufficient to reconstruct the actual content. A GAN approach can help to ensure that such inpainting areas are filled with realistic content.

Multiplane Images (MPIs). A Multiplane Image (MPI) consists of multiple depth planes, which store RGB and alpha values. Once computed for a set of input images, novel views can be synthesized very efficiently by warping and blending the individual layers. This is memory and time efficient, since the MPI has to be calculated only once. Note that a MPI is only a representation for a static scene, and if the scene changes slightly, a new MPI must be estimated.

One can attempt to predict MPIs from single input images [55, 52]. Tucker and Snavely [52] train a network to estimate scale-invariant depth and require additional sparse point clouds to recover scale. Multiview approaches [12, 1, 56] use information from additional camera poses to place surfaces at the correct MPI layer. Plane sweeping [1] or warping [12] at different depths creates a suitable representation for the network. Mildenhall et al. [56] blend the layers of multiple MPIs to generate novel views with local light fields. Recently, Attal et al. [57] extended the key idea of multiplane images to multisphere images, to synthesize 360° images in real-time, although at lower resolution.

While inspired by MPI approaches, FaDIV-Syn bypasses MPI generation and instead determines a novel view from multiple warped planes directly and is applicable for dynamic scenes in real-time.

Layered Depth Images (LDIs) A Layered Depth Image (LDI) [58] represents an efficient data structure for rendering scenes from another viewpoint. Instead of storing a 2D array of depth pixels, a 2D array of Layered Depth Pixels is stored. Each Layered Depth Pixel stores a set of pixels including its color and depth information, where each pixel represents the location where a camera ray meets a surface in the scene. These surfaces can be occluded in the camera pose, where the LDI is stored in. Hence disocclusions, resulting from camera movement, can be filled by the occluded depth pixels stored in the data structure.

Recent approaches [59, 21, 60] build on Layered Depth Images [58]. Snavely, Tucker, and Tulsiani [59] train a CNN to predict a two-layered LDI, where the network learns to predict occluded pixels. Shih et al. [21] use an LDI representation to turn a single image into a 3D photo by inpainting color and depth of the occluded areas. The key idea is that they inpaint disoccluded depths and colour areas. This can be done by (1) disconnecting the LDI across edges or discontinuities (2) inpainting the background regions and (3) merging these generated contiguous 2D regions back into the LDI. The 3D LDI representation can then be used to generate numerous novel views in a time efficient way. However, this approach can only be used for static scenes, as the LDI estimation is quite slow. Unlike this, our FaDIV-Syn approach is motivated for non-static scenes and therefore allows a significantly faster calculation time.

Neural Rendering. Very recently, view synthesis approaches based on Neural Radiance Fields (NeRF) [61] have been introduced, which employ a neural network as a learnable density and radiance function over the scene volume. Novel views can be synthesized using classical volume rendering techniques. The sub-sequent improvements [62, 63, 38, 64] show impressive results on a variety of scenes. However, with the exception of Wang et al. [38], NeRFs have to be trained on the target scene, making them unsuitable for dynamic scenes, and are typically given more than two input images. While methods designed for dynamic scenes exist [64, 65, 66, 67], they require offline training or processing phases as well.

RealEstate10k Dataset. The RealEstate10k dataset [1] was made for novel view synthesis and is already utilized by many approaches [1, 52, 12, 2, 21, 45] for evaluation. It consists of more than 70k high-resolution indoor/outdoor scenes with different camera movements, extracted from real estate YouTube videos. Thus, on the one hand, there is a high degree of diversity and on the other hand, there are no longer any hard restrictions for camera movement, as in lightfield datasets. The videos have been automatically annotated with camera intrinsics and camera trajectories using ORB-SLAM2 [68] and bundle adjustment. Monocular SLAM cannot recover global scale, so the sequences have been scaled so that the near geometry lies at approximately 1.25 m [1].

4 Single View Synthesis with Self-Supervised Depth

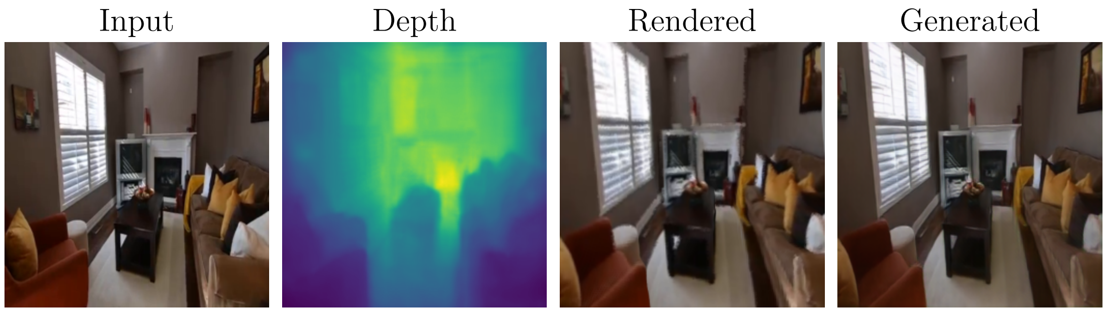


Figure 4.1: Example sequence of an image generated along the camera trajectory.

Our single view approach is similar to Wiles et al. [2], as it no longer requires ground truth depth. We therefore introduce a disparity network D, which estimates depth monocularly, and a refinement network R that generates the output view. The approach is motivated by the fact that dense and consistent depth is more valuable than patchy sensor depth, as present in Sun3D [69] or similar RGB-D datasets. Moreover, depth sensors are not always available.

A well-known problem of monocular depth estimation is that networks struggle with the correct scale estimation. Tucker and Snavely [52] therefore use sparse pointclouds, in order to approximate the correct scale. At the same time their network is trained to estimate scale invariant depth [51]. Our method, however, assumes only one RGB input image, and we note that sparse pointclouds can easily be embedded into our approach for more precise depth maps. Furthermore, the pipeline requires given camera instrinsics and camera poses. Given a source camera C_S with its camera pose ${}^{S}T_{W}$ and the obtained source Image I_S, we target to generate a novel view at the target pose ${}^{O}T_{W}$ of camera C_O.
Similar to SynSin [2] our forward pass is seperated into different stages:

1. Estimate pseudo disparity $d_{I_S} = D(I_S)$.

2. Convert pseudo disparity to depth: $z_I(d_I) = \frac{\tilde{z}}{d_I} - (\tilde{z} - min_z)$, where $\tilde{z}$ denotes a constant heuristic parameter, which is further explained in Section 4.1.

3. Project the pixels of I_S regarding to the depth d_{I_s} into the target camera frame C_O using the homogeneous transformation Matrix ${}^{S}T_{O} = {}^{W}T_{S}{}^{-1}\,{}^{W}T_{O}$. The projection $I_{S \to O} = PR(I_S, z_{I_S}, {}^{S}T_{O})$ is performed with a differentiable Pointcloud Renderer PR as described in Section 2.2.4.

4. Finally the fully convolutional refinement network R inpaints missing areas and further refines areas that are corrupted due to the projection and splatting effects (see Section 4.2).

4.1 Disparity Network

We argue that the disparity network is the bottleneck of the whole pipeline, since an incorrect depth prediction can lead to a bad initial state for the refinement network. Hence we focus on improving the depth estimation compared to Wiles et al. [2]. Inspired by Godard et al. [3], we use an encoder/decoder architecture, where the encoder is of a ResNet18 [70] pretrained on ImageNet [23]. Further, the decoder is designed to allow training the disparity network in multiple scales. This has shown great results in recent monocular depth estimation methods and prevents the disparity network from getting stuck in local minima [3]. We train the disparity network without direct depth supervision. Inspired by recent monocular depth estimation approaches [3, 4, 5, 6] we alternatively target to generate depth that minimizes a photometric reprojection error, which is described in Section 4.4.1.

For a given input image I_S we predict a disparity map $d_{I_S} = D(I_S)$, where each disparity pixel $d_{I_S}(x, y)$ corresponds to the RGB pixel $I_S(x, y)$ in the input image. Instead of estimating the depth z_I as:

$$z_I(d_I) = \frac{min_z}{d_I}, \text{ or} \tag{4.1}$$

$$z_I(d_I) = \frac{1}{10 d_I + 0.01} \tag{4.2}$$

as proposed by Wiles et al. [2], we found a more stable and softer formula

$$z_I(d_I) = \frac{\tilde{z}}{d_I} - (\tilde{z} - min_z), \tag{4.3}$$

where $\tilde{z}$ is a heuristic parameter that optimizes the distribution so that the network has a larger value range for close pixels. Figure 4.2 shows a comparison of the distributions. Note that the our distribution can take any values in range (min_z, inf), as d_I is normalized to $(0, 1)$ by a sigmoid activation. In training we add a very small constant to d_I for numerical stability. The heuristic parameter

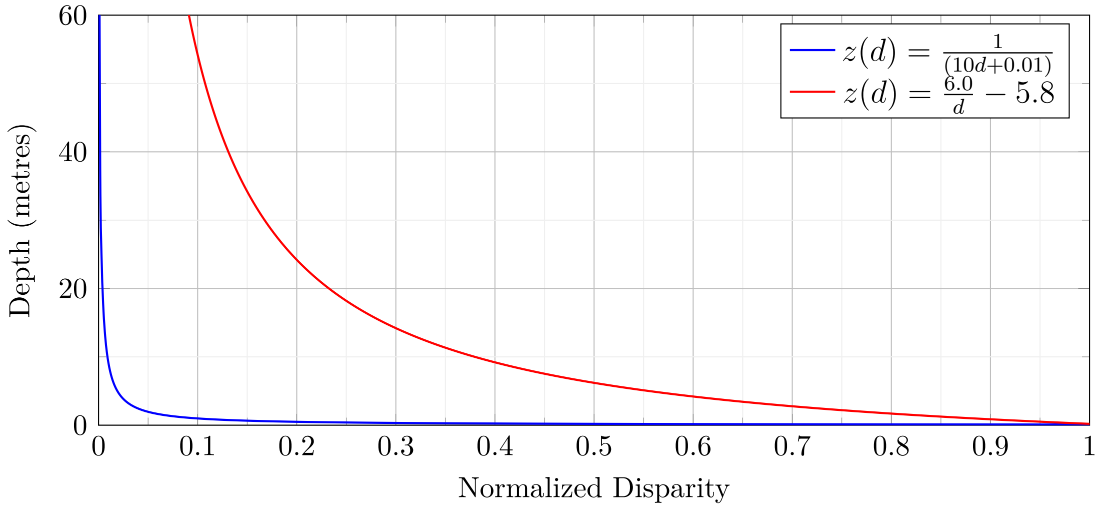


Figure 4.2: Comparison of our depth distribution, with $\tilde{z} = 6.0$ and $min_z = 0.2$ (red) and the depth distribution proposed in [2] (blue).

$\tilde{z}$ is set so that the network has half of the value range $d_I \in [0.5, 1.0)$ available to represent depth values z with

$$min_z < z \leq \tilde{z} + min_z$$

and the other half $d_I \in (0, 0.5)$ to represent depth values

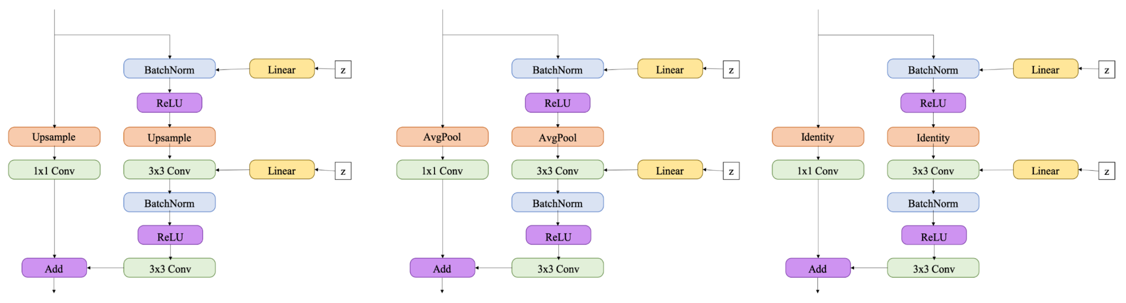
$$z > \tilde{z} + min_z.$$

The larger $\tilde{z}$ is chosen, the steeper the distribution becomes. An intuitive approach, would be to set $\tilde{z}$ to the average distance within the dataset. Thereby our disparity network has a wider value range to display the near geometry, where small errors in depth have larger impacts. However, since there is no ground truth depth in the RealEstate10k dataset, we heuristically set $\tilde{z} = 6.0$ (see Figure 4.2).

4.2 Refinement Network

The refinement network has the task of inpainting disocclusion areas and further to refine other areas that are corrupted due to incorrect depth or splatting. In order to investigate the effect of different architectures we evaluate two different refinement networks:

1. 9 ResNet blocks including two down/upsampling layers similar to SynSin [2]



(a) ResNet block. (b) ResNet block with an average pool block. (c) ResNet block with an identity block.

Figure 4.3: Modified from [2]. ResNet blocks that perform (a) bilinear upsampling (b) avarage pooling (downsampling) or (c) identity (no changes in resolution). Yellow boxes denote layers that inject noise to the batch normalization [2].

2. A U-Net architecture [71] with four down/upsampling layers, transposed convolutions and gated convolutions [15].

Note that the disparity network is trained end-to-end and thereby a U-Net architecture with a receptive field, that covers large areas is not suitable for training, since photometric pixel errors spread too much throughout the refinement network in backpropagation. Consequently the pipeline is always pretrained with the 9-block ResNet architecture, which has a significantly smaller receptive field.

ResNet Architecture. Our ResNet architecture is nearly the same as the refinement network of Wiles et al. [2]. It uses identity, downsampling and upsampling blocks as shown in Figure 4.3. Wiles et al. [2] use a refinement network with eight blocks, including four identity, two downsampling and two upsampling blocks. Except for another single identity block at the beginning, our network is the same. For further implementation details we refer to Wiles et al. [2].

U-Net Architecture. Our U-Net architecture, as shown in Figure 4.4, has a larger receptive field than the ResNet. We achieve this by using two more downsampling layers and a dilated convolution the middle. The downsampling is done by strided convolutions and the upsampling is performed using transposed convolutions. Each downsampling block performs a gated convolution (see Figure 4.4), which reduces the number of features by factor 2.

We input the rendered image in combination with a binary mask, that classifies the inpainting areas. This has recently shown good results in combination with gated convolutions for image inpainting [16], as the network can learn to adapt its behaviour in mask areas. As shown in Table 4.6 the U-Net is more than three times faster than our ResNet architecture for an image size of 256×256.

Input	k_1	c_1	k_2	c_2	Output
Rendered + Mask	3	32	3	64	*initial*
$\mathbf{G}(initial)$	3	64	3	128	$down_1$
$\mathbf{G}(down_1)$	3	128	3	256	$down_2$
$\mathbf{G}(down_2)$	3	256	3	512	$down_3$
$\mathbf{G}(down_3)$	3	512	3	1024	$down_4$
$\mathbf{G}(down_4)$	3	512	3	512	*dilated*
$\mathbf{Up}(dilated, G(down_3))$	3	512	3	256	up_1
$\mathbf{Up}(up_1, G(down_2))$	3	256	3	128	up_2
$\mathbf{Up}(up_2, G(down_1))$	3	128	3	64	up_3
$\mathbf{Up}(up_3, G(initial))$	3	64	3	32	up_4
up_4	1	3	–	–	*pred*

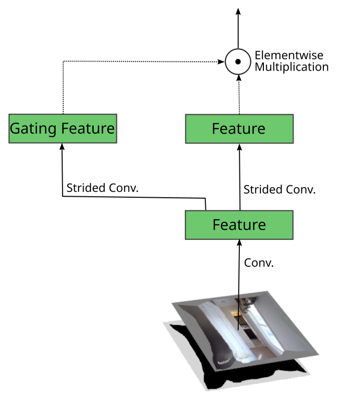


Figure 4.4: Left: Detailed network architecture of our U-Net refinement network. Right: Single downsampling layer of our U-Net including a gated convolution [15].

4.3 Pointcloud Renderer

We set the following parameters for our differentiable pointcloud renderer (see Section 2.2.4):

- $K = 64$, which is the z-buffer size,
- $r = 2.5$, which is the splatting radius, and
- $\gamma = 1.0$, which is the blending hyperparameter.

Figure 4.5 shows examples of images rendered with these parameters.

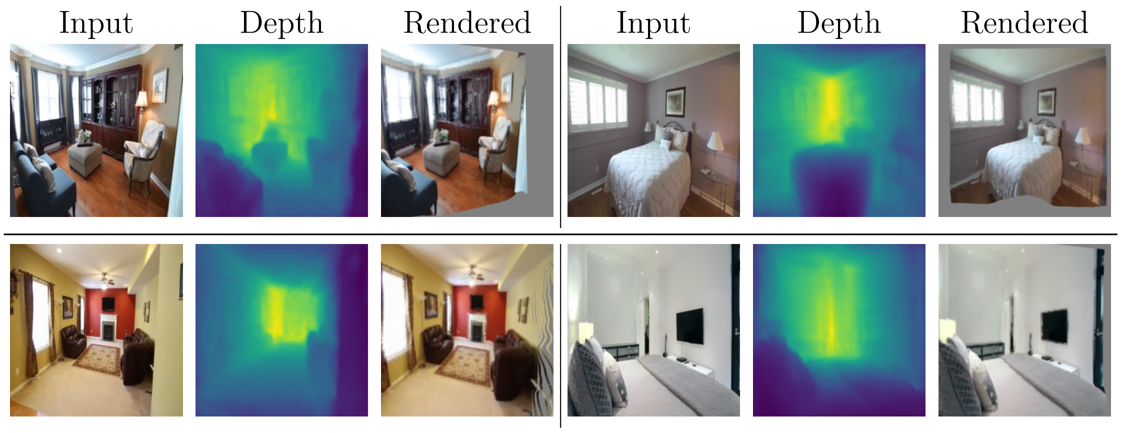


Figure 4.5: Example sequences of images rendered from the input image and estimated depth, along the camera trajectory. All images are extracted from the SynSin RealEstate test set of Wiles et al. [2].

4.4 Training

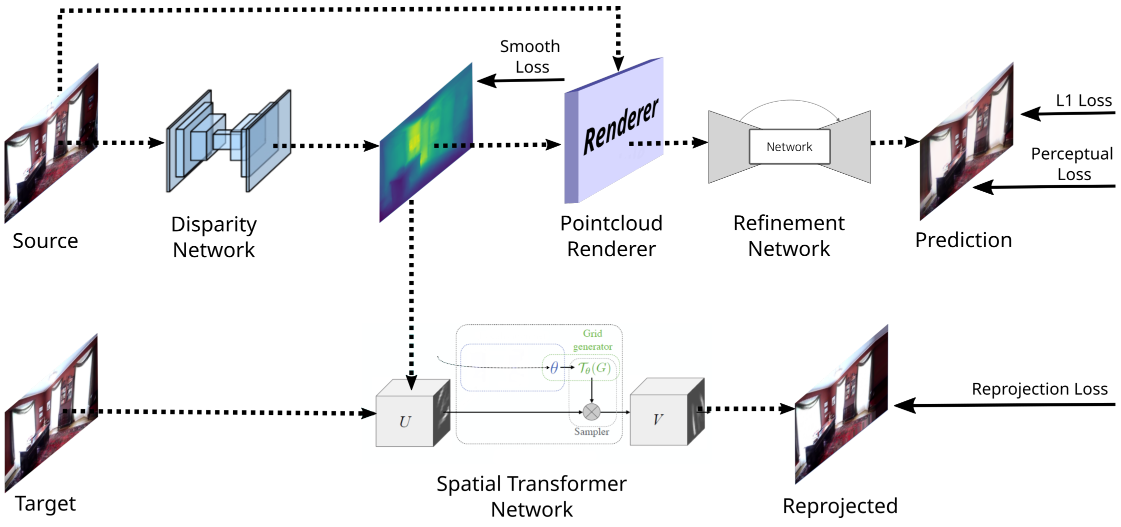


Figure 4.6: Our full training pipeline, including reprojection loss, L_1 loss, Perceptual loss and smoothness loss, as described in Section 4.4.1 (Spatial Transformer Network image modified from [7]).

Wiles et al. [2] have shown impressive results when training their whole pipeline with only end-to-end losses. Unlike SynSin we seperate our losses into three different kinds:

- **End-to-end losses** that train the whole pipeline,
- **Disparity losses** that only train the disparity network, and
- **GAN losses** that only affect the refinement network R (finetuning only).

The training pipeline is visualized in Figure 4.6.

4.4.1 Losses

1. End-to-end Losses

As shown in Figure 4.6 the whole pipeline is trained with L_1 and Perceptual loss (see Section 2.3.2). Since we use a differentiable pointcloud renderer, all gradients can be propagated between the refinement network R and the disparity network D. The L_1 loss ensures good reconstructions in homogeneous surfaces. Nevertheless, blurred edges are usually the result, since noisy depth causes errors in the projection. In these regions the spatial location of projected edges does not perfectly correspond with the ground truth, which is disadvantageous for a pixelwise loss. We therefore use a higher weighted Perceptual loss, which results in significantly better reconstructions in edge regions. Due to the convolutional preprocessing by a VGG-19 [22] network, the spatial environment is more important than pixelwise correspondence as in L_1 loss.

2. Disparity-only Losses

Wiles et al. [2] limit their training to end-to-end losses only. However, we believe that gradients in the disparity network can be distorted by the refinement network as follows:

1. Due to the receptive field of R edge regions are influenced by surrounding pixels as well.
2. Incorrect depth predictions can be refined by R, so that essantial gradients are eliminated.
3. Large inpainting areas cause weak reconstructions and might therefore influence surrounding areas, where the predicted depth was correct.

Masking the Losses. To eliminate the influence of large inpainting areas on the disparity network, we seperate the end-to-end losses into a D and R part. While R continues to be trained with the end-to-end losses as described above, we create a binary mask m to manipulate the losses for D. Given a rendered image I_R from the pointcloud renderer we define the mask m as:

$$m(I_R, x, y) := \begin{cases} 1.0 & \text{if } I_R(x, y) \text{ contains a pixel} \\ 0.0 & \text{otherwise} \end{cases} \tag{4.4}$$

The straight-forward way would be to directly mask the prediction I_O and ground-truth $\tilde{I}_O$ before calculating the losses. To include small inpainting regions we modify the mask m as follows. We predefine a maximum inpainting kernelsize $k = (5, 5)$ and define the eroded mask $\hat{m}$ as:

$$\hat{m}(m, x, y) := \begin{cases} 1.0 & \text{if } \mathcal{N}^{5\times 5}(x, y, m) = 1 \\ 0.0 & \text{otherwise} \end{cases} \text{, with} \tag{4.5}$$

$$\mathcal{N}^{5\times 5}(x, y, m) := \frac{1}{5\times 5} \sum_{i=-2}^{2} \sum_{j=-2}^{2} m(x+i, y+j), \tag{4.6}$$

where $\mathcal{N}^{5\times5}(x,y,m)$ is the averaged sum across a 5×5 neighbourhood of the center pixel (x,y). Note that this is equivalent with eroding the image with an 5×5 kernel for one iteration. To keep the thickness of inpainting areas that are larger than 5×5 pixel we dilate the mask $\hat{m}$ for one iteration. Hence our eroded and dilated mask $\tilde{m}$ can be defined as:

$$\tilde{m}(m,x,y) := \begin{cases} 1.0 & \text{if } \exists_{-2<i,j<2} : \mathcal{N}^{5\times5}(x+i,y+j,m) = 1 \\ 0.0 & \text{otherwise} \end{cases} \tag{4.7}$$

Figure 4.7 visualizes the difference between the raw mask m, the eroded mask $\hat{m}$ and the eroded & dilated (E/D) mask $\tilde{m}$. For acceleration we have implemented a GPU version of erosion and dilation [72], which are based on simple convolutions and thresholding.

We can then use $\tilde{m}$ to mask the losses:

$$\mathcal{L}_1^{\tilde{m}}(I_O,\tilde{I}_O) = \mathcal{L}_1((1-\tilde{m})I_O,(1-\tilde{m})\tilde{I}_O) \tag{4.8}$$

$$\mathcal{L}_{perceptual}^{\tilde{m}}(I_O,\tilde{I}_O) = \mathcal{L}_{perc}((1-\tilde{m})I_O,(1-\tilde{m})\tilde{I}_O). \tag{4.9}$$

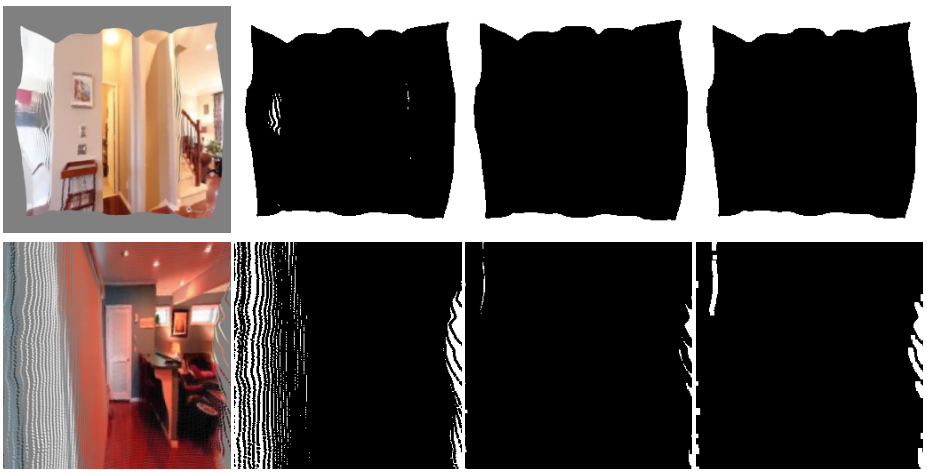


Figure 4.7: Steps to create the mask $\tilde{m}$, which we use for our proposed loss masking.

Reprojection Loss. In order to tackle the negative influence of a trained refinement network, as described above in Section 4.4.1, we follow Godard et. al. [3] and additionally generate raw gradients for D. As supervision is not possible, we formulate a photometric reprojection loss that implicitly gives a quality measure of the generated depth. We do so by using the predicted depth to reproject the target frame $\tilde{I}_O$ into the source frame's camera pose, as shown in Figure 4.6. The projection is performed using a Spatial Transformer Network as decribed in 2.2.3. However, since this loss is generated directly on the output of the disparity network, it can only be improved by a more accurate depth prediction (self-supervised learning). Following Godard et al. [3] we formulate the reprojection loss as

$$\mathcal{L}_{reproj}(I_S, I_{\tilde{O}\to S}) = \lambda_w SSIM(I_S, I_{\tilde{O}\to S}) + (1-\lambda_w)\mathcal{L}_1(I_S, I_{\tilde{O}\to S}), \tag{4.10}$$

where *SSIM* denotes the Structural Similarity loss (see Section 2.3.3), $\lambda_w \in [0,1]$ is the weighting ratio, which we set to 0.85, and $I_{\tilde{O}\to S}$ is the target image $\tilde{I}_O$ reprojected into the source frame I_S regarding to the predicted source depth z_S. Similar to Godard et al. [3] we mask the output so that gradients at pixels (x, y) where

$$\mathcal{L}_{reproj}(I_S, \tilde{I}_O)(x,y) \leq \mathcal{L}_{reproj}(I_S, I_{\tilde{O}\to S})(x,y) \tag{4.11}$$

are eliminated. This guarantees that only sufficiently large camera movements generate gradients. Furthermore, objects that move with the camera are masked out. This is important since the reprojection loss assumes a camera moving in a static scene and hence non-static objects can cause wrong gradients that lead to instabilities and artifacts.

The reprojection loss is estimated in three scales $s \in [1, 0.5, 0.25]$. Following Godard et al. [3] we first bilinearly upsample the disparities to $s = 1$ and therefore calculate all losses with the original input image resolution. The scale $s = 1$ is weighted with $w_1 = 0.5$ and scales $s \in [0.5, 0.25]$ are weighted with $w_{2,3} = 0.25$ in training.

Smoothness Loss. Often image areas which are homogenous in color also have low frequency disparity changes. To keep the disparity consistent with the edges in the image, we use a smoothness loss

$$\mathcal{L}_{smooth}(I,d) = \mid \partial_x \frac{d}{\bar{d}} \mid e^{-|\partial_x I|} + \mid \partial_y \frac{d}{\bar{d}} \mid e^{-|\partial_y I|}, \tag{4.12}$$

where d is the dispartity and $\frac{d}{\bar{d}}$ is the mean normalized disparity as proposed by Wang et al. [50]. Thereby gradients at $d(x,y)$ are strongly penalized if gradients at $I(x,y)$ are small and lightly penalized if the gradients at $I(x,y)$ are large. By normalizing the disparity we avoid a consequent shrinking of the estimated depth [50].

Consequently we train the refinement network with

$$\mathcal{L}_R(I_O, \tilde{I}_O) = \lambda_1 \mathcal{L}_1(I_O, \tilde{I}_O) + \lambda_p \mathcal{L}_{perceptual}(I_O, \tilde{I}_O), \tag{4.13}$$

and the disparity network with

$$\begin{aligned}\mathcal{L}_D(I_S, d_S, I_O, \tilde{I}_O, I_{\tilde{O}\to S}) = {} & \lambda_1 \mathcal{L}_1^{\tilde{m}}(I_O, \tilde{I}_O) + \lambda_p \mathcal{L}_{perceptual}^{\tilde{m}}(I_O, \tilde{I}_O) \\ & + \lambda_r \mathcal{L}_{reproj}(I_S, I_{\tilde{O}\to S}) + \lambda_s \mathcal{L}_{smooth}(I_S, d_S).\end{aligned} \tag{4.14}$$

4.4.2 Finetuning

In the finetuning phase, we fix all parameters of the disparity network. Therefore, we have two options to train the refinement network R:

1. Train R from scratch, or

2. Finetune R, so that it can optimize itself to the fixed weights of D.

When training R from scratch we can also utilize our U-Net architecture, which has a larger receptive field.

3. GAN Loss

Many view synthesis approaches use a discriminator in order to generate GAN losses (see Section 2.3.4). However, we believe that GAN losses can be harmful for the disparity network, since there is no direct supervision. Furthermore, one benefits from GAN losses mainly in large inpainting areas [16]. Therefore, we limit the use of a discriminator to finetuning only, where we fix the weights of the disparity network. Here, we use a least-square discriminator *DIS* [31], and train both the refinement network and the discriminator as described in Section 2.3.4. Hence during finetuning the refinement network is trained with

$$\mathcal{L}_R(I_O, \tilde{I}_O) = \lambda_1 \mathcal{L}_1(I_O, \tilde{I}_O) + \lambda_p \mathcal{L}_{perceptual}(I_O, \tilde{I}_O) + \lambda_g \mathcal{L}_{GAN}(I_O). \tag{4.15}$$

4.4.3 Data Loading

Both the training and the evaluation are carried out on the RealEstate10k dataset. In training we distinguish between two different data loading methods.

End-to-end Training. When training the whole pipeline, we first select a random scene and a random source image in this scene. Then, similar to Wiles et al. [2], we select images from a range of $[-30, 30]$ frames apart the source image. Only images that meet a translation-threshold $\lambda_{translation}$= 0.15 m and a rotation-threshold $\lambda_{rotation}$= 5 ° are candidates for the random election of the target image. This ensures that there is always enough distance between the source and target image, which increases the likelihood that suboptimal depth will lead to gradients, that improve disparity, in larger homogeneous areas. If no image meets the thresholds, we randomly choose one.

Finetuning. During finetuning the parameters of the disparity network are fixed. Hence we can also synthesize views at smaller distances and we can further increase the frame offset range to $[-40, 40]$, to have even more variance in the training set. In total, we select 70% as before but in the interval $[-40, 40]$ and the remaining 30% from a near environment (with a maximum of 8 frames apart) without rotation and translation thresholds.

4.5 Evaluation

In this section we evaluate our single view approach and compare the different variations. Furthermore, we compare selected models with Wiles et al. [2]. For each training setup the RealEstate10k dataset is subdivided into approximately 54K/13.5K/7K training/validation/test sequences. We stop the training when the network gets worse on the validation set or does not improve for a given time. Additionally, we set an upper limit of approximately 100k train iterations when training the whole pipeline. During finetuning we increase this limit to 200k iterations. All networks are trained with 2 NVIDIA TITAN RTX GPUs. The training takes approximately 4.5 days for 100k iterations.

We evaluate our models with the official test-set published by Wiles et al. [2]. To remain comparable with SynSin [2], we train with an image size of 256×256. We train the full pipeline with batchsize 36 and finetune with batchsize 60.

4.5.1 Full Model

Our full model is trained end-to-end, and thus without GAN losses (see Section 4.4.1). In our evaluation we always change a parameter and compare to this reference model. During initial testing we have determined the optimal weighting of the losses and parameters as:

- ResNet refinement network with 9 blocks and a final tangent hyperbolic (tanh) non-linearity,
- learning-rate = 0.0001,
- $\lambda_{perceptual} = 4.0$, which is the weighting of the Perceptual loss,
- $\lambda_1 = 1$, which is the weighting of the L_1 loss,
- $\lambda_{smooth} = 0.00005$, which is the weighting of the smoothness loss,
- $\lambda_{reproj} = 2$, which is the weighting of the reprojection loss,
- $\gamma = 1$, $K = 64$, $r = 2.5$, which are the parameters for the pointcloud renderer,
- $z(d) = \frac{\tilde{z}}{d} - (\tilde{z} - min_z)$, which is our depth distribution for the disparity output. We set $\tilde{z} = 6.0$ and $min_z = 0.2$.

Furthermore, we train the disparity network in multiple scales $s \in [1, 0.5, 0.25]$ and with loss masking as described in Section 4.4.1.

4.5.2 Results

Masking vs. No-Masking

To investigate the utility of our loss masking (see Section 4.4.1), the **no-masking** variation is trained like our full model but without masking. In Table 4.1 the full model is compared with the no-masking model. The full model that uses loss masking performs better than the no-masking model. We further made the observation, that our proposed loss masking leads to faster learning, as shown in Figure 4.8 and Table 4.1. One reason might be that the large inpainting areas have a negative influence on the gradients in the disparity network. This effect seems to be stronger in the beginning, where the refinement network lacks in inpainting performance.

In terms of L_1 loss a single wrongly predicted pixel affects pixels in the neighbourhood with the size of the receptive field of R. In Perceptual loss, this effect increases even more with the receptive field of the VGG-19 network. Conclusively it makes sense to eliminate the influence of large inpainting areas to the disparity network.

Table 4.1 clearly shows that the full network has a larger lead at the beginning, which becomes smaller towards the end.

Model	PSNR↑	SSIM↑	LPIPS↓	Model	PSNR↑	SSIM↑	LPIPS↓
full (25K)	**21.30**	**0.712**	**1.349**	full (100K)	**21.84**	**0.730**	**1.262**
no-masking (25K)	21.08	0.708	1.393	no-masking (100K)	21.73	0.726	1.278

Table 4.1: Comparison of our full model and a variation that is trained without our proposed loss masking. We show the results on the SynSin RealEstate test set after 25K and 100K training iterations.

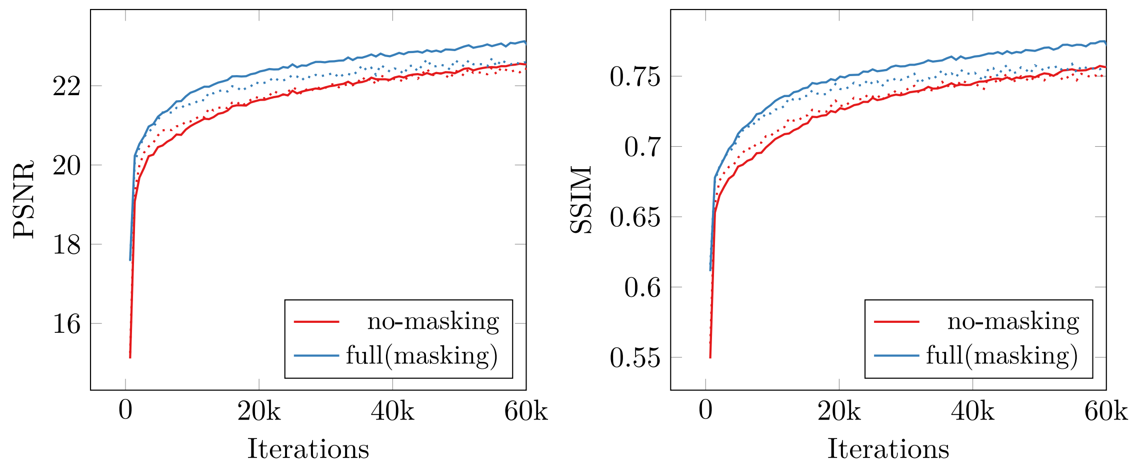


Figure 4.8: PSNR (left) and SSIM loss (right) of the full network that uses loss masking and the no-masking model. The full model shows better performance on the train (solid) and validation dataset (dotted).

Depth Distribution

In order to investigate the utility of our proposed depth distribution $z(d) = \frac{\tilde{z}}{d} - (\tilde{z} - min_z)$, we train two variations that are trained like the full model, but with different depth distribution:

- **naïve depth distribution** $\rightarrow z(d) = \frac{0.2}{d}$
- **SynSin depth distribution** [2] $\rightarrow z(d) = \frac{1}{10d + 0.01}$

The comparison in Table 4.2 shows that the naïve model performs worst. Further, the SynSin depth model achieves only slightly worse results than the full model. During the training, however, we noticed some drops in performance of the SynSin depth network, as shown in Figure 4.9. We believe that this could be due to the fact that the SynSin depth distribution uses the whole value range $d \in [0.1, 1.0]$ for depth values $z < 1.0\,\text{m}$ and therefore only a small value range (of approximately 10%) to display all $z \geq 1.0\,\text{m}$. This is a poorly chosen distribution,

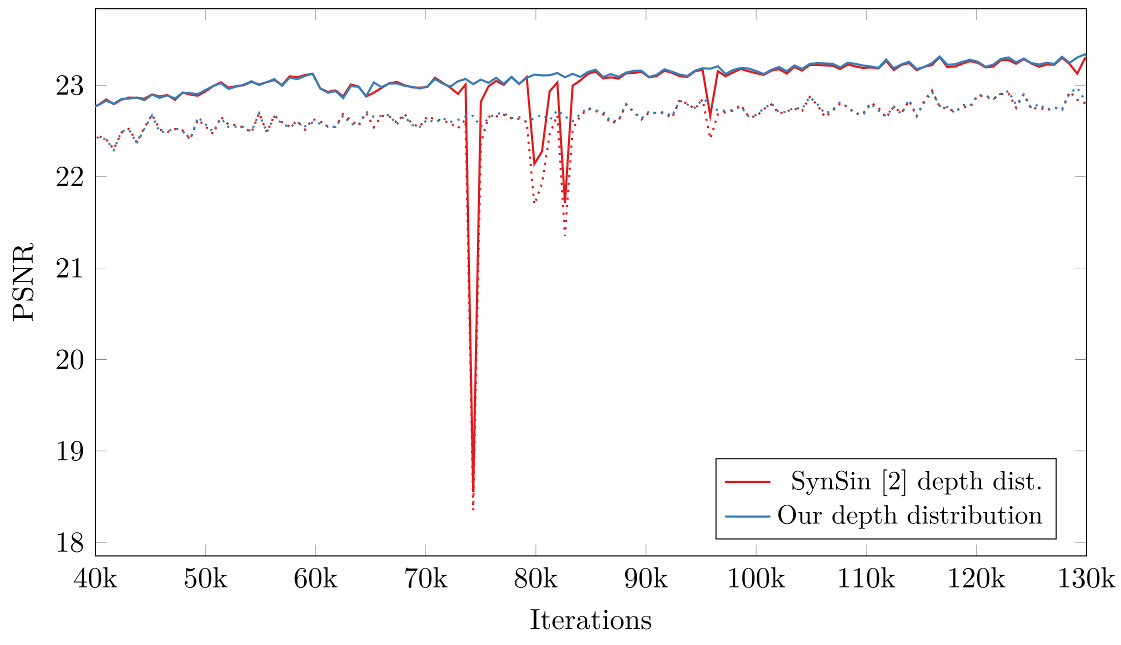


Figure 4.9: Comparison of the training (solid) and validation (dotted) progress of our depth distribution compared to the depth distribution proposed by Wiles et al. [2].

as objects rarely appear close to the camera in the RealEstate10k dataset. Further, in the annotation process the close geometry was set to 1.25 m to reconstruct the global scale. Consequently, when training with the SynSin depth distribution, small changes in the disparaty network have a larger impact on the output and thus also on the stabiliy of the training. Our proposed depth distribution, however, has approximately 88% of its value range to display depth values $z > 1.0\,\mathrm{m}$ and is therefore more stable throughout the training.

We conclude that we have found a more stable depth distribution, which is more suitable for the RealEstate10k dataset, compared to SynSin [2].

Model	PSNR↑	SSIM↑	LPIPS↓
full(our depth)	**21.84**	**0.730**	**1.262**
SynSin depth	21.82	**0.730**	1.263
naïve depth	21.80	0.728	1.269

Table 4.2: Comparison of different depth distributions on the SynSin RealEstate test set [2].

Smoothness Loss

To check whether the smoothness loss weighting is too small, we train a **high-smooth** variation with $\lambda_{smooth} = 0.001$, which is 20 times higher compared to our full model. As shown in Table 4.3, the results are quite similar, but the full model performs slightly better than the high-smooth network.

The comparison of the generated depth maps in Figure 4.10 shows that the high-smooth-model is slightly better at representing the edges in the image. We believe that the refinement network can adapt to both similarly, which is why there are only minor differences in the results. Note that our smoothness loss is based on the assumption that gradients in the color image are also gradients in the disparity image. However, this is not always correct, as for heavily textured surfaces. Therefore, the smoothness loss should not be weighted too heavily. We conclude that $\lambda_{smooth} = 0.00005$ is a well chosen weighting.

Model	PSNR↑	SSIM↑	LPIPS↓
full	**21.84**	**0.730**	**1.262**
high-smooth	21.83	**0.730**	1.263

Table 4.3: Comparison of our full model with our high-smooth model, on the SynSin RealEstate test set.

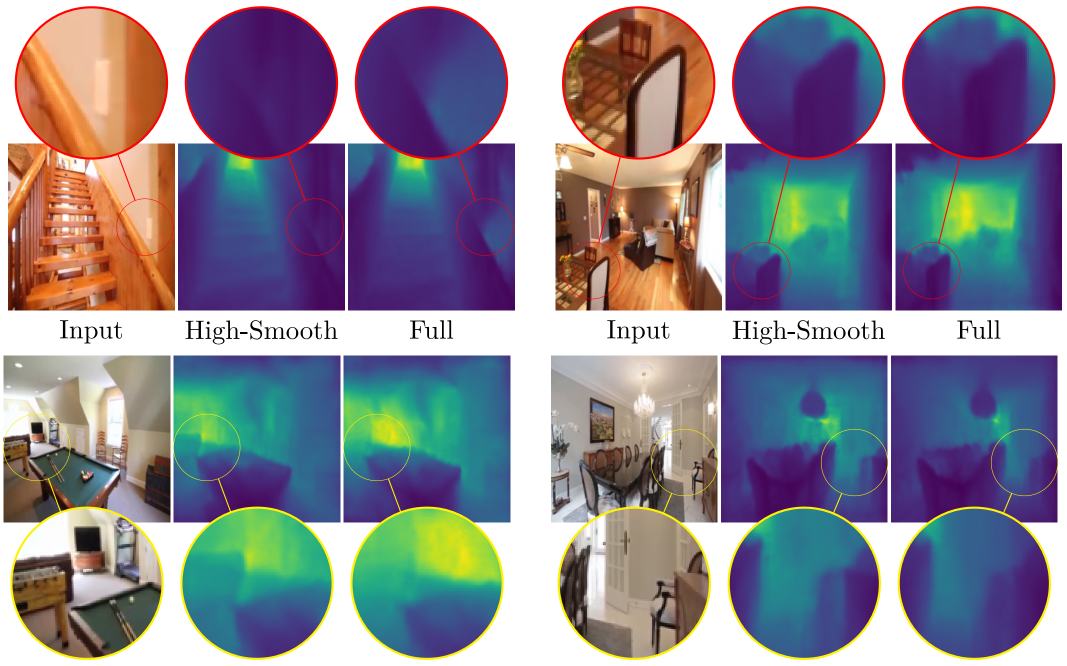


Figure 4.10: Predicted depth of our high-smooth model compared with the full model on the SynSin RealEstate test set.

4.5.3 Comparison with State of the Art

Full Pipeline (end-to-end) Results

Since we do not project extracted features, we evaluate against the SynSin(RGB) version [2], which has a similar complexity (two CNNs). Table 4.4 shows that our pipeline performs better than SynSin(RGB) in all three quality metrics. As our refinement network is similar to Wiles et al. [2], we mainly explain our better performance with more precise depth maps generated by our disparity network. Figure 4.14 and Figure 4.15 show depth maps from SynSin and depth maps generated by our pipeline.

Model	PNSR↑	SSIM↑	LPIPS↓
ours(full)	**21.84**	**0.73**	**1.26**
SynSin(RGB)	20.92	0.68	1.67

Table 4.4: Comparison of our full model with SynSin(RGB) on the SynSin RealEstate test set [2]. Both networks have a similar complexity.

Finetuning Results

Our U-Net refinement network has a larger receptive field than our ResNet refinement network and is not suitable for training the disparity network in an end-to-end fashion. Therefore, we use a pretrained disparity network and train a U-Net refinement network (see Figure 4.4) from scratch. Furthermore, the disparity network parameters are fixed. The U-Net refinement network is trained adversarially, as described in Section 2.3.4 and Section 4.4.2. We weight the GAN losses with $\lambda_g = 1$. The discriminator is trained with a learning rate of 0.001. To investigate the effect of GAN training we additionally train a UNet without GAN losses.

Finetuning the pipeline with our U-Net network gave great results as shown in Table 4.5. Both U-Nets perform better than the ResNet which was trained only in end-to-end mode (see full-e2e in Table 4.5). Since we downsample four times instead of just two times we enhance the receptive field without losing throughput in inference. Furthermore, our UNet-GAN performs a bit better than U-Net in all metrics. It is thus shown, that our supervised training setup can benefit from additional GAN losses.

Comparing against the full SynSin model [2], which projects features instead of RGB shows that we achieve better results in PSNR and same results for SSIM and LPIPS. Note that SynSin(full) uses an additional feature extractor network, that enhances the complexity and reduces the throughput of the whole pipeline. We believe that our pipeline could generate similar improvements, as from SynSin(RGB) to SynSin(full), by adding a feature extractor CNN as well.

In conclusion we achieve better results than SynSin(full), with a pipeline that is (i) less complex, (ii) uses only two instead of three CNNs and (iii) is faster (see Section 4.5.4).

Model	PSNR↑	SSIM↑	LPIPS↓
ours(full-e2e)	21.84	0.730	1.262
ours(U-Net)	22.53	0.740	1.183
ours(U-Net-GAN)	**22.57**	**0.742**	**1.182**
SynSin(full)	22.31	**0.74**	**1.18**
SynSin(RGB)	20.92	0.68	1.67

Table 4.5: Our evaluation results on the SynSin RealEstate test set compared with both SynSin variations. Wiles et al. [2] published their results with an accuracy of only two decimal places. Note that SynSin(full) projects features, which are extracted with an additional CNN.

4.5.4 Inference Time

Table 4.6 gives an overview for the inference times of the individual submodules within the different pipelines. The feature network is only used by SynSin(full) and therefore creates a higher complexity and additional processing time. Our disparity network is a bit faster than SynSin's. Nevertheless, as shown in Table 4.6, both are very fast. The renderer is slowest for SynSin(full) since 64 extracted feature-maps have to be projected. Unlike this our model projects RGB information and hence only three feature-maps. Furthermore, we achieve a clear advantage through our renderer parameters K=64 (z-buffer size) and r=2.5 (splatting radius), which we choose to be significantly smaller than SynSin (K=128, r=4). This can be seen in Table 4.6 at the comparison of our models to SynSin(RGB).

A significant limitation is that the renderer throughput drops enormously when enhancing the image size. With an image size of 768×768, our renderer already takes approximately 170ms for rendering, which is almost five times as long as for 512×512. Therefore we can only process image resolutions of 512×512 in real-time. Our ResNet refinement network is a bit slower compared to SynSin, since we use an additional block at the beginning (see Section 4.2). The U-Net network has more parameters than all ResNet refinement networks, but is more than twice as fast on a GPU.

Overall, we achieve even better results than SynSin(full) (see Table 4.5) with our finetuned U-Net pipeline that is almost three times as fast.

	Timings [ms] (Torch)									
	Feature		Disparity		Renderer		Refinement		$\sum$	
Model	256^2	512^2	256^2	512^2	256^2	512^2	256^2	512^2	256^2	512^2
Full(ours)	-	-	2.55	4.34	4.64	34.5	16.18	40.3	23.4	79.1
U-Net(ours)	-	-	2.55	4.34	4.64	34.5	5.21	17.3	**12.4**	**56.1**
SynSin(RGB)[2]	-	-	2.95	5.02	8.08	48.3	14.56	36.7	25.6	90.0
SynSin(full)[2]	10.9	23.1	2.95	5.02	17.5	87.9	14.64	37.3	46.0	153.3

Table 4.6: Inference times (in PyTorch) of our models (Full, U-Net) and the SynSin models (SynSin(RGB), SynSin(full) [2]). We measure timings of the different networks: Feature (only SynSin(full)), Disparity and Refinement on a single NVIDIA RTX 3090 GPU. Further, we measure the point-cloud renderer on a single NVIDIA TITAN V GPU. We show timings for the training resolution 256^2:=256×256 and the generalized resolution 512^2:=512×512.

4.5.5 Qualitative Results

Large Inpainting Areas. Extreme camera movements often cause large inpainting areas. Inpainting these areas is a very challenging task, as the pixel information have to be guessed by the network. As Figure 4.11 shows our U-Net architecture performs better than the ResNet refinement network in large inpainting areas. We believe that this is mainly due to the larger receptive field of the U-Net. The receptive field of the ResNet is often not sufficient to inpaint anything at all, as shown in Figure 4.11.

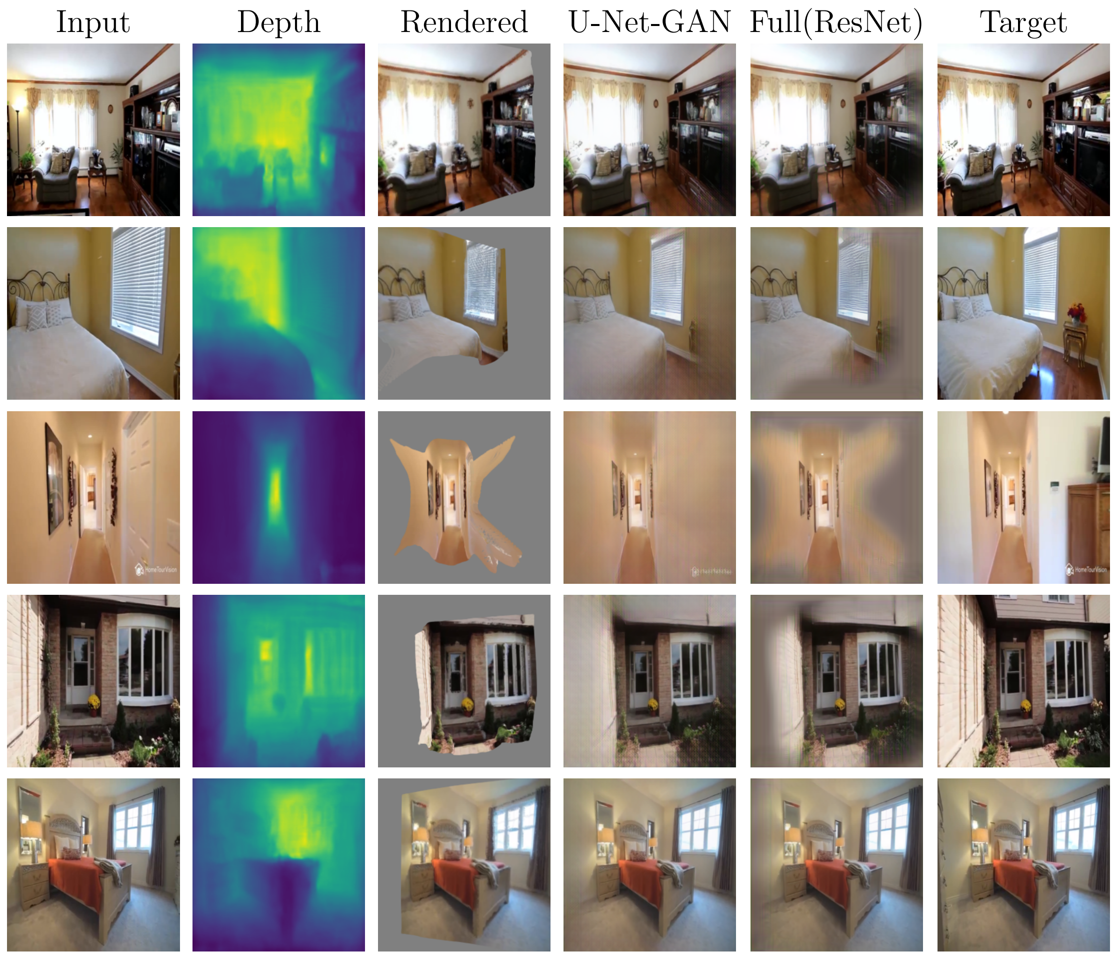


Figure 4.11: Example sequences for large inpainting areas along the trajectory. Our U-Net (finetuning) is compared with the full model that uses the ResNet refinement network. All sequences are extracted from the SynSin RealEstate test set [2].

Small Camera Movements. Our pipeline achieves excellent results for small camera offsets (see Figure 4.12). When the camera moves slightly, often no inpainting areas occur. However, the refinement network must still correct areas which are corrupted due to splatting. If one only intends to synthesize smaller distances, it is probably more useful to project the pixels directly, i.e. without splatting. Figure 4.12 visualizes example sequences with small camera movements.

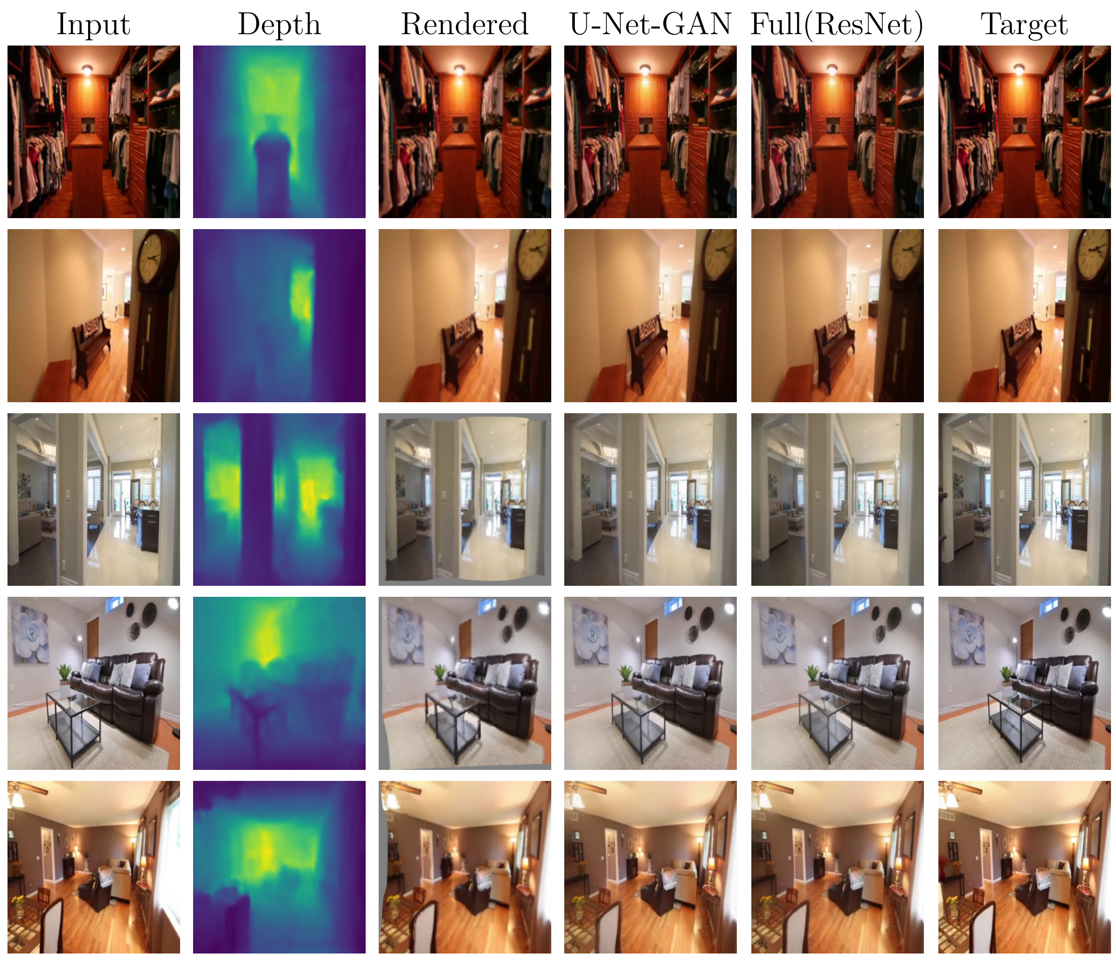


Figure 4.12: Example sequences for small camera offsets along the trajectory. Our U-Net (finetuning) is compared with the full model that uses the ResNet refinement network. All sequences are extracted from the SynSin RealEstate test set [2].

Medium-Large Camera Movements Towards the Scene. Inpainting areas especially occur when the camera moves backwards (see Figure 4.11). When the camera moves towards the scene, large inpainting areas rarely occur since the renderer uses splatting to cover regions with an insufficient point density (see Section 2.2.4). However, the network must learn to improve the quality in such areas. This is comparable to the super resolution task [73], where the resolution of images is to be improved. As Figure 4.13 shows, our pipeline generates impressive results on the SynSin RealEstate test set [2] for medium to large offsets, where the camera moves closer to the scene.

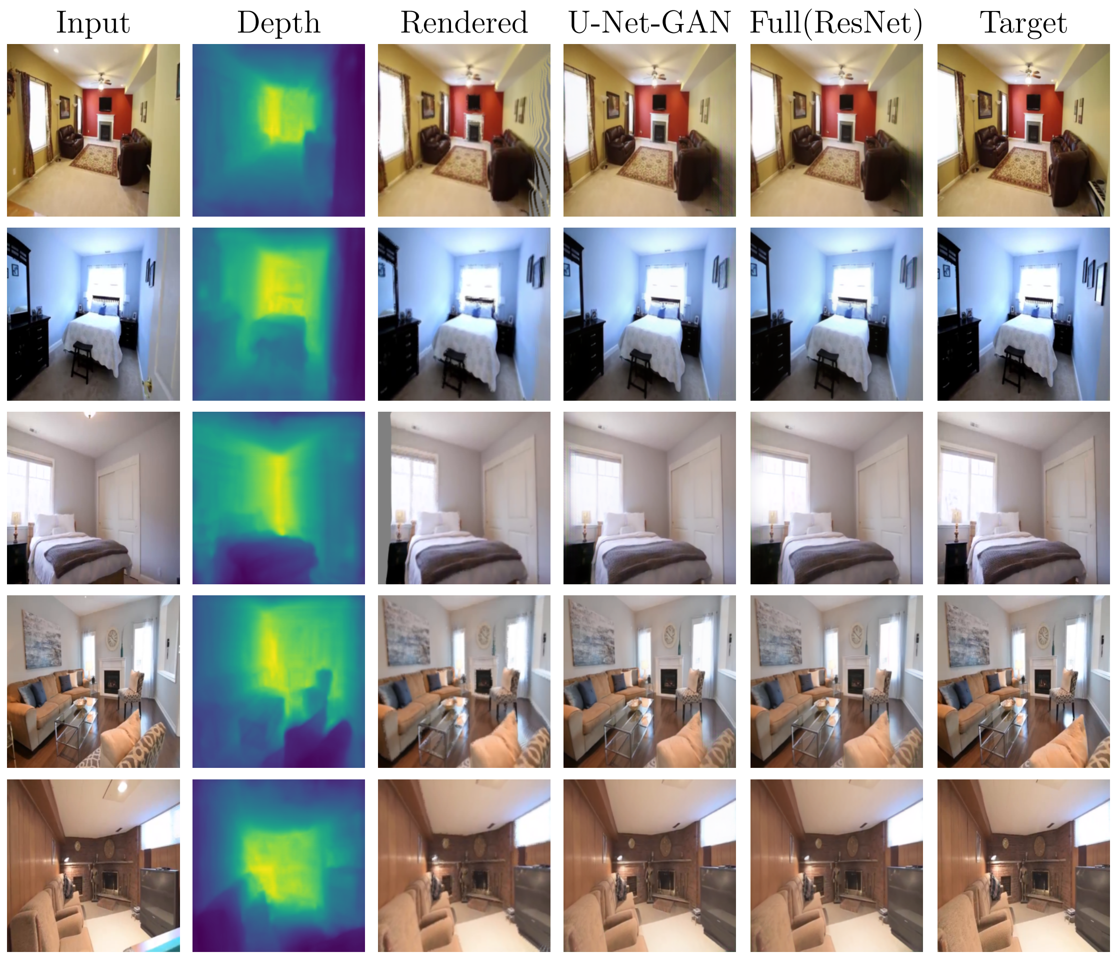


Figure 4.13: Example sequences for medium to large camera movements along the trajectory but always towards the scene. Our U-Net (finetuning) is compared with the full model that uses the ResNet refinement network. All sequences are extracted from the SynSin RealEstate test set [2].

Depth: Ours vs. SynSin Figure 4.14 and Figure 4.15 show a comparison of depth maps generated with our pipeline and depth maps generated with the pipeline of Wiles et al. [2]. As the figures show, we generate significantly better and smoother depth maps. At the SynSin depth maps one can clearly see that gradients in the colour image often also represent gradients in the depth image. This is especially wrong in planar regions with a lot of texture.

Almost all SynSin depth maps have incorrect values at the image borders (see yellow borders in Figure 4.14 and Figure 4.15). We explain our better depth estimation mainly with the introduced losses (see Section 4.4.1), the way we train our networks, and the optimized architecture of our disparity network.

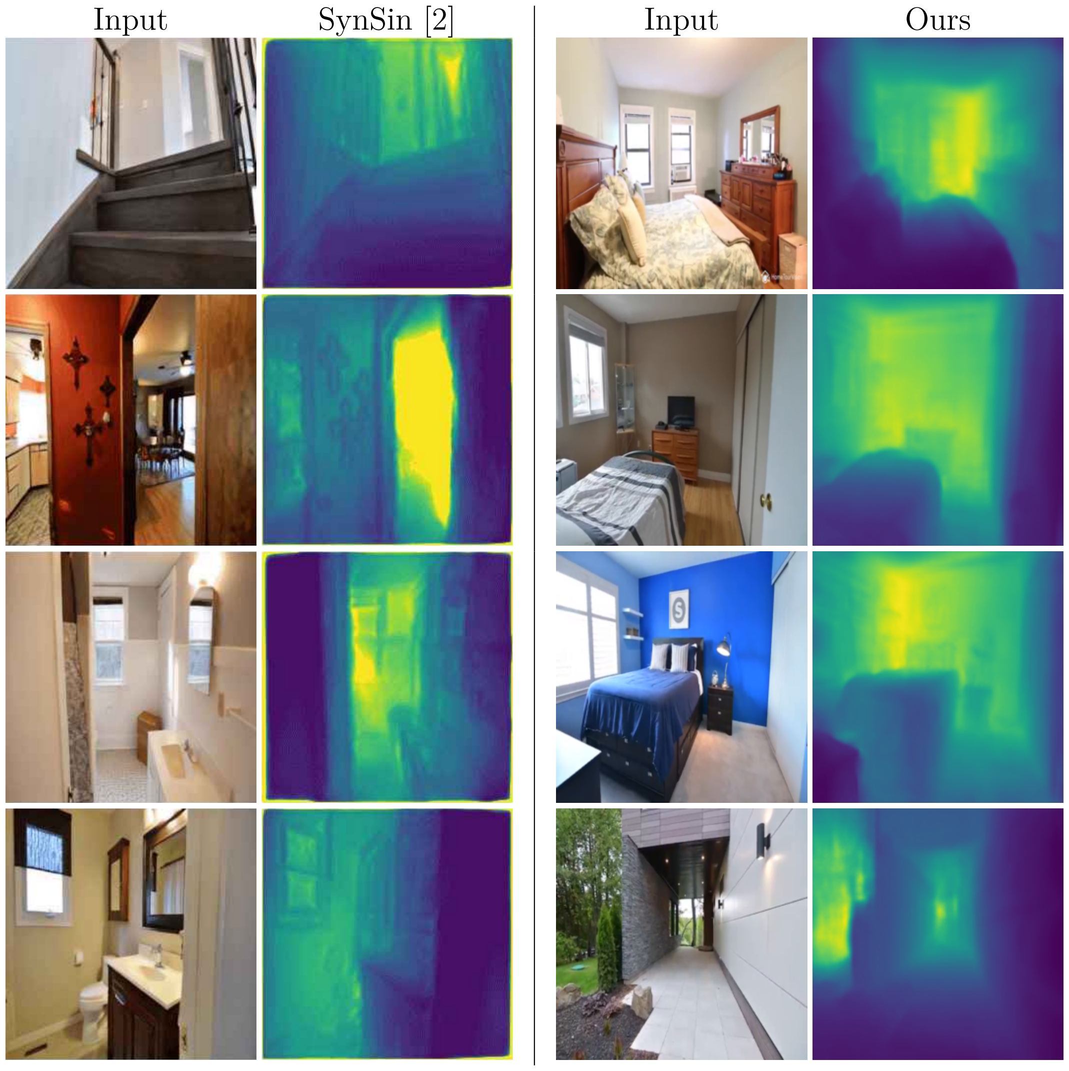


Figure 4.14: Depth generated by SynSin(full) [2] (left) and our pipeline (right). All of our images are extracted from the SynSin RealEstate test set.

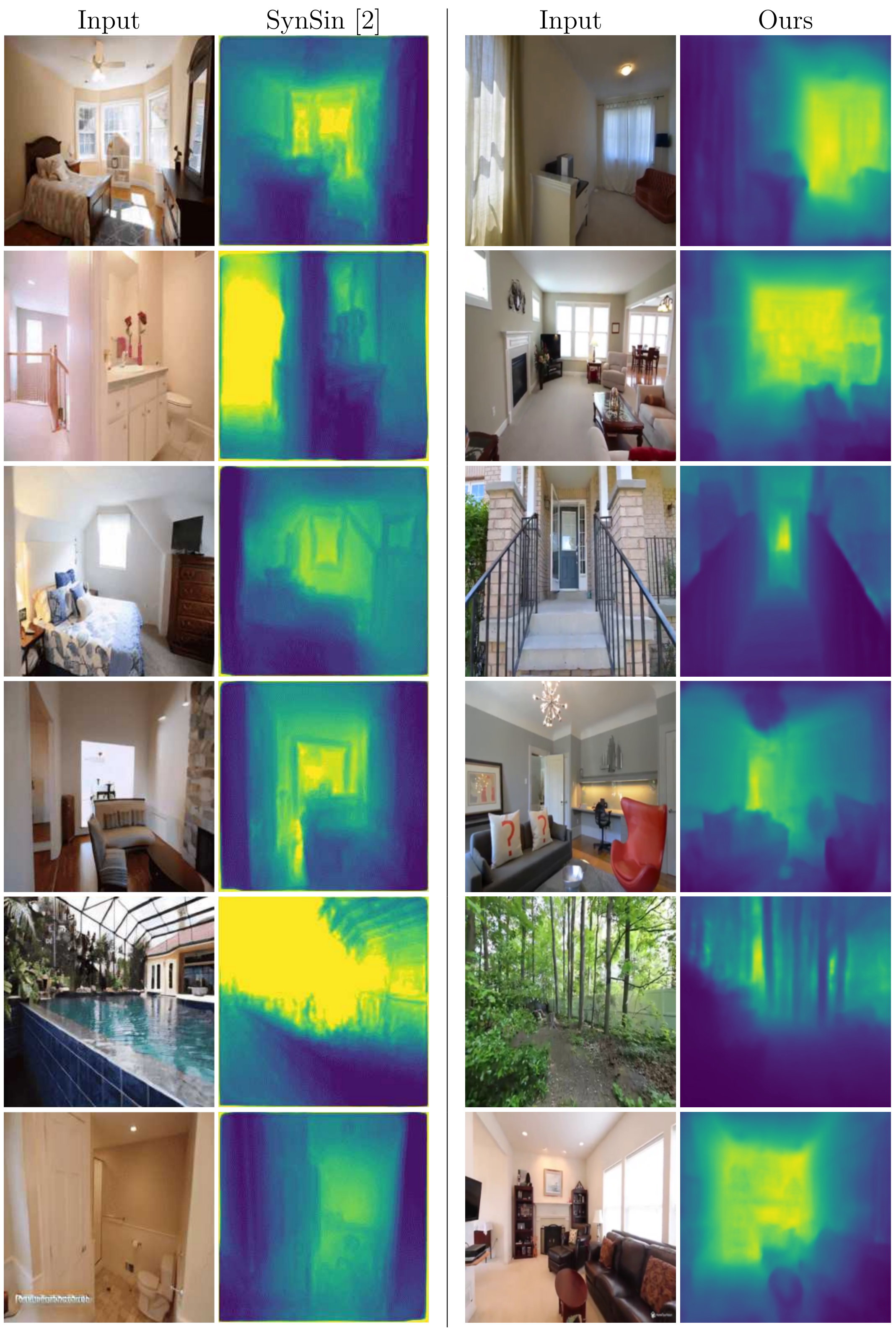


Figure 4.15: Depth generated by SynSin(full) [2] (left) and our pipeline (right). All of our images are extracted from the SynSin RealEstate test set.

4.5.6 Limitations

Due to the relatively small receptive field of the ResNet refinement network, large inpainting areas are problematic and often cause blurred regions, as shown in Figure 4.11. Our U-Net network has a larger receptive field but still significantly fewer parameters than ordinary inpainting networks as, for example, introduced by Yu et al. [16]. In inpainting regions the content has to be guessed by the context and therefore often does not correspond with the target image (see Figure 4.16). Since this approach consists of several processing stages (depth-estimation, rendering, refining), real-time with high-resolution images (as in our second approach FaDIV-Syn) becomes problematic. The latency of our U-Net pipeline is about 12.4 ms for an image resolution of 256×256 and 56.1 ms for 512×512.

Furthermore, this approach can only process individual images and thus forgoes other available cameras that are especially helpful when estimating depth.

We focused on improving the quality of the monocular estimated scene geometry with a simultaneous shrinking in complexity and improvement of the latency. However, monocular depth estimation remains an ill-posed problem, which is the reason why our disparity network often struggles in estimating the correct scale. Incorrect scaling for given camera motions lead to incorrect projections and hence errors in prediction, as illustrated in Figure 4.16.

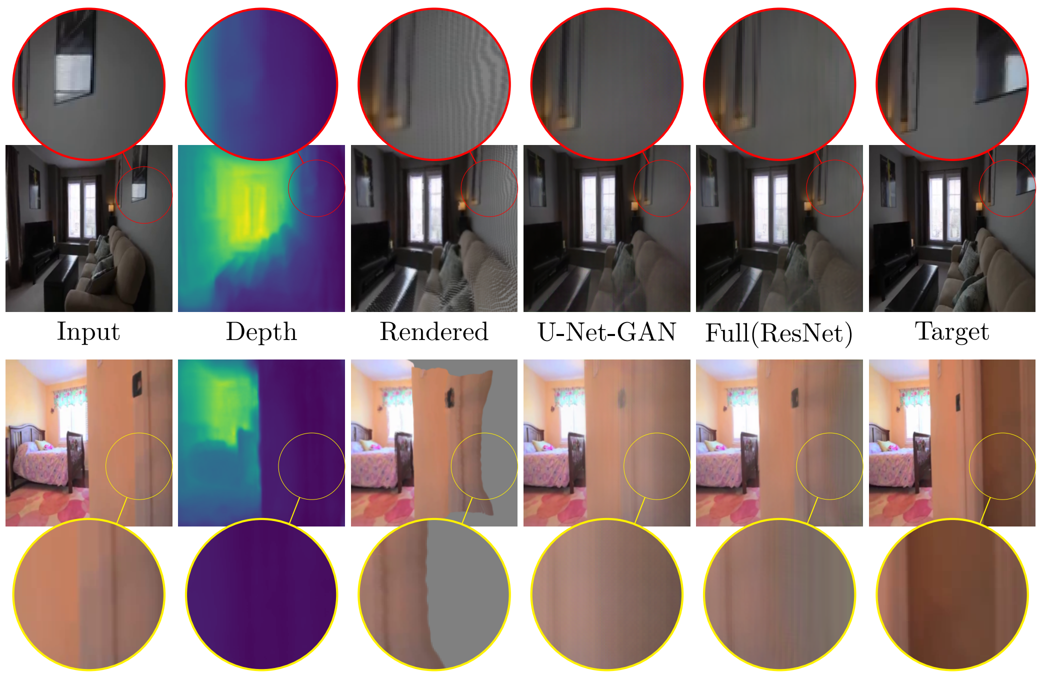


Figure 4.16: Limitations of our pipeline: The upper image visualizes an incorrect scale estimation and the lower image shows that inpainting areas often do not corresponds with the target image.

5 Fast Depth-Independent View Synthesis (FaDIV-Syn)

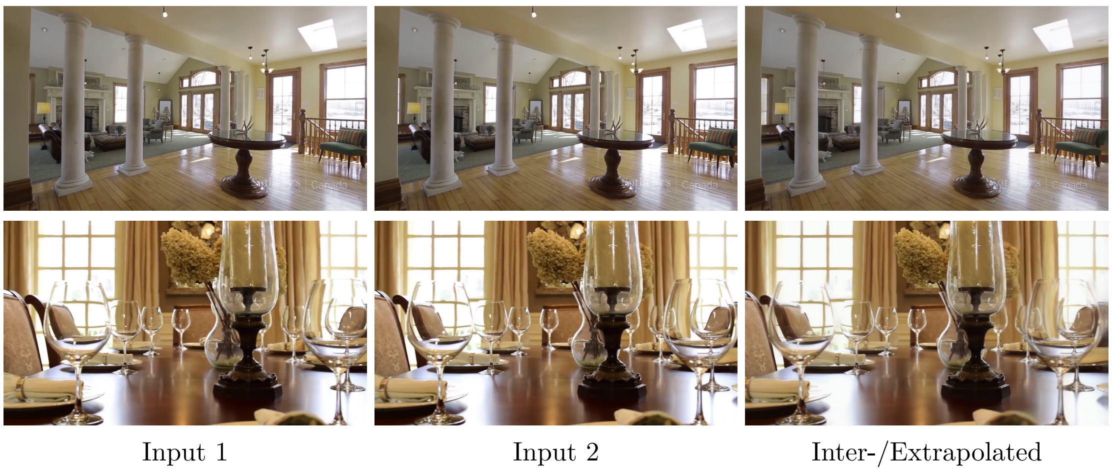


Figure 5.1: FaDIV-Syn inter- and extrapolates images from two views. Input images from RealEstate10k [1] test set. This figure is animated—if the videos are not visible we refer to our supplementary material.

Our first approach deals exclusively with single view synthesis. For an avatar, however, a stereo camera pair is usually available, and even more cameras can be easily added. One could extend the previous approach to multiple input cameras, but this is not very real-time-capable due to the multilevel processing. Furthermore, the computed depth in the input images would have to correspond pairwise to be of real use. Unfortunately it is not trivial to extract corresponding depth in a multiview setup as appearing in RealEstate10k. Choi et al. [45] estimate depth probabilities instead of a fixed depth value for each camera ray. When combining the depth probabilities of multiple input images, they search for intersecting rays to predict depth in the target image. Since their depth estimation is very time consuming, real-time processing becomes problematic. Hence we decided to develop a new approach which is totally independent of depth and is based on image warping only.

The key idea of FaDIV-Syn is to preprocess and transform the input images into the target frame without losing information. Of course, the transformation requires depth information. Instead of estimating depth, we sample multiple depth variants (see Section 5.0.2) and present the resulting possibilities to the network. The induced representation is well-suited for the view synthesis task.

While in principle FaDIV-Syn can operate with any number of input images, we present and evaluate the method for two input images (in the following called I_1 and I_2) and one output image (I_O). For both input cameras, we sample N depth planes and for each plane we assume that the entire image lies on it. When projecting these planes into the target view, one could, for $N \to \infty$, determine each pixel's 3D position by searching for correspondences in the warped planes (see Figures 5.2 and 5.3). When performing the correspondence estimation and merging task with a learned network, we can reduce N to a small number, since the network can learn to interpolate between planes with adjacent depth levels.

Hence we introduce FaDIV-Net that is fully convolutional and processes the warped planes (P_1, P_2) of both input frames. Since we strongly discretize the number of depth planes a perfect correspondence occurs only rarely. However, the network can learn to recognize the best two planes and approximate the correct solution with surrounding textures, as visualized in Figure 5.2.
Our entire forward pipeline consists of the following steps:

1. Choose a target frame $\tilde{I}_O$, two input frames $I_{1,2}$ and their corresponding camera poses ${}^{O}T_{W}$, ${}^{1}T_{W}$, ${}^{2}T_{W}$ and camera intrinsics K.

2. Align N predefined depth planes in the target frame pose as described in Section 5.0.1.

3. Use the aligned depth planes $P^i_{1,2} \in P$, with $i \in \mathbb{N}_{<N}$ and assume they are textured with the entire images.

4. Warp the textured planes into the target frame using the inverse homography matrix (see Section 2.2.1) and the transformations ${}^{1}T_{O}$ and ${}^{2}T_{O}$.

5. Concatenate the warped planes in their channel dimension so that $\tilde{P}$ has shape $(2 \cdot 3 \cdot \mathrm{N}) \times \mathrm{H} \times \mathrm{W}$, where $\tilde{P}$ are the concatinated warped planes of both input images.

5. Run the foward pass of FaDIV-Net: $F(\tilde{P}) = I_O$, where F directly predicts the output image I_O from the given plane sweep volume $\tilde{P}$.

5.0.1 Plane Sweep Volume and Alignment

Planar geometry is especially well-suited for camera-to-camera projection, since the resulting warping operation can be done efficiently. A naïve approach might

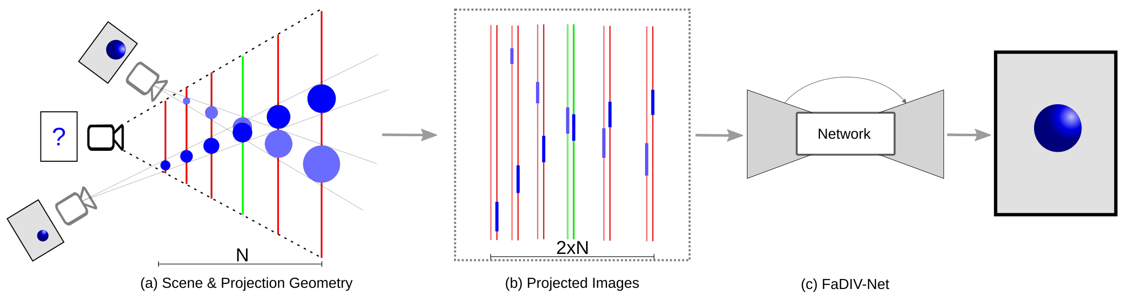


Figure 5.2: The FaDIV-Syn architecture. (a) Input images (gray) are projected into the target camera (black) for each depth plane (red/green) defined in the target frame. For a particular surface in the scene (blue circle), there will be a depth plane where the projections most closely align (green). This plane corresponds to the true object depth. (b) The resulting projected images are stacked and fed into the view synthesis network (c), which directly predicts the target image.

be to define depth planes in both input images I_1 and I_2. This corresponds to computing two PSVs and then attempting to merge them. However, this is not a well-designed representation, since corresponding elements might be on very different depth planes (see Figure 5.4).

Instead, we define the planes in the target image I_O (see Figure 5.2), as it is commonly done in plane sweeping multi-view stereo approaches [14]. For each plane i, we define $P_k^{(i)} \in P$ as the image resulting from projecting I_k onto the plane, and then into I_O. Using this representation, we can define a “hard-wired” view synthesis method f:

$$f(I_1, I_2, p) = \begin{cases} P_1^{(D(p))}(p) & \text{if } p \text{ visible in } I_1, I_2 \\ g(I_1, I_2, p) & \text{otherwise ,} \end{cases} \tag{5.1}$$

$$D(p) = \arg\max_j Q(P_1^{(j)}, P_2^{(j)}, p), \tag{5.2}$$

where $p = (x, y)$ is a pixel in the target image I_O, g is an inpainting method, D is the (internal) depth estimate, Q is a correspondence quality estimator, and j denotes the plane with optimal correspondence. Note that only images in P of the same depth need to compared. If the planes were not aligned (Figure 5.4), we would need to compute $\arg\max_j(\max_i(Q(P_1^{(j)}, P_2^{(i)}, p))$, which is obviously more complex and harder to learn. Furthermore, we achieve a higher generalization to any camera setup, since the alignment creates a pose-invariant representation. Figure 5.3 shows an exemplary PSV where the idea presented in Equation (5.1) will be immediately apparent. Since perfect Q and g are not known, we will learn a CNN approximating f.

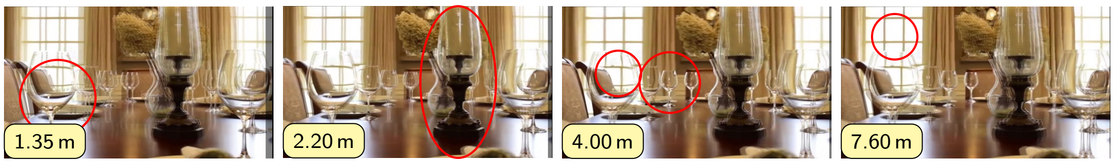


Figure 5.3: Plane sweep volume (PSV). Shown are four planes of the Figure 5.1 scene with α-blended projections of the two input images. Note that RealEstate10k has only estimated global scale, so the given plane distances are only accurate up to scale.

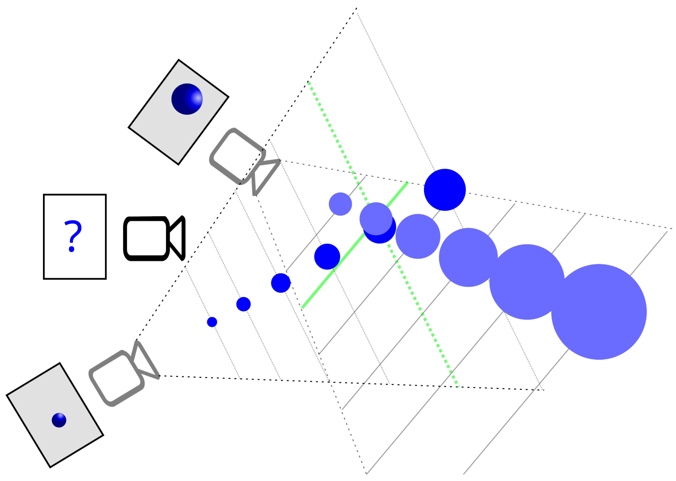


Figure 5.4: When defining depth planes depending on the individual camera's coordinate system (compare against Figure 5.2), correspondences end up on different depth planes, making the representation harder to process.

Plane Alignment. Given a plane $L_O^{(j)} = (0 \quad 0 \quad 1 \quad -d_j)^T$ defined in the target frame I_O, we can find the plane in frame I_i as:

$$L_i^{(j)} = {}^O T_i^{\;-T} \cdot L_O^{(j)T}, \tag{5.3}$$

where ${}^O T_i$ is the homogenous transformation between output and input frame i. We warp images efficiently using the inverse homography matrix, as described in Section 2.2.1.

5.0.2 Depth Discretization

In real world scenarios, the depth relative to the camera lens can take any positive value $d \in \mathbb{R}_{>0}$. For increasingly distant objects, the spatial error of projected pixels caused by wrong depth decreases. Therefore, it is legitimate to set a maximum depth. Our main network uses only 17 depth planes, where we distribute the planes uniformly in disparity space for the depth range [0.3,8] m and one additional background plane at 16 m. Compared to related work [12, 1], this discretization is quite severe. The RealEstate10k dataset, in turn, contains a large number of drone-captured scenes with very large distances but also close geometry, which makes the task even more difficult.

5.0.3 Generator Network

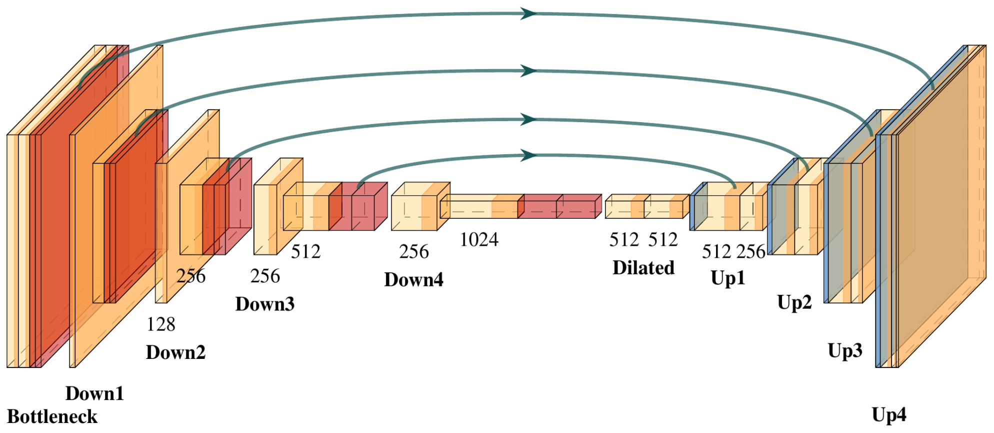


Figure 5.5: FaDIV-Net base architecture without group convolution stage. Yellow blocks denote Conv2D+BN+ReLU, red blocks show gating, and blue blocks transposed convolutions. For a more detailed description, we refer to Table 5.1.

When projecting an image according to a known depth map, the disocclusion areas requiring inpainting are those where no pixel from the image is mapped to. When considering a PSV, disocclusion areas are no longer trivial to determine. Thus, our network must learn to distinguish between (1) areas of sufficient correspondences in the warped planes, (2) areas of no correspondence but sufficient correspondence in different planes and (3) areas of disocclusion and occlusion. Hence the network must learn under the constraint of geometric consistency to fuse and correct sufficiently corresponding areas from warped planes and recognize inpainting areas to fill them with realistic content. In contrast to Thies et al. [41], who also learn implicit blending, we avoid early decisions in a depth estimation stage, which may lead to quality degradation later in the pipeline.

Our main network is based on a U-Net [71] architecture as shown in Figure 5.5. It consists of four downsampling and four upsampling blocks with skip connections and a dilated convolution in the middle. For upsampling we use transposed convolutions. Each downsampling block consists of two convolutions with batch normalization and ReLU activation. The second convolution has a stride of two for downsampling. Afterwards, we split the feature dimension in half and multiply the two parts to realize gating, as explained in Section 2.1.2. Since the network is supposed to process high-resolution input images of dimension 2×N×3×H×W, we send the entire input at the first gated convolution through a bottleneck. This way, the network can eliminate non-corresponding and combine corresponding areas.

Gated Convolutions. Gated convolutions [15] have recently shown promising results in image inpainting [16]. Inpainting areas especially occur at the image edges or in disocclusion areas, where information is available in none of the input frames. The Equation (2.2) in Section 2.1.2 shows that gating directly influences the actual feature extraction and can thus adapt it to the context. Gating layers can thus help the network especially with performing masking-like operations (e.g. when recognizing corresponding depth planes), performing blending, or determining inpainting areas [16].

5.0.4 Extended Architecture

Aligning the depth planes allows us to further extend the network. The alignment ensures that corresponding areas only appear in corresponding planes (see Figure 5.2). Hence we preprocess the corresponding planes by a gated group convolutional planes (GGC) before they enter the base network. This can be done under the assumption that other planes are initially irrelevant to the considered pair. Grouping saves processing time and makes it easier for the network to comprehend the context of the task to be learned. Additionally, the alignment allows us to pre-compute a pairwise correspondence metric between the planes. We concatenate a Structural Similarity [24] map to each corresponding plane pair, which can be used by the network as an aid. Especially gated convolutions can help the network to understand the connections between the similarity maps and the task to be accomplished in these areas, as Yu et al. [16] showed in the context of image inpainting. Furthermore, one can enhance the receptive field of the network in the group convolutional part if necessary, which is more time-efficient than in the base network. A detailed description of the extended architecture can be found in Table 5.1.

5.0.5 Training

The network is trained in a supervised manner from a triple $(I_1, I_2, \tilde{I}_O)$ with known camera poses and intrinsics. We define the loss function

$$\mathcal{L}(I_O, \tilde{I}_O) = \lambda_1 \mathcal{L}_1(I_O, \tilde{I}_O) + \lambda_p \mathcal{L}_{perceptual}(I_O, \tilde{I}_O), \tag{5.4}$$

Input	k_1	c_1	k_2	c_2	Output
P	3	102	3	204	*groupconv*
$\mathbf{G}(groupconv)$	3	112	3	112	*bottleneck*
$\mathbf{G}(bottleneck)$	3	112	3	224	$down_1$
$\mathbf{G}(down_1)$	3	128	3	256	$down_2$
$\mathbf{G}(down_2)$	3	256	3	512	$down_3$
$\mathbf{G}(down_3)$	3	256	3	1024	$down_4$
$\mathbf{G}(down_4)$	3	512	3	512	*dilated*
$\mathbf{Up}(dilated, G(down_3))$	3	512	3	256	up_1
$\mathbf{Up}(up_1, G(down_2))$	3	256	3	224	up_2
$\mathbf{Up}(up_2, G(down_1))$	3	224	3	112	up_3
$\mathbf{Up}(up_3, G(bottleneck))$	3	112	3	32	up_4
up_4	1	3	–	–	*pred*

Table 5.1: FaDIV-Full-17 architecture. Each row shows two convolutional layers, where k is the kernel size and c is the number of output features.

where the Perceptual loss is based on a VGG-19 [22] network, as described in Section 2.3.2.

We train the network with a batch size of 20 and the Adam optimizer with $\beta_{1,2} = (0.4, 0.9)$ on two NVIDIA RTX 3090 GPUs with 24 GiB RAM. Training takes four days for images of 288p resolution. Higher resolution training at 576p is only possible with a batch size of six and takes up to three weeks.

For 288p images, we train the networks for 300k-350k iterations and for 576p images we increase the number of iterations accordingly to adjust to the smaller batch size. Within this range, we use early stopping based on the validation score to select the model for evaluation. Furthermore, we train all our models with a learning rate of 1e-4, and a final tangent hyperbolic (tanh) non-linearity unless we explicitly specify it with "lin" (linear).

Interpolation. The first interesting problem setting is interpolation, i.e. when the target camera pose is roughly between the two input frames. For this, we randomly choose a target image $\tilde{I}_O$ and source frames Δt before (I_1) and after (I_2) it. Δt is uniformly sampled from the interval $[4, 13]$. Note that extrapolation areas occur, since the camera never moves perfectly on a straight line. We train and evaluate with a resolution of 518×288 (288p). All processing steps in the pipeline can be performed on the GPU, where the homography warping takes less than 1 ms to warp 34 rgb-textured-planes in 288p and approximately 1.5 ms for 34 rgb-planes in 540p.

Extrapolation. In order to investigate whether FaDIV-Syn is also suitable for extrapolation, we add extrapolation triplets to the training. Extrapolation is much more difficult to learn, since the network must now inpaint in disoccluded areas. Further, it must learn to detect those disocclusion areas in the warped planes. To learn such extrapolations we use an alternative data loading method that additionaly provides extrapolation sequences to the network. Here we randomize the triplet positions as follows:

1. **Frame 1 index:** $i_1 = x$
2. **Frame 2 index:** $i_2 = i_1 + n$, $n \in [3, 4, 5]$
3. **Target frame index:** $i_O = i_2 + m$, $m \in [5, 6, 7]$.

Note that the skip values n, m are chosen uniformly and x is a random position within a RealEstate10k scene. Extrapolation and interpolation triplets are mixed 80:20 during training and training is started from a pre-trained interpolation network. Accordingly, we only train 200k-250k iterations.

Shih et al. [21] evaluated different state-of-the-art extrapolation approaches, with a resolution of 1024×576. For a fair comparison, we introduce a variation of our FaDIV network called FaDIV-Big. FaDIV-Big has one more downsampling layer and a larger Gated-Group kernel size, where $k_1 = 7$ and $k_2 = 5$, as shown in Table 5.2. In our evaluation, we follow the setup of Shih et al. [21] to be comparable.

Input	k_1	c_1	k_2	c_2	Output
P	7	102	5	204	*groupconv*
$\mathbf{G}(groupconv)$	3	112	3	112	*bottleneck*
$\mathbf{G}(bottleneck)$	3	112	3	224	$down_1$
$\mathbf{G}(down_1)$	3	128	3	256	$down_2$
$\mathbf{G}(down_2)$	3	256	3	512	$down_3$
$\mathbf{G}(down_3)$	3	256	3	1024	$down_4$
$\mathbf{G}(down_4)$	3	512	3	2048	$down_5$
$\mathbf{G}(down_5)$	3	1024	3	1024	*dilated*
$\mathbf{Up}(dilated, G(down_4))$	3	1024	3	512	up_1
$\mathbf{Up}(up_1, G(down_3))$	3	512	3	256	up_2
$\mathbf{Up}(up_2, G(down_2))$	3	256	3	224	up_3
$\mathbf{Up}(up_3, G(down_1))$	3	224	3	112	up_4
$\mathbf{Up}(up_4, G(bottleneck))$	3	112	3	64	up_5
up_5	1	3	–	–	*pred*

Table 5.2: FaDIV-Big-17 architecture. Each row shows two convolutional layers, where k is the kernel size and c is the number of output features. The second block (*bottleneck*) contains one convolution (k_1,c_1) and two convolutions (k_2, c_2).

5.1 FaDIV-Syn Evaluation

We evaluate our method on the challenging RealEstate10k dataset. Since some of the YouTube videos cannot be downloaded anymore, we only managed to obtain 74.5k video clips. We split the official *train* split further into 54k training and 13.5k validation sequences. All our tests are done using the official test split, except the extrapolation experiments, where we use the data provided in [21]. We train and evaluate with a resolution of 518×288 (288p) unless otherwise mentioned. For both modes (interpolation and extrapolation), we evaluate the quality of generated images with the PSNR, SSIM [24], and LPIPS [25] metrics.

Reference Network: FaDIV-full-17

FaDIV-full-17 is the reference network and uses the extended network architecture, as described in Section 5.0.4. We set the following parameters:

- $\lambda_p = 3.0$, which is the weighting of the Perceptual loss,
- $\lambda_1 = 1$, which is the weighting of the L_1 loss,
- learning-rate = 0.0001,
- $|P_i| = 17$, which are the number of planes per view,
- Final non-linear tanh activation at the output,
- Gated-Group-Convolutions (GGC) as described in Section 5.0.4,
- RGB only $\Rightarrow$ No similarity maps given (see Section 5.0.4).

5.1.1 (Gated) Group Convolutions

We compare a FaDIV-17 network with gated group convolutions (GGC) and group convolutions without gating (no-GGC). Furthermore, we implemented a variation that uses structural-similarity maps as input for all plane pairs (GGC+SSIM), as described in Section 2.3.3. Note that all networks share the same base network (including gated convolutions). To save time, we perform this experiment on a 35% fraction of the real dataset and train only for 200k-250k iterations. As Table 5.3 shows, the gated variant gives better results on the test dataset. Appending pre-estimated Structural Similarity [24] maps in the input performs a little better for image triplets with further distances Δt. We conclude that similarity maps are helpful, but this effect is less pronounced for closer image triplets.

	$\Delta t = 2$			$\Delta t = 5$			$\Delta t = 10$		
Variant	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS
no-GGC	35.358	.9626	.0182	32.087	.9388	.0334	28.408	.8923	.0631
GGC	**36.150**	.9658	**.0158**	32.523	.9417	**.0307**	28.594	.8944	**.0605**
GGC+SSIM	35.860	**.9661**	.0170	**32.541**	**.9436**	.0314	**28.748**	**.8984**	.0609

Table 5.3: Evaluation of gated group convolutions (GGCs).

5.1.2 Interpolation Results

We achieved the best interpolation results with a FaDIV-Net with tanh activation (see Table 5.4). Again, we noticed that the SSIM variant shows an increasingly better performance for further image distances (especially for LPIPS).

In order to push the boundaries of view interpolation inference speed, we investigate a very fast variation with only 13 depth planes and tanh activation (full-13). Comparing the results shows that FaDIV-full-13 still has very good performance but cannot match the results of a FaDIV network with 17 planes. Averaged across all metrics and image distances we obtain a performance decrease of only -3.79%.

	$\Delta t = 2$			$\Delta t = 5$			$\Delta t = 10$		
Variant	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS
full-17-lin	36.120	.9669	.0159	32.418	.9420	.0321	28.526	.8940	.0620
full-17	**36.485**	**.9684**	.0155	**32.799**	**.9452**	.0304	28.820	.8983	.0606
full-17-ssim	36.024	.9662	**.0151**	32.662	.9450	**.0295**	**28.870**	**.9006**	**.0590**
full-13	34.848	.9635	.0172	31.666	.9376	.0326	28.056	.8881	.0634

Table 5.4: Interpolation results on RealEstate10k.

5.1.3 Generalization and Data Efficiency

Data Efficiency. To investigate the dependency of FaDIV-Net on available training samples, we train on smaller fractions of the full RealEstate10k training dataset, and evaluate on the full test set. The dataset size is reduced by randomly choosing scenes until the specified size is met (35%, 5%, 1%, and 0.1%).

As Table 5.5 shows, all sizes from 35% to 1% give sufficiently good results, where 35% even performs similarly or slightly better than our model trained on the full dataset. We note that training was stopped after 350k iterations, so it is possible that training further on the full dataset may yield advantages. However, we conclude that 35% of RealEstate10k still contains enough scene and pose variance to prevent the network from overfitting. This is to be expected, since the triplet sampling during training greatly augments the number of training samples. If we train on 5% of the data, we loose only -2.1% in performance compared to the full training set. The 1% network maintains its performance for SSIM and PSNR but starts losing significant performance in LPIPS. Finally, the 0.1% network loses significant performance in all metrics and seems to be outside of the boundary for satisfactory results. As Figure 5.6 shows, we observed significant drops in validation performance for the 1% and 0.1% training split.

Starting with 35%, we observe that the validation score is actually better than the training score (see Figure 5.6), which is caused by batch normalization: The average parameters used during evaluation seem to work more robustly than the on-line statistics computed for each batch during training. As Table 5.5 shows, FaDIV-Net already achieves very good results with only the base-network architecture (Section 5.0.3).

Overall, we conclude that from 35% on there are no indications of overfitting at all. Furthermore, FaDIV-Net has very good generalization ability and thus already reaches a similar level of performance with significantly fewer data.

Train	Model	$\Delta t = 2$			$\Delta t = 5$			$\Delta t = 10$		
		PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓
0.1%	full-17-lin	$31.91_{-11.7\%}$	$.9361_{-3.19\%}$	$.0381_{+140\%}$	$28.03_{-13.5\%}$	$.8825_{-6.32\%}$	$.0712_{+122\%}$	$24.70_{-13.4\%}$	$.8129_{-9.07\%}$	$.1257_{+103\%}$
1%	full-17-lin	$34.93_{-3.31\%}$	$.9607_{-0.64\%}$	$.0191_{+20.1\%}$	$31.38_{-3.20\%}$	$.9320_{-1.06\%}$	$.0360_{+12.2\%}$	$27.60_{-3.24\%}$	$.8789_{-1.69\%}$	$.0695_{+12.1\%}$
5%	full-17-lin	$35.19_{-2.58\%}$	$.9616_{-0.55\%}$	$.0173_{+8.81\%}$	$31.92_{-1.54\%}$	$.9366_{-0.57\%}$	$.0327_{+1.87\%}$	$28.20_{-1.16\%}$	$.8874_{-0.74\%}$	$.0631_{+1.77\%}$
35%	base17-lin	$35.85_{-0.76\%}$	$.9655_{-0.15\%}$	$.0161_{+1.26\%}$	$32.27_{-0.45\%}$	$.9403_{-0.18\%}$	$.0314_{-2.18\%}$	$28.41_{-0.42\%}$	$.8918_{-0.25\%}$	$.0615_{-0.81\%}$
	full-17-lin	$\mathbf{36.15}_{+0.08\%}$	$.9658_{-0.11\%}$	$\mathbf{.0158}_{-0.63\%}$	$\mathbf{32.52}_{+0.32\%}$	$.9417_{-0.03\%}$	$\mathbf{.0307}_{-4.36\%}$	$\mathbf{28.59}_{+0.24\%}$	$\mathbf{.8944}_{+0.05\%}$	$\mathbf{.0605}_{-2.42\%}$
100%	full-17-lin	$36.12_{+0.00\%}$	$\mathbf{.9669}_{+0.00\%}$	$.0159_{+0.00\%}$	$32.42_{+0.00\%}$	$\mathbf{.9420}_{+0.00\%}$	$.0321_{+0.00\%}$	$28.53_{+0.00\%}$	$.8940_{+0.00\%}$	$.0620_{+0.00\%}$

Table 5.5: Data efficiency experiment. The *Train* column shows the training dataset size relative to the full RealEstate10k train split.

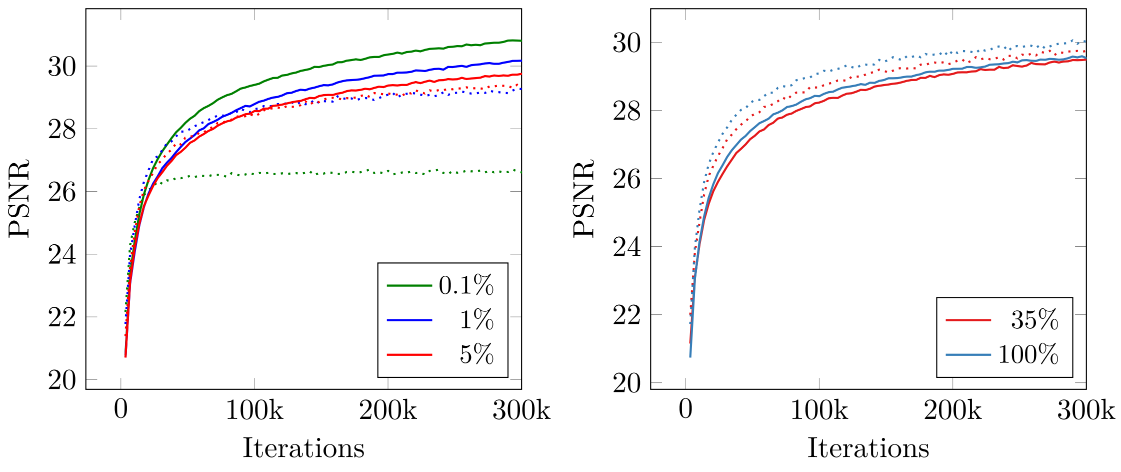


Figure 5.6: PSNR on the different training (solid) and validation (dotted) splits of the RealEstate10k dataset during generator network training.

Generalization to higher resolutions. We also evaluate full-17 and the SSIM variant on higher-resolution images. The networks are trained for image sizes of 512×288 and are afterwards tested for image sizes of 1024×576. The comparison of the networks in Table 5.6 and Table 5.7 shows that FaDIV-Syn generalizes surprisingly well for higher resolution images. Furthermore, it seems that using Structural Similarity maps gives a positive contribution to maintaining quality. This high quality can also be seen in Figure 5.10.

		$\Delta t = 2$			$\Delta t = 5$			$\Delta t = 10$		
Size	Variant	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS	PSNR	SSIM	LPIPS
288p	full-17	**36.485**	**.9684**	.0155	**32.799**	**.9452**	.0304	28.820	.8983	.0606
	+SSIM	36.024	.9662	**.0151**	32.662	.9450	**.0295**	**28.870**	**.9006**	**.0590**
576p	full-17	**35.713**	**.9575**	.0236	31.303	.9254	.0496	26.993	.8677	.1017
	+SSIM	35.360	.9552	**.0230**	**31.315**	**.9265**	**.0484**	**27.134**	**.8728**	**.0993**

Table 5.6: Interpolation results for generalization at inference time to a higher resolution (576p). All models are trained with 288p and have 17 depth planes.

5.1.4 Extrapolation

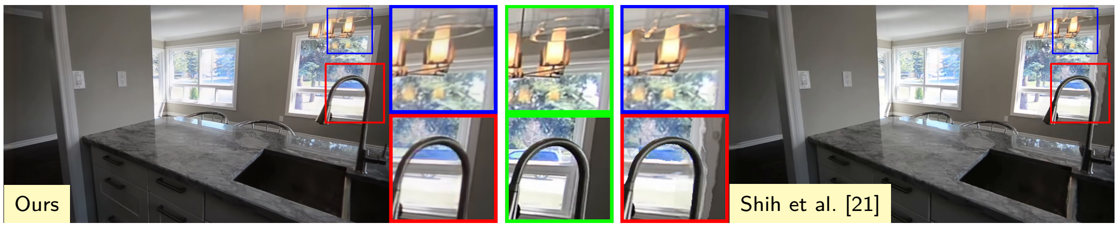


Figure 5.7: Extrapolation results of our method (left) compared with ground truth (center) and Shih et al. [21] (right).

Shih et al. [21] evaluated an array of related methods for extrapolation tasks. In order to test view synthesis accuracy, they generated 1500 random triplets from the test dataset. Note that these experiments were carried out at 1024×576, double the resolution of our previous experiments. As Table 5.7 shows, we outperform current state-of-the-art methods on RealEstate10k. It is interesting that FaDIV-Syn learns to recognize inpainting areas even in disocclusion areas, as can be seen in Figures 5.1 and 5.8.

FaDIV-17 (17 planes). Table 5.7 shows that all our models outperform state-of-the-art results in LPIPS and PSNR. Our SSIM results are better than [56, 12, 45], however [1, 21] report better SSIM results. The last row of Table 5.7 shows our results on smaller images with 512×288, which is interesting but not directly comparable. For a direct comparison with [21], we refer to Section 5.1.8 and Figure 5.7. As shown in Table 5.7, FaDIV-Syn also generalizes to higher resolutions for extrapolation tasks. We outperform state-of-the-art results (LPIPS and PSNR) with a network that has never been trained on images with a resolution of 1024×576. By finetuning to 576p, we only improve the results slightly.

The largest errors mainly occur in scenes with landscapes where objects are very far away from the camera. Unlike [1, 12] we use significantly fewer depth planes, where the last plane is at a distance of 16 m. Therefore, it is very challenging for FaDIV-Net to correctly generate the target view for scenes with large distances. We conclude that a larger depth range or techniques such as depth plane resampling [39] would probably help in these cases.

FaDIV-19 (19 planes). In order to investigate whether we can improve the performance with more planes, especially in larger depth ranges, we train a FaDIV-full19^{+1} model with 19 depth planes, and one additional downsampling layer. Insteat of distributing the first 16 planes between $[0.3, 8]$ m we expand the range

to $[0.4, 11]$ m. Furthermore, we add three background planes at 16 m, 32 m and 96 m. We train FaDIV-full-19^{+1} for 576p explicitly and just like the other networks to be comparable.
As shown in Table 5.7 FaDIV-full-19^{+1} outperforms all FaDIV-17 networks in SSIM and PSNR and all state-of-the-art methods in PSNR and LPIPS. Only FaDIV-17-BIG achieves slightly better LPIPS values. Furthermore, FaDIV-19 reduces the lead of Zhou et al. [1] and Shih et al. [21] in SSIM.
We conclude that a few more planes at larger distances are helpful and give a reasonable performance boost. However, they are not necessary and FaDIV already achieve very good results with 17 planes and a maximum plane distance of 16 m.

To investigate how well a FaDIV-19 network generalizes to higher resolutions and whether we can train longer without overfitting, we finetune a FaDIV-full-19 network for 600k extrapolation iterations. This is significantly longer than the training time of other networks trained on 288p. The training is done on an image resolution of 512×288 and the evaluation is measured on 1024×576. We found that adding a 5th downsampling layer harms the generalization ability of the network. Hence FaDIV-19 keeps the original architecture with four downsampling layers. The last row of Table 5.7 shows that the FaDIV-19$_{(600k)}$ network generalizes very well to higher resolutions and achieves better PSNR and SSIM results than all our networks with 17 planes. We have thus shown that FaDIV-Syn still benefits from more training iterations and consequently we have not yet detected any overfitting.

Note that both FaDIV-19 networks are a bit slower than FaDIV-full-17, but still significantly faster than FaDIV-17-BIG, as can be seen in Table 5.8.

Method	576p	SSIM↑	PSNR↑	LPIPS↓
Stereo-Mag [1]	✓	**0.8906**	26.71	0.0826
PB-MPI (32 Layers) [12]	✓	0.8717	25.38	0.0925
PB-MPI (64 Layers) [12]	✓	0.8773	25.51	0.0902
PB-MPI (128 Layers) [12]	✓	0.8700	24.95	0.1030
LLFF [56]	✓	0.8062	23.17	0.1323
Xview [45]	✓	0.8628	24.75	0.0822
3D-Photo [21]	✓	0.8887	27.29	0.0724
FaDIV-17-BIG	✓	0.8800	28.13	**0.0634**
FaDIV-17	✓	0.8790	28.13	0.0674
FaDIV-17		0.8750	27.96	0.0695
(FaDIV-17 tested on 288p)		(0.8990)	(29.25)	(0.0426)
FaDIV-19^{+1}	✓	0.8866	**28.44**	0.0647
FaDIV-19$_{(600k)}$		(0.8840)	(28.55)	(0.0682)

Table 5.7: Extrapolation results on RealEstate10k [1]. All methods without ✓are trained with 288p, but evaluated on 576p.

5.1.5 Inference Time

Table 5.8 shows the inference time of the different models on a single NVIDIA RTX 3090 GPU. The fastest model that uses the extended network architecture (see Section 5.0.4) is FaDIV-full-13, as it only operates on 13 depth planes. FaDIV-BIG was designed to be comparable with non-real-time approaches. Nevertheless, it is still fast on large images, as we mainly enhanced the receptive field in the group convolution layers.

In addition to results on vanilla PyTorch, Table 5.8 also shows inference times in TensorRT [74], which already boosts performance for float32 precision without losing accuracy. We achieve a throughput of approx. 30 fps for our full models on 540p. TensorRT offers possibilities to quantize the weights of neural networks to float16 or even int8. Our test shows that float16 quantization retains almost 100% accuracy and leads to a significant performance boost: We can achieve up to 85 fps on 960×540 images and more than 300 fps for 512×288 resolution. In comparison, the approaches beating our SSIM score, 3D-Photo and Stereo-Mag, take 2-3 min and 93 ms per 540p image.

	Timings [ms]						Relative Metrics [%]		
	Torch		TRT-32		TRT-16		TRT-16		
Model	288p	540p	288p	540p	288p	540p	SSIM ↑	PSNR ↑	LPIPS ↓
base-17	11.5	37.6	9.5	29.8	3.3	11.8	100.00	99.99	99.92
full-13	11.6	38.1	9.8	30.8	4.2	15.0	100.03	99.99	99.90
full-17	14.7	48.9	11.3	35.8	4.8	18.3	99.98	99.99	100.04
big-17	24.5	78.0	19.5	62.9	8.0	26.2	99.98	99.99	100.03
full-19^{+1}	18.1	57.6	13.4	42.2	5.4	19.9	99.93	100.00	100.78
full-19	16.3	54.2	12.1	39.8	5.2	19.3	99.93	99.95	100.96

Table 5.8: Inference times (ms) of generator networks on NVIDIA RTX 3090. We show native PyTorch as well as TensorRT (TRT) [74] float32/float16 versions. We also analyze the quantized model quality relative to the float32 models (relative metrics). The times do not include PSV generation (1.5 ms @ 540p).

5.1.6 Qualitative Results

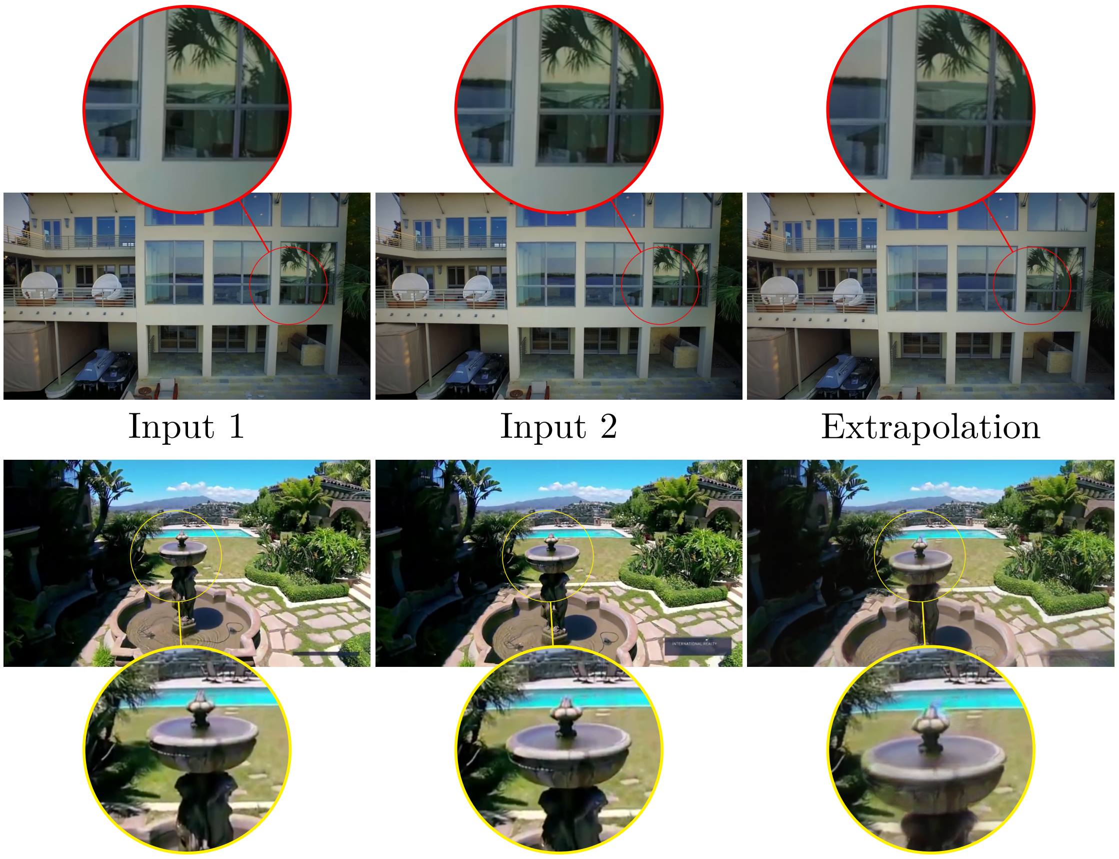


Figure 5.8: Extrapolation on the RealEstate10k test set.

Continuous Depth. The fixed depth planes do not constrain FaDIV-Syn. This effect can especially be seen on straight lines across different depths, which are preserved by our approach (see Figure 5.1 and Section 5.1.8). We conclude that FaDIV-Net does not directly propagate information from the warped planes, but uses them as an orientation.

Occlusions & Disocclusions. Figures 5.1 and 5.8 show examples of disocclusions. FaDIV-Syn can handle disocclusions in both interpolation and extrapolation tasks. FaDIV-Syn separates the pixel information into different depth planes, detects correspondences in these and fuses the planes together. As Figures 5.1 and 5.9 show, FaDIV can handle and represent occlusions.

Reflections and Transparency. Often methods that use depth have problems representing transparencies and reflections, since there is often more than one depth value at a certain pixel location. FaDIV-Syn is designed in such a way that there is not only one depth for each pixel, but a multitude of information in the different depth planes. This allows recognition of the correct position of both the surface and the reflection on it, as shown in Figure 5.9. Examples for transparencies can be found in Figures 5.1 and 5.7.

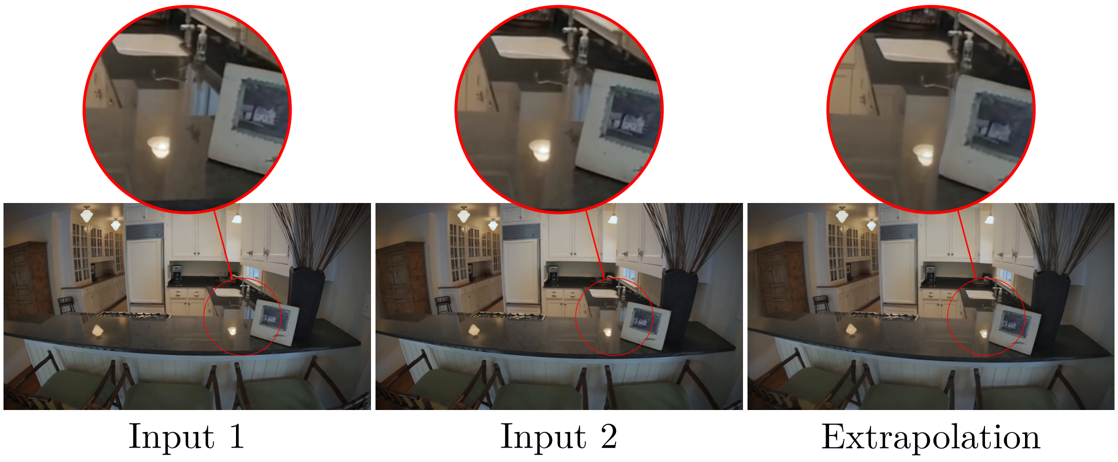


Figure 5.9: FaDIV-Syn is able to represent multiple planes of depth at one location and thus handles reflections correctly. The input images are extracted from the RealEstate10k test set.

Moving Objects. The RealEstate10k dataset does not only contain static scenes. While dynamic scenes are more difficult for view synthesis (and indeed, one could argue that the problem is ill-posed), FaDIV-Syn nonetheless shows interesting behavior on such scenes. For example, the network has learned to interpolate between different positions of moved objects (see Figure 5.10). This behavior would be hard to achieve using a pipeline which estimates depth first, or only blends pixel information.

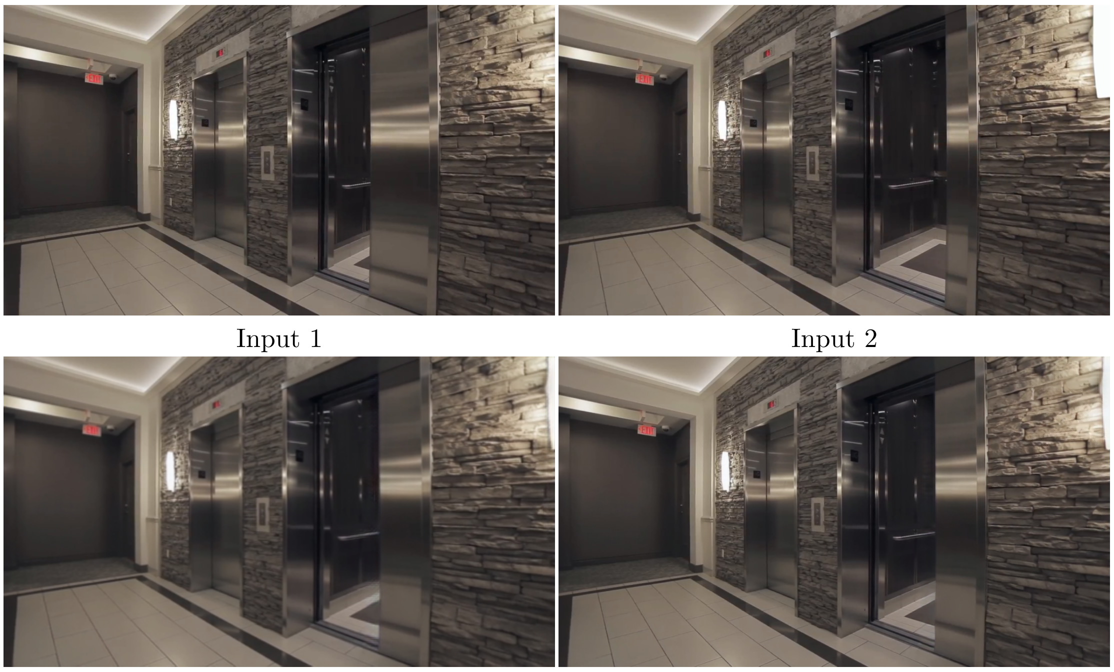


Figure 5.10: Interpolation from two reference views. The lower right video shows the results of a full-17-ssim network trained on 288p but evaluated on 576p. This figure is animated—if the video does not play, we refer to our supplementary material. The input images are extracted from the RealEstate10k test set.

5.1.7 Limitations

Our approach sometimes results in blurred objects under large camera movements. One example is the fountain shown in Figure 5.8. We believe that one reason for this is that we have only trained with smaller camera offsets for extrapolation. Furthermore, we expect that semi-supervised techniques such as [49] could be used to increase the robustness of the method against arbitrary target poses, since the supervision offered by RealEstate10k only covers inter- and extrapolation on smooth camera trajectories. Additionally, the supported depth range is currently limited and could be extended as discussed in Section 5.1.4. Finally, the inference time is limited by the network itself, where compression techniques [75] could be applied to reduce network runtime even further.

5.1.8 Additional Results

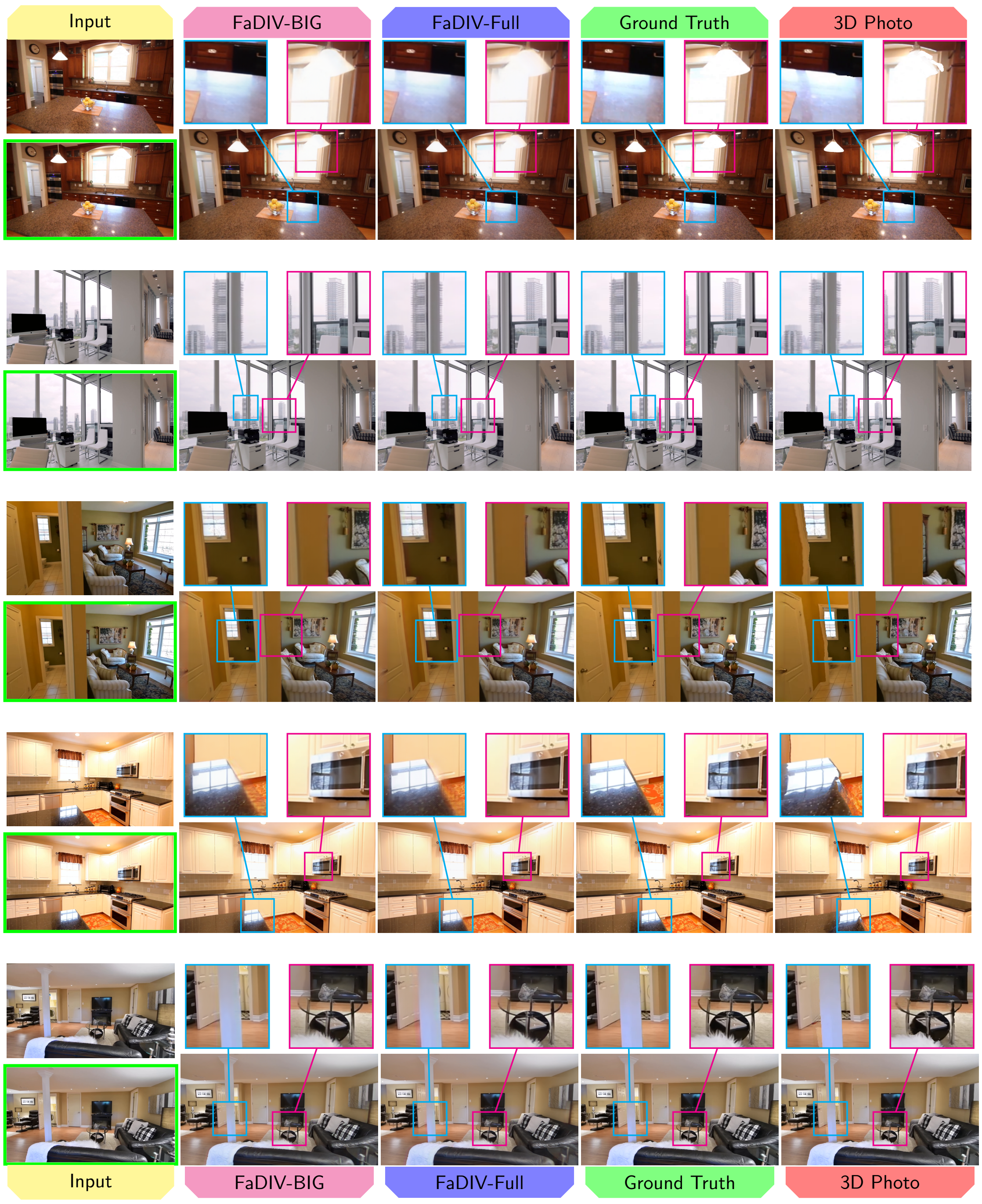


Figure 5.11: Extrapolation comparison of our FaDIV-BIG-17 (high quality) and FaDIV-Full-17 (fast) networks against ground truth and 3D Photo [21]. The input frame closer to the target frame is marked in green for easier comparison.

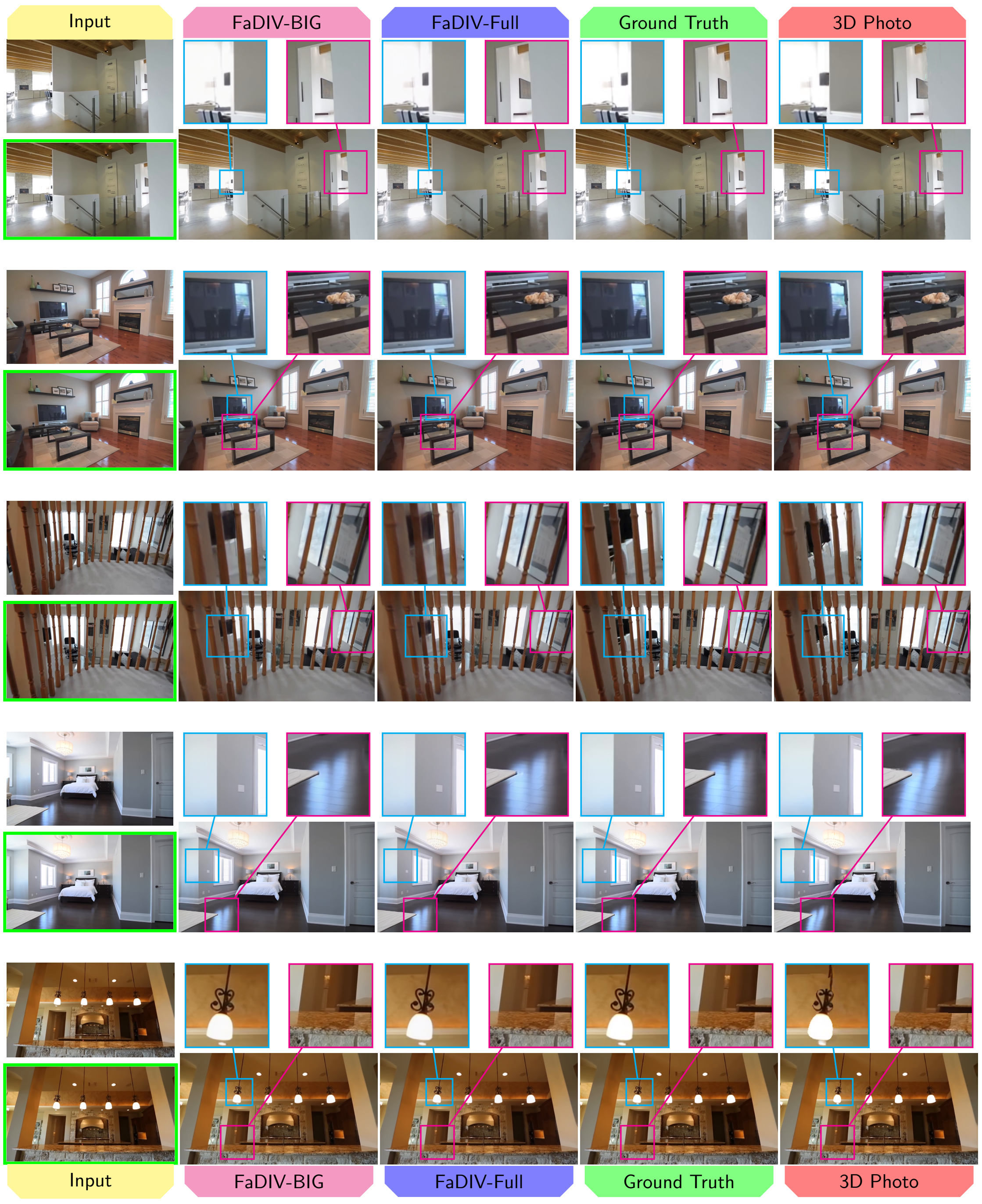


Figure 5.12: Extrapolation comparison of our FaDIV-BIG-17 (high quality) and FaDIV-Full-17 (fast) networks against ground truth and 3D Photo [21]. The input frame closer to the target frame is marked in green for easier comparison.

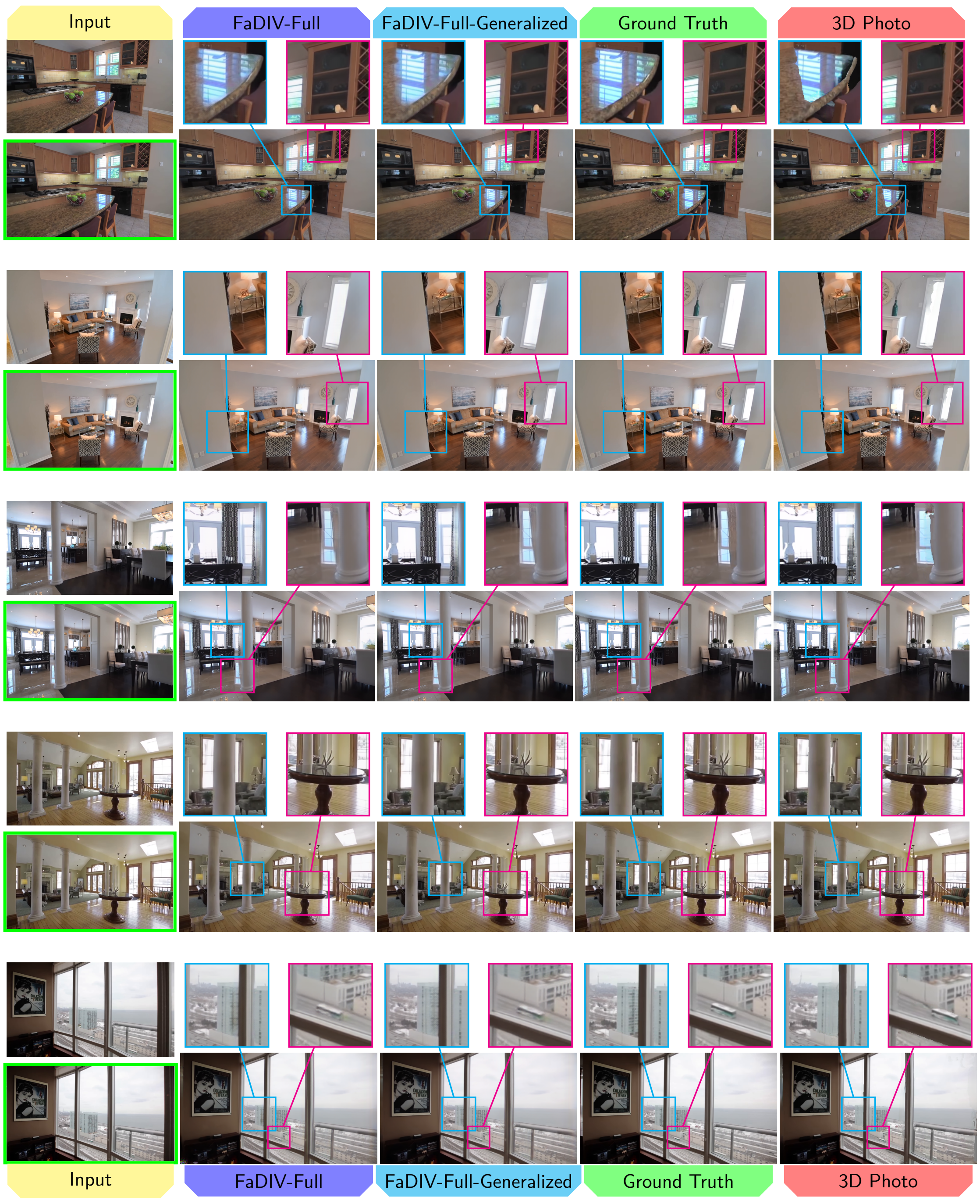


Figure 5.13: Extrapolation comparison of FaDIV-Full-17 trained on 540p (Full) and 288p (Full-Generalized) against ground truth and 3D Photo [21]. The Generalized-Full network runs inference in 540p. The input frame closer to the target frame is marked in green for easier comparison.

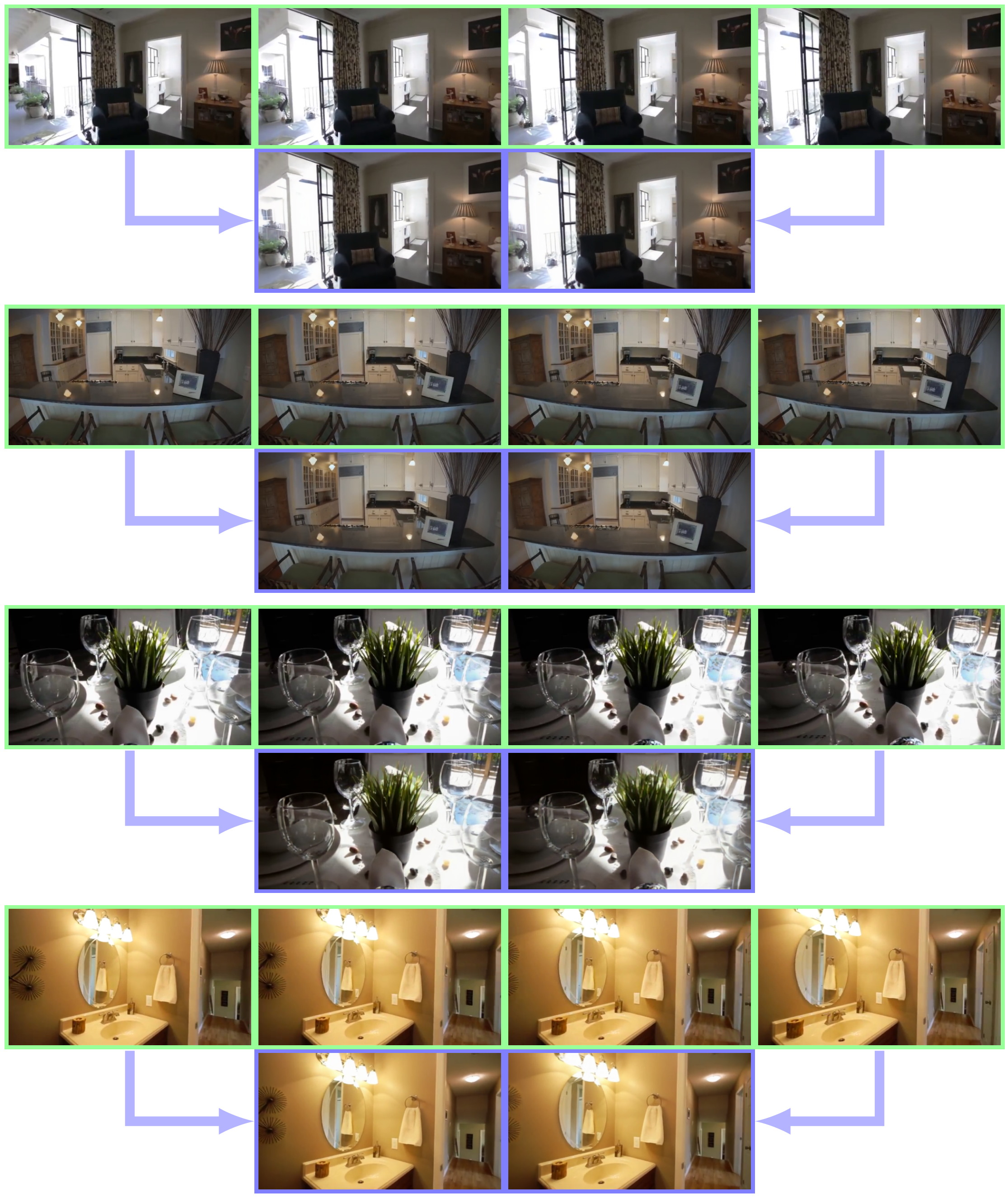


Figure 5.14: Interpolation using our fast FaDIV-Full-17 network. In every block, the top row (green) shows the ground truth trajectory from RealEstate10k (test), while the bottom row (blue) presents the interpolated result corresponding to the ground truth camera poses.

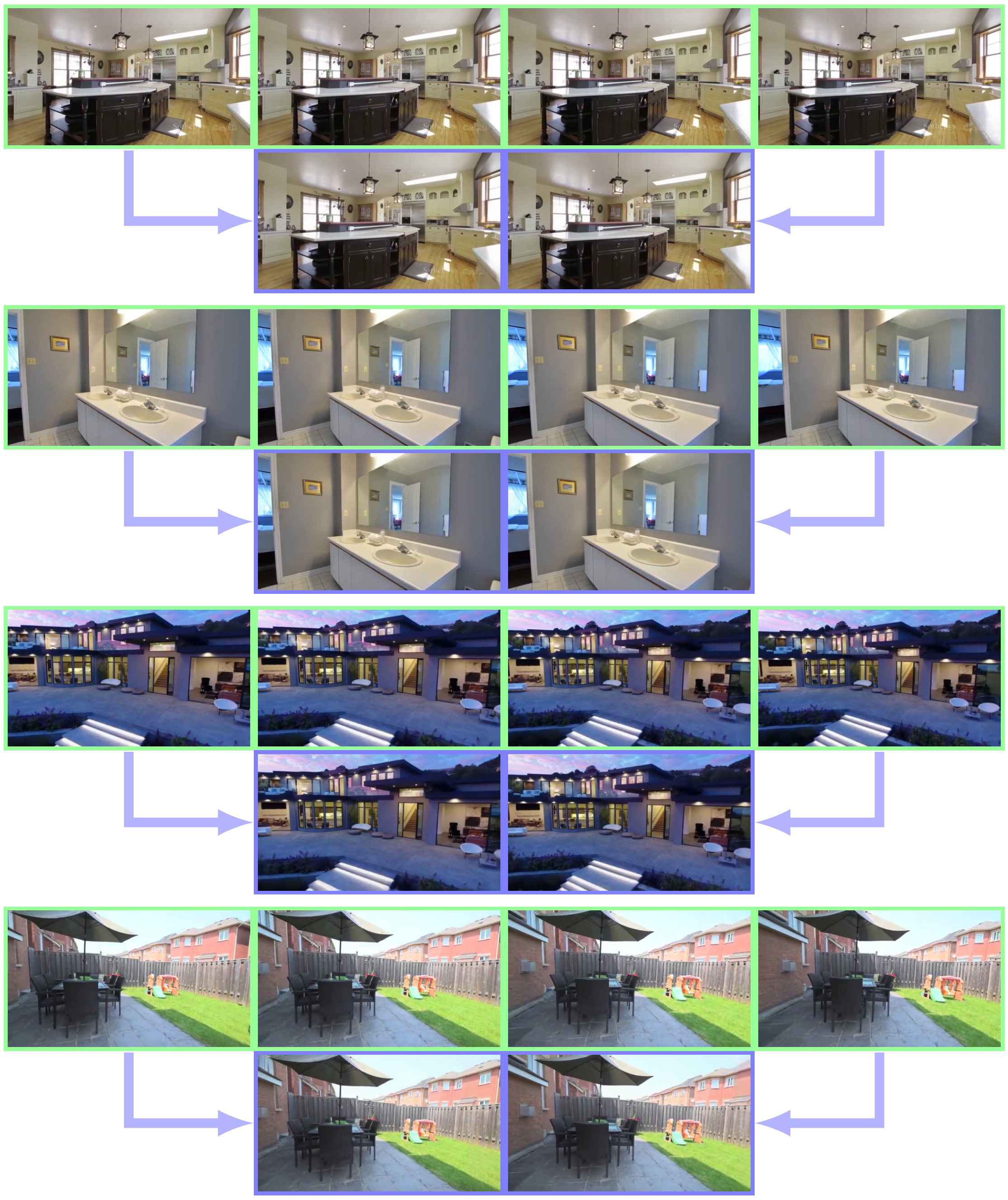


Figure 5.15: Interpolation using our fast FaDIV-Full-17 network. In every block, the top row (green) shows the ground truth trajectory from RealEstate10k (test), while the bottom row (blue) presents the interpolated result corresponding to the ground truth camera poses.

6 Conclusion

We have presented two different methods to synthesize novel views. While our first method goes the conventional way of (i) depth estimation, (ii) rendering and (iii) refining, our second approach (FaDIV-Syn) decouples the pipeline from costly and time-consuming depth computations and contributes an approach that processes on PSVs directly.

We introduced a single view approach that builds upon SynSin [2] and further optimized the architectures and losses to achieve better performance in a more efficient way. We have shown a detailed evaluation and outperform SynSin with a pipeline that has fewer processing stages. We are therefore capable of real-time processing images with a size of 512×512. Further research in this direction could focus on improving the scale estimation of the disparity network, which is the most challenging task in the pipeline. Possible approaches could be to make use of sparse point clouds, as proposed in [52] or a second disparity network decoder that is trained to estimate a scale more precisely.

For multiple source views we introduced FaDIV-Syn, a fast depth-independent novel view synthesis method. We demonstrated state-of-the-art performance in extrapolation on the RealEstate10k dataset. Our model trained with 288p generalizes well to higher resolutions and even outperforms state-of-the-art results on 576p. Furthermore, our method is real-time-capable with 85-300 fps depending on output resolution. We think direct usage of the PSV for RGB view synthesis is a promising approach especially for real-time applications and further research in this direction is warranted. This research could investigate self supervised techniques, as proposed in [49], which allows to decouple the network from synthesizing only views inside strict camera trajectories. Furthermore, it could be profitable to warp planes with learned features instead of RGB information only, as Wiles et al. [2] do. Instead of assuming fixed depths of the planes, one could also attempt to learn them self-supervised.

A direct comparison of both approaches is problematic, as FaDIV-Syn has more favourable prerequisites. However, if we extend our single view approach to several input images, indeed one could predict better depth, but the pipeline would still not be real-time capable on 540p unlike FaDIV-Syn. We believe that with FaDIV-Syn we have made an important contribution to the novel view synthesis problem.

List of Figures

List of Tables

Bibliography

[1] Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. “Stereo Magnification: Learning View Synthesis using Multiplane Images.” In: *ACM Transactions on Graphics (TOG)*. 2018.

[2] Olivia Wiles, Georgia Gkioxari, Richard Szeliski, and Justin Johnson. “SynSin: End-to-end View Synthesis from a Single Image.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 7467–7477.

[3] Clément Godard, Oisin Mac Aodha, Michael Firman, and Gabriel J. Brostow. “Digging into Self-Supervised Monocular Depth Prediction.” In: *IEEE International Conference on Computer Vision (ICCV)*. 2019, pp. 3828–3838.

[4] Clément Godard, Oisin Mac Aodha, and Gabriel J. Brostow. “Unsupervised Monocular Depth Estimation with Left-Right Consistency.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 270–279.

[5] Ravi Garg, Vijay Kumar B.G., Gustavo Carneiro, and Ian Reid. “Unsupervised CNN for Single View Depth Estimation: Geometry to the Rescue.” In: *European Conference on Computer Vision (ECCV)*. 2016, pp. 740–756.

[6] Tinghui Zhou, Matthew Brown, Noah Snavely, and David G. Lowe. “Unsupervised Learning of Depth and Ego-Motion From Video.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 1851–1858.

[7] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. “Spatial Transformer Networks.” In: *arXiv preprint arXiv:1506.02025* (2015).

[8] Sunghoon Im, Hae-Gon Jeon, Stephen Lin, and In So Kweon. “DPSNet: End-to-end Deep Plane Sweep Stereo.” In: *International Conference on Learning Representations (ICLR)*. 2018.

[9] B. Ruf, Bastian Erdnüß, and Martin Weinmann. “Determining Plane-Sweep Sampling Points In Image Space using the Cross-Ratio for Image-Based Depth Estimation.” In: *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences (ISPRS)* XLII-2/W6 (2017), pp. 325–332.

[10] P. Huang, K. Matzen, J. Kopf, N. Ahuja, and J. Huang. “DeepMVS: Learning Multi-view Stereopsis.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 2821–2830.

[11] S. Cheng, Z. Xu, S. Zhu, Z. Li, L. E. Li, R. Ramamoorthi, and H. Su. “Deep Stereo Using Adaptive Thin Volume Representation With Uncertainty Awareness.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 2521–2531.

[12] P. P. Srinivasan, R. Tucker, J. T. Barron, R. Ramamoorthi, R. Ng, and N. Snavely. “Pushing the Boundaries of View Extrapolation With Multiplane Images.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 175–184.

[13] Nima Khademi Kalantari, Ting-Chun Wang, and Ravi Ramamoorthi. “Learning-Based View Synthesis for Light Field Cameras.” In: *ACM Transactions on Graphics (TOG)* 35.6 (2016), pp. 1–10.

[14] Robert T Collins. “A Space-Sweep Approach to True Multi-Image Matching.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1996, pp. 358–363.

[15] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. “Language Modeling with Gated Convolutional Networks.” In: *International Conference on Machine Learning (ICML)*. 2017, pp. 933–941.

[16] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S. Huang. “Free-Form Image Inpainting With Gated Convolution.” In: *IEEE International Conference on Computer Vision (ICCV)*. 2019, pp. 4471–4480.

[17] Xudong Lin, Lin Ma, Wei Liu, and Shih-Fu Chang. “Context-Gated Convolution.” In: *arXiv preprint arXiv:1910.05577* (2019).

[18] Peter Sturm. “Pinhole Camera Model.” In: *Computer Vision: A Reference Guide*. Ed. by Katsushi Ikeuchi. Springer, 2014, pp. 610–613.

[19] Kenichi Kanatani, Naoya Ohta, and Yasushi Kanazawa. “Optimal Homography Computation with a Reliability Measure.” In: *IEICE Transactions on Information and Systems (TOIS)* 83.7 (2000), pp. 1369–1374.

[20] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. “Accelerating 3D Deep Learning with PyTorch3D.” In: *arXiv preprint arXiv:2007.08501* (2020).

[21] Meng-Li Shih, Shih-Yang Su, Johannes Kopf, and Jia-Bin Huang. “3D Photography using Context-aware Layered Depth Inpainting.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 8028–8038.

[22] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition.” In: *arXiv preprint arXiv:1409.1556* (2014).

[23] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. “ImageNet: a Large-Scale Hierarchical Image Database.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2009, pp. 248–255.

[24] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. “Image Quality Assessment: From Error Visibility to Structural Similarity.” In: *IEEE Transactions on Image Processing (TIP)* 13.4 (2004), pp. 600–612.

[25] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. “The Unreasonable Effectiveness of Deep Features as a Perceptual Metric.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 586–595.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. “ImageNet Classification with Deep Convolutional Neural Networks.” In: *International Conference on Neural Information Processing Systems (NeurIPS)* 25 (2012).

[27] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. “Generative Adversarial Nets.” In: *International Conference on Neural Information Processing Systems (NeurIPS)*. 2014, pp. 2672–2680.

[28] Aliaksandr Siarohin, Stéphane Lathuilière, Sergey Tulyakov, Elisa Ricci, and Nicu Sebe. “First Order Motion Model for Image Animation.” In: *International Conference on Neural Information Processing Systems (NeurIPS)* 32 (2019), pp. 7137–7147.

[29] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. “Image-to-Image Translation with Conditional Adversarial Networks.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 1125–1134.

[30] Martin Arjovsky, Soumith Chintala, and Léon Bottou. “Wasserstein GAN.” In: *arXiv preprint arXiv:1701.07875* (2017).

[31] Xudong Mao, Qing Li, Haoran Xie, Raymond Lau, Wang Zhen, and Stephen Smolley. “Least Squares Generative Adversarial Networks.” In: *IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 2813–2821.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778.

[33] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.” In: *International Conference on Machine Learning (ICML)*. 2015, pp. 448–456.

[34] Eric Penner and Li Zhang. “Soft 3D Reconstruction for View Synthesis.” In: *ACM Transactions on Graphics (TOG)* 36.6 (2017).

[35] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. “Deep Blending for Free-viewpoint Image-based Rendering.” In: *ACM Transactions on Graphics (TOG)* 37.6 (2018), pp. 1–15.

[36] Peter Hedman, Tobias Ritschel, George Drettakis, and Gabriel Brostow. “Scalable inside-out image-based rendering.” In: *ACM Transactions on Graphics (TOG)* 35.6 (2016), pp. 1–11.

[37] R. O. Cayon, A. Djelouah, and G. Drettakis. “A Bayesian Approach for Selective Image-Based Rendering Using Superpixels.” In: *International Conference on 3D Vision (3DV).* 2015, pp. 469–477.

[38] Qianqian Wang, Zhicheng Wang, Kyle Genova, Pratul Srinivasan, Howard Zhou, Jonathan T. Barron, Ricardo Martin-Brualla, Noah Snavely, and Thomas Funkhouser. “IBRNet: Learning Multi-View Image-Based Rendering.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2021).

[39] Phong Nguyen, Animesh Karnewar, Lam Huynh, Esa Rahtu, Jiri Matas, and Janne Heikkila. “RGBD-Net: Predicting color and depth images for novel views synthesis.” In: *arXiv preprint arXiv:2011.14398* (2020).

[40] Junyuan Xie, Ross Girshick, and Ali Farhadi. “Deep3D: Fully Automatic 2D-to-3D Video Conversion with Deep Convolutional Neural Networks.” In: *European Conference on Computer Vision (ECCV).* 2016, pp. 842–857.

[41] Justus Thies, Michael Zollhöfer, Christian Theobalt, Marc Stamminger, and Matthias Nießner. “Image-guided Neural Object Rendering.” In: *International Conference on Learning Representations (ICLR).* 2020.

[42] Gernot Riegler and Vladlen Koltun. “Free view synthesis.” In: *European Conference on Computer Vision (ECCV).* 2020, pp. 623–640.

[43] Gernot Riegler and Vladlen Koltun. “Stable View Synthesis.” In: *arXiv preprint arXiv:2011.07233* (2020).

[44] Kara-Ali Aliev, Artem Sevastopolsky, Maria Kolos, Dmitry Ulyanov, and Victor Lempitsky. “Neural Point-Based Graphics.” In: *arXiv preprint arXiv:-1906.08240v3* (2020).

[45] Inchang Choi, Orazio Gallo, Alejandro Troccoli, Min H Kim, and Jan Kautz. “Extreme View Synthesis.” In: *IEEE International Conference on Computer Vision (CVPR).* 2019, pp. 7781–7790.

[46] Pratul P. Srinivasan, Tongzhou Wang, Ashwin Sreelal, Ravi Ramamoorthi, and Ren Ng. “Learning to Synthesize a 4D RGBD Light Field From a Single Image.” In: *IEEE International Conference on Computer Vision (ICCV).* 2017, pp. 2243–2251.

[47] Xu Chen, Jie Song, and Otmar Hilliges. “Monocular Neural Image Based Rendering with Continuous View Control.” In: *IEEE International Conference on Computer Vision (ICCV).* 2019, pp. 4090–4100.

[48] K. Olszewski, S. Tulyakov, O. Woodford, H. Li, and L. Luo. “Transformable Bottleneck Networks.” In: *IEEE International Conference on Computer Vision (ICCV).* 2019, pp. 7647–7656.

[49] Nicolai Hani, Selim Engin, Jun-Jee Chao, and Volkan Isler. “Continuous Object Representation Networks: Novel View Synthesis without Target View Supervision.” In: *International Conference on Neural Information Processing Systems (NeurIPS)* 33 (2020).

[50] C. Wang, J. M. Buenaposada, R. Zhu, and S. Lucey. “Learning Depth from Monocular Videos Using Direct Methods.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2018, pp. 2022–2030.

[51] David Eigen, Christian Puhrsch, and Rob Fergus. “Depth Map Prediction from a Single Image Using a Multi-Scale Deep Network.” In: *International Conference on Neural Information Processing Systems (NeurIPS).* 2014, pp. 2366–2374.

[52] Richard Tucker and Noah Snavely. “Single-view View Synthesis with Multiplane Images.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2020, pp. 551–560.

[53] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. “High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2018, pp. 8798–8807.

[54] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. “Semantic Image Synthesis with Spatially-Adaptive Normalization.” In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2019, pp. 2337–2346.

[55] Diogo C. Luvizon, Gustavo Sutter P. Carvalho, Andreza A. dos Santos, Jhonatas S. Conceicao, Jose L. Flores-Campana, Luis G. L. Decker, Marcos R. Souza, Helio Pedrini, Antonio Joia, and Otavio A. B. Penatti. “Adaptive Multiplane Image Generation from a Single Internet Picture.” In: *arXiv preprint arXiv:2011.13317* (2020).

[56] Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz-Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. "Local Light Field Fusion: Practical View Synthesis with Prescriptive Sampling Guidelines." In: *ACM Transactions on Graphics (TOG)* 38.4 (2019), pp. 1–14.

[57] Benjamin Attal, Selena Ling, Aaron Gokaslan, Christian Richardt, and James Tompkin. "MatryODShka: Real-time 6DoF Video View Synthesis using Multi-Sphere Images." In: *European Conference on Computer Vision (ECCV).* 2020, pp. 441–459.

[58] Jonathan Shade, Steven Gortler, Li-wei He, and Richard Szeliski. "Layered Depth Images." In: *Conference on Computer Graphics and Interactive Techniques (SIGGRAPH).* 1998, pp. 231–242.

[59] Noah Snavely, Richard Tucker, and Shubham Tulsiani. "Layer-structured 3D Scene Inference via View Synthesis." In: *European Conference on Computer Vision (ECCV).* 2018, pp. 302–317.

[60] C. Zitnick, Sing Bing Kang, Matt Uyttendaele, Simon Winder, and Richard Szeliski. "High-Quality Video View Interpolation Using a Layered Representation." In: *ACM Transactions on Graphics (TOG)* 23.3 (2004), pp. 600–608.

[61] 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 (ECCV).* 2020, pp. 405–421.

[62] Kai Zhang, Gernot Riegler, Noah Snavely, and Vladlen Koltun. "NeRF++: Analyzing and Improving Neural Radiance Fields." In: *arXiv preprint arXiv:-2010.07492* (2020).

[63] Alex Yu, Vickie Ye, Matthew Tancik, and Angjoo Kanazawa. "pixelNeRF: Neural Radiance Fields from One or Few Images." In: *arXiv preprint arXiv:-2012.02190* (2020).

[64] Keunhong Park, Utkarsh Sinha, Jonathan T. Barron, Sofien Bouaziz, Dan Goldman, Steven Seitz, and Ricardo Martin-Brualla. "Deformable Neural Radiance Fields." In: *arXiv preprint arXiv:2011.12948* (2020).

[65] Albert Pumarola, Enric Corona, Gerard Pons-Moll, and Francesc Moreno-Noguer. "D-NeRF: Neural Radiance Fields for Dynamic Scenes." In: *arXiv preprint arXiv:2011.13961* (2020).

[66] Guy Gafni, Justus Thies, Michael Zollhöfer, and Matthias Nießner. "Dynamic Neural Radiance Fields for Monocular 4D Facial Avatar Reconstruction." In: *arXiv preprint arXiv:2012.03065* (2020).

[67] Edgar Tretschk, Ayush Tewari, Vladislav Golyanik, Michael Zollhöfer, Christoph Lassner, and Christian Theobalt. “Non-Rigid Neural Radiance Fields: Reconstruction and Novel View Synthesis of a Deforming Scene from Monocular Video.” In: *arXiv preprint arXiv:2012.12247* (2020).

[68] Raul Mur-Artal and Juan D Tardós. “ORB-SLAM2: An open-source SLAM system for monocular, stereo, and RGB-D cameras.” In: *IEEE Transactions on Robotics (T-RO)* 33.5 (2017), pp. 1255–1262.

[69] J. Xiao, A. Owens, and A. Torralba. “SUN3D: A Database of Big Spaces Reconstructed Using SfM and Object Labels.” In: *IEEE International Conference on Computer Vision (ICCV)*. 2013, pp. 1625–1632.

[70] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition.” In: *arXiv preprint arXiv:1512.03385* (2015).

[71] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-Net: Convolutional networks for biomedical image segmentation.” In: *International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. 2015, pp. 234–241.

[72] Pierre Soille. “Erosion and Dilation.” In: *Morphological Image Analysis: Principles and Applications*. Springer, 2004, pp. 63–103.

[73] Tong Tong, Gen Li, Xiejie Liu, and Qinquan Gao. “Image Super-Resolution Using Dense Skip Connections.” In: *IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 4799–4807.

[74] NVIDIA. *TensorRT*. `https://developer.nvidia.com/tensorrt`.

[75] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. “ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression.” In: *IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 5058–5066.