Supplementary Materials: SparseFusion: Distilling View-conditioned Diffusion for 3D Reconstruction

We provide a more detailed overview of denoising diffusion probabilistic models. We provide implementation details for each component our approach, epipolar feature transformer (EFT), View-conditioned Diffusion Model (VLDM), and diffusion distillation. We include 360 visualizations on our project page: https://sparsefusion.github.io/.

1. Background: Denoising Diffusion

Our method adopts and optimizes through denoising diffusion models [2], and here we give a brief summary of the key formulations used.

Training Objective. One can learn denoising diffusion models by optimizing a variational lower bound on the log-likelihood of the observed data. Conveniently, this reduces to a training framework where one adds (time-dependent) noise to a data point $\boldsymbol{x}_0$, and then trains a network $\boldsymbol{\epsilon}_\phi$ to predict this noise given the noisy data point $\boldsymbol{x}_t$.

$$\begin{aligned} \mathcal{L}_{DM} &= \mathbb{E}_{\boldsymbol{x}_0,\boldsymbol{\epsilon},t}\left[w_t\,||\boldsymbol{\epsilon}-\boldsymbol{\epsilon}_\phi(\boldsymbol{x}_t,t)||^2\right] \\ \text{where } \boldsymbol{x}_t &= \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0+\sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon};\;\; \boldsymbol{\epsilon}\sim\mathcal{N}(0,1) \end{aligned} \tag{1}$$

Here, $\bar{\alpha}_t$ is a scheduling hyper-parameter, and the weights w_t depend on this learning schedule, but are often set to 1 to simplify the objective.

Interpretation as Reconstruction Error. The above noise prediction objective, which represents a bound on the log likelihood, can also be viewed as a reconstruction error. Concretely, given a noisy $\boldsymbol{x}_t$, the network prediction $\boldsymbol{\epsilon}_\phi(\boldsymbol{x}_t,t)$ can be interpreted as yielding a reconstruction for the original input, where the learning objective can be rewritten as a reconstruction error:

$$\hat{\boldsymbol{x}}_{0,t} = \frac{\boldsymbol{x}_t-\sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_\phi(\boldsymbol{x}_t,t)}{\sqrt{\bar{\alpha}_t}} \tag{2}$$

$$\mathcal{L}_{DM} = \mathbb{E}_{\boldsymbol{x}_0,\boldsymbol{\epsilon},t}\left[w_t'\,||\hat{\boldsymbol{x}}_{0,t}-\boldsymbol{x}_0||^2\right] \tag{3}$$

While the above summary focused on unconditional diffusion models, they can be easily extended to infer conditional distributions $p(\boldsymbol{x}|\boldsymbol{y})$ by additionally using $\boldsymbol{y}$ as an input for the noise prediction network $\boldsymbol{\epsilon}_\phi$.

Forward Process. The forward diffusion process, which incrementally adds noise to a real image $\boldsymbol{x}_0$ until the image becomes Gaussian noise $\boldsymbol{x}_T$, is defined in Eq. 4. Forward variance β is usually defined by a fixed schedule.

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) = \mathcal{N}(\sqrt{1-\beta_t}\boldsymbol{x}_{t-1},\beta_t\boldsymbol{I}) \tag{4}$$

Reverse Process. The reverse diffusion process reverses the noise added in the forward process, effectively denoising a noisy image. When we generate a sample from a diffusion model, we apply the reverse process T times from $t=T$ to $t=1$. The reverse process is defined in Eq. 5, where posterior mean $\mu_\phi(\boldsymbol{x}_t,t)$ is predicted from a network and posterior variance σ^2 follows a fixed schedule (though other works such as [7] also learn σ^2 with a network).

$$p(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) = \mathcal{N}(\mu_\phi(\boldsymbol{x}_t,t),\sigma^2\boldsymbol{I}) \tag{5}$$

Posterior Mean. Prior works [2, 7] have found that parameterizing the neural network to predict $\boldsymbol{\epsilon}$ instead of $\boldsymbol{x}_{t-1}$ or $\boldsymbol{x}_0$ works better in practice. We write posterior mean in terms of $\boldsymbol{\epsilon}$ in Eq. 6 where $\alpha_t = 1-\beta_t$ and $\bar{\alpha}_t = \Pi_{s=1}^t\alpha_s$.

$$\mu_\phi(\boldsymbol{x}_t,t) = \frac{1}{\sqrt{\alpha_t}}(\boldsymbol{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\phi(\boldsymbol{x}_t,t)) \tag{6}$$

As mentioned in the main text, this parametrization leads to a training framework where one adds (time-dependent) noise to a data point $\boldsymbol{x}_0$, and then trains the network $\boldsymbol{\epsilon}_\phi$ to predict this noise given the noisy data point $\boldsymbol{x}_t$.

In this work, we use conditional diffusion models to infer distributions of the form $p(\boldsymbol{x}|\boldsymbol{y})$ by additionally using $\boldsymbol{y}$ as an input for the noise prediction network $\boldsymbol{\epsilon}_\phi(\boldsymbol{x},\boldsymbol{y},t)$.

2. Implementation Details

We provide detailed implementation and training details for all components of SparseFusion.

2.1. Epipolar Feature Transformer

Overview. Epipolar feature transformer is a feed-forward network that first gathers features along the epipolar lines of input images before aggregating them through a series

of transformers. EFT is inspired by the GPNR approach by Suhail *et al*. [8], but we modify the feature extractor backbone to better suit the sparse-view setup and additionally use epipolar features for conditional diffusion. We describe our implementation below.

Notation: Let g_ψ be the RGB branch and h_ψ be the feature branch.

Inputs: $C \equiv (\boldsymbol{x}_m, \boldsymbol{\pi}_m)$, a set of input images with known camera poses and a query pose $\boldsymbol{\pi}$ – note that the poses are w.r.t. an arbitrary world-coordinate system and we only use their relative configuration.

Outputs: an RGB image $\boldsymbol{x}$ and a feature grid $\boldsymbol{y}$ corresponding to the query viewpoint $\boldsymbol{\pi}$.

Feature Extractor Backbone. Given input views $C \equiv (\boldsymbol{x}_m, \boldsymbol{\pi}_m)$ where $\boldsymbol{x}_m$ is the m^{th} masked (black background) input image of shape (256, 256, 3). We use ResNet18 [1] as our backbone to extract pixel-aligned features by concatenating intermediate features from the first 4 layer groups of ResNet18, using bilinear upsampling to ensure all features are 128 by 128. For each image $\boldsymbol{x}_m$, we arrive at a feature grid of shape (128, 128, 512).

Epipolar Points Projector. Given a query camera $\boldsymbol{\pi}$, each pixel in its image plane corresponds to some ray. Our Epipolar Transformer seeks to infer per-pixel colors or features, and does so by processing each ray using the multi-view projections of points along it. For each ray $\boldsymbol{r}$ (parameterized by its origin and direction), we project 20 points along the ray direction with depth values linearly spaced between z_near and z_far. We set z_near to $s - 5$ and z_far to $s + 5$ where s is the average distance from scene cameras to origin computed per scene. The 20 points, with shape (20, 3), are then projected into the screen space of each of the m input cameras, giving us epipolar points with shape (M, 20, 2). We use bilinear sampling to sample image features at the epipolar points, giving us combined epipolar features of shape (M, 20, 512) per ray. This becomes the input to our epipolar feature transformer.

Epipolar Feature Transformer. EFT aggregates the epipolar features from a single ray with a series of three transformers to predict an RGB pixel color and a 256-dimension feature. We visualize the EFT in Figure 1. We show details of the transformers in Table 1. All transformer encoders have hidden and output dimensions of 256. Both the depth aggregator and view aggregator transformers are followed by a weighted average operation, where the output features from the transformers are multiplied by a weight, which sums to 1 along the sequence length dimension. The relative weights are predicted by a linear layer before passing through softmax. This effectively performs weighted averaging along the sequence dimension.

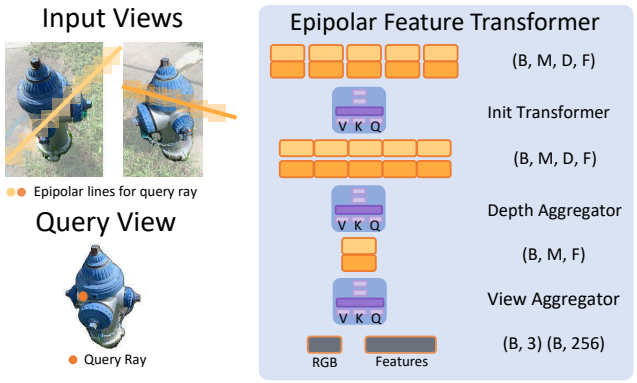

Figure 1. **Epipolar Feature Transformer** We show a diagram of EFT. This module processes each query ray independently, using a transformer to aggregate the projected features across views and across possible depths. For each ray, the output is a predicted RGB color (used as a baseline prediction method), and a pixel-aligned feature (used as conditioning in the diffusion model).

Table 1. **EFT Configuration.** We use default PyTorch hyperparameters for each layer. B is number of rays. M is the number of input views. D is the number of epipolar feature samples along the ray. D is 20.

Transformer	Layers	Sequence Dims / Dim	Output Shape
Init Transformer	Transformer Encoder x4	M	(B, M, D, 256)
Depth Aggregator	Transformer Encoder x4	D	(B, M, D, 256)
	Linear + Softmax	D	(B, M, D, 1)
	Weighted Average		(B, M, 256)
View Aggregator	Transformer Encoder x4	M	(B, M, 256)
	Linear + Softmax	M	(B, M, 1)
	Weighted Average		(B, 256)
Color Branch	Linear		(B, 3)

The inputs to the transformer are the sampled features concatenated with additional ray and depth encodings. Given a point along the query ray $\boldsymbol{r}_q$ at depth d, we denote by $\boldsymbol{p}_{md}$ its projection in the m^{th} context view. In addition to the pixel-aligned feature $\boldsymbol{f}_{md}$ (described in previous paragraph), we also concatenate encodings of the query ray $\boldsymbol{r}_q$, the depth $\boldsymbol{d}$, and the ray $\boldsymbol{r}_{md}$ connecting the m^{th} camera center to the 3D point. We use plucker coordinates to represent each ray, and compute harmonic embeddings for each to $(\boldsymbol{r}_q, \boldsymbol{r}_{md}, \boldsymbol{d})$ (using 6 harmonic functions) before concatenating them with $\boldsymbol{f}_{md}$ to form the input tokens to the transformer.

Training Procedure. We can train the color branch of EFT as a standalone novel view synthesis baseline. In our work, EFT is jointly trained with VLDM. Please see supplementary Section 2.2 for details.

2.2. View-conditioned Diffusion Model

Overview. View-conditioned diffusion model is a latent diffusion model that conditions on a pixel-aligned feature grid $\boldsymbol{y}$.

Notation: Let ϵ_ϕ be the denoising UNet, $\mathcal{E}$ be the VAE encoder, and $\mathcal{D}$ be the VAE decoder.

VAE. We use the VAE from Stable Diffusion [5]. We use the provided v1-3 weights and keep the VAE frozen for all experiments. We use (256, 256, 3) RGB images as input, and the VAE encodes them into latents of shape (32, 32, 4). We refer readers to [5] for more details.

Denoising UNet. Our 400M parameter UNet roughly follows [6]. We construct our UNet using code from [10] with the parameters in Table 2.

Table 2. **UNet Parameters.** We provide parameters for our UNet.

Parameter	Value
channels	4
dim	256
dim_mults	(1,2,4,4)
num_resnet_blocks	(2,2,2,2)
layer_attns	(False, False, False, True)
cond_images_channels	256

The UNet comprises of 4 down-sampling blocks, a middle block, and 4 up-sampling blocks. We show the input and output shape for the modules of the UNet in Table 3. We refer readers to [10] for UNet details. We disable all text conditioning and cross attention mechanisms; instead, we concatenate EFT features, $\boldsymbol{y}$, with image latents, $\boldsymbol{z}_t$. These EFT features are computed for the of 32×32 rays corresponding to the patch centers.

Training Procedure. We train with batch size of 2, randomly chosen number of input views between 2-5, and learning rate of 5e-5 using Adam optimizer with default hyperparameters for 100K steps. We optimize both the UNet weights and also the EFT weights. We optimize the UNet and feature branch of EFT with the simplified variational lower bound [2]. We optimize the color branch of EFT with pixel-wise reconstruction loss.

2.3. Diffusion Distillation

Overview. We optimize a 3D neural scene representation, Instant NGP [4, 9], with our VLDM.

Notation: Let f_θ be the volumetric Instant NGP renderer, $p_\phi(\boldsymbol{z}_{0:\mathcal{T}}|\boldsymbol{\pi}, C)$ be the multi-step denoising process that estimates $\hat{\boldsymbol{z}}_0$. Let Π be an instance-specific camera distribution.

Table 3. **UNet Blocks.** We outline the modules in our denoising UNet.

Modules	Block	Output Shape
Input		(B, 260, 32, 32)
Init. Conv	InitBlock	(B, 256, 32, 32)
Down 1	DownBlock	(B, 256, 16, 16)
Down 2	DownBlock	(B, 512, 8, 8)
Down 3	DownBlock	(B, 1024, 4, 4)
Down 4	DownBlock Self-attention	(B, 1024, 4, 4) (B, 1024, 4, 4)
Middle	MiddleBlock	(B, 1024, 4, 4)
Up 1	UpBlock Self-attention	(B, 1024, 8, 8) (B, 1024, 8, 8)
Up 2	UpBlock	(B, 512, 16, 16)
Up 3	UpBlock	(B, 256, 32, 32)
Up 4	UpBlock	(B, 256, 32, 32)
Final Conv.	Conv2D	(B, 4, 32, 32)

Instant NGP. We use the PyTorch Instant NGP implementation from [9]. We set scene bounds to 4 with desired hashgrid resolution of 8,192. We use a small 3 layer MLP with hidden dimension of 64 to predict RGB and density. We do not use view direction as input.

Camera Distribution. Given a set of input cameras $C_I \equiv (\boldsymbol{\pi}_m)$ and a query camera $\boldsymbol{\pi}_q$, we first find the look-at point P_{at} by finding the nearest point to all $m+1$ rays originating from camera centers. Then, we fit a circle O in 3D space with center being the mean of all camera centers. Let the normal of circle O be $\boldsymbol{n}$. To sample a camera, we first sample a point P_i on O and jitter the angle between $\overline{P_{at}P_i}$ and $\boldsymbol{n}$ by $\mathcal{N}(0, 0.17)$ radians to get jittered point $P_i^{'}$. We then construct a camera $\boldsymbol{\pi}$ with center $P_i^{'}$ looking at P_{at}.

Multi-step Diffusion Guidance. Given a rendered image $\boldsymbol{x}_0$, we encode it to obtain $\boldsymbol{z}_0$. Then, we uniformly sample $t \sim (0, T]$ and construct a noisy image latent $\boldsymbol{z}_t$. We perform multi-step denoising to obtain $\hat{\boldsymbol{z}}_0$ by iteratively sampling $\hat{\boldsymbol{z}}_{t_{k-1}} \sim p_\phi(\boldsymbol{z}_{t_{k-1}}|\hat{\boldsymbol{z}}_{t_k}, y)$ on an interval of time steps $\mathcal{T} = (t_1, ..., t_k, t)$ using a linear multi-step method [3]. We construct $\mathcal{T}$ by linearly spacing $k+1$ time steps between $(0, t]$. We define k with a simple scheduler:

$$k + 1 = \left\{ \begin{array}{ll} \frac{100t}{T}, & \text{if } t \leq \frac{T}{2} \\ 50, & \text{if } t > \frac{T}{2} \end{array} \right\} \tag{7}$$

Finally, given $\hat{\boldsymbol{z}}_0$, we get the predicted image $\hat{\boldsymbol{x}}_0 = \mathcal{D}(\hat{\boldsymbol{z}}_0)$. We do not compute gradients through multi-step diffusion and treat $\hat{\boldsymbol{x}}_0$ as a detached tensor.

Distillation Details. We perform 3,000 steps of distillation, optimizing weights of the MLP θ with Adam optimizer and learning rate 5e-4. During each step of diffusion distillation, we sample $\boldsymbol{\pi} \sim \Pi$ and render an image $\boldsymbol{x}_0 = f_\theta(\boldsymbol{\pi})$. For the first 1,000 steps, we compute rendering loss between $f_\theta(\boldsymbol{\pi})$ and $g_\psi(\boldsymbol{\pi}|C)$. During the remaining steps, we compute loss between $f_\theta(\boldsymbol{\pi})$ and $\hat{\boldsymbol{x}}_0$ and use weighting $w_t = 1 - \bar{\alpha}_t$. To avoid out-of-memory error, we render images at reduced resolution (128, 128) and apply bilinear up-sampling before performing multi-step diffusion. In addition, we compute rendering loss between $f_\theta(\boldsymbol{\pi}_m)$ and $\boldsymbol{x}_m$ on all m input images. Optimizing a single scene takes roughly 1 hour on an A5000 GPU.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 2

[2] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *NeurIPS*, 2020. 1, 3

[3] Luping Liu, Yi Ren, Zhijie Lin, and Zhou Zhao. Pseudo numerical methods for diffusion models on manifolds. In *ICLR*, 2022. 3

[4] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 2022. 3

[5] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *CVPR*, 2022. 3

[6] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, Burcu Karagol Ayan, S Sara Mahdavi, Rapha Gontijo Lopes, et al. Photorealistic text-to-image diffusion models with deep language understanding. In *NeurIPS*, 2022. 3

[7] Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. In *ICLR*, 2021. 1

[8] Mohammed Suhail, Carlos Esteves, Leonid Sigal, and Ameesh Makadia. Generalizable patch-based neural rendering. In *ECCV*, 2022. 2

[9] Jiaxiang Tang. Torch-ngp: a pytorch implementation of instant-ngp, 2022. https://github.com/ashawkey/torch-ngp. 3

[10] Phil Wang. Implementation of imagen, google's text-to-image neural network, in pytorch, 2022. https://github.com/lucidrains/imagen-pytorch. 3