Viewset Diffusion: (0-)Image-Conditioned 3D Generative Models from 2D Data Supplementary Material

Stanislaw Szymanowicz Christian Rupprecht Andrea Vedaldi
Visual Geometry Group — University of Oxford
{stan,chrisr,vedaldi}@robots.ox.ac.uk

1. Additional results

Visit the project website https://szymanowiczs.github.io/viewset-diffusion.html for more visualisations of non-cherry-picked results of single-view 3D reconstruction and unconditional 3D generation across all classes.

2. Minecraft Dataset

We provide further details on the Minecraft dataset we contribute as part of this work, in particular: distribution of camera poses, distribution of character articulation poses, image backgrounds, validation images and visualisations of both the training set and the 'Ambiguous' test set.

Cameras. Cameras are placed at a fixed distance from the origin, are directed towards the origin and have identical focal lengths and principal points. Camera location is therefore parameterised by the azimuth and elevation angles that it forms with the world x-axis (the radius is fixed). Yaw and elevation are sampled randomly and independently for each camera: azimuth is distributed uniformly in $[0, 2\pi]$ and elevation is distributed uniformly in $[-\frac{\pi}{8}; \frac{\pi}{8}]$.

Articulation. Characters have a fixed torso location and orientation, facing in the direction of positive z-axis. Both arms, both legs and the head are randomly and independently articulated. Pitch of arms is distributed uniformly in $[-20°, 45°]$; roll is distributed uniformly in $[0°, 10°]$ for the left arm and $[-10°, 0°]$ for the right arm. Pitch of legs is distributed uniformly in $[-30°, 30°]$; roll is distributed uniformly in $[0°, 10°]$ for the left leg and $[-10°, 0°]$ for the right leg. Head pitch is distributed uniformly in $[-10°, 10°]$, head pith is distributed uniformly in $[-5°, 5°]$ and head yaw is distributed normally with mean $10°$ and standard deviation $10°$.

Backgrounds. Character skins have varied colours, so there is no single background colour on which all skins clearly stand out from the background. Therefore, for each example we choose a random background colour, with RBG distributed uniformly in $[0, 255]$. The training examples are available with the alpha channel, therefore allowing for sampling of random backgrounds in every training iteration. Testing examples are saved with a fixed, randomly chosen background channels so that metrics are consistently measured between different training runs. Alpha channel was not used in any way other than to apply randomly coloured backgrounds, *i.e*. we did not use masks to aid network training.

Validation viewpoints. Each training example consists of 3 images and camera poses. In addition to the training images, each example (*i.e*. each articulated character) is associated with a fourth image and camera pose which we do not use for training (*i.e*. our network does not have access to it). However, we can render the reconstructed character from the unseen viewpoint and we use the error in that viewpoint as the validation loss that approximates the quality of 3D shapes output by our method.

Samples. Fig. 1 shows random samples from the Minecraft training dataset: 3 images per character, rendered from different camera viewpoints. Fig. 2 shows samples of test examples in the 'Ambiguous' test set. Ambiguities can be due to occlusion, where one limb is occluded by the torso. Another type of ambiguity is projective ambiguity, *i.e*. from a frontal image it can be ambiguous if a leg is in front of the body or behind the body. Finally, there are ambiguities due to symmetry – from a side image it can be ambiguous if the right leg is in front of the body and the left leg is behind, or if it is the other way around. Samples in the 'Ambiguous' test set exhibit more ambiguity than a randomly chosen test set would. Our method shows the most improvement on the 'Ambiguous' test set when compared to baselines.

Reproducing. We release the dataset in the form of code to run to exactly render the Minecraft dataset used for training, validation and testing.

Figure 1: **Minecraft dataset – training samples.** We show 24 random examples from the dataset. Each training example consists of 3 images (shown) and associated camera poses.

3. Data

3.1. CO3D Preprocessing

The data normalisation protocol for CO3D [5] objects aims to make objects approximately the same scale, place them in the centre of the voxel grid and align them vertically with the 'world' vertical direction.

1. **Translation normalization.** We find the centre of mass of the point cloud $\bar{x}$, shift the point cloud $\{x^{(i)}\}$ so that its new centre of mass is at the world centre: $\{x^{(i)}\}' = \{x^{(i)} - \bar{x}\}$ and move the cameras accordingly: $T'_{w2c} = \bar{x}R_{w2c} + T_{w2c}$.

2. **Rotation normalisation.** We estimate the world 'up' direction by leveraging photographer's bias, *i.e.* assuming that photos are taken with approximately zero yaw. Under this assumption, the camera x-vector is approximately perpendicular to the world direction. We form a matrix by stacking x-vectors of all cameras in a sequence (normalized to 0 mean) and run Singular Value Decomposition (SVD).

$$U\Sigma V^T = SVD([1,0,0]R_{w2c}^T)$$

$$\hat{y} = V[0,0,1]^T$$

SVD is only defined up to a direction ambiguity, therefore we set the world 'up' vector and the camera vectors to point in the same direction, *i.e.* ensure that $\hat{y}_{\text{world}} \cdot y_{\text{cam}} > 0$ and flip $\hat{y}$ if needed. We use the the first camera for y_{cam} but check that all cameras satisfy $y_{\text{cam1}} \cdot y_{\text{cam2}} > 0$ and exclude the sequence from training if that is not the case. We also verify that that our assumption of the photographer's bias is valid in a sequence by checking that $\sigma_1^2/\sigma_2^2 < \sigma_2^2/\sigma_3^2$ and exclude a sequence if that is not the case. We find that most sequences pass these checks successfully therefore confirming our intuition about photographer's bias. A visualisation of rotation normalisation is shown in Fig. 3.

3. **Scale normalization.** To normalise scale we first shift the point cloud and cameras so that world centre is halfway between the top and bottom of the point cloud $\{x^{(i)}\}'' = \{x^{(i)}\}' - (y_{max} - y_{min})$. This has effectively the same purpose as translation normalisation, but most videos are taken from above an object, so the point clouds are denser at the top of objects, meaning the point cloud mean $\bar{x}$ is nor an accurate object centre in the vertical direction. Finally, we want the scale of objects to be normalised across sequences, so we normalise the scene by the maximum absolute value of any point coordinate. The scaling factor s for a voxel grid with side length d is

$$s = \frac{d \times 0.95}{2 \times \max_i ||x^{(i)}||_\infty}.$$

All point coordinates $x^{(i)}$ and camera locations are scaled by factor s.

We verify that after the normalisation the distances of the

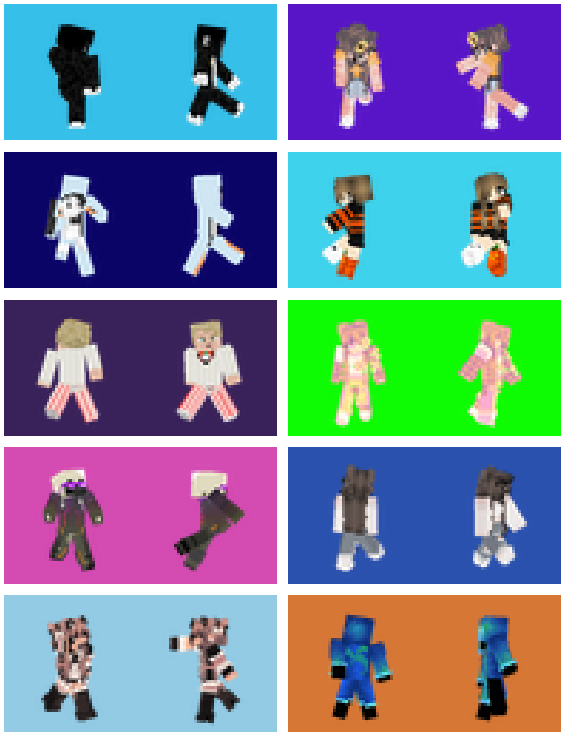

Figure 2: **Minecraft 'Ambiguous' test set.** We show 10 random examples from the 'Ambiguous' test set. In each example, left image is the input conditioning and right image is the ground truth testing view.

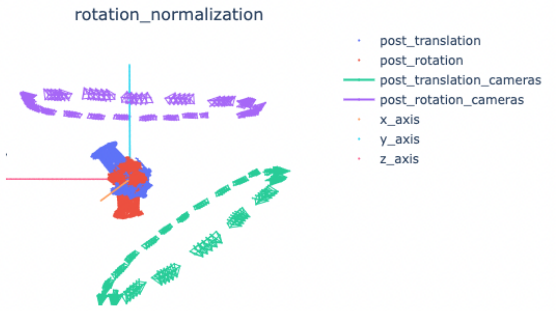

Figure 3: Rotation normalization. Red point cloud and purple cameras correspond to the translation and rotation aligned object. Blue points and green cameras correspond to the object after translation normalization, before rotation normalization.

cameras are in sensible locations. In particular, for a volume of side length 1.2 (which we use) we filter out sequences that have cameras very close to the centre, *i.e*. with camera translation magnitude $||T||_2 < 0.85$. These sequences correspond to sequences with poor point cloud quality where some cameras were incorrectly estimated to be very close to the surface of the objects We also filter out sequences with cameras very far from the centre, *i.e*. with camera translation magnitude $||T||_2 > 6.5$. Such sequences correspond to a failures in foreground / background segmentation which in turn lead to background being included in the point cloud. As a result, scaling using maximum point location results in downscaling the scene too much.

Finally, some sequences in CO3D correspond to cameras being moved in front of a screen displaying objects instead of actual objects. We filter out such sequences by imposing a minimum on standard deviation in depth values. In total, filtering removes 99 sequences for the Hydrant class, 382 sequences for the Plant class, 293 sequences for the Vase class and 473 sequences for the Teddy bear class. We do not remove any testing sequences.

We rescale all images to 128×128 resolution. As the focal lengths and principal points are provided in Normalised Device Coordinates, we do not need to rescale them when scaling the images or rescaling the world size.

3.2. Pose encoding

We encode the camera pose information in the tensor we pass to the network, similarly to 3DiM [8]. Each pixel RGB value is appended with an embedding of length 6, holding the location of the camera origin in world coordinates (length 3) and the normalised ray direction of the ray from the camera origin through the pixel (also length 3). The pose encoding is appended along the channel dimension to the input RGB images.

4. Baselines

4.1. PixelNeRF

In PixelNeRF [10] we use learning rate of $1e-5$ and use a batch size of 4 for CO3D data (corresponding to 132 images, as each batch contains 32 images of the same object) and 32 for Minecraft data (corresponding to 96 images as each batch contains 3 images of one object). We use Softplus activation and tune the beta parameter that prevents collapse into 0 density region - 1.0 for coarse network and 3.0 for fine network. We train for single-view reconstruction for $100k$ iterations for the Minens dataset and $600k$ iterations for CO3D datasets. We use 64 coarse samples and 64 fine samples at training time in the NeRF network for Minens dataset and 128 coarse samples and 64 fine samples for CO3D data. We use the same normalised CO3D data for PixelNeRF and for our method. For evaluation on ShapeNet-SRN we use pretrained PixelNeRF models with official evaluation code and metrics reported in the PixelNeRF paper [10].

4.2. RenderDiffusion

See Table 1 for a summary of the baseline architectures, using notation from Sec.3.5 in the main paper.

RenderDiffusion re-implementation (RD). We re-implemented RenderDiffusion with the publicly available information and we include the details and results here. As in the original paper, we modified the U-Net [2, 6] commonly used in diffusion models so that its output has $3n_f$ channels and reshaped it to form a triplane. We used $n_f = 32$ channels and a two-layer rendering MLP, with 32 hidden units and an intermediate Softplus activation function. Training was done for the same number of iterations as in our method with Adam [4] optimiser and the same hyperparameters as in our method. The noise schedule used was the same as in our method. We will release code for RenderDiffusion re-implementation together with our code.

RenderDiffusion++ (RD++). Our initial experiments with the architecture that RenderDiffusion uses indicate that the textures that RD outputs are low frequency and lack high-frequency details present in the conditioning images, likely due to the absence of local conditioning. Another issue we noticed was that the shapes output by RenderDiffusion are not necessarily plausible, *e.g.* a reconstruction of a Minecraft character with three legs, possibly because the supervision in RenderDiffusion is single view. While the reconstructions shown in the figures of RenderDiffusion have plausible shapes, they are only demonstrated for simple shapes from ShapeNet and CLEVR datasets with little ambiguity (*i.e.* a single image is enough to reconstruct the 3D shape, *e.g.* with symmetry constraints). Our Minecraft dataset exhibits more ambiguity (*e.g.* due to articulation) and thus the reconstructions output by RenderDiffusion are an average between the different plausible shapes. Thus, for fair comparison, we also compare to RD++: RenderDiffusion, but with our architecture and with multi-view supervision. We train a network that receives a single image of an object, with added Gaussian noise, and is tasked with predicting a reconstruction of the clean object.

5. Technical details

5.1. Optimization.

We optimise the parameters of our network with Adam [3] optimizer and learning rate 2×10^{-5}. We use batch size 16 and optimise all diffusion networks for 100k (ShapeNet) / 200k (Minecraft, Hydrant, Teddybear) / 256k (Vase) / 280k (Vase) iterations and the networks without diffusion for 40k iterations. Timestep-dependent weighting strategy $w(\sigma^{(i)})$ is the Min-SNR-5 strategy [1]: $w(\sigma^{(i)}) = \min\{\text{SNR}^{(i)}, 5\}$. SNR is the Signal-to-Noise Ratio: $\text{SNR}^{(i)} = (1 - \sigma^{(i)2})/\sigma^{(i)2}$. Hyperparameter λ for penalising unseen view is $\lambda = 0.1$ in Minens and ShapeNet and $\lambda = 0.2$ in CO3D.

5.2. Diffusion.

We use a diffusion schedule with 1000 diffusion steps and a cosine noise schedule. Our networks are trained with "x_0" formulation. At inference, we use 250 steps of DDIM [7] sampling. Quantitative single-view reconstruction results cited in the main paper were obtained with $N - 1$ noisy views and 1 clean view in the viewset, where $N = 3$ for ShapeNet-SRN, $N = 4$ for Minens and $N = 5$ for CO3D. For generation we used a the same models and viewsets with the same size N, but without any clean views. Varying numbers of images in the viewset are due to varying complexity of data and varying amount of ambiguity.

5.3. Architecture

Our architecture takes in N input frames, each of which can be clean or noisy, and outputs one volume of size $S \times S \times S$. In Minecraft $S = 32$, in all other datasets $S = 64$. Each entry in the output volume holds 4 values: 3 for RGB colour and 1 for opacity.

Encoder. Each of the N images is first passed through a small 2D CNN and an 'Unprojector' (Fig. 4, top left) which pools the RGB colours from the image into a 3D volume along camera rays. Next, each image is independently passed through an encoder which contains a series of 3D ResNet and MaxPool blocks (Fig. 4, bottom). All convolutions are done with $3 \times 3 \times 3$ kernels. GroupNorm layers [9] with 8 groups are applied in every ResNet block to the intermediate output (Fig. 4, top right). There are 4 Convolutional Downscaling Blocks in the encoder, outputting $M = 5$ feature maps for each of N images $I^i,\ i \in 1, \ldots, N$. We use channel dimensions $C_1, C_2, C_3, C_4, C_5 = 64, 64, 128, 256, 512$.

Aggregator. Given N feature maps W_j at level j coming from N images (Fig. 5, left), the 'Aggregator' is tasked with aggregating the features into one feature volume W'_j at level j (Fig. 5, right). Each feature volume is flattened (Fig. 5, top left) and the flattened vectors are concatenated along the sequence dimension, forming a tensor of size $N \times H_j W_j D_j \times C_j$, which is input to an Attention layer with a GroupNorm layer and a residual skip-connection (Fig. 5, bottom middle). Concatenated features F_j are input as the Keys and Values to the attention layer, with the number of frames N being the sequence dimension and the flattened voxels being the batch dimension. Query volume Q_j for feature level j has size $1 \times H_j W_j D_j \times C_j$ and comes either from the previous layer of the decoder (next section) or, for the lowest feature level, is learnt and is the same for all inputs.

Method	3D Representation	σ_t	N_{train}	N_{inf}	$\lambda = 0$	3D Generation?	3D Reconstruction
RD	Triplane	$\{\bar{\sigma}_t\}$	1	1	Yes	Yes	Deterministic
RD++	Grid	$\{\bar{\sigma}_t\}$	1	1	No	Yes	Deterministic
Ours w/o $\mathcal{D}$	Grid	$\{0\}, \{0, 0\}$	1-2	1	No	No	Deterministic
Ours $\mathcal{D}$	Grid	$\{\bar{\sigma}_t\}, \{\bar{\sigma}_t, 0\}, \{\bar{\sigma}_t, \bar{\sigma}_t\}$	1-2	3-5	No	Yes	Generative

Table 1: **Baseline methods.** We summarise key differences of baselines which we compare Viewset Diffusion (Ours w $\mathcal{D}$) against. The methods vary in number of images used at input in training N_{train} and inference N_{inf} and what levels of noise σ_t are applied to the input images during training. In some baselines the unseen view is not penalised $\lambda = 0$, only a subset of them is able to perform 3D Generation and only our method performs 3D Reconstruction in a generative manner.

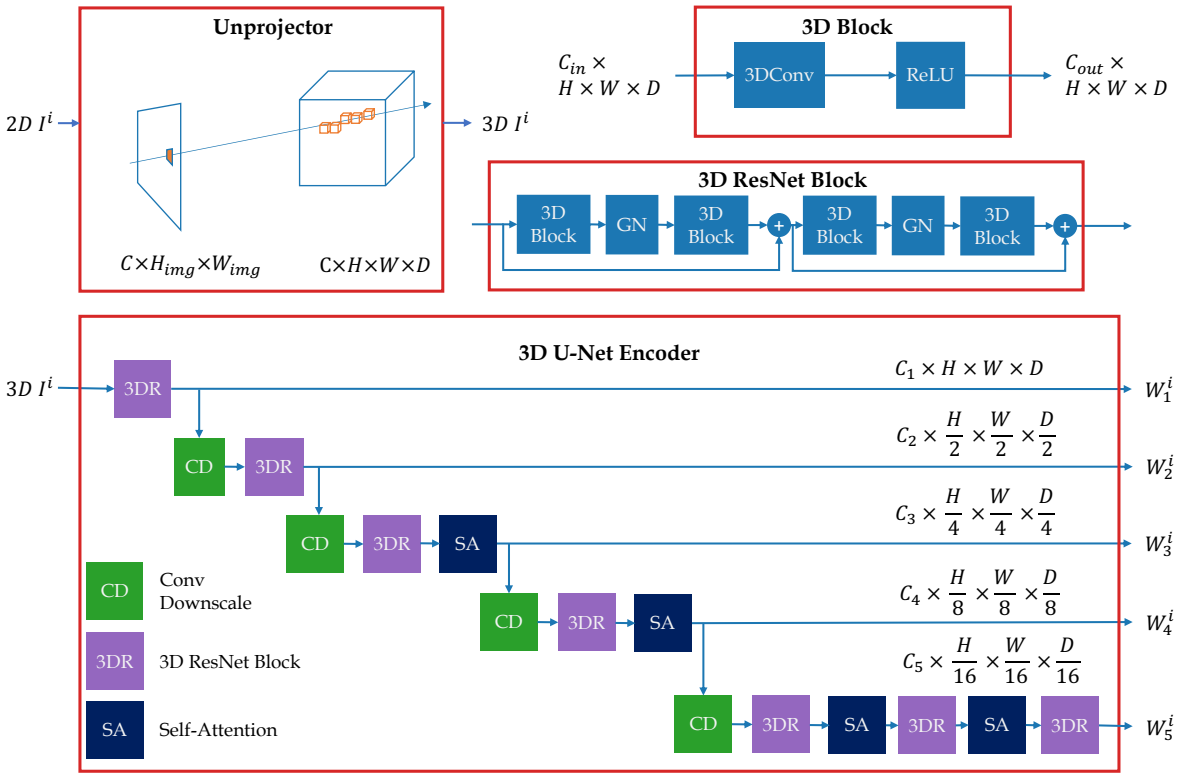

Figure 4: **Encoder.** Each image I^i out of N images input to the reconstructor is first unprojected to 3D (top left) and passed through a series of 3D ResNet Blocks (top right) and Max Pooling layers to output $M = 5$ feature maps W_j^i.

Decoder and query volume. The decoder takes MN feature volumes as input: N feature volumes $W_j^1, \ldots, W_j^N$ for each of M feature levels j (Fig. 6, left). Additionally, it takes as input a learnt feature volume Q_M which is identical for all inputs. At each feature level j, the decoder aggregates feature maps (Fig. 5) $W_j^1, \ldots, W_j^N$ using query volume Q_j to output an aggregated feature volume $W_j^{'}$. Feature volumes Q_j and $W_j^{'}$ are upscaled, concatenated and passed it through a 3D ResNet Convolutional Block (Fig. 6, bottom right). Once feature maps from all levels have been aggregated, the feature volume of size $C_1 \times H \times W \times D$ is passed through a small 5-layer convolutional upscaling subnetwork. Finally, we apply a $1 \times 1 \times 1$ non-activated convolution layer which outputs the radiance field $\boldsymbol{v}$ of shape $4 \times H \times W \times D$.

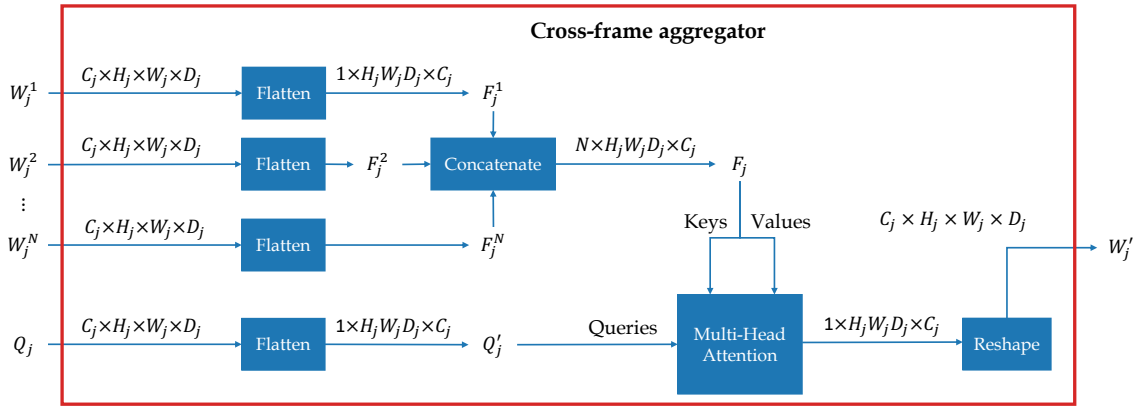

Figure 5: **Aggregator** takes in N feature volumes W_j^i at level j as well as a query volume Q_j and outputs an aggregated feature volume $W_j^{'}$.

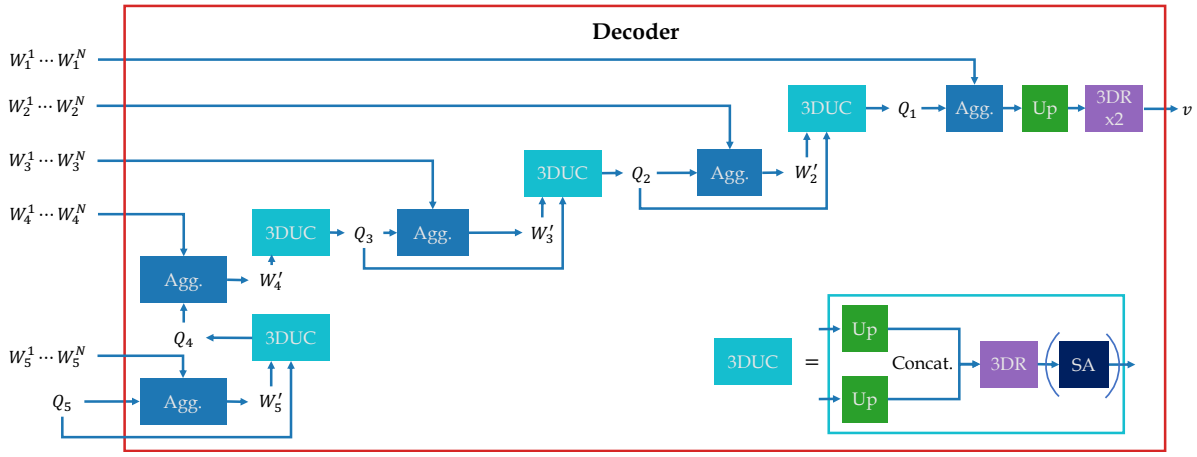

Figure 6: **Decoder** takes sets of feature maps at different levels and a learnt start query volume Q_5. Feature volumes from different images are aggregated, upscaled and passed through convolutional layers. After the features have been decoded, they are passed through a single convolutional layer to output the radiance field $\boldsymbol{v}$. Self attention layers SA are applied when leading to Q_4, Q_3, Q_2, but not when outputting Q_1.

References

[1] Tiankai Hang, Shuyang Gu, Chen Li, Jianmin Bao, Dong Chen, Han Hu, Xin Geng, and Baining Guo. Efficient diffusion training via min-snr weighting strategy. In *arXiv*, 2023. 4

[2] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Proc. NeurIPS*, 2020. 4

[3] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 4

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *Proc. ICLR*, 2015. 4

[5] Jeremy Reizenstein, Roman Shapovalov, Philipp Henzler, Luca Sbordone, Patrick Labatut, and David Novotny. Common Objects in 3D: Large-scale learning and evaluation of real-life 3D category reconstruction. In *Proc. CVPR*, 2021. 2

[6] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Proc. MICCAI*, 2015. 4

[7] Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. *Proc. ICLR*, 2021. 4

[8] Daniel Watson, William Chan, Ricardo Martin-Brualla, Jonathan Ho, Andrea Tagliasacchi, and Mohammad Norouzi. Novel view synthesis with diffusion models. In *Proc. ICLR*, 2023. 3

[9] Yuxin Wu and Kaiming He. Group normalization. In *Proc. ECCV*, 2018. 4

[10] Alex Yu, Vickie Ye, Matthew Tancik, and Angjoo Kanazawa. PixelNeRF: Neural radiance fields from one or few images. In *Proc. CVPR*, 2021. 3