ViewDiff: 3D-Consistent Image Generation with Text-to-Image Models

Supplementary Material

A. Supplemental Video

Please watch our attached video [1] for a comprehensive evaluation of the proposed method. We include rendered videos of multiple generated objects from novel trajectories at different camera elevations (showcasing unconditional generation as in Fig. 4). We also show animated results for single-image reconstruction (Fig. 6) and sample diversity (Fig. 7).

B. Training Details

B.1. Data Preprocessing

We train our method on the large-scale CO3Dv2 [33] dataset, which consists of posed multi-view images of real-world objects. Concretely, we choose the categories `Teddybear`, `Hydrant`, `Apple`, and `Donut`. Per category, we train on 500–1000 objects, with 200 images at resolution 256×256 for each object. We generate text captions with the BLIP-2 model [21] and sample one of 5 proposals per object during each training iteration. With probability p_1=0.5 we select the training images randomly per object and with probability p_2=0.5 we select consecutive images from the captured trajectory that goes around the object. We randomly crop the images to a resolution of 256×256 and normalize the camera poses such that the captured object lies in an axis-aligned unit cube. Specifically, we follow Szymanowicz et al. [42] and calculate a rotation transformation such that all cameras align on an axis-aligned plane. Then, we translate and scale the camera positions, such that their bounding box is contained in the unit cube.

B.2. Prior Preservation Loss

Inspired by Ruiz et al. [34], we create a *prior preservation* dataset of 300 images and random poses per category with the pretrained text-to-image model. We use it during training to maintain the image generation prior. This has been shown to be successful when fine-tuning a large 2D diffusion model on smaller-scale data [34]. For each of the 300 images we randomly sample a text description from the training set of CO3Dv2 [33]. We then generate an image with the pre-trained text-to-image model given that text description as input. During each training iteration we first calculate the diffusion objective (Eq. 3) on the N=5 multi-view images sampled from the dataset and obtain L_d. Then, we sample one image of the *prior preservation* dataset and apply noise to it (Eq. 2). Additionally, we sample a camera (pose and intrinsics) that lies within the distribution of cameras for each object category. We then similarly calculate the loss (Eq. 3) on the prediction of our model and obtain L_p. Since we only sample a single image instead of multiple, this does not train the diffusion model on 3D-consistency. Instead, it trains the model to maintain its image generation prior. Concretely, the cross-frame-attention layers are treated again as self-attention layers and the projection layers perform unprojection and rendering normally, but only from a single image as input. In practice, we scale the prior preservation loss with factor 0.1 and add it to the dataset loss to obtain the final loss: L=$L_d + 0.1L_p$.

[1] https://youtu.be/SdjoCqHzMMk

C. Evaluation Details

C.1. Autoregressive Generation

We showcase unconditional generation of our method in Sec. 4.1. To obtain these results, we employ our autoregressive generation scheme (Sec. 3.3). Concretely, we sample an (unobserved) image caption from the test set for the first batch and generate N=10 images with a guidance scale [13] of λ_{cfg}=7.5. Then we set λ_{cfg}=0 for subsequent batches and create a total of 100 images per object. We found that the results are most consistent, if the first batch generates N images in a 360° rotation around the object. This way, we globally define the object shape and texture in a single denoising forward pass. All subsequent batches are conditioned on all N images of the first batch. To render a smooth trajectory, we sample the camera poses in other batches in a sequence. That is, the next N images are close to each other with only a small rotation between them. We visualize this principle in our supplemental video.

C.2. Metric Computation

We give additional details on how we computed the metrics as shown in Tabs. 1 to 3. To ensure comparability, we evaluate all metrics on images without backgrounds as not every baseline models them.

FID/KID. We report FID [12] and KID [2] as common metrics for 2D/3D generation. We calculate these metrics to compare *unconditional* image generation against HoloFusion [18] and ViewsetDiffusion [42]. This quantifies the similarity of the generated images to the dataset and thereby provides insight about their quality (e.g., texture details and sharpness) and diversity (e.g., different shapes and colors). Following the baselines [18, 19], we sample 20,000 images from the CO3Dv2 [33] dataset for each object category. We remove the background from each object by using the foreground mask probabilities contained in the dataset. Similarly, we generate 20,000 images with each method and remove the

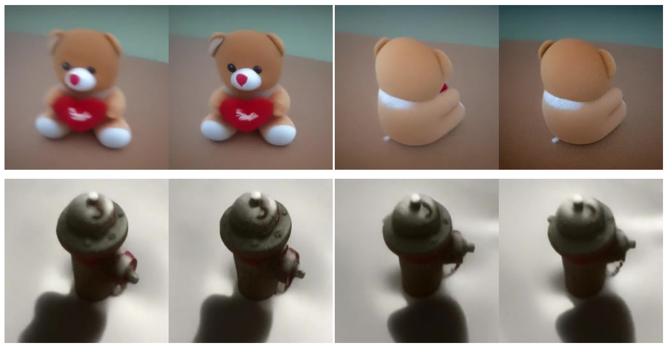


Figure 8. **Limitations.** Our method generates consistent images at different camera poses. However, there can be slight inconsistencies like different sharpness and lighting between images. Since our model is fine-tuned on a real-world dataset consisting of view-dependent effects (e.g., exposure changes), our framework learns to generate such variations across different viewpoints.

background from our generated images with CarveKit [36]. For our method, we set the text prompt to an empty string during the generation to facilitate complete unconditional generation.

PSNR/SSIM/LPIPS. We measure the multi-view consistency of generated images with peak signal-to-noise ratio (PSNR), structural similarity index (SSIM), and LPIPS [60]. We calculate these metrics to compare single-image reconstruction against ViewsetDiffusion [42] and DFM [46]. We resize all images to the resolution 256×256 to obtain comparable numbers. First, we obtain all objects that were not used during training for every method (hereby obtaining a unified test set across all methods). Then, we randomly sample 20 posed image pairs from each object. We use the first image/pose as input and predict the novel view at the second pose. We then calculate the metrics as the similarity of the prediction to the ground-truth second image. We remove the background from the prediction and ground-truth images by obtaining the foreground mask with CarveKit [36] from the *prediction* image. We use the same mask to remove background from both images. This way, we calculate the metrics only on similarly masked images. If the method puts the predicted object at a wrong position, we would thus quantify this as a penalty by comparing the segmented object to the background of the ground-truth image at that location.

D. Limitations

Our method generates 3D-consistent, high-quality images of diverse objects according to text descriptions or input images. Nevertheless, there are several limitations. Our method sometimes produces images with slight inconsistency, as shown in Fig. 8. Since the model is fine-tuned on a real-world dataset consisting of view-dependent effects (e.g., exposure changes), our framework learns to generate such variations across different viewpoints. This can lead to flickering artifacts when rendering a smooth video from a generated set of images (e.g., see the supplemental video). A potential solution is to (i) filter blurry frames from the dataset, and (ii) add lighting-condition through a ControlNet [59].

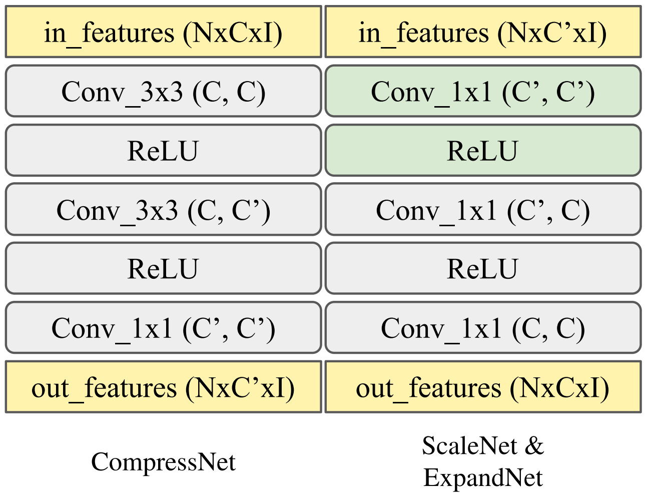


Figure 9. **Architecture of projection layer components.** The projection layer contains the components *CompressNet*, *ScaleNet* (green), and *ExpandNet*. We implement these networks as small CNN networks.

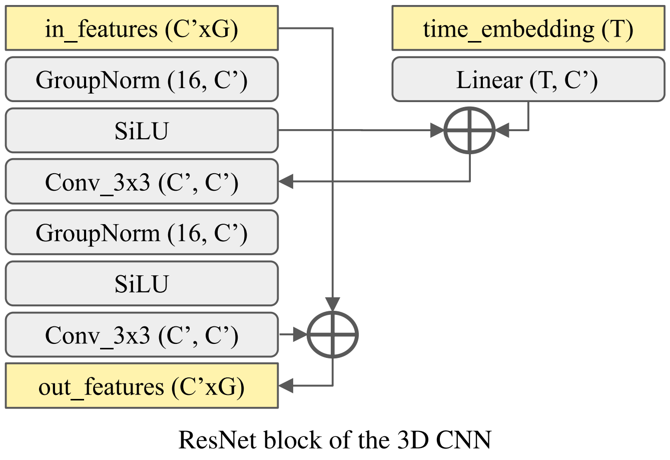
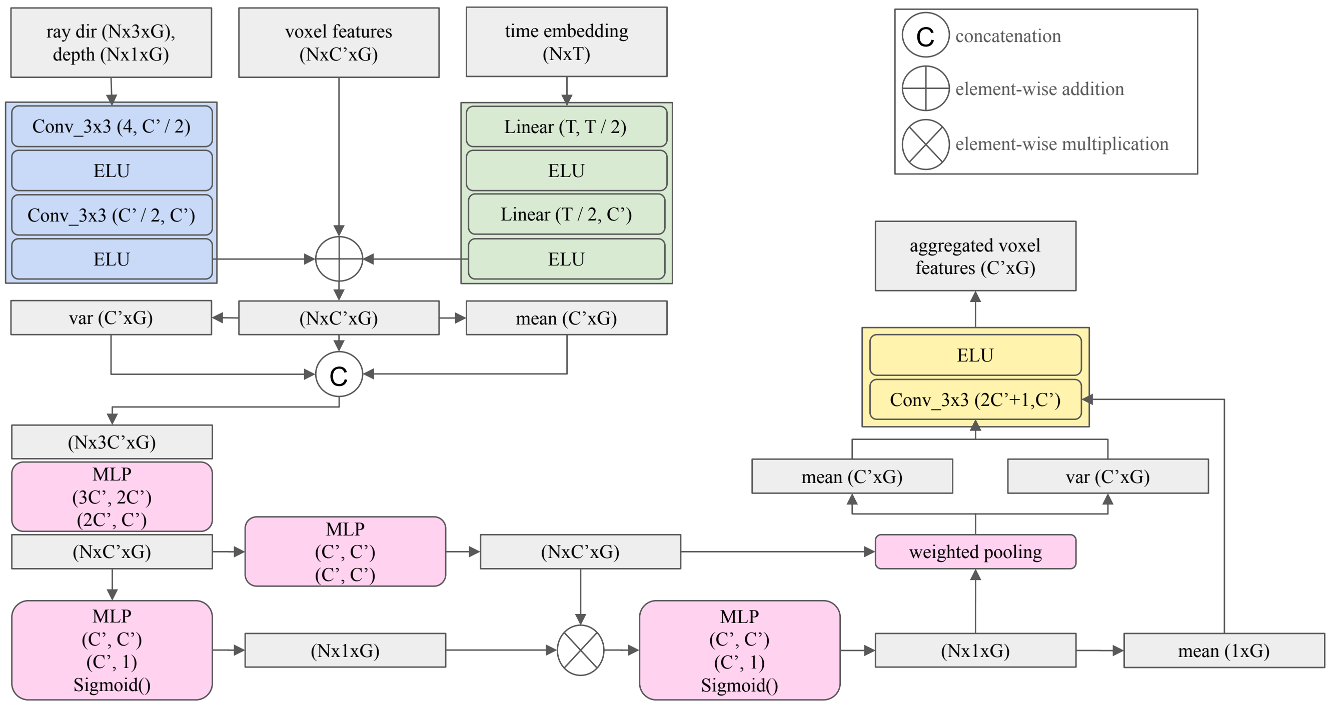


Figure 10. **Architecture of projection layer components.** The projection layer contains the component *3D CNN*. We implement this networks as a series of 5 3D ResNet [11] blocks with timestep embeddings.

E. Projection Layer Architecture

We add a projection layer into the U-Net architecture of pre-trained text-to-image models (see Fig. 3 and Sec. 3.2). The idea of this layer is to create 3D-consistent features that are



Aggregator MLP

Figure 11. **Architecture of projection layer components.** The projection layer contains the component *Aggregator MLP*. First, we combine per-view voxel grids with their ray-direction/depth encodings (blue) and the temporal embedding (green). Inspired by IBRNet [51], the MLPs (pink) then predict per-view weights followed by a weighted feature average. Finally, we combine the per-voxel weights with the mean and variance grids (yellow) to obtain the aggregated feature grid.

then further processed by the next U-Net layers (e.g. ResNet blocks). Concretely, we create a 3D representation from all input features in form of a voxel grid, that is defined inside of the axis-aligned unit cube. We set the 3D feature dimension as $C' = 16$ and define the base resolution of the voxel grid as $128 \times 128 \times 128$. Throughout the U-Net, we apply the same up/downsampling as for the 2D features, i.e., the resolution decreases to $8 \times 8 \times 8$ in the bottleneck layer. The projection layer consists of multiple network components. We show detailed network architectures of these components in Figs. 9 to 11.

E.1. CompressNet and ExpandNet

We apply the 3D layers on features that are defined in a unified dimensionality of C'=16. Since our 3D layers act on dense voxel grids this helps to lower the memory requirements. To convert to/from this compressed feature space, we employ small CNNs, as depicted in Fig. 9. In these schematics, we define N as the number of images in a batch, C as the uncompressed feature dimension and I as the spatial dimension of the features.

E.2. Aggregator MLP

After creating per-view voxel grids via raymarching (see Sec. 3.2), we combine N voxel grids into one voxel grid that represents the features for all viewpoints. To this end, we employ a series of networks, as depicted in Fig. 11. In these schematics, we define N as the number of images in a batch, C' as the compressed feature dimension, T as the dimension of the timestep embedding, G as the 3D voxel grid resolution, and I as the spatial dimension of the features. The MLPs are defined as a sequence of linear layers of specified input and output dimensionality with *ELU* activations in between.

First, we concatenate the voxel features with an encoding of the ray-direction and depth that was used to project the image features into each voxel. We also concatenate the timestep embedding to each voxel. This allows to combine per-view voxel grids of different timesteps (e.g., as proposed in image conditional generation in Sec. 3.3). It is also useful to inform the subsequent networks about the denoising timestep, which allows to perform the aggregation differently throughout the denoising process. Inspired by IBRNet [51], a set of MLPs then predict per-view weights followed by a weighted feature average. We perform this averaging operation elementwise: since all voxel grids are defined in the same unit cube, we can combine the same voxel across all views. Finally, we combine the per-voxel weights with the mean and variance grids to obtain the final aggregated feature grid.

E.3. 3D CNN

After aggregating the per-view voxel grids into a joint grid, we further refine that grid. The goal of this network is to add additional details to the feature representation such as the global orientation of the shape. To achieve this, we employ a series of 5 3D ResNet [11] blocks with timestep embeddings, as depicted in Fig. 10. In these schematics, we define C' as the compressed feature dimension, T as the dimension of the timestep embedding, and G as the 3D voxel grid resolution.

E.4. Volume Renderer and ScaleNet

After we obtain a refined 3D feature representation in form of the voxel grid, we render that grid back into per-view image features (see Fig. 3). Concretely, we employ a volume renderer similar to NeRF [26] and implement it as a grid-based renderer similar to DVGO [40]. This allows to render features in an efficient way that is not a bottleneck for the forward pass of the network. In contrast to NeRF, we render down *features* instead of *rgb* colors. Concretely, we sample 128 points along a ray and for each point we trilinearly interpolate the voxel grid features to obtain a feature vector $f \in \mathbb{R}^{C'}$. Then, we employ a small 3-layer MLP that transforms f into the density $d \in \mathbb{R}$ and a sampled feature $s \in \mathbb{R}^{C'}$. Using alpha-compositing, we accumulate all pairs $(d_0, s_0), ..., (d_{127}, s_{127})$ along a ray into a final rendered feature $r \in \mathbb{R}^{C'}$. We dedicate half of the voxel grid to foreground and half to background and apply the background model from MERF [32] during ray-marching.

We found it is necessary to add a scale function after the volume rendering output. The volume renderer typically uses a *sigmoid* activation function as the final layer during ray-marching [26]. However, the input features are defined in an arbitrary floating-point range. To convert r back into the same range, we non-linearly scale the features with 1×1 convolutions and *ReLU* activations. We depict the architecture of this *ScaleNet* as the green layers in Fig. 9.

F. Additional Results

F.1. Comparison To Additional Baselines

We compare against additional text-to-3D baselines that also utilize a pretrained text-to-image model in Fig. 12. We choose ProlificDreamer [52] as representative of score distillation [29] methods. Rendered images are less photorealistic since the optimization may create noisy surroundings and over-saturated textures. Similar to us, Zero123-XL [24] and SyncDreamer [25] circumvent this problem by generating 3D-consistent images directly. However, they finetune on a large synthetic dataset [7] instead of real-world images. As a result, their images have synthetic textures and lighting effects and no backgrounds. We quantify this in Tab. 4 with the FID between sets of generated images (conditioned on an input view), and real images of the same object (without backgrounds). Our method has better scores since the generated images are more photorealistic.

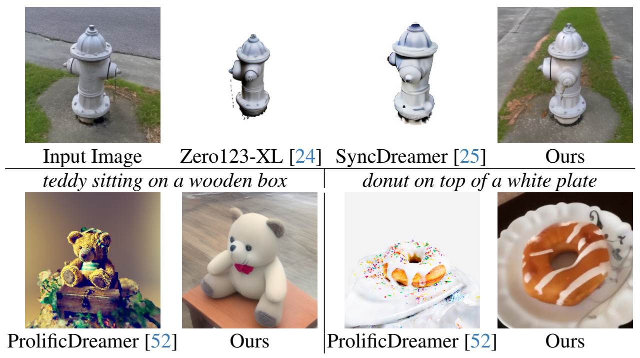


Figure 12. **Comparison to other text-to-3D baselines from image- (top) and text-input (bottom).** Our method produces images with higher photorealism and authentic surroundings.

Table 4. **Comparison of consistency (mid) and photorealism (FID).** Our method shows similar 3D-consistency as baselines, while producing more photorealistic images.

Method	$E_{\text{warp}}\downarrow$	#Points↑	PSNR↑	FID↓
DFM [46]	0.0034	17,470	32.32	—
VD [42]	0.0021	—	—	—
HF [18]	0.0031	—	—	—
SyncDreamer [25]	0.0042	4,126	33.81	135.78
Zero123-XL (SDS) [24]	0.0039	—	—	126.83
Ours	0.0036	18,358	33.65	85.08

We calculate temporal stability (E_{warp}) of video renderings with optical flow warping following [20]. Also, we measure the consistency of generated images for methods that do not directly produce a 3D representation. Concretely, we report the number of point correspondences following [25] and the PSNR between NeRF [26] re-renderings and input images. Table 4 shows that our method is on-par with baselines in terms of 3D consistency, while generating higher quality images.

F.2. Unconditional Generation

We generate images in a similar fashion as in Sec. 4.1. Concretely, we sample an (unobserved) image caption from the test set for the first batch and generate N=10 images with a guidance scale [13] of λ_{cfg}=7.5. Then we set λ_{cfg}=0 for subsequent batches and create a total of 100 images per object. We show additional results in Figs. 13 to 16.

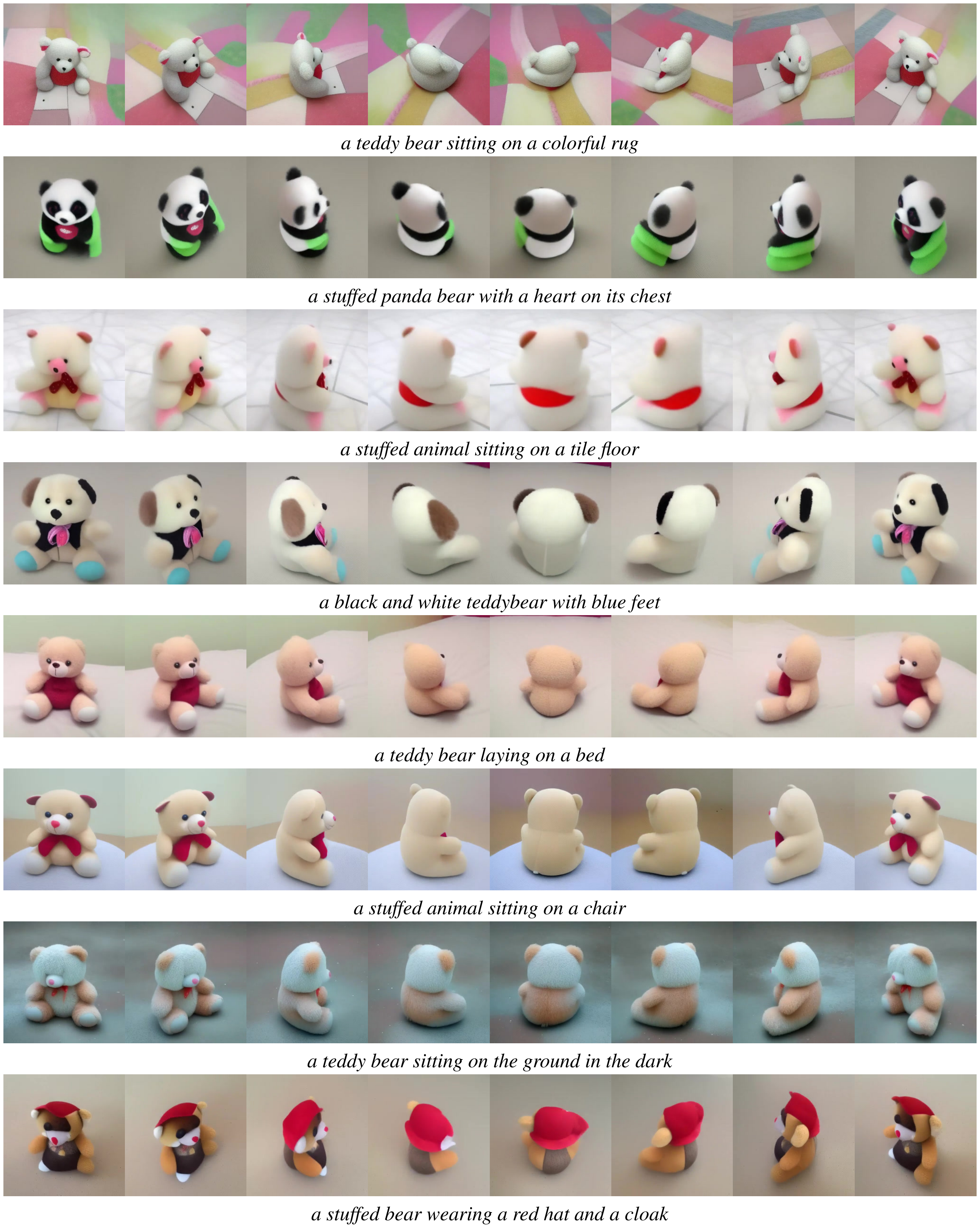
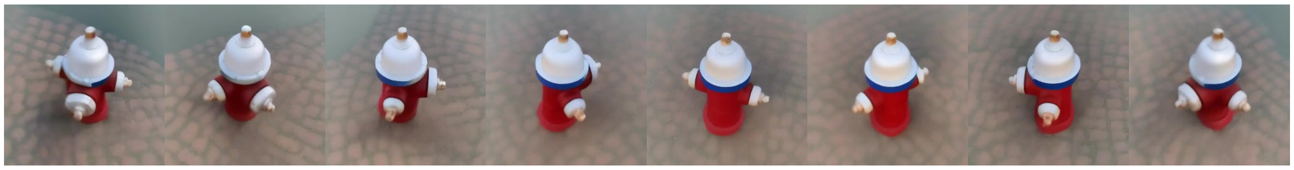
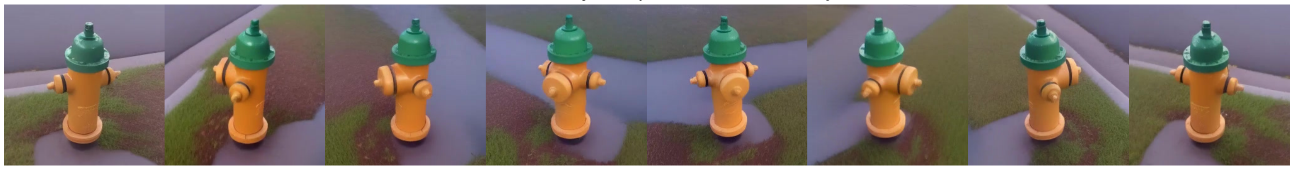
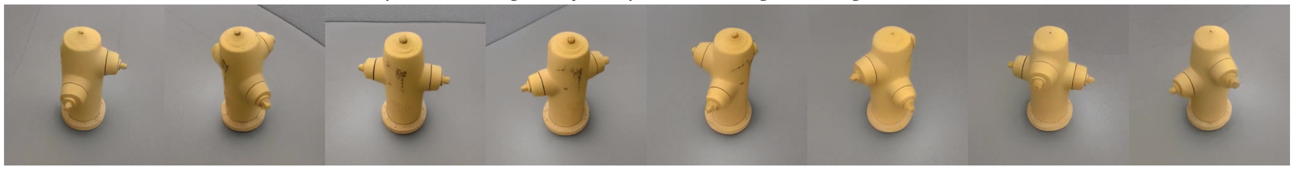
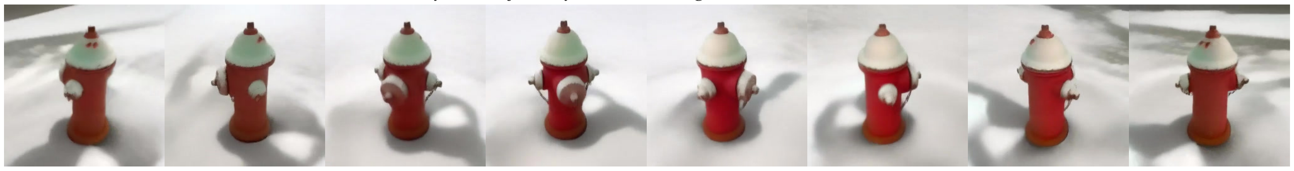

a teddy bear sitting on a colorful rug
a stuffed panda bear with a heart on its chest
a stuffed animal sitting on a tile floor
a black and white teddybear with blue feet
a teddy bear laying on a bed
a stuffed animal sitting on a chair
a teddy bear sitting on the ground in the dark
a stuffed bear wearing a red hat and a cloak

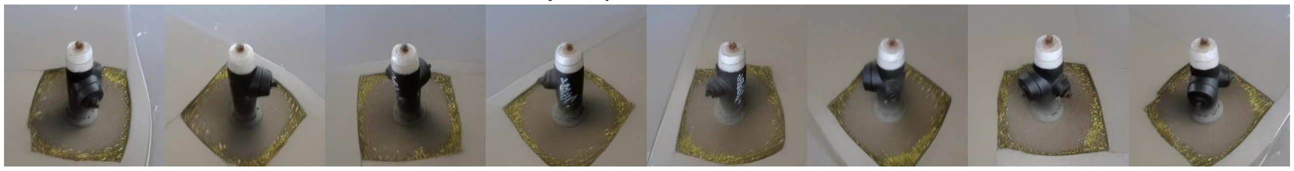
Figure 13. **Additional examples of our method.** Given a text prompt as input, we generate a smooth trajectory around an object with our autoregressive generation scheme (Sec. 3.3). Please see the supplemental video for animations of the generated samples.

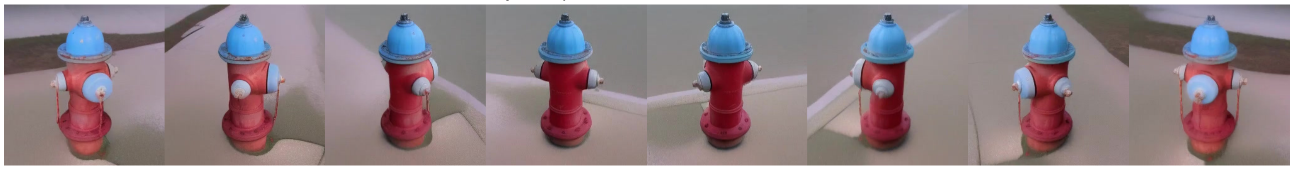

a red and white fire hydrant on a brick floor

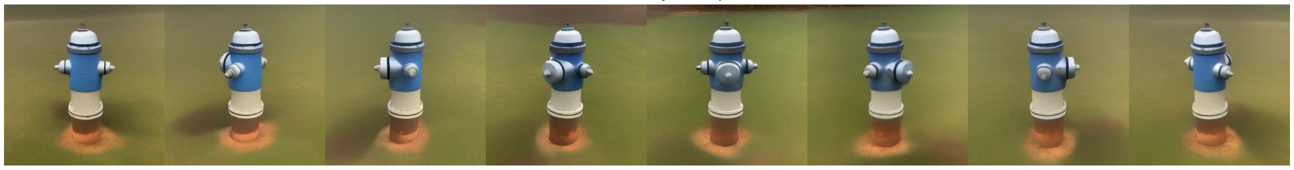

a yellow and green fire hydrant sitting on the ground


a yellow fire hydrant sitting on the sidewalk


a red fire hydrant in the snow


a fire hydrant on the sidewalk


a red and blue fire hydrant


a blue and white fire hydrant sitting in the grass

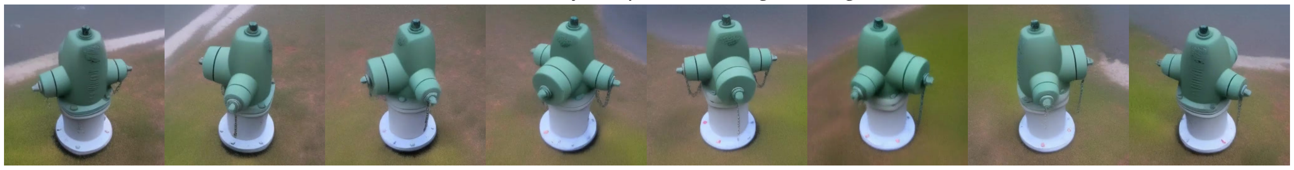
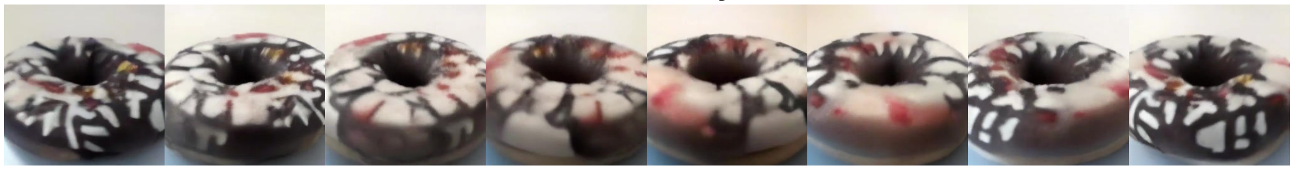

a green fire hydrant with a tag on it

Figure 14. **Additional examples of our method.** Given a text prompt as input, we generate a smooth trajectory around an object with our autoregressive generation scheme (Sec. 3.3). Please see the supplemental video for animations of the generated samples.

a glazed donut sitting on a marble counter

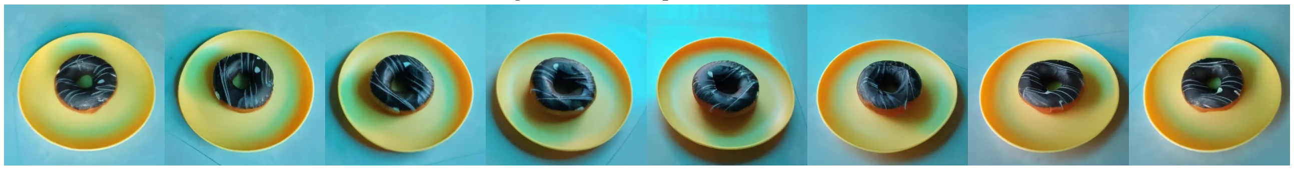
a donut on a clear plate



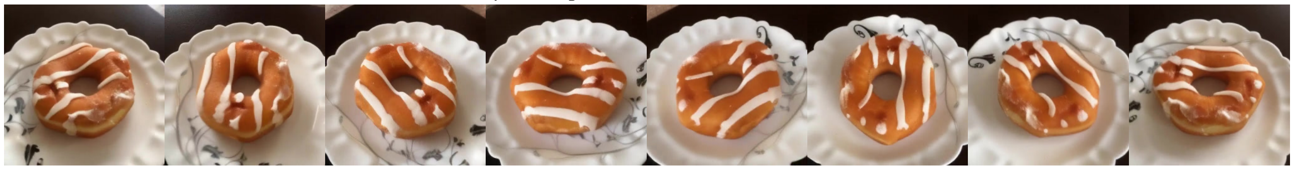
a chocolate donut with sprinkles on it

a donut on a plate with a hole in it

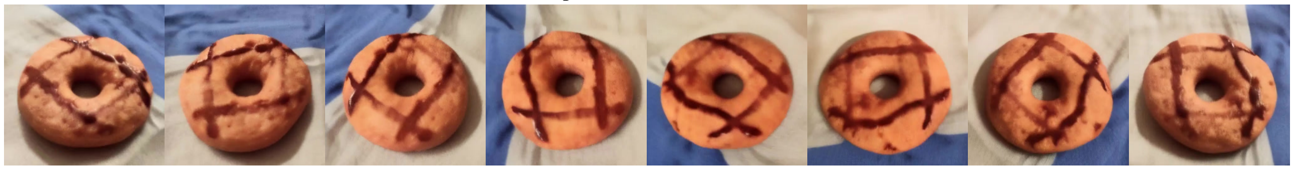
a large donut on a plate on a table



a yellow plate with a donut on it

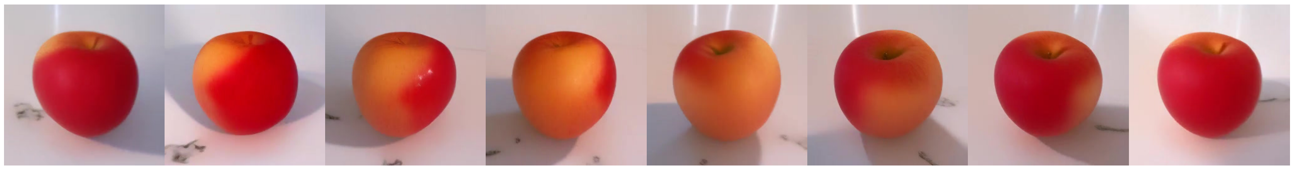


a white plate with a donut on it

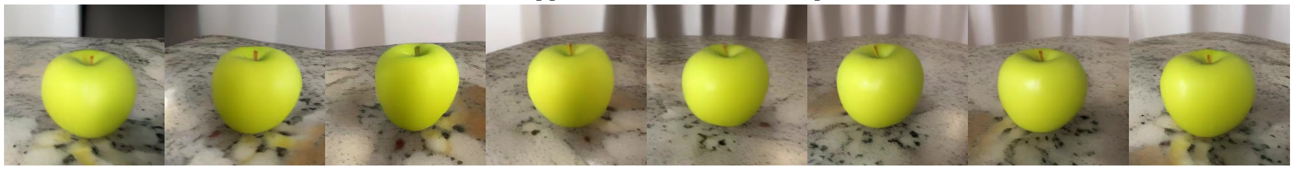


a donut sitting on a cloth

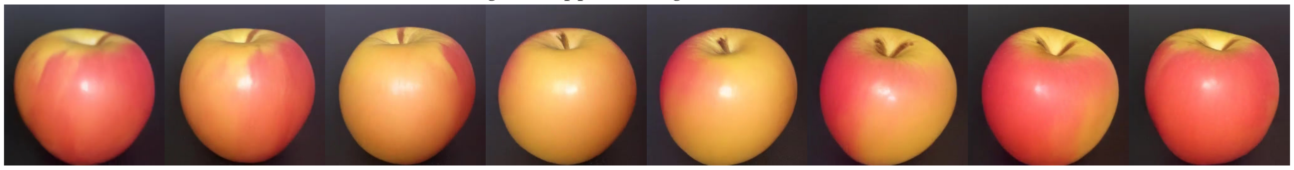
Figure 15. **Additional examples of our method.** Given a text prompt as input, we generate a smooth trajectory around an object with our autoregressive generation scheme (Sec. 3.3). Please see the supplemental video for animations of the generated samples.



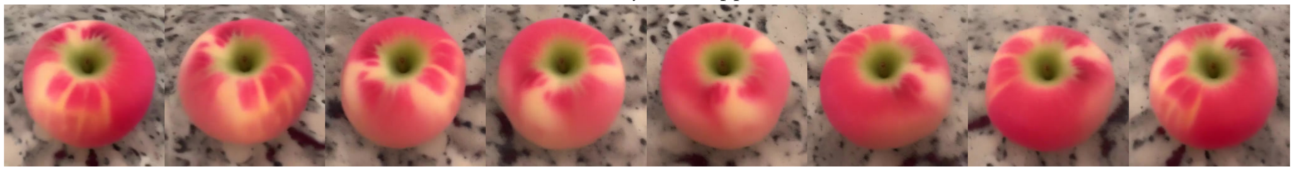
a red apple on a white counter top



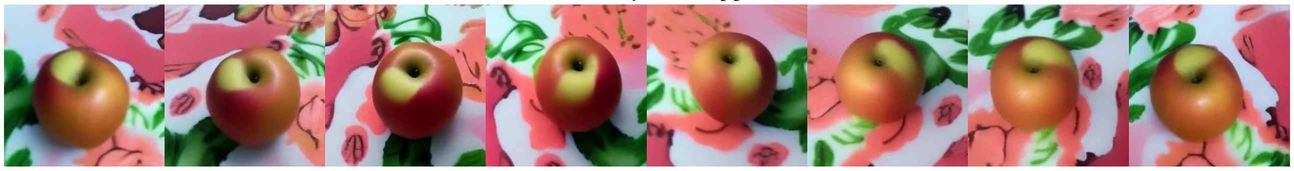
a green apple sitting on a counter



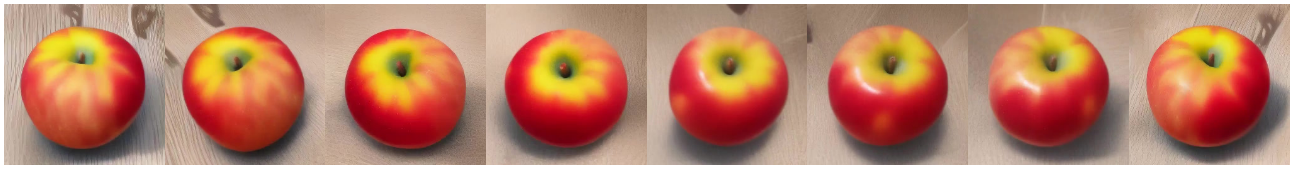
a red and yellow apple



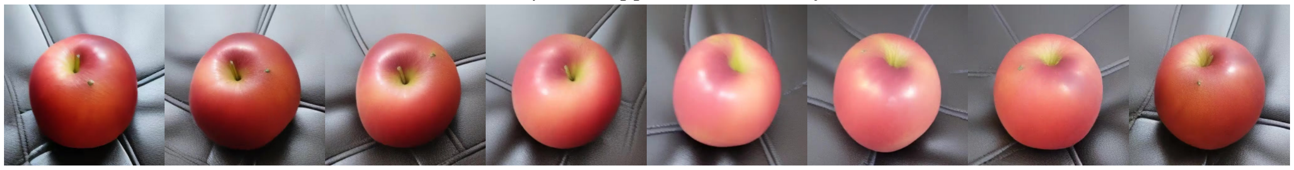
a red and yellow apple



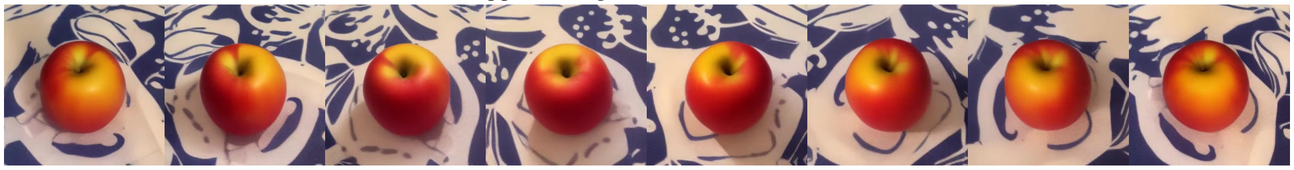
a single apple on a table cloth with a floral pattern



a red and yellow apple on a wooden floor



a red apple sitting on a black leather couch



a red apple on a blue and white patterned table cloth

Figure 16. **Additional examples of our method.** Given a text prompt as input, we generate a smooth trajectory around an object with our autoregressive generation scheme (Sec. 3.3). Please see the supplemental video for animations of the generated samples.

G. Optimizing a NeRF/NeuS

Our method is capable of directly rendering images from novel camera positions in an autoregressive generation scheme (see Sec. 3.3). This allows to render smooth trajectories around the same 3D object at arbitrary camera positions. Depending on the use case, it might be desirable to obtain an explicit 3D representation of a generated 3D object (instead of using our method to autoregressively render new images). We demonstrate that our generated images can be used directly to optimize a NeRF [26] or NeuS [50]. Concretely, we optimize a NeRF with the Instant-NGP [27] implementation from our generated images for 10K iterations (2 minutes). Also, we extract a mesh by optimizing a NeuS with the *neus-facto* implementation from SDFStudio [43, 58] for 20K iterations (15 minutes). First, we remove the background of our generated images by applying Carvekit [36] and then start the optimization with these images. We show results in Figs. 17 to 19.

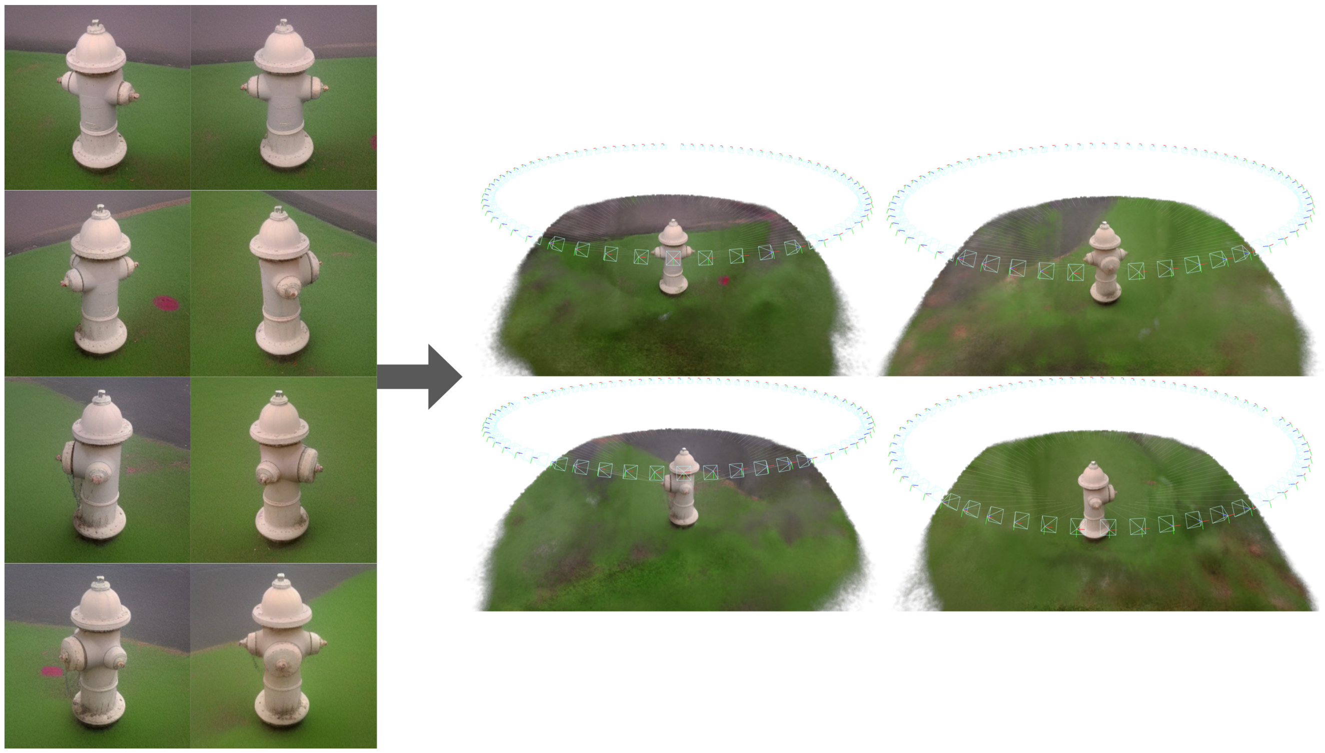


Figure 17. **NeRF [26] optimization from our generated images.** Left: given a text prompt as input, we generate a smooth trajectory around an object with our autoregressive generation scheme (Sec. 3.3). In total, we generate 100 images at different camera positions. Right: we create a NeRF using Instant-NGP [27] from the generated images. We show the camera positions of the generated images on top of the optimized radiance field.

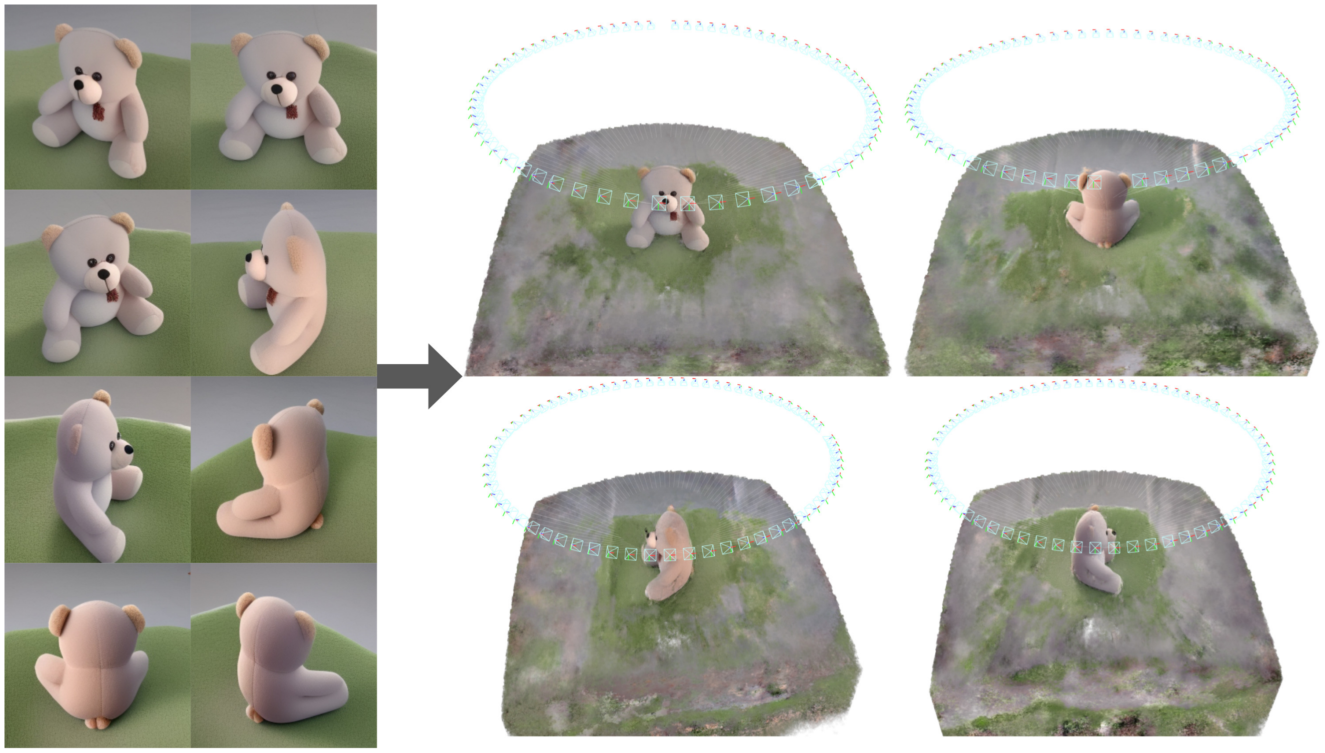


Figure 18. **NeRF [26] optimization from our generated images.** Left: given a text prompt as input, we generate a smooth trajectory around an object with our autoregressive generation scheme (Sec. 3.3). In total, we generate 100 images at different camera positions. Right: we create a NeRF using Instant-NGP [27] from the generated images. We show the camera positions of the generated images on top of the optimized radiance field.

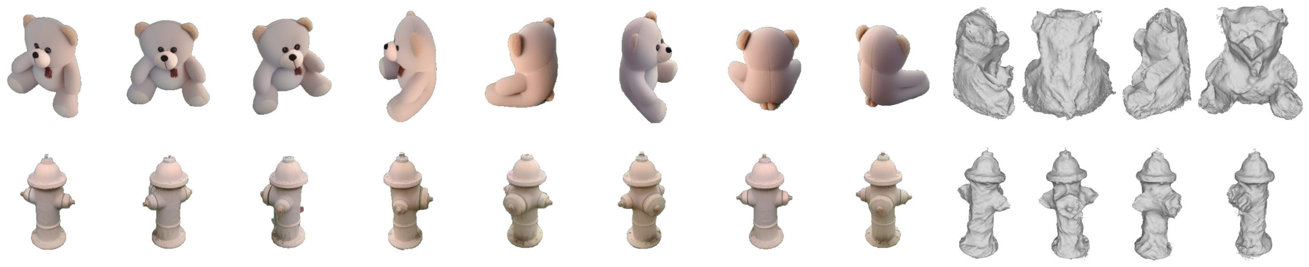


Figure 19. **Mesh extraction from our generated images.** Given a text prompt as input, we generate a smooth trajectory around an object with our autoregressive generation scheme (Sec. 3.3). In total, we generate 100 images at different camera positions and mask-out the background with Carvekit [36]. We then optimize a NeuS [50] and extract the mesh from it (last 4 images per row).