6 Appendices

6.1 Video demonstrations

- Qualitative results on various datasets from SinGRAV.
- Qualitative results from SinGRAV-derived variants.
- Qualitative results from baselines including GRAF [4], pi-GAN [1] and GIRAFFE [5].
- Qualitative results on real-world indoor data.
- Qualitative results for applications.

Please note that a low-resolution video is provided due to the size limit, we can provide a high-resolution video upon the request.

6.2 Implementation details

We describe more implementation details including details for demonstrated applications in 6.2.1, details for evaluation measures in Section 6.2.2 and detailed network structures for SinGRAV in Section 6.2.3. In addition, we provide the detailed training settings for SinGRAV in Section 6.2.4.

6.2.1 Implementation details of applications

In this section, we present the implementation details of the demonstrated applications, including *Scene Animation* and *Scene Editing*. Since the generator G_N at the finest scale only operates in 2D rendering domain and functions mainly as a faithful super-resolving module, we omit it for brevity in the descriptions below.

Scene Animation. To plausibly animate a generated 3D scene for T consecutive time steps, we need to construct a sequence of radiance volumes $\{\tilde{\mathbf{V}}_{N-1}^{(t)}\}_{t=1}^{t=T}$ with similar layouts but slightly changing content. In our implementation, we achieve this by utilizing the generators to produce $\{\tilde{\mathbf{V}}_{N-1}^{(t)}\}_{t=1}^{t=T}$ from the modified versions of the pyramid of noise volumes $\{\boldsymbol{z}_n\}_{n=1}^{n=N-1}$, which is used to generate the original scene $\tilde{\mathbf{V}}_{N-1}$. Specifically, we inject random changes smoothly for T time steps $\{t\}_{t=1}^{T-1}$, and restrict the resulting noise volumes to stay close to the original ones $\{\boldsymbol{z}_i\}_{n=1}^{n=N-1}$. Formally, the way we construct the random walk is given by:

$$\begin{aligned} \boldsymbol{z}_n^{(t+1)} &= \alpha \boldsymbol{z}_n^{(1)} + (1-\alpha)(\boldsymbol{z}_n^{(t)} + \delta_n^{(t+1)}), \\ \delta_n^{(t+1)} &= \xi(\boldsymbol{z}_n^{(t)} - \boldsymbol{z}_n^{(t-1)}) + (1-\xi)\mu_n^{(t+1)}, \end{aligned} \tag{7}$$

where n denotes the index running over the pyramid of the noise volumes. $\boldsymbol{z}_n^{(1)}$ is the noise volume at the first step and is set as the original noise volume at scale n, *i.e.*, $\boldsymbol{z}_n^{(1)} = \boldsymbol{z}_n$. $\delta_n^{(t+1)}$ is the injected change at time step $t+1$, and $\mu_n^{(t)}$ is a noise volume that has the same distribution as $\boldsymbol{z}_n$. α and ξ are two parameters that can be tuned to achieve different animation effects. Concretely, α controls the degree of preserving the original scene, *i.e.*, the similarity between the original scene $\tilde{\mathbf{V}}_{N-1}$ and the animated scene at time step $t+1$. ξ controls the smoothness between the adjacent frames and the rate of introduced changes in the generated frame. Note that the noise volumes at the coarser scales are able to control the global layout of the generated scenes, on which slight modifications might drastically alter the scene. Thus we choose to inject the random changes in finer scales while keeping the noise volumes at the coarser scales unchanged for each time step. For obtaining the demonstrated animation effects for *candles* and *crystal*, we let the random change injection start from the third scale and set $\alpha = 0.58$ and $\xi = 0.45$.

Scene Editing. For editing a scene, we first need to localize the target area, then simply manipulate the volume $\tilde{\mathbf{V}}_{N-1}$, and finally downscale the edited volume and let the volume go through the pyramid of generators to harmonize the edited scene. We provide detailed description of these steps below.

Step1: object area localization. For finding an area of interest, we first extract the mesh from a scene volume $\tilde{\mathbf{V}}_{N-1}$, and interactively locate the bounding box of the area of interest in a 3D mesh visualizer, e.g., Blender in our prototype implementation. Then

we construct a 3D mask $\boldsymbol{M}$, where the area within the bounding box is filled with 1 and the rest is set to 0. Besides, we also sample a point within the area representing 'air' and use its color and density as $\mathbf{c}_{empty}$ and σ_{empty}.

Step2: volume manipulation. We utilize the mask obtained in Step 1 to manipulate the volume in various ways, enabling us to achieve multiple editing tasks, which will be described in detail below.

For the application *Removal*, we manipulate the volume via

$$\tilde{\mathbf{V}}_{N-1}(\boldsymbol{M}=1)=(\mathbf{c}_{empty},\sigma_{empty}). \tag{8}$$

For the application *Duplicate*, we select an area with empty voxels within the volume, which is masked by $\boldsymbol{M}_{empty}$. Then we set the empty area to the values extracted from the object area covered by $\boldsymbol{M}$. That is, we manipulate the volume via

$$\tilde{\mathbf{V}}_{N-1}(\boldsymbol{M}_{empty}=1)=\tilde{\mathbf{V}}_{N-1}(\boldsymbol{M}=1). \tag{9}$$

For the application *Move*, we achieve the effects by first conducting the *Duplicate* operation and then performing the *Removal* operation to $\tilde{\mathbf{V}}_{N-1}$.

For the application *Composition*, we select the object of interest in k generated scenes and construct a group of masks $\{\boldsymbol{M}_i\}_{i=1}^{k}$. Then we set k target bounding boxes within an extra scene, which are destinations where we want to inject the new objects in. Consequently, we get k target masks $\{\mathbf{M}_i^{dest}\}_{i=1}^{k}$. Then we combine the objects into a single scene by putting the extracted sub-volume covered by $\boldsymbol{M}_i$ for each object in corresponding area $\boldsymbol{M}_i^{dest}$.

Step3: neural editing. After each kind of volume manipulation mentioned above, we downscale it to the size of $\tilde{\mathbf{V}}_3$ and then let $\{G_n\}_{n=3}^{N-1}$ to generate the final volume $\tilde{\tilde{\mathbf{V}}}_{N-1}$. This step will smooth the discontinuities caused by the naive volume manipulations. Then we can generate the renderings from the newly generated scene.

6.2.2 More details for evaluation metrics

In this section, we provide more details for calculating the scores for the evaluation metrics including *SIFID-MV* and *Diversity-MV*. Specifically, for each dataset, we randomly sample $M = 40$ pairs of color image $\boldsymbol{x}_m$ and camera pose ps_m, and m denotes the index running over the sampled viewpoints. Then, we generate $J = 50$ scenes, which are rendered under the sampled viewpoints to get 40 rendered images $\tilde{\mathbf{x}}_m^j$ per scene. Note that the notions for scales are omit here for brevity since we only use the rendered image at the finest scale for evaluation. Herein, we calculate the score for *SIFID-MV* according to

$$\text{SIFID-MV}=\frac{1}{M}\frac{1}{J}\sum_{m=1}^{M}\sum_{j=1}^{J}\text{SIFID}(\tilde{\mathbf{x}}_m^j, x_m), \tag{10}$$

where $\text{SIFID}(\cdot,\cdot)$ denotes the SIFID metric proposed in [6], which is designed for measuring the distance between the internal distributions of two single images.

And we calculate the score for *Diversity-MV* according to

$$\text{Diversity-MV}=\frac{1}{M}\sum_{m=1}^{M}\frac{\text{std}(\{\tilde{\mathbf{x}}_m^j\}_{j=1}^{J})}{\text{std}(\boldsymbol{x}_m)}, \tag{11}$$

where $\text{std}(\cdot)$ denotes calculating the standard deviation of each pixel over all of the rendered images under the same viewpoints. As shown in Equation (11), before calculating the averages over all of the viewpoints, we normalize the score under single viewpoint with the standard deviation over all of the pixels within the corresponding input observation.

6.2.3 More details for network design

Detailed structure for $\{G_n\}_{n=1}^{N-1}$ For generators ranging from the coarsest scale to scale $N-1$, we adopt the same structure. Specifically, for generators $\{G_n\}_{n=1}^{N-1}$, we adopt an L-layer structure, which is designed with 3D convolutional layers, Batch Normalization and LeakyReLU activation. The detailed structure of $\{G_n\}_{n=1}^{N-1}$ is summarized in Table 3. As shown in Table 3, the number of input channels, IC_n^V, is adjusted according to the difference of the input volume

for different scales. At the coarsest level, to make the volume adapt to the spatial anchor volume e_{csg}, we set $\mathrm{IC}_1^V = 3$. For the remaining 3D generators, we set $\mathrm{IC}_n^V = 4$. In addition, for generators whose scale is in range $[2, N-1]$, a residual connection is set up between the output volume and the upsampled $\tilde{\mathbf{V}}_{n-1}$, *i.e*, the generators $\{G_n\}_{n=2}^{N-1}$ learn to add more details to the synthesized scene.

Structure of G_N At the finest level, we adopt a 2D neural generator G_N to directly lift the rendered images to the output resolution 320×320. We adopt an upsampling layer proposed in [3] at the first layer, after which we design 4 layers with 2D convolutional layers and Instance Normalization. The detailed structure is given in Table 4. Besides, we utilize a residual connection between the final output and the resized input image, *i.e.*, $\tilde{\mathbf{x}}_{N-1} \uparrow$.

Detailed structures for and $\{D_n^{2D}\}_{n=1}^N$ We design $\{D_n^{2D}\}_{n=1}^N$ in a similar manner to the design of $\{G_n\}_{n=1}^{N-1}$, *i.e.*, all of $\{D_n^{2D}\}_{n=1}^N$ are designed with the same structure implemented with 2D convolutional layers and LeakyReLU() activation. In Table 5, we provide the detailed information of the structure. The discriminators have the same number of layers as the generators $\{G_n\}_{n=1}^{N-1}$. For the first layer of the generators $\{D_n^{2D}\}_{n=1}^{N-1}$, we set $\mathrm{IC}_1^I = 4$ since their input is formed by concatenating the rendered color image $\tilde{\mathbf{x}}_n$ and depth map $\tilde{\mathbf{d}}_n$. For the discriminator D_N^{2D} at the finest scale, we set $\mathrm{IC}_N^I = 6$ as its input is the concatenation of $\tilde{\mathbf{x}}_N$ and $\tilde{\mathbf{x}}_{N-1} \uparrow$, which is the output from G_N and the super-resolved image from the preceding scale respectively.

6.2.4 Training details for SinGRAV

Weight initialization. For the networks at the coarsest level, $\{G_1, D_1\}$, we adopt a random initialization. For generator G_n and discriminator D_n $(n > 1)$, we initialize the weights using the weights from trained networks of the preceding scale, *i.e.*, weights of G_{n-1} and D_{n-1}. We empirically find that using the weights from the pre-trained model of the preceding scale is helpful to stabilize the training.

Parameter settings. The weight of gradient penalty term in the adversarial loss is set to 0.1. We adopt the Adam optimizer for optimizing both the generator and discriminator, for both of which the learning rate is set to 0.0005 and the momentum parameters are set as $\beta_1 = 0.9, \beta_2 = 0.999$.

Training. We sequentially train the networks at different scales in a coarse-to-fine manner and the networks at coarser levels are fixed after being trained. Fixing the networks at the coarser scales is helpful to stabilize the training process of the network at finer scales, since the results at the beginning of the training at each scale is messy. At each scale s, we train the networks for 80 epochs. Specifically, we first train G_n using the reconstruction loss defined in Equation (6) for 20 epochs and then train $\{G_n, D_n^{2D}\}$ together using the loss defined in Equation (5). In each iteration, we alternate between 3 gradient steps for the discriminator D_n^{2D} and 3 steps for the generator G_n.

Time and platform. We train SinGRAV with 4 NVIDIA Tesla V100 GPUs (32GB memory) for around 2 days. For generating a new scene, the inference time is 0.32 seconds and the memory consumption is about 5 GB.

Parameter settings for different data In Table 6, we list the detailed parameter settings used for training SinGRAV on the collected datasets. For most of the data, the settings shown in the first row of Table 6 are applicable. For training on *island*, we slightly adjust the number of layers in $\{G_n\}_{n=1}^{N-1}$ and $\{D_n^{2D}\}_{n=1}^N$ to make them have a smaller receptive field, which is helpful to learn the fine textures existed in *island* dataset. For training on *water lily*, we slightly increase the resolution of $\tilde{\mathbf{V}}_1$ considering the complicated geometries of the water lily. Besides, with the gradually increased resolution of volumes and rendered images, we adjust the batch size at each scale in order to accommodate the training to the memory budget of the hardwares. Note that the parameter tweaking for *island* and *wa-*

Table 3. Detailed structure of the generators $\{G_n\}_{n=1}^{N-1}$. Specifically, "conv3d", "batchNorm3d" represent the 3D convolution layer and batch normalization layer respectively. IC_n^V denotes the number of channels within the input volume. We set $\mathrm{IC}_n^V = 3$ for $n = 1$ and $\mathrm{IC}_n^V = 4$ for $n > 1$. L is set to 5 or 7.

Index	Name	Kernel	Stride	Pad	Input ch.	Output ch.
1	Conv3d + BatchNorm3d + LeakyReLU (0.2)	3×3×3	1	1	IC_n^V	32
2	Conv3d + BatchNorm3d + LeakyReLU (0.2)	3×3×3	1	1	32	32
$[3, L-1]$	Conv3d + BatchNorm3d + LeakyReLU (0.2)	3×3×3	1	1	32	32
L	Conv3d	3×3×3	1	1	32	4

Table 4. Detailed structure of the generators G_N. Specifically, "conv2d" represent the 2D convolution layer. "InsNorm2d" denotes Instance Normalization.

Index	Name	Kernel	Stride	Pad	I/O	Input res	Output res
1	Upsampling layer	-	-	-	3/3	160×160	320×320
2	Conv2d + InsNorm2d + LeakyReLU (0.2)	3×3	1	1	32/16	320×320	320×320
3	Conv2d + InsNorm2d + LeakyReLU (0.2)	3×3	1	1	16/8	320×320	320×320
4	Conv2d + InsNorm2d + LeakyReLU (0.2)	3×3	1	1	8/8	320×320	320×320
5	Conv2d	3×3	1	1	8/3	320×320	320×320

Table 5. Detailed structure of the generators $\{D_n^{2D}\}_{n=1}^{N}$. Specifically, "conv2d" represent the 2D convolution layer. IC_n^I denotes the number of channels within the input for D_n^{2D}. We set $\mathrm{IC}_n^I = 4$ for $n < N$ and $\mathrm{IC}_n^V = 6$ for $n = N$. L is set to 5 or 7.

Index	Name	Kernel	Stride	Pad	Input ch.	Output ch.
1	Conv2d + LeakyReLU (0.2)	3×3	1	0	IC_n^I	32
2	Conv2d + LeakyReLU (0.2)	3×3	1	0	32	32
$[3, L-1]$	Conv2d + LeakyReLU (0.2)	3×3	1	0	32	32
L	Conv2d	3×3	1	0	32	1

Table 6. Training parameters for SinGRAV on different datasets. L denotes the number of layers in G_n or D_n^{2D}.

Datasets	L	Res. of $\tilde{\mathbf{V}}_1$ / $\tilde{\mathbf{x}}_1$	Res. of $\tilde{\mathbf{V}}_{N-1}$ / $\tilde{\mathbf{x}}_{N-1}$	Final image Res.	batch size for $\tilde{\mathbf{x}}_n$ at 6 scales	batch size for $\tilde{\mathbf{x}}_n^*$ at 6 scales
Stonehenge, *grass and flowers*, *mushroom*, *crystal*, *candies*, *volcano*	7	40⁄32	126⁄160	320	$[6, 6, 6, 5, 2, 2]$	$[2, 2, 2, 1, 1, 1]$
Island	5	40⁄25	126⁄160	320	$[6, 6, 6, 6, 3, 2]$	$[2, 2, 2, 1, 1, 1]$
Water lily	7	53⁄32	156⁄160	320	$[6, 6, 4, 2, 1, 1]$	$[2, 2, 2, 1, 1, 1]$

Fig.13. Random scene samples from SinGRAV trained on real-world data. At each row, we show two views of the input scene on the left, followed by three diverse scenes generated by SinGRAV on the right. Each scene is rendered with the same two cameras.

ter lily does not bring significant gains, instead, it just slightly improve the results.

Sampling strategy for volume rendering. In the rendering process, we adopt a uniform sampling strategy, *i.e.*, we uniformly sample a number of points within the range of the *near* point and *far* point along each ray cast from each pixel. Considering the increased fineness of textures within the generated scene at different scales, we progressively enlarge the number of sampling. Specifically, the number of sampling points start from 64 at scale 1 and reaches 128 at the finest scale.

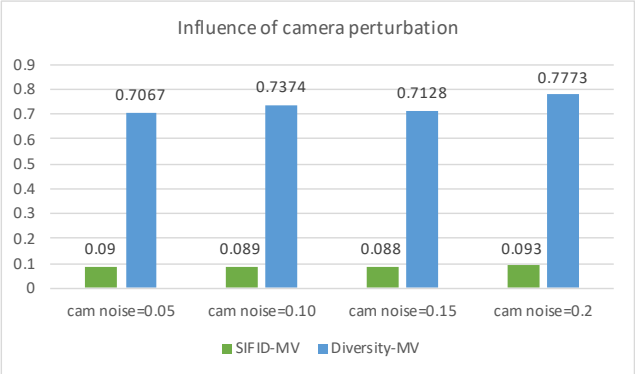

Fig.15. Performance for SinGRAV trained on data obtained with perturbed camera poses. Within a reasonable perturbation range (±20 cm), the performance is stable, denoting that the SinGRAV is robust to the reasonable perturbations for camera pose distribution during the data acquisition.

6.3 Additional experimental results

In this section, we investigate SinGRAV on two more complicated outdoor scenes in 6.3.1 and investigate the robustness of the framework to the underlying camera pose distribution for the multi-view observations used for training in 6.3.2.

6.3.1 Results on real-world outdoor scenes

In this section, to further investigate the capacity of SinGRAV, we provide the results for SinGRAV on two more complicated real scenes, which are collected from Google Earth[5], namely *Devil Tower* and *Bagana Volcano*, respectively.

The first row and the second row in Table 7 show that the numerical results on real-world scenes approaches those demonstrated in Table 2. This suggests that SinGRAV also produces reasonable results on more complex real-world scenes. On the qualitative perspective, we notice that, while our method produces plausible generation visual results, the fine textures of the real-world scene indeed challenge the proposed voxel-based framework as also discussed in the main paper. Note these real-world results are obtained using the default hyper-parameter setting, without ad hoc fine-tuning.

In addition, we also investigate the influence of the depth supervision on one of the above scenes – *Bagana Volcano*, and report the numerical results in the third

[5]https://earth.google.com

Fig.14. Influence of the depth supervisions on the real-world scene. Two generated scenes are provided for SinGRAV and three results are provided for SinGRAV (wo. depth sup.). Red boxes highlight the implausible geometry – holes in the results of SinGRAV (wo. depth sup.), that are caused by the lack of depth supervision. Please note that for the purpose of conducting this experiment, we completely remove depth supervision. However, it is worth mentioning that the reconstructed depth of the input scene can still be obtained and utilized to enhance the quality.

Table 7. Numerical results on real-world data.

Variant	Data	SIFID-MV ↓	Diversity-MV ↑
SinGRAV	Devil Tower	0.1940	0.5801
SinGRAV	Bagana Volcano	0.1480	0.7359
SinGRAV (wo. depth sup.)	Bagana Volcano	0.1710	0.8391

row of Table 7. It can be seen that the numerical results drop slightly, which is similar to the trend shown in Table 2. Again, despite the slight drop in numerical results, totally eliminating depth supervisions on this real-world data also drastically affects the plausibility of the spatial arrangements in the results, which can be observed from the qualitative results shown in Figure 14. Nevertheless, the depth data obtained with COLMAP in our default setting is sufficient for producing reasonable geometric structures in the results, as also stated in the main paper.

6.3.2 Robustness to camera distribution

In real-world scenarios, it is likely that the cameras would deviate from the planned trajectories to some extent, i.e., the hemisphere camera distribution may not be guaranteed under a less controlled setting. Nevertheless, we shall show that SinGRAV does NOT necessitate an absolutely strict distribution for the training cameras, and can work properly if two criteria can be met: a) *high observation completeness*, which suggests the scene of interest needs to be covered as much as possible, and is a reasonable assumption with the prevalent "render-and-discriminate" framework; b) *consistent multi-scale internal distribution*, which means the captured images need to share a similar internal distribution at each scale for learning the generative model.

To account for such factors, we investigate the robustness of SinGRAV under such less controlled environments by simulating with perturbed camera poses. Specifically, we randomly add noise drawn from a Gaussian distribution with different standard deviations to the camera poses during the multi-view images acquisition. We conduct the study on *Stonehenge*, of which the results are demonstrated in Figure 15.

Figure 15 shows that the scores for SIFID-MV and Diversity-MV stay similar when the scale of the Gaussian noise changes within a range of $[-20, +20]$ cm. Experimental results show that SinGRAV is robust to adequate perturbations of the camera poses, which means that SinGRAV just needs the camera poses to come from a roughly satisfying distribution rather than strictly demands that the camera poses come from exactly controlled settings.

6.4 Detailed training settings for baselines

For training the baseline methods, we keep most of the parameters the same as those provided in the exemplar configurations. The parameters we modified according to our settings are listed below:

GRAF [4]: On each dataset, we train more than 4500 epochs until convergence. The patch size used

during training is set to 32. Due to the patch based training process, it supports to be trained at 320×320 resolution and convergences slower.

pi-GAN [1]: We train 685 epochs on each dataset and adopt the progressive training strategy as in [1]. Specifically, we train 150 epochs with batch size $= 30$ at image resolution 32×32, 270 epochs with batch size $= 12$ at image resolution 64×64 and 265 epochs with batch size $= 8$ at image resolution 128×128.

GIRAFFE [5]: We set the parameters according to the instructions provided in the official implementation [6]. Specifically, we set the number of object field to 1 and use a background field. For training on our datasets, we separately set the parameters for the foreground field in the following manner:

- *Stonehenge*: The scaling range is set to $[0.45, 0.55]$ and the translation range is set to $[0.0, 0.0]$ for x-axis, y-axis and z-axis.
- *Grass and flowers*: The scaling range is set to $[0.7, 0.8]$ and the translation range is set to $[0.0, 0.0]$ for x-axis, y-axis and z-axis.
- *island*: The scaling range is set to $[0.45, 0.55]$ for x-axis, y-axis and z-axis. And the translation range for x-/y- axis is set to $[-0.15, 0.15]$, while the translation range for z-axis is set to $[0.0, 0.0]$.

Besides, for training with GIRAFFE [5] method, we let the low-res image output from the volume rendering have a resolution of 20×20, so that the resolution of the super-resolved image reaches 320×320. We train the framework GIRAFFE for more than 1.3 million iterations.

[6] https://github.com/autonomousvision/giraffe