GANcraft: Unsupervised 3D Neural Rendering of Minecraft Worlds Supplementary Material

A. Supplementary video

Our project website is available at https://nvlabs.github.io/GANcraft/. This includes an overview of the method as well as additional results.

We also provide a video, including more visual results and discussion of our work. Specifically, it contains:

- Additional high-resolution video results rendered at 1024×2048 pixels and 30 frames per second
- Style interpolation results
- Additional comparisons with baseline methods
- Illustration of the proposed approach.

Please make sure to check it out at https://www.youtube.com/watch?v=1Hky092CGFQ.

B. Method details

Here, we provide additional details of our approach.

B.1. Numerical volumetric rendering

The integral in Equation 2 of the main paper can be approximated with discrete samples via quadrature [39]. Assume that we sample $N+1$ points at $t_1, ..., t_{N+1}$ along a camera ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{v}$. We define

$$\begin{aligned}\delta_i &= t_{i+1} - t_i,\\ \hat{t}_i &= \frac{t_{i+1} + t_i}{2},\\ \sigma_i &= \sigma\left(\mathbf{r}\left(\hat{t}_i\right), l\left(\mathbf{r}\left(\hat{t}_i\right)\right)\right),\\ \mathbf{c}_i &= \mathbf{c}\left(\mathbf{r}\left(\hat{t}_i\right), l\left(\mathbf{r}\left(\hat{t}_i\right)\right), \mathbf{z}\right),\end{aligned}$$

such that

$$C(\mathbf{r}) \approx \left\{\sum_{i=1}^{N} T_i(1-\exp(-\sigma_i\delta_i))\mathbf{c}_i\right\} + T_{N+1}\mathbf{c}_{\text{sky}}(\mathbf{v}, \mathbf{z}),$$
$$\text{where } T_i = \exp\left(-\sum_{j=1}^{i-1}\sigma_j\delta_j\right).$$

B.2. Point sampling algorithm

In this section, we describe the method we use to efficiently sample points from the sparse voxel grid along a camera ray. Instead of relying on rejection sampling (as in Liu *et al*. [31]) to remove points that have not landed inside any voxel, we first traverse the voxel grid along the ray to obtain the entrance and exit points of each valid voxel that the ray has gone through, and then sample points only on the segments that are inside voxels.

For voxel grid traversal, We implement a 3D version of Bresenham's line algorithm [5], which has a very low computational cost of $O(N)$, where N is the longest dimension of the voxel grid. Its working principle is as follows: Starting from the voxel position where the camera resides, for each step, we traverse to the next voxel which is adjacent to the current voxel by the face which the ray exits from.

B.3. Network Architecture

GANcraft contains 6 trainable neural networks. Here are their descriptions and their respective network architectures:

Per-sample MLP. This is the MLP for representing the implicit radiance field, in conjunction with the voxel features. The network architecture is illustrated in Fig. 6. We condition the output feature on the style code via weight modulation [25]. The detailed implementation of weight modulation is shown in Fig. 10.

Neural sky dome. The sky is modeled with an MLP (Fig. 7) which takes ray direction (represented as a normalized 3D vector) input and produce the color feature for that ray. The network is also conditional on the style feature.

Image space renderer. This is a CNN for converting feature map to RGB image (Fig. 8). As discussed in the main paper, we use very small kernel sizes to reduce the receptive field in order to encourage view consistency. The network is conditional on the style feature.

Style network. Following StyleGAN2 [25], we use an MLP that is shared across all the style conditioning layers to convert the input style code to an intermediate style feature. Its architecture is shown in Fig. 9.

Style encoder. The style encoder is a CNN that predicts the style code given an image. In conjunction with pseudo ground truth and reconstruction loss, this allows GANcraft to produce images that follows the style of a given image. Our style encoder is taken from SPADE [50], which is a 6-layer CNN followed by a linear layer and VAE reparameterization. Please refer to the original paper for the details.

Label-conditional discriminator. The discriminator we use is based on feature pyramid semantics-embedding (FPSE) discriminator [35]. Its construction is shown in Fig. 11. Compared to the patch discriminator used in SPADE [50], the FPSE discriminator is more robust to the distribution mismatch in the label map domain. A patch discriminator which takes the concatenated image and label map as input sometimes lead to training collapse almost immediately after the training starts.

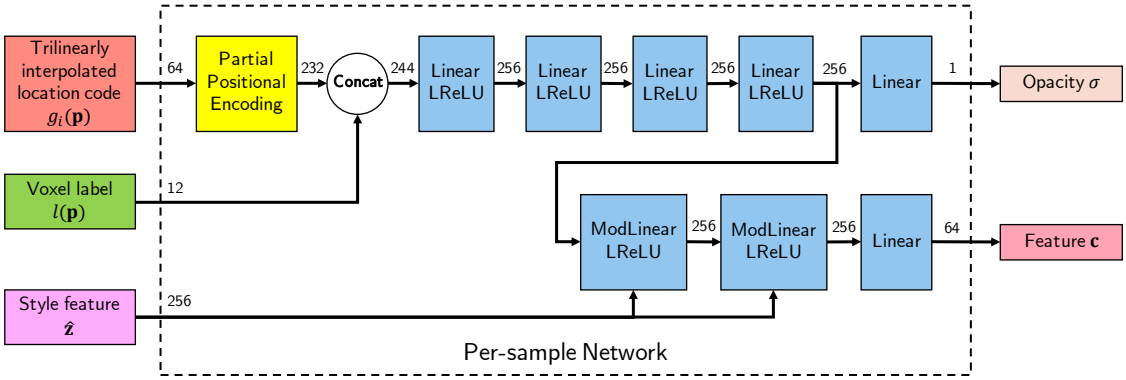

Figure 6: Per-sample MLP for representing the implicit radiance field in conjunction with the voxel features. We use weight modulation to condition the output feature $\mathbf{c}$ on the style feature. This is more computationally efficient than doing affine modulation on the per-layer feature when the same style is applied to a large number of samples. The number on each arrow denotes the number of channels. As a means of conserving the memory, we use partial positional encoding on the location code, which performs positional encoding only on the first 24 channels, and concatenate the result with the remaining 40 channels.

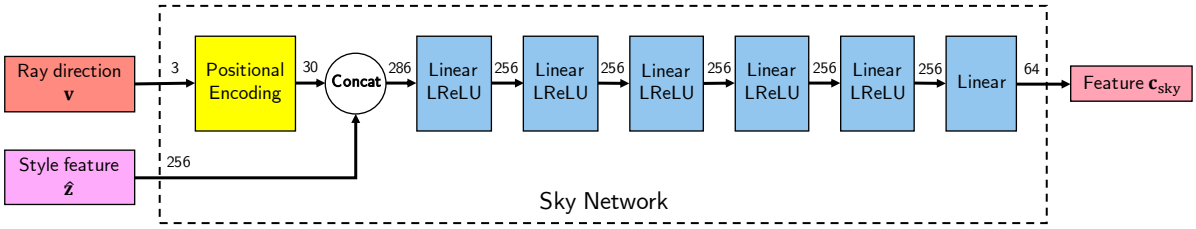

Figure 7: Network architecture for the neural sky dome MLP. The input ray direction is represented as a normalized 3D vector. The numbers on the arrows denote the number of channels.

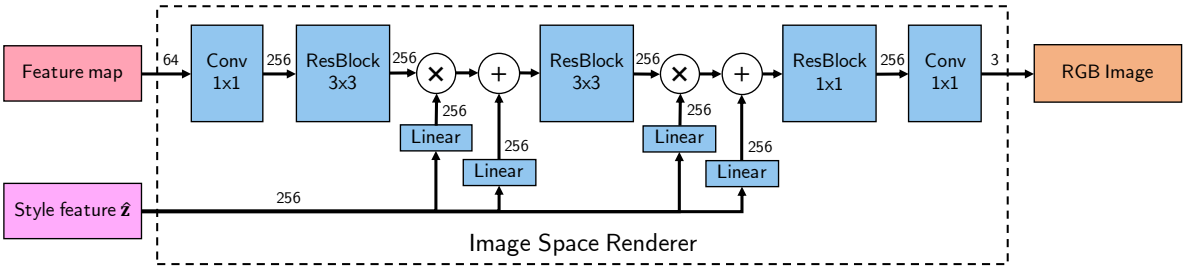

Figure 8: Network architecture for the image space renderer. The kernel sizes are shown inside each block and the channel counts are displayed on the arrows. We apply Leaky ReLU after each Conv block, inside ResBlocks and after the affine modulations. We use hyperbolic tangent activation for generating the final image (omitted here for clarity).

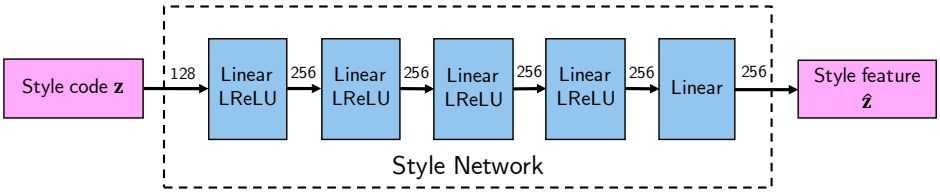

Figure 9: The architecture of style network. Following StyleGAN2 [25], we use a common MLP that is shared across all the style conditioning layers to convert the input style code to an intermediate style feature.

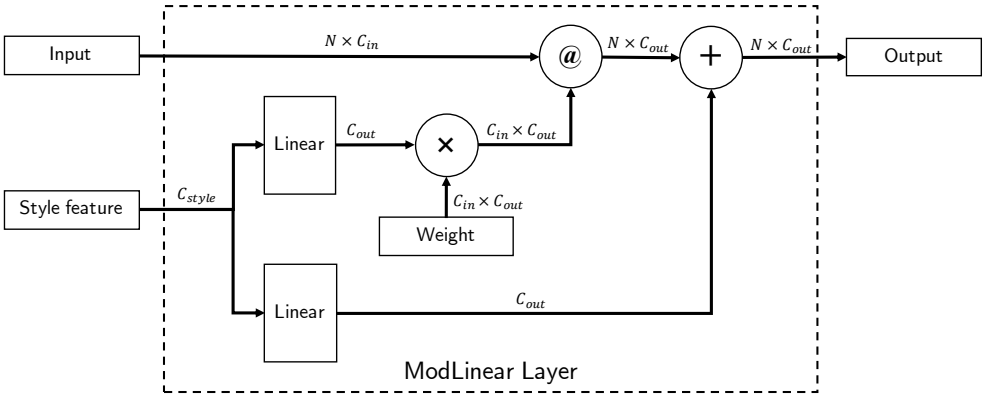

Figure 10: Detailed structure of ModLinear layer used in the per-sample network (Figure 6). '@' denotes matrix multiplication. Shapes of intermediate tensors are denoted on the arrows. The batch dimension is omitted for clarity.

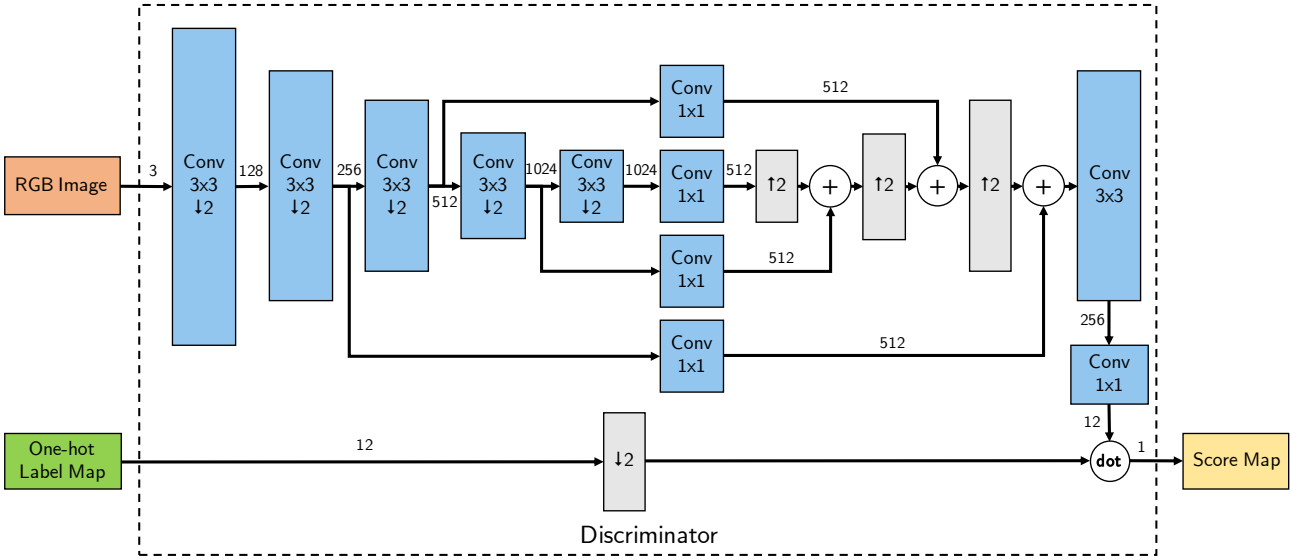

Figure 11: The conditional discriminator used in GANcraft. 'dot' denotes dot product on the channel dimension. '↓ 2' denotes downsample by 2. '↑ 2' denotes upsample by 2. We use bilinear interpolation for upsampling, and stride 2 convolution for downsampling. For label map, we downsample it via nearest neighbor interpolation. We use spectral normalization [40] on all the convolution layers in the discriminator.

B.4. Label Translation

There is significant difference between Minecraft voxel labels, which we use as the starting point of GANcraft, and COCO-Stuff [10] labels, which is the format the pretrained DeepLabV2 model produces and the pretrained SPADE model accepts. In Minecraft Java edition, there are 680 labels in total, mostly describing raw materials (dirt, sand, log, water, etc.) useful for building objects. While in COCO-Stuff, there are 182 higher level labels of common objects such as mountain, tree, river, and sea. Due to the drastic difference in the level of abstraction, it is very difficult to find a one-to-one mapping between Minecraft label and COCO-Stuff label. For example, the water material in Minecraft can be mapped to either sea or river label in COCO-Stuff; the tree material consists of both log and leaf label in COCO-Stuff. We solve the labeling difference in two ways. For the label-conditional discriminator, we introduce a new set of 12 classes with high level of abstraction: *ignore, sky, tree, dirt, flower, grass, gravel, water, rock, stone, sand, and snow*. We then classify every Minecraft and COCO-Stuff label into one of the 12 classes, and use the translated semantic segmentation mask as the conditional input to the discriminator. For generating pseudo-ground truth, however, we will have to convert Minecraft labels to COCO-Stuff labels in order to be recognized by the pretrained SPADE generator. We achieve this by first translating the Minecraft labels to one of the 12 labels, and then map them to COCO-Stuff labels randomly, with equal chance across all the candidate labels. Note that we use the same mapping scheme within a segmentation map. We are able to obtain good result from such a simple measure, as the style encoder is able to explain away the

Figure 12: **Bird's-eye view of the 5 Minecraft worlds used.** Each block is color-coded by its label (brown-sand, blue-water, light green-grass, dark green-trees, white-snow, *etc*.). We use worlds with varying distributions of sand, forest, water, snow, trees, grass, *etc*. The label distribution of each specific world is very different from that of a collection of real images, *e.g*. the first world is >50% sand, and the second is >90% water. Our method works for all these worlds despite the domain gap, indicating the robustness of our framework.

Figure 13: **Outputs of the ablated model that does not use pseudo-ground truths.** This model was trained only with a GAN loss between the outputs and real images, and obtains low FID and KID values, as seen in Table 4. However, the output images look unrealistic and do not learn the correct correspondence between input segmentation labels and realistic textures.

Figure 14: **Blockiness in some outputs.** Certain regions and objects appear blocky due to the underlying blocky geometry that is very different from occurrences in the real world.

randomness in the mapping.

B.5. Voxel Preprocessing

Minecraft voxel world has a sea level of 62, below which most of the voxels are not visible from above. It will be a waste of memory if we still assign voxel features to those invisible voxels. Thus we preprocess the voxel by removing the interior voxels, leaving a 4 voxel thick thin shell. This operation reduces the occupancy of a typical voxel world from 28% to 3%. The effect of preprocessing can be seen at the borders of the voxel worlds in Fig. 12. Note that the preprocessing step is not only useful for Minecraft world. It is applicable to any types of voxel grids.

C. Experiment details

C.1. Minecraft block worlds.

We use 5 different Minecraft worlds for our experiments. An overview of these worlds is shown in Fig. 12. As can be seen, the label distribution of each specific world is very different from that of a collection of real images, *e.g*. the first world is >50% sand, and the second is >90% water. Our method works for all these worlds despite the domain gap, indicating the robustness of our framework.

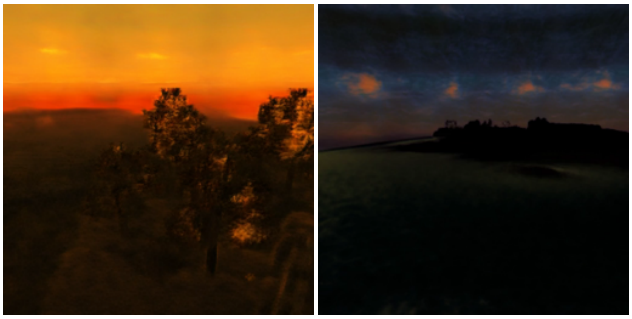

Figure 15: **Incompatible styles.** Certain combinations of styles and worlds give unrealistic outputs, possibly as these styles are outliers.

C.2. GANcraft settings

During training, we generate images at a resolution of 256×256. We sample 24 points along each ray, and truncate the rays to a maximum distance of 3 (distance traveled outside voxels doesn't count). We use a learning rate of 1e-4 for the generator networks, and 4e-4 for the discriminator. For voxel features, we use a higher learning rate of 5e-3. We use a combination of GAN loss, L_2 loss, L_1 loss and perceptual loss, with their weights being 1.0, 10.0, 1.0 and 10.0, respectively. For regularization terms, we use a weight of 0.5 for the opacity regularization, and a weight of 0.05 for the KL divergence needed by the style encoder. We also clip the per-sample feature $\mathbf{c}$ to a range of $[-1, 1]$ before blending to reduce the ambiguity between the opacity and the scale of feature. For random camera pose sampling, we sample two points that are slightly above ground, and use one of the as the camera location and the other one as the point that the camera looks at. We reject any camera pose that produces a depth map with a mean depth below 2 or that produces a segmentation mask with label entropy below 0.75. This guarantees that the segmentation mask along can provide enough scene geometry hint to the SPADE generator for generating a pseudo-ground truth that corresponds well to the actual scene geometry.

During evaluation, we increase the sample count to 32 points per ray. On an NVIDIA Titan V, this takes approximately 10 seconds to render a 1024×2048 frame.

C.3. Baseline settings

For fair comparison, the settings used in the NSVF-W baseline largely resembles GANcraft except for the following differences:

- Only L_2 loss and KL divergence is used during training.
- The weight for KL divergence is reduced to 0.01 to avoid handicapping the style encoder too much in the absence of other reconstruction losses.
- The image space CNN renderer is removed, and the per-sample MLP directly produces an RGB radiance (clipped by a sigmoid function) instead of a feature.

D. Additional results

Method	FID ↓	KID ↓
Full model	78.79	0.043
No CNN	84.86	0.049
No real images	89.95	0.055
No GAN loss	104.58	0.073
No pseudo-ground truth	65.40	0.043

Table 4: **Ablation comparison on automated image quality metrics** (↓ indicates lower is better)**.** We compare ablated versions of our full method on a single block world.

D.1. Ablation study

Here, we present quantitative results for the ablated versions of our full model. Sample outputs from these ablations were shown in Fig. 5 of the main paper. We trained all ablations on one world only, due to computational constraints (each model takes 4 days on 8 NVIDIA V100 GPUs).

The results of automated metric evaluation as shown in Table 4. We computed the FID and KID values with 2000 images generated from random camera poses and 5000 held-out real images. As expected, all ablated versions obtain higher FID and KID scores indicating worse quality. An exception is the model trained without any pseudo-ground truth images, *i.e*. trained with GAN loss between outputs and real images only. Surprisingly, it obtains a lower FID and KID than our full model. However, when we visually inspect the outputs, shown in Fig. 13, it is clear that the model fails to learn a meaningful mapping from Minecraft segmentations to real images. The model seems to have learned to produce unrealistic images that optimize the metrics due to training with the GAN loss. However, similar to MUNIT [21], the outputs are both unrealistic and incorrectly map Minecraft segmentation labels to real images.

We observed that our method can fail in two ways — either producing blocky outputs or producing unrealistic outputs. In the input block world, all objects and regions are made of blocks. Due to this coarse geometry, the method is sometimes unable to learn realistic geometries in the translated world. As a result, boundaries can often appear jagged, as shown in Fig. 14. Further, certain combinations of worlds and style-conditioning images can produce unrealistic outputs as shown in Fig. 15. For example, a forest world paired with a conditioning image of a red sunset can produce unrealistic, or overly dark outputs. As the style encoder is trained exclusively with pseudo-ground truth images that have the same label distribution as the rendered Minecraft images, it has never encountered such combinations.