

Lightplane: Highly-Scalable Components for Neural 3D Fields

Anonymous 3DV submission

Paper ID 170

A. Supplementary Videos

Please watch our attached video for a brief summary of the paper and more results. We include more generated results and 360-degree rendering videos from our reconstructed and generated 3D structures to show their 3D consistency.

B. Social Impact

Our main contribution is *Lightplane Splatter* and *Renderer*, a pair of 3D components which could be used to significantly scale the mapping between 2D images and neural 3D fields. Beyond their integral role in our versatile pipeline for 3D reconstruction and generation, single scene optimization, and LRM with *Lightplane*, these components can also function as highly scalable plug-ins for various 3D applications. We earnestly hope that they will be instrumental in advancing future research.

Based on *Lightplane Splatter* and *Renderer*, we have established a comprehensive framework for 3D reconstruction and generation. Similar to many other generative models [5, 13, 23], it is important to note that the results generated by this framework have the potential to be used in the creation of synthetic media.

C. Limitations & Discussions

Our motivation of introducing contract coordinates [2] is to assist the model in differentiating between foreground and background elements, thereby enhancing the quality of foreground generation and reconstruction. Although contract coordinates could represent unbounded scenes, our main focus is still on foreground objects, and reconstructing or generating unbounded backgrounds is beyond the scope of this paper. Therefore, we only sample limited points in unbounded regions, which leads to floaters, blurriness and clear artifacts in the background, as can be observed in videos. Also, generating diverse and realistic backgrounds is a challenging task and we leave it as a promising future direction.

Lightplane introduces a versatile approach for scaling the mapping between 2D and 3D in neural 3D fields, designed to be compatible with arbitrary 3D hash represen-

tations with differentiable sampling functions. While our validation of this design has focused on voxel and triplane models, its adaptability should allow for easy generalization to other 3D hash representations, such as Hash Table [1] or HexPlane [3, 8]. We pick voxel grids and triplanes as their structures are easy to be processed by the existing neural networks while designing neural networks to process some other 3D hash structures like hash tables is still an open question. Developing neural networks to support other 3D hash structures is a promising direction to explore while beyond the scope of this paper.

Lightplane significantly solves the memory bottlenecks in neural 3D fields, making rendering and splatting a large number of images possible in the current 3D pipelines. Although *Lightplane* has comparable speed to existing methods, rendering and splatting a large number of images is still time-consuming, which may limit its utilization in real applications. For example, doing a forward and backward pass on 512×512 rendered images takes around 5 seconds for each iteration. For *Renderer*, the spent time grows linearly to the ray numbers when ray numbers are huge. Reducing the required time for large ray numbers would be a promising direction.

Sadly, we observe a performance gap between different 3D hash representations (*i.e.*, voxel grids and triplanes) in the versatile 3D reconstruction and generation framework. Without loss of generalization, we use 3D UNet to process voxel grids and 2D UNet to process Triplane. Three planes (XY, YZ, ZX) are concatenated into a single wide feature map and fed to 2DUNet. The self-attention mechanism is then applied across all patches from the three planes, making this network an extension of our 3DUNet designed for voxel grids. However, we observed that this neural network configuration does not yield flawless results. In 3D reconstruction tasks, the images rendered at novel viewpoints exhibit slight misalignments with the ground-truth images. For generative tasks, while the network can produce realistic samples, it occasionally generates flawed outputs that significantly impact the Fidelity (FID) and Kernel Inception Distance (KID) scores. Developing a more efficacious neural network model for TriPlane processing [1, 4, 24], which

could effectively communicate features from three planes, presents a promising avenue for future research.

D. Lightplane Details

D.1. Implementation Details

Starting from a zero-initialized hash θ , *Splatter* is done by accumulating \mathbf{v}_{ij} to the hash cell (*i.e.* voxel grids or triplanes) that contain \mathbf{x}_{ij} , using the same trilinear/bilinear weights used in the *Renderer* operator to sample θ . After accumulating over all M rays, each hash cell is normalized by the sum of all splatting bi/trilinear weights landing in the cell. The normalization operation employed in our method, analogous to average pooling, averages the information splatted at identical positions in the hash θ . This process guarantees that the magnitudes of the splatted features are comparable to those of the input view features, a factor that is beneficial for the learning process.

In the actual implementation, we execute the splatting process twice within the *Splatter* kernel. Initially, we splat the features of the input image into θ . Subsequently, a second set of weight maps is created, matching the spatial dimensions of the input image features, but with a feature of a single-scale: 1. These weight maps are then splatted into θ_{weight} . During the second splatting process within the *Splatter* kernel, we deactivate the Multilayer Perceptrons (MLPs) and suspend sampling from prior hash representations. This modification is implemented because our objective is to tally the frequency and weights of points being splatted into the same position within the hash representations, instead of learning to regress features. Finally, we get θ/θ_{weight} .

Performing the splatting operation twice inevitably results in additional time and memory overhead. In practice, θ_{weight} is relatively lightweight while θ is more memory-intensive. This is because they have the same spatial shape while θ_{weight} has a feature dimension of only 1. The normalization step θ/θ_{weight} , which is implemented in PyTorch, will cache the heavy θ , thereby increasing memory usage. We manually cache θ_{weight} to normalize gradients during backpropagation.

Algorithm 1 *Lightplane Splatter* for Triplane

Input: Prior (=input) 3D structure $\hat{\theta}$, rays \mathbf{r} , pixel features \mathbf{v}

Output: Target 3D hash θ (zero-initialized)

```

for  $i$  in range(num_rays):                                ▷ Parallel spawned kernels
  for  $j$  in range(num_ray_points):                          ▷ Ray-marching inside the kernel
     $\mathbf{x}_{ij} \leftarrow \text{get\_3d\_ray\_point}(\mathbf{r}_i, j)$            ▷ Get  $j$ -th 3D point on  $i$ -th ray
     $h_{\hat{\theta}}(\mathbf{x}_{ij}) \leftarrow \text{sample}(\mathbf{x}_{ij}, \hat{\theta})$          ▷ Sample from input  $\hat{\theta}$ 
     $\tilde{\mathbf{v}}_{ij} \leftarrow g_s(\mathbf{v}_i, h_{\hat{\theta}}(\mathbf{x}_{ij}), \text{direnc}(\mathbf{r}_i))$    ▷ Eq(2): MLP  $g_s$  scales  $\tilde{\mathbf{v}}_{ij}$ 
     $\theta \leftarrow \text{splat}(\mathbf{x}_{ij}, \tilde{\mathbf{v}}_{ij}, \theta)$                  ▷ Splat and update  $\theta$ 

```

Algorithm 2 Splatting point to XY Plane: $\text{splat}(\mathbf{x}, \mathbf{v}, \theta_{XY})$

Input: $\mathbf{x}=(x, y, z) \in [0, X] \times [0, Y] \times [0, Z]$, feature \mathbf{v} , 2D plane θ_{XY}

```

 $x_f = \lfloor x \rfloor; x_c = 1 + x_f; y_f = \lfloor y \rfloor; y_c = 1 + y_f$     ▷ Ceiling, floor coords
 $w_{00}, w_{01}, w_{10}, w_{11} = \text{bilinear\_weights}(x, y, x_f, y_f, y_c)$ 
 $\theta_{XY}(x_f, y_f) += w_{00}\mathbf{v}; \theta_{XY}(x_c, y_f) += w_{10}\mathbf{v}$ 
 $\theta_{XY}(x_f, y_c) += w_{01}\mathbf{v}; \theta_{XY}(x_c, y_c) += w_{11}\mathbf{v}$ 

```

Experimental Details. We use $160 \times 160 \times 160$ voxel grids and 160×160 triplanes in our model. The input images are processed using a VAE-encoder [21] trained on the ImageNet dataset [7] and are converted into 32-dimensional feature vectors. Both the *Splatter* and *Renderer* components are equipped with 3-layer MLPs with a width of 64. Regarding training, we conduct 1000 iterations per epoch. The generative model is trained over 100 epochs, taking approximately 4 days, while the reconstruction model undergoes 150 epochs of training, lasting around 6 days, on a setup of 16 A100 GPUs, processing the entire Co3Dv2 dataset.

For *Splatter*, we sample 160 points along the ray. For *Renderer*, we sample 384 points along the ray, rendering 256×256 images. Instead of using original contract coordinates [2], we use a slightly different version which maps unbounded scenes into a $[-1, 1]$ cube.

$$\text{CC}(\mathbf{x}) = 0.5 * \begin{cases} a * \mathbf{x} & \|\mathbf{x}\| \leq 1 \\ \left((2-a) * (1 - \frac{1}{\|\mathbf{x}\|}) + a \right) \left(\frac{\mathbf{x}}{\|\mathbf{x}\|} \right) & \|\mathbf{x}\| > 1 \end{cases} \quad (1)$$

We introduce a scale a to control the ratio between foreground and background regions, where the foreground regions are mapped to $[-a/2, a/2]$. As we are using explicit 3D hash, mapping foreground regions into larger regions would be helpful to represent details. When $a = 1$, it becomes the normal contract coordinates. We convert X, Y, Z axes into contract coordinates independently.

D.2. Lightplane Performance Benchmark

Besides Autograd Renderer, implemented by pure Pytorch, we additionally compare *Lightplane Renderer* to two baselines: *Checkpointing* and *NerfAcc's Instant-NGP*, shown in Figure 5.

Checkpointing baseline applies the checkpointing technique in Pytorch to Autograd Renderer, which naive recalculates forward pass results during backward pass to save memories. Trivially applying checkpointing on Autograd indeed saves memories both in forward pass and backward pass, while still requires a large amount of memories, and cannot be used for large ray numbers.

NerfAcc's Instant-NGP is the Instant-NGP [18] implemented by NerfAcc [11], which is claimed to be $1.1 \times$ faster than the original version of Instant-NGP, with tremendous optimization tricks for speed. Instant-NGP combines hash grid as 3D structures with fused MLP kernels (tiny-cuda-dnn), which is different from our *Renderer* with triplanes

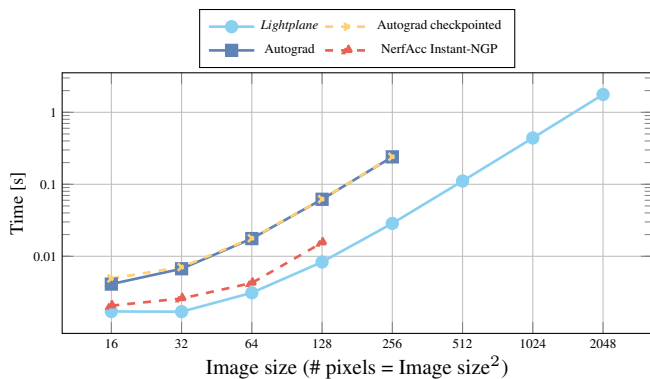


Figure 1. Forward (FW) Time.

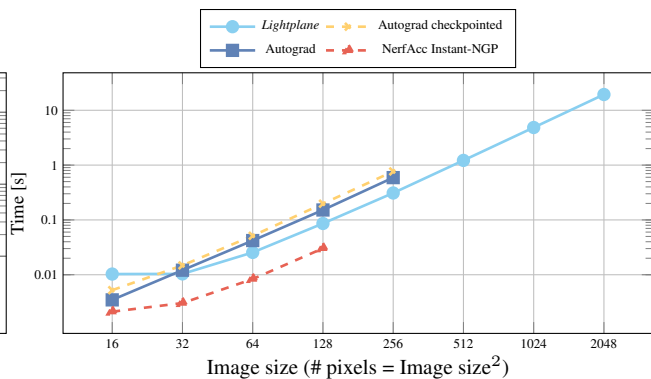


Figure 2. Backward (BW) Time

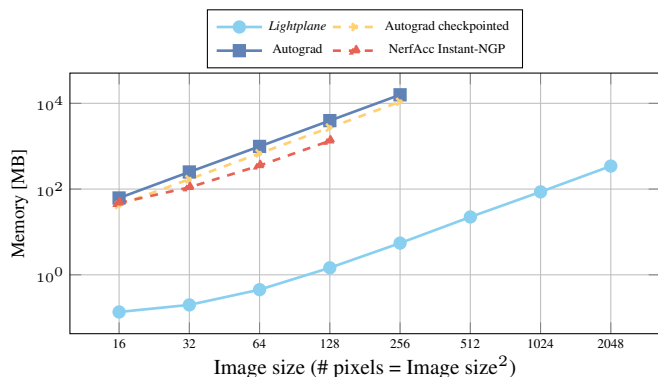


Figure 3. Forward (FW) Memory.

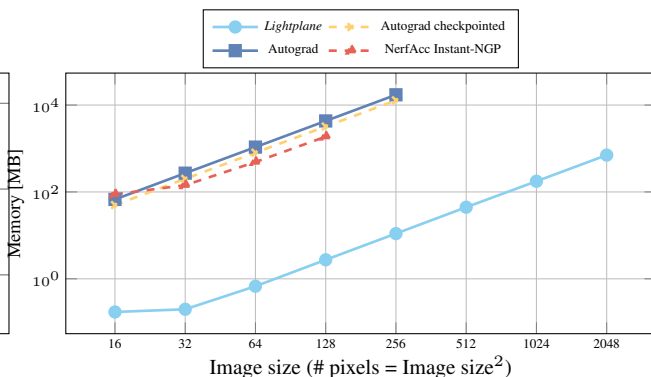


Figure 4. Backward (BW) Memory

Figure 5. **Lightplane Renderer memory & speed benchmark** showing the forward (FW and backward (BW) passes of *Lightplane Renderer*, compared to the *Autograd renderer*, *Checkpointing* (Pytorch checkpointing on *Autograd renderer*), and *NerfAcc Instant-NGP* (Instant-NGP [17] implemented in NerfAcc [12], which is claimed 1.1 \times faster than the original version) *Lightplane* exhibits up to 4 orders of magnitude lower memory consumption at comparable speed. All axes are log-scaled.

as 3D structures, and its internal settings are less flexible to change. To this end, it is hard to do a perfectly fair comparison. But still, we found that instant-NGP cannot work (will crash) with large image sizes, as they heavily rely on the L2 cache of GPUs for optimal speed, which memory is very limited and cannot support large image sizes. While their backward pass speed is significantly faster than *Lightplane Renderer*, it still cannot be extended to large output image sizes.

Deferred-Backpropagation [26]. *LightPlane* allows full forward (fw.) and backward (bw.) passes with superior memory use and speed. Conversely, to compute the gradients for image-level loss, *Deferred-Backpropagation* first computes the fw. on the full image in ‘inference-only’ mode, and then computes the per-pixel gradients using the full-image losses. Given the pre-computed gradients on the rendered images, *Deferred-Backpropagation* sequentially computes the memory-expensive fw. and bw. on a small subset of rays. E.g., rendering a 128 \times 128 image of a Triplane, DB/*Lightplane* uses 1088MB/3MB in 0.23/0.09 sec,

Table 1. Quantitative results on NeRF Synthetic dataset [16].

Method	PSNR \uparrow	SSIM \uparrow	LPIPS \downarrow
NeRF [15]	31.01	0.947	0.081
Plenoxels [25]	31.71	0.958	0.049
DVGO [22]	31.95	0.957	0.053
TensoRF-CP-384 [6]	31.56	0.949	0.076
TensoRF-VM-48 [6]	32.39	0.957	0.057
<i>Lightplane</i>	<u>32.12</u>	0.957	<u>0.050</u>

giving **400 \times** memory saving and **2.5 \times** speed-up.

E. More Results

E.1. Single Scene Optimization

Synthetic NeRF Results. We validate the correctness of *Lightplane* by overfitting on the Synthetic NeRF dataset, shown in Table 1. As the target is to show the convergence

of *Lightplane*, we don't employ any complicated tricks to optimize the performance and speed. *Lightplane* could get promising single-scene optimization results, demonstrating that it could be used as a reliable package in various 3D tasks.

Adversarial Attacking on LVM (Large Vision Model).

We showcase another interesting application empowered by our *Lightplane* by adversarial attacking LVM models, *e.g.* CLIP [19] and BLIP2 [10]. After rendering full images from the neural 3D field overfitted on a specific scene, we feed rendered images into CLIP model and calculate cosine similarity between image feature vectors and target text vectors, which similarity works as a loss to optimize the neural 3D fields.

E.2. Multi-view LRM with *Lightplane*

We show more results of Multi-view LRM with *Lightplane* in Figure 7 and Figure 8.

Rooted on the LRM [9], we plug *Lightplane* into the current system and achieve boosted performance. Specifically, after every-three transformer blocks, we use *Lightplane Splat* to splat the input images features into the existing triplanes (*i.e.* outputs of the transformer blocks). Thanks to the high efficiency of *Lightplane Renderer*, the whole system is trained by LPIPS loss on the rendered and ground-truth images.

E.3. 3D Reconstruction

We show amortized 3D reconstruction results after fine-tuning on a single scene in Figure 9, with voxel grids (*Lightplane-Vox*) and triplanes (*Lightplane-Tri*) as 3D structures. We compare them to overfitting results (training from scratch) using the 3D structures. Overfitting a single scene on Co3Dv2 dataset leads to defective 3D structures, like holes in depths. Initializing from the outputs of our amortized 3D reconstruction model could effectively solve this problem, leading to better results.

For details, *Lightplane-Vox* utilizes a 156^3 voxel grid as the 3D representation and applies a 3D-Conv UNet with attention-layer as the reconstructor. (*Lightplane-Tri*) utilizes 256^2 triplanes as the 3D representations, and uses a 2D-Conv UNet to process three triplanes. Both models take 20 views as inputs and calculate the loss on target views for supervision. The entire model is trained on 16 A100 GPUs for 3 days with a learning rate of $1e-5$. For rendering, we use 400×400 as our target resolution, with 256 number of points along the ray, taking 32 additional points for contract coordinates (background). For splatting, each pixel casts 156 points per ray, with additional 32 points for background.

E.4. Unconditional Generation

We show 360-degree rendering for unconditional generation in Figure 10 and Figure 11.

We follow the exactly the same network architecture as in the 3D reconstruction network, except we additionally take time steps as additional inputs. We follow the standard diffusion training receipts and apply 50 steps DDIM sampling during inference. The whole generation process takes less than 2 minutes.

E.5. Conditioned Generation

We show monocular 3D reconstruction with a single image as input in Figure 12, and text-conditioned generation in Figure 13. For text-conditioning experiments, we follow CAP3D [14]: we use BLIP2 [10] to generate captions of each image insides scenes and utilize LLAMA2 to output the comprehensive caption for the whole scene.

We apply the classification-free guidance (cfg) during the inference to control the quality, with the weights = 2.



Figure 6. **3D Adversarial Attacking on CLIP model.** Given a fitted 3D scene (1st and 3rd column), we optimize the neural 3D fields so that features of rendered images are aligned to a specific text description, *i.e.* giraffe, in CLIP’s feature space, while keeping the appearance perceptually the same.

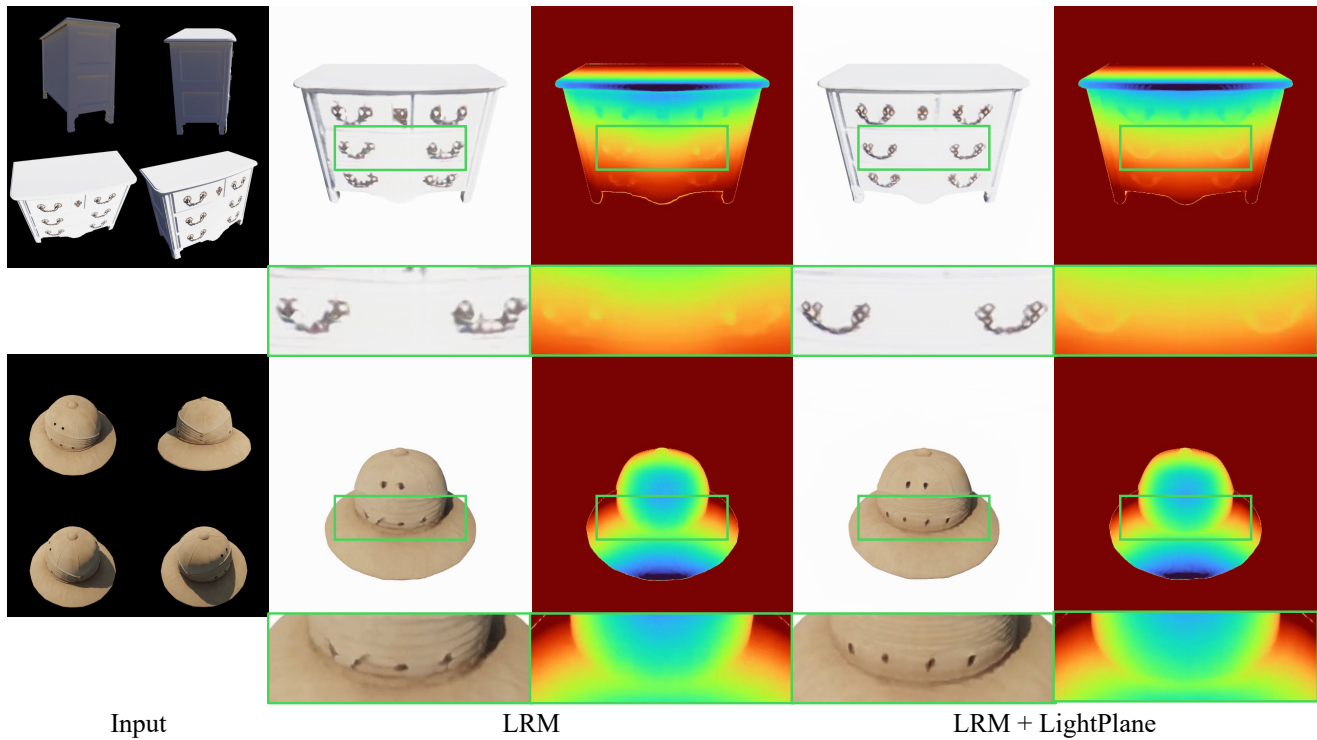


Figure 7. **Reconstruction comparison between LRM and LRM + *Lightplane*.**

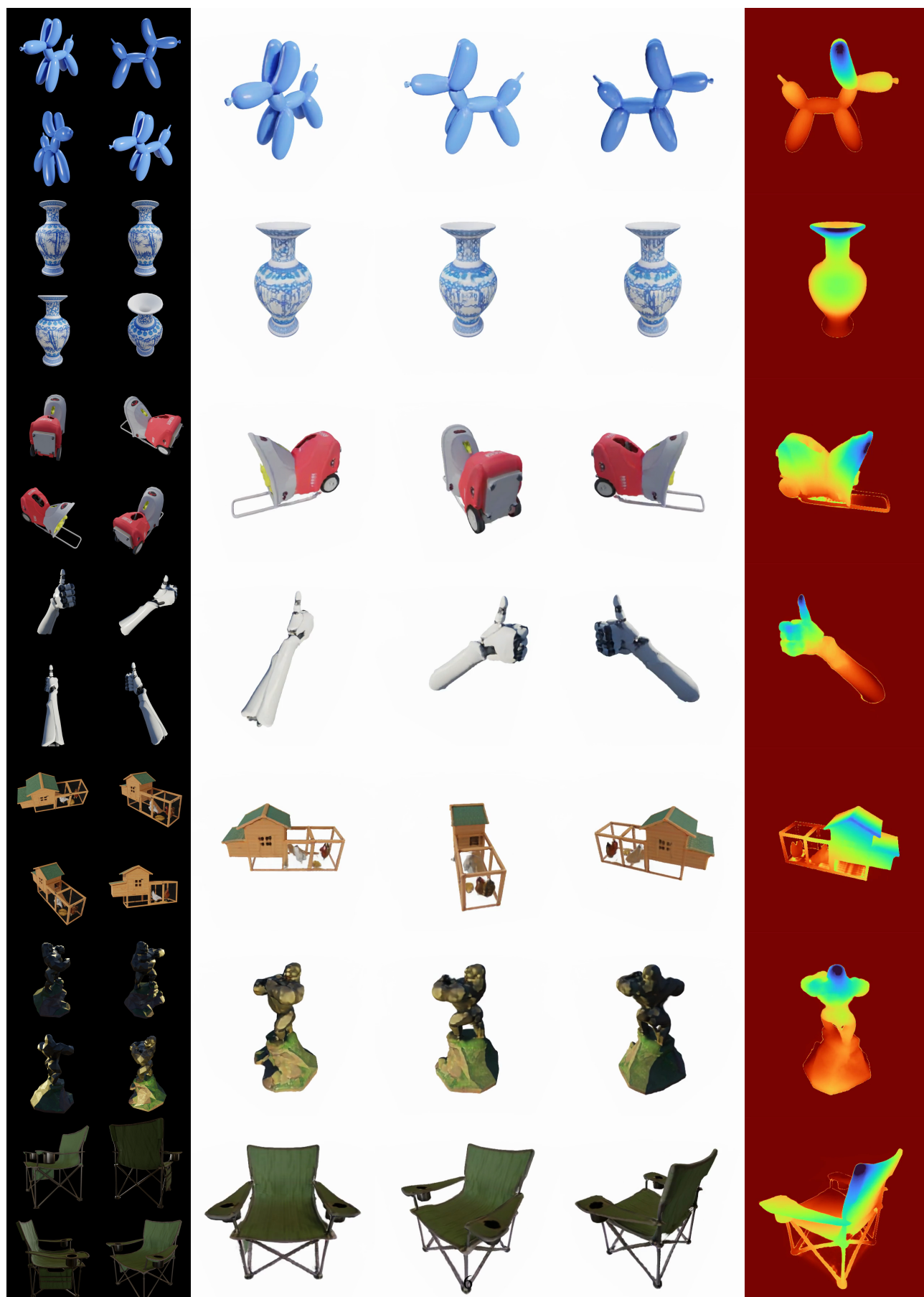


Figure 8. Multi-view LRM with *Lightplane*.

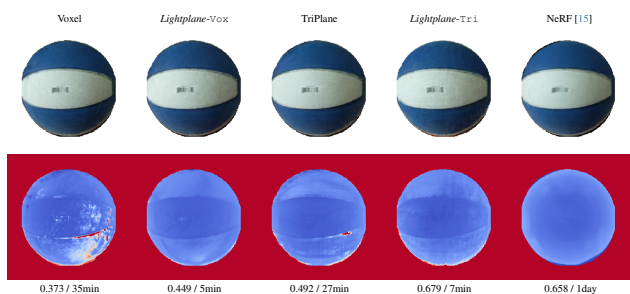


Figure 9. 3D Reconstruction with Learned Initialization. We show rendered images (top row) and depths. Optimizing hashed representations (Voxel, Triplane) on real scenes leads to geometric defects. Using our models (*Lightplane-Vox*, *Lightplane-Tri*), we first learn a reconstruction prior on CO3Dv2. We then initialize reconstruction with a feed-forward pass accepting up to 100 source views of a single-scene. After fine-tuning, we observe improved quality of the reconstructed geometry (columns 3 and 4). We show *Depth Corr.* (↑) and *Overfitting Time* (↓) below images.

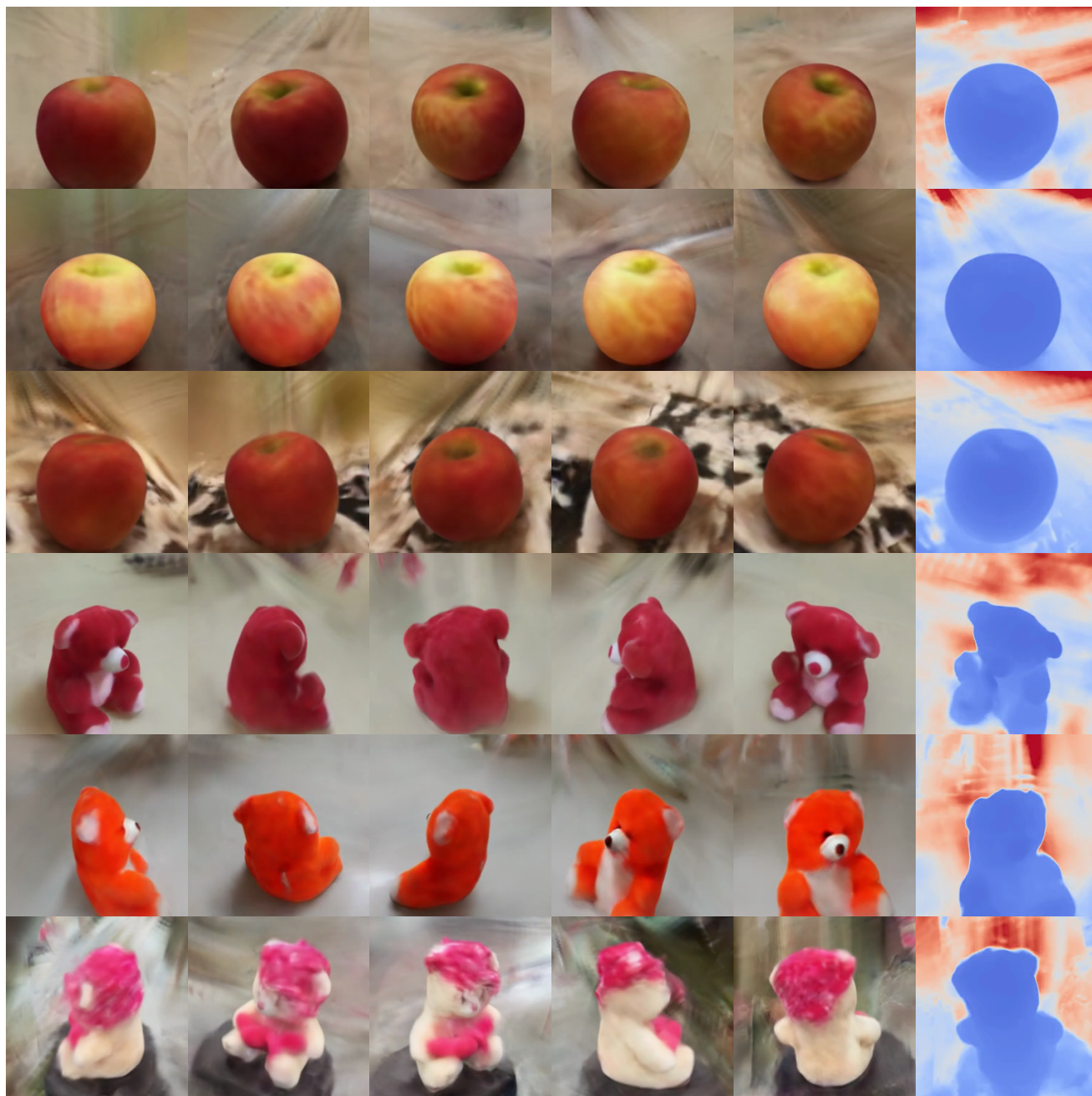


Figure 10. **Unconditional 3D Generation** displaying samples from our *Lightplane*-augmented Viewset Diffusion trained on CO3Dv2 [20].

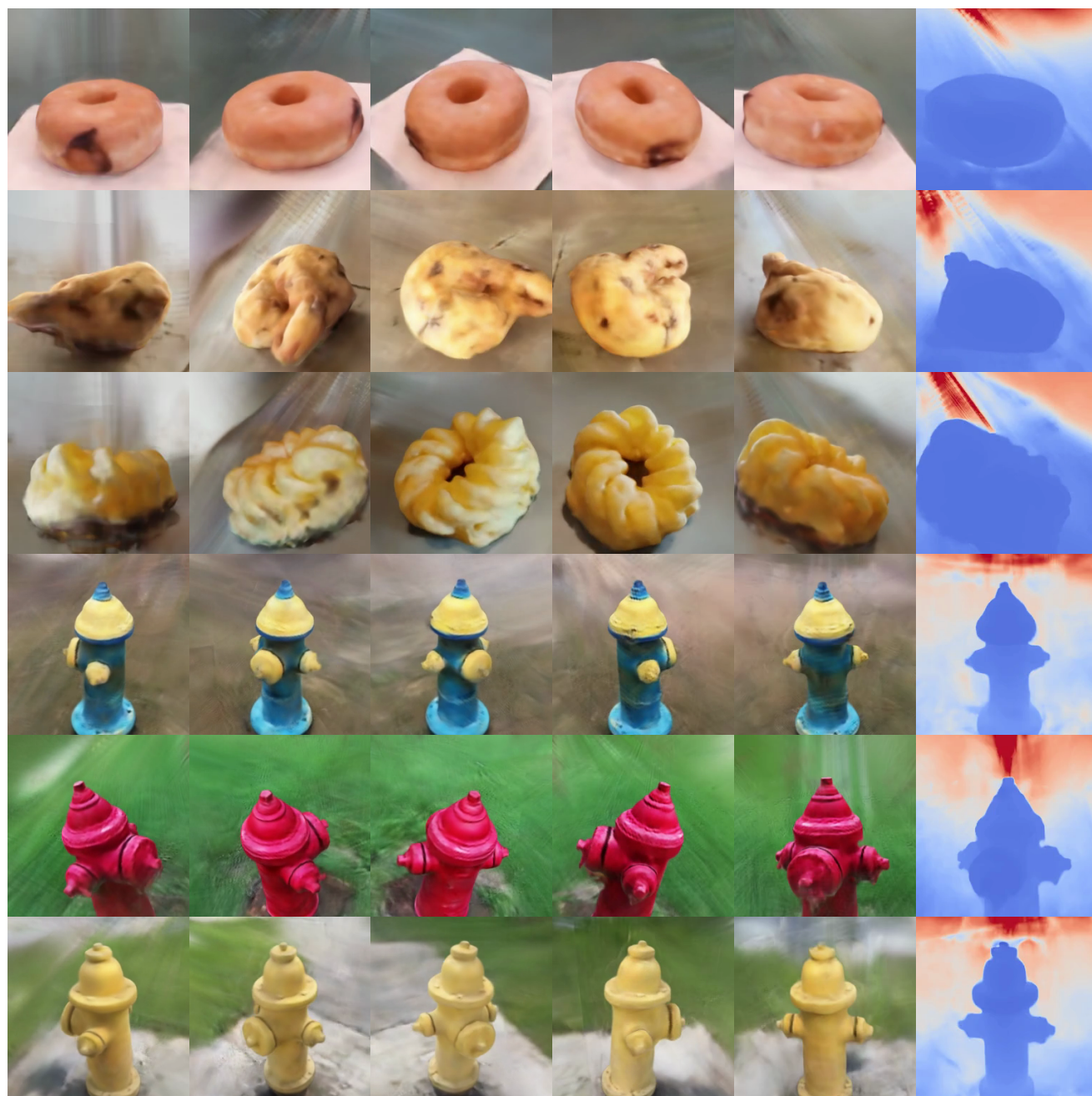


Figure 11. **Unconditional 3D Generation** displaying samples from our *Lightplane*-augmented Viewset Diffusion trained on CO3Dv2 [20].



Figure 12. **Monocular 3D Reconstruction on CO3Dv2.** With a single clean image as input, our model could generate realistic 3D structures matching the input views.



A white bowl with a blue and white fish in the center



A blue and white fire hydrant fire hydrant in a grassy area



A blue and white fire hydrant with a blue cap on the top

Figure 13. **Text-Conditioned Generation on CO3Dv2.** Our pipeline could generate 3D structures with text input as conditions.

References

- [1] Anonymous. Instant3d: Fast text-to-3d with sparse-view generation and large reconstruction model. *Under Review*, 2023. 1
- [2] Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5470–5479, 2022. 1, 2
- [3] Ang Cao and Justin Johnson. Hexplane: A fast representation for dynamic scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 130–141, 2023. 1
- [4] Ziang Cao, Fangzhou Hong, Tong Wu, Liang Pan, and Ziwei Liu. Large-vocabulary 3d diffusion model with transformer. *arXiv preprint arXiv:2309.07920*, 2023. 1
- [5] Eric R. Chan, Koki Nagano, Matthew A. Chan, Alexander W. Bergman, Jeong Joon Park, Axel Levy, Miika Aitala, Shalini De Mello, Tero Karras, and Gordon Wetzstein. GENVS: Generative novel view synthesis with 3D-aware diffusion models. In *ICCV*, 2023. 1
- [6] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. TensoRF: Tensorial radiance fields. In *arXiv*, 2022. 3
- [7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 2
- [8] Sara Fridovich-Keil, Giacomo Meanti, Frederik Rahbæk Warburg, Benjamin Recht, and Angjoo Kanazawa. K-planes: Explicit radiance fields in space, time, and appearance. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12479–12488, 2023. 1
- [9] Yicong Hong, Kai Zhang, Jiuxiang Gu, Sai Bi, Yang Zhou, Difan Liu, Feng Liu, Kalyan Sunkavalli, Trung Bui, and Hao Tan. Lrm: Large reconstruction model for single image to 3d. *arXiv preprint arXiv:2311.04400*, 2023. 4
- [10] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. *arXiv preprint arXiv:2301.12597*, 2023. 4
- [11] Ruilong Li, Matthew Tancik, and Angjoo Kanazawa. NerfAcc: A general nerf acceleration toolbox. *arXiv.cs, abs/2210.04847*, 2022. 2
- [12] Ruilong Li, Hang Gao, Matthew Tancik, and Angjoo Kanazawa. Nerfacc: Efficient sampling accelerates nerfs. *arXiv preprint arXiv:2305.04966*, 2023. 3
- [13] Ruoshi Liu, Rundi Wu, Basile Van Hoorick, Pavel Tokmakov, Sergey Zakharov, and Carl Vondrick. Zero-1-to-3: Zero-shot one image to 3d object. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9298–9309, 2023. 1
- [14] Tiange Luo, Chris Rockwell, Honglak Lee, and Justin Johnson. Scalable 3d captioning with pretrained models. *arXiv preprint arXiv:2306.07279*, 2023. 4
- [15] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020. 3, 7
- [16] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. NeRF: Representing scenes as neural radiance fields for view synthesis. In *Proc. ECCV*, 2020. 3
- [17] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multi-resolution hash encoding. *ACM Transactions on Graphics (ToG)*, 41(4):1–15, 2022. 3
- [18] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multi-resolution hash encoding. In *Proc. SIGGRAPH*, 2022. 2
- [19] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021. 4
- [20] 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 *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10901–10911, 2021. 8, 9
- [21] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2021. 2
- [22] Cheng Sun, Min Sun, and Hwann-Tzong Chen. Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5449–5459, 2021. 3
- [23] Stanislaw Szymanowicz, Christian Rupprecht, and Andrea Vedaldi. Viewset diffusion:(0-) image-conditioned 3d generative models from 2d data. *arXiv preprint arXiv:2306.07881*, 2023. 1
- [24] Tengfei Wang, Bo Zhang, Ting Zhang, Shuyang Gu, Jianmin Bao, Tadas Baltrusaitis, Jingjing Shen, Dong Chen, Fang Wen, Qifeng Chen, et al. Rodin: A generative model for sculpting 3d digital avatars using diffusion. *arXiv preprint arXiv:2212.06135*, 2022. 1
- [25] Alex Yu, Sara Fridovich-Keil, Matthew Tancik, Qinlong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. *arXiv preprint arXiv:2112.05131*, 2021. 3
- [26] Kai Zhang, Nick Kolkin, Sai Bi, Fujun Luan, Zexiang Xu, Eli Shechtman, and Noah Snavely. ARF: Artistic radiance fields. In *Proc. ECCV*, 2022. 3