Supplementary Material for Neural Geometric Level of Detail: Real-time Rendering with Implicit 3D Shapes

Towaki Takikawa[1,2,4*] Joey Litalien[1,3*] Kangxue Yin[1] Karsten Kreis[1] Charles Loop[1]
Derek Nowrouzezahrai[3] Alec Jacobson[2] Morgan McGuire[1,3] Sanja Fidler[1,2,4]

[1]NVIDIA [2]University of Toronto [3]McGill University [4]Vector Institute

nv-tlabs.github.io/nglod

A. Implementation Details

A.1. Architecture

We set the hidden dimension for all (single hidden layer) MLPs to $h = 128$. We use a ReLU activation function for the intermediate layer and none for the output layer, to support arbitrary distances. We set the feature dimension for the SVO to $m = 32$ and initialize all voxel features $\mathbf{z} \in \mathcal{Z}$ using a Gaussian prior with $\sigma = 0.01$. We performed ablations and discovered that we get satisfying quality with feature dimensions as low as $m = 8$, but we keep $m = 32$ as we make bigger gains in storage efficiency by keeping the octree depth shallower than we save by reducing the feature dimension.

The resolution of each level of the SVO is defined as $r_L = r_0 \cdot 2^L$, where $r_0 = 4$ is the initial resolution, capped at $L_{\text{max}} \in \{5, 6\}$ depending on the complexity of the geometry. Note that the octree used for rendering (compare Section 3.4) starts at an initial resolution of 1^3, but we do not store any feature vectors until the octree reaches a level where the resolution $r_0 = 4$. Each level contains a maximum of ${r_L}^3$ voxels. In practice, the total number is much lower because surfaces are sparse in $\mathbb{R}^3$, and we only allocate nodes where there is a surface.

A.2. Sampling

We implement a variety of sampling schemes for the generation of our pointcloud datasets.

Uniform. We first sample uniform random positions in the bounding volume $\mathcal{B} = [-1, 1]^3$ by sampling three uniformly distributed random numbers.

Surface. We have two separate sampling algorithms, one for meshes and one for signed distance functions. For meshes, we first compute per-triangle areas. We then select random triangles with a distribution proportional to the triangle areas, and then select a random point on the triangle using three uniformly distributed random numbers and barycentric coordinates. For signed distance functions, we first sample uniformly distributed points in $\mathcal{B}$. We then choose random points on a sphere to form a ray, and test if the ray hits the surface with sphere tracing. We continue sampling rays until we find enough rays that hit the surface.

Near. We can additionally sample near-surface points of a mesh by taking the surface samples, and perturbing the vector with random Gaussian noise with $\sigma = 0.01$.

A.3. Training

All training was done on a NVIDIA Tesla V100 GPU using PyTorch [11] with some operations implemented in CUDA. All models are trained with the Adam optimizer [5] with a learning rate of 0.001, using a set of $500\,000$ points resampled at every epoch with a batch size of 512. These points are distributed in a 2:2:1 split of surface, near, and uniform samples. We do not make use of positional encodings on the input points.

We train our representation summing together the loss functions of the distances at each LOD (see Equation (**??**)). We use L^2-distance for our individual per-level losses. For ShapeNet150 and Thingi32, we train all LODs jointly. For TurboSquid16, we use a progressive scheme where we train the highest LOD L_{max} first, and add new trainable levels $\ell = L_{\text{max}} - 1, L_{\text{max}} - 2, \ldots$ every 100 epochs. This training scheme slightly benefits lower LODs for more complex shapes.

We briefly experimented with different choices of hyperparameters for different architectures (notably for the baselines), but discovered these sets of hyperparameters worked well across all models.

*Authors contributed equally.

A.4. Rendering

We implement our baseline renderer using Python and PyTorch. The sparse renderer is implemented using CUDA, cub [8], and libtorch [11]. The implementation takes careful advantage of kernel fusion while still making the algorithm agnostic to the architecture. The ray-AABB intersection uses Marjercik et. al. [7]. Section C provides more details on the sparse octree intersection algorithm.

In the sphere trace, we terminate the algorithm for each individual ray if the iteration count exceeds the maximum or if the stopping criteria $\widehat{d} < \delta$ is reached. We set $\delta = 0.0003$. In addition, we also check that the step is not oscillating: $|\widehat{d}_k - \widehat{d}_{k-1}| < 6\delta$ and perform far plane clipping with depth 5. We bound the sphere tracing iterations to $k = 200$.

The shadows in the renders are obtained by tracing shadow rays using sphere tracing. We also enable SDF ambient occlusion [3] and materials through matcaps [15]. Surface normals are obtained using finite differences. As noted in the main paper, the frametimes measured only include the primary ray trace and normal computation time, and not secondary effects (e.g. shadows).

B. Experiment Details

B.1. Baselines

In this section, we outline the implementation details for DeepSDF [10], Fourier Feature Network (FFN) [16], SIREN [14], and Neural Implicits [2]. Across all baselines, we do not use an activation function at the very last layer to avoid restrictions on the range of distances the models can output. We find this does not significantly affect the results.

DeepSDF. We implement DeepSDF as in the paper, but remove weight normalization [13], since we observe improved performance without it in our experimental settings. We also do not use latent vectors, and instead use just the spatial coordinates as input to overfit DeepSDF to each specific shape.

Fourier Feature Network. We also implement FFN following the paper, and choose $\sigma = 8$ as it seems to provide the best overall trade-off between high-frequency noise and detail. We acknowledge that the reconstruction quality for FFN is very sensitive to the choice of this hyperparameter; however, we find that it is time-consuming and therefore impractical to search for the optimal σ per shape.

SIREN. We implement SIREN following the paper, and also utilize the weight initialization scheme in the paper. We do not use the the Eikonal regularizer $|\nabla f| = 1$ for our loss function (and use a simple L^2-loss function across all baselines), because we find that it is important to be able to fit non-metric SDFs that do not satisfy the Eikonal equation constraints. Non-metric SDFs are heavily utilized in practice to make SDF-based content creation easier.

Neural Implicits. We implement Neural Implicits without any changes to the paper, other than using our sampling scheme to generate the dataset so we can control training variability across baselines.

B.2. Reconstruction Metrics

Geometry Metrics. Computing the Chamfer-L^1 distance requires surface samples, of both the ground-truth mesh as well as the predicted SDF. Typically, these are obtained for the predicted SDF sampling the mesh extracted with Marching Cubes [6] which introduces additional error. Instead, we obtain samples by sampling the SDF surface using ray tracing. We uniformly sample $2^{17} = 131\,072$ points in the bounding volume $\mathcal{B}$, each assigned with a random spherical direction. We then trace each of these rays using sphere tracing, and keep adding samples until the minimum number of points are obtained. The stopping criterion is the same as discussed in A.4. We use the Chamfer distance as implemented in PyTorch3D [12].

Image Metrics. We compute the Normal-L^2 score by sampling 32 evenly distributed, fixed camera positions using a spherical Fibonacci sequence with radius 4. Images are rendered at resolution 512×512 and surface normals are evaluated against interpolated surface normals from the reference mesh. We evaluate the normal error only on the intersection of the predicted and ground-truth masks, since we separately evaluate mask alignment with intersection over union (iIoU). We use these two metrics because the shape silhouettes are perceptually important and surface normals drive the shading. We use 4 samples per pixel for both images, and implement the mesh renderer using Mitsuba 2 [9].

C. Sparse Ray-Octree Intersection

We provide more details for the subroutines appearing in Algorithm 1. Pseudo code for the procedure DECIDE is listed below:

```
1:  procedure DECIDE(R, N^(ℓ), ℓ)
2:      for all t ∈ {0, ..., |N^(ℓ)| − 1} do in parallel
3:          {i, j} ← N_t^(ℓ)
4:          if R_i ∩ V_j^(ℓ) then
5:              if ℓ = L then
6:                  D_t ← 1
7:              else
8:                  D_t ← NUMCHILDREN(V_j^ℓ)
9:          else
10:             D_t ← 0
11:     return D
```

The DECIDE procedure determines the voxel-ray pairs that result in intersections. The procedure runs in parallel over (threads) t (line 2). For each t, we fetch the ray and voxel indices i and j (line 3). If ray $\mathcal{R}_i$ intersects voxel $V_j^{(\ell)}$ (line 4), we check if we have reached the final level L (line 5). If so, we write a 1 into list $\mathbf{D}$ at position t (line 6). Otherwise, we write the NUMCHILDREN of $V_j^{(\ell)}$ (i.e., the number of occupied children of a voxel in the octree) into list $\mathbf{D}$ at position t (line 8). If ray $\mathcal{R}_i$ does not intersect voxel $V_j^{(\ell)}$, we write 0 into list $\mathbf{D}$ at position t (line 10). The resulting list $\mathbf{D}$ is returned to the caller (line 11).

Next, we compute the Exclusive Sum of $\mathbf{D}$ and store the resulting list in $\mathbf{S}$. The Exclusive Sum $\mathbf{S}$ of a list of numbers $\mathbf{D}$ is defined as

$$\mathbf{S}_i = \begin{cases} 0 & \text{if } i = 0, \\ \sum_{j=0}^{i-1} \mathbf{D}_j & \text{otherwise.} \end{cases}$$

Note that while this definition appears inherently serial, fast parallel methods for EXCLUSIVESUM are available that treat the problem as a series of parallel reductions [1, 4]. The exclusive sum is a powerful parallel programming construct that provides the index for writing data into a list from independent threads without conflicts (write hazards).

This can be seen in the pseudo code for procedure COMPACTIFY called at the final step of iteration in Algorithm 1:

```
1: procedure COMPACTIFY(N^(ℓ), D, S)
2:     for all t ∈ {0, ..., |N^(ℓ)| − 1} do in parallel
3:         if D_t = 1 then
4:             k ← S_t
5:             N_k^(ℓ+1) ← N_t^(ℓ)
6:     return N^(ℓ+1)
```

The COMPACTIFY subroutine removes all ray-voxel pairs that do not result in an intersection (and thus do not contribute to $\mathbf{S}$). This routine is run in parallel over t (line 2). When $\mathbf{D}_t = 1$, meaning voxel $\mathcal{V}_t^\ell$ was hit (line 3), we copy the ray/voxel index pair from $\mathbf{N}_t^{(\ell)}$ to its new location k obtained from the exclusive sum result $\mathbf{S}_t$ (line 4), $\mathbf{N}^{(\ell+1)}$ (line 5). We then return the new list $\mathbf{N}^{(\ell+1)}$ to the caller.

If the iteration has not reached the final step, i.e. $l \neq L$ in Algorithm 1, we call SUBDIVIDE listed below:

```
1: procedure SUBDIVIDE(N^(ℓ), D, S)
2:     for all t ∈ {0, ..., |N^(ℓ)| − 1} do in parallel
3:         if D_t ≠ 0 then
4:             {i, j} ← N_t^(ℓ)
5:             k ← S_t
6:             for c ∈ ORDEREDCHILDREN(R_i, V_j^(ℓ)) do
7:                 N_k^(ℓ+1) ← {i, c}
8:                 k ← k + 1
9:     return N^ℓ+1
```

The SUBDIVIDE populates the next list $\mathbf{N}^{(\ell+1)}$ by subdividing out $\mathbf{N}^{(\ell)}$. This routine is run in parallel over t (line 2). When $\mathbf{D}_t \neq 0$, meaning voxel $\mathcal{V}_t^{(\ell)}$ was hit (line 3), we do the following: We load the ray/voxel index pair $\{i, j\}$ from $\mathbf{N}_t^\ell$ (line 4). The output index k for the first child voxel index is obtained (line 5). We then iterate over the ordered children of the current voxel $V_j^{(\ell)}$ using iterator ORDERED-CHILDREN (line 6). This iterator returns the child voxels of $V_j^{(\ell)}$ in front-to-back order with respect to ray $\mathcal{R}_i$. This ordering is only dependant on which of the 8 octants of space contains the origin of the ray, and can be stored in a precomputed 8×8 table. We write the ray/voxel index pair to the new list $\mathbf{N}^{(\ell+1)}$ at position k (line 7). The output index k is incremented (line 8), and the resulting list of (subdivided) ray/voxel index pairs (line 9).

D. Additional Results

More result examples from each dataset used can be found in the following pages. We also refer to our supplementary video for a real-time demonstration of our method.

E. Artist Acknowledgements

We credit the following artists for the 3D assets used in this work. In alphabetical order: 3D Aries (*Cogs*), abramsdesign (*Cabin*), the Art Institute of Chicago (*Lion*), Distefan (*Train*), DRONNNNN95 (*House*), Dmitriev Vasiliy (*V Mech*), Felipe Alfonso (*Cheese*), Florian Berger (*Oldcar*), Gary Warne (*Mobius*), Inigo Quilez (*Snail*), klk (*Teapot*), Martijn Steinrucken (*Snake*), Max 3D Design (*Robot*), monsterkodi (*Skull*), QE3D (*Parthenon*), RaveeCG (*Horseman*), sam_rus (*City*), the Stanford Computer Graphics Lab (*Lucy*), TheDizajn (*Boat*), Xor (*Burger*, *Fish*), your artist (*Chameleon*), and zames1992 (*Cathedral*).

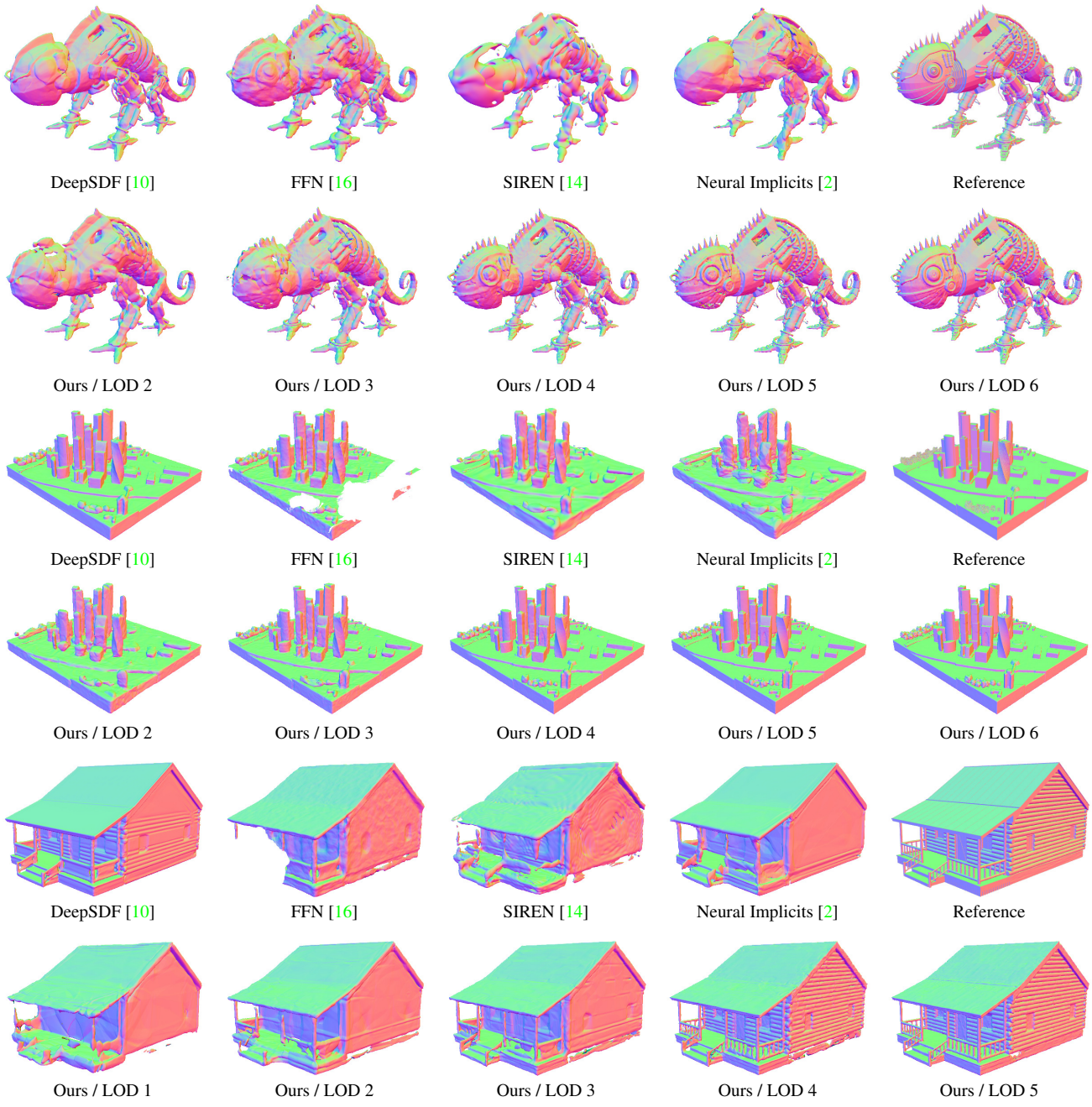

Figure 1: **Additional TurboSquid16 Results.** FFN exhibits white patch artifacts (e.g. City and Cabin) because it struggles to learn a conservative metric SDF, resulting in the sphere tracing algorithm missing the surface entirely. Best viewed zoomed in.

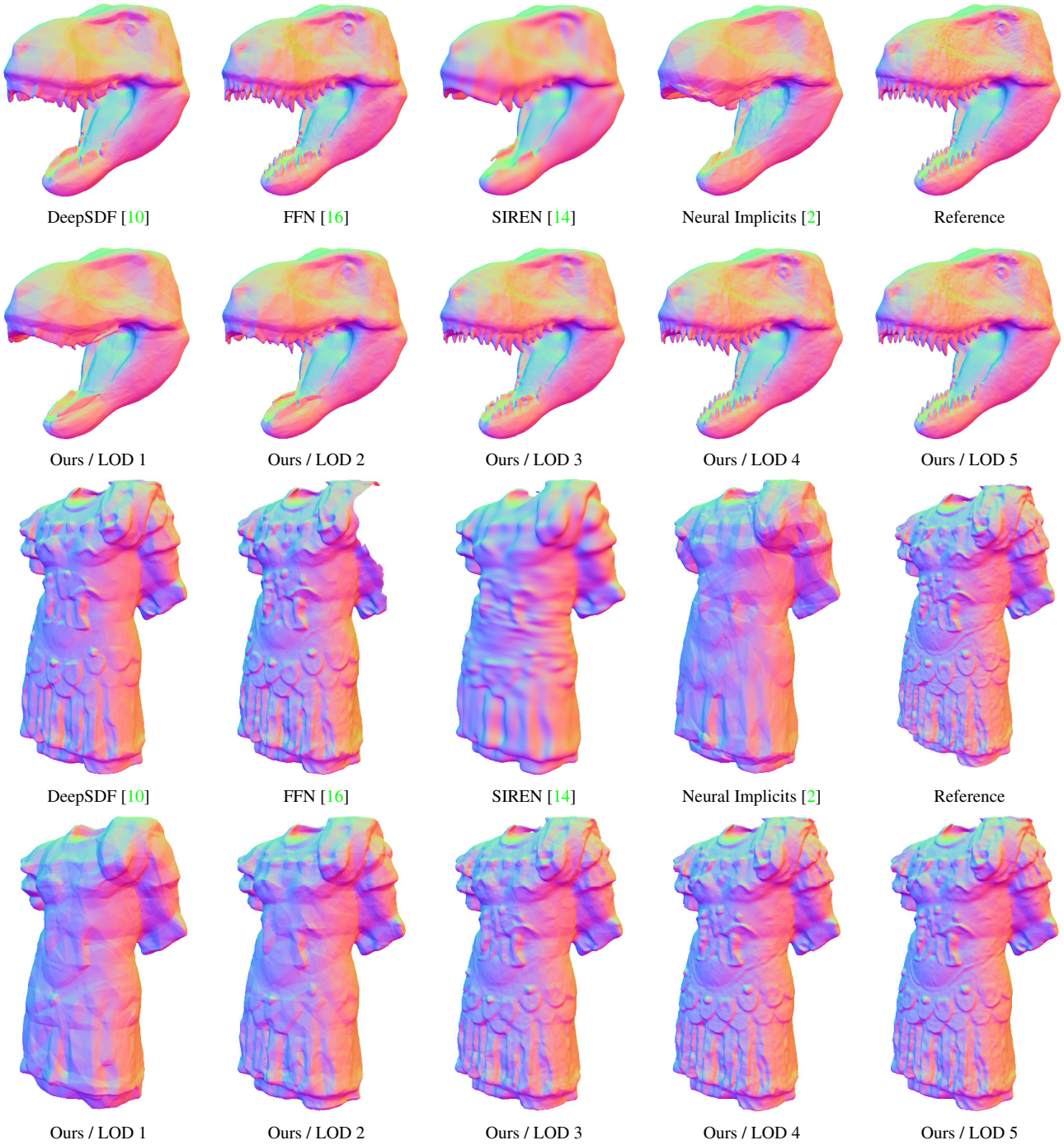

Figure 2: **Additional Thingi32 Results.** Best viewed zoomed in.

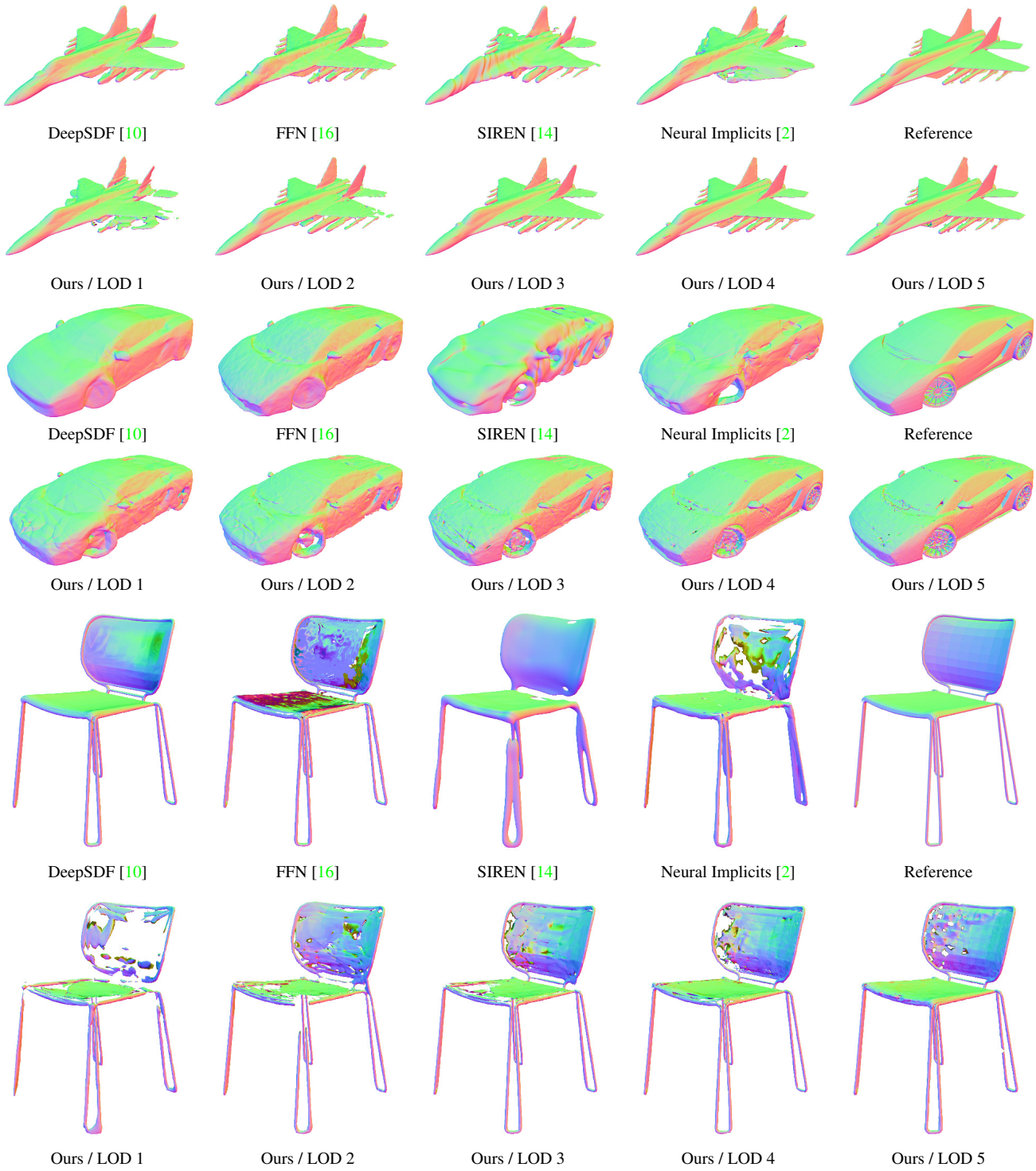

Figure 3: **Additional ShapeNet150 Results.** Our method struggles with thin flat features with little to no volume, such as Jetfighter wings and the back of the Chair. Best viewed zoomed in.

References

[1] Guy E. Blelloch. *Vector models for data-parallel computing*. MIT Press, 1990. 3

[2] Thomas Davies, Derek Nowrouzezahrai, and Alec Jacobson. On the effectiveness of weight-encoded neural implicit 3D shapes. *arXiv preprint arXiv:2009.09808*, 2020. 2, 4, 5, 6

[3] Alex Evans. Fast approximations for global illumination on dynamic scenes. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, page 153–171, 2006. 2

[4] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007. 3

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 1

[6] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, page 163–169, 1987. 2

[7] Alexander Majercik, Cyril Crassin, Peter Shirley, and Morgan McGuire. A ray-box intersection algorithm and efficient dynamic voxel rendering. *Journal of Computer Graphics Techniques (JCGT)*, 7(3):66–81, September 2018. 2

[8] Duane Merrill. CUB: a library of warp-wide, block-wide, and device-wide GPU parallel primitives, 2017. 2

[9] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: A retargetable forward and inverse renderer. *ACM Trans. Graph.*, 38(6), Nov. 2019. 2

[10] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. DeepSDF: Learning continuous signed distance functions for shape representation. In *IEEE Conf. Comput. Vis. Pattern Recog.*, June 2019. 2, 4, 5, 6

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Adv. Neural Inform. Process. Syst.*, pages 8026–8037, 2019. 1, 2

[12] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Accelerating 3D deep learning with PyTorch3D. *arXiv preprint arXiv:2007.08501*, 2020. 2

[13] Tim Salimans and Durk P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Adv. Neural Inform. Process. Syst.*, volume 29, pages 901–909, 2016. 2

[14] Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. In *Adv. Neural Inform. Process. Syst.*, 2020. 2, 4, 5, 6

[15] Peter-Pike J Sloan, William Martin, Amy Gooch, and Bruce Gooch. The lit sphere: a model for capturing NPR shading from art. In *Proceedings of Graphics Interface 2001*, pages 143–150, 2001. 2

[16] Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *Adv. Neural Inform. Process. Syst.*, 2020. 2, 4, 5, 6