# 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

## Abstract

*Neural signed distance functions (SDFs) are emerging as an effective representation for 3D shapes. State-of-the-art methods typically encode the SDF with a large, fixed-size neural network to approximate complex shapes with implicit surfaces. Rendering with these large networks is, however, computationally expensive since it requires many forward passes through the network for every pixel, making these representations impractical for real-time graphics. We introduce an efficient neural representation that, for the first time, enables real-time rendering of high-fidelity neural SDFs, while achieving state-of-the-art geometry reconstruction quality. We represent implicit surfaces using an octree-based feature volume which adaptively fits shapes with multiple discrete levels of detail (LODs), and enables continuous LOD with SDF interpolation. We further develop an efficient algorithm to directly render our novel neural SDF representation in real-time by querying only the necessary LODs with sparse octree traversal. We show that our representation is 2–3 orders of magnitude more efficient in terms of rendering speed compared to previous works. Furthermore, it produces state-of-the-art reconstruction quality for complex shapes under both 3D geometric and 2D image-space metrics.*

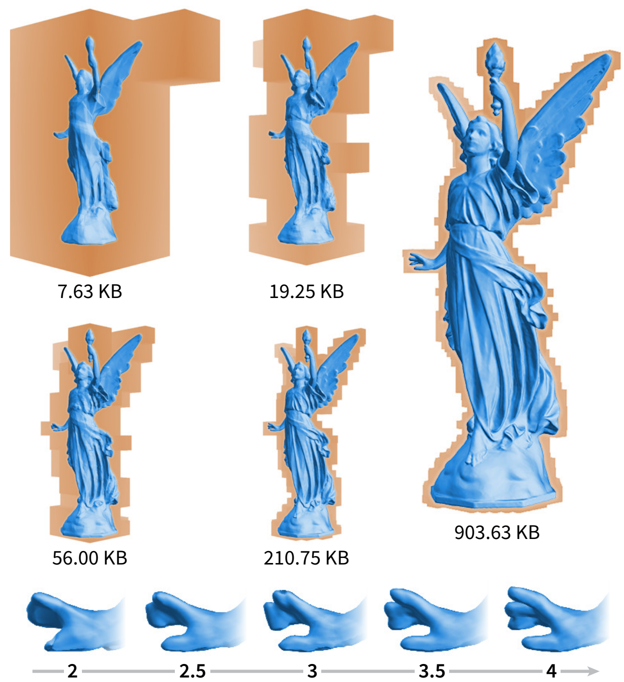



Figure 1: **Levels of Detail.** Our representation pools features from multiple scales to adaptively reconstruct high-fidelity geometry with continuous level of detail (LOD). The subfigures show surfaces (blue) at varying LODs, superimposed on the corresponding coarse, sparse octrees (orange) which contain the features of the learned signed distance functions. These were directly rendered in real-time using our efficient sparse sphere tracing algorithm.

## 1. Introduction

Advanced geometric modeling and rendering techniques in computer graphics use 3D shapes with complex details, arbitrary topology, and quality, usually leveraging polygon meshes. However, it is non-trivial to adapt those representations to learning-based approaches since they lack differentiability, and thus cannot easily be used in computer vision applications such as learned image-based 3D reconstruction. Recently, neural approximations of signed distance functions (neural SDFs) have emerged as an attractive choice to scale up computer vision and graphics applications. Prior works [39, 33, 7, 9] have shown that neural networks can encode accurate 3D geometry without restrictions on topology or resolution by learning the SDF, which defines a surface by its zero level-set. These works commonly use a large, fixed-size multi-layer perceptron (MLP) as the learned distance function.

Directly rendering and probing neural SDFs typically relies on sphere tracing [19], a root-finding algorithm that can

*Authors contributed equally.

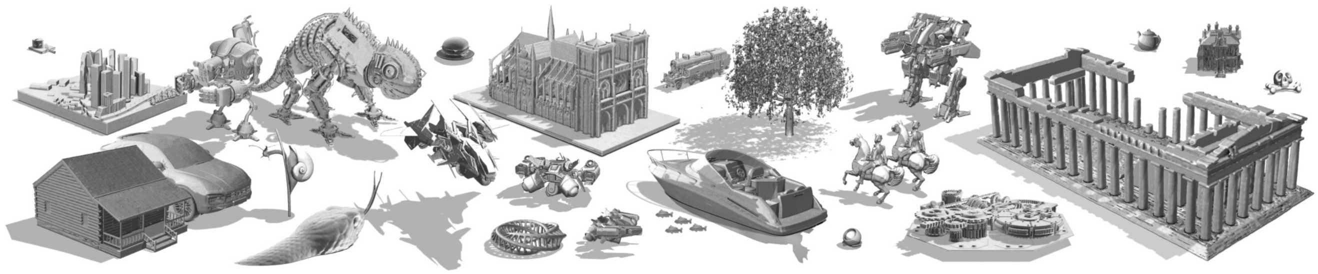


Figure 2: We are able to fit shapes of varying complexity, style, scale, with consistently good quality, while being able to leverage the geometry for shading, ambient occlusion [12], and even shadows with secondary rays. **Best viewed zoomed in.**

require hundreds of SDF evaluations *per pixel* to converge. As a single forward pass through a large MLP-based SDF can require millions of operations, neural SDFs quickly become impractical for real-time graphics applications as the cost of computing a *single* pixel inflates to hundreds of millions of operations. Works such as Davies et al. [9] circumvent this issue by using a small neural network to overfit single shapes, but this comes at the cost of generality and reconstruction quality. Previous approaches also use fixed-size neural networks, making them unable to express geometry with complexity exceeding the capacity of the network.

In this paper, we present a novel representation for neural SDFs that can adaptively scale to different levels of detail (LODs) and reconstruct highly detailed geometry. Our method can smoothly interpolate between different scales of geometry (see Figure 1) and can be rendered in *real-time* with a reasonable memory footprint. Similar to Davies et al. [9], we also use a small MLP to make sphere tracing practical, but without sacrificing quality or generality.

We take inspiration from classic surface extraction mechanisms [28, 13] which use quadrature and spatial data structures storing distance values to finely discretize the Euclidean space such that simple, linear basis functions can reconstruct the geometry. In such works, the resolution or tree depth determines the geometric level of detail (LOD) and different LODs can be blended with interpolation. However, they usually require high tree depths to recreate a solution with satisfying quality.

In contrast, we discretize the space by using a sparse voxel octree (SVO) and we store learned feature vectors instead of signed distance values. These vectors can be decoded into scalar distances using a shallow MLP, allowing us to truncate the tree depth while inheriting the advantages of classic approaches (e.g., LOD). We additionally develop a ray traversal algorithm tailored to our architecture, which allows us to render geometry close to 100$\times$ faster than DeepSDF [39]. Although direct comparisons with neural volumetric rendering methods are not possible, we report frametimes over 500$\times$ faster than NeRF [34] and 50$\times$ faster than NSVF [26] in similar experimental settings.

In summary, our contributions are as follows:

- We introduce the first real-time rendering approach for complex geometry with neural SDFs.
- We propose a neural SDF representation that can efficiently capture multiple LODs, and reconstruct 3D geometry with state-of-the-art quality (see Figure 2).
- We show that our architecture can represent 3D shapes in a compressed format with higher visual fidelity than traditional methods, and generalizes across different geometries even from a single learned example.

Due to the real-time nature of our approach, we envision this as a modular building block for many downstream applications, such as scene reconstruction from images, robotics navigation, and shape analysis.

## 2. Related Work

Our work is most related to prior research on mesh simplification for level of detail, 3D neural shape representations, and implicit neural rendering.

**Level of Detail.** Level of Detail (LOD) [29] in computer graphics refers to 3D shapes that are filtered to limit feature variations, usually to approximately twice the pixel size in image space. This mitigates flickering caused by aliasing, and accelerates rendering by reducing model complexity. While signal processing techniques can filter textures [49], geometry filtering is representation-specific and challenging. One approach is mesh decimation, where a mesh is simplified to a budgeted number of faces, vertices, or edges. Classic methods [15, 20] do this by greedily removing mesh elements with the smallest impact on geometric accuracy. More recent methods optimize for perceptual metrics [25, 24, 8] or focus on simplifying topology [31]. Meshes suffer from discretization errors under low memory constraints and have difficulty blending between LODs. In contrast, SDFs can represent smooth surfaces with less memory and smoothly blend between LODs to reduce aliasing. Our neural SDFs inherit these properties.

**Neural Implicit Surfaces.** Implicit surface-based methods encode geometry in latent vectors or neural network weights, which parameterize surfaces through level-sets.

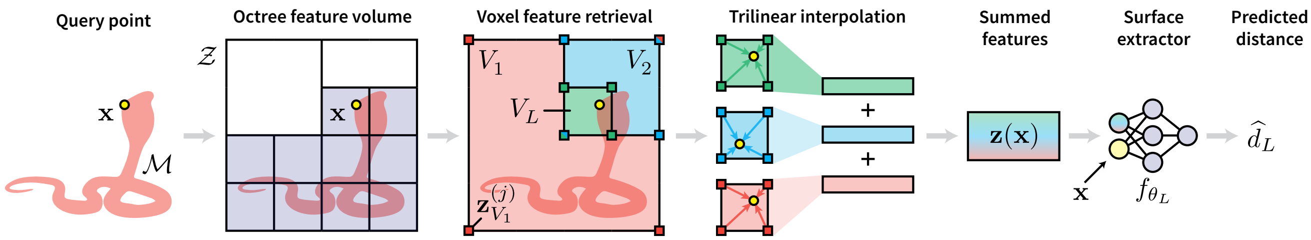



Figure 3: **Architecture.** We encode our neural SDF using a sparse voxel octree (SVO) which holds a collection of features $\mathcal{Z}$. The levels of the SVO define LODs and the voxel corners contain feature vectors defining local surface segments. Given query point $\mathbf{x}$ and LOD $L$, we find corresponding voxels $V_{1:L}$, trilinearly interpolate their corners $\mathbf{z}_V^{(j)}$ up to $L$ and sum to obtain a feature vector $\mathbf{z}(\mathbf{x})$. Together with $\mathbf{x}$, this feature is fed into a small MLP $f_{\theta_L}$ to obtain a signed distance $\widehat{d}_L$. We jointly optimize MLP parameters $\theta$ and features $\mathcal{Z}$ end-to-end.

Seminal works [39, 33, 7] learn these iso-surfaces by encoding the shapes into latent vectors using an auto-decoder—a large MLP which outputs a scalar value conditional on the latent vector and position. Another concurrent line of work [47, 45] uses periodic functions resulting in large improvements in reconstruction quality. Davies et al. [9] proposes to overfit neural networks to single shapes, allowing a compact MLP to represent the geometry. Works like Curriculum DeepSDF [11] encode geometry in a progressively growing network, but discard intermediate representations. BSP-Net and CvxNet [6, 10] learn implicit geometry with space-partitioning trees. PIFu [42, 43] learns features on a dense 2D grid with depth as an additional input parameter, while other works learn these on sparse regular [16, 4] or deformed [14] 3D grids. PatchNets [48] learn surface patches, defined by a point cloud of features. Most of these works rely on an iso-surface extraction algorithm like Marching Cubes [28] to create a dense surface mesh to render the object. In contrast, in this paper we present a method that directly renders the shape at interactive rates.

**Neural Rendering for Implicit Surfaces.** Many works focus on rendering neural implicit representations. Niemeyer et al. [36] proposes a differentiable renderer for implicit surfaces using ray marching. DIST [27] and SDFDiff [22] present differentiable renderers for SDFs using sphere tracing. These differentiable renderers are agnostic to the ray-tracing algorithm; they only require the differentiability with respect to the ray-surface intersection. As such, we can leverage the same techniques proposed in these works to make our renderer also differentiable. NeRF [34] learns geometry as density fields and uses ray marching to visualize them. IDR [50] attaches an MLP-based shading function to a neural SDF, disentangling geometry and shading. NSVF [26] is similar to our work in the sense that it also encodes feature representations with a sparse octree. In contrast to NSVF, our work enables level of detail and uses sphere tracing, which allows us to separate out the geometry from shading and therefore optimize ray tracing, something not possible in a volumetric rendering framework. As mentioned previously, our renderer is two orders of magnitude faster compared to numbers reported in NSVF [26].

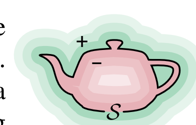
# 3. Method

Our goal is to design a representation which reconstructs detailed geometry and enables continuous level of detail, all whilst being able to render at interactive rates. Figure 3 shows a visual overview of our method. Section 3.1 provides a background on neural SDFs and its limitations. We then present our method which encodes the neural SDF in a sparse voxel octree in Section 3.2 and provide training details in Section 3.3. Our rendering algorithm tailored to our representation is described in Section 3.4.

## 3.1. Neural Signed Distance Functions (SDFs)

SDFs are functions $f : \mathbb{R}^3 \to \mathbb{R}$ where $d = f(\mathbf{x})$ is the shortest *signed* distance from a point $\mathbf{x}$ to a surface $\mathcal{S} = \partial\mathcal{M}$ of a volume $\mathcal{M} \subset \mathbb{R}^3$, where the sign indicates whether $\mathbf{x}$ is inside or outside of $\mathcal{M}$. As such, $\mathcal{S}$ is *implicitly* represented as the zero level-set of $f$:

$$\mathcal{S} = \big\{\mathbf{x} \in \mathbb{R}^3 \,\big|\, f(\mathbf{x}) = 0\big\}. \tag{1}$$




A *neural* SDF encodes the SDF as the parameters $\theta$ of a neural network $f_\theta$. Retrieving the signed distance for a point $\mathbf{x} \in \mathbb{R}^3$ amounts to computing $f_\theta(\mathbf{x}) = \widehat{d}$. The parameters $\theta$ are optimized with the loss $J(\theta) = \mathbb{E}_{\mathbf{x},d}\mathcal{L}\big(f_\theta(\mathbf{x}), d\big)$, where $d$ is the ground-truth signed distance and $\mathcal{L}$ is some distance metric such as $L^2$-distance. An optional input "shape" feature vector $\mathbf{z} \in \mathbb{R}^m$ can be used to condition the network to fit different shapes with a fixed $\theta$.

To render neural SDFs *directly*, ray-tracing can be done with a root-finding algorithm such as sphere tracing [19]. This algorithm can perform up to a hundred distance queries per ray, making standard neural SDFs prohibitively expensive if the network is large and the distance query is too slow. Using small networks can speed up this iterative rendering process, but the reconstructed shape may be inaccurate. Moreover, fixed-size networks are unable to fit highly complex shapes and cannot adapt to simple or far-away objects where visual details are unnecessary.

In the next section, we describe a framework that addresses these issues by encoding the SDF using a sparse voxel octree, allowing the representation to adapt to different levels of detail and to use shallow neural networks to encode geometry whilst maintaining geometric accuracy.

### 3.2. Neural Geometric Levels of Detail

**Framework.** Similar to standard neural SDFs, we represent SDFs using parameters of a neural network and an additional learned input feature which encodes the shape. Instead of encoding shapes using a single feature vector $\mathbf{z}$ as in DeepSDF [39], we use a feature volume which contains a *collection* of feature vectors, which we denote by $\mathcal{Z}$.

We store $\mathcal{Z}$ in a sparse voxel octree (SVO) spanning the bounding volume $\mathcal{B} = [-1, 1]^3$. Each voxel $V$ in the SVO holds a learnable feature vector $\mathbf{z}_V^{(j)} \in \mathcal{Z}$ at each of its eight corners (indexed by $j$), which are shared if neighbour voxels exist. Voxels are allocated only if the voxel $V$ contains a surface, making the SVO sparse.

Each level $L \in \mathbb{N}$ of the SVO defines a LOD for the geometry. As the tree depth $L$ in the SVO increases, the surface is represented with finer discretization, allowing reconstruction quality to scale with memory usage. We denote the maximum tree depth as $L_{\text{max}}$. We additionally employ small MLP neural networks $f_{\theta_{1:L_{\text{max}}}}$, denoted as decoders, with parameters $\theta_{1:L_{\text{max}}} = \{\theta_1, \ldots, \theta_{L_{\text{max}}}\}$ for each LOD.

To compute an SDF for a query point $\mathbf{x} \in \mathbb{R}^3$ at the desired LOD $L$, we traverse the tree up to level $L$ to find all voxels $V_{1:L} = \{V_1, \ldots, V_L\}$ containing $\mathbf{x}$. For each level $\ell \in \{1, \ldots, L\}$, we compute a per-voxel shape vector $\psi(\mathbf{x}; \ell, \mathcal{Z})$ by trilinearly interpolating the corner features of the voxels at $\mathbf{x}$. We sum the features across the levels to get $\mathbf{z}(\mathbf{x}; L, \mathcal{Z}) = \sum_{\ell=1}^{L} \psi(\mathbf{x}; \ell, \mathcal{Z})$, and pass them into the MLP with LOD-specific parameters $\theta_L$. Concretely, we compute the SDF as

$$\widehat{d}_L = f_{\theta_L}\left([\mathbf{x}\,, \mathbf{z}(\mathbf{x}; L, \mathcal{Z})]\right), \tag{2}$$

where $[\,\cdot\,,\,\cdot\,]$ denotes concatenation. This summation across LODs allows meaningful gradients to propagate across LODs, helping especially coarser LODs.

Since our shape vectors $\mathbf{z}_V^{(j)}$ now only represent small surface segments instead of entire shapes, we can move the computational complexity out of the neural network $f_\theta$ and into the feature vector query $\psi : \mathbb{R}^3 \to \mathbb{R}^m$, which amounts to a SVO traversal and a trilinear interpolation of the voxel features. This key design decision allows us to use very small MLPs, enabling significant speed-ups without sacrificing reconstruction quality.

**Level Blending.** Although the levels of the octree are discrete, we are able to smoothly interpolate between them. To obtain a desired *continuous* LOD $\tilde{L} \geq 1$, we blend between different discrete octree LODs $L$ by linearly interpolating the corresponding predicted distances:

$$\widehat{d}_{\tilde{L}} = (1 - \alpha)\, \widehat{d}_{L^*} + \alpha\, \widehat{d}_{L^*+1}, \tag{3}$$

where $L^* = \lfloor \tilde{L} \rfloor$ and $\alpha = \tilde{L} - \lfloor \tilde{L} \rfloor$ is the fractional part, allowing us to smoothly transition between LODs (see Figure 1). This simple blending scheme only works for SDFs, and does not work well for density or occupancy and is ill-defined for meshes and point clouds. We discuss how we set the continuous LOD $\tilde{L}$ at render-time in Section 3.4.

### 3.3. Training

We ensure that each discrete level $L$ of the SVO represents valid geometry by jointly training each LOD. We do so by computing individual losses at each level and summing them across levels:

$$J(\theta, \mathcal{Z}) = \mathbb{E}_{\mathbf{x},d} \sum_{L=1}^{L_{\text{max}}} \left\| f_{\theta_L}\left([\mathbf{x}\,, \mathbf{z}(\mathbf{x}; L, \mathcal{Z})]\right) - d \right\|^2. \tag{4}$$

We then stochastically optimize the loss function with respect to both $\theta_{1:L_{\text{max}}}$ and $\mathcal{Z}$. The expectation is estimated with importance sampling for the points $\mathbf{x} \in \mathcal{B}$. We use samples from a mixture of three distributions: uniform samples in $\mathcal{B}$, surface samples, and perturbed surface samples. We detail these sampling algorithms and specific training hyperparameters in the supplementary materials.

### 3.4. Interactive Rendering

**Sphere Tracing.** We use sphere tracing [19] to render our representation directly. Rendering an SVO-based SDF using sphere tracing, however, raises some technical implications that need to be addressed. Typical SDFs are defined on all of $\mathbb{R}^3$. In contrast, our SVO SDFs are defined only for voxels $V$ which intersect the surface geometry. Therefore, proper handling of distance queries made in empty space is required. One option is to use a constant step size, *i.e.* ray marching, but there is no guarantee the trace will converge because the step can overshoot.

Instead, at the beginning of the frame we first perform a ray-SVO intersection (details below) to retrieve every voxel $V$ at each resolution $\ell$ that intersects with the ray. Formally, if $\mathbf{r}(t) = \mathbf{x}_0 + t\mathbf{d}$, $t > 0$ is a ray with origin $\mathbf{x}_0 \in \mathbb{R}^3$ and direction $\mathbf{d} \in \mathbb{R}^3$, we let $\mathcal{V}_\ell(\mathbf{r})$ denote the depth-ordered set of intersected voxels by $\mathbf{r}$ at level $\ell$.

Each voxel in $\mathcal{V}_\ell(\mathbf{r})$ contains the intersected ray index, voxel position, parent voxel, and pointers to the eight corner feature vectors $\mathbf{z}_V^{(j)}$. We retrieve pointers instead of feature vectors to save memory. The feature vectors are stored in a flatenned array, and the pointers are precalculated in an initialization step by iterating over all voxels and finding corresponding indices to the features in each corner.

**Adaptive Ray Stepping.** For a given ray in a sphere trace iteration $k$, we perform a ray-AABB intersection [30] against the voxels in the target LOD level $L$ to retrieve the first voxel $V_L^* \in \mathcal{V}_L(\mathbf{r})$ that hits. If $\mathbf{x}_k \notin V_L^*$, we advance $\mathbf{x}$ to the ray-AABB intersection point. If $\mathbf{x}_k \in V_L^*$, we query

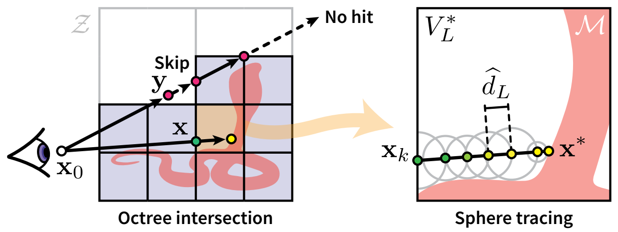



Figure 4: **Adaptive Ray Steps.** When the query point is *inside* a voxel (e.g., $\mathbf{x}$), trilinear interpolation is performed on all corresponding voxels up to the base octree resolution to compute a sphere tracing step (right). When the query point is *outside* a voxel (e.g., $\mathbf{y}$), ray-AABB intersection is used to skip to the next voxel.

our feature volume. We recursively retrieve all parent voxels $V_\ell^*$ corresponding to the coarser levels $\ell \in \{1, ..., L-1\}$, resulting in a collection of voxels $V_{1:L}^*$. We then sum the trilinearly interpolated features at each node. Note the parent nodes always exist by construction. The MLP $f_{\theta_L}$ then produces a conservative distance $\widehat{d}_L$ to move in direction $\mathbf{d}$, and we take a standard sphere tracing step: $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \widehat{d}_L \mathbf{d}$.

If $\mathbf{x}_{k+1}$ is now in empty space, we skip to the next voxel in $\mathcal{V}_L(\mathbf{r})$ along the ray and discard the ray $\mathbf{r}$ if none exists. If $\mathbf{x}_{k+1}$ is inside a voxel, we perform a sphere trace step. This repeats until all rays miss or if a stopping criterion is reached to recover a hit point $\mathbf{x}^* \in \mathcal{S}$. The process is illustrated in Figure 4. This adaptive stepping enables voxel sparsity by never having to query in empty space, allowing a minimal storage for our representation. We detail the stopping criterion in the supplementary material.

**Sparse Ray-Octree Intersection.** We now describe our novel ray-octree intersection algorithm that makes use of a breadth-first traversal strategy and parallel scan kernels [32] to achieve high performance on modern graphics hardware. Algorithm 1 provides pseudocode of our algorithm. We provide subroutine details in the supplemental material.

**Algorithm 1** Iterative, parallel, breadth-first octree traversal

1: **procedure** RAYTRACEOCTREE($L, \mathcal{R}$)
2: $\quad \mathbf{N}_i^{(0)} \leftarrow \{i, 0\}, i = 0, \dots, |\mathcal{R}| - 1$
3: $\quad$ **for** $\ell = 0$ to $L$ **do**
4: $\quad\quad \mathbf{D} \leftarrow \text{DECIDE}(\mathcal{R}, \mathbf{N}^{(\ell)}, \ell)$
5: $\quad\quad \mathbf{S} \leftarrow \text{EXCLUSIVESUM}(\mathbf{D})$
6: $\quad\quad$ **if** $\ell = L$ **then**
7: $\quad\quad\quad \mathbf{N}^{(\ell)} \leftarrow \text{COMPACTIFY}(\mathbf{N}^{(\ell)}, \mathbf{D}, \mathbf{S})$
8: $\quad\quad$ **else**
9: $\quad\quad\quad \mathbf{N}^{(\ell+1)} \leftarrow \text{SUBDIVIDE}(\mathbf{N}^{(\ell)}, \mathbf{D}, \mathbf{S})$

This algorithm first generates a set of rays $\mathcal{R}$ (indexed by $i$) and stores them in an array $\mathbf{N}^{(0)}$ of ray-voxel pairs, which are proposals for ray-voxel intersections. We initialize each $\mathbf{N}_i^{(0)} \in \mathbf{N}^{(0)}$ with the root node, the octree's top-level voxel (line 2). Next, we iterate over the octree levels $\ell$ (line 3). In each iteration, we determine the ray-voxel pairs that result in intersections in DECIDE, which returns a list of *decisions* $\mathbf{D}$ with $\mathbf{D}_j = 1$ if the ray intersects the voxel and $\mathbf{D}_j = 0$ otherwise (line 4). Then, we use EXCLUSIVESUM to compute the exclusive sum $\mathbf{S}$ of list $\mathbf{D}$, which we feed into the next two subroutines (line 5). If we have not yet reached our desired LOD level $L$, we use SUBDIVIDE to populate the next list $\mathbf{N}^{(\ell+1)}$ with child voxels of those $\mathbf{N}_j^{(\ell)}$ that the ray intersects and continue the iteration (line 9). Otherwise, we use COMPACTIFY to remove all $\mathbf{N}_j^{(\ell)}$ that do not result in an intersection (line 7). The result is a compact, depth-ordered list of ray-voxel intersections for each level of the octree. Note that by analyzing the octant of space that the ray origin falls into inside the voxel, we can order the child voxels so that the list of ray-voxel pairs $\mathbf{N}^{(L)}$ will be ordered by distance to the ray origin.

**LOD Selection.** We choose the LOD $\tilde{L}$ for rendering with a depth heuristic, where $\tilde{L}$ transitions linearly with user-defined thresholds based on distance to object. More principled approaches exist [2], but we leave the details up to the user to choose an algorithm that best suits their needs.

# 4. Experiments

We perform several experiments to showcase the effectiveness of our architecture. We first fit our model to 3D mesh models from datasets including ShapeNet [5], Thingi10K [51], and select models from TurboSquid[1], and evaluate them based on both 3D geometry-based metrics as well as rendered image-space metrics. We also demonstrate that our model is able to fit complex analytic signed distance functions with unique properties from Shadertoy[2]. We additionally show results on real-time rendering, generalization to multiple shapes, and geometry simplification.

The MLP used in our experiments has only a single hidden layer with dimension $h = 128$ with a ReLU activation in the intermediate layer, thereby being significantly smaller and faster to run than the networks used in the baselines we compare against, as shown in our experiments. We use a SVO feature dimension of $m = 32$. We initialize voxel features $\mathbf{z} \in \mathcal{Z}$ using a Gaussian prior with $\sigma = 0.01$.

## 4.1. Reconstructing 3D Datasets

We fit our architecture on several different 3D datasets, to evaluate the quality of the reconstructed surfaces. We compare against baselines including DeepSDF [39], Fourier Feature Networks [47], SIREN [45], and Neural Implicits (NI) [9]. These architectures show state-of-the-art performance on overfitting to 3D shapes and also have source code available. We reimplement these baselines to the best

[1] https://www.turbosquid.com
[2] https://www.shadertoy.com

| | Storage (KB) | # Inference Param. | ShapeNet150 [5] gIoU ↑ | ShapeNet150 [5] Chamfer-$L^1$ ↓ | Thingi32 [51] gIoU ↑ | Thingi32 [51] Chamfer-$L^1$ ↓ | TurboSquid16 iIoU ↓ | TurboSquid16 Normal-$L^2$ ↓ | TurboSquid16 gIoU ↑ | TurboSquid16 Chamfer-$L^1$ ↓ |
|---|---|---|---|---|---|---|---|---|---|---|
| DeepSDF [39] | 7 186 | 1 839 614 | 86.9 | 0.316 | 96.8 | 0.0533 | 97.6 | 0.180 | 93.7 | 0.211 |
| FFN [47] | 2 059 | 526 977 | 88.5 | 0.077 | 97.7 | 0.0329 | 95.5 | 0.181 | 92.2 | 0.362 |
| SIREN [45] | 1 033 | 264 449 | 78.4 | 0.381 | 95.1 | 0.0773 | 92.9 | 0.208 | 82.1 | 0.488 |
| Neural Implicits [9] | **30** | 7 553 | 82.2 | 0.500 | 96.0 | 0.0919 | 93.5 | 0.211 | 82.7 | 0.354 |
| Ours / LOD 1 | 96 | 4 737 | 84.6 | 0.343 | 96.8 | 0.0786 | 91.9 | 0.224 | 79.7 | 0.471 |
| Ours / LOD 2 | 111 | 4 737 | 88.3 | 0.198 | 98.2 | 0.0408 | 94.2 | 0.201 | 87.3 | 0.283 |
| Ours / LOD 3 | 163 | 4 737 | 90.4 | 0.112 | 99.0 | 0.0299 | 96.1 | 0.184 | 91.3 | 0.162 |
| Ours / LOD 4 | 391 | 4 737 | 91.6 | 0.069 | 99.3 | 0.0273 | 97.1 | 0.170 | 94.3 | 0.111 |
| Ours / LOD 5 | 1 356 | 4 737 | **91.7** | **0.062** | **99.4** | **0.0271** | 98.3 | 0.166 | 95.8 | 0.085 |
| Ours / LOD 6 | 9 826 | 4 737 | – | – | – | – | **98.5** | **0.167** | **96.7** | **0.076** |

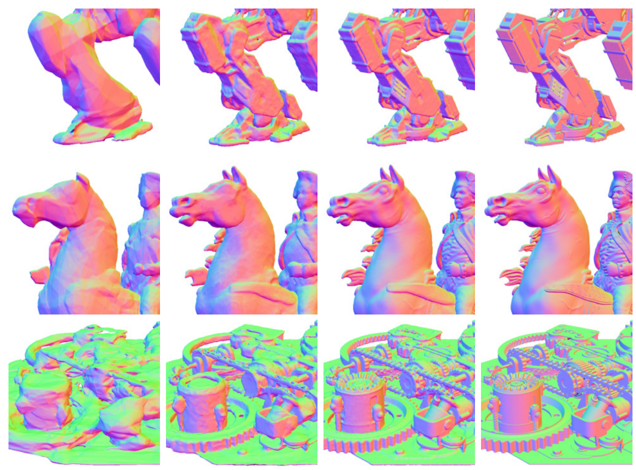
Table 1: **Mesh Reconstruction.** This table shows architectural and per-shape reconstruction comparisons against three different datasets. We see that under all evaluation schemes, our architecture starting from LOD 3 performs much better despite having much lower storage and inference parameters. The storage for our representation is calculated based on the average sparse voxel counts across all shapes in all datasets plus the decoder size, and *# Inference Param.* measures network parameters used for a single distance query.

of our ability using their source code as references, and provide details in the supplemental material.

**Mesh Datasets.** Table 1 shows overall results across ShapeNet, Thingi10K, and TurboSquid. We sample 150, 32, and 16 shapes respectively from each dataset, and overfit to each shape using 100, 100 and 600 epochs respectively. For ShapeNet150, we use 50 shapes each from the *car*, *airplane* and *chair* categories. For Thingi32, we use 32 shapes tagged as *scans*. ShapeNet150 and Thingi32 are evaluated using Chamfer-$L^1$ distance (multiplied by $10^3$) and intersection over union over the uniformly sampled points (gIoU). TurboSquid has much more interesting surface features, so we use both the 3D geometry-based metrics as well as image-space metrics based on 32 multi-view rendered images. Specifically, we calculate intersection over union for the segmentation mask (iIoU) and image-space normal alignment using $L^2$-distance on the mask intersection. The shape complexity roughly increases over the datasets. We train 5 LODs for ShapeNet150 and Thingi32, and 6 LODs for TurboSquid. For dataset preparation, we follow DualSDF [17] and normalize the mesh, remove internal triangles, and sign the distances with ray stabbing [38].

Storage (KB) corresponds to the sum of the decoder size and the representation, assuming 32-bit precision. For our architecture, the decoder parameters consist of 90 KB of the storage impact, so the effective storage size is smaller for lower LODs since the decoder is able to generalize to multiple shapes. The *# Inference Params.* are the number of parameters required for the distance query, which roughly correlates to the number of flops required for inference.

Across all datasets and metrics, we achieve state-of-the-art results. Notably, our representation shows better results starting at the third LOD, where we have minimal storage impact. We also note our inference costs are fixed at 4 737 floats across all resolutions, requiring 99% less inference parameters compared to FFN [47] and 37% less than Neural Implicits [9], while showing better reconstruction quality (see Figure 5 for a qualitative evaluation).



NI [9] FFN [47] Ours / LOD 6 Reference

Figure 5: **Comparison on TurboSquid.** We qualitatively compare the mesh reconstructions. Only ours is able to recover fine details, with speeds 50× faster than FFN and comparable to NI. We render surface normals to highlight geometric details.

**Special Case Analytic SDFs.** We also evaluate reconstruction on two particularly difficult analytic SDFs collected from Shadertoy. The *Oldcar* model is a highly non-metric SDF, which does not satisfy the Eikonal equation $|\nabla f| = 1$ and contains discontinuities. This is a critical case to handle, because non-metric SDFs are often exploited for special effects and easier modeling of SDFs. The *Mandelbulb* is a recursive fractal with infinite resolution. Both SDFs are defined by mathematical expressions, which we extract and sample distance values from. We train these analytic shapes for 100 epochs against $5 \times 10^6$ samples per epoch.

Only our architecture can capture the high-frequency details of these complex examples to reasonable accuracy. Notably, both FFN [47] and SIREN [45] seem to fail entirely; this is likely because both can only fit smooth distance fields and are unable to handle discontinuities and recursive struc-

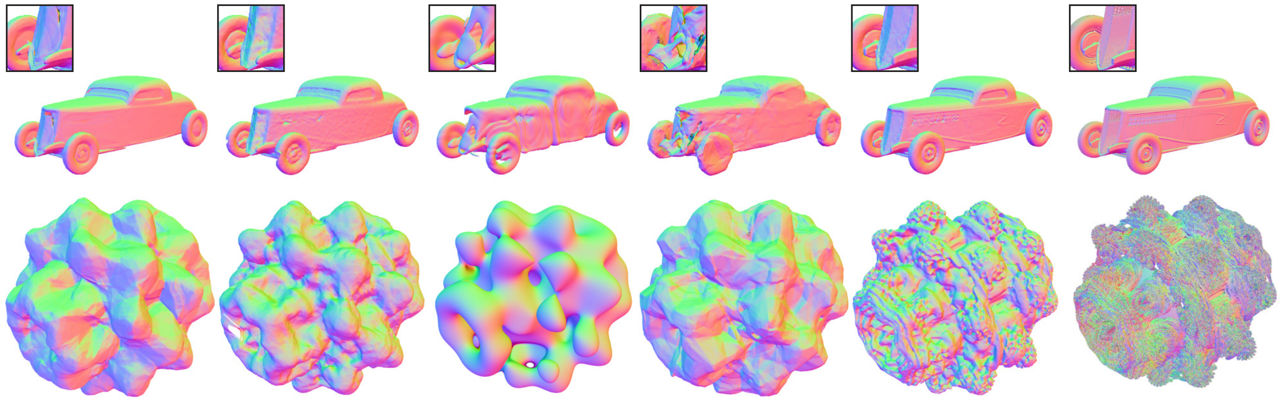



Figure 6: **Analytic SDFs.** We test against two difficult analytic SDF examples from Shadertoy; the *Oldcar*, which contains a highly non-metric signed distance field, as well as the *Mandelbulb*, which is a recursive fractal structure that can only be expressed using implicit surfaces. Only our architecture can reasonably reconstruct these hard cases. We render surface normals to highlight geometric details.

| Method / LOD | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| DeepSDF (100 epochs) [39] | 0.0533 | 0.0533 | 0.0533 | 0.0533 | 0.0533 |
| FFN (100 epochs) [47] | **0.0329** | **0.0329** | 0.0329 | 0.0329 | 0.0329 |
| Ours (30 epochs) | 0.1197 | 0.0572 | 0.0345 | 0.0285 | 0.0278 |
| Ours (30 epochs, pretrained) | 0.1018 | 0.0499 | 0.0332 | 0.0287 | 0.0279 |
| Ours (100 epochs) | 0.0786 | 0.0408 | **0.0299** | **0.0273** | **0.0271** |

Table 2: **Chamfer-$L^1$ Convergence.** We evaluate the performance of our architecture on the Thingi32 dataset under different training settings and report faster convergence for higher LODs.

tures. See Figure 6 for a qualitative comparison.

**Convergence.** We perform experiments to evaluate training convergence speeds of our architecture. Table 2 shows reconstruction results on Thingi32 on our model fully trained for 100 epochs, trained for 30 epochs, and trained for 30 epochs from pretrained weights on the Stanford Lucy statue (Figure 1). We find that our architecture converges quickly and achieves better reconstruction even with roughly 45% the training time of DeepSDF [39] and FFN [47], which are trained for the full 100 epochs. Finetuning from pretrained weights helps with lower LODs, but the difference is small. Our representation swiftly converges to good solutions.

## 4.2. Rendering Performance

We also evaluate the inference performance of our architecture, both with and without our rendering algorithm. We first evaluate the performance using a naive Python-based sphere tracing algorithm in PyTorch [40], with the same implementation across all baselines for fair comparison. For the Python version of our representation, we store the features on a dense voxel grid, since a naive sphere tracer cannot handle sparsity. For the optimized implementation, we show the performance of our representation using a renderer implemented using libtorch [40], CUB [32], and CUDA.

Table 3 shows frametimes on the TurboSquid *V Mech* scene with a variety of different resolutions. Here, we measure frametime as the CUDA time for the sphere trace and normal computation. The *# Visible Pixels* column shows the number of pixels occupied in the image by the model. We see that both our naive PyTorch renderer and sparse-optimized CUDA renderer perform better than the baselines. In particular, the sparse frametimes are more than $100\times$ faster than DeepSDF while achieving better visual quality with less parameters. We also notice that our frametimes decrease significantly as LOD decreases for our naive renderer but less so for our optimized renderer. This is because the bottleneck of the rendering is not in the ray-octree intersect—which is dependent on the number of voxels—but rather in the MLP inference and miscellaneous memory I/O. We believe there is still significant room for improvement by caching the small MLP decoder to minimize data movement. Nonetheless, the lower LODs still benefit from lower memory consumption and storage.

## 4.3. Generalization

We now show that our surface extraction mechanism can generalize to multiple shapes, even from being trained on a *single* shape. This is important because loading distinct weights per object as in [9, 45] incurs large amounts of memory movement, which is expensive. With a general surface extraction mechanism, the weights can be pre-loaded and multi-resolution voxels can be streamed-in on demand.

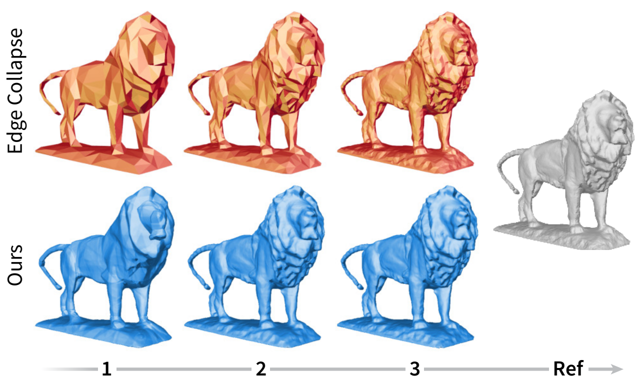
Table 4 shows results on Thingi32. DeepSDF [39], FFN [47] and Ours (overfit) are all overfit per shape. Ours (general) first overfits the architecture on the Stanford Lucy model, fixes the surface extraction network weights, and trains only the sparse features. We see that our representation fares better, even against large networks that are overfitting to each specific shape examples. At the lowest LOD, the surface extractor struggles to reconstruct good surfaces, as expected; the features become increasingly high-level and complex for lower LODs.

| | | Frametimes (ms) / Improvement Factor | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Resolution | # Visible Pixels | DeepSDF [39] | FFN [47] | SIREN [45] | NI [9] | Ours (N) / LOD 4 | Ours (N) / LOD 6 | Ours (S) / LOD 4 | Ours (S) / LOD 6 |
| $640 \times 480$ | 94 624 | 1 693 / 57× | 1 058 / 35× | 595 / 20× | 342 / 11× | 164 / 5× | 315 / 11× | 28 | 30 / 1× |
| $1\,280 \times 720$ | 213 937 | 4 901 / 96× | 2 760 / 54× | 1 335 / 26 × | 407 / 8× | 263 / 5× | 459 / 9× | 50 | 51 / 1× |
| $1\,920 \times 1\,080$ | 481 828 | 10 843 / 119× | 5 702 / 62× | 2 946 / 32× | 701 / 8× | 473 / 5× | 784 / 9× | 93 | 91 / 1× |

Table 3: **Rendering Frametimes.** We show runtime comparisons between different representations, where (N) and (S) correspond to our naive and sparse renderers, respectively. We compare baselines against Ours (Sparse) at LOD 6. # Visible Pixels shows the number of pixels occupied by the benchmarked scene (TurboSquid *V Mech*), and frametime measures ray-tracing and surface normal computation.

| | Chamfer-$L^1$ ↓ | | | | |
|---|---|---|---|---|---|
| Method / $L$ | 1 | 2 | 3 | 4 | 5 |
| DeepSDF (overfit per shape) [39] | 0.0533 | 0.0533 | 0.0533 | 0.0533 | 0.0533 |
| FFN (overfit per shape) [47] | 0.0322 | 0.0322 | 0.0322 | 0.0322 | 0.0322 |
| Ours (overfit per shape) | 0.0786 | 0.0408 | 0.0299 | 0.0273 | 0.0271 |
| Ours (general) | 0.0613 | 0.0378 | 0.0297 | 0.0274 | 0.0272 |

Table 4: **Generalization.** We evaluate generalization on Thingi32. Ours (general) freezes surface extractor weights pretrained on a *single* shape, and only trains the feature volume. Even against large overfit networks, we perform better at high LODs.




| | $L$ | gIoU ↑ | Chamfer-$L^1$ ↓ | iIoU ↑ | Normal-$L^2$ ↓ |
|---|---|---|---|---|---|
| | 1 | 94.4 | 0.052 | 97.4 | 0.096 |
| Decimation [15] | 2 | 97.4 | 0.026 | 98.7 | 0.069 |
| | 3 | 99.1 | 0.019 | 99.5 | 0.044 |
| | 1 | 96.0 / +1.6 | 0.063 / +0.011 | 96.4 / −1.0 | 0.044 / −0.052 |
| Ours | 2 | 97.8 / +0.4 | 0.030 / +0.004 | 97.6 / −1.1 | 0.035 / −0.034 |
| | 3 | 98.8 / −0.3 | 0.023 / +0.004 | 98.2 / −1.3 | 0.030 / −0.014 |

Figure 7: **Comparison with Mesh Decimation.** At low memory budgets, our model is able to maintain visual details better than mesh decimation, as seen from lower normal-$L^2$ error.

### 4.4. Geometry Simplification

In this last experiment, we evaluate how our low LODs perform against classic mesh decimation algorithms, in particular edge collapse [15] in libigl [21]. We compare against mesh decimation instead of mesh compression algorithms [1] because our model can also benefit from compression and *mesh decoding* incurs additional runtime cost. We first evaluate our memory impact, which $M = (m+1)\,|\mathcal{V}_{1:L_{\max}}|$ bytes where $m+1$ is the feature dimension along with the $Z$-order curve [35] for indexing, and $|\mathcal{V}_{1:L_{\max}}|$ is the octree size. We then calculate the face budget as $F = M/3$ since the connectivity can be arbitrary. As such, we choose a conservative budget to benefit the mesh representation.

Figure 7 shows results on the Lion statue from Thingi32. We see that as the memory budget decreases, the relative advantage on the perceptual quality of our method increases, as evidenced by the image-based normal error. The SDF can represent smooth features easily, whereas the mesh suffers from discretization errors as the face budget decreases. Our representation can also smoothly blend between LODs by construction, something difficult to do with meshes.

## 5. Limitations and Future Work

In conclusion, we introduced *Neural Geometric LOD*, a representation for implicit 3D shapes that achieves state-of-the-art geometry reconstruction quality while allowing real-time rendering with acceptable memory footprint. Our model combines a small surface extraction neural network with a sparse-octree data structure that encodes the geometry and naturally enables LOD. Together with a tailored sphere tracing algorithm, this results in a method that is both computationally performant and highly expressive.

Our approach heavily depends on the point samples used during training. Therefore, scaling our representation to extremely large scenes or very thin, volume-less geometry is difficult. Furthermore, we are not able to easily animate or deform our geometry using traditional methods. We identify these challenges as promising directions for future research. Nonetheless, we believe our model represents a major step forward in neural implicit function-based geometry, being, to the best of our knowledge, the first representation of this kind that can be rendered and queried in real-time. We hope that it will serve as an important component for many downstream applications, such as scene reconstruction, ultra-precise robotics path planning, interactive content creation, and more.

**Acknowledgements.** We thank Jean-Francois Lafleche, Peter Shirley, Kevin Xie, Jonathan Granskog, Alex Evans, and Alex Bie for interesting discussions throughout the project. We also thank Jacob Munkberg, Peter Shirley, Alexander Majercik, David Luebke, Jonah Philion, and Jun Gao for help with paper review.

## References


[1] Pierre Alliez and Craig Gotsman. Recent advances in compression of 3D meshes. In *Advances in Multiresolution for Geometric Modelling*, pages 3–26. Springer, 2005. 8

[2] John Amanatides. Ray tracing with cones. *ACM SIGGRAPH Computer Graphics*, 18(3):129–135, 1984. 5

[3] Guy E. Blelloch. *Vector models for data-parallel computing*. MIT Press, 1990. 12

[4] Rohan Chabra, Jan Eric Lenssen, Eddy Ilg, Tanner Schmidt, Julian Straub, Steven Lovegrove, and Richard Newcombe. Deep local shapes: Learning local SDF priors for detailed 3D reconstruction. *arXiv preprint arXiv:2003.10983*, 2020. 3

[5] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, et al. ShapeNet: An information-rich 3D model repository. *arXiv preprint arXiv:1512.03012*, 2015. 5, 6

[6] Zhiqin Chen, Andrea Tagliasacchi, and Hao Zhang. BSP-Net: Generating compact meshes via binary space partitioning. In *IEEE Conf. Comput. Vis. Pattern Recog.*, 2020. 3

[7] Zhiqin Chen and Hao Zhang. Learning implicit fields for generative shape modeling. In *IEEE Conf. Comput. Vis. Pattern Recog.*, June 2019. 1, 3

[8] Massimiliano Corsini, Mohamed-Chaker Larabi, Guillaume Lavoué, Oldřich Petřík, Libor Váša, and Kai Wang. Perceptual metrics for static and dynamic triangle meshes. In *Computer Graphics Forum*, volume 32, pages 101–125, 2013. 2

[9] Thomas Davies, Derek Nowrouzezahrai, and Alec Jacobson. Overfit neural networks as a compact shape representation. *arXiv preprint arXiv:2009.09808*, 2020. 1, 2, 3, 5, 6, 7, 8, 11, 14, 15, 16

[10] Boyang Deng, Kyle Genova, Soroosh Yazdani, Sofien Bouaziz, Geoffrey Hinton, and Andrea Tagliasacchi. CvxNet: Learnable convex decomposition. In *IEEE Conf. Comput. Vis. Pattern Recog.*, June 2020. 3

[11] Yueqi Duan, Haidong Zhu, He Wang, Li Yi, Ram Nevatia, and Leonidas J. Guibas. Curriculum DeepSDF. *arXiv preprint arXiv:2003.08593*, 2020. 3

[12] Alex Evans. Fast approximations for global illumination on dynamic scenes. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, page 153171, 2006. 2, 11

[13] Sarah F. Frisken, Ronald N. Perry, Alyn P. Rockwood, and Thouis R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th annual Conference on Computer Graphics and Interactive Techniques*, pages 249–254, 2000. 2

[14] Jun Gao, Wenzheng Chen, Tommy Xiang, Alec Jacobson, Morgan McGuire, and Sanja Fidler. Learning deformable tetrahedral meshes for 3D reconstruction. In *Adv. Neural Inform. Process. Syst.*, 2020. 3

[15] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, page 209216, 1997. 2, 8

[16] Kyle Genova, Forrester Cole, Avneesh Sud, Aaron Sarna, and Thomas Funkhouser. Local deep implicit functions for 3D shape. In *IEEE Conf. Comput. Vis. Pattern Recog.*, June 2020. 3

[17] Zekun Hao, Hadar Averbuch-Elor, Noah Snavely, and Serge Belongie. DualSDF: Semantic shape manipulation using a two-level representation. *arXiv preprint arXiv:2004.02869*, 2020. 6

[18] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007. 12

[19] John C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996. 1, 3, 4

[20] Hugues Hoppe. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 99108, 1996. 2

[21] Alec Jacobson, Daniele Panozzo, et al. libigl: A simple C++ geometry processing library, 2018. https://libigl.github.io. 8

[22] Yue Jiang, Dantong Ji, Zhizhong Han, and Matthias Zwicker. SDFDiff: Differentiable rendering of signed distance fields for 3D shape optimization. In *IEEE Conf. Comput. Vis. Pattern Recog.*, 2020. 3

[23] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 11

[24] Michéal Larkin and Carol O'Sullivan. Perception of simplification artifacts for animated characters. In *Proceedings of the ACM SIGGRAPH Symposium on Applied Perception in Graphics and Visualization*, pages 93–100, 2011. 2

[25] Peter Lindstrom and Greg Turk. Image-driven simplification. *ACM Trans. Graph.*, 19(3):204241, July 2000. 2

[26] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. Neural sparse voxel fields. *Adv. Neural Inform. Process. Syst.*, 2020. 2, 3

[27] Shaohui Liu, Yinda Zhang, Songyou Peng, Boxin Shi, Marc Pollefeys, and Zhaopeng Cui. DIST: Rendering deep implicit signed distance function with differentiable sphere tracing. In *IEEE Conf. Comput. Vis. Pattern Recog.*, 2020. 3

[28] 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 163169, 1987. 2, 3, 11

[29] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers Inc., 2002. 2

[30] 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. 4, 11

[31] Ravish Mehra, Qingnan Zhou, Jeremy Long, Alla Sheffer, Amy Gooch, and Niloy J. Mitra. Abstraction of man-made shapes. *ACM Trans. Graph.*, 28(5):110, Dec. 2009. 2

[32] Duane Merrill. CUB: a library of warp-wide, block-wide, and device-wide GPU parallel primitives, 2017. 5, 7, 11

[33] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3D reconstruction in function space. In *IEEE Conf. Comput. Vis. Pattern Recog.*, June 2019. 1, 3

[34] 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 *Eur. Conf. Comput. Vis.*, 2020. 2, 3

[35] Guy M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. *Technical Report*, 1966. 8

[36] Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3D representations without 3D supervision. In *IEEE Conf. Comput. Vis. Pattern Recog.*, 2020. 3

[37] 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. 12

[38] Fakir S. Nooruddin and Greg Turk. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):191205, Apr. 2003. 6

[39] 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. 1, 2, 3, 4, 5, 6, 7, 8, 11, 14, 15, 16

[40] 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. 7, 11

[41] 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. 12

[42] Shunsuke Saito, Zeng Huang, Ryota Natsume, Shigeo Morishima, Angjoo Kanazawa, and Hao Li. PIFu: Pixel-aligned implicit function for high-resolution clothed human digitization. In *Int. Conf. Comput. Vis.*, pages 2304–2314, 2019. 3

[43] Shunsuke Saito, Tomas Simon, Jason Saragih, and Hanbyul Joo. PIFuHD: Multi-level pixel-aligned implicit function for high-resolution 3D human digitization. In *IEEE Conf. Comput. Vis. Pattern Recog.*, pages 84–93, 2020. 3

[44] Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *Advances in neural information processing systems*, 29:901–909, 2016. 11

[45] 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. 3, 5, 6, 7, 8, 11, 14, 15, 16

[46] 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. 11

[47] 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. 3, 5, 6, 7, 8, 11, 14, 15, 16

[48] Edgar Tretschk, Ayush Tewari, Vladislav Golyanik, Michael Zollhfer, Carsten Stoll, and Christian Theobalt. PatchNets: Patch-based generalizable deep implicit 3D shape representations. *arXiv preprint arXiv:2008.01639*, 2020. 3

[49] Lance Williams. Pyramidal parametrics. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, pages 1–11, 1983. 2

[50] Lior Yariv, Yoni Kasten, Dror Moran, Meirav Galun, Matan Atzmon, Ronen Basri, and Yaron Lipman. Multiview neural surface reconstruction with implicit lighting and material. *Adv. Neural Inform. Process. Syst.*, 2020. 3

[51] Qingnan Zhou and Alec Jacobson. Thingi10K: A dataset of 10,000 3D-printing models. *arXiv preprint arXiv:1605.04797*, 2016. 5, 6

# 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 $L_{\max} \in \{5, 6\}$ 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 [40] with some operations implemented in CUDA. All models are trained with the Adam optimizer [23] 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 (4)). 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 $\ell = L_{\max} - 1, L_{\max} - 2, \ldots$ 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.

### A.4. Rendering

We implement our baseline renderer using Python and PyTorch. The sparse renderer is implemented using CUDA, cub [32], and libtorch [40]. 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. [30]. 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 [12] and materials through matcaps [46]. 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 [39], Fourier Feature Network (FFN) [47], SIREN [45], and Neural Implicits [9]. 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 [44], 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 $\sigma$ 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 [28] 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 [41].

**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 $512 \times 512$ 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 [37].

## 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($\mathcal{R}, \mathbf{N}^{(\ell)}, \ell$)<br>
2:     **for all** $t \in \{0, \dots, |\mathbf{N}^{(\ell)}| - 1\}$ **do in parallel**<br>
3:         $\{i, j\} \leftarrow \mathbf{N}_t^{(\ell)}$<br>
4:         **if** $\mathcal{R}_i \cap V_j^{(\ell)}$ **then**<br>
5:             **if** $\ell = L$ **then**<br>
6:                 $\mathbf{D}_t \leftarrow 1$<br>
7:             **else**<br>
8:                 $\mathbf{D}_t \leftarrow$ NUMCHILDREN($V_j^{\ell}$)<br>
9:         **else**<br>
10:             $\mathbf{D}_t \leftarrow 0$<br>
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 [3, 18]. 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($\mathbf{N}^{(\ell)}, \mathbf{D}, \mathbf{S}$)<br>
2:     **for all** $t \in \{0, \dots, |\mathbf{N}^{(\ell)}| - 1\}$ **do in parallel**<br>
3:         **if** $\mathbf{D}_t = 1$ **then**<br>
4:             $k \leftarrow \mathbf{S}_t$<br>
5:             $\mathbf{N}_k^{(\ell+1)} \leftarrow \mathbf{N}_t^{(\ell)}$<br>
6:     **return** $\mathbf{N}^{(\ell+1)}$

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($\mathbf{N}^{(\ell)}, \mathbf{D}, \mathbf{S}$)<br>
2:     **for all** $t \in \{0, \dots, |\mathbf{N}^{(\ell)}| - 1\}$ **do in parallel**<br>
3:         **if** $\mathbf{D}_t \neq 0$ **then**<br>
4:             $\{i, j\} \leftarrow \mathbf{N}_t^{(\ell)}$<br>
5:             $k \leftarrow \mathbf{S}_t$<br>
6:             **for** $c \in$ ORDEREDCHILDREN($\mathcal{R}_i, V_j^{(\ell)}$) **do**<br>
7:                 $\mathbf{N}_k^{(\ell+1)} \leftarrow \{i, c\}$<br>
8:                 $k \leftarrow k + 1$<br>
9:     **return** $\mathbf{N}^{\ell+1}$

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 $8 \times 8$ 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*).

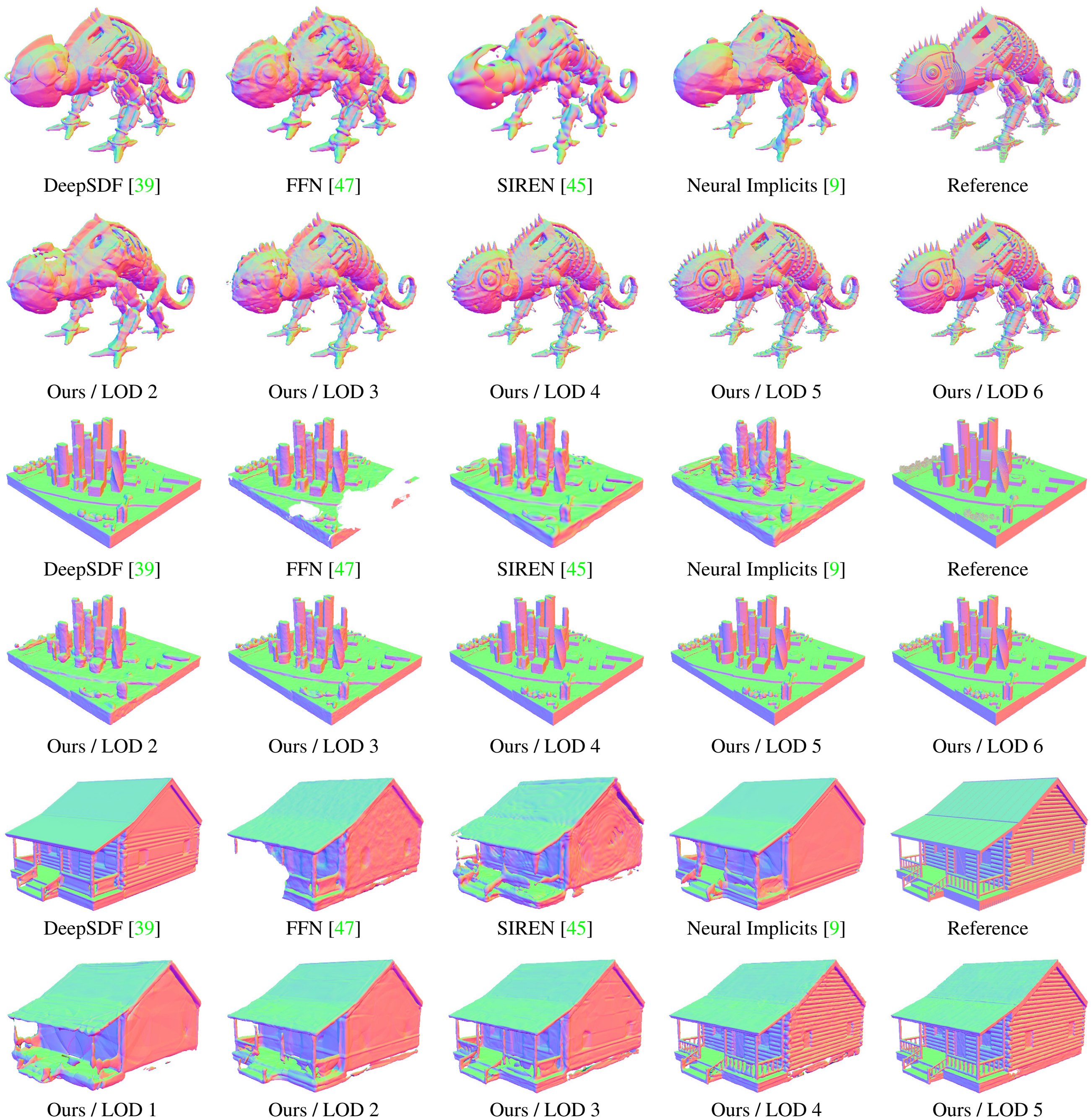



Figure 8: **Additional TurboSquid16 Results.** FFN exhibits white patch artifacts (e.g. City and Cabin) because it struggles to learn a conservative metric SDF, resulting in the sphere tracing algorithm missing the surface entirely. Best viewed zoomed in.

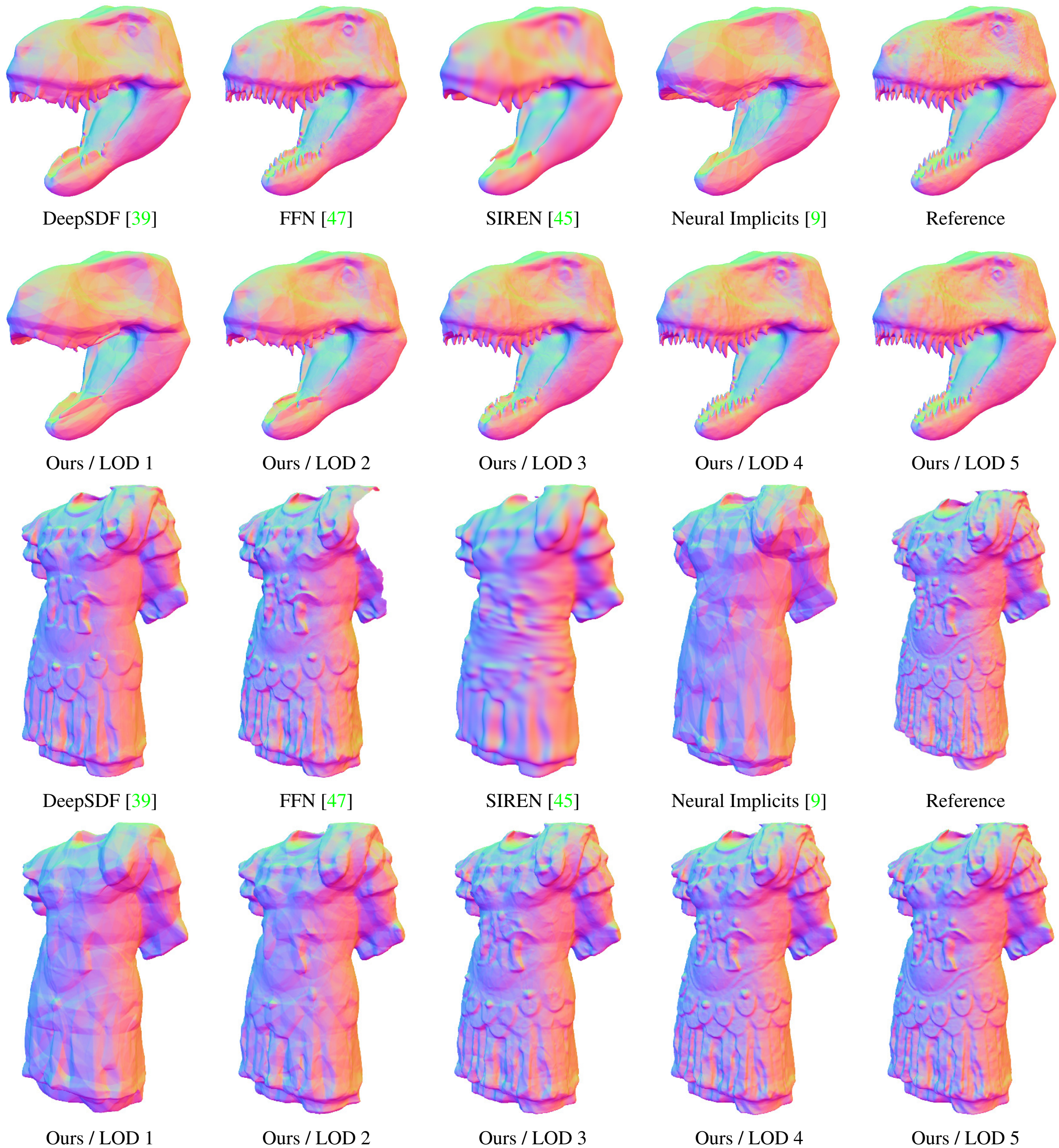



Figure 9: **Additional Thingi32 Results.** Best viewed zoomed in.

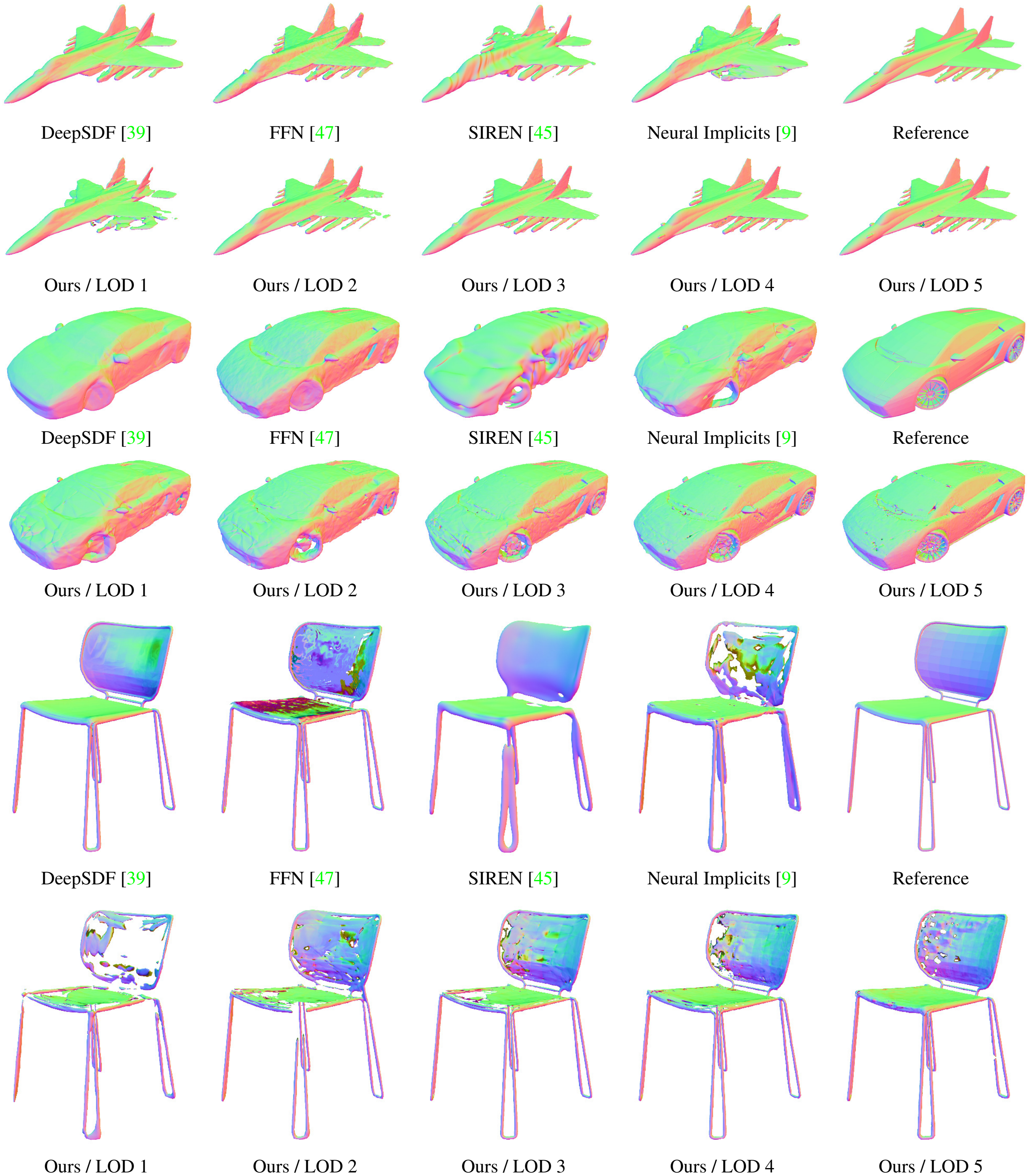



Figure 10: **Additional ShapeNet150 Results.** Our method struggles with thin flat features with little to no volume, such as Jetfighter wings and the back of the Chair. Best viewed zoomed in.