Adaptive Surface Reconstruction with Multiscale Convolutional Kernels

Benjamin Ummenhofer Vladlen Koltun
Intel Labs
benjamin.ummenhofer@intel.com

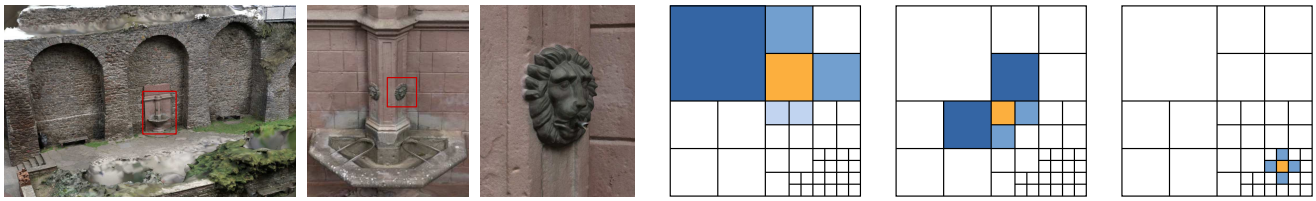


Figure 1: **Left:** Reconstruction of a scene with varying scale. The surface triangle mesh adapts to the scale to capture details such as the fountain. **Right:** Like the surface mesh, our volume discretization adapts to the available information and our multiscale convolutional kernels adapt to the grid, allowing us to efficiently infer and learn an implicit representation of the surface with ConvNets.

Abstract

We propose generalized convolutional kernels for 3D reconstruction with ConvNets from point clouds. Our method uses multiscale convolutional kernels that can be applied to adaptive grids as generated with octrees. In addition to standard kernels in which each element has a distinct spatial location relative to the center, our elements have a distinct relative location as well as a relative scale level. Making our kernels span multiple resolutions allows us to apply ConvNets to adaptive grids for large problem sizes where the input data is sparse but the entire domain needs to be processed. Our ConvNet architecture can predict the signed and unsigned distance fields for large data sets with millions of input points and is faster and more accurate than classic energy minimization or recent learning approaches. We demonstrate this in a zero-shot setting where we only train on synthetic data and evaluate on the Tanks and Temples dataset of real-world large-scale 3D scenes.

1. Introduction

Generating a description of the surface of objects or whole scenes is a key problem in 3D reconstruction. While the acquisition of images and scans becomes easier and easier, combining this information into a global and consistent 3D structure becomes more difficult with the increasing size of the datasets. However, large datasets are particularly interesting as they can digitize our 3D world and enable applications like navigation or virtual sightseeing.

An important part of many 3D pipelines is volumetric fusion, which fuses partial observations into a global 3D description. In this approach, the problem of finding the 2D surface is turned into finding a 3D scalar field from which the surface can be extracted as a level set. An inherent problem of this approach is the cubic growth of the volume leading to high computational costs. Another difficulty arises from noisy input data that requires the use of good priors in the fusion process.

To tackle these challenges many works have proposed to use specialized adaptive data structures like octrees to store 3D grids more efficiently and adapt algorithms to directly operate on these structures. These types of algorithms are often highly specialized and therefore have high engineering costs. They also often employ PDEs, which implement rather simple priors that prefer surfaces with minimum curvature or minimum area.

In contrast to this are learning approaches that can learn complex priors from data, which makes them well suited for the fusion task. Especially ConvNets have become a standard method in image processing pipelines due to their flexibility and efficiency for data laid out in regular 2D grids. While ConvNets naturally generalize to 3D data they also suffer from the cubic growth in complexity.

To make use of ConvNets for volumetric fusion, we propose to generalize the standard convolutional kernels to adaptive grids. Adaptive grids not only allow to efficiently store data but also allow to capture information at different scales as shown in Figure 1, which is important for the reconstruction of large datasets where some regions are more

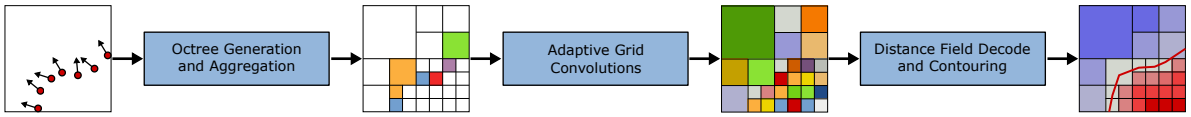


Figure 2: The input for our method is an oriented point cloud. We aggregate the information in an adaptive grid and use our multiscale convolutional kernels to compute distance functions. In the last stage we decode the distance functions and extract the zero-level set.

important. We design multiscale convolutional kernels that span multiple scales. Compared to regular kernels where each element has a distinct spatial position relative to the center, our kernel elements additionally have a relative scale with respect to the center element allowing the network to learn spatial and scale relationships on adaptive grids. We show that we can use our convolutions with a simple U-Net-like architecture to learn an end-to-end trainable volumetric fusion pipeline that computes the signed and unsigned distance field.

Our approach achieves significantly better reconstructions than classic analytical volumetric fusion approaches and recent learning-based methods. We demonstrate this qualitatively and quantitatively in a zero-shot generalization setting on real-world datasets from Tanks and Temples [16] and [8]. In addition, we can show that our method can reconstruct large datasets with hundreds of millions of points and is more than two times faster than the baselines.

2. Related Work

Analytical volumetric fusion. Early approaches like [6] use regular voxel grids and simple averaging to fuse information from multiple scanners to create an implicit representation of the surface, which then can be extracted with marching cubes [19]. This concept has been evolved with adaptive spatial data structures, PDE-based priors and sophisticated optimization algorithms by [14, 2, 18, 17, 30] and many others. We compare to Poisson Surface Reconstruction (PSR) [14], which is the most well-known method, and Smoothed Signed Distance Reconstruction (SSD) [2] in our experiments. Both methods use octrees as spatial data structures and simple PDE-based priors. Methods for fusing datasets with multiple scales have been proposed in [8, 9, 27]. These methods make use of scale information of the input data for steering the generation of the spatial data structure and the fusion process. We adopt the use of scale information and compare to [27] in our evaluation.

Learned surface reconstruction. Learning global implicit functions for representing surfaces has been proposed by [20, 21]. Both works use MLPs to encode the implicit function of shapes of entire objects. The MLPs allow querying the value of the implicit function at arbitrary locations allowing to sample the function at arbitrary resolution after training. [12] and [3] combine the implicit function networks with regular grids and learn to represent local shapes. Jiang *et al*. [12] use an autoencoder to learn the latent code of local shapes and the decoder. Chabra *et al*. [3] directly learn the code and decoder parameters. We adopt the representation of local shapes in our network but instead of optimizing for the latent code during inference, our ConvNet generates for each voxel a code in the forward pass describing the local distances to the surface. This is also similar to [22] combining the occupancy network in [20] with a convolutional network. Peng *et al*. [22] use a regular 3D ConvNet and interpolate features within the grid to evaluate the implicit function, while we use adaptive grids and extrapolate the local functions.

Our method and many other surface reconstruction methods assume per-point normal information to correctly compute the sign or occupancy value for the implicit surface representation. Recent exceptions to this are [1, 7, 4]. Atzmon and Lipman[1] shows that networks can be trained to infer the sign of a distance function from unsigned data and demonstrate this on object-centric datasets. Erler *et al*. [7] uses a patch-based approach which samples additional global points to support the prediction of the correct sign. Their method demonstrates good generalization performance to unseen and real datasets. We use their method as a baseline in our evaluation. Chibane *et al*. [4] propose to use the unsigned distance function as implicit surface representation allowing them to represent open shapes that do not have a well-defined inside or outside. However, extracting an explicit surface representation from unsigned distance fields is more involved and comes at a higher computational cost. Our method predicts the signed as well as the unsigned distance function. We make use of the simplicity of detecting the sign change in the signed distance function and use the unsigned distance function to limit the surface generation to regions that are predicted to be close to the surface.

Learning on sparse data structures. Riegler *et al*. [24] propose to use octrees to accelerate convolutions on sparse 3D grids and show an application for depth map fusion in [23]. While their work aims to accelerate regular convolutions by avoiding redundant computation between voxels of different size, our method adapts the convolution kernel to account for different voxel sizes and incorporate geometric information from multiple scales. Klokov and Lempitsky [15] define networks on kd-trees for segmentation and classification. They assign learnable weights to each split operation in a kd-tree to recursively combine information of child nodes. In contrast to our method, which allows us to define convolutions within an adaptive grid, the oper-

ations in Kd-Networks are only defined between different depth levels, which is more limiting as it strongly couples the network architecture to the tree depth. [11, 5] implement submanifold sparse convolutional networks for sparse N-dimensional grids, which can avoid computation on empty space but would not be suitable for our task where datasets span multiple resolution levels.

3. Overview

Our approach is a pipeline with three stages as depicted in Figure 2, which we summarize here.

Octree Generation and Feature Aggregation. The input to our method is a point cloud with per-point normal information. To efficiently process the input data we aggregate information in an adaptive grid. We, therefore, start our pipeline by building a face-balanced octree. Working with balanced octrees allows us to define efficient convolution kernels for the next stage. To steer the subdivision of the volume, we use the scale information associated with the input points or estimate the scale from the point density. After building the octree, we extract grids at multiple resolutions starting with the leaf nodes and walking up the tree hierarchy. We then aggregate features from the point cloud in the grid with the highest resolution using a continuous convolution. Continuous convolutions define a continuous kernel and allow to process points at arbitrary positions. We define them in Section 4.

Adaptive Grid Convolutions. The main stage of our approach applies the U-Net architecture shown in Figure 3 for processing the aggregated information from the point cloud. The output of this stage are features that encode local distance functions describing the sought surface. Convolutions throughout the network work on the adaptive grids and use multiscale convolutional kernels that contain elements for voxels at multiple resolution levels. Since all grids are face-balanced our kernels are compact and require only a small number of elements. Similar to our grids adapting to the input data, the kernels adapt to the grid. Moreover, unlike standard convolutions, not all kernel elements are active at all grid locations allowing us to efficiently process large grids. We formally define the convolutions in Section 5.

Distance Field Decode and Contouring. The last stage decodes the features generated from our ConvNet to obtain signed and unsigned distances to the surface for each voxel in the grid with the highest resolution. We use an MLP for the decoding that allows us to query the distance values at arbitrary points in the vicinity of the voxel center. Further, we can use the MLP to query the gradient of the distance values providing additional information for the contouring. For the contouring, we implement dual contouring for adaptive grids, which generates a triangle mesh for the zero-level set of the signed distance field. We use the unsigned distance values to restrict vertex and triangle generation to regions near the surface. We give more details in Section 6.

4. Octree Generation and Feature Aggregation

We aggregate information from the input point cloud in an adaptive grid based on an octree. To generate the octree we follow a similar strategy to [27]. We assign to each input point a footprint size σ and use it to steer the subdivision of the octree such that the edge length l of the voxel containing the point is smaller than σ. The footprint size of the point relates to the scale of the measurement. For points generated from MVS methods, this is commonly defined by the camera intrinsics – the sensor pixel size – and the measured depth. For a pinhole camera we use $\sigma = \frac{d}{f}$, where d is the distance to the optical plane and f is the focal length. For point clouds for which the footprint size information is unknown, we define the footprint size of point i as the radius of a sphere centered at the point's position $\mathbf{x}_i$ such that the sphere encapsulates k neighboring points, i.e., $\sigma_i = \max_{j \in \mathcal{N}_k(\mathbf{x}_i)}(\|\mathbf{x}_i - \mathbf{x}_j\|_2)$. Given the footprint size, we discard obvious outliers based on a density threshold and build a linear octree with location keys [10]. To this end, we assign a location key to each point and collect all unique keys. The tree depth d for each point is chosen such that $\arg\min_d(\sigma) > \frac{L}{2^d}$ with L as the edge length of the cubic bounding box. To ensure that the entire domain is covered by the octree, we create all missing parent nodes up to the root node and in a second pass eliminate mixed nodes by creating the missing child nodes. Since our convolution kernels require a face-balanced octree we also subdivide nodes that violate this property as illustrated in Figure 4 until all nodes fulfill the condition. This process increases the number of nodes by a factor of 1.12 on average on our evaluation data.

After constructing the octree, we aggregate information from the input point cloud into the leaf nodes. We make use of continuous convolutions [28], which allow us to perform a convolution between two point clouds with points at arbitrary positions. Our two point clouds are the input points with normal information as depicted in the first stage in Figure 3 and the voxel centers of the leaf nodes of the octree. Following [28] we define the convolution at $\mathbf{x}_j$, the center of the voxel j, as

$$(f * g)(\mathbf{x}_j) = \frac{1}{\psi_j} \sum_{i \in \mathcal{N}(\mathbf{x}_j, R)} a_{ij}\ f_i\ g(\Lambda(\mathbf{x}_i - \mathbf{x}_j)). \quad (1)$$

The feature f_i of input point i is a vector with the normal information. $\mathcal{N}(\mathbf{x}_j, R)$ is the set of input points within a radius R around $\mathbf{x}_j$. The radius R depends on the voxel j and its edge length l_j; we set $R = l_j/2$. a_{ij} is a scalar function that defines the importance of an input point with respect

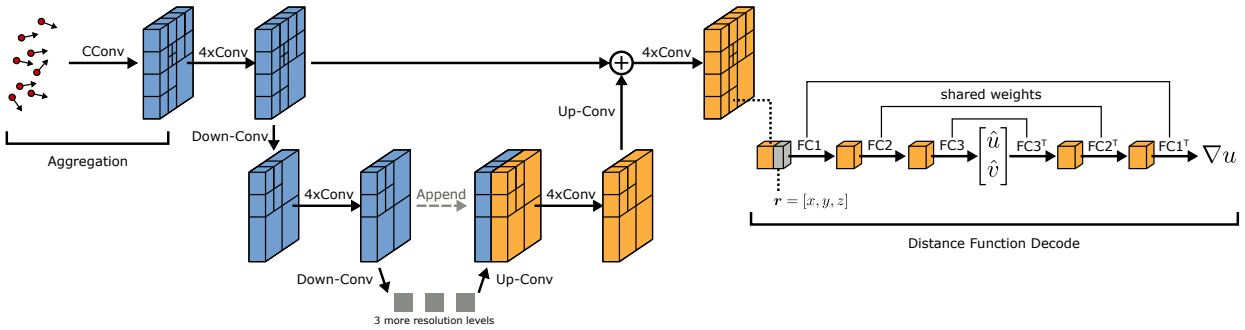


Figure 3: Schematic overview of our network architecture. We give detailed information about parameters in the supplementary material. We use a continuous convolution to aggregate the information from the oriented point set to the finest adaptive grid. We then employ a U-Net-like architecture to process the hierarchy of adaptive grids and generate a code for each voxel that encodes a local distance function, which gives us an implicit representation of the surface. To evaluate the implicit functions $\hat{u}$ and $\hat{v}$ for a voxel we use coordinates $\mathbf{r}$ that are relative to the voxel size and center and a small MLP decoder with 3 layers. Since the decoder is differentiable, we can add operations from the backward pass to the network, visualized here as transposed layers, to compute the gradient of the signed distance ∇u, giving us a decoder with a total of 6 layers of which 3 pairs share weights.

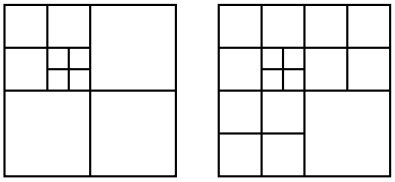


Figure 4: Left: Unbalanced quadtree. Right: Quadtree after balancing. The upper-right and lower-left cells have been subdivided to make the tree face-balanced. The depth difference between face-adjacent nodes is at most 1.

to the voxel that we want to aggregate the information into. We define the importance of a sample based on the compatibility of the scale between the point and the voxel and the distance of the point to the voxel center. Assuming that each point describes a small volume of the distance function, we compute the scale compatibility of point i and voxel j as the ratio of the volumes:

$$c_{ij} = \left(\frac{\min(\sigma_i, l_j)}{\max(\sigma_i, l_j)} \right)^3 . \tag{2}$$

To account for the spatial distance of the point to the voxel center we use a window function [28], which yields the importance a_{ij} as

$$a_{ij} = \begin{cases} c_{ij} \left(1 - \frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}{l_j^2}\right)^3 & \text{for } \|\mathbf{x}_i - \mathbf{x}_j\|_2 < l_j \\ 0 & \text{else.} \end{cases} \tag{3}$$

The normalizer ψ_j is therefore

$$\psi_j = \sum_{i \in \mathcal{N}(\mathbf{x}_j, R)} a_{ij}. \tag{4}$$

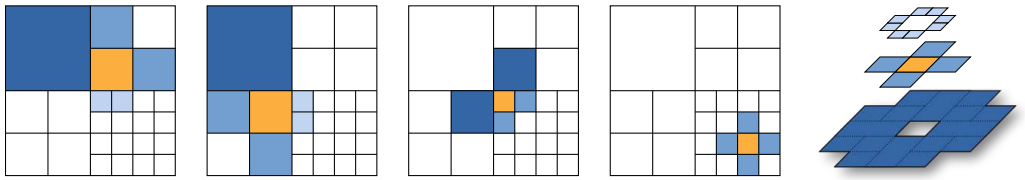
Normalization allows us to be invariant to changes in the absolute point density. We propagate the ψ_j obtained during the aggregation step in our network architecture to retain information about densities. For the convolution filter g we use a resolution of $4 \times 4 \times 4$. Λ is a ball-to-cube mapping, which maps the spherical filter to the cubic kernel.

5. Adaptive Grid Convolution

Our grid adapts to the resolution present in the input point cloud and represents data at multiple scales. As a consequence, regular 3D convolutions which expect the data to be represented at regularly spaced points cannot be used for processing. Sparse convolutions [11, 5] relax the requirements from dense regular grids to sparse regular grids, but again do not account for the irregular spacing found in adaptive grids. OctNet [24] accelerates convolutions on octrees, but the convolutions have regular cubic kernels. In contrast, our kernels adapt to the grid, just as the grid adapts to the input point cloud.

In the following we describe convolutions for face-balanced adaptive grids using multiscale convolutional kernels. Figure 5(a) shows 4 examples of the kernels used. For convolutions within the same grid, we use kernels that cover the center element as well as the face neighbors at the same and adjacent scales. Limiting the convolution to the face neighbors results in a small kernel with 55 elements, which is in between the storage requirements for regular $3 \times 3 \times 3$ and $4 \times 4 \times 4$ kernels. While for regular kernels in standard 3D convolutions all kernel elements are always active, only some of the elements of our multiscale kernels are active at each position. Depending on the spatial configuration of the grid, the number of active elements ranges from 7 to 25, making our kernel computationally efficient, small, and suitable for processing large grids.

Formally, we define the convolution at voxel j in the



(a) (b)

Figure 5: Convolution on an adaptive face-balanced grid. **(a)** shows 4 examples of the kernel shape used to compute the convolution at the orange center element. Our convolution kernel considers all face-adjacent voxels to the center. Adjacent voxels may have a different scale than the center and can have different alignments as can be seen in the first three examples. Since the grid is face-balanced there is only a small number of configurations for neighboring voxel scales and alignments. **(b)** shows all elements of our multiscale kernel for this quadtree example. The bottom shows the superposition of 8 neighboring elements for the next larger scale level. Note that the elements for the larger voxels overlap due to the different alignments to the center element. This can be observed by comparing the alignment of the dark blue voxels in the first two examples in **(a)** with the dark blue voxels in the third example. The kernel in this 2D example has in total 21 elements, while our 3D multiscale kernel has 55 elements in total. At each position in the grid only a subset of the elements is used as in the examples given in **(a)**.

adaptive grid similar to (1) as

$$(f * g)_j = \frac{1}{\eta_j} \sum_{i \in \mathcal{N}_j} a_i \; f_i \; g(I(i,j)). \tag{5}$$

The filter g is here a discrete kernel with 55 elements. The index function I computes the index of the kernel element with respect to the spatial configuration, which can be deduced from the location keys of the adjacent voxels i and j. Similarly, the set of face-adjacent neighbors $\mathcal{N}_j$ can be computed by looking up all neighbors of the location key associated with the output voxel j. The number of neighbors varies with respect to the spatial configuration and also border voxels have fewer neighbors. Lookups in the linear octree can be accelerated with a hash table, yielding $O(1)$ complexity for finding the neighbors of each voxel. In our implementation, we precompute all neighbors during octree generation, which allows us to reuse the neighbor lists for multiple convolutions on the same grid. The order of voxels being processed can be arbitrary but follows the ordering of the location keys in our implementation. For the normalizer η_j and the importance a_i of the neighbor voxel i we use two different configurations. For convolutions where we want to normalize with respect to the neighbors we set

$$\eta_j = \sum_{i \in \mathcal{N}_j} a_i \;\; \text{with} \;\; a_i = \psi_i, \tag{6}$$

where ψ_i are the importance values computed in the aggregation stage (4) or the importance values propagated on the adaptive grid. We use the same stencil as in the convolutions to propagate the importance values and give details in the supplement. For convolutions without normalization we simply set

$$\eta_j = 1 \;\; \text{and} \;\; a_i = 1. \tag{7}$$

In our architecture we mix both configurations, convolution with and convolution without normalization, to incorporate and retain information about the density from the aggregation stage. For the first convolution on each grid level in the encoder, we compute 8 feature channels with normalization in addition to the unnormalized channels.

For transitions between grids of different resolutions we define up- and down-convolutions with kernels that use 9 elements. 8 kernels elements are used for voxels that are either subdivided or merged and another kernel element is used for voxel that remain unchanged in both grids, which is also reflected in the index function I. We give more details on up- and down-convolutions in the supplement.

6. Distance Function Decode and Contouring

Our approach applies volumetric fusion, which computes an implicit representation of a surface in a volume. To extract the surface as a mesh, we use dual contouring and evaluate the signed and unsigned distance function at the voxel centers. The last convolution layer in our network generates features that encode local distance functions akin to [3]. We decode the signed distance value u, its spatial gradient ∇u, and the unsigned distance value v with an MLP. The input to the MLP are the features $\mathbf{f}$, the relative position $\mathbf{r}$, which gives the query position relative to the voxel center $\mathbf{c}$ and the voxel edge length l. Like $\mathbf{r}$, the output of the MLP is normalized with respect to the voxel size. To find the distance values $[u, v]$ for a voxel i at position $\mathbf{x}$ we compute

$$[\hat{u}_i(\mathbf{x}), \hat{v}_i(\mathbf{x})] = h(\mathbf{f}_i, \mathbf{r}, \theta) \tag{8}$$

with $\mathbf{r} = \frac{\mathbf{x} - \mathbf{c}_i}{l_i}$, h as the MLP, and θ as the learned parameters. The direct outputs of the MLP, $\hat{u}_i$ and $\hat{v}_i$, are normalized distances with respect to the voxel edge length l_i, and the unnormalized values can be obtained with $[u(\mathbf{x}), v(\mathbf{x})] = l_i[\hat{u}_i(\mathbf{x}), \hat{v}_i(\mathbf{x})]$. Note that in general $\mathbf{x}$ can be an arbitrary position but is most meaningful if $\mathbf{x}$ is close to $\mathbf{c}_i$. During training we randomly sample $\mathbf{x}$ such that

$\|\mathbf{x} - \mathbf{c}_i\|_\infty < l_i$, which also defines the range that we consider valid for $\mathbf{x}$ when querying the distances for voxel i. Since our MLP is differentiable, we can compute ∇u at $\mathbf{x}$ simply by extending our decoder with

$$\nabla u(\mathbf{x}) = \nabla h_{\hat{u}}(\mathbf{f}_i, \mathbf{r}, \theta), \tag{9}$$

where $h_{\hat{u}}$ is the subset of the MLP that computes $\hat{u}$. The use of extending our decoder with the operations for computing the gradient are twofold. First, the extension allows us to define a loss with respect to the surface normals, which correspond to the gradient of the signed distance function near the surface. Second, we can make use of the gradient in the vertex computation in the dual contouring [13, 27]. In contrast to the marching cubes algorithm, which generates vertices on the edges, dual contouring generates vertices for each dual cell that has edges crossing the surface. We compute the vertex position by minimizing a quadratic error function defined by the values of $\hat{u}$ and ∇u at the centers of the voxels that are connected by the dual cell. The unsigned distance value $\hat{v}$ is used as an additional threshold in the contouring step to restrict the generation of triangles near to the surface. We only generate triangles if we detect a sign change in $\hat{u}$ and $\hat{v} < 1.5$.

7. Training

7.1. Data Generation

Our goal is to train a general surface reconstruction network. The training data and sample generation reflect this choice. We train our method on Creative Commons data from the Thingi10K dataset [31] and the Scan the World project. Both are collections of 3D models for 3D printing with a large variety of models from technical parts to sculptures. For sample generation, we randomly place 3D models and virtual scanners in a scene. To generate the noisy point clouds we follow two approaches.

Our first approach generates depth maps for each scanner using ray tracing and applies a simple noise simulation by altering ray directions and depth values. We then generate the input point cloud from the depth values. This approach can generate point clouds online during training and allows us to smoothly vary the noise level.

In our second approach, we use Blender to render images of the generated scenes and use the patch match stereo implementation from COLMAP [26] to compute the depth maps to generate more realistic point clouds.

For both approaches we compute the ground truth by building the octree as described in Section 4 and sample for each leaf voxel a random point $\mathbf{x}$ close to the voxel center $\mathbf{c}$, i.e., for a voxel i we sample $\mathbf{x}_i$ uniformly such that $\|\mathbf{x} - \mathbf{c}_i\|_\infty < l_i$. Note that $\mathbf{x}$ can lie outside of the voxel similar as in [3]. For each of these points we then compute the closest point on the ground-truth mesh and determine the distance and the sign to obtain the ground truth $u_{\text{gt}}(\mathbf{x}_i)$ and $\nabla u_{\text{gt}}(\mathbf{x}_i)$. We use the Embree library [29] to accelerate the ray casting and closest point computations. During training we sample data equally using the two approaches and randomly flip the sign of the ground-truth distance and the normals for augmentation. We show examples from the training data in the supplementary material and will release code for generating the training data.

Dataset	#Points	#Images	Type	GT
Barn	24 M	410	Object	Laser scan
Caterpillar	24 M	383	Object	Laser scan
Church	51 M	507	Indoor	Laser scan
Courthouse	95 M	1106	Outdoor	Laser scan
Ignatius	7 M	263	Object	Laser scan
Meetingroom	33 M	371	Indoor	Laser scan
Truck	17 M	251	Object	Laser scan
Citywall	359 M	564	Outdoor	n/a

Table 1: Datasets for evaluation. We test our method on all datasets from Tanks and Temples [16] with publicly available ground truth. In addition, we use the Citywall dataset from [8]. The datasets contain indoor and outdoor settings as well as scenes focused on a single object.

7.2. Loss Functions

We define the following loss functions on the network outputs $\hat{u}$, $\hat{v}$, and ∇u.

$$\mathcal{L}_{\text{SDF}} = \frac{1}{N} \sum_i^N \delta_i(\mathbf{x}_i) \left(\hat{u}(\mathbf{x}_i) - \max(-2, \min(2, \hat{u}_{\text{gt}}(\mathbf{x}_i)))\right)^2, \tag{10}$$

which is the squared difference of the predicted and ground-truth signed distance normalized by the respective edge length of the voxels and truncated at ± 2. The normalized ground-truth distance is defined as $\hat{u}_{\text{gt}}(\mathbf{x}_i) = l_i u_{\text{gt}}(\mathbf{x}_i)$ and the mask function $\delta_i(\mathbf{x}_i)$ is defined as

$$\delta_i(\mathbf{x}_i) = \begin{cases} 1, & \text{if } |\hat{u}_{\text{gt}}(\mathbf{x}_i)| < 2 \\ 0, & \text{else} \end{cases} \tag{11}$$

to enable the loss only near the surface.

Similarly we define the loss for the unsigned distance

$$\mathcal{L}_{\text{UDF}} = \frac{1}{N} \sum_i^N \left(\hat{v}(\mathbf{x}_i) - \min(2, |\hat{u}_{\text{gt}}(\mathbf{x}_i)|)\right)^2 \tag{12}$$

but compute the loss for the whole domain.

We define a loss on the normals as

$$\mathcal{L}_{\text{Normal}} = \frac{1}{N} \sum_i^N \rho \left\|\nabla u(\mathbf{x}_i) - \nabla u_{\text{gt}}(\mathbf{x}_i)\right\|_2^2 \tag{13}$$

with the ramp $\rho = \max(0, 1 - \frac{|u_{\text{gt}}(\mathbf{x}_i)|}{2 l_i})$ to linearly fade out the loss $\mathcal{L}_{\text{Normal}}$ with increasing distance to the surface as

Method	Barn	Caterpillar	Church	Courthouse	Ignatius	Meetingroom	Truck	Mean
PSR [14]	47.66	29.03	40.36	16.47	**76.62**	26.26	44.26	40.09
SSD [2]	45.74	19.51	34.94	5.49	74.71	19.04	36.22	33.66
GDMR [27]	46.78	27.74	37.64	14.97	73.97	28.34	46.93	39.48
LIG [12]	27.04	21.17	26.34	12.44	50.34	18.63	23.53	25.64
Points2Surf [7]	16.69	14.26	12.22	7.74	50.87	12.71	15.81	18.61
Points2Surf [7] (fine-tuned on our data)	18.33	18.26	26.01	7.12	43.75	14.76	33.71	23.13
Ours (w/o $\mathcal{L}_{\text{Normal}}$)	49.00	32.94	41.75	21.14	75.47	30.99	56.43	43.96
Ours (w/o rendered data)	**50.59**	35.07	43.28	**21.45**	74.25	30.26	59.18	44.87
Ours	49.83	**35.94**	**43.50**	18.41	75.58	**32.95**	**59.86**	**45.15**

Table 2: F-score on the Tanks and Temples dataset. All methods use the same input point clouds. Parameters were tuned for each method and scene. For PSR and SSD we tune the density threshold and the maximum octree depth. For GDMR we additionally tune the strength of the data term. For LIG we try different scales for the part size and enable backface removal. To compute reconstructions with Points2Surf we use the *max* model and the authors' preprocessing script which normalizes and downsamples the point cloud. We vary the grid size and the target point cloud size and report the best result. Additionally, we also report results for Points2Surf fine-tuned with our training data for 50k iterations. Our method scores higher on all scenes except for Ignatius, on which PSR has a slight edge. Ignatius is the easiest dataset with respect to noise and surface complexity, and all methods except for LIG and Points2Surf produce a good reconstruction. For ablations (bottom), we trained our method without normal loss and without point clouds generated with COLMAP from rendered data. In both cases we see a decrease in performance.

we are more interested in accurate orientations closer to the surface. In all equations, N is the number of voxels at the finest grid level. Our optimization objective is

$$\mathcal{L} = \mathcal{L}_{\text{SDF}} + \mathcal{L}_{\text{UDF}} + \lambda \mathcal{L}_{\text{Normal}}, \tag{14}$$

with empirically determined $\lambda = 0.1$. To learn the network parameters we use the Adam optimizer with a constant learning rate of 1×10^{-3} and train the network for 25,000 iterations. Training takes about 10 hours on an RTX 2080 Ti GPU. We choose a batch size of 1 to fit large training scenes in memory.

8. Results

We focus our evaluation on zero-shot generalization to real-world data. To this end, we use datasets from Tanks and Temples [16] with publicly available ground truth and use the standard evaluation toolbox implemented with Open3D [32]. Moreover, we use the Citywall dataset from [8], which has multiple points of interest and varying detail throughout the scene.For generating the input data we use COLMAP [25, 26]. This produces noisy and incomplete point clouds, which we use as input for our approach and the baselines. Table 1 gives an overview of the type and size of all datasets.

We compare our method to a number of analytical volumetric fusion approaches: Poisson Surface Reconstruction (PSR) [14], Smoothed Signed Distance Surface Reconstruction (SSD) [2], and Global Dense Multiscale Reconstruction (GDMR) [27]. We also compare to recent learning-based approaches: Local Implicit Grid Representations (LIG) [12] and Points2Surf [7]. LIG is specialized for indoor scenes but Jiang *et al*. [12] demonstrated good generalization to unseen data. Points2Surf is a general surface reconstructor that, in contrast to our and the other methods we compare to, does not require normal information.

We use the F-score with the default thresholds [16] to quantify reconstruction performance and show our results in Table 2. We give qualitative examples of the results in Figure 6 and provide more results for all datasets in the supplement.

In addition, we evaluate design decisions such as the loss on the normals $\mathcal{L}_{\text{Normal}}$ and the use of more realistic point clouds generated from rendered images. We report these ablations in Table 2 as well. The importance of the normalization is illustrated qualitatively in the supplement.

For the large Citywall dataset with 359M points we show our reconstruction in Figure 1. Besides our method, we were only able to reconstruct this dataset with the full point cloud with the GDMR baseline. Our method reconstructs the scene more than two times faster (117 min) compared to GDMR (274 min) on an Intel Xeon E7-8890v3 and produces a more faithful surface. We show qualitative comparisons and more runtime measurements in the supplement.

9. Conclusion

We have presented an efficient ConvNet architecture for 3D surface reconstruction. Multiscale convolutional kernels generalize convolutions to adaptive grids, allowing us to process large data sets. We have demonstrated this by reconstructing large real-world scenes with millions of points. We have demonstrated zero-shot generalization to real data. Our method yields better reconstruction accuracy and runtime than well-established analytical approaches and recent learning-based techniques.

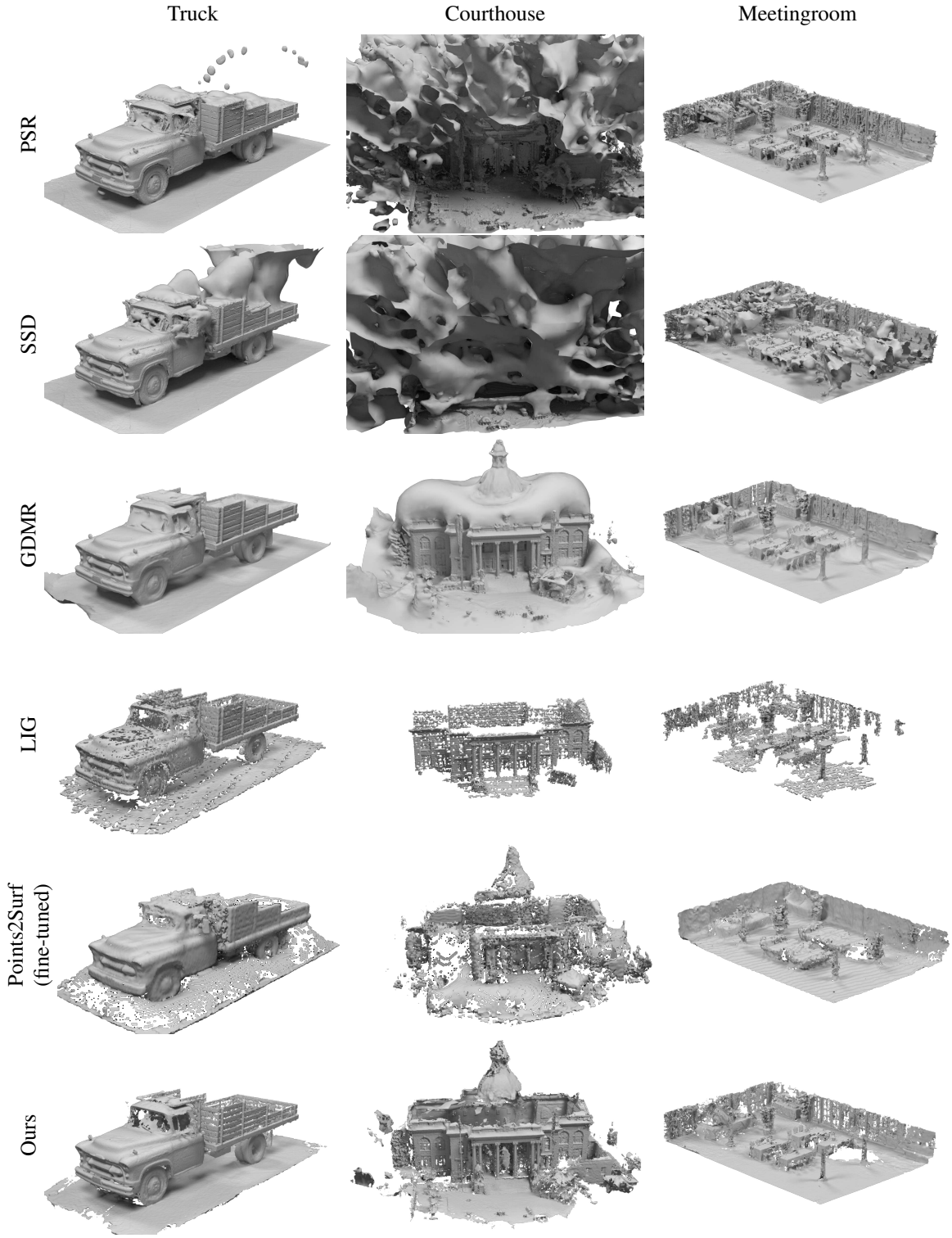


Figure 6: Qualitative comparison of reconstructed surfaces on 3 datasets from Tanks and Temples. The traditional methods like PSR, SSD and GDMR show spurious geometry and ballooning artifacts. The learning approaches LIG and Points2Surf have problems with most scenes and often miss to reconstruct surfaces. The generated reconstructions also use a lower resolution to avoid problems due to the input size of the data. On Meetingroom our reconstruction is less noisy than PSR or SSD and does not oversmooth surfaces as much as GDMR.

References

[1] Matan Atzmon and Yaron Lipman. SAL: Sign agnostic learning of shapes from raw data. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. 2

[2] F. Calakli and G. Taubin. SSD: Smooth Signed Distance Surface Reconstruction. *Computer Graphics Forum*, 30(7), 2011. 2, 7

[3] Rohan Chabra, Jan E. Lenssen, Eddy Ilg, Tanner Schmidt, Julian Straub, Steven Lovegrove, and Richard Newcombe. Deep local shapes: Learning local SDF priors for detailed 3D reconstruction. In *European Conference on Computer Vision (ECCV)*, 2020. 2, 5, 6

[4] Julian Chibane, Aymen Mir, and Gerard Pons-Moll. Neural unsigned distance fields for implicit function learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. 2

[5] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 4D spatio-temporal ConvNets: Minkowski convolutional neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 3, 4

[6] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of SIGGRAPH*, 1996. 2

[7] Philipp Erler, Paul Guerrero, Stefan Ohrhallinger, Niloy J. Mitra, and Michael Wimmer. Points2Surf: Learning Implicit Surfaces from Point Clouds. In *European Conference on Computer Vision (ECCV)*, 2020. 2, 7

[8] Simon Fuhrmann and Michael Goesele. Fusion of depth maps with multiple scales. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, 2011. 2, 6, 7

[9] Simon Fuhrmann and Michael Goesele. Floating Scale Surface Reconstruction. *ACM Trans. Graph.*, 33(4), 2014. 2

[10] Irene Gargantini. Linear octtrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing*, 20(4), 1982. 3

[11] Benjamin Graham, Martin Engelcke, and Laurens van der Maaten. 3D semantic segmentation with submanifold sparse convolutional networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. 3, 4

[12] Chiyu "Max" Jiang, Avneesh Sud, Ameesh Makadia, Jingwei Huang, Matthias Niessner, and Thomas Funkhouser. Local implicit grid representations for 3D scenes. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. 2, 7

[13] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual Contouring of Hermite Data. *ACM Trans. Graph.*, 23(3), 2002. 6

[14] Michael Kazhdan and Hugues Hoppe. Screened Poisson Surface Reconstruction. *ACM Trans. Graph.*, 32(3), 2013. 2, 7

[15] Roman Klokov and Victor Lempitsky. Escape from cells: Deep kd-networks for the recognition of 3d point cloud models. In *IEEE International Conference on Computer Vision ICCV*, 2017. 2

[16] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction. *ACM Trans. Graph.*, 36(4), 2017. 2, 6, 7

[17] K Kolev, T Pock, and D Cremers. Anisotropic Minimal Surfaces Integrating Photoconsistency and Normal Information for Multiview Stereo. In *European Conference on Computer Vision (ECCV)*, 2010. 2

[18] Victor Lempitsky and Yuri Boykov. Global optimization for shape fitting. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2007. 2

[19] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proceedings of SIGGRAPH*, 1987. 2

[20] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3D reconstruction in function space. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 2

[21] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. DeepSDF: Learning continuous signed distance functions for shape representation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 2

[22] Songyou Peng, Michael Niemeyer, Lars Mescheder, Marc Pollefeys, and Andreas Geiger. Convolutional occupancy networks. In *European Conference on Computer Vision (ECCV)*, 2020. 2

[23] Gernot Riegler, Ali Osman Ulusoy, Horst Bischof, and Andreas Geiger. OctNetFusion: Learning depth fusion from data. In *Proceedings of the International Conference on 3D Vision*, 2017. 2

[24] G. Riegler, A. O. Ulusoy, and A. Geiger. OctNet: Learning deep 3D representations at high resolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. 2, 4

[25] J. L. Schönberger and J. M. Frahm. Structure-from-Motion Revisited. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. 7

[26] Johannes L. Schönberger, Enliang Zheng, Jan-Michael Frahm, and Marc Pollefeys. Pixelwise View Selection for Unstructured Multi-View Stereo. In *European Conference on Computer Vision (ECCV)*, 2016. 6, 7

[27] Benjamin Ummenhofer and Thomas Brox. Global, Dense Multiscale Reconstruction for a Billion Points. *International Journal of Computer Vision*, 2017. 2, 3, 6, 7

[28] Benjamin Ummenhofer, Lukas Prantl, Nils Thuerey, and Vladlen Koltun. Lagrangian Fluid Simulation with Continuous Convolutions. In *International Conference on Learning Representations*, 2020. 3, 4

[29] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. Embree: A kernel framework for efficient CPU ray tracing. *ACM Trans. Graph.*, 33(4), 2014. 6

[30] C. Zach, T. Pock, and H. Bischof. A Globally Optimal Algorithm for Robust TV-L1 Range Image Integration. In *IEEE International Conference on Computer Vision ICCV*, 2007. 2

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

[32] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018. 7