Supplementary Material
for
Adaptive Surface Reconstruction with Multiscale Convolutional Kernels

Contents

1. Network Architecture Parameters

Our network uses a U-Net-like architecture as depicted in Figure 1. We provide the kernel sizes and the number of output channels of each layer Table 1. In addition to the total number of output channels, we give the number of channels that we normalize. The first layer uses a continuous convolution to aggregate features from the point cloud and normalizes all channels. We keep the importance values that we compute in the aggregation step and propagate them in the encoder as shown in Figure 4 and use them for computing the normalization in subsequent layers. Note that we use addition instead of concatenation for the last skip connection to reduce computational complexity. The distance function decode part of our network uses an MLP with three dense layers and is applied voxel-wise. During training and inference we reuse the weights of these three layers to compute the gradient for the signed distance.

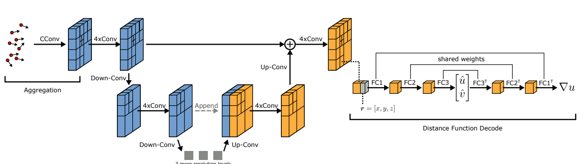


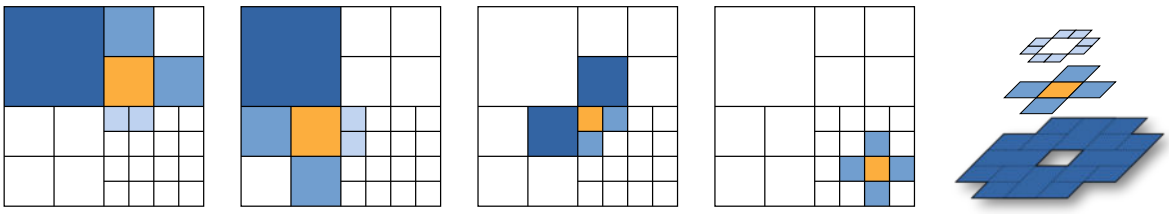
Figure 1: Schematic overview of our network architecture.

Operation	Kernel size	Output channels (normalized channels)
CConv + ReLU	4x4x4	32 (32)
Conv + ReLU	55	64
Conv + ReLU	55	64
Conv + ReLU	55	64
Conv + ReLU	55	64
Down-Conv + ReLU	9	128 (8)
Conv + ReLU	55	128 (8)
Conv + ReLU	55	128
Conv + ReLU	55	128
Conv + ReLU	55	128
Down-Conv + ReLU	9	256 (8)
Conv + ReLU	55	256 (8)
Conv + ReLU	55	256
Conv + ReLU	55	256
Conv + ReLU	55	256
Down-Conv + ReLU	9	256 (8)
Conv + ReLU	55	256 (8)
Conv + ReLU	55	256
Conv + ReLU	55	256
Conv + ReLU	55	256
Down-Conv + ReLU	9	256 (8)
Conv + ReLU	55	256 (8)
Conv + ReLU	55	256
Conv + ReLU	55	256
Conv + ReLU	55	256
Up-Conv + ReLU	9	256
Concat		
Conv + ReLU	55	256
Conv + ReLU	55	256
Conv + ReLU	55	256
Conv + ReLU	55	256
Up-Conv + ReLU	9	256
Concat		
Conv + ReLU	55	256
Conv + ReLU	55	256
Conv + ReLU	55	256
Conv + ReLU	55	256
Up-Conv + ReLU	9	256
Concat		
Conv + ReLU	55	128
Conv + ReLU	55	128
Conv + ReLU	55	128
Conv + ReLU	55	128
Up-Conv + ReLU	9	64
Add		
Conv + ReLU	55	32
Conv + ReLU	55	32
Conv + ReLU	55	32
Conv + ReLU	55	32
Distance Function Decode		
FC + ReLU		32
FC + ReLU		32
FC		2

Table 1: Kernel sizes and number of channels for each layer.

2. Multiscale Kernels

Figure 2(a) shows 2D examples of how the multiscale kernel shown in (b) is applied to the adaptive grid. To design networks with a hierarchy of adaptive grids we need kernels that connect two adaptive grids similar as in strided convolutions.



(a) (b)

Figure 2: 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)**.

Figure 3 shows how we define the kernels that connect two adaptive grids in our down-convolution. Kernels for up-convolutions that increase the resolution are defined analogously.

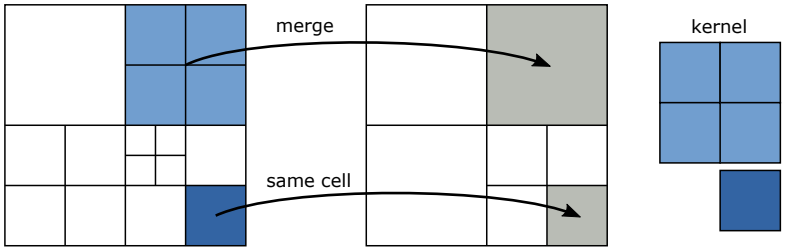


Figure 3: Example for a down-convolution transfering information to a coarser adaptive grid. In the 2D case kernels have 5 elements as shown in the right part of the image and kernels for 3D grids have 9 elements. Note that not all voxels in the grid are merged and some voxels are present in both grids. For voxels that are not merged the single kernel element used is like a 1×1 kernel in a standard convolution, while the 4 kernel elements used on the voxels that are merged act similar to a strided convolution with a 2×2 kernel.

3. Normalization

Normalization allows us to be invariant to changes in the absolute point density. At the same time, we want to retain information about relative densities. To this end, we propagate density information from the aggregation step to each resolution level in the encoder of our network and use them as the importance values a_i in the convolution. Figure 4 shows the propagation procedure. Without normalization, the network overfits to the densities observed during training. We demonstrate this in Figure 5.

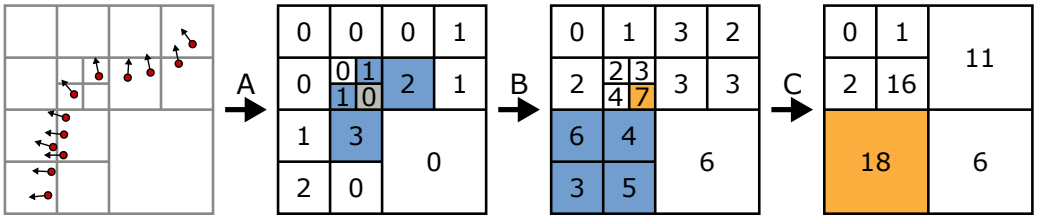


Figure 4: Computation of the importance values a_i for the convolutions on the adaptive grid. **A** shows the aggregation step. For simplicity we use the number of points that lie inside a voxel as values in this example. **B** shows the propagation of the values within the adaptive grid for the first convolution on each level of the grid hierarchy. We use the same stencil as in the convolutions to propagate the importance values. The highlighted voxels in blue and gray with the grey voxel being the center of the stencil show the inputs while the same voxel highlighted in orange to the right of **B** shows the output. **C** shows the computation for a transition to the next coarser level of the grid hierarchy. Note that there is no center voxel and the output is at the highlighted voxel in the next coarser grid.

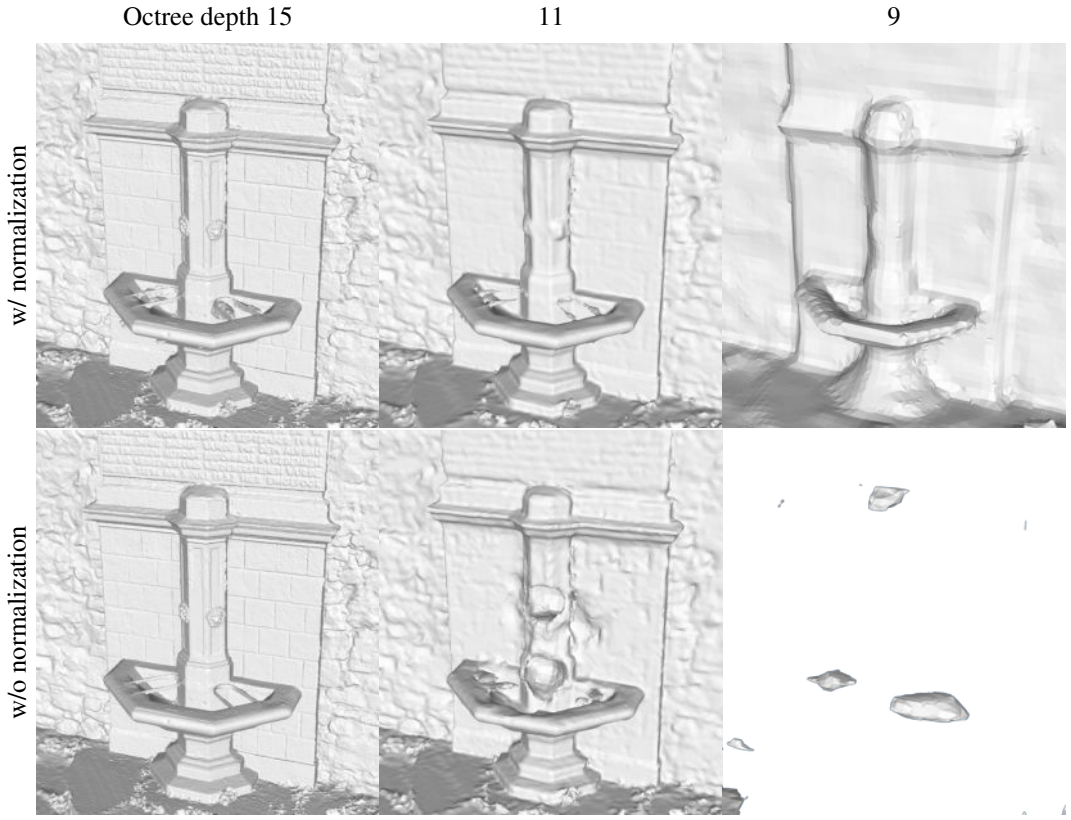


Figure 5: Effect of the importance normalization for different octree depths. We use the same input point cloud but limit the maximum octree depth, which produces strong differences in the point densities. Normalization allows to generalize to point densities not observed during training and enables reconstructions at coarse resolutions. Without normalization, the surface deteriorates quickly when resolution is limited.

4. Training Data Examples

We train our method on Creative Commons and public domain data from the Thingi10K dataset [8] and the Scan the World project[1].

We create scenes by randomly placing up to 3 objects near the scene origin. Each time we add an object we check for intersections to ensure that inside and outside regions are well defined for computing the ground truth signed distance and resample if the intersection test fails. We place 30 to 60 cameras for each scene. Cameras are placed such that the position is outside the circumscribed sphere of the objects and the look-at target is a random vertex of one of the meshes of the objects.

To generate the noisy input point clouds for training our method we use two approaches. Our first approach generates depth maps directly from the meshes using ray casting. To simulate noise we perturb the ray origins and ray directions. In addition, we add noise sampled from a Laplace distribution directly to the computed depth values. This process is fast and allows us to generate noisy input data online during the training.

Our second approach adds a more realistic noise simulation. We use Blender to render images for each virtual camera. We apply random textures to each object and use random environment maps for lighting. To generate the point clouds, we feed the rendered images to the patch match stereo algorithm [6]. We show examples of the rendered images and the generated point clouds in Figure 6.

[1] https://www.myminifactory.com/scantheworld/

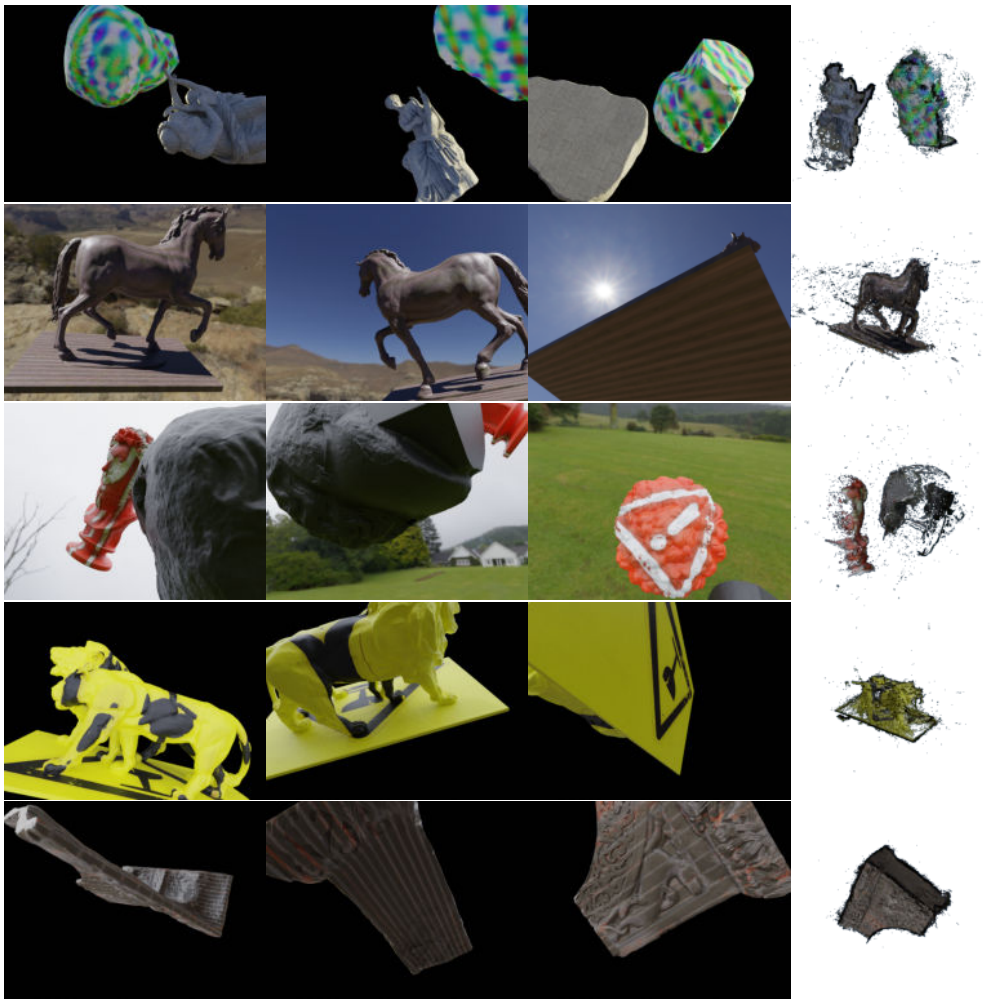


Figure 6: Rendered images and generated point clouds. Each row shows a randomly generated scene. We generate point clouds from the depth maps generated by the patch match stereo implementation of COLMAP. Note that we randomly hide the background producing training examples with fewer outliers.

5. Runtime Comparison

We compare runtimes on the large Citywall dataset for different input sizes. The input point cloud has been generated with COLMAP and contains 359 million points. We subsample the original point cloud to generate smaller problem sizes.

We run all methods on an Intel Xeon E7-8890v3 with 256 GB main memory and 32 threads and report the results in Table 2. The learning-based methods LIG and Points2Surf use the TensorFlow and PyTorch frameworks, respectively, to accelerate computation. Our method uses TensorFlow to build the execution graph but uses custom ops to implement our multiscale adaptive convolutions. PSR and SSD crash if the octree exceeds a depth of 14. We, therefore, report only results for up to 72M points. LIG runs out of memory on the largest input size. For Points2Surf we could not get a result with the tested input sizes. Only GDMR and our method are able to reconstruct the scene with 359M points. Our method is more than 2 times faster than GMDR. GDMR is the most economical method with respect to peak memory consumption, which can be explained by the low temporary memory requirements of iterative algorithms for solving PDEs. Additionally, GDMR uses an octree of depth 16 to reconstruct the scene while our method generates an octree with depth 17, yielding a more faithful reconstruction as shown in the supplementary video.

We compare the runtime of a single convolution in Table 3 for OctNet [5], our method, and a dense 3D convolution. We measure all runtimes on an Intel Core i9-7960X CPU with an Nvidia RTX 2080Ti. Table 3 shows the runtime comparison for the CPU and GPU implementation. For the dense 3D convolution we use the precompiled binaries for TensorFlow 2.5 with

Method	Runtime / memory peak for different input sizes			
	7M	72M	144M	359M
PSR [4]	7 min / 9.9 GB	25 min / 31.7 GB	n/a	n/a
SSD [1]	9 min / 10.2 GB	29 min / 38.4 GB	n/a	n/a
GDMR [7]	4 min / 0.9 GB	51 min / 7.8 GB	101 min / 13.4 GB	274 min / 17.0 GB
LIG [3]	90 min / 13.0 GB	239 min / 71.1 GB	255 min / 141.8 GB	n/a
Points2Surf [2]	n/a	n/a	n/a	n/a
Ours	2 min / 2.9 GB	12 min / 19.9 GB	22 min / 32.7 GB	117 min / 237.3 GB

Table 2: Comparison of runtime and peak memory footprint on the Citywall scene with different input point cloud sizes.

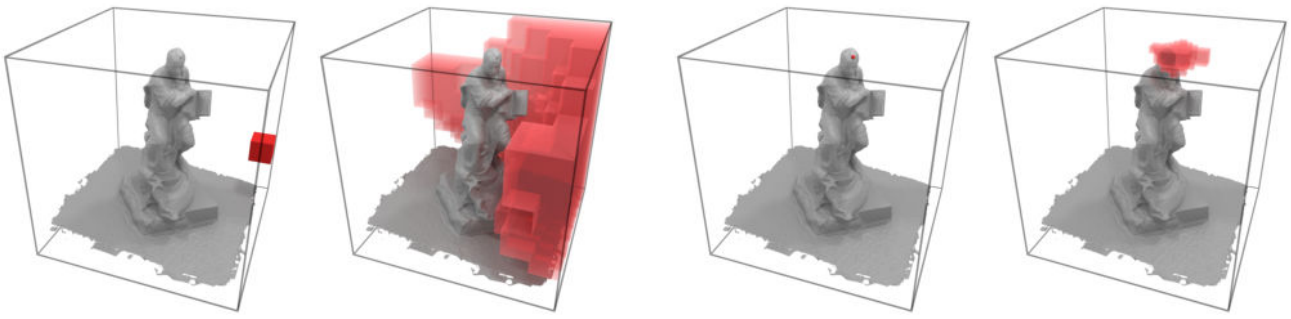
CUDA 11. For OctNet we compile from source with CUDA 10.1. OctNet does not provide a performant implementation for the CPU. Our implementation is compiled as a custom operator for TensorFlow 2.4. We use CUDA 10.2 for the GPU implementation and the Eigen 3.4 matrix library for the CPU implementation.

Method	Leaf nodes	CPU [ms]	GPU [ms]
Dense 3D Conv	16777216	840.5 ± 2.5	173.3 ± 1.1
OctNet [5]	256656	n/a	34.1 ± 3.4
Ours	270726	**48.1** ± 2.8	**17.5** ± 0.6

Table 3: Runtime comparison of a single convolution with 32 filters on a 256^3 grid with 32 channels. We use the point cloud from the Ignatius dataset to create the octree data structures for Octnet and our method. Note that because of the balancing criterion there are more leaf nodes for our method than for Octnet.

6. Receptive Field

The size and shape of the receptive field strongly depends on the adaptive grid and can be different for each voxel. We show this in Figure 7 comparing the receptive field of a voxel near the surface and a voxel near the boundary. In Figure 8 we give a 2D example showing the growth of the receptive field.



(a) (b)

Figure 7: Receptive field within the adaptive grid excluding the aggregation step. The two images in (a) and (b) show the source voxel and the receptive field respectively. The size and shape of the receptive field strongly depends on the grid. (a) shows the receptive field for a voxel far away from the surface covering large parts of the empty space. (b) shows the receptive field for a voxel near the surface which is significantly smaller in size but provides more resolution for details.

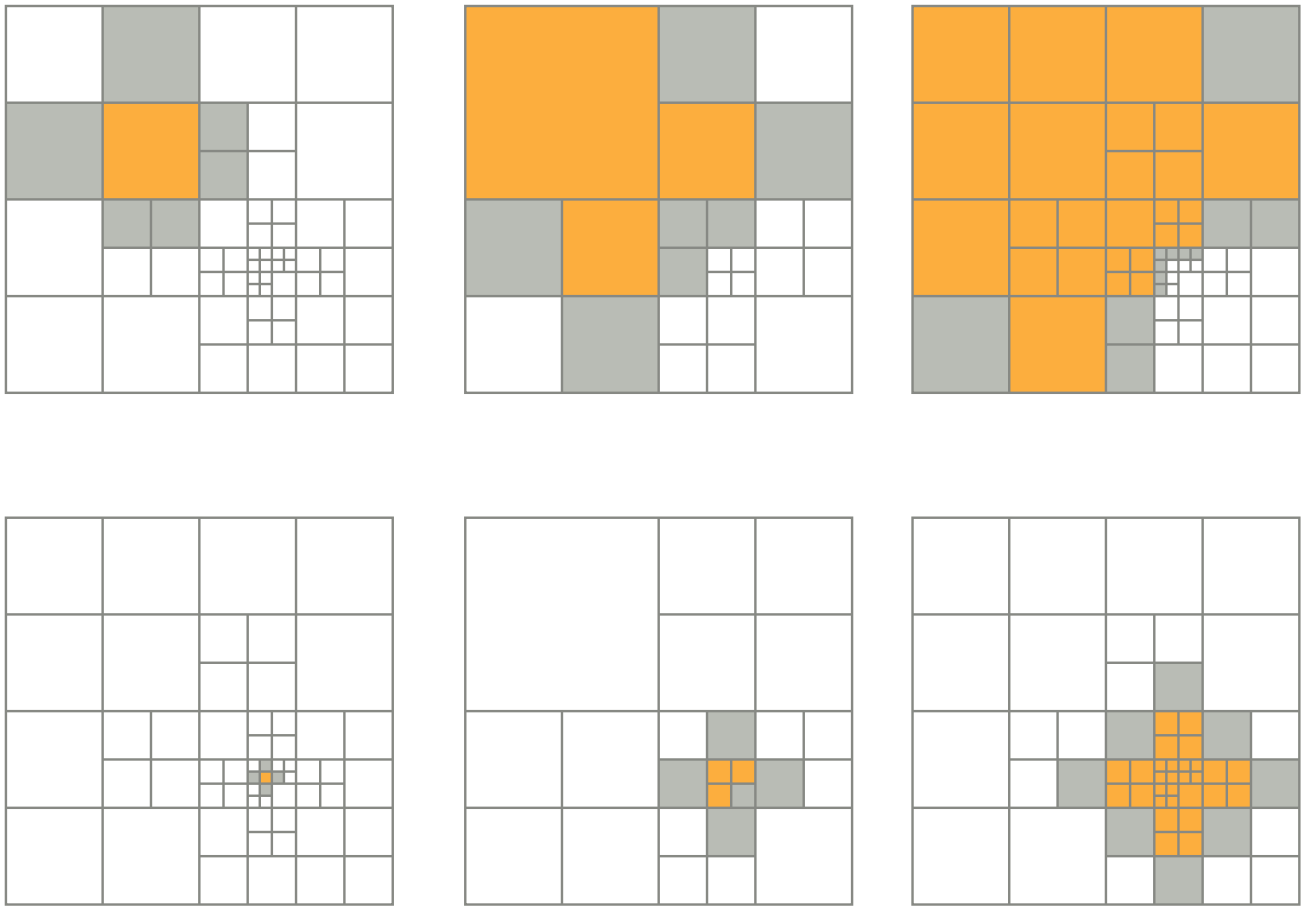


Figure 8: 2D quadtree examples for the receptive field. Each row shows the receptive field for a different start voxel. The image sequence shows the growth of the receptive field for the operations [Conv*, Down-Conv, Conv*, Up-Conv, Conv*]. The images show the receptive fields for the Conv* operation before (yellow) and after (yellow+gray) the convolution. Note that the receptive field in the first row is larger but also contains more voxels in number than the example in the second row.

7. Evaluation Metrics

We evaluate the accuracy of our method on the Tanks and Temples datasets. To this end we use the evaluation toolbox from the dataset, which measures the reconstruction quality using the F-score. We briefly give the definition here for convenience. The precision is defined as

$$P(\tau) = \frac{100}{|\mathcal{R}|} \sum_{\mathbf{r} \in \mathcal{R}} \left[\min_{\mathbf{g} \in \mathcal{G}} \|\mathbf{r} - \mathbf{g}\|_2 < \tau \right], \tag{1}$$

with $\mathcal{R}$ as the reconstructed point cloud, $\mathcal{G}$ as the ground truth point cloud, τ as a dataset specific threshold, and $[\cdot]$ as the Iversion bracket. The recall is defined accordingly as

$$R(\tau) = \frac{100}{|\mathcal{G}|} \sum_{\mathbf{g} \in \mathcal{G}} \left[\min_{\mathbf{r} \in \mathcal{R}} \|\mathbf{g} - \mathbf{r}\|_2 < \tau \right]. \tag{2}$$

The final F-score is then computed as

$$F(\tau) = \frac{2P(\tau)R(\tau)}{P(\tau) + R(\tau)}. \tag{3}$$

To create the reconstructed point cloud $\mathcal{R}$ we sample points uniformly from the meshes created by each method. Additionally, we compute the symmetric Chamfer distance for completeness, which is defined as

$$CD = \frac{1}{|\mathcal{R}|} \sum_{\mathbf{r} \in \mathcal{R}} \min_{\mathbf{g} \in \mathcal{G}} \|\mathbf{g} - \mathbf{r}\|_2^2 + \frac{1}{|\mathcal{G}|} \sum_{\mathbf{g} \in \mathcal{G}} \min_{\mathbf{r} \in \mathcal{R}} \|\mathbf{g} - \mathbf{r}\|_2^2 \tag{4}$$

We report both metrics in Table 4. Note that the Chamfer distance is sensitive to outliers and values across datasets can significantly vary.

Method	Barn		Caterpillar		Church		Courthouse		Ignatius		Meetingroom		Truck		Mean	
	F ↑	CD ↓	F ↑	CD ↓	F ↑	CD ↓	F↑	CD ↓	F ↑	CD ↓	F ↑	CD ↓	F ↑	CD ↓	F ↑	CD ↓
PSR [4]	47.66	(0.009)	29.03	(0.057)	40.36	(**0.151**)	16.47	(77.13)	**76.62**	(6.5e−5)	26.26	(0.046)	44.26	(0.049)	40.09	(11.06)
SSD [1]	45.74	(0.135)	19.51	(0.253)	34.94	(0.466)	5.49	(81.32)	74.71	(2.5e−4)	19.04	(0.214)	36.22	(0.383)	33.66	(11.82)
GDMR [7]	46.78	(0.010)	27.74	(0.045)	37.64	(0.229)	14.97	(10.44)	73.97	(7.0e−5)	28.34	(0.037)	46.93	(0.011)	39.48	(1.54)
LIG [3]	27.04	(0.119)	21.17	(**0.037**)	26.34	(3.752)	12.44	(30.61)	50.34	(6.9e−4)	18.63	(0.371)	23.53	(0.011)	25.64	(4.99)
Points2Surf [2]	16.69	(0.019)	14.26	(0.051)	12.22	(0.455)	7.74	(3.70)	50.87	(9.6e−4)	12.71	(0.062)	15.81	(0.008)	18.61	(0.61)
Points2Surf [2] (fine-tuned on our data)	18.33	(0.326)	18.26	(0.041)	26.01	(0.289)	7.12	(3.78)	43.75	(0.685)	14.76	(0.062)	33.71	(0.016)	23.13	(0.74)
Ours (w/o $\mathcal{L}_{\text{Normal}}$)	49.00	(0.008)	32.94	(0.046)	41.75	(0.194)	21.14	(2.40)	75.47	(6.8e−5)	30.99	(0.036)	56.43	(0.006)	43.96	(0.38)
Ours (w/o rendered data)	**50.59**	(**0.005**)	35.07	(0.043)	43.28	(0.218)	**21.45**	(**2.10**)	74.25	(7.3e−5)	30.26	(0.039)	59.18	(**0.003**)	44.87	(**0.34**)
Ours	49.83	(0.008)	**35.94**	(0.044)	**43.50**	(0.205)	18.41	(2.37)	75.58	(**6.2e−5**)	**32.95**	(**0.035**)	**59.86**	(0.004)	**45.15**	(0.38)

Table 4: F-score and Chamfer distance 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.

8. Qualitative Comparisons

We compare our reconstruction of the Citywall dataset in Figure 9 and in the supplementary video. We show reconstructions of the datasets from Tanks and Temples in Figure 10, 11, 12, 13, 14, 15, and 16.

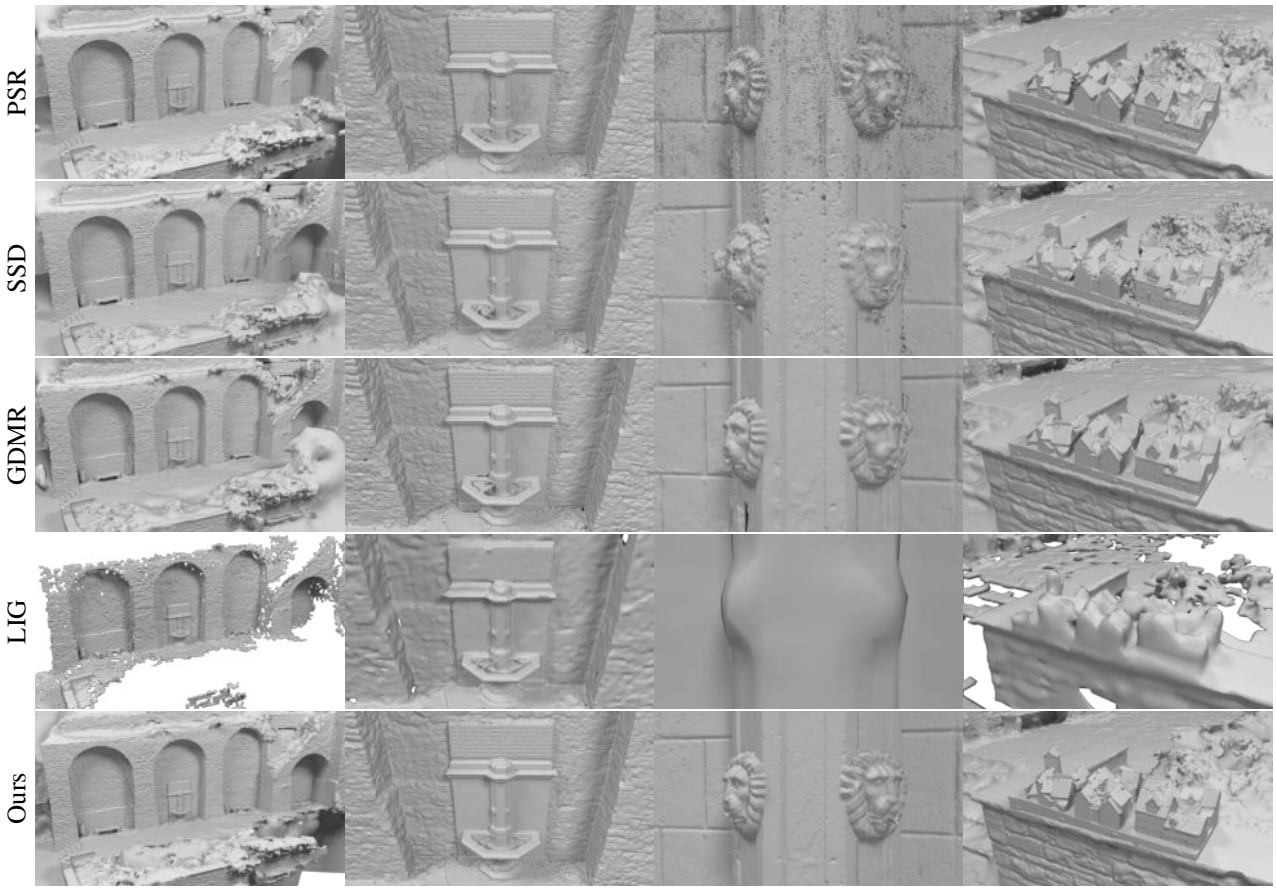


Figure 9: Qualitative comparison of reconstructed surfaces on the Citywall scene. We show for each method the reconstruction with the highest detail. PSR and SSD reconstruct large parts of the scene well but show high frequency artifacts at the points of the scene that require a higher level of detail. GDMR reconstructs the scene well but produces some holes near the fountain and the lion heads. LIG was able to reconstruct the scene with an input size of 149M points but the surface of the reconstructed surface is too low to capture all details. We were not able to reconstruct the scene with a smaller value for the part size parameter on our system. Our method shows overall the best reconstruction with the least amount of artifacts. Points of interest like the lion heads are more detailed than for the baselines.

References

[1] F. Calakli and G. Taubin. SSD: Smooth Signed Distance Surface Reconstruction. *Computer Graphics Forum*, 30(7), 2011. 5, 8

[2] 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. 5, 8

[3] 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. 5, 8

[4] Michael Kazhdan and Hugues Hoppe. Screened Poisson Surface Reconstruction. *ACM Trans. Graph.*, 32(3), 2013. 5, 8

[5] 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. 4, 5

[6] 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. 3

[7] Benjamin Ummenhofer and Thomas Brox. Global, Dense Multiscale Reconstruction for a Billion Points. *International Journal of Computer Vision*, 2017. 5, 8

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

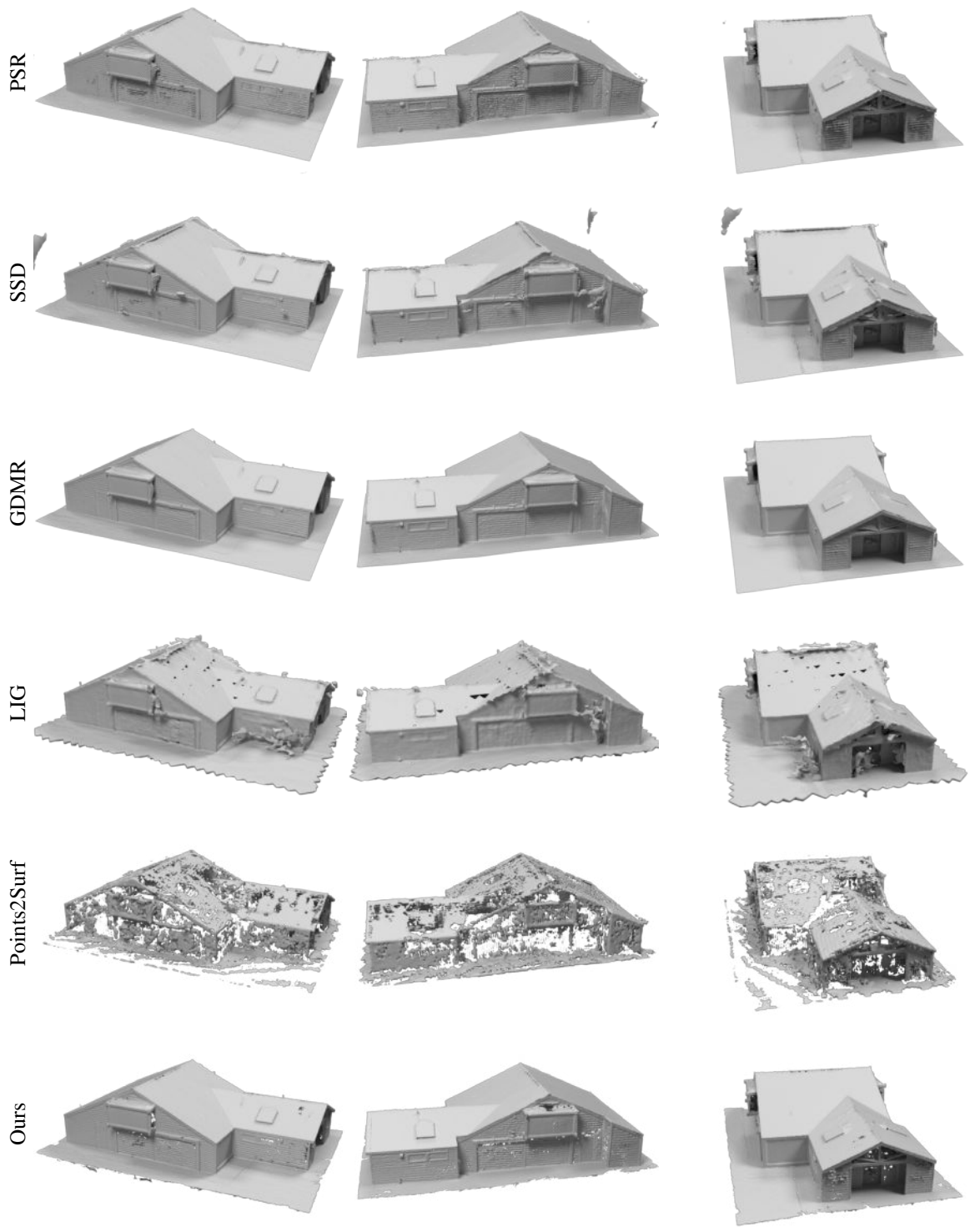


Figure 10: Barn

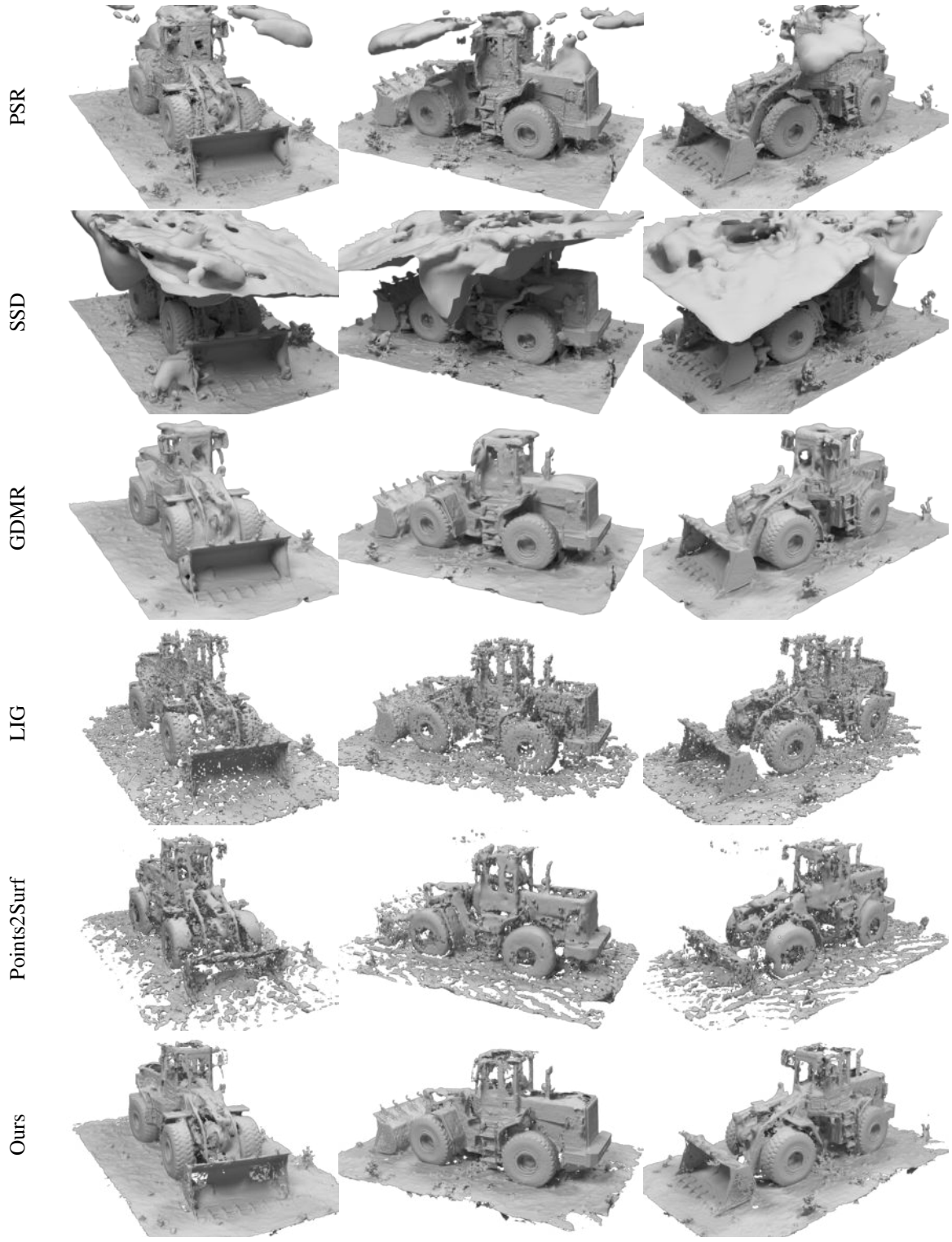


Figure 11: Caterpillar

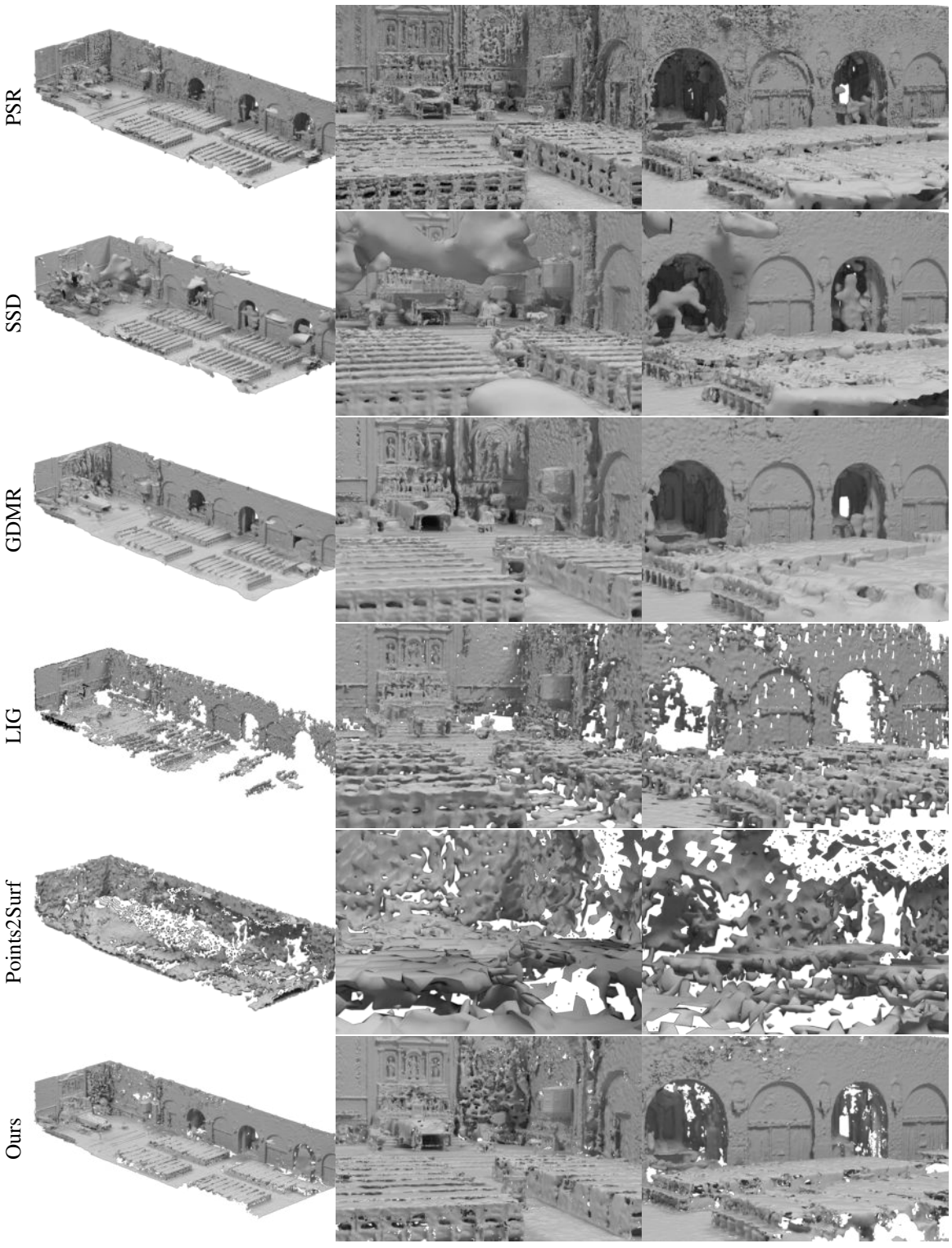


Figure 12: Church

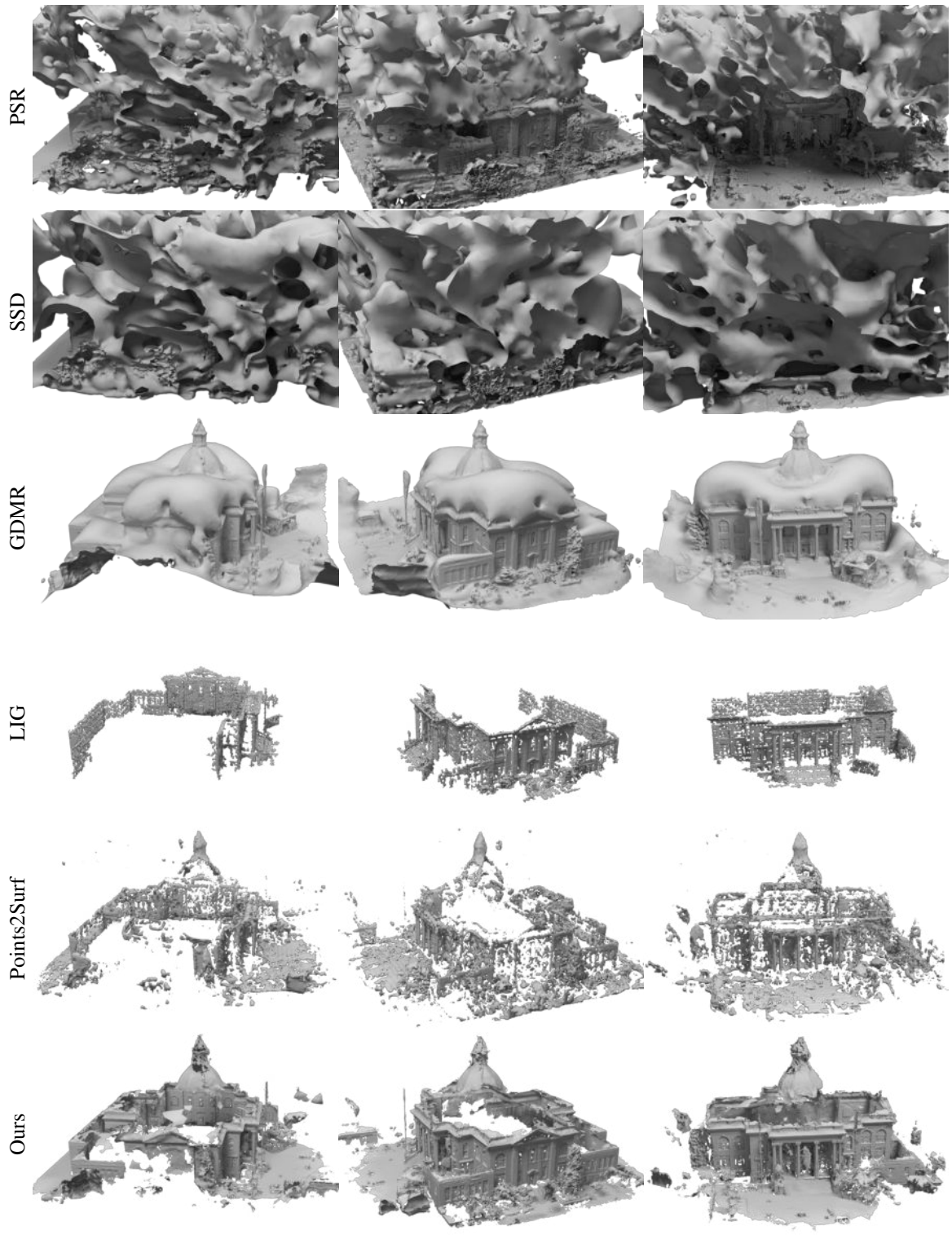


Figure 13: Courthouse

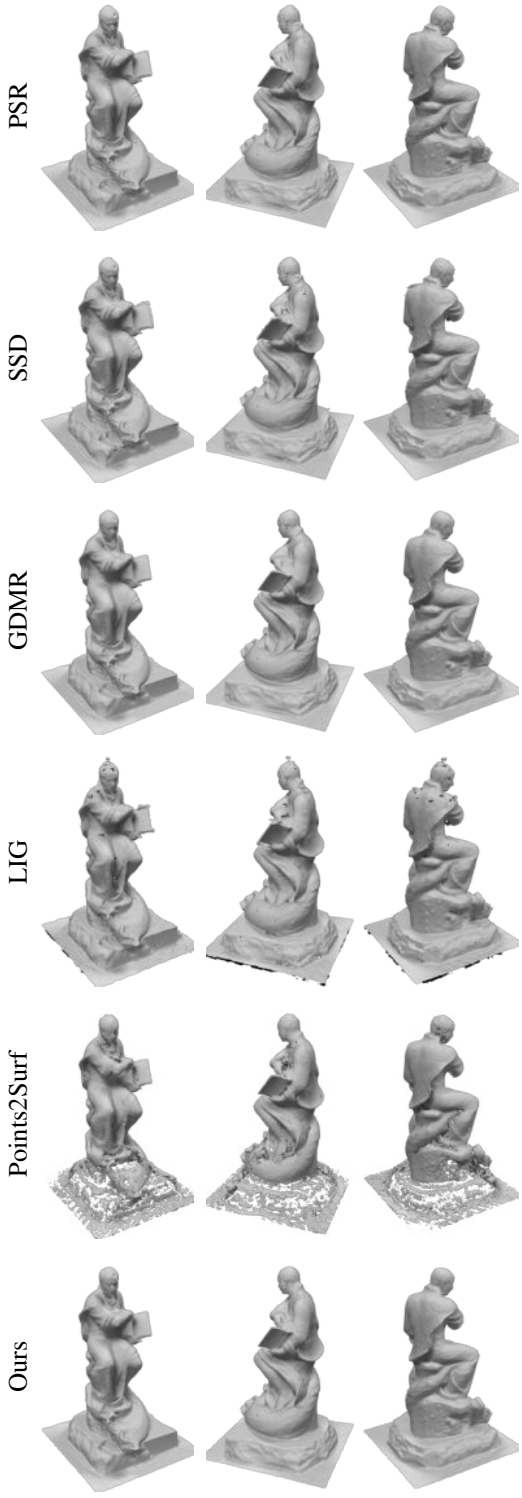


Figure 14: Ignatius

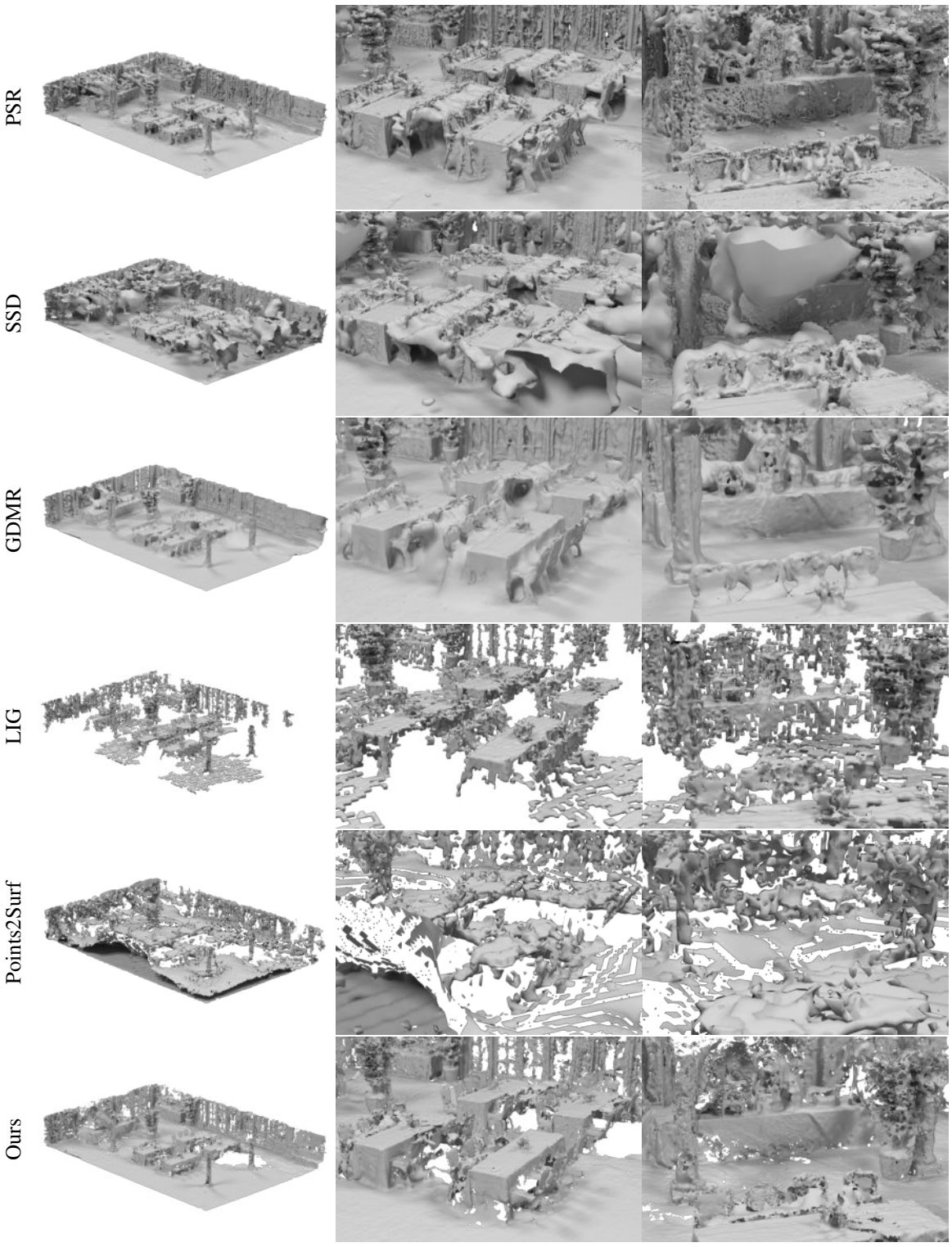


Figure 15: Meetingroom

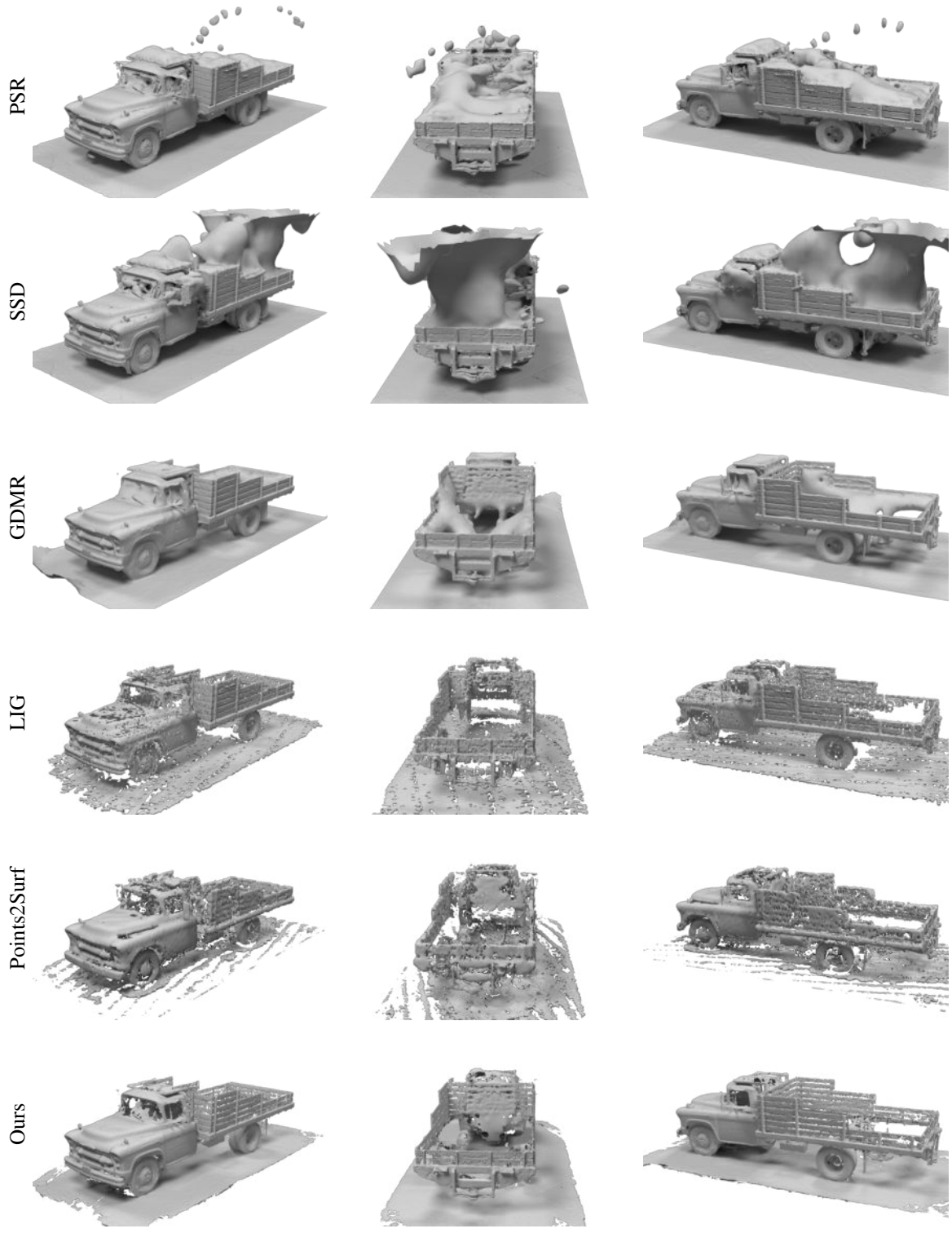


Figure 16: Truck