

Supplementary: Real-time Neural Rendering of LiDAR Point Clouds

1. Depth filtering implementation details

We preprocess the point cloud into a 3D grid where each cell is $0.5\text{m} \times 0.5\text{m}$ for typical point clouds, which facilitates frustum culling. Whenever two points are projected onto the same pixel, if the difference in depth is smaller than 1cm, we average their values to produce a representative depth estimate. After projection, the image is downscaled multiple times ($n = 4$ levels for 1920×1440) using min-pooling 2×2 kernel and stride 2, which shrinks the image by a factor of four at each level. During upscaling, each pixel at a higher resolution is compared to the matching pixel at a smaller resolution. A pixel is kept only if it meets this condition:

$$d_h \leq d_l \cdot 1.025$$

where d_h is the depth at the higher resolution, d_l is the depth at the smaller resolution. If a pixel doesn't meet the condition, the depth value from the smaller resolution is linearly interpolated and used instead. However, issues can occur on edges, removing too many pixels due to the large difference in depth. Hence, we detect edges using a Laplacian filter and increase the neighborhood to 3×3 for pixels that lie on an edge. This process is repeated step-by-step until the depth map is restored to its original size. In the final step, instead of interpolating depth values from smaller resolutions, we keep only the pixels that meet the filtering conditions. The full algorithm can be seen in Algorithm 1.

2. Synthetic data

We construct a synthetic dataset using ScanNet++ v1 data by altering ground truth registered photos to visually resemble raw point cloud projections. The key aspect is the impact of missing points caused by point cloud sparsity and the removal of points through depth-guided filtering. To achieve this, we project the point cloud from a slightly misaligned pose corresponding to the ground truth image, retaining only the depth D^s and alpha channel A^s . The ground truth $(RGB)^{gt}$ pixels are zeroed out wherever A^s is zero, resulting in $(RGB)^s$, forming the final training pair $[(RGBDA)^s, (RGB)^{gt}]$.

To enhance generalization, we simulate multiple scanner poses by introducing random brightness and contrast variations. Specifically, we generate between 2 and 6 random brightness-contrast pairs to mimic imaging differences across scanner positions. For each pixel, we randomly select one of these pairs and apply the corresponding transformation. Additionally, Gaussian noise with

$\sigma = 10.0$ is added to random pixels (with intensity levels from 0 to 255), with an 80% probability of noise being applied to each pixel.

We also trained a variant of our U-Net that does not assume a depth-filtered input. For synthetic training data, we adapt the previous approach to include background leakage from the point cloud. This requires determining whether a filled pixel ($A^s = 1$) belongs to the foreground or background, a decision made using our depth-guided filtering heuristic. Foreground pixels are assigned $(RGB)^{gt}$ values, while background pixels receive $(RGB)^s$. Pose misalignment for background pixels is less critical, as the network is expected to learn to disregard them.

3. Training details

We trained our network for 170 epochs on 36 ScanNet++ v1 scenes, consisting of 19,188 training image pairs and 4,798 test image pairs, where each pair consists of a ground-truth image and a rendered LiDAR point cloud image. We used random crops of 320×320 pixels to reduce memory usage during training.

Algorithm 1 Depth Filtering Algorithm**Require:** $max_filter_depth = 4$, $filter_strength = 1.025$, $input_depth_img \neq null$

```

1: function FILTER( $input\_depth\_img$ )
2:    $imgs \leftarrow$  array of size  $max\_filter\_depth + 1$ 
3:    $imgs[0] \leftarrow input\_depth\_img$ 
4:   for  $i = 0$  to  $max\_filter\_depth - 1$  do                                     ▷ Create hierarchy by minpooling images
5:      $imgs[i + 1] \leftarrow$  MinPool( $imgs[i]$ )
6:   end for
7:   for  $i = filter\_depth$  down to  $1$  do                                     ▷ Upsample the images
8:      $edges \leftarrow$  Laplacian( $imgs[i]$ )
9:      $mask \leftarrow$  Upsample( $imgs[i]$ ,  $imgs[i - 1]$ ,  $edges$ )
10:    if  $i == 1$  then                                                       ▷ Last upsample step only keeps pixels in mask
11:       $imgs[i - 1] \leftarrow$  Apply( $mask$ ,  $imgs[i - 1]$ )
12:    else
13:       $resized \leftarrow$  Resize( $imgs[i]$ )                                     ▷ Resize image with linear interpolation
14:       $imgs[i - 1] \leftarrow$  FillFrom( $resized$ ,  $!mask$ )                       ▷ Fill all pixels not in the mask with value from interpolated image
15:    end if
16:  end for
17: end function
18: function UPSAMPLE( $low\_res$ ,  $high\_res$ ,  $edges$ )
19:    $mask \leftarrow$  array with same size as  $high\_res$ , initialized to 0
20:   for each pixel  $(x_{high}, y_{high})$  in  $high\_res$  do
21:      $(x_{low}, y_{low}) \leftarrow$  round( $(x_{high}, y_{high})/2$ )
22:     if  $edges(x_{low}, y_{low}) == 1$  then                                     ▷ Edge pixel handling, check value with all neighbors of the low-res pixel
23:       if  $high\_res(x_{high}, y_{high}) \leq \max_{(x,y) \in neighbors(x_{low}, y_{low})} (low\_res(x,y) \cdot filter\_strength)$  then
24:          $mask(x_{high}, y_{high}) \leftarrow 1$ 
25:       end if
26:     else                                                                 ▷ Non-edge pixel handling, only check against corresponding low-res pixel
27:       if  $high\_res(x_{high}, y_{high}) \leq low\_res(x_{low}, y_{low}) \cdot filter\_strength$  then
28:          $mask(x_{high}, y_{high}) \leftarrow 1$ 
29:       end if
30:     end if
31:   end for
32:   return  $mask$ 
33: end function

```

Table 1: This table shows the results for 1920×1440 resolution renders using our method and Pointersect with different levels of point cloud decimation (%). Metrics include PSNR, LPIPS, SSIM, DREAMSIM and rendering speed (fps). Our method consistently outperforms Pointersect across all metrics and point cloud sizes.

Method	%	PSNR \uparrow	LPIPS \downarrow	SSIM \uparrow	DREAMSIM \downarrow	FPS \uparrow
Ours	10	15.48	0.36	0.73	0.30	13.76
Ours	40	16.07	0.34	0.74	0.21	13.75
Ours	70	16.10	0.34	0.74	0.20	13.70
Ours	100	16.10	0.34	0.74	0.19	13.63
Pointersect	10	15.00	0.47	0.64	0.31	0.031
Pointersect	40	14.95	0.49	0.60	0.26	0.013
Pointersect	70	14.94	0.49	0.60	0.25	0.008
Pointersect	100	14.94	0.49	0.60	0.24	0.006

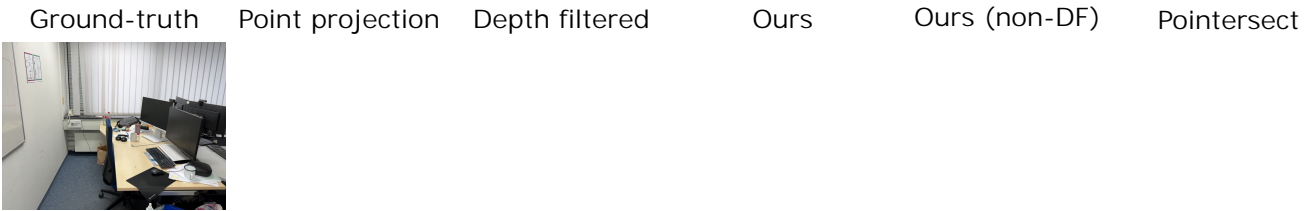


Figure 1: This figure illustrates the steps in our method and compares it with Pointersect and our method without depth filtering (non-DF). We see that our method creates more accurate and detailed images compared to the non-DF method. We also show the impact of depth filtering on the initial point cloud. We want to point out the differences in color between the point cloud data and image data, which impacts the metrics

Table 2: This table presents the results for 1920×1440 resolution renders, comparing our method, our method without depth filtering (non-DF), and Pointersect. The evaluation metrics include PSNR, LPIPS, SSIM, DREAMSIM, and rendering speed (FPS). Interestingly, the non-DF version of our method achieves virtually the same quality metrics as the DF. However, as illustrated in Fig. 1, the non-DF method exhibits more objectionable artifacts, which may be explained by the fact that our ground truth images are not perfectly aligned.

Method	PSNR \uparrow	LPIPS \downarrow	SSIM \uparrow	DREAMSIM \downarrow	FPS \uparrow
ours	16.10	0.34	0.74	0.19	13.63
ours (non-DF)	16.39	0.33	0.76	0.20	13.28
pointersect	14.94	0.49	0.60	0.24	0.006