Supplementary Material for Unbiased Gradient Estimation for Differentiable Surface Splatting via Poisson Sampling

Jan U. Müller[1], Michael Weinmann[2], and Reinhard Klein[1]

[1] Institute of Computer Science, University of Bonn, Germany
[2] Department of Intelligent Systems, Delft University of Technology, Netherlands

I Isotropic Approximation of Multivariate Gaussians

Greengrad and Strain [5] proposed the use of Fast Multipole Methods (FMM) for approximating a discrete Gauss transform (DGT) $G(x)$ which is defined as

$$G(x) = \sum_{r=1}^{N} q_k e^{\frac{\|x-x_k\|_2^2}{2h^2}}$$

over a large number of source centers x_k and target points x in linear time. The FMM assumes that the Gaussian kernels are isotropic and that all source points x_k share the same standard deviation h. However, the (unnormalized) cumulative probability $\sigma_n(x) := \sum_{k \in I_n} f_k \rho_k(x)$ in Algorithm 1 (in the main paper) is the sum of multivariate and anisotropic Gaussian densities. Nevertheless, we use FMM to approximate the weight function under the assumption that the Gaussian densities are close to being univariate and isotropic.

We map $\sigma_n(x)$ to a DGT by defining the weight q_k as the attribute value of point k, i.e. $q_k := |f_k|$. If the attribute value is not a scalar but a vector, we define q_k to be the average value of the attribute value. Next, we introduce an isotropic upper-bound for the Gaussian kernel $\rho_k(x)$: Let $k \in [1 : N]$ be an arbitrary point and $M_k := J_k V_k J_k^T + I$ its covariance matrix were J_k is the Jacobian of the screenspace projection and V_k is a diagonal scaling matrix. Since M_k is a covariance matrix, it is positive definite and there exists an eigendecomposition $M_k = Q \Sigma Q^T$, where Q is an orthogonal matrix of eigenvectors and Σ is a diagonal matrix of the eigenvalues λ_1, λ_2. Let $y := Q^T(x - m(u_k))$ be the rotated mean-centered input which allows us to write the exponential term of the Gaussian kernel ρ_k as

$$\begin{aligned} &\exp\left(-\frac{1}{2}(x - m(u_k))^T M_k^{-1}(x - m(u_k))\right) \\ =&\exp\left(-\frac{1}{2} y^T \Sigma^{-1} y\right) = \exp\left(-\frac{1}{2}(\lambda_1^{-1} y_1^2 + \lambda_2^{-1} y_2^2)\right) \end{aligned}$$

where the last step uses the fact that $y \in \mathbb{R}^2$. Setting $h_k = \frac{1}{\sqrt{\lambda^*}}$ where $\lambda^* = \min\{\lambda_1^{-1}, \lambda_2^{-1}\}$ allows to upper-bound the exponential term with

$$\exp\left(-\frac{1}{2}(\lambda_1^{-1}y_1^2 + \lambda_2^{-1}y_2^2)\right) \leq \exp\left(-\frac{1}{2h_k^2}\|y\|_2^2\right) = \exp\left(-\frac{\|x_q - m(u_k)\|_2^2}{2h_k^2}\right)$$

since Q is orthogonal which implies $\|Qy\|_2 = \|y\|_2$. Given the list of isotropic standard deviations $h_1, ..., h_N$, we define the common standard deviations, which is used in the DGT, to be the average of the individual standard deviations $h := \frac{1}{N}\sum_{k=1}^{N} h_k$.

Intuitively, the Gaussian kernels have an elliptical distribution where the eigenvectors determine the principal directions and the eigenvectors determine their length. We bound these elliptic kernels from above with isotropic kernels whose radius encloses these ellipses. Finally, the circular kernels are approximated with a kernel whose radius is set to be the average of the circular kernels. Naturally this underestimates the influence of some points but we compensate for this problem by increasing the number of points drawn per pixel. Depth-filtering in turn helps to remove points whose probability is overestimated but which have no visual contribution.

In our implementation, we compute the common standard deviation h in a pre-processing step before the tree construction starts. The eigenvalues are computed using the closed-form solution of the eigen-problem for matrices in two dimensions [3]:

$$\lambda_{1\backslash 2}^{-1} = \frac{1}{2}\mathrm{tr}\left(M_k^{-1}\right) \pm \sqrt{\frac{1}{4}\mathrm{tr}\left(M_k^{-1}\right) - \det\left(M_k^{-1}\right)}.$$

To avoid numerical instabilities which arise from the computation of the Jacobians or inversion of the covariance matrix, we clamp the eigenvalues by computing $h_k = \min\{\lambda_1, \lambda_2, 10\}$. The computation of these eigenvalues is performed in parallel on the multi-processor such that each thread computes the standard deviation h_k for a single point $k \in [1 : N]$. The average of these values is computed using the `reduce` function provided in the thrust library [1].

II Additional Implementation Details of Algorithm 1

In order to make the sampling method more accessible to interested readers, we first give an intuitive description of the presented algorithm. We then provide further implementation details and finally analyze the runtime of the algorithm.

II.I Intuitive Description

Algorithm 1 (in the main paper) computes a set of point indices with size m for each pixel in parallel by adapting a sequential Poisson sampling (SPS) design [15]. The set is with probability $1 - \epsilon$ a SPS sample where ϵ is a user-defined

certainty. A set of points is a SPS sample if it contains the m smallest transformed random numbers $\xi_{x,k}$. Furthermore, the algorithm avoids to visit every point which allows its expected runtime to scale near linear with respect to the number of points and pixels.

Let n denote a node of the AABB-Tree, I_n be the indices of points which are contained in the sub-tree of n. Furthermore let $\sigma_n(x)$ denote the cumulative probability of points in I_n. Let x be an arbitrary pixel and $c_{x,n}$ be the capacity of a node n when drawing a sample for pixel x. The initial value of the capacity is set to be one for every node. The proposed procedure then constructs a SPS sample S_x by alternating between a descent and backtracking step:

The traversal in descending direction starts from the root node and continuous down the tree by following the child node with the largest weight $c_{x,n} \cdot \sigma_n(x)$ until the algorithm reaches a node n which has at most m points in its sub-tree i.e. $|I_n| \leq m$. The procedure then computes transformed random values $\xi_{x,k}$ for all points in $k \in I_n$ and sorts the points by their transformed random number to be in ascending order. If the sample S_x is still an empty list, it is filled with the sorted points and their transformed random numbers. Otherwise, the procedure merges the points in I_n with the points in the sample S_x such that the resulting list is a SPS sample, i.e. the points in S_x are in ascending order according to their transformed random values, and the sample has a length of m. Note that the values $\xi_{x,j}$ of points which are currently in the sample $j \in S_x$ are constants.

The backtracking step updates the capacities $c_{x,q}$ along the traversed path which avoids that the algorithm tries to merge the sample S_x with points that have already been considered in a previous iteration. Let $n_1, n_2, ...n_d$ be the nodes on the descent path where n_1 is the root node. The capacity of a visited node n_i is updated by subtracting the fraction of probability mass that was contributed by n_d to the total mass of n_i from the remaining capacity:

$$c_{x,n_i} \leftarrow c_{x,n_i} - \prod_{j=i}^{d} \frac{\sigma_{n_j}(x)}{\sigma_{n_j}(x) + \sigma_{n'_j}(x)}$$

where n'_j is the node in the tree which shares a parent with the node n_j. In the next subsection, we derive the stopping criteria of the algorithm and analyze the algorithm's runtime.

II.II Detailed Runtime Analysis

Algorithm 1 repeatedly alternates between the descent and backtracking step for each pixel x until the product of the value ξ and the remaining probability mass $c_{x,r}\sigma_r(x)$ at the root is smaller than the user-defined threshold ϵ, i.e.

$$\epsilon > \xi \cdot c_{x,r} \cdot \sigma_r(x). \tag{1}$$

The value ξ is updated in each iteration to be the largest transformed random number of any element that is currently in sample S_x.

To derive the bound in Equation 1 for an arbitrary pixel x, we first consider the probability that an individual point changes the current sample S_x in the next iteration of the algorithm. Let I_R be the set of points that have not yet been visited by our algorithm. For an arbitrary point $k \in I_R$ in the remaining points to change the sample S_x, its transformed random value $\xi_{x,k}$ has to be smaller than ξ, which is the largest transformed random value among the points in S_x. Since $\xi_{x,k} = \frac{u_{x,k}}{\tilde{p}_k(x)}$ where $u_{x,k}$ is drawn from a uniform distribution on the interval $[0, 1]$ and ξ is a constant, the probability for k to change S_x is

$$P(\xi_k < \xi) = \tilde{p}_k(x)\xi.$$

Next, recall that $u_{x,k}$ is drawn independently for every point $k \in I_R$, which implies that the probability that the sample S_x changes in the next iteration is the sum of the individual events $\xi \sum_{k \in I_R} \tilde{p}_k(x)$. Let r be the root node of the tree, according to the definition of the weight function $\sigma_r(k) = \sum_{k \in I_r} f_k \rho_k(x)$, where I_r is the set of points in the subtree of root r and therefore the set of every point in the point cloud. Finally, observe that I_R is a subset of I_r and that the ratio between the probability masses in both sets is by definition given by $c_{x,r}$. Therefore, the probability mass of all remaining points I_R is the total probability mass of all points $\sigma_r(x)$ multiplied with the remaining capacity $c_{x,r}$ of the root node

$$\sum_{k \in I_R} \tilde{p}_k(x) = c_{x,r} \cdot \sigma_r(x).$$

Consequently, the probability that S_x changes in the next iteration is $\xi \cdot c_{x,r} \cdot \sigma_r(x)$. By stopping the Algorithm if this probability is smaller than ϵ we obtain a sample which is with probability $1 - \epsilon$ a SPS sample.

Our algorithm always converges since ξ is monotonically decreasing and the capacity of the root is strictly monotonically decreasing. The value of ξ is monotonically decreasing because the merging step in Algorithm 1 always select the points with the m smallest values of $\xi_{x,k}$. The capacity is strictly monotonically since every point has a non-zero contribution to $\sigma_r(x)$ and is only considered once by the algorithm. The runtime of the proposed algorithm is a random variable and depends on the expected value of ξ, but more importantly also on the convergence rate of $c_{x,r} \cdot \sigma_r(x)$ towards zero. A worst-case bound for the expected value of ξ is $\mathbb{E}[\xi] = \frac{1}{2\pi_k} < \frac{1}{2\epsilon}$ if the precision cut-off is ϵ. Unfortunately, the convergence of $c_{x,r} \cdot \sigma_r(x)$ is highly dependent on the instance and the quality of the AABB tree, which makes it difficult to analyze. Additionally the algorithm is unable to sample duplicates since the capacity of nodes visited in a previous iteration is zero and every non-visited node has a weight $\sigma_n(x) > 0$ due to the infinite support of the Gaussian kernels.

II.III Additional Implementation Details

Algorithm 1 (in the main paper) initializes all capacity values $c_{x,n}$ to be 1 and requires access to $c_{x,n}$ for any combination of pixels x and node n in the tree. Our implementation uses the following observation to improve its space efficiency: It

is not necessary to store the capacity values for every combination of pixels x and node n in the tree since the descent stops if a node has fewer than m points in its sub-tree and the algorithm terminates if the expected likelihood that the sample changes is below $1-\epsilon$. Our runtime analysis in the previous section allows us to estimate that the algorithm visits on an average instance at least $\frac{1}{2\epsilon}$ unique nodes. In practice we found that the actual number of unique nodes visited by the algorithm is much smaller and that a hashmap with size $m \cdot h$ is sufficient where m is the number of samples to be drawn and h is the depth of the tree. To efficiently store the capacity for visited nodes our implementation maintains a hashmap for each pixel. Each cell within the map stores a key as an integer and the capacity as a floating point number. When Algorithm 1 (in the main paper) accesses a node's capacity, its index is used as the key to find the node in the pixel's hashmap via linear probing. If the key is not present, the capacity is returned as one and the key is inserted into the hashmap; otherwise the node's capacity is returned or the existing capacity is updated. The index is converted into a key by utilizing Jenkins hash function [7].

An additional implementation detail regarding the sampling algorithm is that the subset I_n of points which are contained in the subtree of node n is not stored explicitly. The implementation instead stores the number of points $|I_n|$ within the subtree for each node n and only if n is a leaf the point's index is stored in the node. If I_n needs to be accessed, the implementation traverses all leafs of the sub-tree using a stack-less traversal.

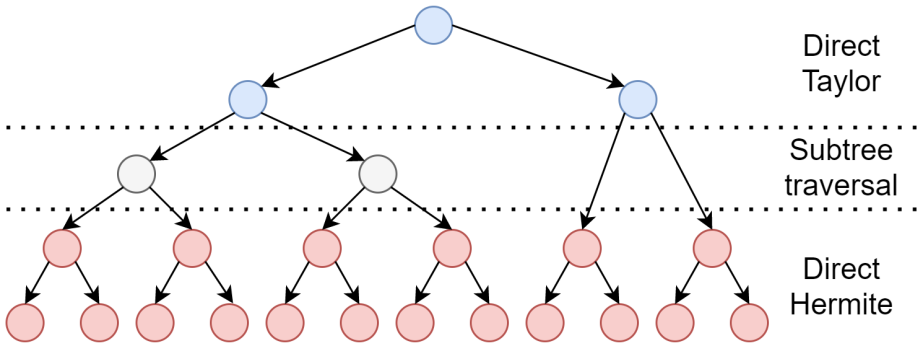


Fig. 1: Example of the FMM evaluation techniques used for different nodes within the tree data-structure. Note that the depth in which Hermite expansions exist is strongly dependent on the point distribution in screen space and can be on different levels for many subtrees.

Hermite to Taylor Conversion Section 3.1 (in the main paper) outlines a simple heuristic which the sampling procedure uses in its pre-processing step to decide which Hermite coefficients are converted into Taylor coefficients. Afterwards $\sigma_n(x)$ is evaluated either by directly evaluating the Taylor or Hermite

expansions if it is available, otherwise the subtree is traversed and Hermite coefficients of the child nodes are accumulated. Figure 1 illustrates the different evaluation methods used throughout the nodes in the tree. The proposed method divides the image grid into pixel cells with side length $l = 4$ and each thread block on the multi-processor processes 16 cells, where the pixels are mapped one to one onto the threads. In a pre-processing step, the threads within a cell compute the Taylor expansions for frequently evaluated nodes and store the expansions in shared-memory. The shared memory is limited to 8192 bytes in order to maintain close to 100% occupancy of the multi-processor. This limits the number of Taylor expansion per cell to $n_t = 5$ given a Taylor degree of $p_t = 4$. Each thread within a cell whose index i is smaller than the maximum number of Taylor expansions computes a Taylor expansion for the node n which has index i in level order. Level order is used because the nodes close to the root are the most frequently evaluated which are also the most computationally expensive since they often have no valid Hermite expansion. Each thread then converts the Hermite expansion into a Taylor expansion using the results by Lee et al. [12] where the expansion point X_Q is the center of the cell. If a node does not have a Hermite expansion, the Hermite expansions within the sub-tree are converted into Taylor expansions and the coefficients are accumulated similar to the approach by Lee et al. [12].

Possible Extension We would like to emphasize that the limit of 40 samples per pixels is the result of the used hardware and is not a fundamental limit of our proposed algorithm. However, we can outline changes to the current implementation to increase its efficiency: To help with debugging, the hashmaps used in Algorithm 1 are currently allocated in a buffer that covers the whole image. Allocating the hashmap on the stack frame of a thread limits the buffer size based on the active number of SM/threads. Also, switch to an atomics-based sorting for the depth filtering, instead of loading all samples into shared memory.

Our algorithm emphasizes near-linear scaling to enable a greater range of applications (e.g. processing room-scale and city-scale scenes). However, an additional code path might use direct evaluation instead of Hermite/Taylor approximations if the instance is small.

III Experiment Details, Parameters and Additional Evaluations

In this section, we describe all parameters, architectures and additional techniques necessary to reproduce the results presented in our associated article. Furthermore, this section contains additional figures that were created during the experiments of the main article but could not be included due to the page limit.

III.I Comparison - Image-based Shape Reconstruction

Dataset, Shading Model and Initialization The object reconstruction is demonstrated on the synthetic datasets provided by Yifan et al.[20]. This dataset contains four point clouds that are reconstruction targets and two point clouds that used as the initial estimate for the optimization procedure. The optimization targets are referred to by Bunny, Teapot, Yoga1 and Yoga6. Each mesh is viewed from 300 unique camera poses, which are distributed on a sphere around the object. The dataset does not contain training or testing images and instead uses the differentiable renderer to render the ground truth point cloud from the selected training poses with a resolution of 256×256 pixels.

The illumination model uses three directional light sources, which are positioned relative to the camera pose (i.e. the lighting is fix in cameras space). The shading is a simple diffuse Lambert BRDF which is evaluated for each point and the resulting color values are interpolated using the splatting algorithm.

The Bunny and Teapot point clouds both have 8000 points and are to be reconstructed from 8000 points that are distributed on a centered sphere. The Yoga1 and Yoga6 point clouds both have 20000 points and use points that are distributed on a cube as initialization. The initial point clouds are depicted in Figure 2. The normals are initialized using a standard PCA procedure based on

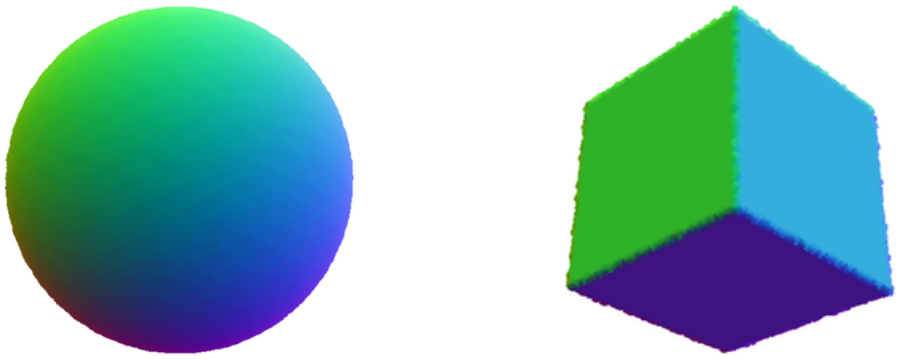


Fig. 2: Point clouds that are used as the initialization in the shape reconstruction experiments and which provided in the dataset by Yifan et al. [20]. The sphere is used for Bunny and Teapot objects whereas Yoga1 and Yoga6 use the right point cloud as their initialization.

the point cloud and the diffuse albedo values are initialized with ones.

Optimization During the shape recovery, the point positions, normals and albedo values are optimized to minimize the L_1 distance to the reference images. The optimization is performed using the "Adam" optimizer [10] with a learning rate of 0.01 and the usual running average parameters of $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

The learning rate is reduced 5 times by a factor of 2.115 at regular intervals during the 300 epochs. Rendering using the proposed method is performed with $m = 40$ samples per pixel, and an error value of $\epsilon = 0.01$ is used for the FMM approximation.

Points are declared to be non-contributing if the sum of albedo values across all channels is less than a threshold $\tau = 1.0$. We chose a high threshold for this application to avoid outliers that have a small contribution to the image loss but would significantly increase the Hausdorff or Chamfer distance. For each non-contributing point $\hat{p}$ a point p, which has a contribution above τ, is chosen at random. An offset $p' = p + \eta d$ from the chosen points is computed where d is a randomly sampled direction on the unit sphere and $\eta = 0.1$ is a user-chosen offset step size. The choice of the parameter η depends on the scale of the objects and the selected splat size. We found that a value close to 1% of the bounding box diagonal works well for this application. Afterwards the point is projected into the tangent plane of p by computing the orthogonal component $o = \langle p' - p, n \rangle$ and subtracting it from the new position $\hat{p} \leftarrow p' - o \cdot n$.

Post-processing We perform a final clean-up step to remove non-contributing points which might remain after the optimization converged. The optimized point cloud is rendered from all camera poses used during training and the depth filtered weights are accumulated for each points. The accumulation is performed by utilizing the per pixel sample $\tilde{\mathcal{N}}_x$ which contains the indices of the sampled points and thus allows a mapping between points and pixels. Afterwards, a point is pruned if the total contribution across all poses is smaller than the threshold $\tau = 0.1$.

Additional Comparison and Discussion We provide a visualization of the convergence rate of different methods in Figure 3. We observe that our method converges faster than DSS [20] and results in a more accurate surface representation than DSS and Pulsar [11]. Table 4 provides a detailed comparison between the runtime of our algorithm and DSS. The runtime values in milliseconds demonstrate that our algorithm scales better to larger instance sizes compared to DSS. In addition, we provide a visualization of the point-wise error computed between the reconstructions and the ground truth point cloud in Figure 4 and 5 that were used to compute the Chamfer and Hausdorff distance in Section 4.

Detailed Statistics The Hausdorff (HD) and Chamfer (CD) distances reported for our method in Table 1 (in the main paper) are averaged over 10 runs and have the following standard deviations: Bunny HD:0.442 ± 0.0013\CD:0.125 ± 0.0013, Teapot HD:0.453 ± 0.0016\CD:0.289 ± 0.00082, Yoga1 HD:3.355 ± 0.0113\CD:1.385 ± 0.0026 and Yoga6 HD:1.551 ± 0.0121\CD :0.517 ± 0.0045.

Additionally, we report the mean of the partial derivatives estimated using Equation 3 (in the main paper) along side their standard deviation in Tab. 5 that occurred within one epoch during the shape reconstruction experiments. Among all epochs and partial derivatives $\frac{\partial L}{\partial w_k}$, we selected the partial derivative

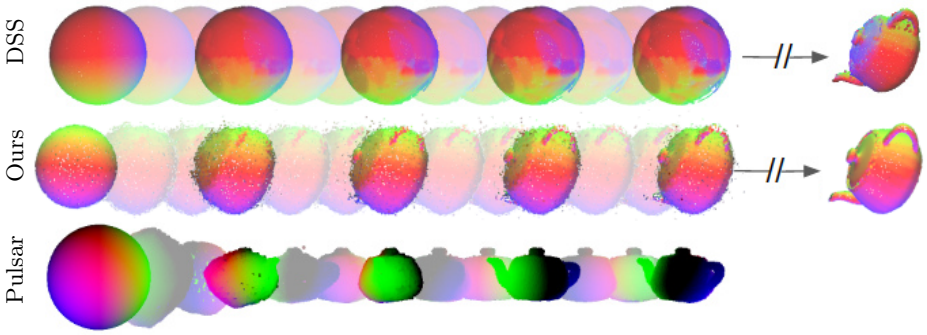


Fig. 3: Comparison between the convergence speed of the deformation between DSS [20], Pulsar [11] and our method. Even though the deformation with the sphere-based Pulsar converges faster than the other methods, its spherebased surface approximation is of lower quality for the same number of points. The proposed method converges faster than DSS. This figure is based on Figure 4(b) in Lassner and Zollhöfer's work[11].

Size	$N = 8 \times 10^3$		$N = 2 \times 10^4$		$N = 1 \times 10^5$	
	Forward	Backward	Forward	Backward	Forward	Backward
DSS	19.3 ms	79.9 ms	42.8 ms	164.6 ms	258.1 ms	680.2 ms
	99.2 ms		207.4 ms		938.3 ms	
Ours	91.9 ± 1.2ms	103 ± 1.8ms	118.3 ± 1.7ms	160.6 ± 2.5ms	366.8 ± 2.6ms	297.1 ± 0.6ms
	194.9 ± 2.8ms		278.9 ± 3.2ms		663.9 ± 2.4ms	

Table 4: Comparison of the runtime in milliseconds between the proposed method and the runtime values reported for DSS by Yifan et al. [20]. The table reports the average time of the forward and backward pass as well as the combined time. Note that the proposed method scales better to larger instances even though the values for DSS were gathered on the faster Nvidia GTX 1080Ti.

with the highest std. deviation. The decreasing std. deviation aligns well with the distances reported in Table. 2 (in the main paper).

III.II Application - Room-scale Scene Refinement

Dataset, Shading Model and Initialization The refinement of large-scale point clouds is demonstrated on the dataset for room-scale SLAM provided by Steinbrücker et al. [18] and Bode et al.[2]. Both datasets contain a sequence of RGB-D images captured using the first and second generation Microsoft Kinect respectively. Bode et al. [2] register the captured RGB image with the depth images before performing the reconstruction. The initial point cloud, camera poses and diffuse albedo values are obtained by using the SLAM procedure proposed by Keller et al. [9]. Before the point cloud is refined, the number of views in each dataset is reduced to 112 by using farthest-point sampling, where

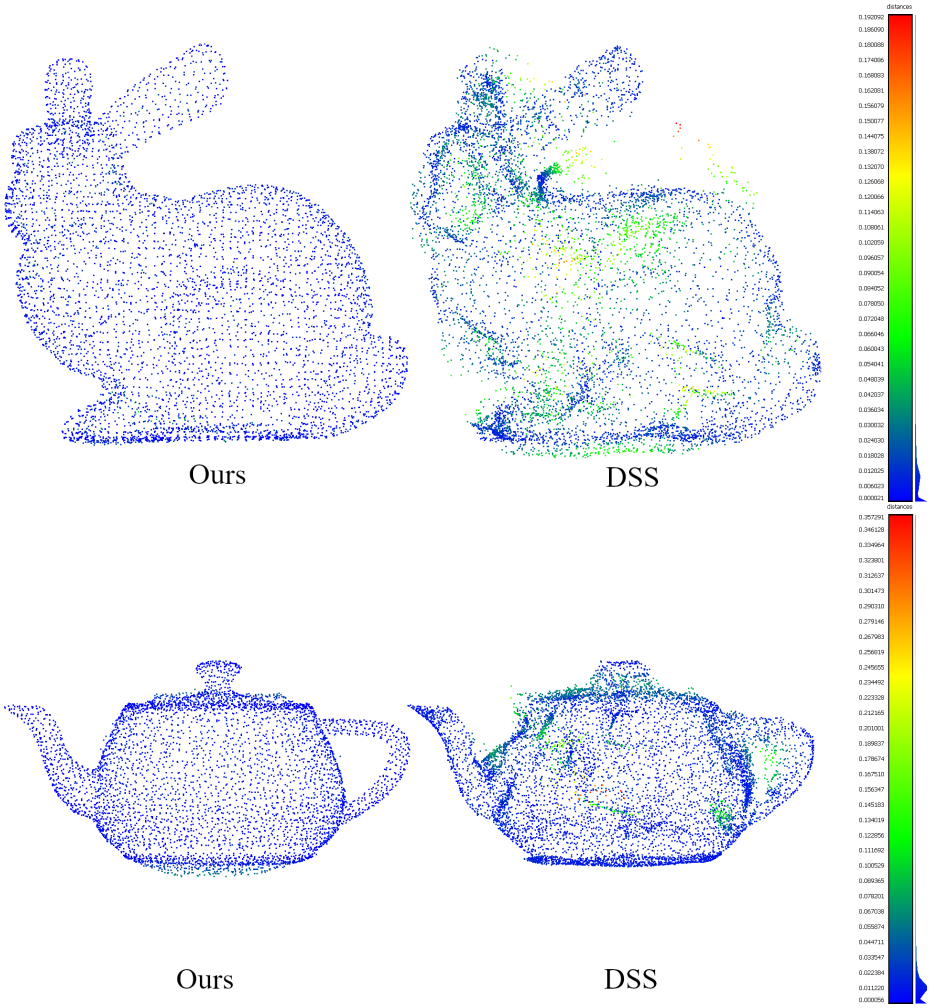


Fig. 4: Point-wise error computed between the reconstructions and the ground truth pointset. The results for DSS were obtained using the code and parameters published by its authors [21].

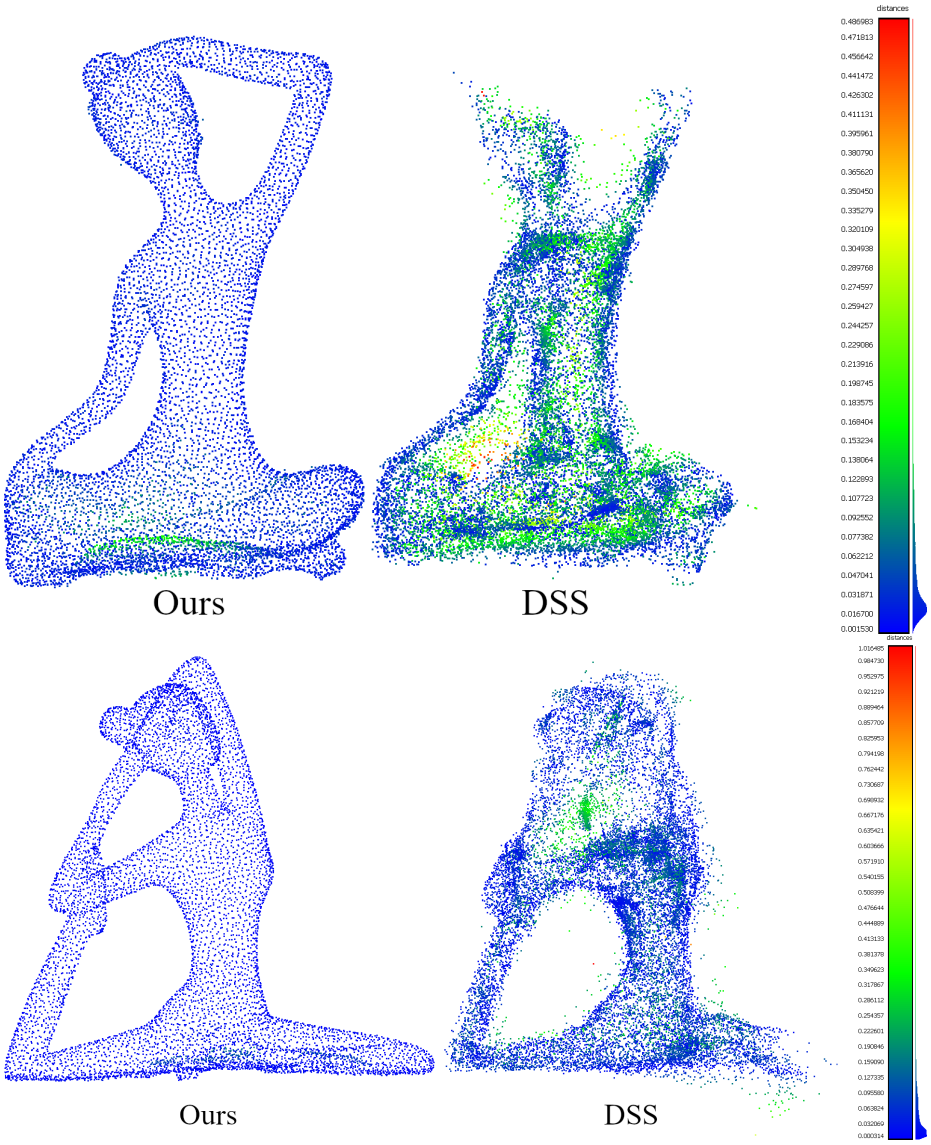


Fig. 5: Point-wise error computed between the reconstructions and the ground truth pointset. The results for DSS were obtained using the code and parameters published by its authors [21].

Set	#Samples per pixel			
	10	20	30	40
Bunny	13.14 ± 3.05	14.02 ± 1.19	16.56 ± 1.05	14.33 ± 0.46
Teapot	-8.79 ± 1.63	-10.58 ± 1.15	-10.16 ± 1.29	-10.99 ± 0.4
Yoga1	0.87 ± 0.57	-0.46 ± 0.4	-0.29 ± 0.13	-0.17 ± 0.08
Yoga6	0.24 ± 0.28	0.15 ± 0.21	0.06 ± 0.17	0.11 ± 0.01

Table 5: Average value of the partial derivative with the highest std. deviations during shape reconstruction on the Bunny, Teapot, Yoga1 and Yoga6 datasets for varying number of samples per pixel. Increasing the sample count per pixel decreases the variance on the partial derivative.

the distance between the estimated viewing directions is used as the criterion. This is done to remove parts of the sequence in which the capture device is stationary and to increase the likelihood that multiple view directions of the same part of the scene are included in the same batch.

In order to represent more complex diffuse illumination the simple shading model is replaced by a deferred shading in which spherical harmonics with 9 coefficients per color channel are used to represent the directional illumination [16]. The deferred shading pipeline uses the proposed method to filter the albedo and normal values to obtain an albedo and normal map, which are then used to calculate the final pixel values. Due to the flexibility of the implementation, changing the shading model only requires minor changes to the Python script without any changes to the CUDA extensions. The coefficients of the spherical harmonics approximation are initialized with uniformly sampled values from the interval $[-0.5, 0.5]$ except for the first coefficient, which is set to be 0.5. Deferred shading with spherical harmonics is more accurate and allows for a more efficient representation of the illumination.

Optimization In this application we alternate between optimizing the camera poses and jointly optimizing the point position, normals, albedo values and spherical harmonics coefficients. The pose of each frame in the dataset is optimized for 20 steps, with the learning rate halved after 10 steps. Then, the remaining scene parameters are jointly optimized for 20 steps, with the learning rate halved after 15 steps. This alternating optimization is repeated 10 times, continuing the optimization of the poses with the learning rate from the last step, while the learning rate of the remaining scene parameters is the same in each step. The learning rate of the poses becomes smaller to get as close as possible to the actual pose and because the optimization of the poses is much more sensitive to the learning rate. The higher learning rate of the other parameters allows to accommodate the changes resulting from the adjustment of the pose. Each camera pose is represented as a 7-dim. vector (3 coefficients for the position and 4 for a quaternion) and optimized using Adam to minimize a masked L_1-distance between the rendered image and the ground truth. The dataset provided by Bode et al. [2] contains reference images in which areas with unreliable

depth values are masked out. The masked L_1 loss discards these areas by using a mask that is derived from the depth images via a threshold $\tau' = 0.01$ which is well below the minimum depth measured by the sensor. The learning rate of the optimizer is 0.0005 with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. After each update, the quaternion is normalized to ensure that it represents a rotation; no further regularization is necessary. The optimization of the remaining parameters follows the setup described in the previous section but also uses the masked L_1 loss and reduces the learning rate to 0.005.

After each of the 20 optimization step we remove points which have a substantially larger error compared to the rest of the point cloud. The outlier removal step is based on the statistical outlier removal [17] but considers the per point error and uses a Laplace distribution as the underlying model. We identify these points by computing the total error of all points across all images in the dataset and analytically fitting a Laplace distribution to these points. The fitting process only estimates the variance of the distribution and keeps the mean zero-centered. Let F_σ^{-1} be the inverse CDF of the zero-centered Laplace distribution with variance σ. We prune all points whose error is larger than $F^{-1}(1e-3)$. The high point cloud densities in this application allow for point pruning while maintaining visual fidelity in the remaining point cloud. Only a few hundreds of the at least 10^6 points are removed in this process.

Additional Comparison and Discussion Section 4 (in our main paper) presents a detailed analysis of the improvement achieved on the dataset by Bode et al. [2] using our differentiable refinement process and a comparison between our method and DSS on the dataset by Steinbrücker et al. [18]. In this paragraph, we provide a more detailed discussion about the results on the dataset by Steinbrücker et al. [18]. In contrast to the dataset by Bode et al. [2], this dataset contains only views that show the scene in frontal view. Nevertheless, the refinement process achieves a 15.1% improvement of the SSIM. Comparing the reference image in Figure 6 with a rendered image before and after the refinement, it can be seen that the reconstruction of the poses on this dataset does not play an as large role as on the previous dataset. Despite this, it can be observed that the geometry and albedo values of the SLAM reconstruction do not match the reference images here either. In the SLAM reconstruction, the albedo values around the screen contain shadows that are not present in the RGB image sequence. The refinement corrects this and improves the reconstruction of the objects on the desk as shown in Figure 6.

III.III Application - Neural Rendering

In this paragraph which focuses on the neural rendering experiments, we describe the initialization, neural network architecture and optimization setup in more detail.

Dataset, Architecture and Initialization The synthetic dataset by Mildenhall et al.[14] is divided into a training and validation split that each contain 100

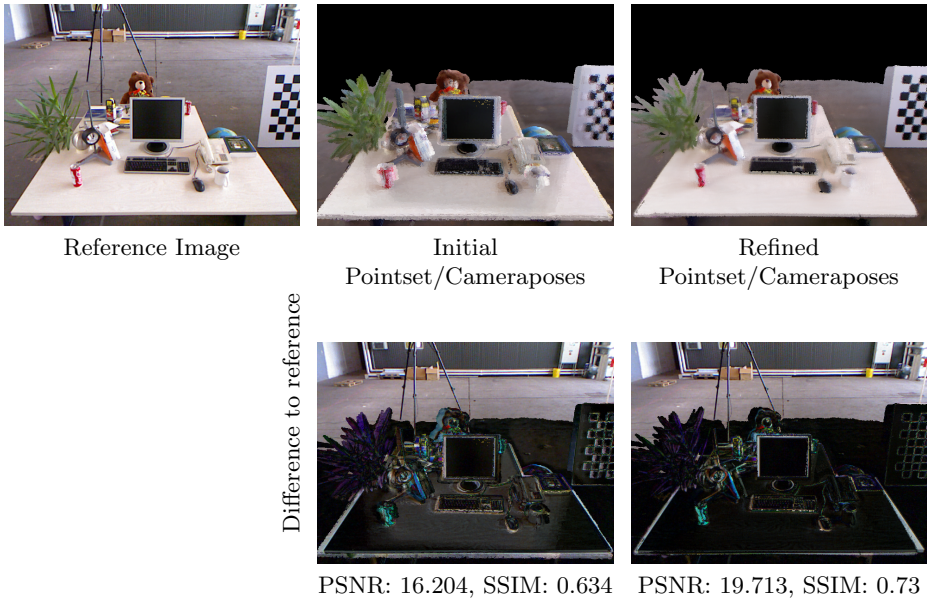


Fig. 6: Comparison between the reference image, a rendering of the point cloud, camera poses and albedo parameter obtained from real-time SLAM on the dataset provided by Steinbrücker et al.[18] and a rendering of the scene parameters after being refined using the proposed algorithm. Additionally, the pixel-wise difference between the reference image and the rendered images is also depicted (darker values correspond to a smaller error). The albedo values obtained using SLAM include shadows that are not present in the reference image. The refinement process is able to address these shadow artifacts and improves the alignment and overall sharpness of the rendered texture. Additionally, the geometry of foreground objects are improved (e.g. the oversized monitor screen border are removed).

images and their associated camera poses. Additionally, the dataset provides a test split with 200 images and poses and also contains the normal and depth images. The images are rendered using path-tracing and utilize image-based lighting. Since the available hardware has limited VRAM, the image resolution of this dataset is reduced from 800×800 to 256×256 pixels.

The design of the neural renderer follows the approaches of Lassner and Zollhöfer [11], which in turn adapt the approach of Wang et al.[19]. Their approach makes use of a Generative Adversarial Network [4] to enhance the image quality beyond what is achievable by only using a pixel-wise loss function. Gen-

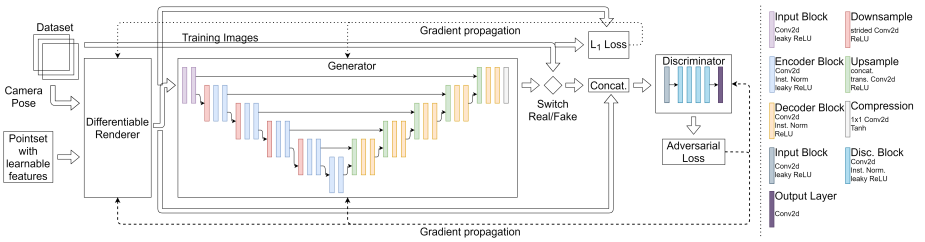


Fig. 7: Architecture of the neural renderer based on the Pix2Pix model proposed by Isola et al.[6].

erally, our setup follows the approach by Lassner and Zollhöfer [11] that obtains an initial estimate of the point cloud by directly optimizing the point cloud with the differentiable renderer and afterward performs an end-to-end training using the differentiable renderer in a neural rendering model. However, a few simplifications to the model were necessary in order to be able to train it on the available hardware. The neural rendering model has three main components: First is the differentiable renderer, which is used to produce a feature map instead of an RGB image, since each point is assigned a 16-dimensional feature vector. To obtain the feature map, the point features are linearly interpolated based on the weight obtained by the point splatting without any additional shading. Next is the generator model that takes the feature map from the differentiable renderer as its input and predicts a view based on this input. In this simplified neural renderer, the generator design follows an U-Net architecture which has been adapted from Isola et al. [6]. The last major component of the model is the discriminator. The discriminator is only used during training to be able to compute the adversarial loss and can be omitted during inference. The discriminator is a conditional PatchGAN discriminator [6]. The PatchGAN discriminator uses only convolutions without any fully connected layers at the end which results in the discriminator returning a likelihood per pixel patch instead of a single value for the entire image. The network layout of the generator, discriminator and neural rendering model are illustrated in Figure 7. The model is trained

end-to-end using Adam [10] with a batch size of 8 to minimize the combined loss

$$L_{\text{model}} := L_{\text{adv}} + \lambda L_{\text{cnt}}$$

where L_{adv} is the adversarial loss component, $L_{\text{cnt}} = |G(f_c) - x|_1$ is the L_1 distance between the generator output $G(f_c)$ and the reference image x, and $\lambda = 100$ is a weighting term. The conditional adversarial loss uses the Least square formulation introduced by Mao et al. [13] and is defined to be the mean square error between the discriminator values computed on real and generated views and the nominal value for real or generated samples:

$$L_{adv} := \frac{1}{2P} \sum_{i=1} \mathbb{E}_{x \sim p_{\text{data}}(x)} (D(x, f_c)_i - 1)^2 + \mathbb{E}_{z \sim p_z(z)} (D(G(f_c), f_c)_i + 1)^2.$$

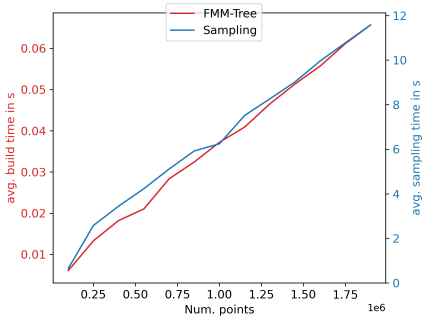
Note that the discriminator not only takes the real images x and generated images $G(f_c)$ but is conditioned with the generator prior f_c, which is the output of the differentiable renderer. The adversarial loss is the least-squares GAN objective averaged over the P patches. The pre-training is performed for 300 epochs and afterwards the point cloud parameter and neural network weights are jointly optimized for 2000 epochs.

IV Detailed Scaling

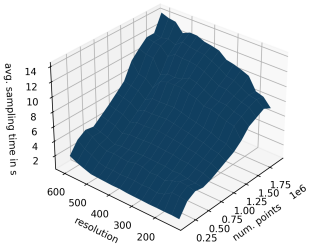
To more accurately assess the scaling of the sampling algorithm, the parameters are varied individually relative to a synthetic baseline instance. The baseline instance assumes that $N = 10^5$ points are uniformly distributed on a square image with a resolution of $i_w, i_h = 128$ pixels. The precision of the FMM approximation is $\epsilon = 0.01$ and the standard deviation is $h = 0.4$. The points are uniformly distributed, since points can only be combined in a Hermite expansion if they are close enough to each other. In an uniform distribution, the distance between points increases only slowly when increasing the number of points in the same sized screen space, which is a difficult instance for the algorithm. The graphs in Figure 8 report the runtime required by the tree construction and the sampling step in seconds for varying instance parameters. Note that these experiments were performed on a Nvidia GTX 760 and do not include the additional runtime added to the rendering steps by depth filtering or shading calculations. Consequently, the timings reported in Table 4 and the runtime values reported in the previous section are not directly comparable.

The plot of the runtime in Figure 8a, which plots an increasing number of points from 10^5 to 1.9×10^6 against the runtime, highlights the linear time complexity of both the construction algorithm as well as the sampling algorithm. This is consistent with the observations made by Karras [8] and Lee et al.[12] on which the tree construction is based.

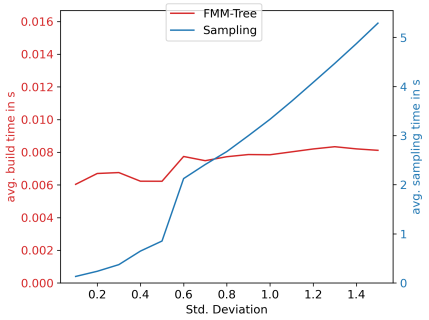
To further demonstrate that the sampling is not only linear with respect to the number of points but also scales linearly with regard to the point cloud size and resolution, we provide a 3D plot in which both variables are scaled



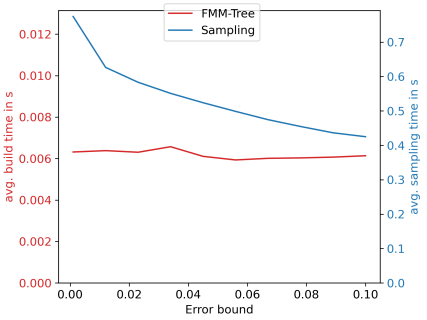
(a) Number of points



(b) Image resolution



(c) Isotropic std. deviation



(d) FMM precision

Fig. 8: Illustration of the scaling behaviour of the proposed sampling algorithm. The avg. runtime in all figures is reported in seconds. Note that Figures 8a, 8c, 8d have two y scales. The left y-scale plots the tree construction time whereas the right y-scale is for the runtime of the sampling step.

simultaneously in Figure 8b. Note that the construction time required to build the BVH-tree is by design independent from the image resolution and is therefore constant for a fixed number of points. The vertical and horizontal resolution is increased from 128 pixels to 608 pxiels in steps of 32 pixels. The resulting surface validates that the sampling scales linearly in both resolution and point cloud size i.e. the runtime is in $(N+i_w \cdot i_h)$ instead of $O(N \cdot i_w \cdot i_h)$. This can be quantified by fitting a linear model onto the data which returns a coefficient of determination of 0.957.

During the reconstruction and refinement experiments in Sections 4 (of the main paper) we observed the values of the isotropic standard deviation h to be between 0.25 and 1.5. In this range, the sampling algorithm scales linearly with respect to the standard deviation as is illustrated in the graph in Figure 8c. It can be assumed that the steep increase of the runtime from a standard deviation of 0.5 to 0.6 is due to the fact that for values smaller than 0.5 no points can be combined in a Hermite expansion. If the necessary point density and standard

deviation is reached, the constant term from the Hermite calculations increases the runtime before it amortizes.

Lastly, Figure 8d highlights that decreasing the approximation precision by increasing the error bound ϵ' is negatively correlated with the required runtime by the sampling algorithm. While the tree construction is mostly unaffected by a change to the error bound, the time required to compute the samples scales with the inverse logarithm of the error bound.

References

1. Bell, N., Hoberock, J.: Thrust: A productivity-oriented library for cuda. In: GPU computing gems Jade edition, pp. 359–371. Elsevier (2012)
2. Bode, L., Merzbach, S., Stotko, P., Weinmann, M., Klein, R.: Real-time multi-material reflectance reconstruction for large-scale scenes under uncontrolled illumination from rgb-d image sequences. In: 2019 International Conference on 3D Vision (3DV). pp. 709–718. IEEE (2019)
3. Deledalle, C.A., Denis, L., Tabti, S., Tupin, F.: Closed-form expressions of the eigen decomposition of 2 x 2 and 3 x 3 Hermitian matrices. Ph.D. thesis, Université de Lyon (2017)
4. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. Advances in neural information processing systems **27** (2014)
5. Greengard, L., Strain, J.: The fast gauss transform. SIAM Journal on Scientific and Statistical Computing **12**(1), 79–94 (1991)
6. Isola, P., Zhu, J.Y., Zhou, T., Efros, A.A.: Image-to-image translation with conditional adversarial networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 1125–1134 (2017)
7. Jenkins, B.: A hash function for hash table lookup, `http://www.burtleburtle.net/bob/hash/doobs.html`
8. Karras, T.: Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In: Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics. pp. 33–37 (2012)
9. Keller, M., Lefloch, D., Lambers, M., Izadi, S., Weyrich, T., Kolb, A.: Real-time 3d reconstruction in dynamic scenes using point-based fusion. In: 2013 International Conference on 3D Vision-3DV 2013. pp. 1–8. IEEE (2013)
10. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
11. Lassner, C., Zollhofer, M.: Pulsar: Efficient sphere-based neural rendering. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 1440–1449 (2021)
12. Lee, D., Moore, A.W., Gray, A.G.: Dual-tree fast gauss transforms. In: Advances in Neural Information Processing Systems. pp. 747–754 (2006)
13. Mao, X., Li, Q., Xie, H., Lau, R.Y., Wang, Z., Paul Smolley, S.: Least squares generative adversarial networks. In: Proceedings of the IEEE international conference on computer vision. pp. 2794–2802 (2017)
14. Mildenhall, B., Srinivasan, P.P., Tancik, M., Barron, J.T., Ramamoorthi, R., Ng, R.: Nerf: Representing scenes as neural radiance fields for view synthesis. In: European conference on computer vision. pp. 405–421. Springer (2020)

15. Ohlsson, E.: Sequential poisson sampling. Journal of official Statistics **14**(2), 149 (1998)
16. Ramamoorthi, R., Hanrahan, P.: An efficient representation for irradiance environment maps. In: Proceedings of the 28th annual conference on Computer graphics and interactive techniques. pp. 497–500 (2001)
17. Rusu, R.B., Cousins, S.: 3d is here: Point cloud library (pcl). In: 2011 IEEE international conference on robotics and automation. pp. 1–4. IEEE (2011)
18. Steinbrücker, F., Sturm, J., Cremers, D.: Real-time visual odometry from dense rgb-d images. In: 2011 IEEE international conference on computer vision workshops (ICCV Workshops). pp. 719–722. IEEE (2011)
19. Wang, T.C., Liu, M.Y., Zhu, J.Y., Tao, A., Kautz, J., Catanzaro, B.: High-resolution image synthesis and semantic manipulation with conditional gans. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 8798–8807 (2018)
20. Yifan, W., Serena, F., Wu, S., Öztireli, C., Sorkine-Hornung, O.: Differentiable surface splatting for point-based geometry processing. ACM Transactions on Graphics (TOG) **38**(6), 1–14 (2019)
21. Yifan, W., Serena, F., Wu, S., Öztireli, C., Sorkine-Hornung, O.: Github - yifita/dss: Differentiable surface splatting (2019), `https://github.com/yifita/DSS/tree/44732f9b771ca7e5ee4cfebeaf8528be1d097e3e`