Pointersect: Neural Rendering with Cloud-Ray Intersection

Jen-Hao Rick Chang[1], Wei-Yu Chen[1,2*], Anurag Ranjan[1], Kwang Moo Yi[1,3*], Oncel Tuzel[1]
[1]Apple, [2]Carnegie Mellon University, [3]University of British Columbia
https://machinelearning.apple.com/research/pointersect

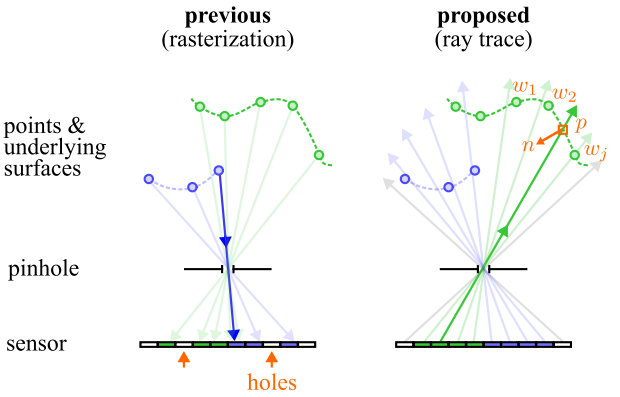
Abstract

We propose a novel method that renders point clouds as if they are surfaces. The proposed method is differentiable and requires no scene-specific optimization. This unique capability enables, out-of-the-box, surface normal estimation, rendering room-scale point clouds, inverse rendering, and ray tracing with global illumination. Unlike existing work that focuses on converting point clouds to other representations—e.g., surfaces or implicit functions—our key idea is to directly infer the intersection of a light ray with the underlying surface represented by the given point cloud. Specifically, we train a set transformer that, given a small number of local neighbor points along a light ray, provides the intersection point, the surface normal, and the material blending weights, which are used to render the outcome of this light ray. Localizing the problem into small neighborhoods enables us to train a model with only 48 meshes and apply it to unseen point clouds. Our model achieves higher estimation accuracy than state-of-the-art surface reconstruction and point-cloud rendering methods on three test sets. When applied to room-scale point clouds, without any scene-specific optimization, the model achieves competitive quality with the state-of-the-art novel-view rendering methods. Moreover, we demonstrate ability to render and manipulate Lidar-scanned point clouds such as lighting control and object insertion.

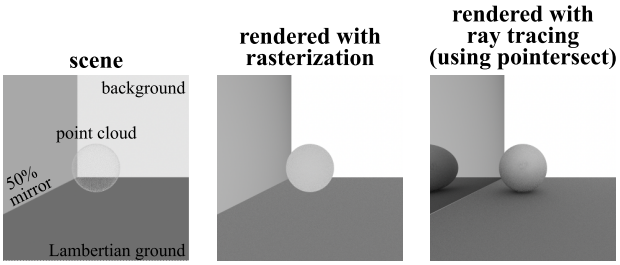
1. Introduction

Point clouds are abundant. They are samples of surfaces, and can be captured by sensors such as Lidar, continuous-wave time-of-flight, and stereo camera setups. Point-cloud representation provides a straightforward connection to the location of the surfaces in space, and thus is an intuitive primitive to represent geometry [15].

Despite being ubiquitous, a core limitation of point clouds is that they are non-trivial to render. Each point in the point cloud occupies no volume—one cannot render them into images as is. Therefore, existing methods either assign each point a volume-occupying shape, *e.g.*, an oriented disk [36, 53], a sphere [28], or turn it into other shape representations like meshes [27] or implicit functions [14, 18, 22, 30, 35]. However, it is difficult to determine the ideal shape, *e.g.*, the radius of spheres or disks, for rendering. Small shapes would cause holes, while large shapes would cause blobby renderings, producing artifacts in the rasterized images. While the artifacts can be alleviated



(a) Illustration of the proposed method



(b) Rendered results of a sphere represented by points

Figure 1. We propose *pointersect*, a novel method to perform cloud-ray intersection. (a) Instead of projecting points onto the sensor, suffering from holes, we trace rays from the sensor and estimate the intersection point p between a ray and the underlying surface represented by the points. We additionally estimate the surface normal $\vec{n}$, and the convex combination weights of points near the ray to blend material or color, w_j. (b) The capability to perform cloud-ray intersection enables us to render point clouds with the standard ray tracing method, *i.e.*, path tracing. The result shows the effect of global illumination, *e.g.*, the cast shadow and reflection.

*Work done at Apple. Corresponding author: jenhao_chang@apple.com

by finetuning the rasterized images with an additional neural rendering step, the operation often requires per-scene training [3, 9, 28]. On the other hand, transforming point clouds into other shape representations complicates the pipeline and prevents gradients passing back to the point cloud through the new shape representation (in the case of inverse rendering). For example, turning point clouds into a mesh or a Signed Distance Function (SDF) [18, 30] would require any changes on the point cloud to trigger retraining of these representations, which would clearly be prohibitive.

Recent works [34, 51] raise new ideas to directly perform ray-casting on point clouds. For each scene, these methods first learn a feature embedding for each point; then they aggregate features near each camera ray to predict colors. However, these methods require per-scene training since the feature embedding is scene-dependent. In our case, we aim for a solution that does not require scene-specific optimization and can be applied to any scene.

In this work, we propose *pointersect*, an alternative that can directly *ray-trace* point clouds by allowing one to use point clouds as surface primitives, as shown in Figure 1. That is, we propose to train a neural network that provides the surface intersection point, surface normal, and material blending weights—the necessary information to render (or ray trace) a surface—given a point cloud and a query ray.

Implementing this idea requires paying attention to details. A core observation is that the problem to find the intersecting surface point is SE(3) equivariant—any rigid transform on the input (*i.e.*, the point cloud and the query ray) should result in the rigid transformation of the output (*i.e.*, the intersection point and the surface normal). Naively training a neural network would require the network to learn this equivariance, which is non-trivial [5, 13, 45]. Instead, we opt to remove the need for learning this equivariance by canonicalizing the input according to the queried light ray. In addition, pointersect should be invariant to the order in which the points are provided—we thus utilize a transformer to learn the set function.

It is important to note that finding intersection points between rays and surfaces is a highly atomic and localized problem which can be solved only with local information. Thus, we design our method to only consider nearby points, where the surface would have been, and how the surface texture and normal can be derived from these nearby points. By constraining the input to be a small number ($\sim$100) of neighboring points associated to a query ray, our method can be trained on only a handful of meshes, then be applied to unseen point clouds. As our experiments show, while only trained on 48 meshes, pointersect significantly improves the Poisson surface reconstruction, a scene-specific optimization method, on three test datasets. We also demonstrate the generality and differentiability of pointersect on various applications: novel-view synthesis on room-scale point clouds, inverse rendering, and ray tracing with global illumination. Finally, we render room-scale Lidar-scanned point clouds and showcase the capability to directly render edited scenes, without any scene-specific optimization.

In short, our contributions are:

- We propose *pointersect*, a neural network performing the *cloud-ray intersection* operation. Pointersect is easy to train, and once learned, can be applied to unseen point clouds—we evaluate the same model on three test datasets.
- We demonstrate various applications with pointersect, including room-scale point cloud rendering, relighting, inverse rendering, and ray tracing.
- We apply pointersect on Lidar-scanned point clouds and demonstrate novel-view synthesis and scene editing.

We encourage the readers to examine results and videos of novel-view rendering, relighting, and scene editing in the supplemental material and website (https://machinelearning.apple.com/research/pointersect).

2. Related work

This section briefly summarizes strategies to render point clouds. Please see Table 4 in the supplementary for a detailed overview on the capabilities/limitations of each method.

Rasterizing point clouds. Rasterization is a common method to render point clouds. The idea is to project each point onto the sensor while making sure closer points occlude farther points. Filling holes between points is the key problem in point-cloud rasterization. Classical methods like visibility splatting [36, 53] cover holes by replacing points with oriented disks. However, since the size and shape of the holes between projected points depend on the distribution of the points in space, the method cannot fill in all gaps. Recently, Zhang et al. [54] achieve high-quality results via alpha-blending surface splatting [59], coarse-to-fine optimization, and point insertion. However, their method requires per-scene optimization on the point cloud.

Recent works propose to combine rasterization with neural networks. Given a point cloud and RGB images of a scene, Aliev et al. [3] and Rückert et al. [42] learn an embedding for each point by rasterizing feature maps and minimizing the difference between the rendered images and the given ones. The hole is handled by downsampling then upsampling the rasterized feature map. Similarly, Dai et al. [9] aggregate points into multi-plane images, combined with a 3D-convolutional network. Huang et al. [19] use a U-Net refinement network to fill in the holes. Recently, Rakhimov et al. [39] show that point embedding can be directly extracted from input images, which enables their method to render unseen point clouds without per-scene optimization.

It is difficult to render global illumination with these methods, since rasterization does not support such an operation. Moreover, we show empirically in Section 4 that our method

renders higher quality images than recent methods while directly working on xyz and rgb, without a feature extractor.

Converting into other representations. An alternative way to render a point cloud is converting it to other primitives, such as an indicator function [22, 35], a SDF [30], or a mesh [18]. Poisson surface reconstruction [22, 23] fits an indicator function (*i.e.*, 1 inside the object and 0 otherwise) to the input point cloud by solving a Poisson optimization problem. Meshes can then be extracted from the learned function, allowing ray tracing to be performed. However, Possion surface reconstruction requires vertex normal, which can be difficult to estimate even when the points only contain a small amount of noise, and it is difficult to support non-watertight objects. Peng et al. [35] learn the indicator function with a differentiable solver and incorporate Poisson optimization in a neural network.

Ma et al. [30] learn a SDF represented as a neural field, and Hanocka et al. [18] fit a deformable mesh to an objects with a self-prior. These methods require per-scene optimization, and to the best of our knowledge, have not been extended to render surface colors.

Alternatively, Feng et al. [14] propose a new primitive, Neural Point, or a collection of local neural surfaces extracted from the point cloud. The method supports new scenes and estimates surface normal; however, dense sampling or cube marching is needed to render novel views, and the method does not render color.

Ray-casting. Instead of rasterization, point clouds can also be rendered by ray-casting. Early work [1, 2, 4, 16, 26, 48] develop iterative algorithms or formulate optimization programs to intersect a ray with the approximated local plane constructed by nearby points. These methods rely on neighborhood kernels, which is non-trivial to determine [15, 25]. Recently, Xu et al. [51] and Ost et al. [34] learn feature embedding at each point and aggregate the features along a query ray. Xu et al. [51] march camera rays through the point cloud, average neighbor features, predict density and color by a Multi-Layer Perceptron (MLP), and render the final color via volumetric rendering [32]. Ost et al. [34] utilize a transformer to aggregate point features into a feature associated with the query ray and predict color by an MLP. Both methods do not estimate surface intersection points or normal, and they need per-scene training [34] or per-scene fine-tuning [51] to achieve high-quality results. As we show later, our method can be applied to completely novel classes of scenes without any retraining.

Neural rendering from 2D images. Recently, Neural Radiance Field (NeRF) and similar methods [11, 12, 28, 29, 32, 40, 44, 49, 52, 56] have demonstrated high-quality novel-view synthesis results. Due to the lack of immediately available 3D information, most of these methods require a per-scene optimization or surface reconstruction. In this work, we focus on rendering a point cloud, where 3D information is available, without additional per-scene training.

3. Method

We aim to directly perform ray casting with point clouds. We thus first introduce the generic surface-ray intersection. We then introduce how we enable cloud-ray intersection and discuss how it can be used for actual rendering.

3.1. Problem formulation

Surface-ray intersection. Surface-ray intersection is a building-block operator in physics-based rendering [37]. At a high level, it identifies the contacting point between a query ray and the scene geometry so that key information like material properties, surface normal, and incoming light at the point can be retrieved and computed [37]. Most graphics primitives allow the intersecting point to be easily found. For example, with triangular meshes we simply intersect the query ray with individual triangles (while using accelerated structures to reduce the number of triangles of interest). However, for point clouds, finding the intersection point becomes a challenging problem.

Cloud-ray intersection. We formulate the cloud-ray intersection as follows. We are given the following information:

- a set of points $\mathcal{P} = \{p_1, \dots, p_n\}$ that are samples on a surface $\mathcal{S}$, where $p_i \in \mathbb{R}^3$ is i-th point's coordinate;
- optionally, the material (*e.g.*, RGB color or Cook-Torrance coefficients [8]) associated with each point, $c_i \in \mathbb{R}^d$;
- a querying ray $\mathbf{r} = (r_o, \vec{r_d})$, where $r_o \in \mathbb{R}^3$ is the ray origin and $\vec{r_d} \in \mathbb{S}^2$ is the ray direction.

Our goal is to estimate the following quantities:

- the probability, $h \in [0, 1]$, that $\mathbf{r}$ intersects with the underlying surfaces of $\mathcal{P}$;
- the intersection point $p \in \mathbb{R}^3$ between $\mathbf{r}$ and $\mathcal{S}$;
- the surface normal $\vec{n}$ at p;
- and optionally the blending weights $\mathbf{w} = [w_1, \dots, w_n]$, where $w_i \in [0, 1]$ and $\sum_{i=1}^{n} w_i = 1$, to estimate the material property at p: $c(p) = \sum_{i=1}^{n} w_i \, c_i$.

Note that this problem is under-determined—there can be infinitely many surfaces passing through points in $\mathcal{P}$. Thereby, hand-crafting constraints such as surface smoothness are typically utilized to solve the problem [25]. In this work, we learn surface priors from a dataset of common objects using a neural network, so manual design is not needed.

3.2. Pointersect

Pointersect is a neural network, $f_\theta(\mathbf{r}, \mathcal{P}) \mapsto (h, p, \vec{n}, \mathbf{w})$, that estimates the intersection point and surface normal between $\mathbf{r}$ and the underlying surface of $\mathcal{P}$. We learn the network by formulating a regression problem, using a dataset

of meshes. We generate training data by randomly sampling rays and point clouds on the meshes, *i.e.*, our inputs, and running a mesh-ray intersection algorithm [37] to acquire the ground-truth outputs.

Despite the simplicity of the framework, we note, however, care must be taken when designing f_θ. Specifically, we incorporate the following geometric properties into the design of f_θ to ease training and allow generalization:

First, we utilize the fact that the intersection point is along the query ray and design f to estimate the distance t from the ray origin, *i.e.*, $p = r_o + t\,\vec{r_d}$.

Second, we utilize the SE(3) equivariance—rotating and translating the ray and the points together should move the intersection point in the same way. Mathematically, for all rotation matrix $R \in \mathrm{SO}(3)$ and translation $b \in \mathbb{R}^3$, we have

$$f_\theta(\mathbf{r}, \mathcal{P}) \mapsto (h, t, \vec{n}, \mathbf{w}) \Rightarrow f_\theta(T(\mathbf{r}), T(\mathcal{P})) \mapsto (h, t, R\vec{n}, \mathbf{w}),$$

where $T(\mathcal{P}) = \{R\,p_i + b\}_{i=1\ldots n}$ and $T(\mathbf{r}) = (T(r_o), R\,\vec{r_d})$. Additionally, this holds for any point along the ray, thus

$$f_\theta(\mathbf{r}, \mathcal{P}) \mapsto (h, t, \vec{n}, \mathbf{w}) \Rightarrow f_\theta((r_o + t'\vec{r_d}, \vec{r_d}), \mathcal{P}) \mapsto (h, t - t', \vec{n}, \mathbf{w}),$$

for all $0 \le t' \le t$. Thereby, to eliminate this ambiguity, given a query ray $\mathbf{r} = (r_o, \vec{r_d})$, we rotate and translate both the ray and the scene such that $R\,\vec{r_d} = (0, 0, 1)$ and the closest point in the half space defined by $\mathbf{r}$ has $z = 0$.

Third, the intersection point can be estimated by using the local neighborhood of the ray. Given $\mathbf{r}$ and $\mathcal{P}$, we form a new set of points, $\mathcal{P}_\mathbf{r}$, by keeping only the closest k points (in terms of their orthogonal distances to $\mathbf{r}$) within a cylinder of radius δ surrounding the ray.

Last, since $\mathcal{P}$ is a set, *i.e.*, the order of the points in $\mathcal{P}$ is irrelevant, we use a set transformer [47] as our architecture of choice. See Figure 2 for an overview and Appendix C for detailed descriptions.

We train f_θ by optimizing the following problem:

$$\min_\theta \mathop{\mathbb{E}}_{\mathbf{r},\mathcal{P}} \hat{h}\left(\lambda\|t - \hat{t}\|_2^2 + \|\vec{n} \times \hat{\vec{n}}\|_2^2 + \|c - \hat{c}\|_1\right) + \hat{h}\log h + (1 - \hat{h})\log(1 - h), \quad (1)$$

where $\hat{t}$, $\hat{\vec{n}}$, $\hat{c}$, and $\hat{h} \in \{0, 1\}$ are the ground-truth ray traveling distance, surface normal, color, and ray hit, respectively, and $c = \sum w_i\,c_i$ is the output color. The expectation is over the query rays and the point clouds, which we sample randomly every iteration from a dataset of meshes. We omit the dependency on $\mathcal{P}$ and $\mathbf{r}$ in the notations for simplicity.

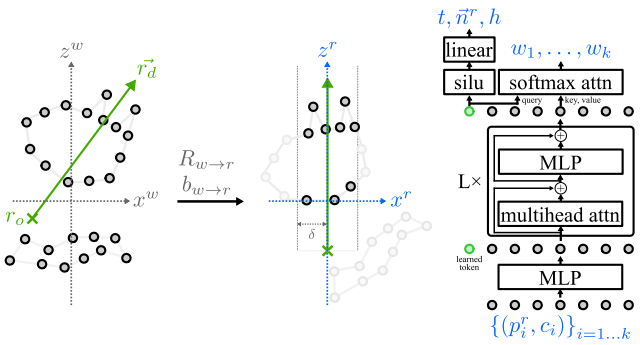
3.3. Rendering point clouds with pointersect

Our ability to perform cloud-ray intersection allows two main techniques to render a point cloud.

Image-based rendering. Given a point cloud $\mathcal{P} = \{(p_i, c_i)\}_{i=1\ldots n}$, where each point has both position $p_i \in \mathbb{R}^3$ and RGB color $c_i \in \mathbb{R}^3$, and the target camera intrinsic and extrinsic matrices, we can simply cast camera rays toward $\mathcal{P}$. The final color of a camera ray, $c(\mathbf{r})$ can be computed by

$$c(\mathbf{r}) = \sum_{i=1}^{n} w_i(\mathbf{r}, \mathcal{P})\,c_i, \quad (2)$$

where $w_i(\mathbf{r}, \mathcal{P})$ is the blending weight of p_i estimated by f_θ, and $w_i(\mathbf{r}, \mathcal{P}) = 0$ if p_i is not a neighbor point of $\mathbf{r}$.



(a) Pre- rotation and translation (b) Architecture

Figure 2. The pointersect model f is composed of a pre-processing step and a transformer. (a) The pre-processing step rotates and translates the world coordinate such that the ray lies on the z^r-axis. The new coordinate origin is selected such that the closest point to the ray origin lies on the $z^r = 0$ plane. (b) The transformer takes as inputs the points in the new coordinate, *i.e.*, (p_i^r, c_i). The output of a learned token is used to estimate t, the traveling distance between the ray origin and the intersection point, $\vec{n}^r$, the surface normal in the new coordinate, and h, the output of a sigmoid function on a logit (not drawn) to predict the probability that the ray hits a surface. It is also used as the query in the softmax attention to calculate the weight $\mathbf{w}$. Finally, we transform $\vec{n}^r$ back to the world coordinate and make sure it point to the opposite direction of the ray. The transformer has 4 layers ($L = 4$) and a feature dimension of 64.

Rendering with ray tracing. A unique capability of pointersect is ray tracing. Ray tracing allows occlusion and global illumination effects like cast shadow and specular reflection to be faithfully rendered. Suppose we are given a point cloud $\mathcal{P} = \{(p_i, c_i)\}_{i=1\ldots n}$, where each point has both position $p_i \in \mathbb{R}^3$ and material information $c_i \in \mathbb{R}^d$ like albedo, Cook-Torrance [8], or emission coefficients, and we have the environment map and the target camera intrinsic and extrinsic matrices. We can use standard ray tracing techniques like path tracing [20] to render the image. At a high level, we trace a ray through multiple intersections with the point cloud until reaching the environment map or background. At each intersection point, we calculate the material property by interpolating the material of neighboring points (using the blending equation (2)), shade the intersection point based on the reflection equation [20], and determine the direction to continue tracing [43].

When we end the ray tracing with a single bounce, *i.e.*,

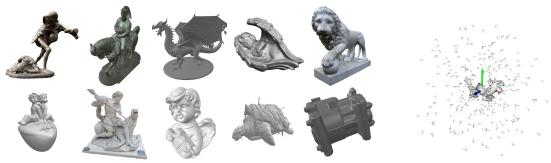


Figure 3. Training meshes (left) and example camera poses (right).

using pointersect to determine the first intersection point with the scene, and directly shade the intersection point with the environment, we get an equivalent algorithm of rasterization with deferred shading for meshes [10].

4. Experiments

We evaluate the proposed method's capability to estimate intersection points and surface normal. We then showcase the use of pointersect in various applications, including rendering point clouds with image-based rendering and ray-tracing, and inverse rendering. Note that the same model is used for all experiments—no per-scene optimization is used.

4.1. Model training

The model is trained on 48 training meshes in the sketchfab dataset [38].We show 10 training meshes in Figure 3 and all 48 training meshes and their download links and credits in Figure 17 in Appendix J. The meshes are centered and scaled such that the longest side of their bounding box is 2 units. For each training iteration, we randomly select one mesh and construct 30 input cameras and 1 target camera, which capture RGBD images using the mesh-ray intersection method in Open3D [58] without anti-aliasing filtering. We design the input and target camera poses to be those likely to appear in novel-view synthesis. Specifically, we uniformly sample camera position within a spherical shell of radius 0.5 to 3, looking at a random position in the unit-length cube containing the object, as shown in Figure 3. During testing, the camera poses are non-overlapping with the training poses — the poses used in Table 1, shown in Figure 4, have a spiral trajectory with radius changing periodically between 3 to 4 and those used in room scenes (Figure 6 and 7) are chosen to follow the room layouts.

The input RGBD images create the input point cloud, where each point carries only point-wise information, including xyz, rgb, and the direction from input camera to the particular point. To support point clouds without these information, we randomly drop rgb and other features independently 50 % of the time—during inference, we use only xyz and rgb for all experiments. We also use a random $k \in [12, 200]$ at every iteration. To help learning blending weights, at every iteration we select a random image patch from ImageNet dataset as the texture map for the mesh. We train the model for 350,000 iterations, and it takes 10 days on 8 A100 GPUs. Please see more details in Appendix E.

4.2. Evaluating pointersect

We use three datasets to evaluate the estimated intersection points, surface normal, and blending weights.

- 7 meshes provided by Zhou et al. [57], including the Stanford Bunny, Buddha, *etc*.
- 30 meshes in ShapeNet Core dataset [7] containing sharp edges, including chairs, rifles, and airplanes.
- 13 test meshes in the sketchfab dataset [38].

For each mesh, 6 RGBD images taken from front, back, left, right, top, and bottom are used to create the input point cloud; 144 output RGBD images at novel viewpoints are estimated and compared with the ground-truth rendering from Open3D. See Figure 4 for an illustration of the camera poses. All cameras are 200×200 resolution and has a field-of-view of 30 degrees. The input RGBD images are used only to construct the input point cloud, *i.e*., we do not extract any image-patch features from the RGB images (except for NPBG++, see below).

We compare with four baselines:

- *Visibility splatting* is the default point-cloud visualization method in Open3D. We set the point size to 1 pixel. We use Open3D to estimate vertex normal at input points. The main purpose of including the baseline is to illustrate the input point cloud from the target viewpoint.
- *Screened Poisson surface reconstruction* [23] fits a scene-specific indicator function (1 inside the object and 0 outside) to reconstruct a surface from the point cloud. It is the workhorse for point-based graphics. We use the implementation in Open3D with the default parameters, and we provide ground-truth vertex normal from the mesh.
- *NPBG++* [39] uses rasterization and downsampling to fill in holes on the image plane. It supports unseen point clouds but requires input RGB images to extract features. We use the implementation and pretrained model from the authors. We do not perform scene-specific finetuning.
- *Neural Points* [14] fits local implicit functions on the point clouds. Once the functions are fit, they can be used to increase the sampling rate and estimates surface normals. We use the original implementation and pretrained model from the authors. For visualization, we use the method to upsample the point clouds by 96 times.
- *Oracle* directly rasterizes the mesh.

For pointersect, we use $k = 40$, $\delta = 0.1$, and image-based rendering for all experiments unless otherwise noted. We provide only point-wise xyz and rgb to pointersect; no other feature is used. For all methods, we compute the Root Mean Square Error (RMSE) of the intersection point estimation, the average angle between the ground-truth and the estimated normal, the accuracy on whether a camera ray hit the surface, the Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index Measure (SSIM) [50], and LPIPS [55] between

Table 1. Test results on unseen meshes in three datasets.

Method	Metrics	tex-models	ShapeNet	Sketchfab
Visibility splatting	depth (RMSE) ↓	0.25 ± 0.20	0.08 ± 0.08	0.20 ± 0.12
	normal (angle (°)) ↓	12.57 ± 4.60	11.78 ± 5.40	14.04 ± 4.73
	hit (accuracy (%)) ↑	98.0 ± 1.5	98.5 ± 1.7	98.0 ± 2.0
	color (PSNR (dB))↑	19.2 ± 2.2	22.5 ± 4.1	19.6 ± 2.9
	color (SSIM)↑	0.8 ± 0.2	0.9 ± 0.1	0.7 ± 0.2
	color (LPIPS)↓	0.19 ± 0.13	0.10 ± 0.10	0.21 ± 0.13
Poisson surface recon.	depth (RMSE) ↓	$\mathbf{0.02 \pm 0.04}$	$\mathbf{0.03 \pm 0.05}$	0.06 ± 0.08
	normal (angle (°)) ↓	8.48 ± 3.97	16.89 ± 7.96	14.58 ± 6.29
	hit (accuracy (%)) ↑	$\mathbf{99.8 \pm 0.1}$	98.8 ± 1.8	99.5 ± 0.5
	color (PSNR (dB))↑	25.7 ± 2.4	-	25.0 ± 3.1
	color (SSIM)↑	0.9 ± 0.0	0.9 ± 0.1	0.9 ± 0.1
	color (LPIPS)↓	0.08 ± 0.04	0.06 ± 0.05	0.10 ± 0.04
Neural points [14]	depth (RMSE) ↓	0.07 ± 0.10	0.04 ± 0.03	0.06 ± 0.05
	normal (angle (°)) ↓	11.28 ± 3.30	17.14 ± 6.01	14.28 ± 3.44
	hit (accuracy (%)) ↑	98.9 ± 0.4	98.9 ± 0.7	99.1 ± 0.2
	color (PSNR (dB))↑	not supp.	not supp.	not supp.
	color (SSIM)↑	not supp.	not supp.	not supp.
	color (LPIPS)↓	not supp.	not supp.	not supp.
NPBG++ [39]	depth (RMSE) ↓	not supp.	not supp.	not supp.
	normal (angle (°)) ↓	not supp.	not supp.	not supp.
	hit (accuracy (%)) ↑	not supp.	not supp.	not supp.
	color (PSNR (dB))↑	16.5 ± 2.1	19.3 ± 4.0	18.0 ± 1.6
	color (SSIM)↑	0.7 ± 0.1	0.8 ± 0.1	0.7 ± 0.1
	color (LPIPS)↓	0.27 ± 0.07	0.18 ± 0.08	0.24 ± 0.08
Proposed	depth (RMSE) ↓	0.05 ± 0.09	$\mathbf{0.03 \pm 0.03}$	$\mathbf{0.05 \pm 0.04}$
	normal (angle (°)) ↓	$\mathbf{6.77 \pm 2.71}$	$\mathbf{11.29 \pm 5.08}$	$\mathbf{8.53 \pm 2.79}$
	hit (accuracy (%)) ↑	$\mathbf{99.8 \pm 0.2}$	$\mathbf{99.6 \pm 0.6}$	$\mathbf{99.8 \pm 0.1}$
	color (PSNR (dB))↑	$\mathbf{28.2 \pm 1.9}$	$\mathbf{28.0 \pm 6.4}$	$\mathbf{28.1 \pm 2.7}$
	color (SSIM)↑	$\mathbf{1.0 \pm 0.0}$	$\mathbf{1.0 \pm 0.0}$	$\mathbf{0.9 \pm 0.0}$
	color (LPIPS)↓	$\mathbf{0.04 \pm 0.03}$	$\mathbf{0.04 \pm 0.04}$	$\mathbf{0.06 \pm 0.04}$

Table 2. Test results on the Hypersim dataset. The test is conducted on 10 new images not used to create the point cloud.

Metric	Vis. splatting	Poisson recon.	NPBG++	NGP [33]	Proposed
PSNR (dB) ↑	11.7 ± 1.1	26.3 ± 3.1	27.9 ± 3.7	28.5 ± 5.8	$\mathbf{29.8 \pm 5.1}$
SSIM ↑	0.12 ± 0.05	0.85 ± 0.06	0.89 ± 0.04	0.90 ± 0.06	$\mathbf{0.91 \pm 0.04}$
LPIPS ↓	0.79 ± 0.04	0.37 ± 0.04	0.33 ± 0.03	0.31 ± 0.08	$\mathbf{0.25 \pm 0.04}$

ground-truth and estimated color images. To ensure fair comparisons between all methods, we compute the errors of surface normal and depth map only for camera rays that both ground-truth and the testing method agree to hit a surface.

The results are shown in Table 1, and we provide examples in Figure 4. Optimizing each input point cloud directly, Poisson surface reconstruction achieves high PSNR and low normal errors, outperforming the scene-agnostic baselines like NPBG++ and Neural Points. Pointersect outperforms all prior methods even though it is scene-agnostic. Trained on the sketchfab dataset, it performs best on the sketchfab dataset. Nevertheless, it still outperforms the scene-specific Poisson reconstruction on unseen meshes in the ShapeNet dataset and performs comparably on the tex-models dataset. Moreover, pointersect is the only method supporting estimating intersection points, surface normal, and blending weights of any query ray without scene-specific optimization.

4.3. Room-scale rendering with RGBD images

Next, we evaluate pointersect on a room-scale scene, whose geometry is very different from the training meshes. We randomly select a room scene in the Hypersim dataset [41] which contains 100 RGBD images captured in the room. We randomly select 90 of them to construct the input point cloud and use the rest for evaluation. We downsample the images by 2 (from 1024×768 to 512×384), and we use uniform voxel downsampling to reduce the number of points (the room size is 4 units and the voxel size is 0.01 units). We provide the ground-truth vertex normal from the dataset to Poisson reconstruction. We also train a state-of-the-art NeRF method, NGP [33, 46], using the same 90 input images, for 1000 epochs, taking 2 hours on 1 A100 GPU. Note that NGP and pointersect are not directly comparable—NGP utilizes scene-specific optimization whereas pointersect utilizes depth. We provide it as a reference baseline.

Figure 5, Figure 6, and Table 2 show the results on the 10 test RGBD images. As can be seen, while pointersect is trained on small meshes, it generalizes to room-scale scenes and outperforms prior state-of-the-art baselines.

4.4. Ray tracing with point clouds

As mentioned before in Section 3, pointersect provides a unique capability of ray-tracing with global illumination that is difficult to achieve with methods like NeRFs. We build a simple path tracer in PyTorch, following [43]. We also implement the Cook-Torrance microfacet specular shading model (with the split sum approximation), following [21]. We construct a scene composed of a Lambertian floor (albedo = 0.2), a vertical mirror (kbase = 0.5, roughness = 0, metallic = 1, specular = 0.5), a sphere (kbase = 0.7, roughness = 0.7, metallic = 0.5, specular = 0), and an all-white environment map. The floor and the mirror are represented analytically, and the sphere is represented with 5000 points. We trace 4 bounces and 2000 rays per pixel.

As shown in Figure 1b, the result clearly shows the effect of global illumination, *e.g.*, the reflection of the sphere in the mirror and the cast shadow on the floor.

4.5. Inverse rendering

Another unique capability of pointersect is its differentiability. As a neural net, pointersect allows gradient calculation of color (blending weights) and surface normal with respect to the point cloud (*e.g.*, xyz and rgb). We optimize a noisy point cloud's xyz and rgb given 100 clean input RGB images, camera poses, and binary foreground segmentation maps. The results (Figure 12 and Table 3 in the supplementary) show that the optimization effectively denoises the point cloud. Please refer to Appendix F for details.

4.6. Real Lidar-scanned point clouds

Finally, we test pointersect on real Lidar-scanned point clouds. The goal is to evaluate how it handles a small amount of noise in scanned point clouds, even though it is trained on

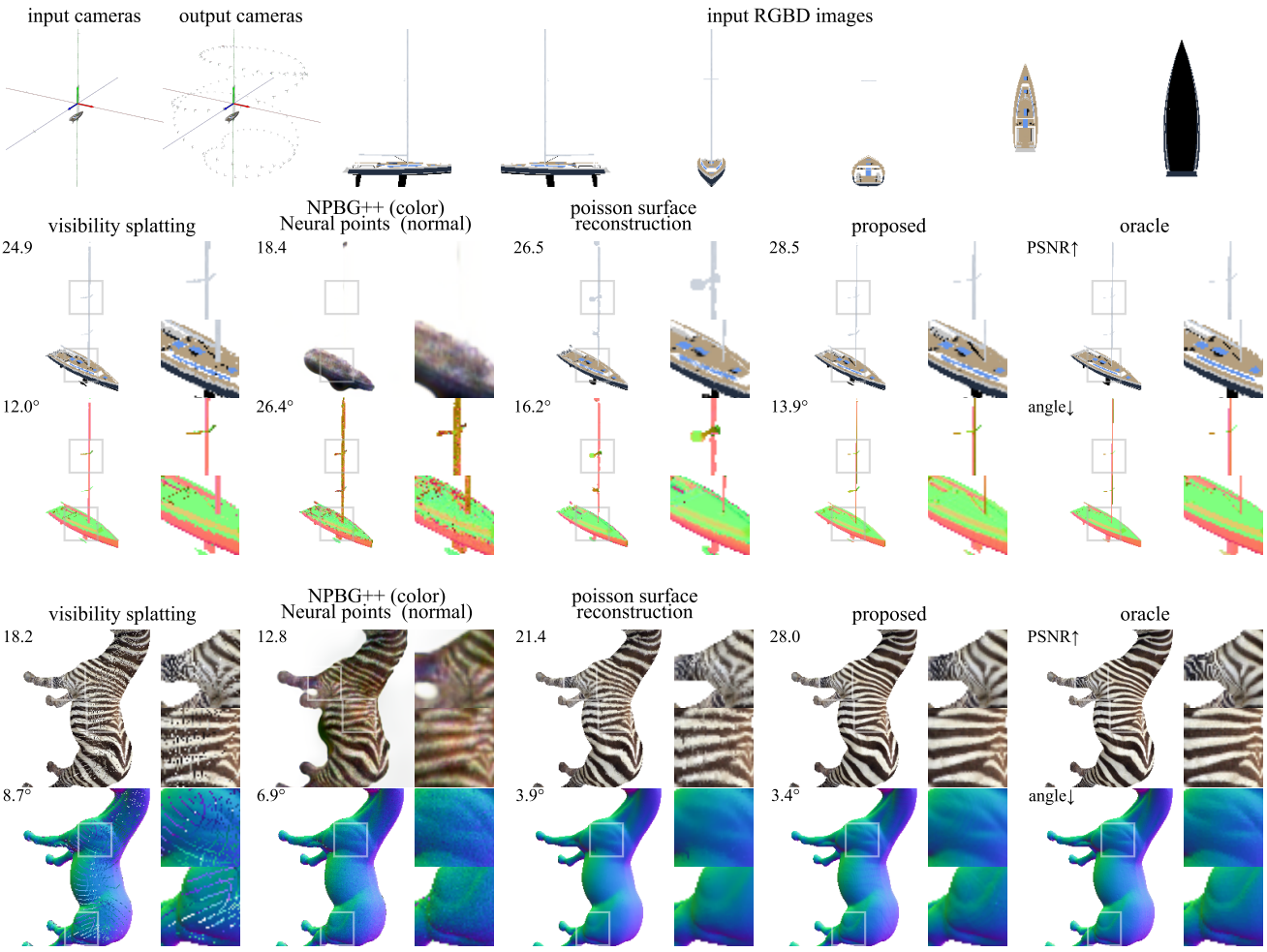


Figure 4. Example results of pointersect and baselines. Please see supplementary material for novel-view rendering videos.

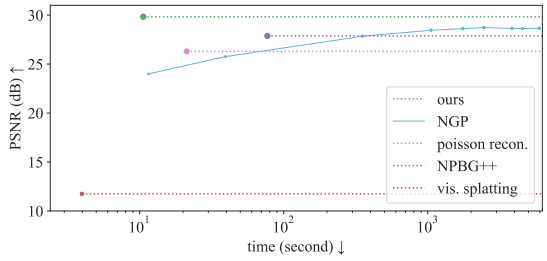


Figure 5. PSNR *vs*. time of Figure 6. The x axis is the time to render 10 validation images of resolution 512×384. We train the NGP [33, 46] from 1 to 1000 epochs, and the time includes both training on the 90 input images and the rendering.

Table 3. Optimize a noisy point cloud with pointersect.

	100 input views		144 novel views	
	before opt.	after opt.	before opt.	after opt.
PSNR (dB) ↑	10.1 ± 0.5	24.6 ± 0.5	13.9 ± 0.8	25.7 ± 1.2
normal (angle (°)) ↓	54.0 ± 0.9	17.8 ± 2.1	55.3 ± 0.4	18.3 ± 2.7
depth (rmse) ↓	0.46 ± 0.08	0.09 ± 0.06	0.33 ± 0.08	0.10 ± 0.07

clean ones. We take two Lidar point clouds from the ARKitScenes dataset [6], perform uniform voxel downsampling to reduce the number of points (the voxel size is 0.01 and the scene size is around 14 units), and rendered with Poisson reconstruction and pointersect (with $k = 100$ and $\delta = 0.2$ since the scene is larger and not unit-length normalized). As can be seen in Figure 7, our model successfully renders the real point clouds, whereas the output quality of Poisson reconstruction degrades significantly.

Scene editing. One advantage of utilizing point clouds as the scene representation instead of an implicit representation like NeRF is that we can easily edit the scene (by directly moving, adding, removing points). In Figure 7, we present results of scene relighting which utilizes the estimated surface normal and scene editing where we insert new and change the size and location of point-cloud objects in the scene.

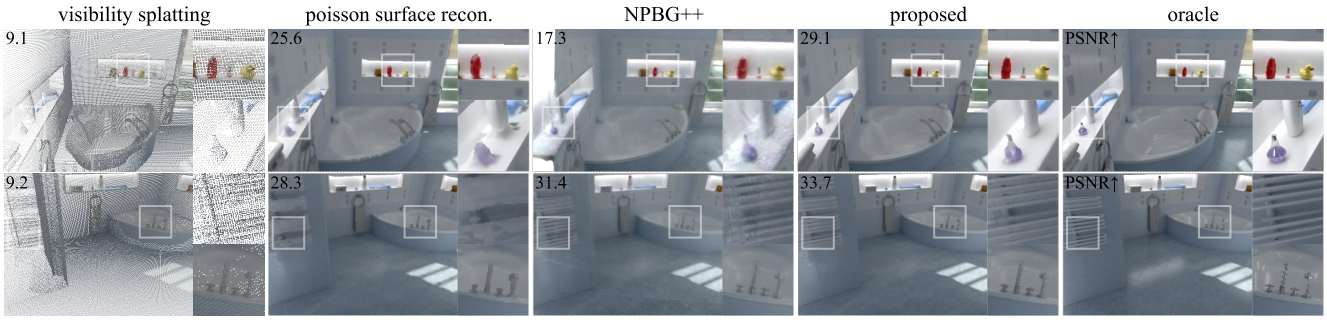


Figure 6. Room-scale point cloud rendering results. Please see supplementary material for novel-view rendering videos.

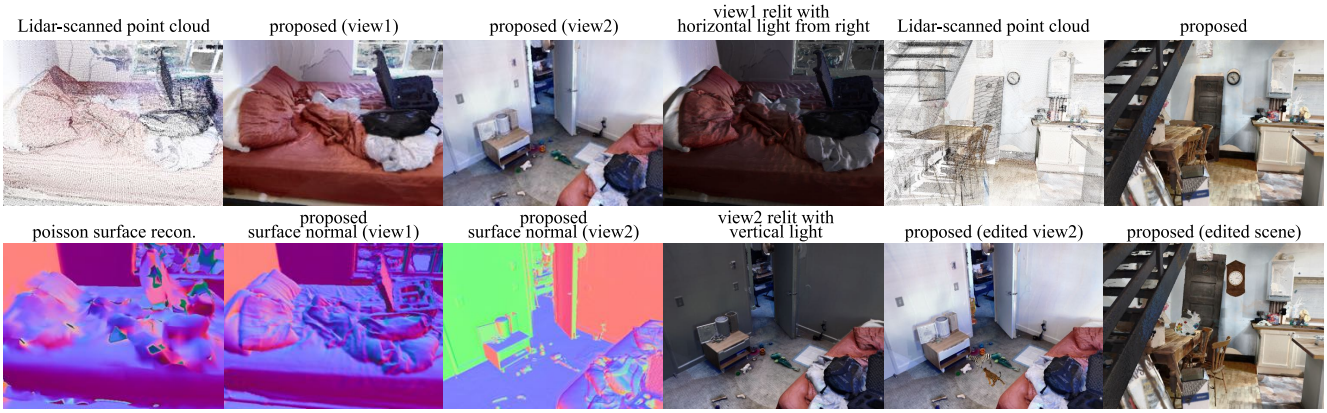


Figure 7. We test our model on a Lidar-scanned point cloud. Due to the small amount of noise, the quality of Poisson reconstruction degrades significantly. In comparison, pointersect is less affected. With the estimated surface normal, we re-render the scene with directional-dominant light using the colors as albedo. Using point cloud as the scene representation enables easy scene editing. We edit the input point clouds and render the new scene using pointersect. Can you spot the differences? Please see supplementary material for novel-view rendering, relighting, and scene-editing videos. CC clock: Gush [17].

5. Discussions

Rendering speed. Pointersect requires one transformer evaluation per query ray. In contrast, NeRF and SDF methods require multiple evaluations per ray (in addition to per-scene training). Figure 5 shows the time to render 10 test images in Section 4.3. With a resolution of 512×384 and 700k points, the current rendering speed of pointersect is ~ 1 frame per second (fps) with unoptimized Python code. The speed can be greatly improved with streamlined implementation and accelerated attention [24, 31]. See Appendix D for detailed complexity and runtime analysis.

Known artifacts. Currently, our pointersect model produces two types of artifacts. First, pointersect may generate floating points when a query ray is near an edge where the occluded background is far away or at the middle of two parallel edges. Second, we currently pass a fixed number of neighboring points to the pointersect model. Thus when there are multiple layers of surfaces, the actual (*i.e.*, first) surface may receive only a small number of points, reducing the output quality.

Connection to directed distance fields. Aumentado-Armstrong et al. [5] and Feng et al. [13] propose to represent a mesh with a Directed Distance Field (DDF). Similar to a SDF, a DDF, $\mathcal{D}(\mathbf{r})$, is learned specifically for each mesh, and it records the traveling distance of ray $\mathbf{r}$ to the nearest surface. The proposed pointersect, $f(\mathbf{r}, \mathcal{P})$, can be thought of as an estimator of DDFs. Aumentado-Armstrong et al. [5] derive several geometric properties of DDFs, which also apply to pointersect.

6. Conclusion

We have introduced pointersect, a novel method to render point clouds as if they are surfaces. Compared to other scene representations like implicit functions and prior point-cloud rendering methods, pointersect provides a unique combination of capabilities, including direct rendering of input point clouds without per-scene optimization, direct surface normal estimation, differentiability, ray tracing with global illumination, and intuitive scene editing. With the ubiquitousness of point clouds in 3D capture and reconstruction, we believe that pointersect will spur innovations in computer vision as well as virtual and augmented reality technology.

Acknowledgment. We thank Yi Hua for all the interesting general discussions about geometry.

References

[1] Bart Adams, Richard Keiser, Mark Pauly, Leonidas J Guibas, Markus Gross, and Philip Dutré. Efficient raytracing of deforming point-sampled surfaces. In *Computer Graphics Forum*, volume 24, pages 677–684, 2005. 3

[2] Anders Adamson and Marc Alexa. Approximating and intersecting surfaces from points. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 230–239, 2003. 3

[3] Kara-Ali Aliev, Artem Sevastopolsky, Maria Kolos, Dmitry Ulyanov, and Victor Lempitsky. Neural point-based graphics. In *European Conference on Computer Vision*, pages 696–712. Springer, 2020. 2

[4] Nina Amenta and Yong Joo Kil. Defining point-set surfaces. *ACM Transactions on Graphics (TOG)*, 23(3):264–270, 2004. 3

[5] Tristan Aumentado-Armstrong, Stavros Tsogkas, Sven Dickinson, and Allan D Jepson. Representing 3d shapes with probabilistic directed distance fields. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 19343–19354, 2022. 2, 8

[6] Gilad Baruch, Zhuoyuan Chen, Afshin Dehghan, Tal Dimry, Yuri Feigin, Peter Fu, Thomas Gebauer, Brandon Joffe, Daniel Kurz, Arik Schwartz, and Elad Shulman. ARKitScenes - a diverse real-world dataset for 3d indoor scene understanding using mobile RGB-D data. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. 7

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

[8] Robert L Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics (TOG)*, 1(1):7–24, 1982. 3, 4

[9] Peng Dai, Yinda Zhang, Zhuwen Li, Shuaicheng Liu, and Bing Zeng. Neural point cloud rendering via multi-plane projection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7830–7839, 2020. 2

[10] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. *SIGGRAPH*, 22(4):21–30, 1988. 5

[11] Stavros Diolatzis, Julien Philip, and George Drettakis. Active exploration for neural global illumination of variable scenes. *ACM Transactions on Graphics (TOG)*, 41(5):1–18, 2022. 3

[12] Stefano Esposito, Daniele Baieri, Stefan Zellmann, André Hinkenjann, and Emanuele Rodolà. KiloNeuS: Implicit neural representations with real-time global illumination. *arXiv preprint arXiv:2206.10885*, 2022. 3

[13] Brandon Yushan Feng, Yinda Zhang, Danhang "Danny" Tang, Ruofei Du, and Amitabh Varshney. PRIF: Primary ray-based implicit function. In *European Conference on Computer Vision (ECCV)*, 2022. 2, 8

[14] Wanquan Feng, Jin Li, Hongrui Cai, Xiaonan Luo, and Juyong Zhang. Neural points: Point cloud representation with neural fields for arbitrary upsampling. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 18633–18642, 2022. 1, 3, 5, 6

[15] Markus Gross and Hanspeter Pfister. *Point-based graphics*. Elsevier, 2011. 1, 3

[16] Gaël Guennebaud and Markus Gross. Algebraic point set surfaces. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, page 23–es, New York, NY, USA, 2007. Association for Computing Machinery. 3

[17] Joseph Gush. Antique wall clock. https://sketchfab.com/3d-models/antique-wall-clock-1542879be00b4c1d8d4330aac9669927. 8

[18] Rana Hanocka, Gal Metzer, Raja Giryes, and Daniel Cohen-Or. Point2mesh: a self-prior for deformable meshes. *ACM Transactions on Graphics (TOG)*, 39(4):126–1, 2020. 1, 2, 3

[19] Xiaoyang Huang, Yi Zhang, Bingbing Ni, Teng Li, Kai Chen, and Wenjun Zhang. Boosting point clouds rendering via radiance mapping. In *Proceedings of the AAAI conference on artificial intelligence*, 2023. 2

[20] James T. Kajiya. The rendering equation. *SIGGRAPH*, 20(4):143–150, aug 1986. 4

[21] Brian Karis and Epic Games. Real shading in unreal engine 4. *Proc. Physically Based Shading Theory Practice*, 4(3):1, 2013. 6

[22] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, volume 7, 2006. 1, 3

[23] Michael Kazhdan and Hugues Hoppe. Screened poisson surface reconstruction. *ACM Transactions on Graphics (TOG)*, 32(3):1–13, 2013. 3, 5

[24] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations (ICLR)*, 2020. 8

[25] Leif Kobbelt and Mario Botsch. A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801–814, 2004. 3

[26] Ravikrishna Kolluri. Provably good moving least squares. *ACM Transactions on Algorithms (TALG)*, 4(2):1–25, 2008. 3

[27] Patrick Labatut, Jean-Philippe Pons, and Renaud Keriven. Efficient multi-view reconstruction of large-scale scenes using interest points, delaunay triangulation and graph cuts. In *IEEE International Conference on Computer Vision (ICCV)*, pages 1–8, 2007. 1

[28] Christoph Lassner and Michael Zollhofer. Pulsar: Efficient sphere-based neural rendering. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1440–1449, 2021. 1, 2, 3

[29] Linjie Lyu, Ayush Tewari, Thomas Leimkühler, Marc Habermann, and Christian Theobalt. Neural radiance transfer fields for relightable novel-view synthesis with global illumination. In *European Conference on Computer Vision (ECCV)*, pages 153–169. Springer, 2022. 3

[30] Baorui Ma, Zhizhong Han, Yu-Shen Liu, and Matthias Zwicker. Neural-pull: Learning signed distance function from point clouds by learning to pull space onto surface. In *International Conference on Machine Learning (ICML)*, pages 7246–7257. PMLR, 2021. 1, 2, 3

[31] Dmitrii Marin, Jen-Hao Rick Chang, Anurag Ranjan, Anish

Prabhu, Mohammad Rastegari, and Oncel Tuzel. Token pooling in vision transformers. *arXiv preprint arXiv:2110.03860*, 2021. 8

[32] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021. 3

[33] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics (TOG)*, 41(4):102:1–102:15, July 2022. 6, 7

[34] Julian Ost, Issam Laradji, Alejandro Newell, Yuval Bahat, and Felix Heide. Neural point light fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18419–18429, 2022. 2, 3

[35] Songyou Peng, Chiyu Jiang, Yiyi Liao, Michael Niemeyer, Marc Pollefeys, and Andreas Geiger. Shape as points: A differentiable poisson solver. *Advances in Neural Information Processing Systems (NeurIPS)*, 34:13032–13044, 2021. 1, 3

[36] Hanspeter Pfister, Matthias Zwicker, Jeroen Van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342, 2000. 1, 2

[37] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016. 3, 4

[38] Yue Qian, Junhui Hou, Sam Kwong, and Ying He. Pugeo-net: A geometry-centric network for 3d point cloud upsampling. In *European Conference on Computer Vision (ECCV)*, pages 752–769. Springer, 2020. 5

[39] Ruslan Rakhimov, Andrei-Timotei Ardelean, Victor Lempitsky, and Evgeny Burnaev. NPBG++: Accelerating neural point-based graphics. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 15969–15979, 2022. 2, 5, 6

[40] Gernot Riegler and Vladlen Koltun. Stable view synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12216–12225, 2021. 3

[41] Mike Roberts, Jason Ramapuram, Anurag Ranjan, Atulit Kumar, Miguel Angel Bautista, Nathan Paczan, Russ Webb, and Joshua M Susskind. Hypersim: A photorealistic synthetic dataset for holistic indoor scene understanding. In *IEEE International Conference on Computer Vision (ICCV)*, pages 10912–10922, 2021. 6

[42] Darius Rückert, Linus Franke, and Marc Stamminger. ADOP: Approximate differentiable one-pixel point rendering. *ACM Transactions on Graphics (TOG)*, 41(4):1–14, 2022. 2

[43] Peter Shirley. Ray tracing in one weekend. *Amazon Digital Services LLC*, 1, 2018. 4, 6

[44] Vincent Sitzmann, Semon Rezchikov, Bill Freeman, Josh Tenenbaum, and Fredo Durand. Light field networks: Neural scene representations with single-evaluation rendering. *Advances in Neural Information Processing Systems*, 34:19313–19325, 2021. 3

[45] Weiwei Sun, Andrea Tagliasacchi, Boyang Deng, Sara Sabour, Soroosh Yazdani, Geoffrey E Hinton, and Kwang Moo Yi. Canonical capsules: Self-supervised capsules in canonical pose. *Advances in Neural Information Processing Systems (NeurIPS)*, 34:24993–25005, 2021. 2

[46] Towaki Takikawa, Or Perel, Clement Fuji Tsang, Charles Loop, Joey Litalien, Jonathan Tremblay, Sanja Fidler, and Maria Shugrina. Kaolin wisp: A pytorch library and engine for neural fields research. https://github.com/NVIDIAGameWorks/kaolin-wisp, 2022. 6, 7

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)*, 30:5998–6008, 2017. 4

[48] Ingo Wald and Hans-Peter Seidel. Interactive ray tracing of point-based models. In *Proceedings of the Second Eurographics / IEEE VGTC Conference on Point-Based Graphics*, SPBG'05, page 9–16, Goslar, DEU, 2005. Eurographics Association. 3

[49] Qianqian Wang, Zhicheng Wang, Kyle Genova, Pratul P Srinivasan, Howard Zhou, Jonathan T Barron, Ricardo Martin-Brualla, Noah Snavely, and Thomas Funkhouser. Ibrnet: Learning multi-view image-based rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4690–4699, 2021. 3

[50] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004. 5

[51] Qiangeng Xu, Zexiang Xu, Julien Philip, Sai Bi, Zhixin Shu, Kalyan Sunkavalli, and Ulrich Neumann. Point-nerf: Point-based neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5438–5448, 2022. 2, 3

[52] Lior Yariv, Yoni Kasten, Dror Moran, Meirav Galun, Matan Atzmon, Basri Ronen, and Yaron Lipman. Multiview neural surface reconstruction by disentangling geometry and appearance. *Advances in Neural Information Processing Systems*, 33:2492–2502, 2020. 3

[53] Wang Yifan, Felice Serena, Shihao Wu, Cengiz Öztireli, and Olga Sorkine-Hornung. Differentiable surface splatting for point-based geometry processing. *ACM Transactions on Graphics (TOG)*, 38(6):1–14, 2019. 1, 2

[54] Qiang Zhang, Seung-Hwan Baek, Szymon Rusinkiewicz, and Felix Heide. Differentiable point-based radiance fields for efficient view synthesis. In *SIGGRAPH Asia 2022 Conference Papers*, New York, NY, USA, 2022. Association for Computing Machinery. 2

[55] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. 5

[56] Xiuming Zhang, Pratul P Srinivasan, Boyang Deng, Paul Debevec, William T Freeman, and Jonathan T Barron. Nerfactor: Neural factorization of shape and reflectance under an unknown illumination. *ACM Transactions on Graphics (TOG)*, 40(6):1–18, 2021. 3

[57] Kun Zhou, Xi Wang, Yiying Tong, Mathieu Desbrun, Baining Guo, and Heung-Yeung Shum. Texturemontage: Seamless texturing of arbitrary surfaces from multiple images. *ACM Transactions on Graphics (TOG)*, 24(3):1148–1155, 2005.

Available at http://kunzhou.net/tex-models.htm. 5

[58] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018. 5

[59] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. Surface splatting. In *Proceedings of Computer Graphics and Interactive Techniques*, pages 371–378, 2001. 2