# Incorporating dense metric depth into neural 3D representations for view synthesis and relighting

## Supplementary Material

This supplementary document inherits the figure, equation, table, and reference numbers from the main document. Additional results may be viewed at https://stereomfc.github.io.

## 8. Representations and implementation details

Our scene representation consists of two networks – an intrinsic network $\mathcal{N}(\theta)$ and an appearance network $\mathcal{A}(\phi)$. We follow [62] to build and train $\mathcal{N}(\theta)$. We use 18 levels of hashgrid encodings [76] to encode the input and a two layer (128 neurons/layer) MLP to generate the intrinsic embedding. The first channel of the embedding, $\mathcal{S}(\theta)$ is trained with Eq. (2) to recover a signed distance field of the scene as described in Sec. 3.1. The rest of the 127 channels of the embedding $\mathcal{E}(\theta)$ are passed on to the appearance network $\mathcal{A}(\phi)$ as an input.

The appearance network takes $\mathcal{E}(\theta)$, the viewing direction (encoded with 6 levels of sinusoidal encodings following [96]), and optionally the illumination direction (if recovering BRDF) to generate colors. The neural network is built with 2 layers of fully connected MLPs (128 neurons/layer) with skip connections.

The neural signed distance field $\mathcal{S}(\theta)$ is optimized to return the signed distance of a point from its nearest surface $\mathcal{S}(\theta) : \mathbb{R}^3 \rightarrow \mathbb{R}$. The surface of the object can be obtained from the zero-level set of $\mathcal{S}(\theta)$ – i.e. for all surface points $\mathbf{x}_s \in \mathbb{R}^3 \mid \mathcal{S}(\mathbf{x}_s|\theta) = 0$. We train $\mathcal{S}(\theta)$ by minimizing a geometric loss $\ell_D$ (Eq. (2)). We follow [105] to transform the distance of a point $\vec{p}_i = \vec{r}|_{t_i}$ in a ray to its closest surface $s_i = \mathcal{S}(\theta, \vec{p}_i)$ to the scene density (or transmissivity).

$$\Psi_\beta(s) = \begin{cases} 0.5\exp(\frac{s}{\beta}), & s \leq 0 \\ 1 - 0.5\exp(\frac{-s}{\beta}), & \text{otherwise.} \end{cases} \tag{4}$$

To render the color $\mathbf{C}$ of a single pixel of the scene at a target view with a camera centered at $\vec{o}$ and an outgoing ray direction $\vec{d}$, we calculate the ray corresponding to the pixel $\vec{r} = \vec{o} + t\vec{d}$, and sample a set of points $t_i$ along the ray. The networks $\mathcal{N}(\theta)$ and $\mathcal{A}(\phi)$ are then evaluated at all the $\mathbf{x}_i$ corresponding to $t_i$ and the per point color $\mathbf{c}_i$. The transmissivity $\tau_i$ is obtained and composited together using the quadrature approximation from [70] as:

$$\mathbf{C} = \sum_i \exp(-\sum_{\mathrm{j}<\mathrm{i}} \tau_\mathrm{j}\delta_\mathrm{j})(1-\exp(-\tau_\mathrm{j}\delta_\mathrm{j}))\mathbf{c}_\mathrm{i}, \quad \delta_\mathrm{i} = \mathrm{t_i} - \mathrm{t_{i-1}} \tag{5}$$

The appearance can then be learned using a loss on the estimated and ground truth color $\mathbf{C}_{gt}$

$$\ell_C = \mathbb{E}\left[||\mathbf{C} - \mathbf{C}_{gt}||^2\right] \tag{6}$$

The appearance and geometry are jointly estimated by minimizing the losses in Eq. (7) using stochastic gradient descent [55].

$$\ell = \ell_C + \lambda_g \ell_D + \lambda_c \mathbb{E}(|\nabla^2_{\mathbf{x}} \mathcal{S}(\mathbf{x}_s)|) \tag{7}$$

$\lambda$s are hyperparameters and the third term in Eq. (7) is the mean surface curvature minimized against the captured surface normals. As the gradients of the loss functions $\ell_C$ and $\ell_D$ propagate through $\mathcal{A}$ and $\mathcal{N}$ (and $\mathcal{S}$ as it is part of $\mathcal{N}$) the appearance and geometry are learned together.

### 8.1. Details of our baselines

We implemented four baselines to investigate the effects of incorporating dense metric depth and depth edges into neural view synthesis pipelines.

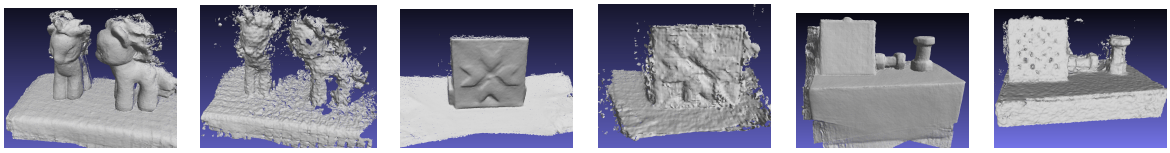
**VolSDF**$^{++}$ is our method similar to VolSDF [105] and MonoSDF[108]. We represent the scene with $\mathcal{N}$ and $\mathcal{A}$ and train it with metric depth and color by minimizing Eq. (7). The samples for Eq. (6) are drawn using the "error-bounded sampler" introduced by [105].

**NeUS**$^{++}$ represents a modified version of NeUS [100], where we use the training schedule and structure of $\mathcal{N}$ from [62], the appearance network $\mathcal{A}$ is adopted from NeUS and we optimize Eq. (2) along with Eq. (6). In addition to $\mathcal{A}$, NeUS$^{++}$also has a small 4 layer MLP (32 neurons per layer) to learn the radiance of the background as recommended in the original work by [100].

**UniSurf**$^{++}$ is our method inspired by UniSurf[77]. We represent the scene's geometry using a pre-optimized implicit network $\mathcal{N}$ as outlined in Sec. 3.1. We follow the recommendations of [77] to optimize $\mathcal{A}$. UniSurf exposes a hyperparameter to bias sampling of Eq. (5) towards the current estimate of the surface. As we pre-optimize the surface, we can find the surface point $\mathbf{x}_s = \mathbf{o} + t_s\mathbf{d}$ through sphere tracing $\mathcal{S}$ along a ray. The intersection point $t_s$ can then be used to generate $N$ samples along the ray to optimize Eq. (6).

$$t_i = \mathcal{U}\left[t_s + \left(\frac{2i-2}{N} - 1\right)\Delta, t_s + \left(\frac{2i}{N} - 1\right)\Delta\right] \tag{8}$$

Equation (8) is the distribution used to draw samples and $\Delta$ is the hyperparameter that biases the samples to be close to the current surface estimate. We optimize $\mathcal{S}$ independent



(a) (b) (c) (d) (e) (f)

**Figure 9. AdaShell$^{++}$can work with noisy depth** with little loss in view interpolation performance and training time. However, the surface recovered is also noisy. Figures 9a, 9c and 9e are the geometries recovered with no noise in depth and Figs. 9b, 9d and 9f are with noisy depths. Although AdaShell$^{++}$does not de-noise the geometry, there is very little degradation in performance (training speed and view synthesis quality) with noisy depth. NeUS$^{++}$struggles at the task of view synthesis with noisy depth. Details in Sec. 5.4 and Tab. 6.

of Eq. (5) by just minimizing Eq. (2) with registered depth maps (see Sec. 3.1). We use this method to study the effects of volumetric rendering versus surface rendering. We found this strategy to be very sensitive to the hyperparameter $\Delta$ and its decay schedule as the training progressed. While best parameters for some sequences resulted in very quick convergence, they were very hard to come across and generally, poorer choices led to undesirable artifacts (see e.g. Fig. 4).

We describe **AdaShell$^{++}$** in the following section.

## 8.2. AdaShell$^{++}$: Accelerating training with dense depth

The slowest step in training and inference for neural volumetric representations is generally the evaluation of Eq. (5). In this section we describe our method to accelerate training by incorporating metric depth.

A method to make training more efficient involves drawing the smallest number of the most important samples of $t_i$ for any ray. The sampling of $t_i$ is based on the current estimate of the scene density and although these samples can have a large variance, given a large number of orthogonal view pairs (viewpoint diversity), and the absence of very strong view dependent effects, the training procedure is expected to recover an unbiased estimate of the true scene depth (see e.g. [43]). We can accelerate the convergence by a) providing high quality biased estimate of the scene depth and b) decreasing the number of samples for $t_i$ along the rays.

Given the high quality of modern deep stereo (we use[103]) and a well calibrated camera system, stereo depth can serve as a good initial estimate of the true surface depth. We use stereo depth, aligned across multiple views of the scene to pre-optimize the geometry network $\mathcal{S}(\theta)$. The other channels $\mathcal{E}(\theta)$ of $\mathcal{N}$ remain un-optimized. A pre-optimized $\mathcal{S}$ can then be used for high quality estimates of ray termination depths.

[77, 100, 105] recommend using root finding techniques (e.g. bisection method) on scene transmissivity (Eq. (4)) to estimate the ray termination depth. The samples for Eq. (5) are then generated around the estimated surface point. Drawing high variance samples as $\mathcal{N}$ and $\mathcal{A}$ are jointly optimized reduces the effect of low quality local minima, especially in the initial stages of the optimization. As we have a pre-trained scene transmissivity field ($\mathcal{S}$ transformed with Eq. (4)), we can draw a few high-quality samples to minimize the training effort.

We found uniformly sampling around the estimated ray-termination depth (UniSurf$^{++}$baseline in Sec. 3.3 and Fig. 4) to be unsuitable. Instead, we pre-calculated a discrete sampling volume by immersing $\mathcal{S}$ in an isotropic voxel grid and culling the voxels which report a lower than threshold scene density. We then used an unbiased sampler from [105] to generate the samples in this volume. This let us greatly reduce the number of root-finding iterations and samples, while limiting the variance by the dimensions of the volume along a ray. As the training progresses, we decrease the culling threshold to converge to a thinner sampling volume around the surface while reducing the number of samples required.

We show the sampling volume (at convergence) and our reconstruction results in Figs. 7 and 10 respectively. We retain the advantages of volumetric scene representation as demonstrated by the reconstruction of the thin structures in the scene, while reducing training effort. We dub our method AdaShell$^{++}$to acknowledge [101], which demonstrates a related approach to accelerate inference.

The "shells" shown in Figs. 7 and 10 and the shells recovered by [101] for small scenes are physically similar quantities. [101] dilate and erode the original level-set of the scene (approximated by $\mathcal{S}$) using a hyperparameter. Our "shells" are also jointly estimated with the geometry as the training progresses. [101] estimate the fall-off of the volume density values along a ray to determine the hyperparameters, which in turn determines the thickness of the "shell". They subsequently use uniform sampling (similar to Eq. (8), where the $\Delta$ now denotes the local thickness of the shell) to generate samples for rendering. Our work takes a discrete approach by immersing the zero-level set (in form of pre-optimized $\mathcal{S}$) in a dense isotropic voxel grid and culling the voxels which have a lower volume density, according to a preset hyperparameter that determines the thickness of the shell. Once the shell has been estimated, we use a unbiased density weighted sampler (instead of a uniform sampler) to generate samples along the ray inside the shell. We roughly follow [95] to generate samples

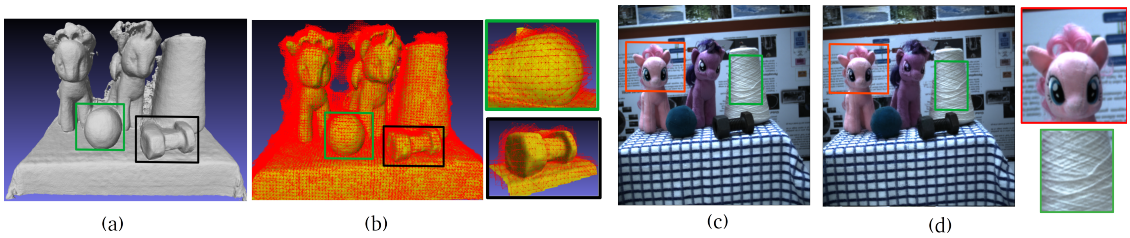



**Figure 10. Another example of shells recovered by AdaShell$^{++}$.** Fig. a shows the geometry recovered with 5 RGBD tuples. Figure b displays the sampling volumes around the geometry after AdaShell$^{++}$has converged – we note the similarity of this step with [101]. Figs. c and d are the ground-truth and reconstructed test images. Related example in Fig. 7.

along a segment of the ray guided by the voxels it intersects. The spatial density of samples is inversely proportional to their distance from the estimated surface. We implement this using the tools from NerfStudio[61, 97]. Our sampling strategy is more robust to errors in estimated geometry (as shown in Sec. 5.4 and Fig. 9) than other approaches –notably NeUS$^{++}$and the original work (Adaptive Shells [101]) which is based on NeUS([100]).

### 8.3. Training Details

We ran our experiments on a Linux workstation with an Intel Core i9 processor, 64GB RAM, and an Nvidia RTX3090Ti graphics card with 25GB of vRAM. Across all the experiments for learning scene radiance, we implemented a hard cut-off of 100K gradient steps amounting to less than 4.5 hours of training time across all the experiments.

Across all our baselines (AdaShell$^{++}$, VolSDF$^{++}$, NeUS$^{++}$, and UniSurf$^{++}$) we used the intrinsic network proposed in [62], with 2 layers of MLPs (128 neurons per fully connected layer) and 18 levels of input hash encodings activated gradually. Our input activation curriculum was based on the recommendations of [62], and was used jointly with our edge-aware sampling strategy Sec. 3.2, Eq. (3), and Fig. 3 across all the scenes. The implementations of our baselines, design and bill-of-materials for the multi-flash camera system, dataset and the hyperparameters will be released soon. Figure 7 denotes the training steps graphically.

### 8.4. Difference between our and prior work on neural scene understanding with depth

**IGR[45]** was among the first to fit a neural surface to point samples of the surface. Our pipeline is largely inspired by that work. However, we have two main differences – we use a smaller network, and periodically activate multi-resolution hash encodings as recommended by [62] instead of using a fully connected set of layers with skip connections. Additionally, as we have access to depth maps, we identify the variance of the neighborhood of a point on the surface through a sliding window filter. We use this local estimate of variance in a normal distribution to draw samples for $\mathbf{x}_s^{\Delta}$ along each ray. Our strategy assumes that image-space pixel neighbors are also world space neighbors, which is incorrect along the depth edges. However, as the Eikonal equation should be generally valid in $\mathbb{R}^3$ for $\mathcal{S}$, the incorrect samples do not cause substantial errors and only contribute as minor inefficiencies in the pipeline. A more physically based alternative, following [45], would be executing nearest neighbor queries at each surface point along the rays to estimate the variance for sampling. With about 80k rays per batch, $\sim$ 200K points in $(\mathbf{x}_s)$, and about 40k gradient steps executed till convergence, and a smaller network, our approach was more than two orders of magnitude faster than [45], with no measurable decrease in accuracy of approximating the zero-level set of the surface.

**NeuralRGBD[9]** is the closest prior work based on data needed for the pipeline and its output. The scene is reconstructed using color and aligned dense metric depth maps. The authors aggregate the depth maps as signed distance fields and use the signed distance field to calculate weights for cumulative radiance along samples on a ray (Eq. (5) in text). The weights are calculated with

$$w_i = \sigma\left(\frac{D_i}{tr}\right) \times \sigma\left(-\frac{D_i}{tr}\right) \tag{9}$$

where the $D_i$ is the distance to the surface point along a ray, and the truncation $tr$ denotes how fast the weights fall off away from the surface. Equation (9) yields surface biased weights with a variance controlled by the parameter $tr$. Notably, the depth map aggregation does not yield a learned sign distance field (no Eikonal regularizer in the loss). The authors also include a 'free-space' preserving loss to remove "floaters". As implemented, the pipeline needs the truncation factor to be selected per-scene. As the depth maps are implicitly averaged by a neural network, it is implicitly smoothed and therefore the pipeline is robust to local noise in the depth map.

**MonoSDF[108]** is mathematically the closest prior method to our work and it uses dense scene depths and normals obtained by a monocular depth and normal prediction network (OmniData[33]). MonoSDF defines the ray length weighted with the scene density as the scene depth $\mathbf{d}_{pred}$

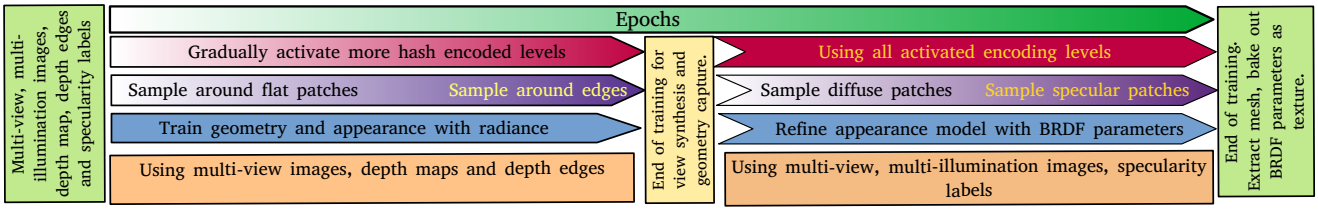



**Figure 11. Our approach to recovering 3D assets from captured data**. In the first part, for NeUS$^{++}$and VolSDF$^{++}$we jointly optimize geometry and appearance by minimizing Eqs. (2) and (6). For AdaShell$^{++}$and UniSurf$^{++}$, we first optimize Eq. (2) for a fixed number of gradient steps before the joint optimization. At this stage the geometry is optimized and appearance is recovered as radiance. Following this, we use multi-illumination images with a truncated BRDF parametrization to refine the appearance model, given the geometry, to learn the reflectance parameters.

and minimizes

$$\ell_D = \sum_r ||\mathbf{w}\mathbf{d}_{mono} + \mathbf{q} - \mathbf{d}_{pred}||_2^2 \tag{10}$$

where $\{\mathbf{w}, \mathbf{q}\}$ are scale and shift parameters. Estimating an affine transformation on the monocular depth $\mathbf{d}_{mono}$ is important because in addition to gauge freedom ($\mathbf{w}$), monocular depths also have an affine degree of freedom ($\mathbf{q}$). The scale and shift can be solved using least squares to align $\mathbf{d}_{mono}$ and $\mathbf{d}_{pred}$. The scene normals are calculated as gradients of $\mathcal{S}$ weighted with scene density along a ray. Through a scale and shift invariant loss, MonoSDF calculates one set of $(\mathbf{w}, \mathbf{q})$ for all the rays in the batch corresponding to a single training RGBD tuple. In the earlier stages of the training, this loss helps the scene geometry converge. The underlying assumption is that there is an unique tuple $\{\mathbf{w}, \mathbf{q}\}$ per training image that aligns $\mathbf{d}_{mono}$ to the actual scene depth captured by the intrinsic network $\mathcal{N}$.

Our experiments with MonoSDF indicate that the network probably memorizes the set of $(\mathbf{w}, \mathbf{q})$ tuples per training image. Explicitly passing an unique scalar tied to the training image (e.g. image index as proposed in [68]) speeds up convergence significantly. Success of MonoSDF in recovering both shape and appearance strongly depends on the quality of the monocular depth and normal predictions. Our experiments on using MonoSDF on the Wild-Light dataset([21]) or the ReNe dataset ([98]) failed because the pre-trained Omnidata models performed poorly on these datasets. Unfortunately, as implemented, MonoSDF also failed to reconstruct scene geometry when the angles between the training views were small – ReNe dataset views are maximally 45° apart. However, it demonstrates superior performance on the DTU and the BlendedMVS sequences while training with as low as three pre-selected views. Finally, our scenes were captured with a small depth of field and most of the background was out of focus, so the scene background depth was significantly more noisy than the foreground depth. We sidestepped this problem by assigning a fixed 1m depth to all the pixels that were in the background. Although this depth mask simplifies our camera pose estimation problem (by segregating the foreground from the background), it assigns multiple infeasible depths to a single background point. As we aggregate the depth maps into the intrinsic network ($\mathcal{N}$) by minimizing Eq. (2), the network learns the mean (with some local smoothing) of the multiple depths assigned to the single background point. However, the scale and shift invariant loss is not robust to this and with masked depth maps, we could not reliably optimize MonoSDF on our sequences. We suspect that this is because the scale and shift estimates for each instance of Eq. (10) on the background points yielded very different results, de-stabilizing the optimization.

**[83]** and **[30]** use sparse scene depth in the form of SfM triangulated points. [83] use learnt spatial propagation [20] to generate dense depth maps from the sparse depth obtained by projecting the world points triangulated by SfM. [30] assign the closest surface depth at a pixel obtained by projecting the triangulated points to the image plane. Neither of these pipelines recover a 3D representation of the scene and focus on view synthesis using few views.

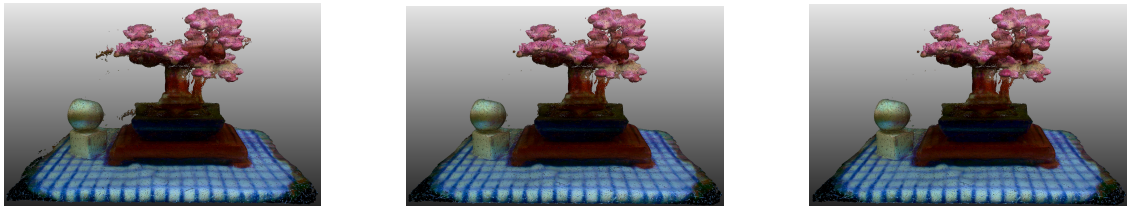
**[84]** introduce a novel 3D representation – "Neural point clouds" which includes geometric and appearance feature descriptors (small MLPs) grounded to a point in 3D. The geometry is recovered as the anchors of the "neural points". The appearance is calculated using a volumetric renderer which composits the outputs of the appearance descriptors of the neural points with the transmissivity of the neural points along the ray. The transmissivity of a neural point is calculated as a function of distances of a pre-set number of neighboring neural points.

### 8.5. Capturing approximate BRDF and generating textured meshes

Multi-illumination images captured by our camera system can be used to estimate surface reflectance properties. We recover a truncated Disney BRDF model([14]. Our model consists of a per pixel specular albedo, a diffuse RGB albedo, and a roughness value to interpret the observed appearance under varying illumination. To estimate the spatially varying reflectance, we first train a model (AdaShell$^{++}$, VolSDF$^{++}$or NeUS$^{++}$) to convergence to learn the appearance as radiance. At convergence, the first channel $\mathcal{S}$ of the intrinsic network $\mathcal{N}$ encodes the geom-

**(a)** Limited parametrization **(b)** Full Disney BDRF parametrization

**Figure 12. Optimizing for the full Disney BRDF is difficult**. Figure 12a shows our results with only specularity, roughness and metallic BRDF parameters. Figure 12b depicts identical results utilizing the complete range of Disney BRDF parameters as outlined in [21]. Note the excessive glossy appearance of Fig. 12b due to the dominance of the clearcoat and clearcoat-gloss parameters. Details in Sec. 8.5, meshes rendered with [92].



**(a)** AdaShell$^{++}$ **(b)** VolSDF$^{++}$ **(c)** NeUS$^{++}$

**Figure 13. All the pipelines** can be used to extract the "base-color" of the scene. We calculate texture of the meshes from the radiance at convergence (PSNR 27.5+) for one of the scenes in Fig. 1. The textured meshes are rendered with MeshLab[23].

etry and the appearance network $\mathcal{A}$ encodes the radiance. We use two of the embedding channels of $\mathcal{E}$ to predict the roughness and specular albedo at every point on the scene. The diffuse albedo is obtained as the output of the converged appearance network $\mathcal{A}$. To calculate the appearance, we apply the shading model ([14]) to calculate the color at every sample along a ray and volumetrically composite them using Eq. (5) to infer appearance as reflectance. Figure 11 describes our steps graphically.

Optimizing for the full set of the Disney BRDF parameters, following [21] did not work with our aproach as the optimization often got stuck at poor local minima. Figure 12b shows one instance of optimizing the pipeline of [21], where the strengths of the recovered 'clearcoat' and 'clearcoat-gloss' parameters dominated over the optimization of the other parameters, resulting in a waxy appearance. Choosing a more conservative set of parameters (only 'base-color', 'specular' and 'roughness') in Fig. 12a led to a more realistic appearance. WildLight[21] is based on [100] – we substituted [100] with NeUS$^{++}$and the apperance model of [21] was not changed, minimizing the chances of introduction of a bug causing the artifact.

Our process of generating texture and material properties roughly follows the methods described by [21] and [97]. We proceed through the following steps:

1. At convergence (see Fig. 11), we extracted the scene geometry using the method described in [72].
2. We calculate a depth mask by thresholding the depth images at every training view with an estimate of the scene depth to segregate the foreground from the background.
3. Next, we cull the resulting triangular mesh (step 1) by projecting rays from every unmasked (foreground) pixel corresponding to all the camera views. This lets us extract the main subject of our scene as a mesh. We use Embree[102] to implement this.
4. We generate texture coordinates on the culled mesh using "Smart UV Unwrap" function from [11]. These results were qualitatively better than [107] and our implementation of [93]. We then rasterize the culled mesh from step 3 to get points on the surface corresponding to the texture coordinates.
5. We project each of these surface points back on to each of the training views to get the image coordinates. Rays originating from a rasterized surface point and intersecting the surface before reaching the camera are removed to preserve self occlusion.
6. For all the valid projected points, we cast a ray onto the scene and use either AdaShell$^{++}$, VolSDF$^{++}$, or NeUS$^{++}$to generate the color at the pixel along the ray using Eq. (5). This is repeated for all the training views.
7. At the end of the previous step we have several measurements of colors at every texture coordinate of the scene. We apply a median filter (per color channel) to choose the color – taking averages or maxima of the samples introduces artifacts. If using the radiance as texture is sufficient (often the case for diffuse scenes) this textured mesh can be exported. Figure 13 demonstrates using each of AdaShell$^{++}$, VolSDF$^{++}$and, NeUS$^{++}$to calcu-

late the diffuse color of the scene in Fig. 1.

8. To generate material textures, we follow the same procedures with the corresponding material channels after AdaShell$^{++}$, VolSDF$^{++}$or NeUS$^{++}$has been trained on multi illumination images using the schedule outlined in Fig. 11.
9. The material properties are also volumetrically composited using Eq. (5) and median filtered like the base colors. This is different from just querying the value of the network at the estimated surface point in [21].

We use [92], a web browser based tool that supports physically based rendering with the Disney BRDF parameters, and [11] to generate the images in Figs. 1 and 12 respectively.

## 9. A multi-flash stereo camera

We capture the scene using a binocular stereo camera pair with a ring of lights that can be flashed at high intensity. For our prototype, we use a pair of machine vision cameras ([38]) with a 1", 4MP CMOS imaging sensor of resolution of 2048 $\times$ 2048 pixels. As we focus mainly on small scenes, we use two sets of lenses that yield a narrow field of view – 12mm and 16mm fixed focal length lens ([32]). We use 80W 5600K white LEDs ([25]) flashed by a high-current DC power supply switched though MOSFETs controlled with an Arduino microcontroller. At each pose of our rig, we captured 12 images with each of the flash lights on (one light at a time) and one HDR image per camera. The cameras are configured to return a 12 bit Bayer image which is then de-Bayered to yield a 16 bit RGB image.

For the HDR images, we performed a sweep of exposures from the sensor's maximum (22580 microseconds) in 8 stops and used [71] to fuse the exposures captured with ambient illumination (fluorescent light panels in a room). Following the recommendations of [49] we used an f-stop of 2.8 to ensure the whole scene is in the depth of field of the sensors. We found the recommendations from [74] to be incompatible with our pipeline, so we used Reinhard tonemapping ([81]) to re-interpret the HDR images. Our image localization pipeline, and stereo matching also worked better with tonemapped images.

We set the left and right cameras to be triggered simultaneously by an external synchronization signal. We configured the camera frame acquisition and the illumination control programs to run in the same thread and synchronized the frame acquisition with the flashes through blocking function calls. Figure 5 presents a schematic of our prototype device.

Through experiments we observed that the vignetting at the edges of the frames were detrimental to the quality of reconstruction, so we only binned the central 1536 $\times$ 1536 pixels. A 16bit 1536 $\times$ 1536 frame saved as a PNG image was often larget than 10MB. To achieve a faster capture and training time without sacrificing the field of view, we down sampled the images to a resolution of 768 $\times$ 768 pixels for our experiments. Centered crops of our initial larger frames lead to failures of our pose-estimation pipelines due to the field of view being too narrow(Sec. 4.2), so we chose to down sample the images instead. For the images lit by a single LED, we used the camera's auto exposure function to calculate an admissible exposure for the scene and used 80% of the calculated exposure time for imaging – the built-in auto-exposure algorithm tended to over-expose the images a bit. Estimating the exposure takes about 2 seconds. Once the exposure value is calculated, it is used for all of the 12 flashes for each camera.

Several instances of these RGBD tuples are collected and the colored depth maps are registered in the 3D space in two stages – first coarsely using FGR [113] and then refined by optimizing a pose graph[22]. At the end of this global registration and odometry step, we retain a reprojection error of about 5 - 10 pixels. If the reprojection errors are not addressed, they will cause the final assets to have smudged color textures. To address it, we independently align the color images using image-feature based alignment techniques common in multi-view stereo ([85, 88]), so that a sub 1 pixel mean squared reprojection error is attained. The cameras aligned in the image-space are then robustly transformed to the world space poses using RANSAC[37] with Umeyama-Kabsch's algorithm[99]. Finally, we mask out the specular parts of the aligned images and use ColorICP [78] to refine the poses. The final refinement step helps remove any small offset in the camera poses introduced by the robust alignment step. A subset of the data collected can be viewed on the project website.

### 9.1. Identifying pixels along depth edges

To identify pixels along depth edges, we follow [79] and derive per-pixel likelihoods of depth edges. Assuming that the flashes are point light sources and the scene is Lambertian, we can model the observed image intensity for the $k^{\text{th}}$ light illuminating a point $\mathbf{x}$ with reflectance $\rho(\mathbf{x})$ on the object as

$$\mathbf{I}_k(\mathbf{x}) = \mu_k \rho(\mathbf{x}) \langle \mathbf{l}_k(\mathbf{x}), \mathbf{n}(\mathbf{x}) \rangle \tag{11}$$

where $\mu_k$ is the intensity of the $k^{\text{th}}$ source and $\mathbf{l}_k(\mathbf{x})$ is the normalized light vector at the surface point. $\mathbf{I}_k(\mathbf{x})$ is the image with the ambient component removed. With this, we can calculate a ratio image across all the illumination sources

$$\mathbf{R}(\mathbf{x}) = \frac{\mathbf{I}_k(\mathbf{x})}{\mathbf{I}_{max}(\mathbf{x})} = \frac{\mu_k \langle \mathbf{l}_k(\mathbf{x}), \mathbf{n}(\mathbf{x}) \rangle}{\max_i (\mu_i \langle \mathbf{l}_i(\mathbf{x}), \mathbf{n}(\mathbf{x}) \rangle)} \tag{12}$$

It is clear that the ratio image $\mathbf{R}(\mathbf{x})$ of a surface point is exclusively a function of the local geometry. As the light source to camera baselines are much smaller than the camera to scene distance, except for a few detached shadows and inter-reflections, the ratio images (Eq. (12)) are more

sensitive to the variations in geometry than any other parameters. We exploit this effect to look for pixels with largest change in intensity along the direction of the epipolar line between the camera and the light source on the image. This yields a per-light confidence value of whether $\mathbf{x}$ is located on a depth edge or not. Across all 12 illumination sources, we extract the maximum values of the confidences as the depth edge maps. Unlike [79], we use 12 illumination sources $30^\circ$ apart, and we do not threshold the confidence values to extract a binary edge map. This lets us extract more edges especially for our narrow depth of field imaging system and gets rid of hyper parameters used for thresholding and connecting the edges.

Often parts of our scene violate the assumption of Lambertian reflectances resulting in spurious depth edges. When we use depth edges for sampling, these errors do not affect the accuracy of our pipeline. When using depth edges for enhancing stereo matching (Sec. 4.1) we ensure that the stereo pairs do not contain too many of these spurious edge labels to introduce noise in our depth maps.

### 9.2. Identifying patches with non-Lambertian reflectances

We modified the definition of differential images in the context of near-field photometric stereo introduced by [16, 63] to identify non-Lambertian patches. Assuming uniform Lambertian reflectances, Eq. (11) can be expanded as

$$\mathbf{I}_k(\mathbf{x}) = \mu_k^* \rho(\mathbf{x}) \mathbf{n}(\mathbf{x})^T \frac{\mathbf{s}_k - \mathbf{x}}{|\mathbf{s}_k - \mathbf{x}|^3} \tag{13}$$

where $\mathbf{s}_k$ is the location and $\mu_k^*$ is the power of the $k^{\text{th}}$ light source. We define the differential images as $\mathbf{I}_t = \frac{\partial \mathbf{I}}{\partial \mathbf{s}} \mathbf{s}_t$ where, $\mathbf{s}_t = \frac{\partial \mathbf{s}}{\partial t}$, which when applied to Eq. (13) can be expanded as

$$\mathbf{I}_t(\mathbf{x}) = \mathbf{I}(\mathbf{x}) \frac{\mathbf{n}^T \mathbf{s}_t}{\mathbf{n}^T(\mathbf{s} - \mathbf{x})} - 3\mathbf{I}(\mathbf{x}) \frac{(\mathbf{s} - \mathbf{x})^T \mathbf{s}_t}{|\mathbf{s} - \mathbf{x}|^2} \tag{14}$$

Observing that the light sources move in a circle around the center of projection on the imaging plane, $\mathbf{s}^T \mathbf{s}_t = 0$. Also, the second term of Eq. (14) is exceedingly small given that the plane spanned by $\mathbf{s}_t$ is parallel to the imaging plane and our choice of lenses limit the field of view of the cameras. The second term is further attenuated by the denominator $|\mathbf{s} - \mathbf{x}|^2$ because the camera-to-light baselines ($\mathbf{s}$) are at least an order of magnitude smaller than the camera to object distance ($\mathbf{x}$). As a result, under isotropic reflectances (Lambertian assumed for this analysis) the differential images $\mathbf{I}_t(\mathbf{x})$ are invariant to circular light motions. Any observed variance therefore can be attributed to the violations of our isotropic BRDF assumptions. We identify specular patches by measuring the variance of this quantity across the 12 instances of the flashlit images.

Although our pipelines for identifying depth edges and patches of varying appearances demonstrate satisfactory qualitative performance, sometimes they yield wrong labels because Eqs. (12) and (14) do not include additional terms for spatially varying BRDFs and interreflections respectively. These errors do not have any significant effect in our reconstruction pipeline as we use this information to generate samples during different phases of training to minimize photometric losses and we do not directly infer shape or reflectances from these steps.

### 9.3. Difference between [36, 79] and our hardware

**[79]** was the first to propose pairing flashes with cameras and laid the groundwork for identifying depth edges from multi-flash images from a single viewpoint. However, [79] considered a monocular camera and only four flashes along the horizontal and vertical directions of the camera in the demonstrated device. Researchers (see e.g. [17]) have since extended it by placing multiple light sources far apart from a monocular camera and have demonstrated locating depth edges on objects with strictly Lambertian reflectances. In this work, we retain the original light and camera configuration from [79] and increase the number of lights from four to 12.

**[36]** also investigated a stereo camera in a multi-flash configuration aimed at edge preserving stereo depth maps. They do not extend the application to synthesizing geometry or appearance by capturing and assimilating multiple views of the scene. For obtaining stereo depth maps, we use [103], which performs much better than conventional stereo matching ([47, 109]) largely deployed in off-the shelf systems ([54]).

**Both [35, 79]** discuss methods to detect specularities (termed "material edges") through different transforms of the multi-light images. However, we achieve a more continuous circular motion of the lights around the cameras, so we choose to use the photometric invariants described by [16] instead.