A. Runtime and Memory

To break down the time and memory cost of our method, Figure 12 profiles the simulation and rendering modules as the number of synthetic objects increases. In the test scene, we throw $n \in \{1, 10, 100, 1000, 10000\}$ balls onto a Fox [47] represented by NeRF. The rendering resolution is 900×450.

As seen in the figure, the simulation and rendering memory stay constant at a low level (0.5 GB and 1.0 GB, respectively) throughout the entire experiment. The run time for both modules scales practically linearly w.r.t. the number of objects.

Moreover, the simulation module is independent of the resolution, and the rendering time scales linearly w.r.t. the number of pixels (see left). This pipeline can run in real time (20 FPS) at 600×300 resolution.

B. Comparison with density fields

We compare the simulation quality with different collision geometry. In Figure 14, we drop four balls above a messy counter top [5]. In this simulation scene, (a) models the collision geometry using learned SDF by our NeuS-NGP and (b) directly uses Marching-Cube mesh from instant-NGP density fields (with resolution $256 \times 256 \times 256$). Two of the balls in (b) 'sink' into the countertop, which is unrealistic. In contrast, our SDF-based simulation in (a) can correctly simulate the interaction between the balls and the background objects. More dynamics results can be found in our supplementary video.

C. Comparison with other NeRF-based simulation.

There are two recent works [14, 60] that can simulate in neural fields. [14] establish a collision model based on the density field. In our simulation, we found that the density fields are usually noisy and inadequate to model a surface well for accurate contact processing. NeuPhysics[60] aims at differentiable simulation and rendering, and it extracts hexahedra mesh from learned SDF [75] and simulates the mesh using [18]. NeuPhysics can only simulate existing NeRF objects instead of synthetic objects. Table 13 shows that our method runs at least an order of magnitude faster when simulating a bouncing ball.

D. Light Source Estimation - Implementation Details

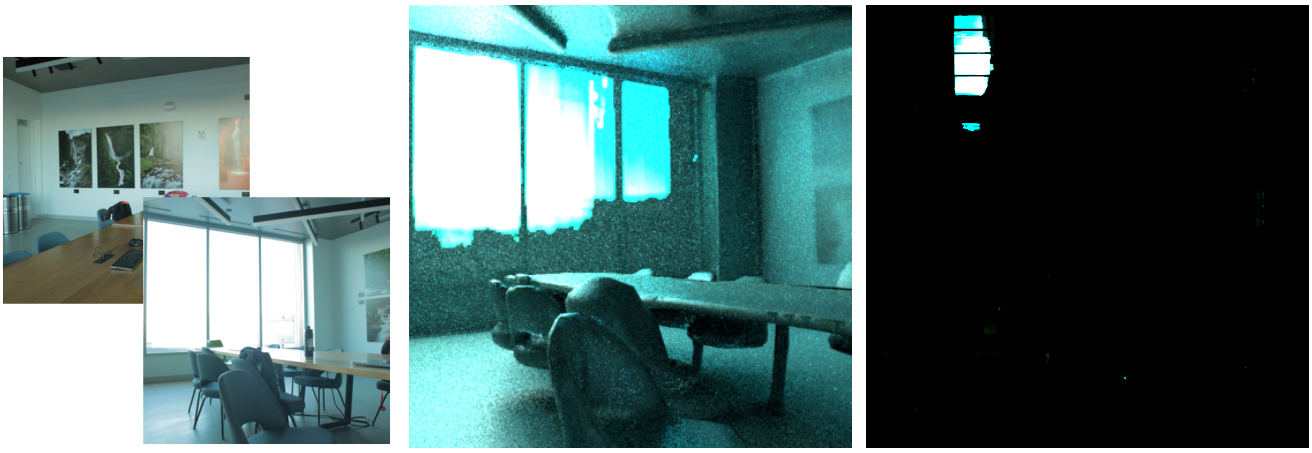
D.1. Geometry Reconstruction

Given our choice to represent light sources as area lights defined on an explicit mesh, we first need to reconstruct the geometry of the scene. We choose to use MonoSDF [86], which leverages monocular geometric priors (depth and normal estimation) in the neural implicit surface reconstruction process. Importantly, these priors especially help in reconstructing surfaces that have only a low degree of visual texture, which is very common for walls and flat tabletop surfaces. For differentiable surface rendering, having reasonably accurate geometry is critical for achieving good convergence, as noisy geometry will tend to bias the result to bad local minima. Empirically, MonoSDF achieves sufficiently accurate geometry for this purpose. Once we have an optimized signed distance field, we convert it to an explicit mesh using Marching Cubes, then UV unwrap the mesh to obtain a UV texture map, using SDFStudio [85, 67]. This step also results in a UV texture map whose values are determined by querying the underlying MonoSDF appearance model, which we use as an albedo UV texture map. The UV-unwrapped mesh is the input to the differentiable surface render, described below.

D.2. Differentiable Surface Rendering

We implement our differentiable surface rendering optimization using Mitsuba3 [27], which is built on Dr. Jit [28], a just-in-time compiler that is specialized for rendering use cases. Our inverse rendering procedure is generally based on [51], though our specific implementation details are as follows. The UV-unwrapped scene mesh is assigned a Principled BSDF material [7], as well as an emission UV texture map, which constitute the inputs to the differentiable renderer. The emission UV texture map, $T_{emission}$, with dimensions (h_{tx}, w_{tx}, c_{tx}), is the variable that we are interested in optimizing. It is initialized uniformly with near-zero values. The other BSDF parameters remain fixed (albedo texture map, specular transmission map with default value 1.0, and roughness map with default value 0.5). For rendering, we use a Path-Replay Backpropagation (PRB) integrator [72], and limit the light transport simulation to a maximum depth of 3 (2 bounces). This stands in contrast to differentiable rasterization approaches to inverse rendering, which are limited to a maximum depth of 2 (1 bounce), and are therefore unable to account for indirect lighting in the optimization. We also directly render emitters (as opposed to only emitting light onto surfaces without being directly visible).

Given a set of ground-truth 32-bit HDR images $\{I_1, \cdots, I_n\}$ with dimensions (h_{im}, w_{im}, c_{im}) and known camera poses, our primary objective is to minimize the rendering loss (mean-squared error) between the ground truth images, and images



(a) Ground Truth Images (b) Resulting image rendered with Mitsuba3 (c) Optimized Emission UV Texture Map

Figure 10: **Optimizing Emission UV Texture Map.** We optimize the (c) Emission UV Texture Map by minimizing a rendering loss objective between (a) ground truth images and (b) images rendered with Mitsuba3, a differentiable surface rendering engine.

rendered from identical camera poses. Since the variables that we are optimizing are the emission UV texture map values, we also find that applying uniform L1 regularization to the optimization variable corresponds well to the inductive bias that the majority of the scene does not emit light, and therefore the emission UV texture map should be encouraged to be sparse. For each optimization step $i \in [1, n]$ in a given optimization epoch, our objective is therefore to minimize the following, where α is a tunable hyperparameter:

$$minimize \ \frac{1}{(h_{im} * w_{im} * c_{im})} \|I_i - \hat{I}_i\|^2 + \frac{\alpha}{(h_{tx} * w_{tx} * c_{tx})} |T_{emission}|$$

In addition to the rendering loss and L1 regularization in texture space, we also find that periodically clipping low values and boosting large values in the emission texture space encourages convergence toward correct results. The motivation is that periodically during the optimization procedure (according to a tunable cadence, where we assume a default period of every 2 epochs), emission values below a given brightness threshold (also a tunable hyperparameter, with a default value of 0.2), are clipped to 0. By assuming that very dim values are not light sources, we avoid local minima. At the same time, values that are above this threshold are likely to be light sources, and therefore, we boost these values, in order to accelerate the optimization, since extreme emission values in the 32-bit linear color space may be arbitrarily larger than 1. This part of the optimization should be self-correcting: values that are boosted when they shouldn't be will then be brought down in order to satisfy the rendering loss objective. See Figure 10 for an example of the converged result of optimizing the emission texture map. Finally, once the optimization has converged, we post process the mesh by pruning off any triangular faces whose vertices all fall below a threshold, by looking up the brightness of each vertex in the emission UV texture map, according to that vertex's associated UV coordinates. This results in a greatly reduced mesh that only contains faces that should emit light onto the rest of the scene, and can therefore be used as an area light for computing the shadow pass of our Hybrid Renderer.

E. HDR Rendering Experiment Details

E.1. Data acquisition

To train the HDR NeRF, we first construct a set of 32-bit HDR images of a scene, where each image is analagous to a single LDR image in the baseline NeRF formulation. To create a single one of these HDR images, we shoot a bracketed series of LDR exposures that consists of several images (7 in our experiments) captured from a fixed camera pose. We use a single Canon 5D MKIII with a 28mm lens, and set the camera to record images in 8-bit JPEG format. Specifically, the camera is mounted to a tripod to minimize any minute difference in camera pose between each of the exposures in the bracket, and to minimize any optical differences between the images (e.g. depth of field, bloom, etc.) apart from raw exposure value, we only adjust the exposure duration (i.e. shutter speed) between images, while the aperture remains fixed. The *relative* difference in exposure between any consecutive images in the bracket is kept constant, though the magnitude of this difference depends

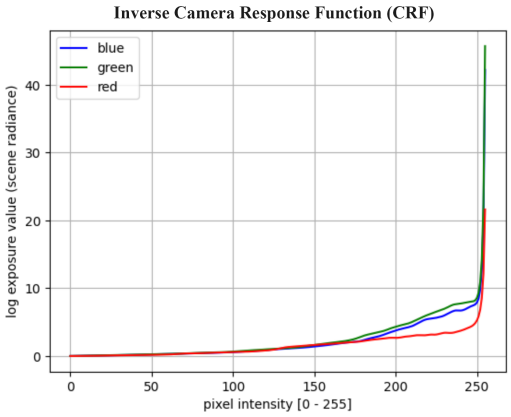


Figure 11: **Inverse Camera Response Function (CRF)** showing the recovered nonlinear function mapping from sRGB 8-bit pixel color to linear 32-bit radiance.

on the degrees of dynamic range of the scene being captured; ideally, within the range of exposures from darkest to brightest, every pixel of the image should be "well-exposed" in at least one of the images. We generally captured 7 images in the bracketed exposure, with 1-2 stops difference between each image.

E.2. Data preprocessing

After capturing the scene, we preprocess the data using OpenCV. For each bracketed exposure consisting of n images taken from the same camera pose with different exposures, we align the images to ensure the best possible quality of the HDR image result, as even slight perturbations in the camera pose may cause degradation. Then, we select a single bracketed series that is "representative" of the lighting conditions of the full set, roughly covering the upper and lower limits of the overall dynamic range. The representative bracketed series is used to recover the camera's nonlinear tone-mapping function, which we will refer to as the Camera Response Function (CRF) [Figure 11] using the method from [17].

To improve computational efficiency, we recover the CRF from only one representative image, although the linear system that must be solved could theoretically incorporate pixels from *all* of the bracketed series as constraints. By applying the recovered CRF uniformly to all series, we achieve consistent levels across all images. After applying the CRF to all bracketed series, we downsample each resulting 32-bit image by a factor of $4x$. Finally, for numerical stability during downstream training, we normalize all of the images by linearly scaling radiance values to the range of $[0, 255]$. While this may seem similar to reducing the HDR result back to LDR, all values are scaled uniformly so the *relative* differences between values are preserved, and they are **not** quantized to integer values. The resulting HDR images are then stored as EXR files.

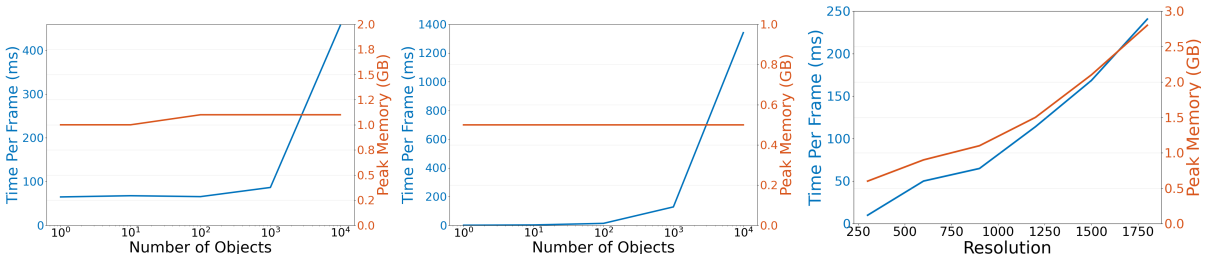
Note that for performing camera pose estimation using COLMAP, as is widely done in the NeRF literature, we only use a single 8-bit LDR image from each bracketed series. Similarly, when training an LDR model for comparison against its HDR counterpart, we use a single image from each bracketed series, where all of the singleton images have the same exposure duration.

E.3. HDR NeRF Implementation Details

Training the HDR NeRF is largely the same as training the baseline (LDR) NeRF. In fact, the Instant-NGP implementation already supports training with 32-bit HDR images in EXR format. We simply need to toggle flags 'is_hdr' in Instant-NGP's data loader and 'linear_colors' in the GUI. Therefore, training is supervised purely in (scaled) linear color space, and correspondingly, the radiance component of the neural field is defined in linear color space. Optionally, during rendering only, we may choose to apply a tonemapping function uniformly to all of the accumulated pixel radiance values to obtain the final image in sRGB color space.

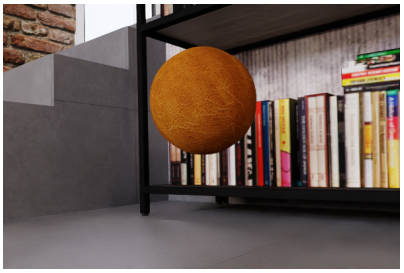
F. Details about the Rendering comparison

We conducted a user study where participants were asked to rank the realism, quality, and correctness of rendered results from three methods, including ours, on the following examples: Mirror (main paper Fig. 2), Garden, and Bicycle (main paper



(a) Rendering module (a) Simulation module (c) Rendering module

Figure 12: **Scaling the number of the inserted objects and image resolution**. When we increase the number of simulated objects from 1 to 10^4, the peak simulation memory and rendering memory stay nearly constant. The (a) simulation time scales linearly w.r.t. the number of objects, while the (b) rendering time can increase when the number of the inserted meshes is too high. The (c) rendering time and memory usage in rendering modules scales linearly w.r.t. the number of pixels.



	Rendering	Simulation
Ours	~ 0.1 s	~ 0.02 s
NeuPhysics	~ 5 s	~ 0.5 s

Figure 13: **Run time comparison with NeuPhysics [60]**. We simulate a ball that bounces around (left). Compared to NeuPhysics, our method supports hybrid NeRF and mesh rendering and runs 25x-50x faster.

(a) Using learned SDF to simulate (b) Using mesh from density fields

Figure 14: **Simulation Comparison.** Our simulation (a) using learned SDF as collision proxy geometry can correctly handle the contact between the counter and four balls. (b) Using macrching-cube mesh directly from the density field, it fails to resolve collision and two of the four balls sink into the counter.

Fig. 5). Of 15 participants, 13/15 ranked our results highest for Mirror, while 12/15 did so for Garden and 15/15 for Bicycle (see Fig. 15). Binomial tests indicate that our images are statistically superior in all three scenes, with significant results at a significance level of 0.05. The corresponding p-values were found to be $0.035, 0.007$, and 6×10^{-5}, respectively. We will include the quantitative results in the revised paper.

We also consider metrics such as FID and CLIP [5] scores. However, FID requires a ground truth distribution we cannot access. CLIP is a large language/vision model, but its scores are sensitive to prompts, and it is likely prone to misinterpreting reflections. Defining a scalable and robust quantitative metric to measure the visual fidelity of generative tasks remains an open problem.

MirrorNeRF designs a convenient capture system utilizing mirrors, but it does not focus on the rendering problem, thus does not overlap with the scope of our work.

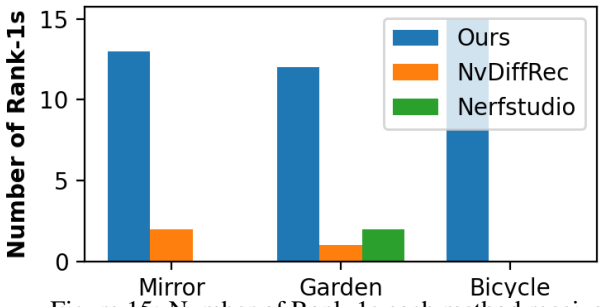


Figure 15: Number of Rank-1s each method receives.

Figure 16: Comparison between Luma AI (left) and Ours (right).

LumaAI recently released a UE plugin in April *after* the submission deadline. They provide binaries but no source code, so we only have a limited understanding of how it works. It is stated on their blog that NeRF cannot be used to simulate (otherwise, users need to add collider objects *manually*). Moreover, when we put a mirror in the Garden, shadows in the mirror reflections look strange. By contrast, our reflections and shadows are superior (see Fig. 16).