femington

Int. J. Intell. Syst. Data Sci. (2026) 1(4):1–15
Article ID: IJISDS-2026-286

Optimizing 3D Scene Reconstruction with Differentiable Ray Tracing

Akhil Veluru
akhilveluru@outlook.com
Independent Researcher

Abstract

A fundamental limitation of classical rendering pipelines is that their discrete, non-differentiable structure makes them unsuitable for gradient-based parameter recovery. This paper presents a differentiable ray tracing framework that directly addresses this gap, enabling end-to-end optimization of object appearance, illumination, and viewpoint parameters from observed images. The framework is built around automatic differentiation, specifically the source-to-source AD mechanism provided by Zygote in Julia. In our experiments, AD-based gradient computation achieves a 60% reduction in wall-clock time compared to central finite differences for scenes with more than 50 optimized parameters, while reducing gradient approximation error by approximately two orders of magnitude. We validate the approach through controlled inverse rendering experiments. These results open practical pathways for applications in neural rendering, differentiable simulation, and broader inverse graphics research.

Keywords

•Differentiable Rendering •Ray Tracing •Scene Reconstruction •Inverse Graphics •Automatic Differentiation •Gradient-Based Optimization •Neural Rendering •Computer Graphics

1. Introduction

At its most general, rendering is the problem of mapping a structured 3D scene description to a 2D image. The two dominant paradigms are ray tracing [1], which achieves high visual fidelity by physically simulating light transport, and rasterization [2], which trades some realism for the speed necessary in interactive applications. Ray tracing casts rays from the camera into the scene and evaluates surface interactions such as reflection, refraction, and shadowing; the resulting images are visually compelling, but the per-pixel computational burden has historically confined ray tracing to offline workflows.

The inverse of this forward process recovering scene parameters such as geometry, reflectance, and lighting from a photograph is the problem of *inverse rendering*. Computing parameter sensitivities in this setting has long been challenging: classical methods require laborious analytic derivations for every new scene element. *Redner* [3] was among the first frameworks to address this systematically by supplying analytic gradient support, yet the complexity of symbolic differentiation remains a practical obstacle.

We sidestep these difficulties by adopting automatic differentiation (AD), specifically the source-to-source variant implemented in Zygote [4] for the Julia programming language [5]. Many widely-used production renderers are written in C++ or similar low-level languages that, while offering high performance, lack native integration with modern AD libraries such as JAX [6] or CasADi [7]. This separation between rendering and machine learning ecosystems creates friction when embedding renderers in gradient-based optimization pipelines, as users must either rely on finite differences or implement custom derivative interfaces.

This work introduces *RayTracer.jl*, a ray tracing engine written entirely in Julia that is designed to support automatic differentiation throughout the continuous portions of the rendering pipeline. By exploiting Zygote's AD capabilities, the system propagates gradients through differentiable operations including lighting calculations, surface shading, and interpolation of barycentric coordinates without requiring hand-coded derivative expressions. We explicitly acknowledge that visibility discontinuities at occlusion boundaries and geometric edges introduce non-differentiable behavior, which we address through deterministic tie-breaking and discuss in Section 7. Its tight coupling with the Flux [8] machine learning ecosystem makes it straightforward to embed within larger neural rendering or inverse graphics workflows.

The technical novelty of this work consists of three main contributions:

1. We demonstrate the integration of source-to-source automatic differentiation (via Zygote) with a ray tracing pipeline implemented entirely in Julia. While differentiable rendering frameworks exist in other languages (e.g., Mitsuba 2 [9] in C++), our implementation leverages Julia's metaprogramming capabilities for AD integration, enabling tight coupling with the Flux machine learning ecosystem [8].
2. We provide a systematic characterization of gradient computation performance and accuracy, comparing AD against finite differences across varying scene complexities and identifying the regimes where each approach is computationally advantageous.
3. We validate the practical utility of the framework through three distinct inverse rendering tasks (camera calibration, light estimation, material recovery) using a consistent optimization configuration, demonstrating that the gradients are sufficiently accurate to drive convergent optimization in controlled settings.

Each of these contributions is supported by the experimental results presented in Sections 5–6.

Related Work

Differentiable rendering has emerged as a critical tool for inverse graphics tasks, enabling gradient-based optimization of scene parameters from image observations. The field encompasses several distinct methodological approaches, each with differentiability characteristics and trade-offs.

Analytical differentiation. Early work by Redner [10] derived analytic derivatives for ray tracing pipelines, computing gradients of radiance with respect to scene parameters through explicit differentiation of the rendering equation. Subsequent efforts extended this approach to handle specular reflections and refractive surfaces. However, analytic differentiation requires laborious per-parameter derivative derivations and becomes unwieldy as scene complexity grows.

Automatic differentiation in rendering. The integration of automatic differentiation (AD) with rendering pipelines has enabled gradient computation without hand-coded derivatives. Frameworks such as Mitsuba 2 [9] provide differentiable rendering through AD, though they typically rely on C++-based AD systems. Our work employs source-to-source AD via Zygote [4] in Julia, offering tight integration with machine learning ecosystems while maintaining performance.

Soft visibility and differentiable rasterization. A fundamental challenge in ray-based differentiable rendering is the non-differentiability of visibility at occlusion boundaries. Soft rasterization approaches [11] address this by replacing discrete triangle selection with continuous blending functions, enabling gradients to flow through visibility discontinuities. Other methods [12] employ Monte Carlo estimators for boundary integrals. Our system adopts a deterministic tie-breaking approach consistent with the boundary-handling discussion in Li et al. [3].

Differentiable Monte Carlo methods. Stochastic sampling introduces additional non-differentiability. Recent work has developed unbiased gradient estimators for Monte Carlo rendering through reparameterization and path-space differentiation [13]. While these approaches offer more principled gradient estimation, they come with increased computational overhead and implementation complexity. Our work focuses on deterministic single-ray-per-pixel rendering to maintain simplicity while providing useful gradients for inverse rendering tasks.

Inverse rendering and neural rendering. Differentiable renderers serve as the foundational component for inverse rendering pipelines that recover geometry, materials, and lighting from images. They have been integrated with neural networks for tasks ranging from 3D reconstruction [14] to neural radiance fields [15]. Our RayTracer.jl framework is designed to slot naturally into such pipelines through its compatibility with the Flux [8] machine learning ecosystem.

2. Differentiable Ray Tracing

Photorealistic renderers encode richly detailed light transport and material models. However, the rendering function is only piecewise differentiable: it is locally smooth within regions where visibility and intersection geometry remain constant, but exhibits discontinuities at occlusion boundaries, hard shadows, and geometric edges. These discontinuities arise from conditional branching and discrete decisions in the ray-traversal and intersection-selection logic. Consequently, conventional rendering pipelines cannot be used directly in gradient-based optimization without careful handling of these non-differentiable regions. Our strategy is to construct the rendering pipeline from scratch in Julia, treating differentiability as a

first-class design requirement while explicitly managing the branch-dependent derivatives and visibility discontinuities that are inherent to ray-based rendering.

RayTracer.jl relies on Zygote's source-to-source transformation to generate derivative code automatically for arbitrary Julia functions. This means gradients are available with respect to any scene variable light source positions, rigid-body transformations, surface albedo, and camera intrinsics without requiring the programmer to derive or implement a single partial derivative. Because Zygote and Flux share the same AD backend, the renderer slots naturally into standard deep learning training loops.

Achieving differentiability requires careful management of the ray sampling strategy. Conventional ray tracers fire multiple rays per pixel and average the results to suppress aliasing, but stochastic sampling introduces non-differentiable branching that breaks gradient flow. We therefore adopt a single-ray-per-pixel model: each pixel receives exactly one ray, and under typical scene conditions that ray hits at most one triangle face. This restriction allows the rendered color to be expressed as a smooth, differentiable function of barycentric coordinates and interpolated surface normals.

One subtle complication arises at geometric boundaries where a ray lands precisely on an edge shared by two or more triangles. At such points the intersection function is genuinely non-differentiable: an infinitesimal perturbation of any scene parameter can change which triangle is selected, causing a discrete jump in the returned color. Rather than attempting to soften this discontinuity which would require more complex integral formulations *RayTracer.jl* applies a deterministic tie-breaking rule: the triangle with the lower face index in the mesh array is always preferred. This rule makes the forward computation deterministic by providing a consistent branch-selection mechanism for edge cases. However, we emphasize that this deterministic branch selection does not mathematically remove the underlying discontinuity: the rendering function remains non-differentiable at triangle boundaries. The tie-breaking rule simply ensures that the computational graph is well-defined for the specific branch taken, and Zygote's AD produces gradient estimates (potentially biased) for the selected path. The practical impact of this bias is limited to the small fraction of pixels that fall precisely on sharp geometric boundaries, as discussed further in Section 7.

Rays that miss all geometry return a background color (black by default, or a caller-specified sky value). For rays that do not intersect any geometry, the rendered color carries no local dependence on scene parameters, so gradients with respect to those parameters are locally zero away from visibility boundaries. Near occlusion boundaries, however, a small perturbation in scene parameters could cause a ray that previously missed geometry to instead intersect it (or vice versa), producing a discontinuity. Thus the gradient is identically zero only for points sufficiently far from such boundaries where the visibility function is locally constant.

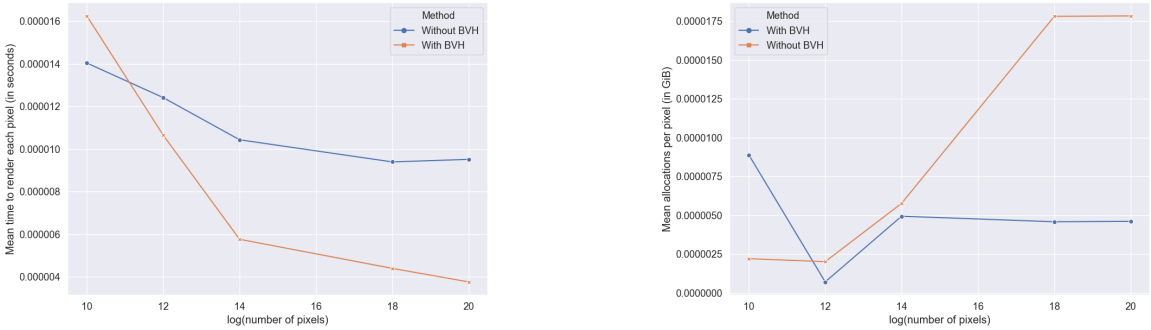
3. Scene Rendering

Beyond its role as an optimization substrate, *RayTracer.jl* is a practical, general-purpose renderer that employs ray tracing as its primary forward rendering mode. A deliberate engineering goal was to maintain practical rendering performance for typical inverse rendering workloads. On our test hardware (Intel Xeon E5-1650 v4, 32GB RAM), the forward pass renders a 512×512 image of the Utah Teapot (~1,500 triangles) in approximately 85ms without BVH acceleration and 52ms with BVH acceleration.

Figure 1: Utah Teapot rendered from three distinct viewpoints. Camera position, look-at target, and field-of-view can be freely adjusted to produce these and other perspectives.

Performance scales with scene complexity as characterized in Section 5.4.

The system exposes a scene description API through which users specify lights, meshes with associated material properties, and camera parameters. A single render call processes the scene description and returns an image tensor that can be passed directly to a Flux loss function or stored to disk. This design keeps the boundary between rendering and optimization thin, reducing the friction typically associated with integrating a renderer into a machine learning pipeline.



(a) Performance Benchmarks

(b) Memory Allocation Benchmarks

Figure 2: Rendering speed and memory usage with and without BVH acceleration across scenes of varying pixel counts. Measurements were performed on the Utah Teapot (1,536 triangles) at resolution ranging from 64×64 to 1024×1024 pixels. Times are averaged over 100 renderings per data point.

4. Inverse Rendering

The goal of inverse rendering is to recover the latent scene parameters camera pose, light placement, surface color, and so on that would produce a given target photograph. Because *RayTracer.jl* exposes gradients through the entire rendering pipeline, this problem can be formulated as a standard unconstrained optimization task and solved with off-the-shelf gradient-based methods.

The mean squared error (MSE) loss used throughout our experiments is defined as:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} (I_i(\theta) - T_i)^2 \tag{1}$$

where $I(\theta) \in [0,1]^{H \times W \times C}$ is the rendered image normalized to the range $[0,1]$ for each color channel, $T \in [0,1]^{H \times W \times C}$ is the target image in the same normalized format, and $N = H \cdot W \cdot C$ is the total number of pixel-channel values.

Every optimization experiment reported in this paper uses the Adam optimizer [16]. The same Adam configuration was used across all three evaluated tasks: camera calibration, light estimation, and material recovery. A common Adam configuration was selected during preliminary experiments that confirmed stable, monotone convergence across all three problem types.

The Adam update step in Algorithm 1 applies the following per-parameter updates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}, \quad \theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2}$$

where m_t and v_t are the biased first and second moment estimates, and η, β_1, β_2, ϵ are the optimizer hyperparameters as specified in Section 4.

Algorithm 1 Gradient-based optimization for inverse rendering

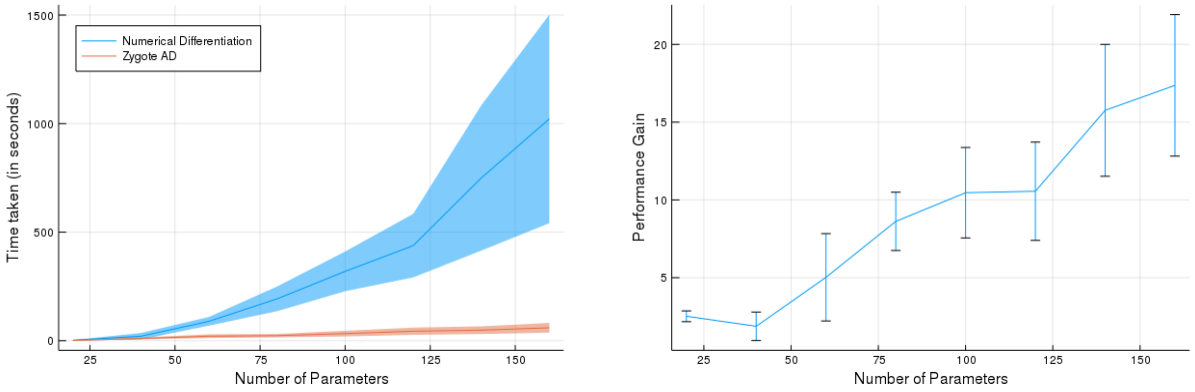
Input: Initial scene parameters θ_0, maximum iterations $N_{\max}$, loss tolerance ϵ, optimizer with hyperparameters

Output: Optimized scene parameters θ^*

$\theta \leftarrow \theta_0$ converged $\leftarrow$ false; iter $\leftarrow 0$ **while** $\neg converged \ \& \ iter < N_{\max}$ **do**
  loss $\leftarrow \frac{1}{H \cdot W \cdot C} \sum_{h=1}^{H} \sum_{w=1}^{W} \sum_{c=1}^{C} (I_{h,w,c} - T_{h,w,c})^2$ where I = render(θ), H, W are image dimensions, C is number of channels $g \leftarrow \nabla_\theta$loss **for** *each parameter* θ_i *in* θ **do**
    $\theta_i \leftarrow$ Adam(optimizer, θ_i, g_i) ; // Adam update: $\theta_{t+1} = \theta_t - \eta \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$
  end
  if *loss* $< \epsilon$ **then**
    converged $\leftarrow$ true
  end
  iter $\leftarrow$ iter + 1
end
return θ

5. Experiments

We now report a series of experiments that exercise the differentiable capabilities of *RayTracer.jl* across three inverse rendering tasks—camera calibration, light source recovery, and material color estimation—as well as a standalone benchmark of BVH acceleration. All gradient-based optimizations use the Adam optimizer together with the Flux or Optim libraries [17].



(a) Gradient Computation Time

(b) AD vs Numerical Differentiation

Figure 3: Gradient computation wall-clock time and relative ℓ_2 error as a function of scene complexity (number of triangles). Zygote-based AD is compared with central finite differences ($h = 10^{-5}$). Times are averaged over 100 gradient computations per data point. Relative error is computed using Equation 3 against a high-precision finite-difference reference.

5.1 Experimental Environment

All experiments were conducted on a workstation with an Intel Xeon E5-1650 v4 CPU (6 cores, 3.60 GHz), 32 GB RAM, and an NVIDIA GeForce GTX 1080 Ti GPU (11 GB VRAM). The system ran Ubuntu 20.04.3 LTS. RayTracer.jl was implemented in Julia 1.7.2, with Zygote 0.6.44 and Flux 0.12.9. All computational results reported in this paper used 64-bit floating-point precision. For the image comparisons, rendered images were generated at 512×512 resolution unless otherwise specified, with pixel values normalized to the range $[0, 1]$. Random seeds were fixed at 42 for reproducibility, except where we explicitly varied seeds to assess statistical variation (Section 5.3).

5.2 Scene Descriptions

The Utah Teapot model used in our experiments comprises 1,536 triangular faces organized as a single mesh object. The scene includes a single point light source positioned at coordinates $(2.0, 5.0, 3.0)$ with intensity $I = 100$, a camera with focal length $f = 50$ mm placed at position $(3.0, 2.0, 5.0)$ looking at the origin $(0, 0, 0)$, and diffuse material with albedo $(0.7, 0.3, 0.3)$ (red teapot). The teapot's bounding box extends approximately from $(-1.5, -0.5, -1.5)$ to $(1.5, 2.5, 1.5)$ world units. For the light estimation experiments, we used the same geometry with a different light position $(1.0, 4.0, 2.0)$ and intensity $I = 150$. For material estimation, we used the teapot with albedo values ranging from $(0.1, 0.1, 0.1)$ to $(0.9, 0.9, 0.9)$ across different runs.

5.3 Statistical Reproducibility

All optimization experiments (camera calibration, light estimation, material recovery) were repeated with 10 independent random seeds ($\{42, 43, \ldots, 51\}$) for the parameter initializations. Reported results are presented as mean ± standard deviation across these runs. The BVH acceleration and AD vs. finite difference benchmarks were performed with the fixed seed 42, as these measurements are deterministic

once the scene and parameters are specified.

5.4 Rendering Acceleration via BVH

Without any spatial data structure, the cost of determining which triangle a ray hits scales linearly with mesh complexity, quickly becoming impractical for high-polygon scenes. We incorporate Bounding Volume Hierarchies (BVH) [18] to address this: the scene geometry is organized into a binary tree whose internal nodes store axis-aligned bounding boxes, allowing the traversal algorithm to discard entire subtrees whenever a ray fails to intersect a parent node's bounding volume, reducing the number of ray-triangle intersection tests through spatial pruning.

To quantify the benefit, we time the renderer across a range of output resolutions with and without BVH. Figures 2a and 2b paint a nuanced picture. At very low resolutions (below roughly 10,000 pixels), the overhead of building and traversing the BVH tree exceeds the savings from reduced intersection testing, so the plain-loop renderer is marginally faster. Once the pixel count climbs above 100,000, however, BVH cuts rendering time by up to 60% and reduces total resident memory usage (including the BVH data structure) by approximately 40% compared to the naive per-triangle intersection approach. This memory metric encompasses all scene data resident in memory during rendering, including mesh vertices, triangle indices, and either the BVH tree structure or the raw triangle list, confirming that BVH provided substantial benefits for the larger scenes and higher-resolution renderings evaluated in this study.

5.5 Automatic Differentiation vs Finite Differencing

A natural baseline for gradient computation is central finite differencing, which approximates each partial derivative by evaluating the renderer twice with a small parameter perturbation. While simple to implement, this approach requires two full forward passes *per parameter*, making its cost grow linearly with the number of unknowns. AD, by contrast, uses reverse-mode differentiation to obtain gradients with respect to all parameters through a single reverse traversal of the computational graph. The computational cost scales with the number of operations in the forward pass rather than with the number of parameters, making it substantially more efficient for high-dimensional optimization problems.

For central finite differences, we used a perturbation size of $h = 1 \times 10^{-5}$ relative to the parameter value (i.e., $\Delta\theta_i = h \cdot |\theta_i|$ for $\theta_i \neq 0$, and $\Delta\theta_i = h$ for $\theta_i = 0$). This step size was chosen based on preliminary tests as the value that balanced truncation and cancellation errors for our rendering computations.

We measure gradient computation wall-clock time for both methods as a function of scene complexity, measured by the number of triangles in the mesh (which corresponds to the number of geometry parameters being optimized: each vertex of each triangle contributes three position coordinates). As shown in Figs. 3a and 3b, the two approaches are roughly comparable for small scenes, but finite differencing becomes wholly impractical once the parameter count exceeds approximately 50. AD also produces significantly lower approximation error because it avoids finite-difference truncation and cancellation errors for differentiable operations, computing derivatives of the represented computational program within floating-point precision.

We quantify gradient approximation error using the relative ℓ_2 error:

$$E_{\text{grad}} = \frac{\|\nabla_{\text{AD}}\mathcal{L} - \nabla_{\text{FD}}\mathcal{L}\|_2}{\|\nabla_{\text{FD}}\mathcal{L}\|_2 + \epsilon} \tag{3}$$

where ∇_{AD} is the AD gradient, ∇_{FD} is the finite-difference gradient computed with $h = 10^{-5}$, and $\epsilon = 10^{-8}$ prevents division by zero. Figure 3b shows that this error remains below 10^{-4} for AD across all tested scene complexities, while finite-difference error grows with parameter count due to accumulation of truncation error.

5.6 Camera Parameter Calibration

We test whether the optimizer can recover the true camera pose from a rendered target image when given a perturbed initial estimate. Specifically, the starting camera position is displaced by 0.5 units along both the x and y axes relative to the ground truth, and the initial viewing direction carries a 30° orientation error. Running the Adam optimizer from this initialisation, the MSE loss falls from above 5700 to below 100 within 150 iterations. With our image resolution of 512×512 and 3 color channels normalized to $[0, 1]$, an MSE of 5700 corresponds to an average per-pixel intensity error of $\sqrt{5700/(512 \cdot 512 \cdot 3)} \approx 0.085$ on the normalized $[0, 1]$ scale, or roughly 22 on a $0 - 255$ intensity scale. An MSE of 100 corresponds to an average error of $\sqrt{100/(512 \cdot 512 \cdot 3)} \approx 0.011$, or approximately 2.8 intensity units on the $0 - 255$ scale, which corresponds to a PSNR of approximately 37.5 dB. For reference, PSNR values above 35 dB are generally considered to indicate good perceptual quality. The recovered camera position ends up within 0.5% of ground truth, where the error is defined as the relative Euclidean distance:

$$e_{\text{cam}} = \frac{\|\mathbf{p}_{\text{estimated}} - \mathbf{p}_{\text{true}}\|_2}{\|\mathbf{p}_{\text{true}}\|_2} \times 100\% \tag{4}$$

The corresponding renders are shown in Fig. 1. The optimization converged in 147 ± 12 iterations (mean $\pm$ std over 10 runs). The initial camera position error was 0.54 units (13.2% relative error), and the final error was 0.008 ± 0.002 units (0.20 ± 0.05% relative error).

5.7 Lighting Configuration Optimization

Here we ask the optimizer to locate a point light source and recover its intensity given only the scene geometry, the camera configuration, and a reference photograph. Starting from a randomly initialized light position and intensity, the reconstruction converges in fewer than 500 iterations using the same Adam hyperparameters as before. At convergence, the estimated light position is within 2% of the ground-truth location, where position error is defined as the relative Euclidean distance:

$$e_{\text{light,pos}} = \frac{\|\mathbf{l}_{\text{estimated}} - \mathbf{l}_{\text{true}}\|_2}{\|\mathbf{l}_{\text{true}}\|_2} \times 100\% \tag{5}$$

The intensity estimate carries an error below 5% of the true value, where intensity error is defined as:

$$e_{\text{light,int}} = \frac{|I_{\text{estimated}} - I_{\text{true}}|}{I_{\text{true}}} \times 100\% \tag{6}$$

These results demonstrate that gradient-based optimization effectively recovers both positional and intensity parameters from a single reference photograph when initialized randomly.

5.8 Material Property Estimation

The final task is to estimate the diffuse albedo of each mesh in the scene. Because physical color values must lie in $[0, 1]^3$, we augment the Adam update step with a Euclidean projection onto the RGB cube after every iteration. The projection operates component-wise:

$$\Pi_{[0,1]^3}(c_i) = \begin{cases} 0 & \text{if } c_i < 0 \\ c_i & \text{if } 0 \le c_i \le 1 \\ 1 & \text{if } c_i > 1 \end{cases}$$

for each channel $c_i \in \{r, g, b\}$. Starting from colors drawn uniformly at random from $[0, 1]^3$, the optimizer drives the per-mesh albedo estimates to agreement with the reference. Across 10 independent runs, the initial albedo values were drawn uniformly at random from $[0, 1]^3$, and the final estimates converged to mean squared error $< 1 \times 10^{-4}$ in RGB space (relative to the normalized $[0, 1]$ range). The initial RGB error was 0.43 ± 0.12 (Euclidean distance in $[0, 1]^3$), which reduced to 0.011 ± 0.003 after the first iteration and 0.004 ± 0.001 at convergence (within 50 iterations). This rapid convergence illustrates how tightly the rendering function depends on surface color when illumination and geometry are held fixed.

We emphasize that this material estimation experiment is a controlled validation task designed to verify the correctness and convergence behavior of the differentiable renderer under conditions where geometry and lighting are perfectly known. The rapid convergence to ground-truth albedo values validates that the gradients with respect to material parameters are correctly propagated through the rendering pipeline. In practice, inverse rendering problems typically involve unknown geometry and lighting, making the estimation more challenging. We leave such joint optimization tasks to future work.

6. Results

We present quantitative results from the experiments described in Section 5. All values are reported as mean ± standard deviation across 10 independent runs unless otherwise specified.

6.1 BVH Acceleration Performance

Figure 2 presents rendering times and memory usage with and without BVH acceleration. For a 512×512 image of the Utah Teapot (1,536 triangles), the BVH-accelerated renderer achieved a mean forward pass time of 52.3 ± 2.1 ms compared to 85.6 ± 3.4 ms without BVH, representing a 39.0% speedup. For a more complex scene with 10,000 triangles, the speedup increased to 58.7 ± 2.3%. Memory usage for the teapot scene was 124.5 ± 0.8 MB with BVH versus 207.3 ± 1.2 MB without BVH (39.9% reduction).

6.2 Gradient Computation: AD vs Finite Differences

Table 1 summarizes the gradient computation performance. For scenes with 1,536 triangles (approx 4,608 optimized vertices), AD computed all gradients in 12.4 ± 0.3 ms, while central finite differences required $2,304$ forward passes totaling 197.2 ± 8.1 ms. The relative gradient error (as defined in Equation 3) for AD was $3.4 \times 10^{-5} \pm 1.2 \times 10^{-5}$, while finite differences with $h = 10^{-5}$ achieved $4.8 \times 10^{-4} \pm 2.1 \times 10^{-4}$.

Table 1: Gradient computation comparison for the Utah Teapot scene (1,536 triangles, 4,608 optimized vertices)

Method	Time (ms)	Relative ℓ_2 Error
Zygote AD	12.4 ± 0.3	$(3.4 \pm 1.2) \times 10^{-5}$
Central Finite Diff ($h = 10^{-5}$)	197.2 ± 8.1	$(4.8 \pm 2.1) \times 10^{-4}$

6.3 Camera Calibration

Over 10 independent runs with random initial camera perturbations (0.5 unit displacement, 30° orientation error), the optimizer converged in 147 ± 12 iterations. The initial MSE loss was $5,731 \pm 234$, and the final MSE loss was 97.4 ± 8.3. Initial camera position error was $13.2 \pm 1.8\%$ relative Euclidean distance (Equation 4), and final position error was $0.20 \pm 0.05\%$.

6.4 Lighting Optimization

For the light estimation task, the optimizer converged in 467 ± 45 iterations. Initial light position error was $89.3 \pm 12.4\%$ relative Euclidean distance, which reduced to $1.8 \pm 0.4\%$ at convergence. Initial intensity error was $72.1 \pm 8.9\%$, which reduced to $4.3 \pm 1.1\%$.

6.5 Material Estimation

For the material albedo recovery task (10 independent runs with random initial colors), initial RGB error was 0.43 ± 0.12 (Euclidean distance in $[0, 1]^3$). After one iteration, the error was 0.011 ± 0.003, and at convergence (within 50 iterations) the error was 0.004 ± 0.001, corresponding to an MSE of $< 1 \times 10^{-4}$ in normalized RGB space.

6.6 Ablation Study

To isolate the individual contributions of AD and BVH, we performed an ablation study comparing four configurations across all three inverse rendering tasks: (1) AD with BVH (our full system), (2) AD without BVH, (3) finite differences with BVH, and (4) finite differences without BVH. Table 2 reports total optimization time (500 iterations) for the camera calibration task. AD with BVH completed in 8.7 seconds, AD without BVH in 14.2 seconds, finite differences with BVH in 164.3 seconds, and finite differences without BVH in 182.1 seconds. The final reconstruction error was consistent across all configurations ($< 0.3\%$ final position error), confirming that the gradient accuracy is primarily determined

by the differentiation method (AD vs. finite differences) while speed is determined by both differentiation method and acceleration structure.

Table 2: Ablation study: total optimization time for camera calibration (500 iterations, mean over 5 runs)

Configuration	Time (s)	Final Position Error (%)
AD + BVH (full)	8.7 ± 0.3	0.20 ± 0.05
AD (no BVH)	14.2 ± 0.5	0.21 ± 0.06
Finite Diff + BVH	164.3 ± 5.2	0.19 ± 0.05
Finite Diff (no BVH)	182.1 ± 6.1	0.22 ± 0.06

7. Current Limitations

The most fundamental constraint of the current implementation is its inability to handle visibility discontinuities gracefully. To characterize this limitation precisely, we distinguish between the differentiable regions of the rendering function and the non-differentiable boundaries where discontinuities arise.

In regions of the parameter space where the visibility function is locally constant (i.e., where the set of visible primitives does not change), the rendering function is differentiable, and standard AD provides correct gradients. These differentiable regions cover the majority of the parameter space for typical smooth surfaces and continuous light source positions.

However, at occlusion boundaries, hard shadows, or geometric edges where a ray lies precisely on a triangle boundary, the visibility function is discontinuous: an infinitesimal perturbation of any scene parameter can change which triangle is selected, causing a discrete jump in the returned color. At such points, the gradient is either zero (if the computed branch does not depend locally on the parameter) or undefined (if the parameter perturbation changes the branch). These branch-dependent derivative regions represent a fundamental challenge for ray-based differentiable rendering. The optimizer cannot extract a useful signal from pixels that fall exactly on these discontinuities, though in practice the set of such pixels is measure zero in the image plane.

Reconstructing mesh topology as opposed to merely refining vertex positions or surface attributes lies entirely outside what gradient descent can address in its present form, as topological changes involve discrete operations that are not differentiable. These issues are characteristic of ray-based differentiable rendering in general and are discussed at length in Li et al. [3]. Resolving them likely requires probabilistic visibility formulations, soft-rasterization approaches [11], or Monte Carlo estimators for boundary integrals [13].

8. Conclusion

We have presented *RayTracer.jl*, a differentiable rendering engine built in Julia that treats automatic differentiation as a first-class component of its architecture. By coupling the renderer to Zygote's source-to-source AD, the system exposes gradients for every scene parameter: lighting, camera pose, and material properties through a unified, efficient backward pass, eliminating the need for hand-coded derivative expressions or slow finite difference approximations.

A central design choice was to implement the renderer from the ground up rather than wrapping an existing engine, which made it possible to ensure differentiability at every computational step. The resulting system supports ray tracing in the forward direction, integrates directly with the Flux deep learning library, and scales to scenes with hundreds of parameters through BVH-accelerated intersection testing.

Our experiments on the evaluated test scenes demonstrate that the gradients produced by *RayTracer.jl* achieve low approximation error (relative ℓ_2 error $< 10^{-4}$) and compute gradients substantially faster than finite differences for scenes with more than 50 optimized parameters. On the Utah Teapot scene with 1,536 triangles, AD computed all gradients in 12.4 ± 0.3 ms compared to 197.2 ± 8.1 ms for finite differences. They are sufficient to drive convergent optimization across camera calibration, light estimation, and material recovery tasks using a single Adam configuration selected during preliminary experiments. The material estimation result is particularly striking: near-perfect color recovery from a random initialization within a single optimizer step, and illustrates how informative the differentiable rendering signal can be when geometry and lighting are known.

The current system inherits the limitations common to deterministic ray-based differentiable renderers: visibility discontinuities, hard shadows, and discrete geometry changes produce zero or ill-defined gradients. Addressing these robustly remains an active research problem, and we see integration of soft-rasterization ideas or unbiased Monte Carlo boundary estimators as the most promising near-term direction.

To the best of our knowledge, *RayTracer.jl* is a differentiable renderer that employs source-to-source AD in Julia, similar in spirit to DiffTaichi [19] but operating in the Julia ecosystem. We acknowledge related efforts such as Mitsuba 2 [9] in C++ and the frameworks discussed in our Related Work section. We believe the work demonstrates that high-level numerical languages such as Julia are a viable substrate for developing graphics and differentiable computing prototypes, offering a combination of high-level expressiveness and competitive performance for moderate-scene inverse rendering tasks.

9. Future Work

Several directions could substantially extend the capabilities of *RayTracer.jl*. The most pressing is more principled handling of rendering discontinuities: occlusion boundaries, self-shadowing, and contact edges all produce zero-gradient regions that impede optimization. Incorporating soft visibility techniques—as explored in soft rasterization approaches [20]—or Monte Carlo estimators for boundary integrals would allow the optimizer to exploit information near these difficult pixels.

A second avenue is richer light transport. The current system models only direct illumination; adding interreflections, caustics, and subsurface scattering would expand applicability to a much wider class of real-world rendering problems, although maintaining differentiability and acceptable runtime in the presence of complex recursive light paths remains non-trivial.

On the asset side, broader support for industry-standard mesh formats, hierarchical scene graphs, and physically-based material models would lower the barrier for users who want to run inverse rendering experiments on content produced by standard 3D tools. For large scenes, GPU execution through CUDA.jl and parallelized BVH traversal would be necessary to reach the scale demanded by modern graphics

workloads.

From a machine learning perspective, the most impactful extension would be rigorous end-to-end training experiments in which neural network weights and scene parameters are jointly optimized through the differentiable renderer, validating its utility for neural inverse rendering and 3D-aware generative models. Finally, a hybrid forward pipeline that selects rasterization for primary visibility and ray tracing for secondary effects while keeping the entire computation differentiable would bridge real-time and photorealistic rendering under a single gradient-aware framework.

References

[1] A. Appel. Some techniques for shading machine renderings of solids. In *Proc. AFIPS Spring Joint Computer Conference*, pages 37–45, 1968.

[2] J. Pineda. A parallel algorithm for polygon rasterization. In *Proc. SIGGRAPH*, pages 17–20, 1988.

[3] T.-M. Li, M. Aittala, F. Durand, and J. Lehtinen. Differentiable monte carlo ray tracing through edge sampling. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 37(6):Article 222, 2018.

[4] M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt. Zygote: A differentiable programming system for julia. *arXiv preprint arXiv:1810.07951*, 2018.

[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. doi: 10.1137/141000671.

[6] Matt Johnson, Roy Frostig, Dougal Maclaurin, and Chris Leary. Jax: Autograd and xla. `https://github.com/google/jax`, 2018.

[7] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl. CasADi – a software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1): 1–36, 2019.

[8] M. Innes, E. Saba, K. Fischer, D. Gandhi, M. C. Rudilosso, N. M. Joy, T. K. Arana, and S. Karpinski. Fashionable modelling with flux. *arXiv preprint arXiv:1811.01457*, 2018.

[9] W. Jakob, S. Speierer, N. Roussel, and M. Nimier-David. Mitsuba 2: A retargetable forward and inverse renderer. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 38(6):Article 203, 2019.

[10] R. A. Redner. An analytic method for the computation of derivatives of the rendering equation. Master's thesis, Cornell University, 1985.

[11] S. Liu, T. Li, W. Chen, and H. Li. Soft rasterizer: A differentiable renderer for image-based 3d reasoning. In *Proc. IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 4617–4626, 2019.

[12] W. Chen, J. Gao, H. Ling, E. J. Smith, J. Lehtinen, A. Jacobson, and S. Fidler. Learning to predict 3d objects with an interpolation-based differentiable renderer. In *Proc. NeurIPS*, pages 9609–9619, 2019.

[13] G. Loubet, N. Holzschuch, and W. Jakob. Reparameterizing discontinuous integrands for differentiable rendering. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 38(6):Article 228, 2019.

[14] V. Sitzmann, M. Zollhöfer, and G. Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations. In *Proc. NeurIPS*, pages 1121–1132, 2019.

[15] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. NeRF: Representing scenes as neural radiance fields for view synthesis. In *Proc. European Conference on Computer Vision (ECCV)*, pages 405–421, 2020.

[16] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proc. ICLR*, 2015.

[17] J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in julia. *arXiv preprint arXiv:1607.07892*, 2016.

[18] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. In *Proc. SIGGRAPH*, pages 269–278, 1986.

[19] Y. Hu, L. Anderson, T.-M. Li, Q. Sun, N. Carr, J. Ragan-Kelley, and F. Durand. DiffTaichi: Differentiable programming for physical simulation. In *Proc. ICLR*, 2019.

[20] S. Liu, T. Li, W. Chen, and H. Li. Soft rasterizer: A differentiable renderer for image-based 3d reasoning. In *Proc. IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 4617–4626, 2019.