Automatic Sampling for Discontinuities in Differentiable Shaders

YASH BELHE, University of California San Diego, USA
ISHIT MEHTA, University of California San Diego, USA
WESLEY CHANG, University of California San Diego, USA
ILIYAN GEORGIEV, Adobe Research, UK
MICHAËL GHARBI, Reve, USA
RAVI RAMAMOORTHI, University of California San Diego, USA
TZU-MAO LI, University of California San Diego, USA

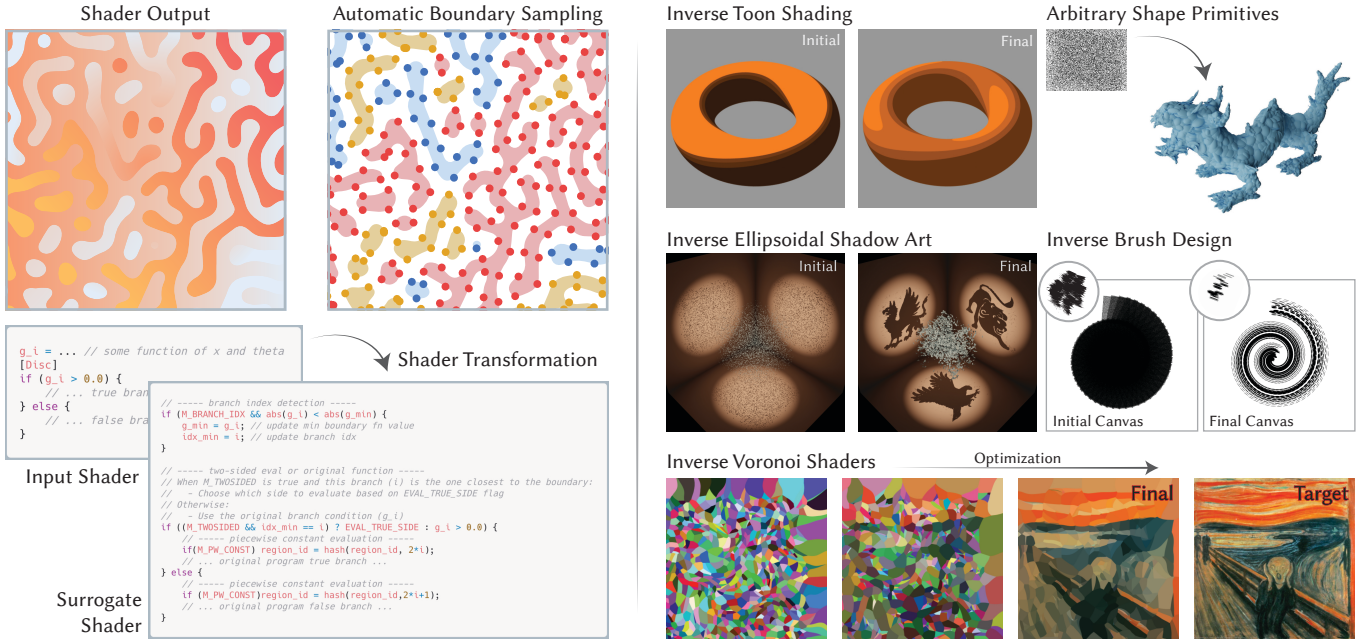


Fig. 1. We introduce automatic boundary sampling for discontinuities in differentiable shaders. Given a shader that renders a piecewise-continuous output, we first transform it into a surrogate, piecewise-constant shader (shown on the left). This transformation enables our method to sample jump discontinuities by evaluating the shader along randomly selected line segments. We call this approach *segment snapping*; it removes the need for specialized boundary-sampling routines, which are tedious to implement and often ill-defined. Our method unlocks a variety of new applications (shown on the right).

We present a novel method to differentiate integrals of discontinuous functions, which are common in inverse graphics, computer vision, and machine learning applications. Previous methods either require specialized routines to sample the discontinuous boundaries of predetermined primitives, or use reparameterization techniques that suffer from high variance. In contrast, our method handles general discontinuous functions, expressed as shader programs, without requiring manually specified boundary sampling routines. We achieve this through a program transformation that converts discontinuous functions into piecewise constant ones, enabling efficient boundary sampling through a novel segment snapping technique, and accurate derivatives at the boundary by simply comparing values on both sides of the discontinuity. Our method handles both explicit boundaries (polygons, ellipses, Bézier curves) and implicit ones (neural networks, noise-based functions, swept surfaces). We demonstrate that our system supports a wide range of applications, including painterly rendering, raster image fitting, constructive solid geometry, swept surfaces, mosaicing, and ray marching.

CCS Concepts: • **Computing methodologies** → **Rendering**; • **Mathematics of computing** → **Differential calculus**.

ACM Reference Format:
Yash Belhe, Ishit Mehta, Wesley Chang, Iliyan Georgiev, Michaël Gharbi, Ravi Ramamoorthi, and Tzu-Mao Li. 2025. Automatic Sampling for Discontinuities in Differentiable Shaders. *ACM Trans. Graph.* 44, 6, Article 209 (December 2025), 19 pages. https://doi.org/10.1145/3763291

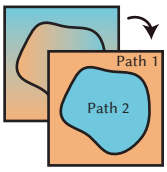
Authors' addresses: Yash Belhe, ybelhe@ucsd.edu, University of California San Diego, USA; Ishit Mehta, ibmehta@ucsd.edu, University of California San Diego, USA; Wesley Chang, wec022@ucsd.edu, University of California San Diego, USA; Iliyan Georgiev, Adobe Research, UK, igeorgiev@adobe.com; Michaël Gharbi, Reve, USA, michael@reve.art; Ravi Ramamoorthi, ravir@ucsd.edu, University of California San Diego, USA; Tzu-Mao Li, tzli@ucsd.edu, University of California San Diego, USA.

1 INTRODUCTION

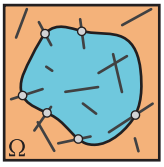
A wide class of inverse problems in graphics, vision, and machine learning can be solved by computing the derivative of integrals: $\nabla_\theta \int_\Omega f(x,\theta)\mathrm{d}x$. Traditional automatic differentiation methods can produce incorrect results when the integrand f is discontinuous, since they ignore the Dirac delta distribution that arises from differentiating step discontinuities, which needs to be integrated. We call this integral over Dirac delta a *boundary integral* since it integrates over the boundary of the discontinuities. In this work, we propose a new method for numerically estimating the derivative of integrals that enables us to derive an automatic differentiation method that is applicable to an extremely wide range of problems (Fig. 1).

The boundary integral requires a different numerical estimator than the original integral and can be challenging to evaluate. For sampling, the Monte Carlo estimator needs to be aware of the decision boundaries. The differentiable rendering literature has extensively studied Monte Carlo estimators for computing derivatives of discontinuous functions under integral sign [Li et al. 2018, 2020; Bangaru et al. 2020; Loubet et al. 2019; Zhang et al. 2020]. However, these estimators are specialized to particular rendering problems and are not straightforward to generalize outside of the original problem setting. Even handling programmable shaders with discontinuities in existing differentiable renderers can be very challenging [Zhao et al. 2020]. Recently, a class of differentiable programming languages emerged that generalizes these derivative estimators to a broader class of problems [Bangaru et al. 2021; Michel et al. 2024; Yang et al. 2022]. However, these languages are either still restricted by the class of programs that can be differentiated, require extensive user guidance on the sampling routines, or introduce non-negligible approximations to the derivative computation.

We propose a general automatic differentiation algorithm that "just works" with discontinuities, as long as the decision boundary itself is differentiable. We implement our algorithm inside a standard shader programming language [Bangaru et al. 2023]. We focus on low-dimensional integrals (2D or 3D). Our method is based on two key components that go hand-in-hand. The first is an automatic shader transformation (right inset) that turns a piecewise continuous shader into a piecewise constant one.



Shader Transformation



Segment Snapping

The shader transformation enables our second key component, segment snapping (left inset), to perform automatic boundary detection by randomly sampling line segments in the integration domain, and checking whether the two end points are in the same region. Once we obtain the boundary samples by a bisection search along the segments, we perform kernel density estimation to compute the probability density of the samples for Monte Carlo integration. Moreover, the appropriate difference in function values on both sides can be computed simply by comparing values on both sides of the branch in the shader program. Our method is theoretically consistent, enjoys low variance, and scales to complex decision boundaries with a large number of conditions. Our program transformation fully happens at compile time. It preserves the original program structure, which facilitates modularity [Michel et al. 2024] while maintaining efficiency. The generality of our method unlocks a wide range of applications that were not possible before.

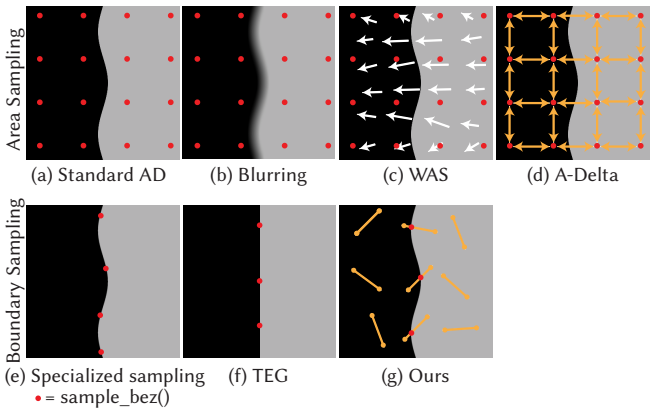


Fig. 2. *Taxonomy.* Previous work in boundary derivative computation can be classified into two approaches: boundary sampling and area sampling. Boundary sampling methods achieve high accuracy with fewer samples (red points) but are limited by their requirement for specialized sampling routines. In contrast, area sampling methods offer greater generality in the functions they can differentiate, but require higher sampling rates to do so. Our method eliminates the need for specialized routines while maintaining the efficiency of boundary sampling, making it applicable to a broader range of discontinuous functions.

Our contributions are:

- A program transformation that converts discontinuous shaders into surrogate shaders with piecewise-constant outputs, enabling reliable detection of parametric discontinuities.
- A segment snapping approach that enables boundary sampling for general discontinuous functions without requiring specialized sampling routines.
- A practical implementation in SLANG.D that allows users to write discontinuous shaders with automatic derivative computation.
- Applications including non-photorealistic rendering, inverse rendering with a diverse set of primitives, shadow art, and inverse brush design. (Fig. 1).

2 RELATED WORK

We review the different types of numerical methods in the context of computing the derivative (with respect to θ) of a discontinuous function $f(x,\theta)$ after integration (with x) given by $\nabla_\theta \int_\Omega f(x,\theta)\mathrm{d}x$. Fig. 2 shows an illustration of different classes of methods. The two main categories of methods are *area sampling* and *boundary sampling*: area sampling methods place samples across the entire domain, and boundary sampling methods place samples at the decision boundaries.

Automatic differentiation of non-differentiable functions. Automatic differentiation [Griewank and Walther 2008] has been proven to be correct *almost everywhere* if the function itself is differentiable

almost everywhere [Lee et al. 2020; Kakade and Lee 2018]. However, these analyses do not consider distributional derivatives [de Amorim and Lam 2022] that appear when differentiating an integral over discontinuous functions—in such a case, automatic differentiation can lead to incorrect results even when the integral is differentiable everywhere [Bangaru et al. 2021; Lew et al. 2023; Michel et al. 2024]. Intuitively, this is because the conditions inside if statements are ignored in standard automatic differentiation, and these conditions are required to reason about how the decision boundary moves. Arya et al. [2022] address differentiation of the expectation of discrete randomness using score estimators [Mohamed et al. 2020] with correlated sampling to significantly reduce variance, but this method has limited application to reverse-mode automatic differentiation. Suh et al. [2022] discussed the consequences of ignoring discontinuities in control applications.

Blurring the discontinuities. A commonly used method to remove discontinuities in programs is to transform them into smooth gradual changes (Fig. 2b). This has been used to make, e.g., sorting, differentiable [Qin et al. 2010; Berthet et al. 2020]. A few differentiable rasterizers [de La Gorce et al. 2011; Liu et al. 2019; Laine et al. 2020] also applied this strategy. For programs, this can be formulated as a convolution of a discontinuous program over a smooth function [Chaudhuri and Solar-Lezama 2010; Kreikemeyer and Andelfinger 2023]. Furthermore, there is a class of derivative-free optimization methods that blur the discontinuities in *parameter space* (θ) instead of over the integration domain (x), also effectively removing the discontinuities [Staines and Barber 2012; Rechenberg and Eigen 1973; Le Lidec et al. 2021; Fischer and Ritschel 2023; Deliot et al. 2024].

Unfortunately, these methods risk biasing the gradients by changing the function being differentiated. Choosing the blurring strength automatically is challenging: strong blurring removes details, and weak blurring leads to sparse derivatives. Derivative-free methods further face scalability issues when the parameter space is high-dimensional. Our method does not suffer from these issues by avoiding blurring altogether, directly sampling the boundary, and being used in combination with reverse mode automatic differentiation [Griewank and Walther 2008].

Boundary sampling. The boundary sampling methods (Fig. 2e) place samples explicitly on the decision boundaries, directly solving the boundary integral [Li et al. 2018; Lee et al. 2018]. Existing boundary sampling methods all require a way to parameterize and sample on the boundary. For piecewise linear discontinuities, this can be easily automated [Lee et al. 2018; Bangaru et al. 2021] (Fig. 2f) However, when the decision boundaries become more complex, it is often required to design a problem-specific parameterization of the decision boundary. For example, Li et al. [2020] designed specialized solvers for differentiable vector graphics rendering, whereas Zhang et al. [2020] specialize for path-space differentiable rendering. Specialized adaptive importance sampling strategies [Yan et al. 2022; Zhang et al. 2023] or Markov-chain Monte Carlo mutation [Xu et al. 2024] have been proposed too. Our method belongs to the boundary sampling class, but does not require specialized sampling routines tailored to a specific type of boundary. This enables applications that were difficult before, e.g., differentiating decision boundaries encoded by an implicit coordinate neural network.

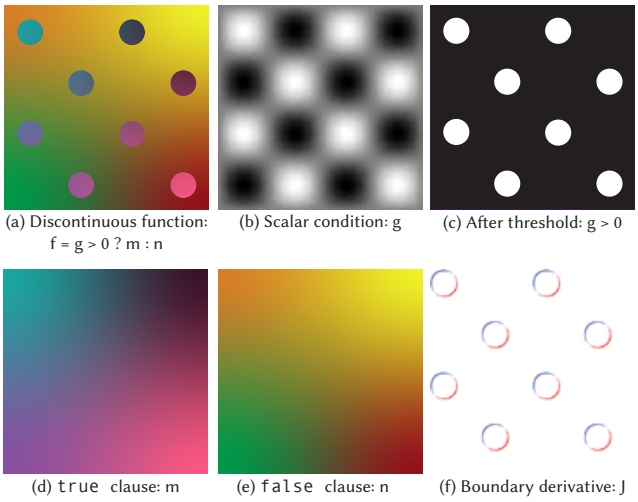


Fig. 3. *Discontinuous function and its boundary derivative.* The discontinuous function f (a) is constructed using a branching operator that selects between two functions m, n (d,e) based on the sign of a scalar condition g (b,c). Formally, f is defined as $f(x,\theta) = m(x,\theta)$ when $g(x,\theta) > 0$ and $f(x,\theta) = n(x,\theta)$ when $g(x,\theta) \leq 0$, exhibiting a jump discontinuity at the boundary defined by the zero level set of g. When $g(x,\theta)$ is differentiable in θ, perturbation in θ induces a change in f's boundaries (f), pushing them inwards (red) or outwards (blue). The associated derivative is the boundary derivative integral $\partial_\theta I_b$ Eq. (3).

Area sampling. Since sampling the boundary can be difficult, some recent works [Loubet et al. 2019; Bangaru et al. 2020, 2022; Vicini et al. 2022; Zeltner et al. 2021; Xu et al. 2023] instead convert the boundary integral back to its original domain by constructing an appropriate velocity field and applying the divergence theorem, or equivalently, applying a reparameterization of the area integral to remove discontinuities (Fig. 2c). The integral in its original domain can then be estimated without any boundary sampling. Unfortunately, as we show in the results, the commonly used harmonic-interpolation-based velocity fields often suffer from high (and sometimes infinite) variance. A-δ [Yang et al. 2022], on the other hand, derives an approximated area sampling solution by sampling on a grid and using neighboring information to detect boundaries (Fig. 2d). Unfortunately, when the grid sampling frequency is lower than the discontinuities' frequency, their approximation often leads to significant bias in the gradient. Our method is boundary sampling based, has low variance, and is theoretically consistent, and as such, it does not suffer from these issues.

3 MOTIVATION AND BACKGROUND

We are interested in applications that involve parametric integrals $I(\theta)$ with discontinuous integrands, $f(x,\theta) : \Omega \times \mathbb{R}^k \to \mathbb{R}^d$:

$$I(\theta) = \int_\Omega f(x;\theta)\,\mathrm{d}x. \tag{1}$$

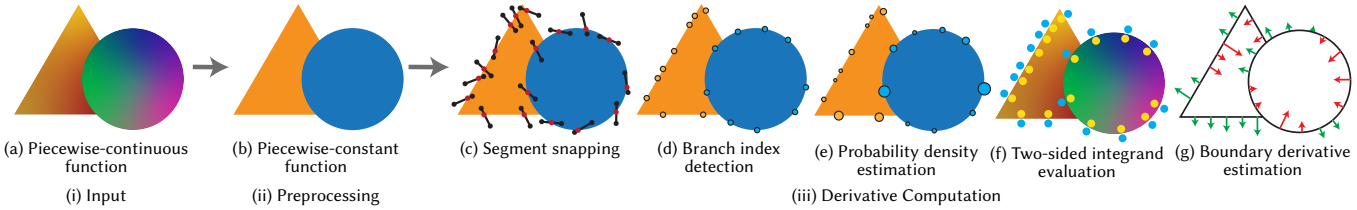


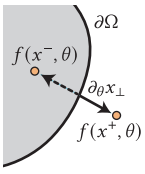
Fig. 4. *Overview.* Our method automatically samples a discontinuous function's boundary to estimate the boundary derivative integral, Eq. (3). Starting with a piecewise-continuous function (a) whose program source code is provided as input, we perform a function transformation to remove its continuous variation and turn it into a piecewise-constant function (b) in a preprocessing step. We use this function to sample points on the boundary by snapping line segments on to it (c). Next, we map the boundary samples to their branching operator in code (d), estimate the density of the points on the boundary (e), and evaluate the integrand on both sides of the boundary (f). Finally, we put all these together and estimate the boundary derivative integral in a single reverse-mode pass (g).

We assume the integration domain Ω to be in a low-dimensional Euclidean space, *i.e.*, $\Omega \subset \mathbb{R}^{\{1,2,3\}}$. The parameter set $\theta \in \mathbb{R}^k$ is a vector of arbitrarily large dimension k. Given a set of measurements (samples of the integrand) for I (*e.g.*, the pixel integral in Fig. 3a), we wish to recover an optimal set of parameters θ (*e.g.*, the positions of the enveloped disks). By the Reynolds transport theorem [1903], the gradient $\nabla_\theta I$ is a sum of two terms:

$$\nabla_\theta I = \int_\Omega \nabla_\theta f(x;\theta)\,\mathrm{d}x + \partial_\theta I_b. \tag{2}$$

The first integral, *i.e.*, interior term, is computed using standard `autodiff`. The second term is a boundary integral that is evaluated over the discontinuities $\partial\Omega$

$$\partial_\theta I_b = \int_{\partial\Omega} J(x,\theta)\,\mathrm{d}x = \int_{\partial\Omega} \left(f(x^+,\theta) - f(x^-,\theta)\right) \partial_\theta x_\perp\,\mathrm{d}x. \tag{3}$$



Estimating the integrand, or *boundary derivative* $J(x,\theta)$, involves three key steps: (i) drawing samples on the discontinuous boundary $\partial\Omega$; (ii) efficiently computing the normal component of the boundary velocity $\partial_\theta x_\perp$, for all components of θ; and (iii) evaluating the integrand difference at the boundary, $f(x^+) - f(x^-)$, where the points $x^\pm = \lim_{\epsilon\to 0^+} x \pm \epsilon\hat{n}$ are on either side of the boundary with unit normal $\hat{n}$.

The main challenge is to formulate an explicit and well-defined routine to draw samples on the boundary $\partial\Omega$. We propose a method to address this for a large class of functions, described next.

4 METHOD

We introduce an automatic boundary sampling method to estimate the boundary derivative integral in Eq. (3). Our method assumes access to the full program source code that defines a discontinuous function f using branching operators (*i.e.*, `if-else` conditionals). Figure 3 illustrates how this simple code structure can be used to identify discontinuities. Our method rests on two key ideas: (i) transform piecewise-continuous functions (Fig. 4-a) into surrogate functions (Fig. 4-b) with piecewise-constant output (§4.2), and (ii) snap randomly sampled line segments onto the discontinuities using a bisection method to detect the boundary (Fig. 4-c, §4.3). We associate each boundary sample to the corresponding branching operator (Fig. 4-d, §4.4). Then, we estimate the boundary sampling density (Fig. 4-e, §4.5), evaluate the integrand on the boundary (Fig. 4-f, §4.6), and estimate the boundary velocity (Fig. 4-g) to compute the boundary derivative integral (§4.7).

4.1 Discontinuous integrands and scope

We require that f be represented as a directed acyclic graph (DAG) $G = (V, E)$, with discontinuities expressed as `if-else` constructs, which is a mild requirement that subsumes most practical shaders, as visualized in Fig. 5b. Our method uses static analysis to identify all unique branching operations. The evaluation point and parameters (x, θ) act as the source nodes in the graph. The interior nodes can be of one of the following two types: *differentiable* operations in a predetermined set D, such as $+$, $*$, sin, exp, etc.; or discontinuity-inducing *branching* operators:

$$\mathrm{B}(x,\theta) = \begin{cases} h^+(x,\theta) & \text{if } g(x,\theta) > 0 \\ h^-(x,\theta) & \text{otherwise.} \end{cases} \tag{4}$$

B selects among its two successor nodes $h^\pm$ based on the value of its scalar predecessor g, called the *boundary function* (the discontinuity is at $g(x,\theta) = 0$). B is equivalent to an `if-else` statement. To compute the boundary velocity $\partial_\theta x_\perp$, we require $g(x,\theta)$ to be differentiable almost everywhere[1], *i.e.*, all nodes in g*'s subgraph* are differentiable, and that $||\partial_x g|| > 0$. Equivalently, g's subgraph includes no type-B nodes. As g is continuous, its zero-level set $\partial\Omega$ forms a closed curve in 2D (or a closed surface in 3D), which we will use to detect discontinuities (§4.3). There are no such restrictions on $h^\pm$, which allows us to nest B operators (Fig. 5).

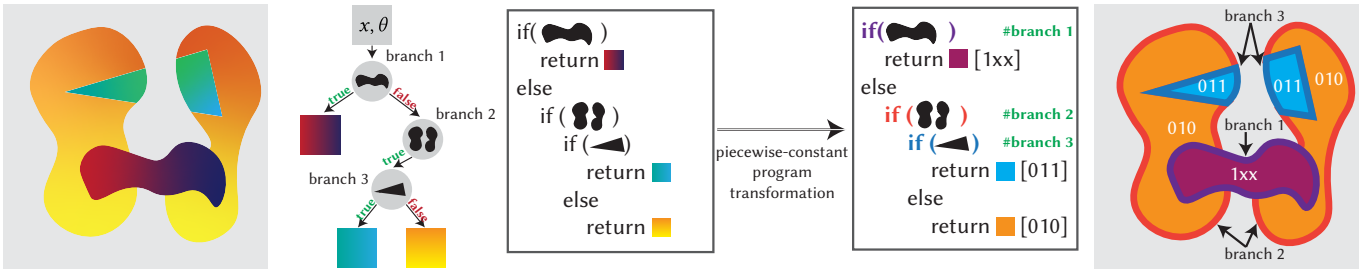
Let B_i and g_i be the i-th B operator and branch function. The set of discontinuous boundaries $\partial\Omega$ is the union of all locations x such that $g_i(x,\theta) = 0$ and B_i is executed:

$$\partial\Omega = \left\{ x \middle|\, \vee_{i=1}^{|\mathrm{B}|}\, g_i(x,\theta) = 0 \wedge \mathrm{B}_i \text{is executed} \right\}. \tag{5}$$

4.2 Piecewise-constant surrogate for boundary detection

The fundamental challenge in detecting discontinuities is that changes in function values can arise from either continuous variations or discontinuous boundaries. This ambiguity makes it impossible to reliably detect discontinuities by examining only the function values. Our solution is to derive a piecewise-constant surrogate function f_{pc} that is a graph-coloring transform of f (Fig. 4-b). It preserves the

[1] g is allowed to be discontinuous on a set of measure zero, as we discuss in §5



(a) original function's output (b) function depicted as a DAG (c) original program (d) piecewise constant program (e) piecewise-constant function's output

Fig. 5. *Piecewise-constant transformation.* We transform a piecewise-continuous function (a), represented as a DAG (b), into a piecewise-constant function (e) through a source-code transformation (c,d) that tracks the result of all branching decisions (`if` conditions) as the function is evaluated in a ternary vector (with values $\times, 0, 1$ corresponding to true branch, false branch and branch not reached).

graph structure (the discontinuity curves $\partial\Omega$), eliminates continuous variation, and ensures that neighboring regions have unique identifiers (Fig. 5). Both functions f and f_{pc} share the same discontinuities, arising from the evaluation paths dictated by a sequence of branching decisions. Let m be the number of B operators. We represent the piece-wise constant function output $f_{\text{pc}}(x)$ at a given point x using a ternary state vector $s_x = (s_i)_{i=1}^{m}$, where each element s_i is defined as follows:

$$s_i = \begin{cases} \times & \text{branch } i \text{ not evaluated,} \\ 0 & \text{predicate of branch } i \text{ evaluated as } \texttt{true}, \\ 1 & \text{predicate of branch } i \text{ evaluated as } \texttt{false}. \end{cases} \tag{6}$$

In a single forward program pass, our method tracks each branching predicate that is evaluated and updates the corresponding element in the state vector according to Eq. (6). Figure 5 illustrates this process. Importantly, a single evaluation of f_{pc} at x is linear in the number m of branches; unlike A-δ, we do *not* evaluate all branches. Each continuous region of f is uniquely identified by a base-3 integer, that is: every x in the region, $f_{\text{pc}}(x)$ evaluates to the same state vector. This transformation removes all continuous variations from f, thereby facilitating robust boundary sampling (§4.3).

4.3 Sampling the boundary by segment snapping

We sample the boundary by point-sampling our piecewise constant function f_{pc} (Fig. 4c). We can reliably detect when a line segment crosses a discontinuity by comparing the function values at the segment's endpoints (Fig. 6). If these values differ, we know with certainty that the segment intersects a boundary. We can then further localize the boundary efficiently using a bisection search.

Initialization. We start with an initial set of line segments uniformly distributed over Ω. Each segment is created by generating a set of stratified jittered points over a grid as the first endpoint, then adding a second endpoint at a fixed distance (equal to the initial grid spacing) in a random direction.[2] This grid is used only for segment initialization, the final function evaluation is performed at

[2]The segment length can also be a random variable $\mathcal{U}[0, 1/\text{gridsize}]$, which ensures all points on the boundary are sampled with non-zero probability; in practice we found this to not be necessary for our applications.

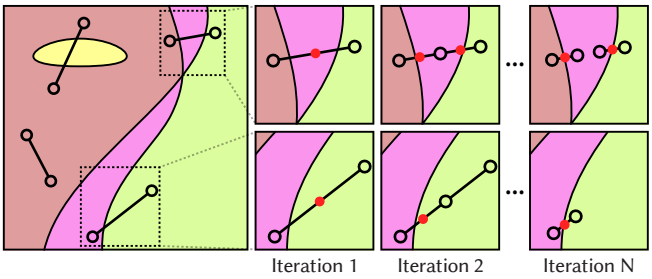


Fig. 6. *Segment snapping.* Starting with a set of randomly initialized line segments (left), we evaluate the piecewise-constant function *f_pc* at both of its endpoints. If the values differ, we know that the segment intersects a discontinuity, which we localize using bisection search (right). If the segment crosses multiple discontinuities, we may localize points on multiple boundaries (right-top). If the values are the same, we discard the segment; this may miss thin regions, but we have found it to not be a major limitation in practice.

the boundary after snapping, unlike A-δ [Yang et al. 2022], which operates on the initial regular-grid points.

Bisection search. Now, for each segment, we evaluate f_{pc} at both endpoints. If the values differ, indicating a boundary crossing, we split the segment at its midpoint and recursively apply the same process to the two spawned halves. This bisection search continues until the segment length becomes sufficiently small (typically after about 15 iterations, reducing the original width by a factor of 2^{-15}), at which point the midpoint provides an accurate sample of the boundary location. Figure 6 illustrates this process.

4.4 Branch index detection

After locating points on the discontinuity boundary via segment snapping, our method determines which B operator defines the boundary through each point (Fig. 4d). This information supports three tasks: (i) computing boundary velocity through automatic differentiation; (ii) estimating the sample density required by our Monte Carlo estimator of the boundary integral in Eq. (10); and (iii) evaluating the integrand on both sides of the boundary.

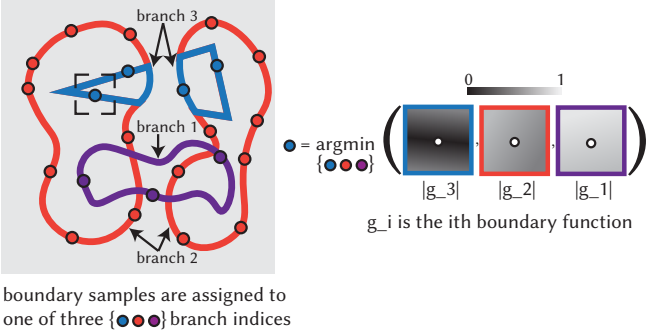


Fig. 7. *Assigning image-space boundary samples to branch indices.* Segment snapping produces samples on the discontinuous boundaries of the function f; however, we only know each sample's coordinates in image space. In order to compute the boundary derivative, we must also identify the index of the branch operator B_i whose boundary each sample lies on. To do this, we use the fact that the boundary of B_i is the zero level set of its boundary function $g_i(x;\theta)$. Therefore, we associate each boundary point with the branch i for which $|g_i(x;\theta)|$ is minimal during function evaluation.

For each boundary point x, we know that it must lie on the zero level set of some implicit function g_i corresponding to a B operator. In theory, we should have $g_i(x;\theta) = 0$ for the correct operator. However, due to numerical precision in floating-point arithmetic and due to finite number of bisection steps in segment snapping, the point may not lie exactly on the boundary. Therefore, we identify the responsible B operator by finding the index i that minimizes $|g_i(x;\theta)|$ during function evaluation. We visualize branch index detection in Fig. 7.

This approach *does not* require exhaustive evaluation of all boundary functions (for example, we do not evaluate g_i for branches that are not taken), only those that are encountered during f's evaluation path for a particular input are tested. While in theory multiple boundary functions close to zero may result in incorrect branch detection, in practice we have found it to be mostly not an issue; we further discuss this and a failure case in §6.

Our approach not only provides the index of the relevant B operator but also gives us access to the value of g_i, which we maintain in the automatic differentiation computation graph; it is essential for computing g_i's derivatives needed for the boundary velocity.

4.5 Probability density estimation

The distribution of points on the boundary resulting from segment snapping is not uniform in general, as it depends on both the initial line segment distribution and the boundary geometry $\partial\Omega$. To account for this non-uniformity in our Monte Carlo estimator of the boundary derivative integral, we estimate the probability density of the samples using kernel density estimation (Fig. 4e). For each boundary point, we compute the probability density based on its local neighborhood. Specifically, when the domain is 2D, for a point x on the boundary, we estimate its probability density as

$$p(x) = \frac{k}{n}\frac{1}{2R_k}, \tag{7}$$

where the hyperparameter k is the number of nearest neighbors, n is the total number of samples, and R_k is the distance to the k-th nearest neighbor [Mack and Rosenblatt 1979]. This estimator is biased but consistent [Berry and Sauer 2017]. We discuss its properties and extensions to more dimensions in Appendix A.

4.6 Two-sided integrand evaluation by program transform

The boundary derivative at each point x on the boundary depends on the value of the integrand f on either side of the boundary. Prior work [Li et al. 2018, 2020] evaluates this by finding two points (x^+ and x^-) on opposite sides of the boundary and computing $f(x^+)$ and $f(x^-)$. However, this approach requires carefully choosing an offset ϵ along the normal direction, which can be inaccurate when the boundary is highly curved or when the integrand varies rapidly near the boundary.

Instead, we transform f again, this time to evaluate both sides exactly at the boundary point (Fig. 4f). For a boundary point x corresponding to branch operator i, we create a transformed function that takes an additional input sign $\in \{+,-\}$ and forces the branch operator to take the corresponding path regardless of the sign of $g_i(x)$. As shown in List. 7, this is implemented by overriding the branch condition for the identified operator, allowing us to evaluate both sides of the discontinuity exactly at the boundary point.

4.7 Boundary velocity and derivative computation

We now have all the components necessary to compute the boundary derivative. First, the set of points $\{x_i\}_{i=1}^n$ on the boundary $\partial\Omega$ by snapping line segments to the boundary (§4.3). Second, the sampling probabilities $\{p_i\}_{i=1}^n$ for these points, computed using kernel density estimation (§4.5). Third, the integrand difference $f(x_i^+,\theta) - f(x_i^-,\theta)$ for each boundary point (§4.6). We now discuss how to compute the boundary velocity together with the boundary derivative.

The boundary velocity $\partial_\theta x_\perp$ represents how a point $x \in \partial\Omega$ moves in response to changes in the parameter θ. The implicit function theorem relates $\partial_\theta x_\perp$ to the derivatives of the boundary function $g_i(x;\theta)$, given by

$$\partial_\theta x_\perp = -\frac{\partial_\theta g_i(x)}{\|\partial_x g_i(x)\|}, \tag{8}$$

and we assume $\|\partial_x g_i(x)\| > 0$ everywhere. g only needs to be differentiable *almost* everywhere; its first derivative can have jump discontinuities, which we exploit in some applications §5.

Although the implicit function theorem gives us Eq. (8), we do not compute $\partial_\theta g_i(x)$ directly for each boundary point (or each branch) since this would require multiple reverse-mode passes. Instead, we compute $g_i(x)/\|\partial_x g_i(x)\|$ and leave $g_i(x)$ in the numerator undifferentiated. $\partial_x g_i(x)$ is computed using forward-mode `autodiff`. Now, substituting the boundary velocity from Eq. (8) in the boundary derivative integral Eq. (3), we get

$$\partial_\theta I_b = -\int_{\partial\Omega} \left(f(x^+;\theta) - f(x^-;\theta)\right) \frac{\partial_\theta g(x)}{\|\partial_x g(x)\|}\,\mathrm{d}x. \tag{9}$$

We can now simultaneously compute the boundary velocity and derivative using the sampled points and applying reverse-mode

`autodiff` with respect to the parameters θ to the expression

$$\partial_\theta I_b = -\mathrm{AD}_\theta \left\{ \sum_{i=1}^{n} \frac{\left(f(x_i^+, \theta) - f(x_i^-, \theta)\right)}{p_i} \frac{g_i(x_i)}{\|\partial_x g_i(x_i)\|} \right\}. \quad (10)$$

To ensure the correct boundary velocity (and derivative) is computed, we only propagate the gradients backward through $g_i(x_i)$ (which results in $\partial_\theta g_i(x_i)$ once differentiated) and "detach" all other terms (in autograd terms). This allows us to compute the boundary derivative for all points, irrespective of the boundary they are on, and with respect to all components of θ simultaneously using only a single reverse-mode `autodiff` pass.

4.8 Practical Implementation

We implement our approach as a Python-based compiler that transforms discontinuous shader programs written in SLANG.D [Bangaru et al. 2023] supplemented with a *new* type of tag—`[Disc]` to mark discontinuous `if-else` statements. Our compiler converts the input program into another that efficiently computes boundary derivatives. Our program transformation is fast (30ms on average for our examples). It only requires two passes through the program's DAG structure. Crucially, our compilation time scales only linearly with the number of branches, rather than enumerating all possible evaluation paths, which scales exponentially (e.g., like A-δ).

Segment snapping, probability density estimation and boundary derivative computation, and the gradient-based optimization code are implemented in PyTorch [Paszke et al. 2017]. We invoke each of the transformed SLANG.D programs from Python during boundary derivative computation as needed. The continuous parts of the derivative are handled by PyTorch and SLANG.D's automatic differentiation system. The two systems interoperate by invoking the SLANG.D shader's forward and backward subroutines through a custom PyTorch autograd function.

We provide an outline of our practical implementation in Appendix B.2. We will release our source code upon acceptance.

5 RESULTS AND APPLICATIONS

We compare our method with previous boundary-derivative computation methods in §5.1 and showcase several applications in §5.2.

5.1 Comparison with boundary derivative methods

Broadly, there are two approaches to computing boundary derivatives, (i) boundary sampling methods [Bangaru et al. 2020; Li et al. 2020] that require specialized sampling routines, and (ii) area sampling methods that make approximation errors [Yang et al. 2022; Laine et al. 2020] or produce high-variance estimates [Bangaru et al. 2020]. With equal samples, boundary-sampling methods usually estimate derivatives with greater accuracy. Ours is a boundary-sampling method that reduces constraints on discontinuous programs. Neither does it require specialized boundary sampling routines nor diffeomorphism constraints [Bangaru et al. 2021]. Contrary to prior methods that are highly optimized for specific problems, our goal is to support a wide variety of discontinuous programs.

Discontinuities with explicit sampling routines. We first consider the simple example of computing a circle's derivative in Fig. 8, for which all methods compute a high quality derivative.

Discontinuities without explicit sampling routines. Next, we differentiate a more complicated program in Fig. 9. Constructing a sampling routine for it is challenging since its boundary cannot be expressed explicitly, which limits our comparisons to area-sampling methods. Discontinuity blurring requires careful tuning of the blurring width for accurate gradients and the warp field induces high variance for WAS. A-δ and our method handle this example well.

Area sampling cannot resolve high-frequency features at low sampling rates. Area-sampling methods can be accurate when the sampling rate is high relative to the integrand's variations (Figs. 8 and 9). However, their accuracy degrades significantly at lower sampling rates. We demonstrate this with two specific cases: a) multiple discontinuities in close proximity in Fig. 10; b) high frequency continuous variations in Fig. 11. Both limitations are expected, since without sufficient sampling at or near the boundary, area-sampling methods cannot resolve high-frequency features near it. Our method handles these examples well because it samples the boundary.

A-δ struggles with a large number of discontinuities and branching. Among current systems, A-δ typically handles the most general set of discontinuous programs so we focus the rest of our comparisons on it. Applications used in practice often contain a large number of discontinuities (rasterizers) and can have large tree-like structures (CSG trees). For the applications in §5.2 our method is able to support we find that A-δ fails to compile most of the times because the number of discontinuities is fairly large.

We first compare A-δ with our method and DiffVG for painterly rendering in Fig. 12. Here, the goal is to optimize the colors, position and radius of a set of ordered opaque disks to match a target image. This example highlights A-δ's first limitation at larger scales — its compiler is not well suited to a large number of discontinuous statements (each circle corresponds to one) and times out for more than 200 disks. On the other hand, the compilation effort required by our system is minimal (two DAG traversals) and is independent of the number of disks, taking ~30ms. Queries can also be accelerated with bounding volume hierarchies (BVH) and quadtrees; we use a regular-grid-based acceleration structure. Our implementation takes ~90s for 2000 circles, while DiffVG takes ~160s; these numbers are only indicative of performance potential, not a direct comparison, since DiffVG is a more complete system than our simple vector graphics program.

The second issue is that A-δ always evaluates all branching statements (see App D.3 in their paper), regardless of whether the current evaluation point reaches them. Consider the checkerboard example in Fig. 13, which is expressed using two binary trees. For a query point, our method traverses the tree and evaluates exactly one `if-else` statement at each level of the tree, resulting in linear runtime growth with tree depth. In contrast, A-δ evaluates all branches at each level and its runtime grows exponentially with tree depth, severely limiting the maximum depth it can support. At larger depths, A-δ's derivative accuracy is also compromised since there can be multiple discontinuities between evaluation points.

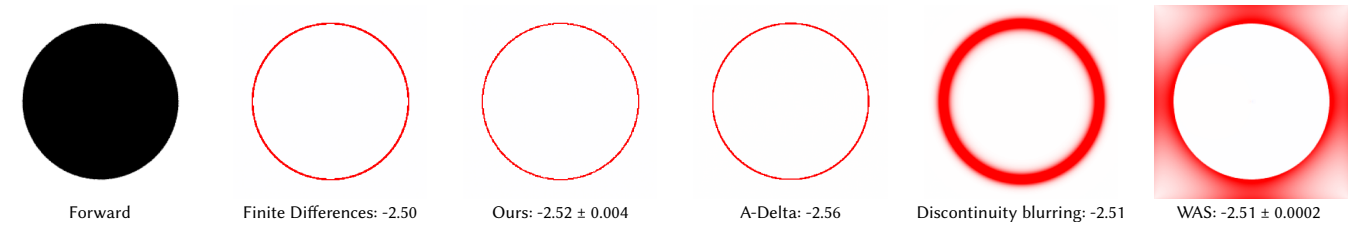


Fig. 8. *Boundary sampling routine available.* In the simple example above, we compute the derivative of the integral of a circle (0 inside, 1 outside) over the image plane with respect to its radius. Its derivative is $\partial_r I_b = -2\pi r \approx 2.51$, ($r = 0.4$). All methods compute it fairly accurately at equal sample count: DiffVG [Li et al. 2020] (derivative = 2.51, not visualized) with its specialized edge sampling routine, discontinuity blurring, WAS [Bangaru et al. 2020], A-δ and ours.

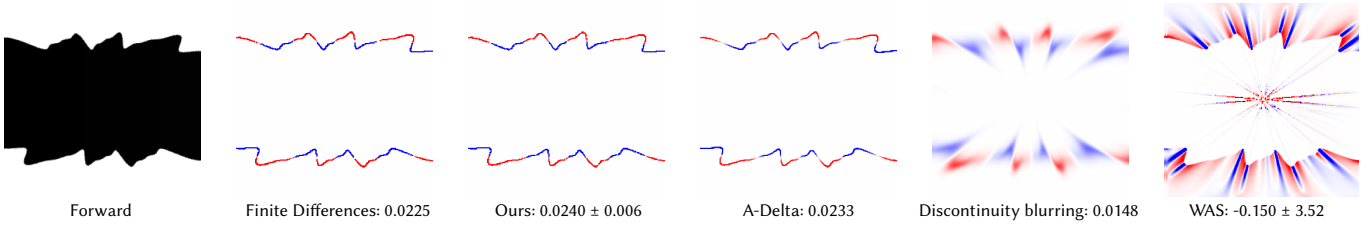


Fig. 9. *No explicit boundary sampling routine.* In this example, the shape above does not have an easily computable boundary sampling routine, so we only compare with area-sampling methods. Blurring the boundary has a noticably high bias, which can be reduced with a smaller blurring width, at the cost of increased variance (determining a blurring width that works generally in general settings is non-trivial). Warped area sampling (WAS) suffers from high variance since the complicated nature of the boundary induces a rapidly changing warp field, including high variance regions in the center away from the boundary. Both A-δ and our method compute high-quality derivatives.

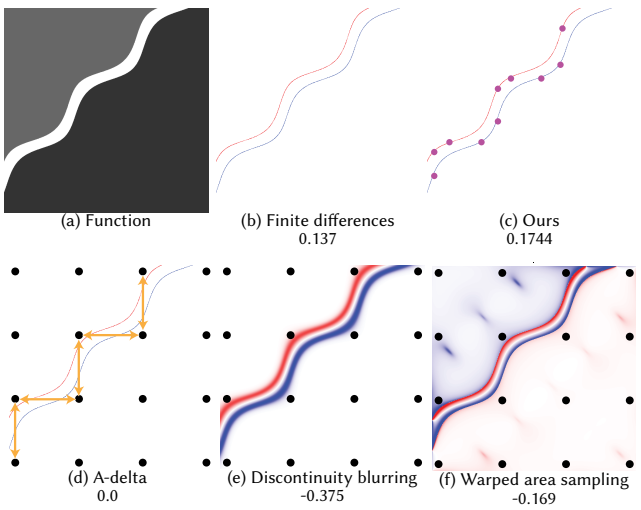


Fig. 10. *Insufficient sampling near multiple discontinuities.* For the piecewise-constant function (a), we compute derivatives of the translation of both discontinuities (b). Using a 4x4 sampling grid (with additional samples for segment snapping in our method), we compare different approaches. Our method achieves accurate sampling and correct derivative signs. A-δ produces zero derivatives due to multiple discontinuities between evaluation points, while both discontinuity blurring (e) and warped area sampling (f) yield incorrect signs due to insufficient sampling near discontinuities.

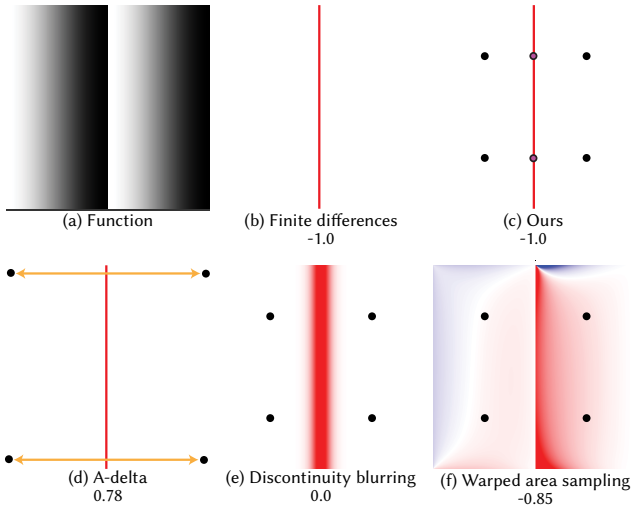


Fig. 11. *Insufficient sampling near high-frequency continuous variation.* When the integrand has high-frequency variation between the discontinuity and the evaluation points, A-Delta's gradient sign can flip. Discontinuity blurring only samples regions of zero contribution. Both WAS and ours compute derivatives with the correct sign, but WAS has higher variance.

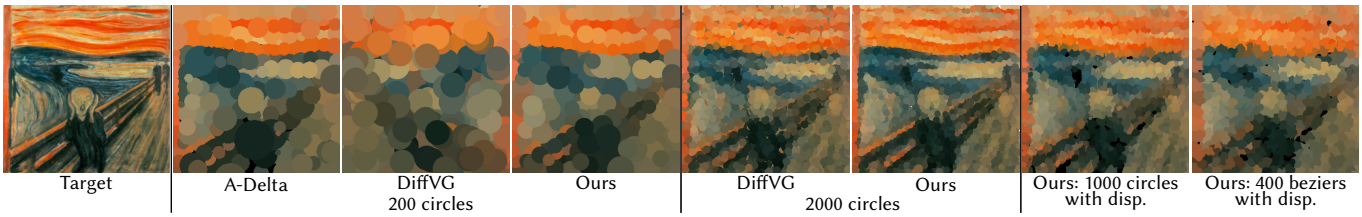
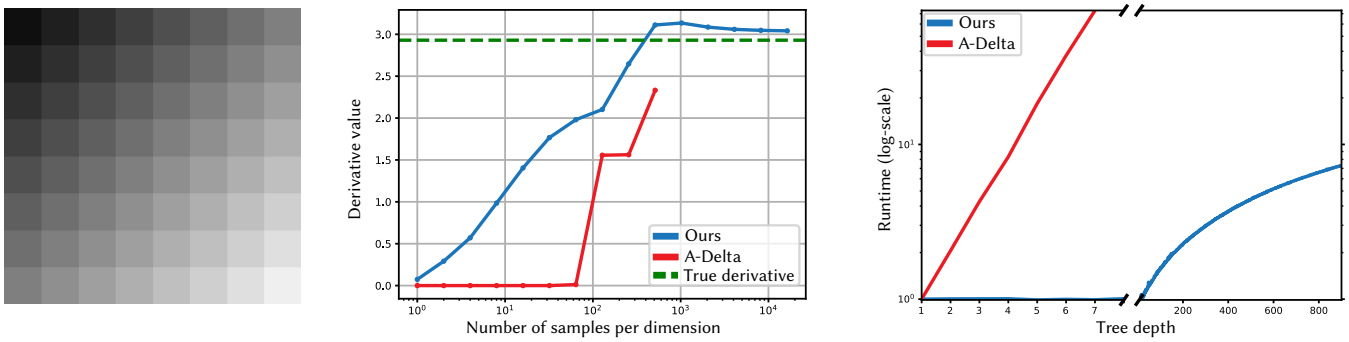


Fig. 12. *Painterly rendering.* Given a target image (left), we optimize a set of randomly initialized primtives using an L_2 loss to produce a painterly renderings. With 200 circles, all methods capture the rough structure of the target. With 2000 circles, both our method and DiffVG produce painterly renderings with lower reconstruction error, while A-δ does not scale to the large number of primitives (Fig. 13). Unlike DiffVG, our method does not require specialized boundary sampling routines; it supports non-standard primitives like noise-deformed circles and bezier curves (final two columns) .



(a) Checkerboard with depth=3

(b) Derivative accuracy vs number of samples

(c) Forward runtime vs depth of the tree

Fig. 13. *Branching structures.* A checkerboard pattern (a) is implemented as the sum of the outputs of two binary trees, one along each axis. Each axis has $2^d - 1$ discontinuities (d is the tree depth). We compute the derivative with the relative position of all discontinuities and compare its accuracy at different sampling rates for $d = 7$ in (b). Up until 64^2 samples, A-δ computes zero derivatives because there are multiple discontinuities between all evaluation points (Fig. 10). A-δ runs out of memory at 512^2 samples; this could be because it assumes all branches are taken and evaluates all 2^d nodes in the tree, irrespective of whether they are reached. Our method consistently computes non-zero derivatives and converges to the true derivative faster than A-δ. Since A-δtakes all branches its runtime grows exponentially with d, while ours grows linearly. We normalize both method's runtimes at depth $d = 1$ for visualization purposes; our method appears to have no increase in runtime up until depth 50 because our runtime is dominated by kernel launch overhead.

5.2 Applications

Next, we show a wide range of applications implemented using our method. In all code listings, for conciseness, we only show the piecewise-constant transformation along with the input program (see Appendix B.2 for the keywords used in the listings).

```
float bilerp_step(float2 x, float* p, int& region) {
  float s = bilerp(x, theta); // Bilinear
       interpolation of grid values in "p"
  [Disc]
  if (s > 0.0) { // Step activation
    if (M_PW_CONST) region = hash(region, 0);
    return 1.0;
  } else {
    if (M_PW_CONST) region = hash(region, 1);
    return 0.0;
  }
}
```

Listing 1. We optimize a binary-valued function, constructed by thresholding a bilinearly interpolated grid, by automatically computing its boundary derivatives.

5.2.1 Binary function optimization. We fit a binary-valued target function by optimizing the parameters of a binary-valued discontinuous program which thresholds a bilinearly interpolated grid of values (List. 1) Since the program is binary-valued, only its boundary derivative is non-zero. When the target is provided as a set of discrete values (like a raster image), we convert it to a continuous signal using nearest-neighbor interpolation. We do not have access to the discontinuity locations of the target.

Figure 14 shows an example fitting a binary raster image. Figure 15, compares with two baselines, by replacing our threshold with: i) a ReLU activation (clamped between 0 and 1 as done by Karnewar et al. [2022]), and ii) a sigmoid activation. We optimize an L2 loss as an integral over the image space with 1000 steps of the Adam optimizer, tuning learning rates for each method. This takes about 30 seconds for all methods. As shown by Belhe et al. [2023], the continuous ReLU and sigmoid activations blur the discontinuity. Our method extends theirs (for binary functions) to handle the setting where the discontinuity boundary is not known apriori and can be optimized. The resulting output is discontinuous by construction and accurately preserves the target's discontinuities.

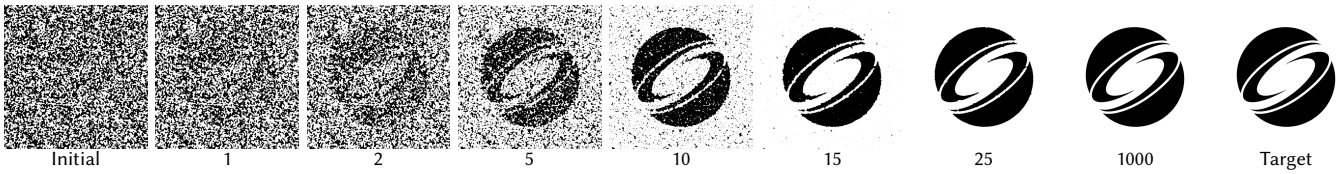


Fig. 14. *Binary raster image fitting.* We start with a random initialization of a bilinearly interpolated thresholded grid, resulting in several discontinuous islands. The boundary derivative encourages each point on these discontinuities to either move along or against the normal direction (causing the islands to expand or shrink) to better match the target, resulting in a sharp reconstruction. See Fig. 15 for a close up comparison.

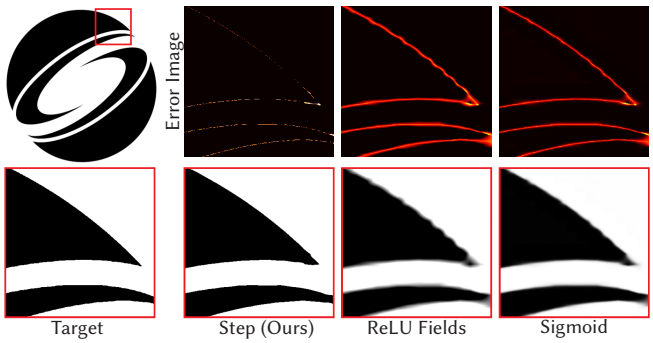


Fig. 15. *Non-discontinuous activations blur the discontinuity.* In this example, we fit a binary-valued raster image, with three different methods, each of which bilinarly interpolates scalar values on a grid followed by their respective activations. The sharp discontinuities in the input can only be preserved by our method which uses a discontinuous step activation, whereas the other two use continuous activations resulting in blurring.

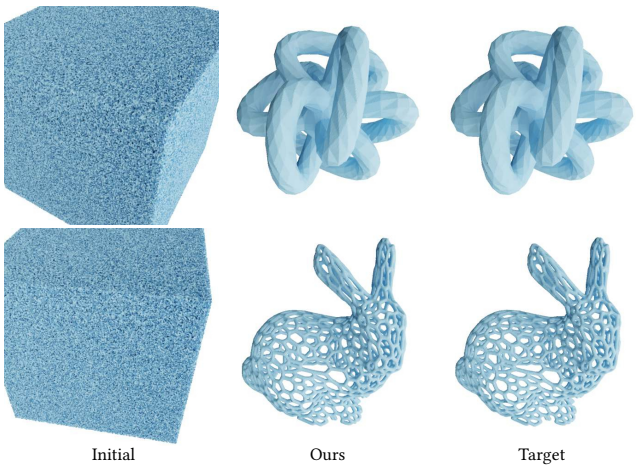


Fig. 16. *Fitting 3D binary functions.* We fit a 3D winding number field (thresholded at 0.5) using a trilinearly interpolated thresholded grid. Starting with a randomly initialized grid (left), we optimize it to fit the target surface (right), the final reconstruction is in the middle. We render all three for visualization. Since we perform optimization in 3D, segment snapping, probability density estimation and all other steps are done in 3D too.

Swept surfaces. We use the same shader to fit a 2D swept surface (Fig. 17b), formed by sweeping a 2D brush along a 1D curve (Fig. 17a). Evaluating a swept surface requires checking if the query point is inside the brush for multiple brush orientations along the curve, which is computationally expensive, especially for brushes represented by costly functions like neural SDFs. Fitting a binary-valued shader enables fast inference, offloading the expensive inside-outside tests to the optimization phase. Our optimization takes around 20 seconds for 400 iterations; inference for our method takes 0.2ms for a 512^2 input, 10x faster than the original swept surface (in which the brush has 100 repeated steps along the curve).

3D inside-outside tests. The same shader extends to 3D function fitting. We show this by fitting a 3D winding number field using a trilinearly interpolated thresholded grid in Fig. 16. The fitting process takes around 9 minutes for 1000 iterations. The training time is dominated by the target evaluation on CPU; inference on our method is fast, taking 1ms for an input of 256^3 points.

We can also optimize (recover) the geometry of the primitives in a CSG tree (with a fixed topology) (Fig. 18) by expressing it as a program. The tree of Listing 2 returns 1.0 if the evaluation point is inside the shape and 0.0 otherwise. The SDF values are used as boundary functions g.

```
// We evaluate the SDFs at each point x and store
     the result in the following variables
sphere, cube, cyl_z, cyl_x, cyl_y = // SDF values
[Disc]
if (sphere > 0.0) { // Inside sphere
  if (M_PW_CONST) region = hash(region, 0);
  [Disc]
  if (cube > 0.0) { // Inside cube
    if (M_PW_CONST) region = hash(region, 2);
    [Disc]
    if (cyl_z < 0.0) { // Outside cylinder z
      if (M_PW_CONST) region = hash(region, 4);
      [Disc]
      if (cyl_x < 0.0) { // Outside cylinder x
        if (M_PW_CONST) region = hash(region, 6);
        [Disc]
        if (cyl_y < 0.0) { // Outside cylinder y
          if (M_PW_CONST) region = hash(region, 8);
          res[thread_idx] = 1.0;
          return;
        }
      }
    }
  }
}
return 0.0;
```

Listing 2. We support nested `[Disc]` if statements, enabling derivative computation of CSG primitives are only differentiable *almost* everywhere.

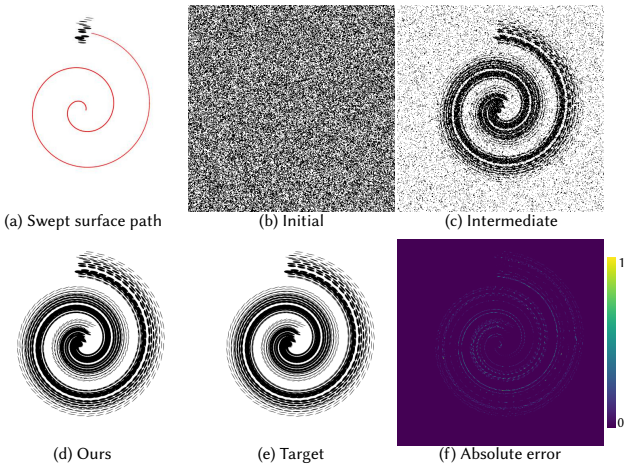


Fig. 17. *2D Swept Surface.* Here, the goal is to fit the inside-outside test of a brush swept along a curve (a) resulting in a binary-valued swept surface (e). Similar to Fig. 15, we use a binary-valued grid to fit the target. Starting from a random initialization (b), our method progressively improves the fit (c) and is able to accurately fit the target (d,f), enabling much faster inference of this swept surface.

```
float voronoi_mosaic(float2 x, float* p, int out) {
  float3 color = 0.0;
  float min_dist = INF;
  // Loop over all points
  [MaxIters(N_POINTS)]
  for (int i = 0; i < N_POINTS; i++) {
    float2 curr_p = float2(p[5*i], p[5*i+1]);
    float3 curr_color = float3(p[5*i+2], p[5*i+3],
        p[5*i+4]);
    // Check if the current point is the closest
    [Disc]
    if (length(x - curr_p) < min_dist) {
      // pw-const hash uses a different index (2*i)
          for every if statement in the loop
      if (M_PW_CONST) region = hash(region, 2*i);
      // Update color and min distance
      color = curr_color;
      min_dist = length(x - curr_p);
    }
  }
  return color, region; // color is the output of
      the original program, region is the output of
      our pw-constant transform
}
```

Listing 3. We support `[Disc]` if statements within for loops, each of which is treated independently which allows us to optimize multiple decision boundaries in the voronoi mosaic program.

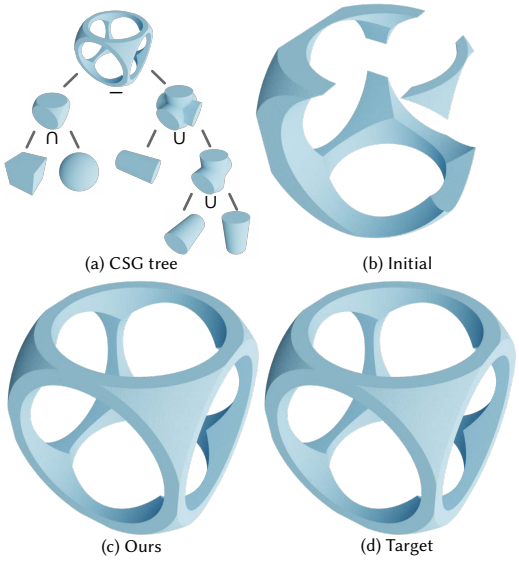


Fig. 18. *Constructive Solid Geometry (CSG).* Given a CSG tree with fixed topology (a), we optimize the position and scale of each primitive to match a target in 3D (d). We can match the target accurately (c) starting with randomly perturbed initial parameters (b).

5.2.2 Non-photorealistic rendering (NPR). NPR techniques such as mosaicing and painterly rendering abstract images using a collection of primitives. Our method enables these applications by optimizing the primitive's parameters through gradient-based optimization.

Mosaicing. Image mosaicing captures the overall structure of an image using a mosaic of tiles. While Haeberli [1990] pioneered the use of Voronoi diagrams through random point placement, this approach lacked optimization of point locations, resulting in tiles that failed to align with edge features of the original image; Hausner [2001] overcame this using a post-processing edge-avoidance step.

We represent the image using a Voronoi program (Listing 3) which takes a set of 2D points and their colors as input and outputs the color of the closest point. Figure 19 (top row) shows the resulting mosaic, where tile edges align with image edges. We can also directly extend our approach to mosaics with complicated boundaries warped with Perlin noise, see Fig. 19 (bottom row).

Painterly rendering. Painterly rendering [Hertzmann 1998] represent images through a collection of brush strokes or other primitives. Li et al. [2020] produce painterly rendering by optimizing simple primitives (ellipses, polygons, and Bézier curves), but their approach is difficult to extend to more complicated primitives. We handle all the primitives they can (Fig. 20) as well as others that they cannot (Fig. 12, two rightmost columns). See our program in Listing 4.

Cel shading. Our method also supports NPR-based shading programs like cel shading and estimates derivatives with camera locations and cel thresholds automatically. These can be used to match target images provided by artists, which we demonstrate in Fig. 21.

5.2.3 Differentiable ray marching. Geometry reconstruction from multi-view images commonly represents geometry as an implicit function, typically a signed distance function (SDF), maps its zero-level set to a volume via a *blurring* operation and then render images through volumetric ray-marching [Yariv et al. 2021; Wang et al. 2021; Miller et al. 2024]. The volumetric mapping is crucial since it converts the discontinuous boundary derivative (for the surface) which cannot be handled by standard auto-diff systems into a continuous

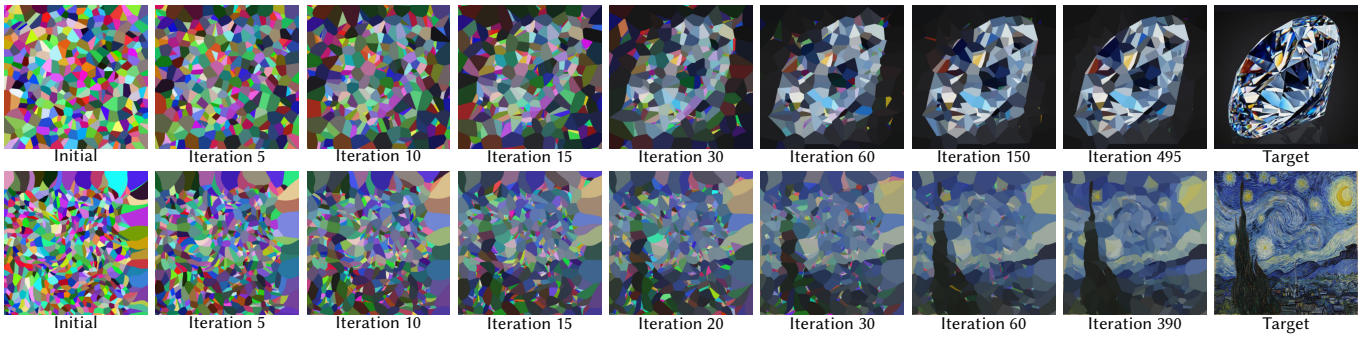


Fig. 19. *Mosaicing.* We fit target images (rightmost) using two different mosaic programs: a Voronoi mosaic program List. 3 (top row) and a Perlin noise-warped mosaic program (bottom row). Starting from random point locations and colors (leftmost), we optimize these parameters using an L_2 loss to match the target. The derivatives with respect to point locations arise solely from the boundary term, guiding the points to automatically align with the target's edges over the course of optimization (middle columns).

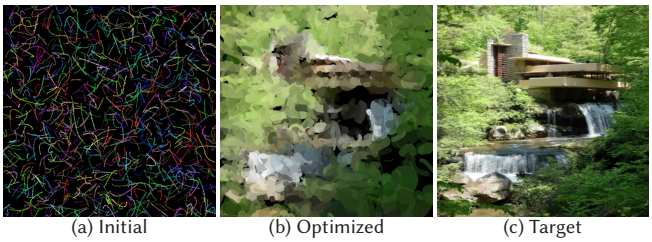


Fig. 20. *Painterly rendering with Bézier curves.* Since our method computes derivatives of arbitrary differentiable implicit boundaries automatically, it supports optimizing Bézier curves (we use a quadratic Bézier's here). Given a target (c) and a random initialization (a), our method produces a painterly rendering of the target (b) through gradient-based optimization.

```
p, opacity, prim_color = // params that define
     primitive geometry, opacity and color
     respectively
float3 final_c = 0.0;
float alpha_acc = 1.0;
[MaxIters(N_PRIMS)]
for (int i = 0; i < N_PRIMS; i++) {
  int t = primitive_type[i];
  float o = opacity[i];
  float3 c = prim_color[i];
  float impl_val = 0.0;
  if (t == 0) { // Ellipse primitive
    impl_val = ellipse_implicit(x, p, i);
  } else if (t == 1) { // Triangle primitive
    impl_val = triangle_implicit(x, p, i);
  } else if (t == 2) {
    // Other primitives like beziers, polygons etc.
  }
  [Disc]
  if (impl_val < 0.0) {
    if (M_PW_CONST) region = hash(region, 2*i)
    final_c += alpha_acc * o * c;
    alpha_acc *= (1.0 - o);
  }
}
return final_c;
```

Listing 4. We can also write a differentiable rasterizer in our system that supports a wide variety of primitives given just their implicit functions.

area derivative (for the volume). Our method skips this volume mapping and directly computes the boundary derivative.

Some previous works [Vicini et al. 2022; Bangaru et al. 2022] also directly estimate the boundary derivatives by reparameterizing the integrand's domain and others blur the boundary only for the derivative (not for the primal) [Wang et al. 2024]. All these methods are specialized to SDFs and can support high-order light transport effects. However, they all require specialized routines to compute derivatives, while we do not.

We take a different approach and show that differentiating a simple ray-marching loop with an arbitrary implicit function (not necessarily an SDF) using our method can recover geometry well. The program in Listing 5 steps along the ray and returns the color at the first point inside the object. Even with this simple setup, we are able to recover geometry well in challenging cases like Fig. 22. Here, the implicit function is a trilinearly interpolated scalar-valued grid that is not restricted to be an SDF. Since our method is not limited to implicit functions of a specific form, it can be directly applied to optimize other implicit geometries.

5.2.4 Differentiable 3D rasterization. In the differentiable rendering literature, there exist several works focused on rasterization, *e.g.* Laine et al. [2020] rasterize triangles and Kerbl et al. [2023] rasterize 3D Gaussians. These systems are highly performant and support a large number of primitives. However, implementing these systems requires significant effort, especially implementing routines for derivative estimation, which is error-prone.

Instead, we can implement a general-purpose rasterizer in our system. While it is highly performant too, its goal is not to outperform the highly-engineered systems discussed above, but rather to minimize user effort during implementation. Given a set of sorted and projected primitives, we can directly differentiate the rasterization loop below to compute their boundary derivatives (List. 4).

The code in Listing 4 has `if-else` statements to select between primitive types that are not marked with the `Disc` tag, these do not affect derivative computation. This rasterizer is the backbone of several of our applications like painterly rendering (which supports vector graphics primitives such as lines, Bezier curves, polygons,

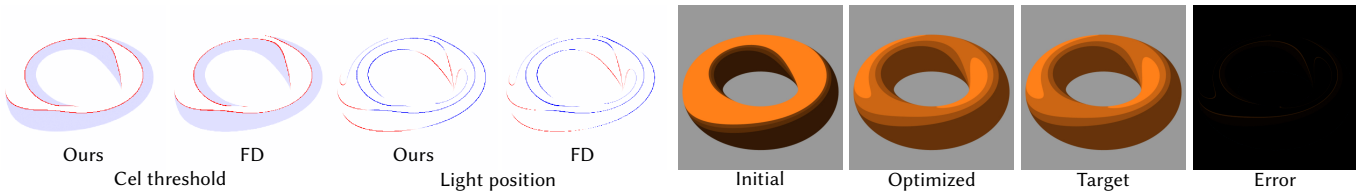


Fig. 21. *Cel shading.* Recovering cel thresholds from target images (*e.g.* provided by an artist) can reduce the manual effort in hand-tuning shader parameters. In general, the shading discontinuities are not parameterized easily, so automatically sampling them is crucial. (Left) We differentiate with respect to cel thresholds and light positions; our derivatives match finite differences (FD) closely. FD shows some artifacts due to the difficulty in tuning ϵ. (Right) We demonstrate that these derivatives can be used to recover the unknown cel thresholds of a shader given a target raster image.

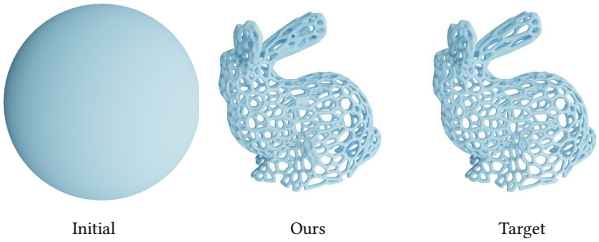


Fig. 22. *Differentiable ray-marching of implicit surfaces.* Our method automatically computes boundary derivatives of a ray-marching shader (List. 5) with arbitrary differentiable implicit geometry (a non-SDF trilinearly-interpolated grid in this example). The goal is to recover the high-genus target geometry given multi-view images. Starting from a sphere and using Chang et al. [2024]'s optimizer, we recover the challenging target geometry, without any specialized boundary sampling routines.

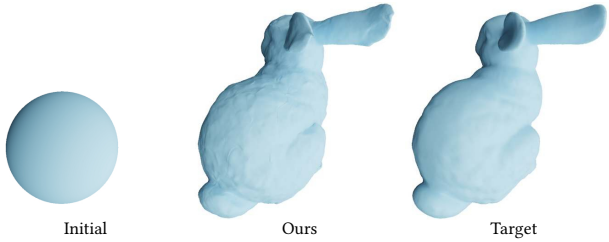


Fig. 23. *Differentiable triangle mesh rasterization.* Given multi-view images and known material and lighting, we optimize an initial sphere triangle mesh to a bunny using differentiable triangle rasterization and Laplacian preconditioning [Nicolet et al. 2021]. Unlike prior work which requires specialized handling of visibility discontinuities, ours does not require any special handling; it automatically samples the boundary and computes derivatives.

```
org, dir, dt = ... // ray origin, direction and step size
p = ... // params of implicit geometry and shading
color = ... // output color
[MaxIters(MAX_STEPS)] // ray marching loop
for (int i = 0; i < MAX_STEPS; i++) {
  // Current point on the ray
  float3 pos = org + i*dt*dir;
  // Discontinuous inside-outside test
  float implicit_value = implicit_function(pos, p);
  [Disc,SkipLoop]
  if (implicit_value < 0.0) {
    // SkipLoop treats all ifs in the loop as the
    // same, they are all hashed with index 0
    if (M_PW_CONST) region = hash(region, 0);
    color = shading(pos, p); // Shading computation
    break;
  }
}
return color;
```

Listing 5. We can differentiate multiple if statements, each corresponding to a different depth-plane in the ray-marching for loop, enabling geometry optimization of arbitrary differentiable implicit geometry.

ellipses and circles) discussed in §5.2.2, and differentiable rasterization of triangles and ellipsoids discussed below. In our applications, we additionally combine it with a grid-based acceleration structure, enabling it to scale to hundreds of thousands of primitives.

Triangle meshes. In Fig. 23, we show that our simple differentiable rasterizer can be used to optimize triangle meshes to fit the geometry of an object from multi-view images. Our method can optimize triangle meshes with hundreds of thousands of triangles without any additional engineering effort to compute derivatives. We stress that although our method is not as performant as `nvdiffrast` [Laine et al. 2020] (nor is it as fully-featured), it is much simpler to implement, making it ideal for quick prototyping and exploring non-standard primitives for rasterization.

Ellipsoids. Next, we demonstrate our method's flexibility by rasterizing 3D ellipsoids using the same rasterizer. We fit the ellipsoids to binary-valued multi-view targets, this ensures all geometry-related derivatives are purely from the boundary derivative. Fig. 24 shows an example of occupancy function fitting and Fig. 25 shows an example of shadow art. In this setup, we splat 3D ellipsoids (initialized randomly) as 2D ellipses with constant opacity and fixed color; we then alpha blend them to compute pixel colors. Differently from Kerbl et al. [2023], we use ellipsoids instead of 3D Gaussians and so there is no continuous opacity fall-off in screen space, it discontinuously drops to zero at the boundaries of the 2D ellipses, which ensures the geometry-related derivatives are only due to discontinuities. Our method is able to fit the binary targets in both of the examples with high accuracy.

5.2.5 *Inverse swept surfaces.*

In Fig. 26, we solve the inverse of the swept surface problem from Fig. 17. Given a swept surface and a

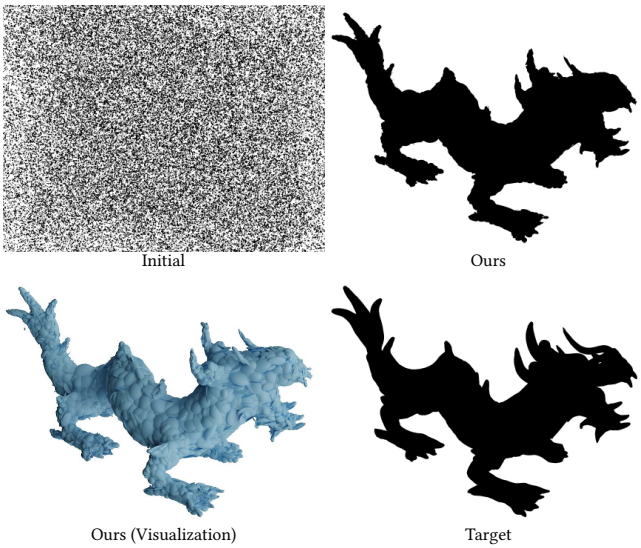


Fig. 24. *Differentiable ellipsoid rasterization.* The goal is to learn 3D object occupancy represented as constant opacity 3D ellipsoids, given multi-view binary images. Starting from a set of randomly initialized ellipsoids, they are splatted onto the screen as 2D ellipses and alpha blended to form the pixel intensity. All ellipsoid derivatives excluding opacity (scaling, rotation and translation) are *purely discontinuous*. Using our method, all parameters can be optimized without any interior or continuous derivatives. To refine the number of ellipsoids, we add and remove ellipsoids as needed, similar to 3D Gaussian Splatting [Kerbl et al. 2023].

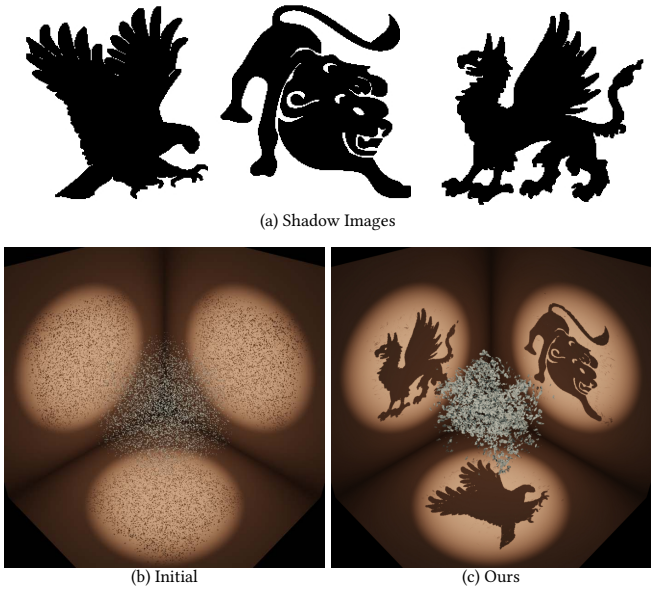


Fig. 25. *Shadow art using differentiable ellipsoid rasterization.* (a) Given 3 target images, (c) the goal is to construct a set of opaque or translucent 3D ellipsoids whose shadows cast by 3 orthogonal spotlights onto 3 walls form each image, in the spirit of Mitra and Pauly [2009]. (b) To fit each shadow image, a set of randomly initialized 3D ellipsoids are differentiably rasterized to 2D ellipses from the position and direction of each spotlight, using the same method as in Figure 24. The size of the ellipsoids in (b) are scaled up by 5× for visualization.

trajectory, we recover the shape of the brush that produced the swept surface. We model the brush as a binary-valued grid (from List. 1) initialized randomly and optimize for its parameters to match the swept surface under an L_2 loss. Our method is able to recover the stroke parameters with a high degree of accuracy.

5.2.6 Discontinuous texture optimization. While popular differentiable renderers [Jakob et al. 2022; Li et al. 2018; Zhang et al. 2020; Laine et al. 2020] can all compute boundary derivatives with respect to geometry discontinuities, they only support continuous textures, even though textured objects in the real world are often piecewise continuous. To demonstrate the ability of our differentiable rasterizer to support discontinuous textures, we optimize the albedo texture of the earth in Figure 27, where the albedo is discontinuous across the land and water boundary. For every point on the surface of the earth, if the point is on the land, the albedo is retrieved from a texture map as usual, or else if it is water, the albedo is a single water color. The land and water boundary is represented using a bilinearly interpolated thresholded grid. Compared to continuous textures, discontinuous textures can provide unique guarantees (e.g., the water color is constant) and enable semantic user editing (see Figure 27, rightmost column).

5.3 Other properties and ablations

We discuss some properties of our algorithm and demonstrate its sensitivity to hyperparameters next.

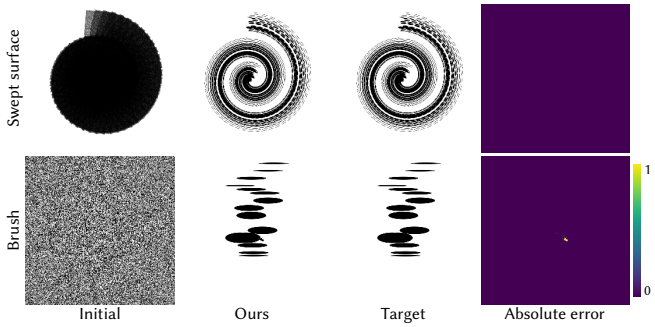


Fig. 26. *Inverse 2D Swept Surface.* Given a target swept surface and the same brush path as in Figure 17, we aim to optimize the brush shape (bottom row), using a bilinearly interpolated thresholded grid as in Figure 15. Starting from a randomly initialized grid, our method can fit the target image accurately up to minor ambiguity due to overlap in the swept brush.

5.3.1 Our method is occlusion-aware. An interesting consequence of our method's detection of discontinuities in image-space is that it automatically reasons about occlusion. In Fig. 28, we have a scene with a large number of overlapping (and fully occluded) primitives. Our piecewise-constant function f_pc produces an output similar to the forward rendering (Fig. 28a) but with constant region identifiers instead of constant colors; it automatically culls the occluded parts of the boundary. Thus, segment snapping reliably detects only the

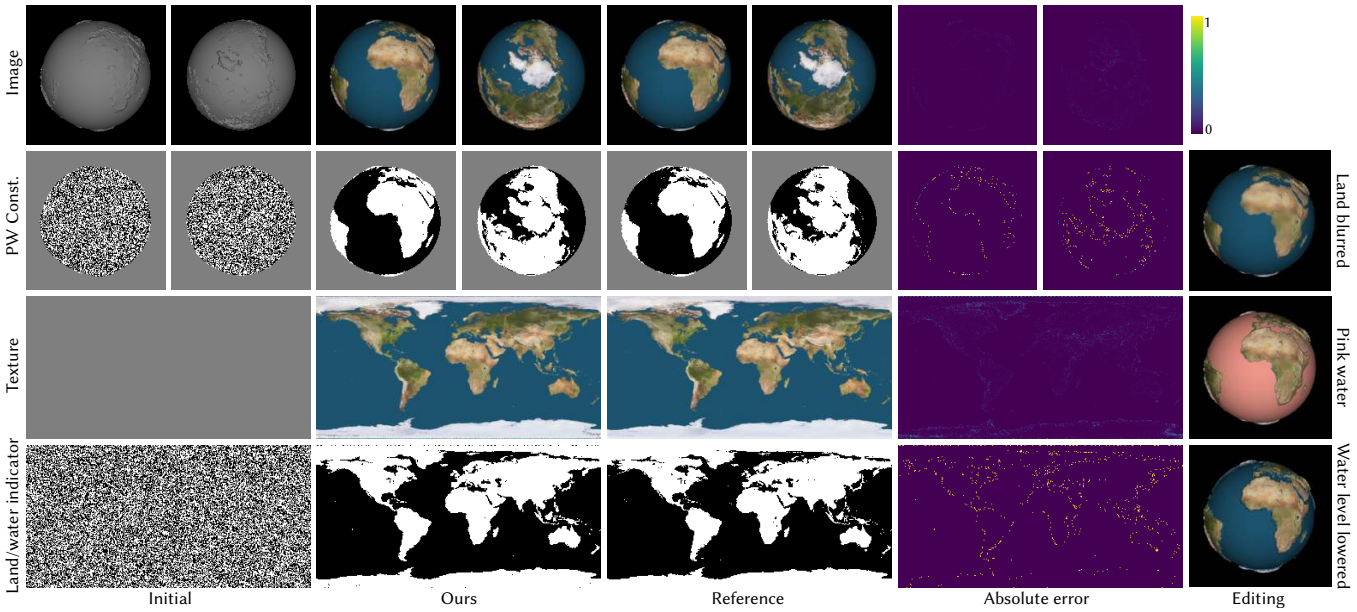


Fig. 27. *Discontinuous texture optimization.* Given 6 multi-view images of the earth (2 shown in top row), with known geometry, uv-mapping, and lighting, we fit a discontinuous texture so that the land color is spatially-varying while the water color is constant. The discontinuous texture is represented by a bilinearly interpolated thresholded grid to determine land or water (indicator, bottom row), a texture for the land, and a single color for the water. We composite all into a single texture image for visualization (third row). We also show the result of the piecewise constant transform from §4.2 with three colors: white, black, and gray (second row). Starting from a randomly initialized grid and constant land and water colors (left), our method can accurately fit the texture and water color, as well as learn the land and water boundary. We use our discontinuous texture representation to easily edit land and water-specific parts of the scene (rightmost column): we can blur the land texture while maintaining the land and water boundary (top), change the color of the water (middle), or even displacement map the geometry to lower the water level (bottom).

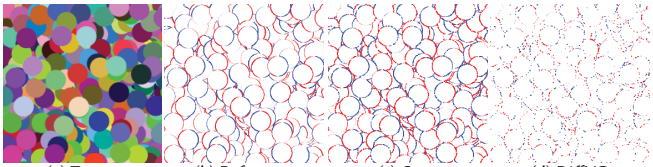


Fig. 28. *Image-space boundary sampling.* Our method samples the boundary in image-space, where occluded boundary are not visible and thus not sampled. In contrast, DiffVG starts by sampling an object and then a point on its boundary; in this scene, most boundary points are occluded, so this results in sparse derivatives.

unoccluded parts resulting in a high-quality derivative Fig. 28b,c. In contrast, specialized-routine-based boundary samplers [Li et al. 2020] sample an object and then a point on its boundary without taking occlusion into account, resulting in sparse derivatives Fig. 28d.

5.3.2 Hyperparameter ablations. To show the effect of our hyperparameters, we compare derivative error with finite differences in Figure 29. It practically demonstrates that our method converges to the reference derivative value with increasing sample count. We globally set $k = 14$ for kernel density estimation and the number of bisection iterations to 15, which we demonstrate are sufficient for derivative estimation.

6 LIMITATIONS AND FUTURE WORK

Our method makes some assumptions on the input function, which limits its applicability in some settings and may lead to incorrect results in others. Handling these limitations is an interesting avenue for future work. We detail them in the following paragraphs.

Higher dimensions. Our method snaps an initial set of line segments stratified over a regular grid onto discontinuities. It requires evaluating f_{pc} at least once at all these locations. The total number of initial samples scales exponentially with the input dimension d for a fixed number of samples per dimension, limiting our method's applicability to higher-dimensional settings.

Inaccurate discontinuity detection. Our method relies on bisection search to detect discontinuities resulting in an exponential decrease in the distance to the discontinuity with the number of steps. However, if the number of bisection steps is insufficient, a discontinuity may be detected at a point that is not close enough to the discontinuity, resulting in additional bias (in practice, we have not yet found this to be an issue with 15 or more bisection steps).

Incorrect branch assignment. Branch assignment maps samples x to the branch index i along the function evaluation path that minimizes $|g_i(x)|$. If there are multiple g_i's close to zero, this can result in incorrect branch assignment. Although it is possible to construct

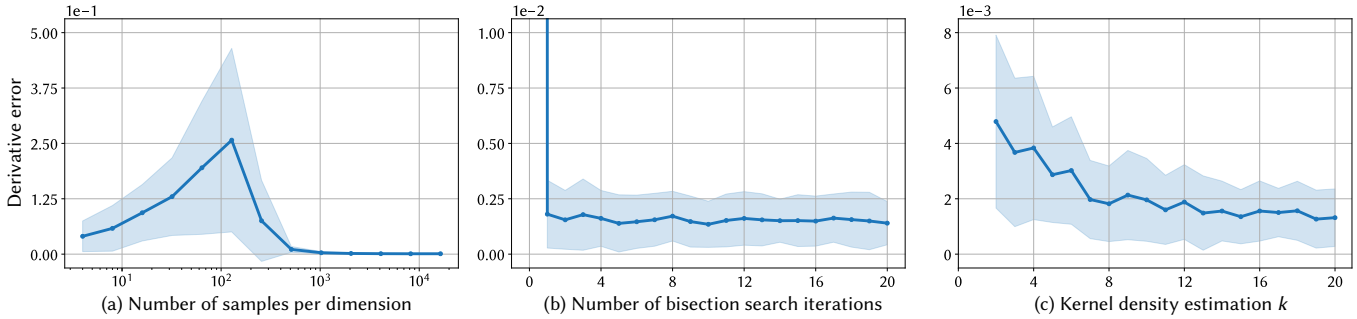


Fig. 29. *Hyperparameter ablations.* We show the mean and standard deviation of the absolute error (compared against finite differences) of the derivative of the shape from Figure 9 for our three hyperparameters: (a) the number of segment samples, (b) the number of bisection search iterations for segment snapping, and (c) the number of neighbors k for kernel density estimation. Our method is consistent and converges to the reference derivative with more samples (a); a small number of bisection iterations ($10 - 15$) is typically enough to locate the boundary; kernel density estimation is accurate for $10 < k < 20$.

a failure case (*e.g.*, introducing $g_i(x) = e^{-10}$ into the program), we have not found it to be a practical issue in our applications.

Sources of randomness. Applications that rely on random numbers seed them uniquely for each point in the domain. This seeding process is discontinuous, breaking the assumption that all discontinuities are expressed as `[Disc]` if conditions, which limits our applicability to these settings. Developing program transformations that can handle these scenarios will unlock several new applications and is an exciting avenue for future work.

Limitations of implementation. Our compiler produces piecewise-constant transformation. To enable this transformation, all loops must have static upper bounds on iterations (as is required by SLANG.D), no mutable state or side effects to global memory, and no calls to dynamically dispatched functions, as these prevent the compiler from statically determining the total number of branch operators and otherwise make the transformation infeasible. Future implementations could potentially compute the transformation dynamically at runtime to avoid these limitations.

Uniform boundary sampling. Since we assume the boundary functions g_i to be continuous, we can cast boundary sampling as a (zero) level-set sampling problem. We do so by segment snapping (§4.3) followed by kernel density estimation (§4.5) to account for non-uniformities. Recent works [Ling et al. 2025; Chiu 2022] have explored efficient methods to *uniformly* sample level-sets through ray-casting and Markov Chain Monte Carlo. Replacing segment snapping with them would obviate the need for kernel density estimation and may be an interesting avenue for future work.

7 CONCLUSION

We present the first boundary sampling algorithm that can handle decision boundaries that are not parameterized by the user. Our method works automatically and requires almost zero user guidance, on a standard shader programming language [He et al. 2018; Bangaru et al. 2023]. We show many applications that were not possible before, due to the lack of a good derivative computation method. We believe our work is a significant step towards general differentiation methods for arbitrary user-defined functions and shaders.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants 2341952, 2105806 and 2402583. We also acknowledge gifts from Adobe, Google, Qualcomm and Rembrand, the Ronald L. Graham Chair and the UC San Diego Center for Visual Computing. We are grateful to the following sources for textures and 3D models: Planet Pixel Emporium (Fig. 27), the Stanford Graphics Group Shadow Art project (Figs. 1, 25), the Stanford 3D Scanning Repository (Figs. 1, 16, 22–24), and Wikimedia Commons (Figs. 1, 16, 19, 20).

REFERENCES

Gaurav Arya, Moritz Schauer, Frank Schäfer, and Chris Rackauckas. 2022. Automatic differentiation of programs with discrete randomness. In *Advances in Neural Information Processing Systems*. Article 758, 13 pages.

Sai Bangaru, Michael Gharbi, Tzu-Mao Li, Fujun Luan, Kalyan Sunkavalli, Milos Hasan, Sai Bi, Zexiang Xu, Gilbert Bernstein, and Fredo Durand. 2022. Differentiable Rendering of Neural SDFs through Reparameterization. Article 22, 9 pages.

Sai Bangaru, Jesse Michel, Kevin Mu, Gilbert Bernstein, Tzu-Mao Li, and Jonathan Ragan-Kelley. 2021. Systematically Differentiating Parametric Discontinuities. *ACM Trans. Graph. (Proc. SIGGRAPH)* 40, 107 (2021), 107:1–107:17.

Sai Bangaru, Lifan Wu, Tzu-Mao Li, Jacob Munkberg, Gilbert Bernstein, Jonathan Ragan-Kelley, Fredo Durand, Aaron Lefohn, and Yong He. 2023. SLANG.D: Fast, Modular and Differentiable Shader Programming. *ACM Transactions on Graphics (SIGGRAPH Asia)* 42, 6 (December 2023), 1–28. https://doi.org/10.1145/3618353

Sai Praveen Bangaru, Tzu-Mao Li, and Frédo Durand. 2020. Unbiased Warped-Area Sampling for Differentiable Rendering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 39, 6 (2020), 245:1–245:18.

Yash Belhe, Michaël Gharbi, Matthew Fisher, Iliyan Georgiev, Ravi Ramamoorthi, and Tzu-Mao Li. 2023. Discontinuity-aware 2D neural fields. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 41, 6 (2023). https://doi.org/10.1145/3550454.3555484

Tyrus Berry and Timothy Sauer. 2017. Density estimation on manifolds with boundary. *Computational Statistics and Data Analysis* 107 (2017), 1–17. https://doi.org/10.1016/j.csda.2016.09.011

Quentin Berthet, Mathieu Blondel, Olivier Teboul, Marco Cuturi, Jean-Philippe Vert, and Francis Bach. 2020. Learning with differentiable perturbed optimizers. In *Advances in Neural Information Processing Systems*. Article 797, 12 pages.

Wesley Chang, Xuanda Yang, Yash Belhe, Ravi Ramamoorthi, and Tzu-Mao Li. 2024. Spatiotemporal Bilateral Gradient Filtering for Inverse Rendering. In *ACM SIGGRAPH Asia 2024 Conference Proceedings* (Tokyo, Japan) *(SIGGRAPH Asia '24)*. Association for Computing Machinery, New York, NY, USA, Article 70, 11 pages. https://doi.org/10.1145/3680528.3687606

Swarat Chaudhuri and Armando Solar-Lezama. 2010. Smooth interpretation. In *Conference on Programming Language Design and Implementation (PLDI)*. 279–291.

Erica Chiu. 2022. Uniform Sampling over Level Sets.

Pedro H Azevedo de Amorim and Christopher Lam. 2022. Distribution theoretic semantics for non-smooth differentiable programming. *arXiv preprint arXiv:2207.05946* (2022).

Martin de La Gorce, David J Fleet, and Nikos Paragios. 2011. Model-based 3D hand pose estimation from monocular video. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 9 (2011), 1793–1805.

Thomas Deliot, Eric Heitz, and Laurent Belcour. 2024. Transforming a Non-Differentiable Rasterizer into a Differentiable One with Stochastic Gradient Estimation. *Proc. ACM Comput. Graph. Interact. Tech. (Proc. I3D)* 7, 1, Article 3 (2024), 16 pages.

Michael Fischer and Tobias Ritschel. 2023. Plateau-Reduced Differentiable Path Tracing. In *Computer Vision and Pattern Recognition*. 4285–4294.

Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (second ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Paul Haeberli. 1990. Paint by numbers: abstract image representations. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques* (Dallas, TX, USA) *(SIGGRAPH '90)*. Association for Computing Machinery, New York, NY, USA, 207–214. https://doi.org/10.1145/97879.97902

Alejo Hausner. 2001. Simulating decorative mosaics. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. Association for Computing Machinery, New York, NY, USA, 573–580. https://doi.org/10.1145/383259.383327

Yong He, Kayvon Fatahalian, and Tim Foley. 2018. Slang: Language Mechanisms for Extensible Real-time Shading Systems. *ACM Trans. Graph. (Proc. SIGGRAPH)* 37, 4 (2018), 141:1–141:13.

Aaron Hertzmann. 1998. Painterly rendering with curved brush strokes of multiple sizes. In *SIGGRAPH*. 453–460.

Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, Merlin Nimier-David, Delio Vicini, Tizian Zeltner, Baptiste Nicolet, Miguel Crespo, Vincent Leroy, and Ziyi Zhang. 2022. *Mitsuba 3 renderer*. https://mitsuba-renderer.org.

Sham M. Kakade and Jason D. Lee. 2018. Provably correct automatic subdifferentiation for qualified programs. In *Advances in Neural Information Processing Systems*. 7125–7135.

Animesh Karnewar, Tobias Ritschel, Oliver Wang, and Niloy Mitra. 2022. ReLU fields: The little non-linearity that could. In *SIGGRAPH Conference Proceedings*. 1–9.

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 2023. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Transactions on Graphics* 42, 4 (July 2023). https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/

Justin N. Kreikemeyer and Philipp Andelfinger. 2023. Smoothing Methods for Automatic Differentiation Across Conditional Branches. *IEEE Access* 11 (2023), 143190–143211.

Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. 2020. Modular Primitives for High-Performance Differentiable Rendering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 39, 6 (2020).

Quentin Le Lidec, Ivan Laptev, Cordelia Schmid, and Justin Carpentier. 2021. Differentiable rendering with perturbed optimizers. In *Advances in Neural Information Processing Systems*, Vol. 34. 20398–20409.

Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2020. On Correctness of Automatic Differentiation for Non-Differentiable Functions. In *Advances in Neural Information Processing Systems*.

Wonyeol Lee, Hangyeol Yu, and Hongseok Yang. 2018. Reparameterization gradient for non-differentiable models. In *Advances in Neural Information Processing Systems*. 5553–5563.

Alexander K. Lew, Mathieu Huot, Sam Staton, and Vikash K. Mansinghka. 2023. ADEV: Sound Automatic Differentiation of Expected Values of Probabilistic Programs. *Proc. ACM Program. Lang.* 7, POPL, Article 5 (2023), 33 pages.

Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. 2018. Differentiable Monte Carlo Ray Tracing through Edge Sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 6 (2018), 222:1–222:11.

Tzu-Mao Li, Michal Lukáč, Gharbi Michaël, and Jonathan Ragan-Kelley. 2020. Differentiable Vector Graphics Rasterization for Editing and Learning. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 39, 6 (2020), 193:1–193:15.

Selena Ling, Abhishek Madan, Nicholas Sharp, and Alec Jacobson. 2025. Uniform Sampling of Surfaces by Casting Rays. arXiv:2506.05268 [cs.GR] https://arxiv.org/abs/2506.05268

Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. 2019. Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning. *International Conference on Computer Vision* (2019).

Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. 2019. Reparameterizing discontinuous integrands for differentiable rendering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 38, 6 (2019), 228.

Y.P Mack and M Rosenblatt. 1979. Multivariate k-nearest neighbor density estimates. *Journal of Multivariate Analysis* 9, 1 (1979), 1–15. https://doi.org/10.1016/0047-259X(79)90065-4

Jesse Michel, Kevin Mu, Xuanda Yang, Sai Praveen Bangaru, Elias Rojas Collins, Gilbert Bernstein, Jonathan Ragan-Kelley, Michael Carbin, and Tzu-Mao Li. 2024. Distributions for Compositionally Differentiating Parametric Discontinuities. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 893–922.

Bailey Miller, Hanyu Chen, Alice Lai, and Ioannis Gkioulekas. 2024. Objects as Volumes: A Stochastic Geometry View of Opaque Solids. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 87–97.

N. J. Mitra and M. Pauly. 2009. Shadow Art. *ACM Transactions on Graphics* 28, 5 (2009). to appear.

Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. 2020. Monte Carlo Gradient Estimation in Machine Learning. *J. Mach. Learn. Res.* 21, 132 (2020), 1–62.

Baptiste Nicolet, Alec Jacobson, and Wenzel Jakob. 2021. Large Steps in Inverse Rendering of Geometry. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 40, 6 (Dec. 2021). https://doi.org/10.1145/3478513.3480501

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*.

Tao Qin, Tie-Yan Liu, and Hang Li. 2010. A general approximation framework for direct optimization of information retrieval measures. *Inf. Retr.* 13, 4 (2010), 375–397.

Ingo Rechenberg and Manfred Eigen. 1973. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog Verlag.

Osborne Reynolds, Arthur William Brightmore, and William Henry Moorby. 1903. *The sub-mechanics of the universe*. Vol. 3. University Press.

Joe Staines and David Barber. 2012. Variational optimization. *arXiv preprint arXiv:1212.4507* (2012).

Hyung Ju Suh, Max Simchowitz, Kaiqing Zhang, and Russ Tedrake. 2022. Do Differentiable Simulators Give Better Policy Gradients?. In *International Conference on Machine Learning*, Vol. 162. 20668–20696.

Delio Vicini, Sébastien Speierer, and Wenzel Jakob. 2022. Differentiable signed distance function rendering. *ACM Trans. Graph. (Proc. SIGGRAPH)* 41, 4 (2022), 1–18.

Peng Wang, Lingjie Liu, Yuan Liu, Christian Theobalt, Taku Komura, and Wenping Wang. 2021. NeuS: Learning Neural Implicit Surfaces by Volume Rendering for Multi-view Reconstruction. *NeurIPS* (2021).

Zichen Wang, Xi Deng, Ziyi Zhang, Wenzel Jakob, and Steve Marschner. 2024. A Simple Approach to Differentiable Rendering of SDFs. In *ACM SIGGRAPH Asia 2024 Conference Proceedings*.

Peiyu Xu, Sai Bangaru, Tzu-Mao Li, and Shuang Zhao. 2023. Warped-Area Reparameterization of Differential Path Integrals. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 42, 6, Article 213 (2023), 18 pages.

Peiyu Xu, Sai Bangaru, Tzu-Mao Li, and Shuang Zhao. 2024. Markov-Chain Monte Carlo Sampling of Visibility Boundaries for Differentiable Rendering. Article 118, 11 pages.

Kai Yan, Christoph Lassner, Brian Budge, Zhao Dong, and Shuang Zhao. 2022. Efficient estimation of boundary integrals for path-space differentiable rendering. *ACM Trans. Graph. (Proc. SIGGRAPH)* 41, 4, Article 123 (2022), 13 pages.

Yuting Yang, Connelly Barnes, Andrew Adams, and Adam Finkelstein. 2022. Aδ: Autodiff for Discontinuous Programs – Applied to Shaders. *ACM Trans. Graph. (Proc. SIGGRAPH)* (2022).

Lior Yariv, Jiatao Gu, Yoni Kasten, and Yaron Lipman. 2021. Volume rendering of neural implicit surfaces. In *Thirty-Fifth Conference on Neural Information Processing Systems*.

Tizian Zeltner, Sébastien Speierer, Iliyan Georgiev, and Wenzel Jakob. 2021. Monte Carlo estimators for differential light transport. *ACM Trans. Graph. (Proc. SIGGRAPH)* 40, 4 (2021), 1–16.

Cheng Zhang, Bailey Miller, Kai Yan, Ioannis Gkioulekas, and Shuang Zhao. 2020. Path-Space Differentiable Rendering. *ACM Trans. Graph. (Proc. SIGGRAPH)* 39, 6 (2020), 143:1–143:19.

Ziyi Zhang, Nicolas Roussel, and Wenzel Jakob. 2023. Projective Sampling for Differentiable Rendering of Geometry. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 42, 6, Article 212 (2023), 14 pages.

Shuang Zhao, Wenzel Jakob, and Tzu-Mao Li. 2020. Physics-Based Differentiable Rendering: From Theory to Implementation. In *SIGGRAPH Courses*. Article 14, 30 pages.

A KERNEL DENSITY ESTIMATION ON MANIFOLDS

Please see Berry and Sauer [2017] (Section 3 in their paper) for a detailed treatment of kernel density estimation on manifolds; we restate the relevant results here for convenience. Our kernel density estimator (Eq. (7)) computes the density of points on a manifold $\partial\Omega$ of codimension 1. Since it uses Euclidean distances in ambient space Ω and not geodesic distances (which account for curvature of the manifold) in $\partial\Omega$ as the metric, it results in bias due to the curvature of the manifold. Nonetheless, the overall estimator, like standard KDE, is consistent. Intuitively, as Berry and Sauer [2017] explain,

```
g_i = ... // some function of x and theta
[Disc]
if (g_i > 0.0) {
    // ... true branch ...
} else {
    // ... false branch ...
}
```

Listing 6. We start with an input program that contains explicitly tagged `[Disc]` (discontinuous) `if-else` statements.

this is because as the bandwidth of the kernel shrinks, the kernel is localized to a very small region within which the Euclidian distance is approximately equal to the geodesic distance.

B SYSTEM IMPLEMENTATION

B.1 Discontinuous program requirements

The program can contain two types of operations. The first type consists of differentiable operations, which include any function supported by SLANG.D's automatic differentiation system, such as arithmetic operations, trigonometric functions and operations with manually specified derivatives. The second type is discontinuous `if-else` statements, which are preceeded with a `[Disc]` tag, a language extension we introduce to mark discontinuities; we detail it in the next subsection.

The program's control flow can include (nested) `for`-loops and each iteration can contain different discontinuities. Function calls are also supported, allowing composition of simpler discontinuous functions into more complex ones. However, recursion is not supported as it can violate the DAG requirement.

Common non-differentiable operations like `min` and `max` must be expressed through explicitly tagged if-else statements. . Non-tagged if-conditions are also supported, but their derivatives will not be computed . The presence of non-tagged if-conditions removes all correctness guarantees for the derivative (even of parameters associated with tagged if-else statements) and requires careful reasoning; we show an example in List. 4.

B.2 Program transformations

The transformation process converts the input discontinuous program into a form that can compute boundary derivatives. We traverse the program's DAG in topological order and assign a unique index i to each branch statement. The input program contains discontinuous branches that evaluate a boundary function $g_i(x, \theta)$ to determine which branch to take (Listing 6). In our system, we implement all function transformations within the *same* program and switch between them using boolean flags. These include the piecewise constant transformation §4.2 when `M_PW_CONST` is set, the branch index detection §4.4 when `M_BRANCH_IDX` is set, the two-sided evaluation transform §4.6 when `M_TWOSIDED` is set, and the original program when all flags are false as shown next.

Piecewise-Constant (`M_PW_CONST`). This mode implements the piecewise-constant transformation from §4.2. It updates a region identifier (`region_id`) for each branch encountered during program execution — this has the same effect as updating the ternary state s

```
1  g_i = ... // some function of x and theta
2
3  // ----- branch index detection -----
4  // map the image-space boundary sample x to the
       branch condition whose boundary it lies on
5  // if the boundary fn has the lowest |g_i| of all
6  // branches thus far, update g_min, idx_min
7  if (M_BRANCH_IDX && abs(g_i) < abs(g_min) {
8      g_min = g_i; // update min boundary fn value
9      idx_min = i; // update branch idx
10 }
11
12 // ----- two-sided eval or original function -----
13 // When M_TWOSIDED is true and this branch (i) is
       the one closest to the boundary (i == idx_min):
14 //   - Choose which side to evaluate based on
       EVAL_TRUE_SIDE flag
15 // Otherwise:
16 //   - Use the original branch condition (g_i)
17 if ((M_TWOSIDED && idx_min == i) ? EVAL_TRUE_SIDE :
       g_i > 0.0) {
18     // ----- piecewise constant evaluation -----
19     if(M_PW_CONST) region_id = hash(region_id, 2*i);
20     // ... original program true branch ...
21 } else {
22     // ----- piecewise constant evaluation -----
23     if (M_PW_CONST)region_id = hash(region_id,2*i+1);
24     // ... original program false branch ...
25 }
```

Listing 7. Transforming a discontinuous branch statement

which tracks the branching sequence taken by the program. Like s, `region_id` is initialized to the same value for all points. During evaluation, the region identifier is updated using a hash function that combines its current value with a unique value for the branch and the side taken: $2i$ if the condition is `true` and $2i+1$ if it is `false`, see L19 and L23 in List. 7.

Any two inputs x_1, x_2 that take the same path through the branches will end up with the same identifier, while points that take different paths will end up with different identifiers (with negligible collision probability due to the 32-bit hash space). We opt for the hash-based approach as it is simple to implement and also allows for custom user-defined piecewise constant transformations §5.2.3.

Branch Index Detection (`M_BRANCH_IDX`). During derivative computation, segment snapping §4.3 uses the piecewise-constant mode above to draw samples on the boundary. In this mode, we assign these samples x to the branch i whose boundary they lie on. We do this, by invoking List. 7 with `M_BRANCH_IDX` set and `g_min`=∞. Now as we evaluate the program, we update `g_min` and `idx_min`, to keep track of the index i that achieves the minimum absolute value $|g_i(x, \theta)|$ for each branch we encounter, see L7-L10 in List. 7. We return both the index `idx_min` which is used for two-sided evaluation and probability density estimation, and the boundary function value `g_min` for boundary velocity estimation.

Two-Sided Evaluation (`M_TWOSIDED`). We use `idx_min` for the two-sided evaluation $f(x^+) - f(x^-)$. For each boundary sample x, we also need to evaluate the integrand value on either side of the boundary. To do this, we first perform the branch index detection above, and then invoke List. 7 with `M_TWOSIDED` set and this time we also

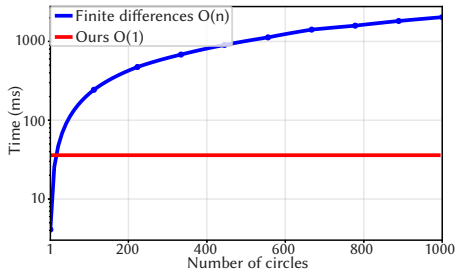


Fig. 30. *Timing comparison.* For the scene in Fig. 12, we compare the derivative computation time of our method with finite differences for a different number of circles. Finite differences scales as $O(n)$, ours as $O(1)$, see discussion in Appendix B.3.

set *idx_min* to the index we retrieved from the branch index detection. Now, during execution, when we encounter the branch for which `idx_min == i`, we forcefully evaluate either the true or false branch using the `EVAL_TRUE_SIDE` flag, see L15-L16 in List. 7. This allows for precise, side-specific evaluation at the boundary, without relying on arbitrary epsilon offsets as used in previous works [Li et al. 2018, 2020].

B.3 Timing comparison with finite differences

Our method computes derivatives using reverse-mode `autodiff` which scales as $O(1)$ with the number of parameters being differentiated, which is more efficient than forward-mode `autodiff` (finite differences) which scales as $O(n)$, we show an example in Fig. 30.