# Aδ: Autodiff for Discontinuous Programs – Applied to Shaders

YUTING YANG, Princeton University, U.S.
CONNELLY BARNES, Adobe Research, U.S.
ANDREW ADAMS, Adobe Research, U.S.
ADAM FINKELSTEIN, Princeton University, U.S.

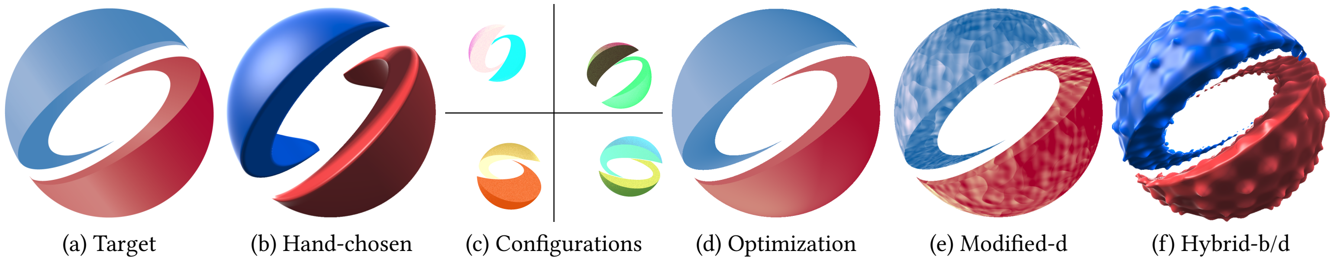


Fig. 1. Our compiler automatically differentiates discontinuous shader programs by extending reverse-mode automatic differentiation (AD) with novel differentiation rules. This allows efficient gradient-based optimization methods to optimize program parameters to best match a target (a), which is difficult to do by hand (b). Our pipeline takes as input a shader program initialized with configurations (c) that look very different from the reference, and converges to be nearly visually identical (d) within 15s. The compiler can also output the shader program with optimized parameters to GLSL, which allows programmers to interactively edit or animate the shader, such as adding texture (e). The optimized parameters can also be combined with other shader programs (e.g. b) to leverage their visual appearance while keeping the geometry close to the reference. For animation results please refer to our supplemental video.

Over the last decade, automatic differentiation (AD) has profoundly impacted graphics and vision applications — both broadly via deep learning and specifically for inverse rendering. Traditional AD methods ignore gradients at discontinuities, instead treating functions as continuous. Rendering algorithms intrinsically rely on discontinuities, crucial at object silhouettes and in general for any branching operation. Researchers have proposed *fully*-automatic differentiation approaches for handling discontinuities by restricting to affine functions, or *semi*-automatic processes restricted either to invertible functions or to specialized applications like vector graphics. This paper describes a compiler-based approach to extend reverse mode AD so as to accept arbitrary programs involving discontinuities. Our novel gradient rules generalize differentiation to work correctly, assuming there is a single discontinuity in a local neighborhood, by approximating the prefiltered gradient over a box kernel oriented along a 1D sampling axis. We describe when such approximation rules are first-order correct, and show that this correctness criterion applies to a relatively broad class of functions. Moreover, we show that the method is effective in practice for arbitrary programs, including features for which we cannot prove correctness. We evaluate this approach on procedural shader programs, where the task is to optimize unknown parameters in order to match a target image, and our method outperforms baselines in terms of both convergence and efficiency. Our compiler outputs gradient programs in TensorFlow, PyTorch (for quick prototypes) and Halide with an optional auto-scheduler (for efficiency). The compiler also outputs GLSL that renders the target image, allowing users to interactively modify and animate the shader, which would otherwise be cumbersome in other representations such as triangle meshes or vector art.

CCS Concepts: • **Mathematics of computing → Automatic differentiation**; • **Software and its engineering → Domain specific languages**.

Additional Key Words and Phrases: Automatic Differentiation, Differentiable Programming, Differentiable Rendering, Domain-Specific Language

**ACM Reference Format:**
Yuting Yang, Connelly Barnes, Andrew Adams, and Adam Finkelstein. 2022. Aδ: Autodiff for Discontinuous Programs – Applied to Shaders. *ACM Trans. Graph.* 41, 4, Article 135 (July 2022), 24 pages. https://doi.org/10.1145/3528223.3530125

## 1 INTRODUCTION

Many graphics and vision optimization tasks rely on gradients. When outputs can be expressed as explicit functions of given parameters, automatic differentiation (AD) can provide gradients. However, most AD methods assume that such functions are continuous with respect to the input parameters, and produce incorrect gradients at discontinuities resulting from if/else branches, for example. Such AD-based methods therefore struggle to optimize functions involving factors like object boundaries, visibility, and ordering.

In certain cases, the gradient at a discontinuity can be computed analytically. For example the derivative of a step function is the *Dirac delta* distribution (informally, infinity at the discontinuity and zero elsewhere). Likewise, the gradient of certain pre-filtered discontinuous functions can be derived analytically as a convolution with Dirac deltas[1]. Building on these properties, this paper proposes a framework and compiler we call Aδ for automatic differentiation of programs – including proper differentiation of the discontinuities.

Authors' addresses: Yuting Yang, Princeton University, U.S., yutingy@princeton.edu; Connelly Barnes, Adobe Research, U.S., cbarnes@adobe.com; Andrew Adams, Adobe Research, U.S., andrew.b.adams@gmail.com; Adam Finkelstein, Princeton University, U.S., af@cs.princeton.edu.

0730-0301/2022/7-ART135
https://doi.org/10.1145/3528223.3530125

[1] Technically, such Dirac delta distributions act on a test function.

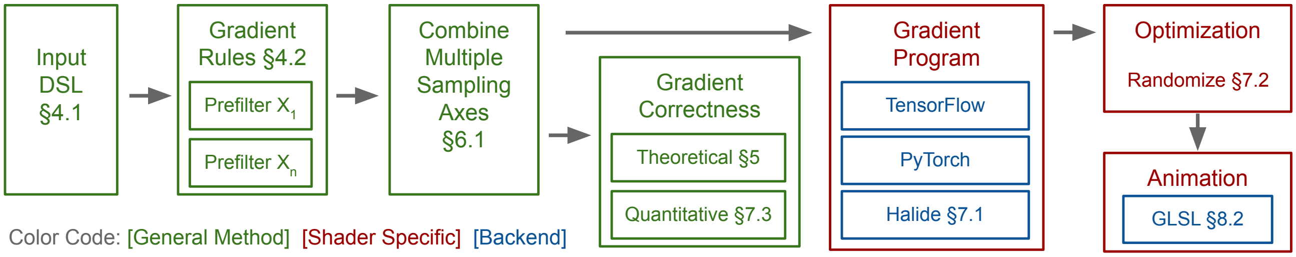


Fig. 2. Overview: green boxes indicate general components; red boxes are components specific to shaders; blue boxes indicate specific backend languages that our compiler outputs to (details in gray). Our compiler takes as input an arbitrary program in our DSL (§ 4.1), and approximates the gradient by pre-filtering a 1D box kernel along sampling axes (§ 4.2). Approximations along multiple sampling axes are later combined (§ 6.1). We verify that our gradients are accurate in two ways: we prove that a subset of programs are first-order correct (§ 5), and we also design a quantitative error metric (§ 7.3) to evaluate any gradient program empirically. For practical applications, our compiler outputs the gradient program to three backends: TensorFlow, PyTorch and Halide. For efficiency we further explore the scheduling space in Halide that trades-off register usage vs memory I/O and provide an optional autotuner (§ 7.1). The gradient program can then be applied to optimization tasks that find optimal parameters for a shader program to best match a reference image. We find it helpful to add per pixel random noise to Dirac parameters, as this makes the discontinuities observable in more pixel locations (§ 7.2). Finally, the program representation with optimized parameters can be output to GLSL, which allows interactive animation.

We develop gradient rules for efficient automatic differentiation with respect to input parameters that discontinuous operators depend on, which we call *Dirac parameters* because their partial derivatives often contain Dirac deltas. (They are defined formally in Section 4.1.)

Graphics researchers have derived several application-specific solutions for differentiating Dirac parameters, targeting specialized domains such spline shapes for vector graphics [Li et al. 2020] or triangle meshes rendered in a path tracer [Bangaru et al. 2020; Li et al. 2018a; Loubet et al. 2019]. While they address these specific domains, they are not readily adapted to arbitrary functions. TEG [Bangaru et al. 2021] on the other hand, systematically differentiates parametric discontinuities on a limited scope of programs. Their system correctly handles discontinuities that are represented by differentiable and invertible functions, and is only fully automatic when the discontinuities are represented by affine transformations: in all other cases, the inversion or reparameterization needs to be provided by the programmer. This leaves out many real world programming patterns such as discontinuity compositions, or discontinuities represented by non-invertible functions.

Similarly to TEG, our method also targets general discontinuous programs, written in our domain specific language (DSL). But instead of proving correctness under a limited set of programs, we make several assumptions that allow us to handle a broader set of programs. Our compiler approximates the pre-filtered gradient over a 1D box kernel, where we denote the kernel orientation as the *sampling axis*, under three assumptions:

- (A1) There is at most one discontinuity between each sample and its nearest neighbor along the sampling axis (a *sample pair*).
- (A2) Function values and some partial derivatives within the computation at a sample pair can be used to estimate gradients at locations between the pair.
- (A3) Most discontinuities can be projected to the sampling axis.

Intuitively, A1 can be easily achieved in most locations with a high enough sample frequency. Similarly, we also show A2 becomes more accurate for higher sample frequencies as continuous functions expressible in our DSL are locally Lipschitz continuous. A2 is the key that allows us to efficiently expand to a larger set of programs. Because we can use function and gradient values at nearby sample locations as proxies, we do not need extra samples to locate the discontinuity, or limit the discontinuity to certain types of functions that can be easily inverted. Additionally, as will be shown in Section 3, A2 also allows efficient reverse-mode AD, because gradients with respect to the sampling axis can be approximated by finite differences. Lastly, A3 allows us to evaluate most discontinuities using minimal samples, along the sampling axis. In case one sampling axis is not sufficient to observe every discontinuity, our framework can also combine approximations from multiple sampling axes.

Beyond proposing gradient rules for efficiently back-propagating through discontinuous programs, this paper investigates other questions such as how to evaluate the quality of our gradient and how to generate efficient GPU code for the gradients.

We demonstrate the applicability of the gradient program in the domain of procedural shader programs capable of rendering graphical designs. Using these gradients we are able to optimize the parameters of such shaders to match target reference images that we found on the Web. The optimization does so more effectively and quickly than using baseline methods such as finite differences, and even for programs that occasionally violate assumptions A1-3.

Figure 2 shows an overview of our framework. The primary contribution of this paper a set of approximate derivative rules that can be applied to a large set of general programs. We show for a subset of programs, the approximation error is bounded by a first order term scaled by the size of the pre-filtering kernel. A second contribution is the implementation of a system that efficiently carries out practical applications in shader programs. A third contribution is the design of a novel error metric to quantitatively evaluate our gradient approximation. Our code is available at: https://github.com/yyuting/Adelta.

Table 1. Comparison between ours and related work on differentiating discontinuous programs: traditional Auto-Differentiation (AD); finite difference (FD); TEG [Bangaru et al. 2021]; differentiable vector graphics [Li et al. 2020] and diffrentiable path tracers (DPT) [Bangaru et al. 2020; Li et al. 2018a; Loubet et al. 2019]. We compare these methods under four criteria: whether they can sample discontinuities, whether the method can reduce to AD in the absence of discontinuities, time complexity in terms of how many evaluations of the original program are needed as a function of parameter dimension $n$, and what set of programs each method can handle. Our method handles every program expressible in our DSL (Section 4.1); AD and finite difference work with arbitrary programs; TEG works with a limited subset of programs whose discontinuities are represented by diffeomorphisms (Diff); while task-specific methods DVG and DPT only apply to their specific tasks: vector graphics (VG) and path tracer (PT) respectively.

| Taxonomy | Ours | AD | FD | TEG | DVG | DPT |
|---|---|---|---|---|---|---|
| Discontinuities | ✓ | × | ✓ | ✓ | ✓ | ✓ |
| Reduce to AD | ✓ | ✓ | × | ✓ | ✓ | ✓ |
| Time Complexity | O(1) | O(1) | O(n) | O(1) | O(1) | O(1) |
| Generality | DSL | All | All | Diff | VG | PT |

## 2 RELATED WORK

*Automatic differentiation of parametric discontinuities.* TEG [Bangaru et al. 2021] systematically differentiates integrals with discontinuities. When the program is posed as integrals of discontinuous functions, TEG correctly differentiates the program by eliminating Dirac deltas residing within the integrals. The remaining integral dimensions are sampled and differentiated using trapezoidal rule. However, the set of programs TEG can correctly handle is restricted. In order to correctly eliminate Dirac deltas, the discontinuities are limited to be represented by differentiable, invertible functions, and TEG can only automatically handle the affine case. All other cases rely on programmer provided inversion or reparameterization. This restriction limits the set of programs that can benefit from their automatic pipeline: composition of discontinuities and non-invertible functions are excluded entirely, and non-affine invertible functions require extra manual effort to define the inverse for each of them. Unlike TEG, our method approximates the gradient of discontinuous programs, with a weaker correctness guarantee of the error being first order in the step size for sufficiently small steps, and we show this theoretical result applies to a larger set of programs. Moreover, Section 8.2 shows empirically that our method can also handle a larger set of shaders than the set we analyze theoretically – and which is expressive enough to reconstruct real world images found online. Table 1 compares our method with TEG and other baselines such as traditional AD ([Moses et al. 2021; Nimier-David et al. 2019]) and finite difference, as well as application-specific methods that directly differentiate discontinuities, discussed next.

*Application-specific differentiation of parametric discontinuities.* Many application-specific methods differentiate pre-filterings of the discontinuous functions. For example, in the domain of vector graphics [Li et al. 2020], path tracers [Bangaru et al. 2020; Li et al. 2018a; Loubet et al. 2019; Zhang et al. 2021b], and other physics-based renderers [Zhou et al. 2021], specialized rules are derived analytically, and algorithms are also designed for efficiently carrying out the computation of these specialized gradients [Nimier-David et al. 2020; Zeltner et al. 2021; Zhang et al. 2021a]. While these manually derived rules are correct and efficient for the particular application, they are limited to a small subset of programs. Our method on the other hand, is general, and can be applied to arbitrary programs expressible in the syntax of our DSL.

*Replacing discontinuities with continuous proxies.* Another strategy for differentiating parametric discontinuities is to use a process such as smoothing to replace the original function with a continuous proxy before taking the gradient. Although similar to pre-filtering, these methods only differentiates the proxy, and does not attempt to sample discontinuities directly. Soft rasterizer [Liu et al. 2019] replaces step discontinuities with sigmoids and builds a continuously differentiable path tracer, but is application-specific and has no formal guarantees for the approximation. Another approach, mean-variance program smoothing [Yang and Barnes 2018], can correctly smooth out a certain class of procedural shader programs, and briefly discusses using AD to derive one approximation term; but it does not investigate differentiation further and is unable to scale to complicated programs. In contrast, our method applies to a broader set of programs, and we show our approximation has low error both by mathematical proof and by evaluating under a quantitative metric. Researchers have also investigated a variety of strategies for converting complicated functions with neural proxies, for example: approximating a "black-box" ISP camera pipeline [Tseng et al. 2019], using neural textures or neural implicit 3D representation for differentiable rendering [Jiang et al. 2020; Niemeyer et al. 2019; Park et al. 2019; Sitzmann et al. 2019; Thies et al. 2019], and using NeRF as a surrogate for geometry and reflectance [Martin-Brualla et al. 2021; Mildenhall et al. 2020]. These representations are inherently differentiable, and leverage all of the recent progress in neural representations, but the resulting representation is a network which is difficult to interpret and manipulate in general ways. In contrast our approach provides gradients in the original program representaion (and does so quickly relative to the training of most neural methods).

*Scheduling efficient Halide kernels.* To allow practical applications, our compiler targets Halide as one of the backends and provides a simplified scheduling space with an optional autoscheduler. While most state-of-the-art Halide autoschedulers focus on efficiently scheduling nested image pipelines and utilize scheduling choices such as tiling and fusion [Adams et al. 2019; Mullapudi et al. 2016; Sioutas et al. 2019], our Halide backend targets the original and gradient program for procedural pixel shaders, and the performance bottleneck in our case is caused by long gradient tapes that can cause register spilling inside GPU kernels. Li et al. [2018b] proposed scheduling options for the gradient of image pipelines. However, their work focuses on efficiently scheduling convolutional scattering and gather operations (e.g. convolution and its gradient), whereas ours focuses on trade-offs between avoiding register spilling and minimizing memory I/O.

## 3 MOTIVATION

We begin by describing a simple example: $f(x, \theta) = H(x + \theta)$. Here $x$ is a *sampling axis* along which we sample discontinuities, and which we will discuss in more depth shortly; and $\theta$ is a parameter for which we wish to obtain a derivative. $H$ is a Heaviside step function that evaluates to 1 when $x + \theta \geq 0$, and 0 otherwise. Mathematically, the gradient of this step function is a Dirac delta distribution $\delta$, which informally evaluates to $+\infty$ at the discontinuity, 0 otherwise, and integrates over the reals to one. In real-world applications, differentiating discontinuous functions is usually approximated by first pre-filtering with kernel to avoid the need to exactly sample the discontinuity, which is measure zero. For example, pre-filtering over a 1D box kernel in the $x$ dimension:

$$\frac{\partial}{\partial\theta}\int H(x'+\theta)\phi(x-x')dx' = \int \delta(x'+\theta)\phi(x-x')dx' = \phi(x+\theta)$$

Here we use $\phi$ to represent the P.D.F. of the uniform continuous distribution $U[-\epsilon, \epsilon]$. The gradient evaluates to $\frac{1}{2\epsilon}$ if $x + \theta \in [-\epsilon, \epsilon]$, and 0 elsewhere. Note that because the discontinuity depends on both $x$ and $\theta$, we can differentiate with respect to (wrt) $\theta$ while pre-filtering along $x$.

A key motivation of our approach is that in many applications, there are a few dimensions that most parametric discontinuities depend on, such as time for audio or physics simulation programs, or the 2D image axes for shader programs. As a result, the computational challenge of sampling discontinuities in high dimensions can be greatly reduced by placing samples along these axes, which are much lower dimension than the entire parameter space. We denote them as *sampling axes.* In principle, sampling axes can be arbitrary, and we do not need *every* discontinuity to be projected on a *single* axis. For a set of sampling axes, as long as discontinuities of interest project to one of them, their gradient will be included.

We propose to approximate the gradient wrt *every* parameter by first prefiltering using a 1D box kernel on the sampling axes. For example, for a continuous function $c$, we can differentiate $H(c(x, \theta))$ pre-filtered by a kernel $\phi(x)$ wrt $\theta$ as follows, assuming $\frac{dc}{dx} \neq 0$ at the discontinuity $x_d$, and apply Dirac Delta's scaling property.

$$\frac{\partial}{\partial\theta}\int H(c(x',\theta))\phi(x-x')dx' = \int \delta(c(x',\theta))\frac{dc}{d\theta}\phi(x-x')dx'$$
$$= \int \frac{\delta(x'-x_d)\frac{dc}{d\theta}}{|\frac{dc}{dx}|}\phi(x-x')dx' = \frac{\frac{dc}{d\theta}}{|\frac{dc}{dx}|}\,|_{x_d}\,\phi(x-x_d) \tag{1}$$

We choose 1D box (boxcar) kernels to minimize the extra compute needed for locating the discontinuity $x_d$ and computing $\phi(x - x_d)$. Previous work either relies on simplifying assumptions such as $c$ is invertible [Bangaru et al. 2021], or has to use recursive algorithms to find exact location of $x_d$ [Li et al. 2020]. Unlike previous work, because a box kernel $\phi$ is piece-wise constant, we can simplify computing $\phi(x - x_d)$ into sampling whether $x - x_d \in [-\epsilon, \epsilon]$.

## 4 OUR MINIMAL DSL AND GRADIENT RULES

This section formally defines the set of programs expressible in a minimalistic formulation of our domain specific language (DSL). We present the minimal DSL first to simplify the exposition, but later in the paper we extend our DSL to include a ternary `if` or `select` function in Section 6.2 and a ray-marching construct in Appendix D. After presenting the minimal DSL, we will present our gradient rules which can be used to extend typical reverse-mode AD.

Table 2. Gradient rules for our compiler and traditional AD. *: in our compiler implementation (but not our theoretical results), to avoid numerical instability, the division in our function composition rule is safeguarded, and evaluates to $h'$ whenever $|g^+ - g^-| \leq 10^{-4}$.

| Op | Ours (k = O) | AD (k = AD) |
|---|---|---|
| $\frac{\partial_k H(g)}{\partial\theta}$ | $\begin{cases} \frac{\frac{\partial_k g}{\partial\theta}}{\lvert g^+ - g^- \rvert} & \text{if } H(g^+) \neq H(g^-) \\ 0 & \text{else} \end{cases}$ | 0 |
| $\frac{\partial_k (g+h)}{\partial\theta}$ | $\frac{\partial_k g}{\partial\theta} + \frac{\partial_k h}{\partial\theta}$ | $\frac{\partial_k g}{\partial\theta} + \frac{\partial_k h}{\partial\theta}$ |
| $\frac{\partial_k (g\cdot h)}{\partial\theta}$ | $\frac{1}{2}(h^+ + h^-)\frac{\partial_k g}{\partial\theta} + \frac{1}{2}(g^+ + g^-)\frac{\partial_k h}{\partial\theta}$ | $h\frac{\partial_k g}{\partial\theta} + g\frac{\partial_k h}{\partial\theta}$ |
| $\frac{\partial_k h(g)}{\partial\theta}$ | $\begin{cases} h'\frac{\partial_k g}{\partial\theta} & \text{if } h(g) \text{ is statically differentiable} \\ \frac{h(g^+)-h(g^-)}{g^+ - g^-}\frac{\partial_k g}{\partial\theta} & \text{otherwise}^* \end{cases}$ | $h'\frac{\partial_k g}{\partial\theta}$ |

### 4.1 Our Minimal DSL Syntax

We formally define the set of programs expressible in our minimal DSL using Backus-Naur form. The set of all programs expressible in our language can be defined as below, where $C$ represents any constant scalar value, $x$ represents any variable that is a sampling axis, $\theta$ represents any parameters we want to differentiate wrt, and $f$ are continuous atomic functions supported by our DSL (presently, `sin`, `cos`, `exp`, `log`, and `pow` with constant exponent).

$$e_d ::= C \mid x \mid \theta \mid e_d + e_d \mid e_d \cdot e_d \mid H(e_d) \mid f(e_d)$$

Using this syntax, we formally define *Dirac parameters* as any parameters $\theta$ that expressions of the form of $H(e_d)$ depend upon.

### 4.2 Our Gradient Rules

This subsection formally defines our pre-filtering process, and presents novel gradient rules that approximate the derivatives of the pre-filtered function. We define a function $f : \text{dom}(f) \longrightarrow \mathbb{R}$ that maps a subset of $\mathbb{R}^{n+1}$ to a scalar output in $\mathbb{R}$. For prefiltering purposes, we assume $f$ to be locally integrable. We define dom$(f)$ to be the set $\{(x, \vec{\theta}) \in \mathbb{R}^{n+1} : f \text{ is not computationally singular at } (x, \vec{\theta})\}$, where computationally singular is defined in Appendix A.1 as the points where any intermediate value in the program is undefined. Note that our framework and implementation also support multi-dimensional outputs $\mathbb{R}^k$ such as RGB colors for $k = 3$, but since the same gradient process is applied to each output independently, for a simpler notation but without loss of generality we assume the codomain of $f$ is $\mathbb{R}$. In our compiler, multidimensional outputs are implemented for efficiency using a *single* reverse mode pass as described in Section 6. We pre-filter $f$ by convolving with a box kernel along sampling axis $x$, giving pre-filtered function $\hat{f}$.

$$\hat{f}(x, \vec{\theta}; \epsilon) = \frac{1}{(\alpha+\beta)\epsilon}\int_{x-\alpha\epsilon}^{x+\beta\epsilon} f(x', \vec{\theta})dx'$$

Here $\vec{\theta}$ is the vector of parameters that we wish to differentiate with respect to, and $\alpha, \beta$ are non-negative constants for each pre-filtering with $\alpha + \beta > 0$, that control the box's location.

In our approximation, we locate discontinuities by placing two samples at each end of the kernel support, denoted as $f^+$ and $f^-$ respectively. When $\epsilon$ is small enough, $f^+$ and $f^-$ can be viewed as approximating the right and left limit of $f$.

Both ours and AD approximate the derivative of functions, with differentiation rules summarized in Table 2. We further denote gradient approximations as $\partial_k$, where $k \in \{O, AD\}$ indicates ours and the traditional AD rule respectively. These rules contain a minimum set of operations from which any program from the set $e_d$ can be composed. For example, $g - h = g + (-1) \cdot h$ and $g/h = g \cdot (h)^{-1}$. Boolean operators can be rewritten into compositions of step functions based on De Morgan's law. Forward mode AD can be carried out by applying Table 2 directly. Reverse-mode AD derivative rules can be obtained by replacing in Table 2 $\theta$ with the input argument $g$ or $h$, and treating the other input argument as constant for addition and multiplication. When combined with reverse-mode differentiation on $n$ parameters, both ours and traditional AD have O(1) complexity. But AD can be faster due to its simpler rules. This is in contrast with finite difference, which has O(n) complexity and is inefficient for programs with many parameters.

*Heaviside step.* Our rule is motivated by Equation 1 with $\phi$ being a 1D box kernel. But instead of computing $\frac{\partial g}{\partial x}$ analytically, we approximate it using finite difference to avoid extra back-propagation passes. Because $g^+$, $g^-$ are computed as an intermediate value as part of the computation of $H(g^+)$, $H(g^-)$, no extra computational passes are needed for the finite difference.

*Multiplication.* Our derivative rules work correctly when there is at most one discontinuity in the local region. However, when authoring programs, intermediate values that depend on the same discontinuity may further interact with each other, leading to multiplications where both arguments are discontinuous. For example, in shader programs, the lighting model to a 3D geometry may depend on discontinuous vectors such as surface normal, point light direction, reflection direction, or half-way vectors. These vectors can be discontinuous at the intersection of different surfaces, at the edge where a foreground object occludes a background object. When computing the intensity, these vectors are usually normalized first, therefore each of the discontinuous elements $n$ needs to be squared and be expressed as $n \cdot n$. For simplicity, we assume $n$ is a Heaviside step function and motivate our rule by showing differentiating $f = H(x+\theta) \cdot H(x+\theta)$ using the AD rule is already incorrect.

Because $f = H(x+\theta) \cdot H(x+\theta)$ can be simplified into $H(x+\theta)$, its pre-filtered gradient is already discussed in Section 3. Assuming a discontinuity sampled within the kernel, we can plug in $c(x,\theta) = x + \theta$ and $\phi(x - x_d) = \frac{1}{2\epsilon}$ into Equation 1 and get the following:

$$\frac{\partial \hat{f}}{\partial \theta}(x, \theta; \epsilon) = \frac{1}{2\epsilon}$$

Directly differentiating with the AD rule leads to zero because AD cannot correctly handle step functions. Even if we replace the Heaviside step gradient rule with ours and use the AD multiplication rule on $f = H \cdot H$, we still get the following incorrect result:

$$\frac{\partial_{AD} f}{\partial \theta} = H(x+\theta)\frac{1}{|2\epsilon|} + H(x+\theta)\frac{1}{|2\epsilon|} = \frac{H(x+\theta)}{\epsilon} \neq \frac{\partial \hat{f}}{\partial \theta}$$

Because $H(x+\theta)$ only samples exactly at the discontinuity with measure zero, in practice the above expressions will always evaluate to either $\frac{1}{\epsilon}$ or 0, both leaving $\frac{\partial_{AD} f}{\partial \theta}$ incorrect. Intuitively, AD fails because it treats the function as continuous, and therefore is always biased to either side of the branch. Because TEG's [2021] multiplication rule is equivalent to that of AD, this also leads to the degeneracy discussed in their Section 4.6: differentiating the multiplication of two identical step functions involves multiplication of a step function with a Dirac Delta, both being singular at the same position. Thus integrals involving such multiplication are undefined. Unlike TEG and AD, our multiplication rule samples on both sides of the branch, and therefore robustly handles this case.

$$\frac{\partial_O f}{\partial \theta} = \frac{H^+ + H^-}{2}\frac{1}{|2\epsilon|} + \frac{H^+ + H^-}{2}\frac{1}{|2\epsilon|} = \frac{1}{2\epsilon} = \frac{\partial \hat{f}}{\partial \theta}$$

*Function composition.* Similarly to multiplication, Appendix B describes how the AD rule also fails for a similar example $f = H(x+\theta)^2$ when viewed as a square function. Note our approximation applies different rules based on whether $h(g)$ is statically differentiable. Static differentiability of $h(g)$ means either $h(g)$ is statically continuous or $h(g)$ is not statically dependent on $x$. For static continuity, the compiler applies static analysis to the program, and decides static continuity based on whether each node depends on any discontinuous operators in its compute graph.

## 5 FIRST-ORDER CORRECTNESS

In this section, we formally define the notion of *first-order correctness* of a gradient approximation. Then we characterize the subset of programs for which ours and AD is first-order correct. This section targets readers who are interested in the mathematical underpinnings of our framework, but those who are more interested in its application can safely skip to the next section.

### 5.1 First-order Correctness Definition

We define *absolutely* and *relatively* first-order correct gradient approximations, where absolutely is used for local regions where $f$ is continuous and relatively is used otherwise. Intuitively, these say that the partial derivative approximation matches prefiltered derivatives from $\hat{f}$ up to error $O(\epsilon)$.

Definition 1. *A gradient approximation $\frac{\partial_k f}{\partial \theta_i}$ is absolutely first-order correct for parameter $\theta_i$ at $(x, \vec{\boldsymbol{\theta}})$ with kernel size $\epsilon$ if*

$$\frac{\partial_k f}{\partial \theta_i}(x, \vec{\boldsymbol{\theta}}; \epsilon) = \frac{\partial \hat{f}}{\partial \theta_i}(x, \vec{\boldsymbol{\theta}}; \epsilon) + O(\epsilon) \quad (2)$$

Definition 2. *A gradient approximation $\frac{\partial_k f}{\partial \theta_i}$ is relatively first-order correct for parameter $\theta_i$ at $(x, \vec{\boldsymbol{\theta}})$ with kernel size $\epsilon$ if*

$$\frac{\partial_k f}{\partial \theta_i}(x, \vec{\boldsymbol{\theta}}; \epsilon) \;/\; \frac{\partial \hat{f}}{\partial \theta_i}(x, \vec{\boldsymbol{\theta}}; \epsilon) = 1 + O(\epsilon) \quad (3)$$

### 5.2 Subsets of Our Minimal DSL

In general, we do not guarantee first-order correct gradient approximation for every program in $e_d$, although we do show empirically in Sections 8 that ours typically has low error and works in practice for optimizing shader parameters. To show first-order correctness, we progressively define $e_d$ from smaller subsets whose correctness can be shown.

$$\begin{aligned}
e_a &::= C \mid x \mid \theta \mid e_a + e_a \mid e_a \cdot e_a \mid f(e_a)\\
e_b &::= H(e_a) \mid H(e_b) \mid C \cdot e_b \mid e_b + e_b \mid e_b \cdot e_b \mid f(e_b)\\
e_c &::= e_a \mid e_b \mid e_c + e_c \mid e_c \cdot e_c \mid f(e_c)\\
e_d &::= e_c \mid H(e_d) \mid e_d + e_d \mid e_d \cdot e_d \mid f(e_d)
\end{aligned}$$

$e_a$ represents all continuous programs that can be expressed in our DSL. Both ours and AD is correct for this set. For example, a color palette that smoothly changes color according to time and pixel coordinate belongs to this set (Figure 3 (a)).

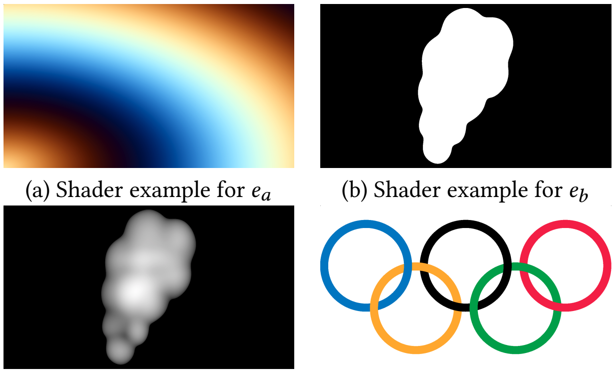
$e_b$ represents a subset of piece-wise constant discontinuous programs, whose discontinuities are either represented by continuous functions, or another $e_b$ function. Our gradient is correct almost everywhere for $\bar{e}_b$: $e_b$ excluding some pathological cases described in the Appendix Definition 13, 14 and 15. For example, a black and white blob whose shape changes parametrically belongs to $e_b$ (Figure 3 (b)).

$e_c$ represents the subset of programs whose discontinuities share a similar constrains as $e_b$, but with arbitrary continuous parts expressible by $e_a$. Our gradient is also correct almost everywhere for $\bar{e}_c$: $e_c$ excluding pathological cases. For example, a blob whose color is rendered according to pixels' distance to object boundary belongs to $e_c$ (Figure 3 (c)).

$e_d$ is the entire set of programs expressible in our minimal DSL. Generally, we do not give any correctness guarantee for this set. However, we can show empirically that gradient approximations in this set have low error for our quantitative metric in Section 7.3.

## 5.3 First-order Correctness Results

We begin by defining sets where $f$ is continuous and discontinuous with respect to different parameters, then we "dilate" the discontinuous sets for reasons related to measure theory that we discuss, and finally we show our theorem on the program sets $e_a, \bar{e}_b, \bar{e}_c$, where $\bar{e}_b, \bar{e}_c$ excludes pathological cases from $e_b, e_c$ respectively.



(a) Shader example for $e_a$

(b) Shader example for $e_b$

(c) Shader example for $e_c$

(d) Shader example for $e_d$

Fig. 3. Shader examples that belong to different subsets of programs.

Definition 3. *Given a function $f$, the continuous set $C_i$ is the set of all points $(x, \vec{\theta})$ in* dom($f$) *where $f$ is continuous with respect to $\theta_i$, or $f$ has a discontinuity with second order root (meaning there exists an intermediate value $H(g)$ where $g$ and $\partial g/\partial x$ evaluate to zero). The discontinuous set $D_i = \mathrm{dom}(f) \setminus C_i$.*

In Appendix Definition 15, we excluded discontinuities with roots of order 3 or above in the $\bar{e}_b, \bar{e}_c$ definition, so this leaves discontinuities with roots of order 2 or below in $\bar{e}_b$ and $\bar{e}_c$. Given $H(g)$, the second order roots for an intermediate value $g$ are of the form that they touch the x axis at a point without crossing it, so along x, the prefiltered derivative ignores this point and this point could be removed from the prefiltered gradient without changing it. Thus, we compare ours with a prefiltered gradient that we consider continuous (wrt x) at this point, so this is why we place the discontinuities with second order roots in $C_i$.

In most cases of practical interest, the discontinuous sets $D_i$ are of Lebesgue measure zero so the convenient Lebesgue measure gives uninteresting results on those sets. Thus we define "dilated" sets $D_i^r$ that expand regions around the discontinuities so discontinuities typically expand into nonzero measure regions as follows:

Definition 4. *The dilated 1D interval $N^r(x, \vec{\theta})$ is defined as $\{(x', \vec{\theta}) \in \mathrm{dom}(f)) : x' \in (x - \beta\epsilon', x + \alpha\epsilon') \ \ for\ \epsilon' = r\epsilon_f(x, \vec{\theta})\}$.*

Definition 5. *The dilated discontinuous set $D_i^r$ is defined as $D_i^r = \bigcup_{\forall (x,\vec{\theta}) \in D_i} N^r(x, \vec{\theta})$.*

Theorem 1. *For our approximation, $\exists \epsilon_f(x, \vec{\theta}) > 0$ such that for all parameters $\theta_i$, for every kernel size $\epsilon \in (0, \epsilon_f(x, \vec{\theta}))$, $f \in e_a \cup \bar{e}_b$ is absolutely first-order correct on $C_i$, and almost everywhere on $C_i$ for $\bar{e}_c$. For our approximation, $\forall r \in (0, 1]$, $\exists \epsilon_f(x, \vec{\theta}) > 0, \tau : D_i^r \to D_i$, such that for all parameters $\theta_i$, for kernel size $\epsilon_i^r = r\epsilon_f(\tau(x, \vec{\theta}))$, $f \in \bar{e}_b \cup \bar{e}_c$ is relatively first-order correct almost everywhere in $D_i^r$.*

Note that as $r$ varies in $(0, 1]$, $\epsilon_i^r$ is proportional to $r$, so the result on $D_i^r$ holds for a variety of kernel sizes. A proof sketch is presented in Appendix A. Our almost everywhere results on $D_i^r$ exclude measure zero sets of locations that have multiple discontinuities: we show this in Appendix A Lemma 2 and 4. An alternative interpretation is that the discontinuous point sets in $D_i$ in general can have Hausdorff dimension up to $n$ (and usually this dimension equals $n$), but the subset of points where our rule is not relatively first-order correct in $D_i$ have Hausdorff dimension strictly less than $n$, so not first order correct points form a lower dimensional subset within $D_i$ (see Appendix A Lemma 3 and 4).

## 6 COMPILER DETAILS

As mentioned in Section 4.2, our compiler supports functions with multidimensional outputs in $\mathbb{R}^k$ such as for $k = 3$ for shader programs that output RGB colors. Assuming we are optimizing a scalar loss $L$, we implement the gradient in a *single* reverse pass for efficiency by first computing the components $\partial L/\partial f^i$ of the Jacobian matrix for each output component $f^i$ of $f$, and the backwards pass simply accumulates (using addition) into $\partial L/\partial g$ for each intermediate node $g$. Our implementation assumes the program is evaluated over a regular grid, such as the pixel coordinate grid for shader programs. This allows small pre-filtering kernels that span between the current and neighboring samples that can still catch small discontinuities so long as they show up when sampling the grid.

Since our gradient approximation only works with a single discontinuity in the local region, our compiler averages between two smaller non-overlapping pre-filtering kernels to reduce the likelihood that the single discontinuity assumption is violated. Specifically, we average the gradient between $U[-\Delta x, 0]$ and $U[0, \Delta x]$, where $\Delta x$ is the sample spacing on the regular grid. This is similar to pre-filtering with $U[-\Delta x, \Delta x]$, but allows our compiler to correctly handle discontinuities whose frequency is below the Nyquist limit. For example, if a discontinuous function in a shader program results in a one pixel speckle on the rendered image, our compiler can still correctly account for discontinuities on both sides of the speckle. In the case of a single sampling axis, we would draw 3 samples along the sampling axis for each location where the gradient is approximated. Furthermore, the samples may be shared between neighboring locations on the regular grid.

Additionally, the compiler conservatively avoids incorrect approximation due to multiple discontinuities: when multiple discontinuities that are represented by different continuous functions are sampled, the compiler nullifies the contribution from that location by outputting zero gradient.

### 6.1 Combining Multiple Sampling Axes

It is common for multiple sampling axes to exist, either because the program is evaluated on a multi-dimensional grid (e.g. the 2D image for our shader applications), or because no single axis can be used to project every discontinuity. A natural question arises: how do we extend Section 4.2 to handle multiple sampling axes? A naive approach is arbitrarily choosing one of the axes, which risks ignoring some discontinuities that are *not* projected to the chosen one. This may happen, for example, when the discontinuity is parallel to the sampling axis (Figure 4(a)(b)). Another approach is to use a multi-dimensional prefiltering kernel (Figure 4(c)). However, integrating against a Dirac delta in $n$-dimensions with $n > 1$ typically results due to the sifting property in a $n-1$ dimensional integration over the set where the Dirac delta's argument is zero, which can be challenging but can be handled by additional sampling [Bangaru et al. 2021] or by recursively finding the intersection between discontinuity and the kernel support [Li et al. 2020].

In our implementation, for simplicity, we instead use a separate 1D kernel for each sampling axis and combine gradient approximations from different sampling axes afterwards. For each location, we adaptively choose approximations from available sampling axes based on the following intuition: the chosen axis should ideally be the one that is closest to perpendicular to the discontinuity (Figure 4(d)). This allows fixed-size small steps along the sampling axis to have a larger probability of sampling the discontinuity. In practice, for a discontinuity $H(c)$, we quantify this feature as $|\frac{\partial c}{\partial x}|$, and choose the axis with the largest value. Because this term corresponds to the denominator in Equation 1, a larger value leads to approximate gradients with smaller magnitude, therefore smaller variance. For $n$ sampling axes, our compiler draws $2n + 1$ samples, which can be potentially shared between neighboring locations.

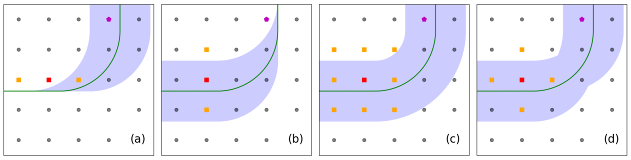


Fig. 4. Visualizing different options for how to combine multiple sampling axes in 2D. The green line demonstrates a discontinuity, and the blue region indicates evaluation locations where discontinuity can be sampled. Naively choosing either the $x$ (a) or the $y$ axis (b) can result in the discontinuity parallel to those axes being sampled at measure zero locations. For example, at the evaluation location indicated with a red square, each method places additional samples (orange squares) to sample discontinuities. Naively choosing the $x$ axis (a) fails because the discontinuity is parallel to the kernel direction. Although naively choosing $y$ axis (b) succeeds, it will fail if evaluated at the purple pentagon instead. Pre-filtering with a 2D kernel (c) allows robust sampling over the discontinuity, but the integration induces a computational burden. Our implementation (d) adaptively chooses from available axes, and ensures discontinuities in any orientation can be sampled with nonzero probability.

### 6.2 Efficient Ternary Select Operator

As can be seen in Section 4.2, in order to robustly handle discontinuities, our multiplication rule places two samples on both ends of the kernel support while AD just needs one. This leads to extra register usage in compiled GPU kernels that increases the likelihood of register spilling, which can result in slower run-times.

To alleviate the problem, our compiler always applies static analysis before multiplication, and switches to AD whenever both arguments are statically continuous. However, the ternary if or select operator can still be frequently expressed as multiplications of discontinuous values using the minimum DSL syntax introduced in Section 4.1. This is because the branching values themselves can be discontinuous, or the condition is a Boolean expression that needs to be expanded into multiplications of step functions using De Morgan's rule. Therefore, we introduce the ternary operator as an extended primitive to the DSL and design specialized optimizations so that differentiating it uses a similar number of registers to the AD rule, while allowing the first-order correctness property to stay the same as claimed in Section 5. See Appendix C for details.

## 7 ADAPTING OUR FRAMEWORK TO SHADERS

We apply our gradient approximations to procedural pixel shader programs on optimization tasks that involve finding good parameters so the shader matches a target image. Parametric discontinuities are common in shaders to control object edges and shape, visibility, and ordering.

Procedural shader programs are usually evaluated over a regular pixel grid, where the workload is embarrassingly parallel. Our compiler outputs a gradient program to two backends that both support highly parallel compute on the GPU. The TensorFlow (TF) and PyTorch backends utilize the pre-compiled libraries that allow for fast prototyping and debugging. The Halide backend on the other hand, grants full control over the kernel scheduling, and can be orders of magnitude faster than TF and PyTorch provided a good schedule. Section 7.1 discusses details about our abstraction to the Halide scheduling space, and an optional autoscheduler.

Unlike other rendering pipelines, the procedural shader representation allows programmers to abstract scenes using their own judgement, not limited by the primitives provided by the system's API. For example, although DVG [Li et al. 2020] provides circles as a basic primitive, in our program representation, a circle equation can be easily modified to represent a parabola, but with the absence of a parabola primitive in DVG, users may resort to manually defining the shape through control points. To fully utilize the ease of modification for program representations, we additionally provide a third backend that outputs the original program (without gradient) encoded with the optimal parameters to GLSL. Users can then interactively modify or animate the program through editors such as the one on `shadertoy.com`.

In the rest of this section, we first provide details regarding scheduling efficient gradient kernels for the Halide backend in Section 7.1. Next, we discuss a random noise technique that helps the convergence of optimization tasks for shader applications. Finally, to quantitatively evaluate the gradient approximation, we propose a metric based on the gradient theorem in Section 7.3.

### 7.1 Halide Scheduling

Because our compiler approximates the gradient of an arbitrary program in reverse-mode AD, the number of forward intermediate values can be arbitrary many. Unlike the work of Li et al.[2018b], which focuses on efficiently scheduling convolutional scattering and gathering operations, because procedural shader programs compute independently per pixel, our scheduling bottleneck is the limited register count per thread. On one hand, assuming unlimited register space, inlining the entire program into one kernel without intermediate checkpointing gives optimal performance as memory access is minimized. On the other hand, assuming unlimited memory, writing to and reading from memory for every intermediate computation is equivalent to building the graph using pre-compiled libraries such as TensorFlow. Because memory bandwidth is limited, this can be orders of magnitude slower than the first approach. Because register space is limited on GPUs, naively adopting the first approach usually results in register spilling, which can cause slowdowns. In principle, register spilling can be avoided by instructing part of the program to be recomputed *within* a GPU kernel. However, in practice we do not have this level of control even within Halide because of the common sub-expression elimination (CSE) optimization pass by CUDA. To work around this, our scheduling choice involves splitting the gradient program into multiple smaller kernels to best utilize available registers while minimizing the memory I/O.

There are two strategies to our scheduling space: each value computed in the forward pass (original program) can trade-off between recomputation (which can potentially lead to register spilling) or checkpointing (which can require extra memory I/O); and the backward pass as a large graph can be split into multiple sub-programs. The first strategy is analogous to the recomputation and memory consumption trade-off for back-propagation in neural networks [Chen et al. 2016; Gruslys et al. 2016]. However, generalizing those methods to arbitrary compute graph is hard: in sequential neural layers, checkpointing a node indicates perfect separation to the computation before and after the checkpoint; but the equivalence in our case would be a min-cut to the compute graph with variable terminal nodes, which is NP-hard. It is nontrivial to adopt classic graph cuts literature (e.g. [Boykov et al. 2001]) to this problem due to the interaction with the complex engineering of the lower-level register allocator and the hardware register space limits.

Table 3. Our compiler provides a simplified Halide scheduling space for a program $f$. For an explanation of each choice refer to Section 7.1.

| Name | Option |
|---|---|
| logged_trace | List of intermediate nodes in $f$ |
| cont_logged_trace | subset of logged_trace |
| separate_cont | {True, False} |
| separate_sample | {axis, kernel, None} |

Therefore, this section describes heuristic-based scheduling options summarized in Table 3. Note this is only a subset of the entire scheduling space, but it is easier to understand and explore, and large enough to contain reasonable scheduling for every shader program shown in this paper. For each intermediate values in the forward pass, we make a decision on checkpointing vs recomputation and encode the list of checkpoint nodes in logged_trace. For the space of splitting backward pass into sub-programs, we reduce it to a combination of the discrete choices in Table 3. Because the gradient wrt non-Dirac parameters can be computed with AD, it usually results a smaller gradient kernel, which can be optionally computed in a separate reverse-mode AD (separate_cont = True). Since this sub-program can be small, it may have extra register space for recomputation and save some memory I/O. Therefore, instead of reading checkpointing values from logged_trace, the gradient to non-Dirac parameters reads checkpoints from cont_logged_trace instead, which is a strict subset to logged_trace. The gradient wrt Dirac parameters, however, is a combination of approximating the gradient by pre-filtering four different kernels: a left and right kernel on image coordinates $x, y$ respectively (Section 6). Therefore, we can compute the gradient approximation to each sampling axis in a different sub-program (separate_sample = axis), or separate the approximation to each pre-filtering kernel into a different parts (separate_sample = kernel).

Even after simplification, the scheduling space is still a combinatoric space that is too large to exhaustively sample. Therefore we provide an optional autoscheduler based on heuristic search. To estimate register usage, we build an approximate linear cost model by counting the number of intermediate computation for each node in the forward pass. Based on the model, we iteratively add nodes to a potential checkpoint list based on greedy search: the newly added nodes should approximately halve the current maximum cost among all nodes. The cost model is updated according to the potential list before every iteration. The list is then combined with discrete options in Table 3 to search for a schedule with best profiled runtime. We first search the best discrete choice with a default list of checkpointing: logged_trace = cont_logged_trace = every output from ray marching loops if the shader involves the RaymarchingLoop primitive, and logged_trace = cont_logged_trace

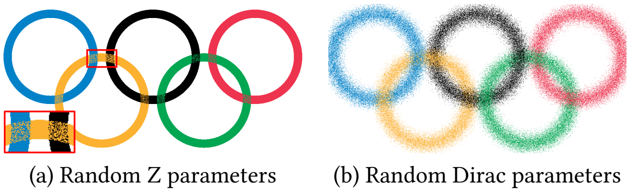

(a) Random Z parameters (b) Random Dirac parameters

Fig. 5. Augmenting parameters with uniformly distributed random variables. This helps to sample discontinuities more often. In (a) we show only augmenting random variables to parameters controlling Z order, which better samples Z ordering discontinuities, and in (b) we show our default choice of augmenting random variables to all Dirac parameters, which also causes object boundary discontinuities to be sampled more frequently.

= [] otherwise. With the best discrete combination, we further find the optimal logged_trace based on the potential checkpointing list, we search for the integer $n$ such that checkpointing every node found before iteration $n$ in the greedy search gives the best runtime. cont_logged_trace simply chooses from the optimal logged_trace or the default checkpointing list with best runtime.

## 7.2 Introducing Random Variables to the Optimization

As discussed before, our gradient approximation works when the discontinuity can be sampled along the sampling axes. The assumption that it suffices to use only a 2D spatial grid for sampling axes may be incorrect, when the discontinuity makes a discrete choice and the rendered image is only exposed to one branch. For example, in Figure 3(d), when rings overlap, the output color corresponds to the ring with the largest Z value. The choice is always consistent for each overlapping region. As a result, the discontinuity generated by comparing the Z value between two rings may *not* be sampled on the current image grid and the vanilla method is unable to optimize the Z values when the interlock pattern is wrong in Figure 3(d).

We propose to solve this problem by introducing auxiliary random variables. Conceptually we can think of these as extending the sampling space where we look for discontinuities from only the 2D spatial coordinates to a higher-dimensional space that includes 2D spatial parameters and Dirac parameters. For each Dirac parameter (with exception of those discussed in Section D.4), its value is augmented by adding a per pixel uniformly independently distributed random variable whose scale becomes another tunable parameter as well. For the ring example, adding random variables to the Z values leads to speckling color in the overlapping region, as shown in Figure 5(a). Each pair of pixel neighbors with disagreeing color represents the discontinuity on different choices to the Z value comparison. Because the compiler does not have semantic information for each parameter, in practice we augment every Dirac parameter with an associated random variable as in Figure 5(b). Instead of sampling discontinuities only at the ring contour, the random variable allows the discontinuity to be sampled at many more pixels.

Our gradient approximation also generalizes to the random variable setting. Instead of sampling along the image coordinate with regularly spaced samples, we now sample along a stochastic direction in the parameter space with sample spacing scaled by both spacing $\epsilon$ on image grid and the maximum scale $s$ among all random variables. Therefore, the width of the pre-filtering kernel is of the form $O(\epsilon) + O(s)$. Correspondingly, the error bound in Theorem 1 is changed from $O(\epsilon)$ to $O(\epsilon) + O(s)$: larger scale in the random variable increases our approximation error, but as the scale goes to 0, the error becomes similar to that without the random noise.

One caveat is that the random variable can not be combined with the RaymarchingLoop for implicitly defined geometry (Appendix D) because assumptions made in its gradient derivation can be violated by random variables. We explain this further in Appendix D.4. In the optimization process, a separate noise scale associated with *every* Dirac parameter (except those RaymarchingLoop primitives depend on) is tuned as well. At convergence, their values are usually optimized to be so small that the random noise is not be perceived during rasterization.

## 7.3 Quantitative Error Metric

Although we mathematically showed the approximation error for some subsets of programs is $O(\epsilon)$ (Section 5), we also wish to numerically evaluate the approximation made by our implementation. One possibility is to compute $L^1$ or $L^2$ distance between our gradient and finite difference [Li et al. 2018a]. However, a flaw with the $L^1$ or $L^2$ norm metrics is that the *same* delta distribution e.g. $\delta(x)$ can be formed as the limit of many different distributions e.g. $G_\epsilon(x)$ and $G_{2\epsilon}(x)$, where we use $G_\sigma(x)$ to identify the distribution for the Gaussian PDF wrt x with standard deviation $\sigma$, and those distributions have nonzero $L^1$ and $L^2$ distance between each other even as $\epsilon \longrightarrow 0$. This is undesirable because two different gradient rules that both give precisely correct derivatives as limits in the distribution theory but with different limiting “shapes" (e.g. $G_\epsilon(x)$ and $G_{2\epsilon}(x)$) would wrongly be considered to have nonzero error. Additionally, finite difference with finite step size and sample count introduces its own approximation error (Section 8). Another possibility is to analytically write down the reference gradient. However, this requires substantial human effort and may not be feasible depending on the complexity of the program.

We therefore avoid computing a reference gradient directly, and propose a quantitative metric according how much the gradient theorem is violated by the approximation. According to the gradient theorem, the line integral through a gradient field can be evaluated as the difference between the two endpoints of the curve. For example, assuming fixed pixel center, and with program parameters moving from $\vec{\theta}_0$ to $\vec{\theta}_1$ along a differentiable curve $\Theta$, Equation 4 should always hold for the reference gradient, and any good approximation should keep the difference between both sides of the equation as small as possible. We refer to the two sides of the equation as LHS (left hand side) and RHS (right hand side).

$$\int_\Theta \nabla f(\vec{\theta}) d\vec{\theta} = f(\vec{\theta}_1) - f(\vec{\theta}_0) \tag{4}$$

Additionally, because the gradient theorem requires a continuously differentiable function, we need to pre-filter the discontinuous program $f$ before applying Equation 4. In practice, the desired pre-filter is a n+2-dimensional box filter in both image × parameter space: $K^* = U[-1,1]^2 \times U[-\xi, \xi]^n$. In image space, the $U[-1,1]^2$ kernel smooths out a 3x3 region centered at the current pixel, and in the parameter space, $\xi$ is chosen such that the diagonal length of

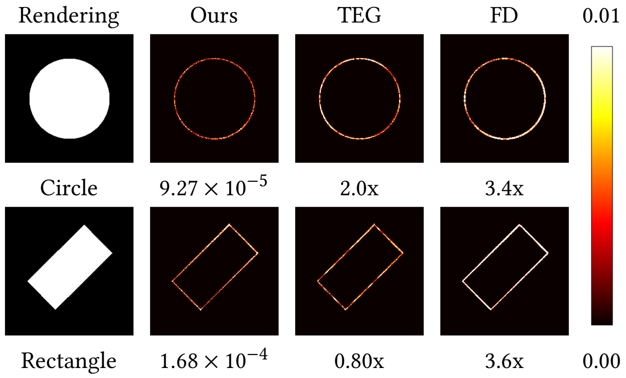


Fig. 6. We compare ours, TEG [Bangaru et al. 2021] and finite difference (FD) using a circle and a rectangle shader. The quantitative errors are reported below as the mean error from all pixels, with baselines relative to ours. FD is evaluated at 1 sample per pixel and is chosen as the lowest error from among five different step sizes ($10^{-i}$ for $i \in \{1, 2, 3, 4, 5\}$).

the n-dimensional box kernel is 1/10 that of the line integral. We keep the filter size relatively small to avoid extra sampling due to the curse of dimensionality. We use the L1 difference between LHS and RHS in Equation 5 as our error metric.

$$\int_{\Theta} \nabla(K * f(\vec{\theta}))d\vec{\theta} = K^* * f(\vec{\theta}_1) - K^* * f(\vec{\theta}_0) \tag{5}$$

The RHS is estimated by sampling the $n$+2-dimensional box filter $K^*$ around $\vec{\theta}_0$ and $\vec{\theta}_1$ and the LHS is estimated by quadrature using the midpoint rule. Because different methods make different prefiltering assumptions internally, the kernel $K$ in the LHS's integrand is adapted for each method so the combination of all prefilters results in the same desired target $n + 2$-dimensional box filter $K^*$ for each method: this is described in Appendix E.

We always evaluate the metric without random variables (Section 7.2) due to computational reasons: random variables introduce an additional pre-filtering integral along all dimensions of the parameter space. Due to the curse of dimensionality, accurately evaluating the LHS and RHS in the case of random variables thus requires a very large number of samples.

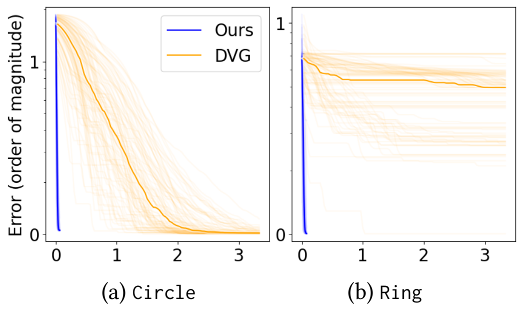
## 8 EVALUATION AND RESULTS

### 8.1 Simple Shader Comparison with Related Work

This section compares our method with TEG [Bangaru et al. 2021] and differentiable vector graphics [Li et al. 2020], using shader programs that are expressible in both their framework and ours.

*8.1.1 Comparison with TEG.* We compare with TEG [Bangaru et al. 2021] using two shaders: rectangle and circle. The discontinuities in the rectangle shader are affine, and can be automatically handled by TEG. For the circle shader, we need to manually apply a Cartesian to polar coordinate conversion to differentiate in TEG.

We evaluate ours, TEG and finite difference (FD) using our error metric (Section 7.3) and report in Figure 6. Because TEG can correctly differentiate both shaders, the low quantitative error is expected, and is solely caused by approximation that TEG makes using its quadrature rule for pre-filtering the 2D kernel.There are three possible sources of error for ours: sampling error for the 2D prefiltering; first order residual error in our gradient approximation ($O(\epsilon)$ term in Definition 2 and 1); and inaccurate approximation when our single discontinuity assumption is violated (e.g. at the four corners of the rectangle). Nevertheless, TEG's error is 2.0x and 0.80x compared to ours in the two examples, indicating that the approximations our method introduces cause minimal error compared to the sampling error of the prefiltering.



(a) Circle (b) Ring

Fig. 7. Comparing performance of ours and differentiable vector graphics [Li et al. 2020] for two optimization tasks. Each plot shows the convergence of 100 random restarts. The x axis reports wall clock time in seconds. The y axis reports log scale $L^2$ error relative to the minimum error $L_{min}$ found over all restarts for both methods. Labels on the $y$ axis are base 10 exponentials: a label $k$ indicates a $L^2$ error of $L_{min}10^k$. Each restart is reported as a transparent line, and the median error within all restarts (at a given time) is shown as the solid line. In the circle example, DVG converges with a slower runtime. In the ring example, DVG hardly converges because its gradient wrt the radius of the ring has a bug: see Appendix Figure 15 for details.

We use TEG's CPU implementation as running its gradient program on the GPU requires manually writing extra CUDA kernels. Therefore we do not compare with TEG using the optimization tasks similar to Section 8.2 because rasterization on the CPU is slow.

*8.1.2 Comparison with Differentiable Vector Graphics.* We compare with Differentiable vector graphics (DVG) [Li et al. 2020] using two shaders: circle and ring. Both of them are expressed as a circle primitive in DVG but the ring has a specified stroke width and blank fill color. Because DVG is integrated in PyTorch and does not provide an API for efficiently extracting per pixel gradient map for arbitrary parameters, our comparison is focused on comparing performance in a gradient-based optimization task.

The reference image for both shaders are rendered with a slightly eccentric, antialiased ellipse such that neither shader can reproduce the target with perfect pixel accuracy. We use L2 image loss and optimize for 100 iterations for each of 100 randomly sampled initializations. Because our method reuses samples from neighboring pixels to sample the discontinuity, the actual number of samples computed is 1 per pixel. However, we find using 1 sample per pixel for DVG generates inaccurate gradients even for continuous parameters such as color. Therefore, we use $2 \times 2$ samples instead.

For the circle shader (Figure 7(a)), both ours and DVG easily converge to images that are similar to the reference image (Appendix Figure 14), but DVG is much slower. For the ring shader, however, DVG fails to converge for most of the restarts (Figure 7(b)). We suspect the non-convergence is caused by a bug in their code that generates incorrect gradients (Appendix Figure 15). However, even disregarding the bug, DVG is slower than our method and can not handle parametric Z ordering as in Figure 10.

## 8.2 Optimizing to Match Illustrations in the Wild

In this section, we demonstrate that our differentiation method robustly applies to a variety of shaders that can be used to match illustrations directly found on the Internet. These shader programs represent similar complexity as those found on `shadertoy.com`, and are usually designed using a programmer's abstraction of the scene. As a result, animating and modifying the program representation is easier because program components as well as parameters have semantic meaning. In general, programs can be written with arbitrary branching compositions. This is in contrary to specialized rendering pipelines, such as using splines to represent 2D scenes and triangle meshes for 3D. Their expressiveness comes from the massive number of parameters rather than the structure of the program. Therefore specialized rules can be developed to differentiate vector graphics or path tracers, as different parameter values still lead to similar discontinuity patterns. However, it is hard to manually animate or control appearance using the parameters in such pipelines, because there are hundreds or thousands of parameters and they typically do not have attached semantic meaning.

In this section, we optimize shader parameters to match the rendering output to target images. All target images presented in this section are directly downloaded from the Web, with the only modifications being resizing and converting RGBA to RGB. We use a multi-scale L2 loss. To avoid local minima, the loss objective alternates from the lowest resolution L2 to the sum of every L2 over a pyramid up to resolution $N$ until $N$ reaches the rendering resolution. This loss alternation is repeated 5 times within 2000 iterations. To further aid convergence, we add a uniformly distributed random variable (Section 7.2) to every Dirac parameter that is *not* dependent on ray marching. The scales of the random variables is also optimized: upon convergence, their values are usually close to zero.

For each optimization task, we restart from 100 random initialization with 2000 iterations per restart. To analyze convergence properties, we say that a restart "succeeds" if it converges to an error lower than twice the minimum error found in all restarts by the default method: ours with random variables. We additionally report an ablation without the random variables. The success threshold is plotted as the grey horizontal line in Figure 8. Based on this definition of success, we compute two metrics: median success time and expected time to success, and report these in Table 4. The median success time is the median time taken for a restart to reach the success threshold across all restarts that succeed. These values are plotted as colored circles on the grey line in Figure 8. Expected time to success, on the other hand, evaluates given sufficient restarts, the mean time until the optimization finally converges. It is computed by repeatedly sampling with replacement from the 100 restarts and accumulating their runtime until a sampled restart converges.

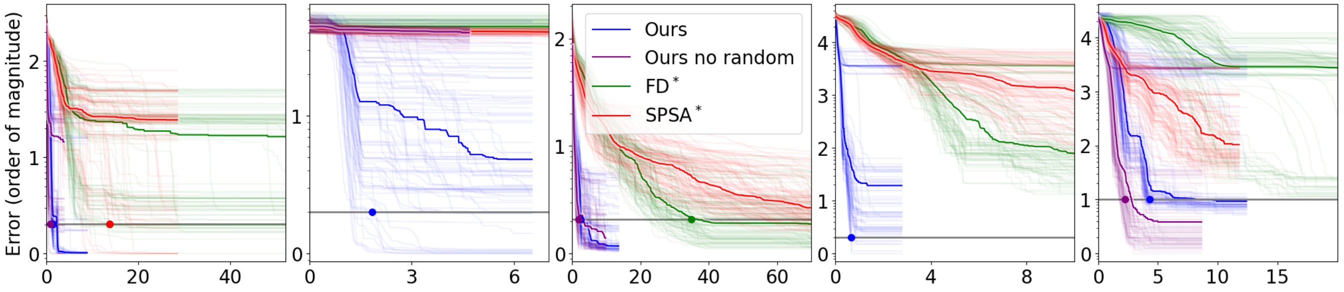
As we discussed before, because of the arbitrary composition patterns present in the shader programs, they cannot be differentiated using other state-of-the-art differentiable renderers. Therefore in this section, we compare our method with finite difference and its stochastic variant SPSA [Spall 1992]. To best benefit the baselines, we run the optimization task with ten variants for finite difference: with or without random variables combined with different step sizes ($10^{-i}$ for $i = 1, 2, 3, 4, 5$) and denote the one with minimum error across all restarts as FD$^*$. Similarly, our SPSA$^*$ baseline chooses from thirty SPSA variants by least error. The 30 variants includes a combination of 5 different step sizes similar to FD$^*$, two choices on the number of samples per iteration (1 vs half of the number of parameters), and three choices for the optimization process (with or without random variables as in FD$^*$, or a vanilla variant that removes loss objective alternation and randomness). Note because low-sample-count SPSA runs faster, we scale up its number of iterations accordingly so that it runs at least as long as our method. We also experimented with AD and zeroth order optimization (Nelder-Mead and Powell). AD hardly optimizes the parameters because most of our shaders have little to no continuous cues for optimization. Zeroth order methods have problems searching in high dimensions, and never succeed according to our criterion under the same time budget. We therefore did not report these results.

### *8.2.1 Shader:* `OlympicRings`.

We design a shader to express the Olympic logo shown in Figure 10(a)-top. The shader uses more rings than are necessary (10) to avoid being stuck in a local minimum. Each ring is parameterized by its location, inner and outer radius, and color. Because the rings are interlocking, each ring is slightly tilted vertically such that the Z value parametrically depends on the pixel's relative distance to the ring center. Unlike DVG which requires a single Z value per shape, as is typical for illustrative workflows, our method allows the users to define a parametric Z ordering and optimizes it automatically. Additionally, because of the interlocking pattern, the shader can not be easily expressed using the circle primitive in DVG, as each primitive has a user-defined constant Z order.

We run the optimization task with 100 restarts and report our result with the minimum error as Figure 10 Optimization, which is almost pixel-wise identical to the target. We also report the convergence across all restarts for Ours, FD$^*$, and SPSA$^*$ in Figure 8(a). For the majority of restarts, both ours and FD$^*$ converge to a low error, but FD$^*$ requires an order of magnitude longer runtime. Additional convergence metrics are reported in Table 4. For both metrics, ours is faster than the baselines by an order of magnitude. We also optimize the same target using a simpler shader with 5 rings. The median success time and expected time to success for ours 5 rings are 0.5x and 1.6x for that of ours 10 rings, respectively. This is because the simpler shader has faster runtime, but converges less often. We additionally quantitatively evaluate the gradient approximations using our error metric (Section 7.3) and show the results in Figure 9. Ours outperforms both baselines by a large margin.

After the optimization, the compiler outputs the shader program with optimized parameters to a GLSL backend, which allows interactive editing on platforms such as `shadertoy.com`. Because the shader program renders extra primitives to avoid local minimum, we apply an EliminateIfUnused() program annotation outside the computation for each ring so the compiler can automatically use



(a) Olympic Rings (b) Celtic Knot (c) SIGGRAPH (d) TF RayMarch (e) TF RayCast

Fig. 8. Runtime-Error plot for five optimization tasks, comparing Ours (with and without random variables) with FD* and SPSA* – using 100 restarts with random initialization and plot axes as in Figure 7. Note we select FD* and SPSA* based on minimal error from among 10 and 30 different variants, as discussed in Section 8.2. Grey horizontal lines denote success threshold, while circles on grey lines mark median success time as in Table 4.

a technique akin to dead code elimination to remove unused rings from the output GLSL code. During optimization, both forward and backward programs emit the computations within EliminateIfUnused(). Upon convergence, the compiler progressively removes any computation in EliminateIfUnused() if it does not increase the optimization error. In the Olympic rings example, this refers to extra rings that are either pushed outside of the image or to the back of the image with the same color as the background. The pruned compute graph is then output to the GLSL backend, and includes only code and parameters for the five rings visible in the rendering. Figure 10 Modified shows an example interactive edit for the GLSL program: we decrease the spacing between rings and thicken them. The interactive edit simply requires modifying corresponding parameter values in the program representation. However, if the target image is represented by multiple filled shapes, editing it to the modified position requires many tedious manual changes such as editing the control points for each shape, and adding new shapes. Adding new shapes may be necessary because typically editors only support a single Z value per shape, and the decreased ring spacing introduces more disconnected regions, such as the small black region inside the blue ring.

### 8.2.2 *Shader:* Celtic Knot.

We modify the ring shader from Section 8.2.1 to match the target image for Celtic Knot in Figure 10(a). The Z ordering of the rings are parameterized similarly to correctly reconstruct the interlocking pattern, but instead of rendering colored rings, the shader renders black at the edge of the ring with a parametric stroke width, and white elsewhere.

The black and white target image [Alexander Panasovsky 2018] brings an extra challenge to the optimization task, because the shader can no longer rely on color hue to match the target, but instead use gradients only from discontinuities. This limits the number of pixels that contribute to the gradient, as discontinuities are only sampled at a sparse set of pixels. Additionally, when the rendering and the target image are poorly aligned, the majority of the gradient contribution is quite noisy, which causes the optimization landscape to be almost flat except for a small neighborhood around the minimum. The problem is alleviated by the random variables discussed in Section 7.2. By randomly perturbing the parameter values, we generate a fuzzy rendering that greatly increases the number of pixels with differently branched neighbors, which permits our method to sample discontinuities more frequently.

Our optimization result is reported in Figure 10. It correctly locates the rings, and correctly models the interlocking pattern. For the convergence plot in Figure 8 (b), ours converges at a lower rate than (a) because of the optimization challenges discussed, but significantly outperforms ours no random, FD* and SPSA*, which do not converge at all. This is also reflected in Table 4. To confirm that the scales of random variable always converge to zero, and that the lower convergence rate is due to the flat optimization landscape, we run an additional optimization task for Ours where the parameters are initialized at their optimal position with the same random variable initialization as in Figure 8 (b). In all 100 restarts, the scale to the random variable always converges close to 0 such that their effects do not influence the rasterization, and the parameters stay optimal to within 1% of the minimum error. Because SPSA* is stochastic and does not always follow the gradient direction, its poor performance on an almost flat landscape is expected. FD* on the other hand, is unable to find a suitable step size: a small step can fail to sample discontinuities especially in the presence of the random variables, but a large step approximates the gradient inaccurately and therefore, similarly to SPSA*, works poorly on the almost flat landscape with or without random variables. Figure 9 quantitatively evaluates the gradient approximation between ours and baselines

Table 4. Time metrics comparing how fast ours, ours without random variables (O/wo) and baselines converge, as discussed in Section 8.2. Symbol × indicates the method never succeeded in all restarts.

| Shader | Med. Success Time | | | | Exp. Time to Success | | | |
|---|---|---|---|---|---|---|---|---|
| | Ours | O/wo | FD* | SPSA* | Ours | O/wo | FD* | SPSA* |
| Olympic Rings | 1.4 | **0.9** | 13.8 | 13.8 | **5.0** | 19.1 | 342.4 | 252.7 |
| Celtic Knot | **2.3** | × | × | × | **17.3** | × | × | × |
| SIGGRAPH | 2.6 | **1.8** | 34.9 | 71.4 | **6.1** | 6.2 | 86.5 | 247.8 |
| TF RayMarch | **0.7** | **0.7** | × | × | **40.3** | **40.3** | × | × |
| TF RayCast | 4.3 | **2.2** | 31.4 | × | 13.9 | **9.0** | 2187.8 | × |

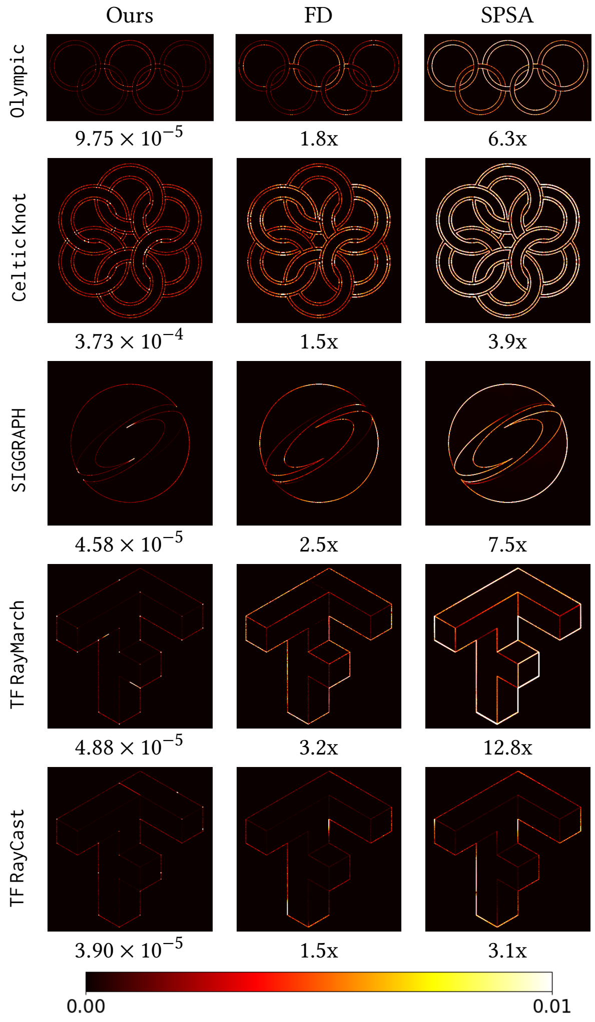


Fig. 9. We quantitatively evaluate the gradient approximation between ours and baselines using the error metric described in Section 7.3. At each row, we evaluate the metric over a scalar function that sums up the pixel values in all three color channels for the corresponding shader program in Figure 1, 10 and 11 using the optimized parameters. Below each method we report the mean error from all pixels. For baselines we report relative error compared to ours. The finite difference (FD) baseline is always evaluated at 1 sample per pixel because FD is always slower than ours in the optimization. For SPSA, the number of samples is chosen so that its runtime per iteration in the optimization task is comparable to ours. Similar to Figure 6, FD and SPSA are chosen as the lowest error from among five different step sizes ($10^{-i}$ for $i \in \{1, 2, 3, 4, 5\}$).

at the optimized parameters. Even when random variable is not present, FD* still has a higher error than ours.

Because the compiler generated code is less readable than manually written programs, we add an Animate() construct to the DSL to facilitate easier interactive edits and animations of the output GLSL shader. The Animate() construct indicates which input variables a programmer who is animating or editing might wish to read and output variables that might be modified, and inserts a empty Animate() function within the GLSL code with input and output variables correctly referenced. That function's body can then be easily edited interactively essentially by performing variable substitutions or used to produce animations. Figure 10 shows an example of coloring the optimized Celtic Knot shader. With our framework, we simply use the Animate() method in GLSL to access the pixel's relative position within each ring, and modify the color values accordingly. The interlocking is automatically handled by the optimized Z ordering. Such an edit can be more cumbersome if directly editing the target image in Photoshop or Illustrator, because users need to manually mask out disconnected regions caused by the overlapping.

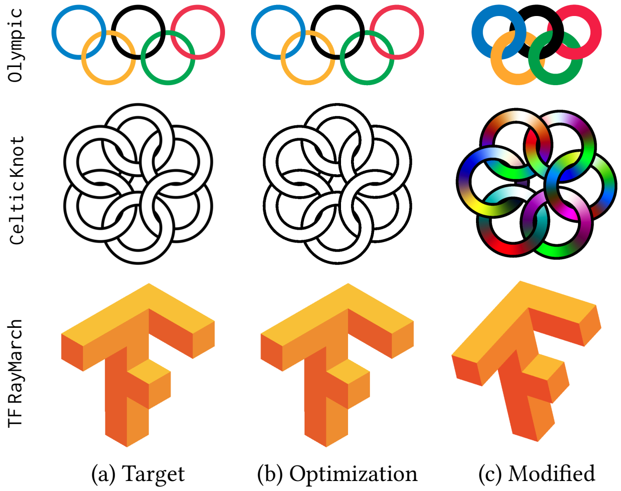


Fig. 10. Target, optimization, and modified results for three shaders discussed in Section 8.2: `Olympic Rings`, `Celtic Knot` and `TF RayMarch`.

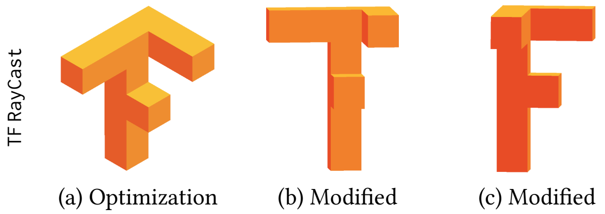


Fig. 11. Optimization and modifications for the shader `TF RayCast`, using target from Figure 10(a). Section 8.2.5 discusses the differences between `TF RayCast` and `TF RayMarch`.

### 8.2.3 *Shader:* `SIGGRAPH`.

In this section, we explore a 3D shader that can be used to reconstruct the SIGGRAPH logo as in Figure 1(a). Our shader is adapted from the shader "SIGGRAPH logo" by Inigo Quilez on `shadertoy.com`. The original shadertoy program is hand-designed with manually picked parameters to best match the target image. However, the rendered output (Figure 1(b)) is still very different from the target. We modified the original shader so that the geometry and lighting model are closer to the target image. Each half of the geometry is represented by the intersection between a sphere and a half-space, from which is subtracted an ellipsoidal cone shape whose apex is at the camera location. The ray intersections are approximated using sphere tracing with 64 iterations. Each half is further parameterized with different ambient colors, and lit separately by a parametric directional light and another point light.

Directly differentiating through the raymarching loop can result in a long gradient tape because the number of loop iterations can be arbitrarily large. As an alternative, we bypass the root-finding process and directly approximate the gradient using the implicit function theorem. We extend the DSL with a RaymarchingLoop primitive, and develop a specialized gradient rule motivated by [Yariv et al. 2020](Appendix D). One caveat is that the specialized gradient can not be combined with random variables, as assumptions made for the gradient derivation can be violated by the randomness. Therefore, we do not associate random variables to *any* parameters that the RaymarchingLoop primitives depend on.

Our optimization is almost identical to the target image and is reported in Figure 1(c). FD* can also achieve a similar low error, but because its runtime scales by the number of parameters, it converges slower than ours by an order of magnitude, as reported in Table 4 and Figure 8(c). The quantitative error metric for the gradient approximation is reported in Figure 9. Note because this shader renders the 3D geometry using sphere tracing, it is differentiated using rules from Appendix D whose accuracy depends on that of the iterative sphere tracer as well. Nevertheless, ours still has lower error by a large margin compared to the baselines.

Because parameters and program components have semantic meaning, this opens many more editing possibilities than we have for the original image. For example, we can similarly insert an Animate() primitive as in Section 8.2.2 and add Perlin noise [Perlin 2002] bump mapping to the geometry (Figure 1(e)). As an alternative, we can also utilize the displacement mapping and lighting model authored by Inigo Quilez in the original SIGGRAPH shader (Figure 1(b), but use optimized parameters to make its geometry similar to the target image. To do this, we modify Inigo Quilez's shader in GLSL so that its geometry and camera model are compatible with our parameterization, and paste the optimized parameters to the new shader. The hybrid modification is shown in Figure 1(f).

*8.2.4 Shader:* TF RayMarch. Similar to Section 8.2.3, we author a 3D ray-marching shader that can be used to reconstruct the Tensorflow logo shown in Figure 10(a). The shader uses a 64 iteration sphere tracing loop to approximate the union of 4 boxes whose positions are constrained based on our observations of the target image, such as they should always be connected to each other by some particular surfaces. The scene is then shaded by a parametric ambient color and directional color lights.

Our optimization result is shown in Figure 10 along with a modified novel view rendering. The convergence and quantitative error comparisons are reported in Figure 8(d) and 9, respectively.

*8.2.5 Shader:* TF RayCast. In this section, we discuss an alternative program representation that directly uses ray-box intersection to attempt to match the same TensorFlow target image as in Section 8.2.4. All other components stay the same as the TF RayMarch variant.

Unlike the raymarching loop, which is differentiated using special rules discussed in Appendix D, ray casting shader programs are differentiated using the general gradient rules in Section 4.2, and so random variables can be applied to every Dirac parameter. As can be seen in Figure 8(e), ours both with or without random variables frequently converges, but for this shader, the no random variant benefits more from the faster runtime and the less noisy gradient approximation. FD* does not converge as well: it struggles to find a variant that is both accurate and able to sample discontinuities. But optimizing the TF logo is easier, because the geometry is colored, so FD* still converges for a few restarts. Due to its stochastic nature, SPSA* is less likely to be stuck at a local minimum, but it trades this for lower accuracy, and so is unable to achieve low enough error. In Figure 11 we show our optimization result (a), and two novel view renderings where the letters T (b) and F (c) are recognizable.

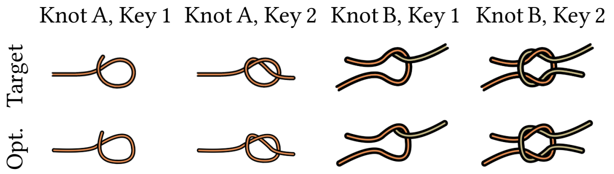


Fig. 12. Optimizations (below) matching animation key frames (above) for two knot tying examples. We manually pick target key frames from the original animation, as described in Section 8.3.1.

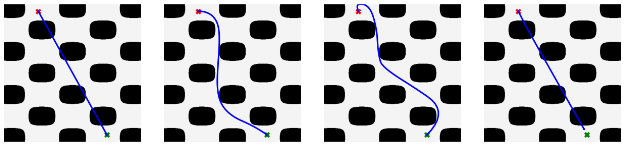
## 8.3 Beyond Optimizing a Single Image

This section demonstrates that the gradient of discontinuous programs can also be applied to other optimization tasks.

*8.3.1 Optimizing a Knot Tying Animation.* This section explores the possibility of using a program representation to reconstruct an animation sequence. Specifically, we experiment with knot tying animations shown in Figure 12. The animations for knot tying tutorials are generated by painstakingly manually specifying every single frame of the animation, where each frame is expressed by a combination of filled shapes or splines without semantic relation between each other. To extract the semantic meaning encoded in the animation, we design a rope shader whose position is modeled by a 2D quadratic Bezier spline. Because ropes can overlap themselves, their depth information is encoded as a quadratic Bezier spline as well.

In our implementation, we manually pick and optimize key frames in the animation sequence and increment the number of Bezier segments by 1 for each key frame. Figure 12 demonstrates our reconstruction of two key frames for each of the two different knots. Experiment details are described in Appendix G. Instead of drawing additional frames in between, we can easily render a smooth animation by interpolating control points for the rope. These animations are shown in our supplemental video.

*8.3.2 Optimizing a Path Planning Task.* In this section, we show our gradient approximation can also be applied to other domains such as path planning in robotics. The optimization task is to solve for a 2D trajectory that allows an object to travel from a start position to the target position without hitting any of the obstacles.



(a) Init (b) Ours (c) FD* (17x) (d) AD (0.9x)

Fig. 13. Path planning task: obstacles we wish to avoid are shaded as black regions, starting and desired ending position of the path are denoted as red and green crosses respectively. The path is initialized as a straight line between them (a). Both Ours (b) and FD* (c) can find a desirable path but AD (d) suffers from avoiding the discontinuous repulsion field. Similar to Section 8.2, FD* is chosen from the five variants with different step sizes. We report relative runtime to ours for FD* and AD. While AD has similar runtime, FD* is significantly slower.

We model the 2D trajectory by initializing the object with zero velocity, with 10 segments of piece-wise constant acceleration from time 0. Each constant piece of acceleration is parameterized by its x, y component and duration. The velocity and position can be represented as piece-wise linear and quadratic forms respectively. An L2 loss encourages the position at the end of the last segment to be close to the target position. We model the obstacles as a discontinuous repulsion field with a constant large value inside the obstacle and 0 everywhere else. A configuration is penalized by the integral of the field along the trajectory. The penalty term can be further reparameterized as an integral over time, and is approximated by quadrature using 10 samples per constant acceleration segment. We additionally add a third loss that minimizes the integral of the magnitude of the acceleration over time, which is proportional to the total fuel consumed assuming constant specific impulse.

Because our gradient approximations are implemented specific to shader programs, for this task we manually implement the gradient program in numpy using syntax and rules described in Section 4. We report the optimization result in Figure 13. While both ours and FD* finds a desirable path, FD* is significantly slower (17x).

## 9 LIMITATIONS, FUTURE WORK, AND CONCLUSION

Our method has a number of limitations, which offer potential avenues for future work. First, our application to shaders requires a programmer who is sufficiently skilled in writing shaders so the given shader for some parameter setting can approximate the target image. We imagined that future work might broaden the scope of applicability to non-programmers by setting up a 2.5D or 3D workspace where primitives can be placed down and properties can be assigned to them such as interior and edge color, visibility, fronto-parallel or planar depth in 2.5D, or geometric properties and relationships in 3D such as radius, abutment, symmetry, or CSG operations. Then the user could optimize user-chosen subsets of these properties to produce different designs.

Additionally, our current heuristic-based Halide auto-scheduler may not generalize to more complicated shader programs. For example, an iterative loop without specialized gradient approximation can generate a very long tape, which can cause trouble for efficient scheduling. Future research can either establish better cost models and checkpointing strategies for better auto-scheduling, or develop GPU kernels that can smartly trade-off between register usage and recomputation.

Further, our specialized gradient approximation for implicit geometry cannot be combined with random variables. This can result in undesirable local minima for optimization applications, such as when one object is entirely occluded by another. We believe a different specialized rule may be designed based on volumetric rendering, so that both foreground and background objects are involved in the differentiation.

Fourth, our gradient works under the assumption of a single discontinuity. For programs sampled on a regular grid, the number of samples that violate this assumption is inversely proportional to the grid's sampling frequency. Future work could explore adaptive sampling along the sampling axis to further increase the likelihood of a single discontinuity. Finally, although in our implementation the sampling axis is independent from the parameter space, we imagine our gradient rules could further be generalized to arbitrary choice of sampling axis, such as a linear combination of the parameters.

In conclusion, this paper proposes an efficient compiler-based approach to approximately differentiate arbitrary discontinuous programs in an extended reverse-mode AD. We validate our approximation both using theoretical error bounds and quantitative error metrics. We demonstrate a useful application to interactive editing and animating illustrations represented as shader programs.

## ACKNOWLEDGMENTS

Thanks to Boris Mityagin for elucidating details of his proof technique [Mityagin 2020] which we followed in the Appendix Lemma 3. TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc. The Olympic rings are the exclusive property of the International Olympic Committee (IOC).

## REFERENCES

Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (jul 2019), 12 pages. https://doi.org/10.1145/3306346.3322967

Alexander Panasovsky. 2018. Celtic. https://thenounproject.com/icon/celtic-1975448/.

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

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

Yuri Boykov, Olga Veksler, and Ramin Zabih. 2001. Fast approximate energy minimization via graph cuts. *IEEE Transactions on pattern analysis and machine intelligence* 23, 11 (2001), 1222–1239.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). arXiv:1604.06174 http://arxiv.org/abs/1604.06174

Pau Gargallo, Emmanuel Prados, and Peter Sturm. 2007. Minimizing the Reprojection Error in Surface Reconstruction from Images. In *2007 IEEE 11th International Conference on Computer Vision*. 1–8. https://doi.org/10.1109/ICCV.2007.4409003

Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-Efficient Backpropagation Through Time. *CoRR* abs/1606.03401 (2016). arXiv:1606.03401 http://arxiv.org/abs/1606.03401

John C Hart. 1996. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer* 12, 10 (1996), 527–545.

Inigo Quilez. 2021. Distance Functions. https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm.

Chiyu Max Jiang, Avneesh Sud, Ameesh Makadia, Jingwei Huang, Matthias Nießner, and Thomas A. Funkhouser. 2020. Local Implicit Grid Representations for 3D Scenes. *CoRR* abs/2003.08981 (2020). arXiv:2003.08981 https://arxiv.org/abs/2003.08981

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

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018b. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH)* 37, 4 (2018), 139:1–139:13.

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.

Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. 2019. Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning. *CoRR* abs/1904.01786 (2019). arXiv:1904.01786 http://arxiv.org/abs/1904.01786

Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. 2019. Reparameterizing discontinuous integrands for differentiable rendering. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 38, 6 (Dec. 2019).

Ricardo Martin-Brualla, Noha Radwan, Mehdi S. M. Sajjadi, Jonathan T. Barron, Alexey Dosovitskiy, and Daniel Duckworth. 2021. NeRF in the Wild: Neural Radiance Fields for Unconstrained Photo Collections. In *CVPR*.

Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In *ECCV*.

Boris Mityagin. 2015. The zero set of a real analytic function. *arXiv preprint arXiv:1512.07276* (2015).

Boris Samuilovich Mityagin. 2020. The zero set of a real analytic function. *Matematicheskie Zametki* 107, 3 (2020), 473–475.

William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. 2021. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 61, 16 pages. https://doi.org/10.1145/3458817.3476165

Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (jul 2016), 11 pages. https://doi.org/10.1145/2897824.2925952

Michael Niemeyer, Lars M. Mescheder, Michael Oechsle, and Andreas Geiger. 2019. Differentiable Volumetric Rendering: Learning Implicit 3D Representations without 3D Supervision. *CoRR* abs/1912.07372 (2019). arXiv:1912.07372 http://arxiv.org/abs/1912.07372

Merlin Nimier-David, Sébastien Speierer, Benoît Ruiz, and Wenzel Jakob. 2020. Radiative Backpropagation: An Adjoint Method for Lightning-Fast Differentiable Rendering. *ACM Trans. Graph.* 39, 4, Article 146 (jul 2020), 15 pages. https://doi.org/10.1145/3386569.3392406

Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. 2019. Mitsuba 2: A Retargetable Forward and Inverse Renderer. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 38, 6 (Dec. 2019). https://doi.org/10.1145/3355089.3356498

Jeong Joon Park, Peter Florence, Julian Straub, Richard A. Newcombe, and Steven Lovegrove. 2019. DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation. *CoRR* abs/1901.05103 (2019). arXiv:1901.05103 http://arxiv.org/abs/1901.05103

Ken Perlin. 2002. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques.* 681–682.

Ken Perlin and Eric M Hoffert. 1989. Hypertexture. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques.* 253–262.

Savvas Sioutas, Sander Stuijk, Luc Waeijen, Twan Basten, Henk Corporaal, and Lou Somers. 2019. Schedule Synthesis for Halide Pipelines through Reuse Analysis. *ACM Trans. Archit. Code Optim.* 16, 2, Article 10 (apr 2019), 22 pages. https://doi.org/10.1145/3310248

Vincent Sitzmann, Michael Zollhöfer, and Gordon Wetzstein. 2019. Scene Representation Networks: Continuous 3D-Structure-Aware Neural Scene Representations. *CoRR* abs/1906.01618 (2019). arXiv:1906.01618 http://arxiv.org/abs/1906.01618

J.C. Spall. 1992. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE Trans. Automat. Control* 37, 3 (1992), 332–341.

Justus Thies, Michael Zollhöfer, and Matthias Nießner. 2019. Deferred Neural Rendering: Image Synthesis using Neural Textures. *CoRR* abs/1904.12356 (2019). arXiv:1904.12356 http://arxiv.org/abs/1904.12356

Ethan Tseng, Felix Yu, Yuting Yang, Fahim Mannan, Karl St. Arnaud, Derek Nowrouzezahrai, Jean-Francois Lalonde, and Felix Heide. 2019. Hyperparameter Optimization in Black-box Image Processing using Differentiable Proxies. *ACM Transactions on Graphics (TOG)* 38, 4 (7 2019). https://doi.org/10.1145/3306346.3322996

Y. Yang and C. Barnes. 2018. Approximate Program Smoothing Using Mean-Variance Statistics, with Application to Procedural Shader Bandlimiting. *Comput. Graph. Forum* 37, 2 (2018), 443–454.

Lior Yariv, Yoni Kasten, Dror Moran, Meirav Galun, Matan Atzmon, Basri Ronen, and Yaron Lipman. 2020. Multiview Neural Surface Reconstruction by Disentangling Geometry and Appearance. *Advances in Neural Information Processing Systems* 33 (2020).

Tizian Zeltner, Sébastien Speierer, Iliyan Georgiev, and Wenzel Jakob. 2021. Monte Carlo Estimators for Differential Light Transport. *ACM Trans. Graph.* 40, 4, Article 78 (jul 2021), 16 pages. https://doi.org/10.1145/3450626.3459807

Cheng Zhang, Zhao Dong, Michael Doggett, and Shuang Zhao. 2021a. Antithetic Sampling for Monte Carlo Differentiable Rendering. *ACM Trans. Graph.* 40, 4 (2021), 77:1–77:12.

Cheng Zhang, Zihan Yu, and Shuang Zhao. 2021b. Path-Space Differentiable Rendering of Participating Media. *ACM Trans. Graph.* 40, 4 (2021), 76:1–76:15.

Yang Zhou, Lifan Wu, Ravi Ramamoorthi, and Ling-Qi Yan. 2021. Vectorization for Fast, Analytic, and Differentiable Visibility. *ACM Trans. Graph.* 40, 3, Article 27 (jul 2021), 21 pages. https://doi.org/10.1145/3452097

# A PROOF SKETCH FOR THEOREM 1

## A.1 Key Math Definitions and Lemmas

In this section, we summarize the most important definitions and lemmas for the program sets discussed in Section 4.1 and briefly justify our claims. We start by defining computational singularity, which refers to the function value being undefined for any intermediate node of $f$.

Definition 6. *A function $f \in e_d$ is computationally singular at $(x, \vec{\theta})$ if any of its intermediate values $g$ satisfies one or more of:*

$$\begin{cases} g = h^C & \text{where constant integer } C < 0 \text{ and } h(x, \vec{\theta}) = 0 \\ g = h^C & \text{where } C \text{ is a constant non-integer and } h(x, \vec{\theta}) \leq 0 \\ g = \log(h) & \text{where } h(x, \vec{\theta}) \leq 0 \end{cases}$$

We now state the local boundedness and continuity results for program set $e_a$. For simplicity but without loss of generality we always assume the function is evaluated on one given sampling axis $x$ and other tunable parameters $\vec{\theta}$. As a reminder, because the definition of the domain of $f$ made in Section 4.2 always excludes computational singularities, our discussion in the appendix also excludes computational singularities.

Lemma 1. *A function $f \in e_a$ is evaluated at $(x, \vec{\theta}) \in \mathrm{dom}(f) \Rightarrow \exists\, \epsilon > 0$ s.t. $f, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial \theta_i}$ are bounded and Lipschitz continuous in $[x - \epsilon, x + \epsilon]$.*

*Proof sketch*: because of the construction of our DSL set $e_a$ from real analytic functions, we can always find $\epsilon > 0$ so that $f \in e_a$ and its gradients are real analytic on the local region. Real analytic therefore leads to local boundedness by boundedness theorem and local Lipschitz continuity by mean value theorem ■.

To characterize when the assumptions of our gradient rules are met, we first define locally zero along the sampling axis, which may lead to pathological functions containing intermediate expressions $H(g)$ that evaluate as $H(0)$ within a local interval. We further define two outlier scenarios where our first-order correctness properties cannot be proved.

Definition 7. *A function $f$ is locally zero along the sampling axis (i.e. $x$) at $(x, \vec{\theta})$ if $\exists\, \epsilon > 0$ s.t. $f(x', \vec{\theta}) = 0\ \forall\, x' \in (x - \epsilon, x + \epsilon)$ whenever $(x', \vec{\theta}) \in \mathrm{dom}(f)$. Note also that to avoid excessively verbose notation, from here on, when there is such an $x'$ we always implicitly assume $(x', \vec{\theta}) \in \mathrm{dom}(f)$, i.e. we assume we are within the set of points where $f$ is defined.*

Definition 8. *A function $f \in e_c$ evaluated at $(x, \vec{\theta})$ is symmetric along the sampling axis $x$ if for some intermediate value $g$ of $f$, $g$ is not statically continuous (defined at the end of Section 4.2), and at $(x, \vec{\theta})$, $g$ is continuous, $\partial g/\partial x$ exists, $\partial g/\partial x$ is not locally zero wrt $x$, and there exists $\epsilon_k > 0$ s.t. for all $\epsilon \in (0, \epsilon_k]$, $g(x + \beta\epsilon, \theta) = g(x - \alpha\epsilon, \theta)$. This implies $\frac{\partial g}{\partial x} = 0$ at $(x, \vec{\theta})$.*

Definition 9. *A function $f \in e_c$ is multi-discontinuous at $(x, \vec{\theta}) \in \mathrm{dom}(f)$ if any two of its intermediate values are of the form $H(g_i), H(g_j)$ such that $g_i, g_j \in e_a$ evaluate to 0, and $\nabla g_i, \nabla g_j$ are linearly independent vectors at the given $(x, \vec{\theta})$, where $\nabla = [\partial/\partial x, \partial/\partial\theta_1, \ldots, \partial/\partial\theta_n]$.*

With these definitions, we can now characterize the set of points that we will show are absolutely first-order correct in Definition 10.

Definition 10. *For a function $f \in e_c$, $(x, \vec{\theta}) \in C_i$ is C-simple if at $(x, \vec{\theta})$, $f$ is continuous along the sampling axis and not symmetric along the sampling axis $x$.*

Similarly, we want to characterize the set of points that are relatively first-order correct. However, because relatively first-order correct is applied on the dilated discontinuous set $D_i^r$, we first need to define a remapping from dilated intervals $(x', \vec{\theta}) \in N_i^r(x, \vec{\theta})$ back to discontinuous locations $(x, \vec{\theta})$ before defining D-simple in Definition 12. Note the existence of the remapping is guaranteed because $D_i^r$ is defined in Definition 5 as a union of $N_i^r(x, \vec{\theta})$.

Definition 11. *$\forall (x', \vec{\theta}) \in D_i^r$, we define a remapping $\tau(x', \vec{\theta}) = (x, \vec{\theta})$ such that $x \in D_i$ and $x' \in (x - \beta\epsilon', x + \alpha\epsilon')$ for $\epsilon' = r\epsilon_f(x, \vec{\theta})$.*

Definition 12. *For a function $f \in e_c$, $(x', \vec{\theta}) \in D_i^r$ is D-simple if $\tau(x', \vec{\theta})$ is not at a multi-discontinuity.*

Lemma 2. *For a function $f \in \bar{e}_c$ and a parameter $\theta_i$, C-simple points are almost everywhere in $C_i$.*

*Proof sketch:* We justify this using a equivalent statement: the set of points $S$ that are discontinuous along the sampling axis or symmetric along the sampling axis is measure zero in $C_i$. If $f$ is discontinuous along the sampling axis at $(x, \vec{\theta})$, for some intermediate value $H(g)$ of $f$ with $g \in e_a \cup e_b$, $g(x, \vec{\theta}) = 0$. If $g \in e_b$, then by recursing on the definition of $e_b$ (recursing into nodes in the graph that are discontinuous wrt $x$), there is also some intermediate value of the form $H(h)$ such that $h \in e_a$, and $h(x, \vec{\theta}) = 0$. If every intermediate value of the form $H(h)$ with $h \in e_a$ had $h(x, \vec{\theta}) = 0$ and $h$ locally zero wrt $x$ at $(x, \vec{\theta})$, then $f$ would be continuous wrt $x$ at $(x, \vec{\theta})$, a contradiction. So there exists some intermediate value of form $H(h)$ with $h \in e_a$ where $h(x, \vec{\theta}) = 0$ and $h$ is not locally zero wrt $x$ at $(x, \vec{\theta})$, and due to the construction of $e_a$, we have $h$ is real analytic as a single-variate function wrt $x$ within an open interval containing $x$ intersected with $\mathrm{dom}(f)$, per Def. 7 of locally zero. Similarly, if $f$ is symmetric along the sampling axis at $(x, \vec{\theta})$, there exists some intermediate value $g \in e_c$ such that $g$ is continuous, $\partial g/\partial x = 0$ and $\partial g/\partial x$ is not locally zero. Now since $g$ is continuous and by Lemma 6 discontinuities are isolated along $x$, we can choose $\epsilon > 0$ s.t. within a local region along $x$ (i.e. an open interval $(x - \epsilon, x + \epsilon)$ intersected with $\mathrm{dom}(f)$, per Def. 7) all intermediate values of $g$ with the form $H(\cdot)$ are constant, so $g$ is a sum, product, and/or composition of functions that are real analytic as a single-variate function of $x$ within that 1D local region, so $g$ and $\partial g/\partial x$ are real analytic single-variate functions within that local region. We can consider at $(x, \vec{\theta})$, the representation of $\partial g/\partial x$ as a single-variate real analytic function on the 1D local region containing $x$ as one possible local real analytic representation of $\partial g/\partial x$: there are finitely many such local real analytic representations even across all $(x, \vec{\theta}) \in \mathrm{dom}(f)$ since each $H(\cdot)$ has only 2 discrete values $\{0, 1\}$.

Given $\vec{\theta}$, consider the set $P(\vec{\theta})$ of points $x$ s.t. $(x, \vec{\theta}) \in C_i$ and along the sampling axis are symmetric or discontinuous wrt $x$. Consider the set $P'(\vec{\theta})$ of all zeros of all 1D real analytic (wrt $x$) functions that we described in the last paragraph after excluding points $x$ where these real analytic functions are locally zero: within the program $f$ there are finitely many such real analytic functions. By the previous paragraph's reasoning, $P \subset P'$. A not identically zero 1D function that is real analytic on an interval has countable zeros, and so the analytic functions (within suitable local regions) used to construct $P'$ have countable zeros, so $\mu_1(P') = \mu_1(P) = 0$, where $\mu_1$ is the Lebesgue measure on $\mathbb{R}$ for the $x$ axis. Define $\mu_k$ to refer to the Lebesgue measure in $\mathbb{R}^k$. Given measure $\mu_{n+1}(S)$, we can replace it with the Lebesgue integral $\int 1_S d\mu = \mu_{n+1}(S)$, apply Fubini's theorem considering $\mu_{n+1}$ as a product measure, and find $\mu_{n+1}(S) = 0$ after setting the innermost integral (over $x$ dimension) as 0 due to $\mu_1(P) = 0$. ■

We would next like to show that D-simple points are almost everywhere in $D_i^r$. Intuitively, we can do this because we have defined multi-discontinuities to be the intersection of two manifolds that each are in general position so they each have dimension $n$, their intersection is dimension $n - 1$, and the dilation operation in $D_i^r$ increases dimension to $n$, which is still measure zero in $\mathbb{R}^{n+1}$. We next formalize this intuition in a general Euclidean setting, which we will then will apply to our D-simple points.

Lemma 3. *Assume $U$ is an open subset of $\mathbb{R}^m$, $G_1, G_2 : U \to \mathbb{R}$ are continuously differentiable on $U$. Define $Z = \{x \in U : G_1(x) = G_2(x) = 0\}$. Assume for each $x \in Z$, $\nabla G_1, \nabla G_2$ are linearly independent vectors at $x$, where we use $\nabla = [\partial/\partial x_1 \ldots \partial/\partial x_m]$. Define $N(Z) = \bigcup_{x \in Z} v(x)$. Let $v(x) = ((x_1 + a(x), x_1 + b(x)) \times \{x_2\} \times \{x_3\} \ldots \times \{x_m\}) \cap U$, $a : U \to \mathbb{R}, b : U \to \mathbb{R}$. Then $\mu_m(N(Z)) = 0$ where $\mu_m$ is the Lebesgue measure in $\mathbb{R}^m$.*

*Proof:* We proceed similarly to Claim 1 of Mityagin [2015; 2020]. If $x \in Z$, by linear independence, we have $||\nabla G_1(x)|| \neq 0$ and $||\nabla G_2(x)|| \neq 0$. Thus, $Z = \cup_k Z_k$ for:

$$Z_k = \{x \in Z : ||x|| \leq k, ||\nabla G_i(x)|| \geq 1/k \text{ for i=1,2}, d(x, U^C) \geq 1/k\}$$

The same as Mityagin we omit the last requirement if $U = \mathbb{R}^m$, i.e. $U^C = \emptyset$. Like Mityagin, we note $Z_k$ is compact: clearly $Z_k$ is bounded, and it can be shown to be closed by noting that for continuous $F$, $\lim_{x \to c} F(x) = F(c)$, the nonstrict inequalities are preserved at the limit point, and so every limit point of $Z_k$ is in $Z_k$, so by the Heine–Borel theorem $Z_k$ is compact.

If $x \in Z$, we can use the implicit function theorem to find locally an $m - 2$ dimensional subspace where $G_1(x) = G_2(x) = 0$ if we have full rank for the Jacobian. Specifically, consider coordinates $x_i$ and $x_j$. Then the Jacobian used in the implicit function theorem is:

$$J^{i,j} = \begin{bmatrix} \partial G_1/\partial x_i & \partial G_1/\partial x_j \\ \partial G_2/\partial x_i & \partial G_2/\partial x_j \end{bmatrix} \tag{6}$$

This has full rank when the two rows are linearly independent. If $x \in Z$, then by the linear independence of $\nabla G_1, \nabla G_2$, we can choose $i \neq j$ such that $J^{i,j}$ is full rank. This can be shown by forming by $\nabla G_1, \nabla G_2$ into a $2 \times n$ matrix of rank 2, observing that row and column rank are equal, so that matrix has two linearly independent columns, which can be taken as $J^{i,j}$. Thus, by the implicit function theorem, any $x \in Z$ has a neighborhood $Q(x)$ such that $x$ is described by coordinates in a $m-2$ dimensional manifold. If $p \in v(x)$ then there exists $t \in [0,1]$ such that $p = x + e_1(a(x) + t(b(x) - a(x)))$, where $e_1 = [1, 0, 0, \dots, 0]$, so we can parameterize $N(Q(x))$ by $m-1$ dimensions. So $\mu_m(N(Q(x))) = 0$.

For $Z_k$, consider the set of all open neighborhoods from the implicit function theorem, i.e. $\{Q(x) : x \in Z_k\}$ this is also an open cover of $Z_k$, which is compact, so choose a finite subcover: choose the cover $Q(x_1), \dots, Q(x_N)$ associated with points $x_1 \in Z_n, \dots, x_N \in Z_n$. So $\mu_m(N(Z_n)) = 0$. But $Z = \cup_k Z_k$, so by subadditivity of measures, $\mu_m(N(Z)) = 0$. ■

Lemma 4. *For a function $f \in e_c$ and a parameter $\theta_i$, D-simple points are almost everywhere in $D_i^r$.*

*Proof sketch:* Using the 1D dilated intervals from Definition 4, define for any set $S$, $N^r(S) = \cup_{s \in S} N^r(s)$. We now justify a statement equivalent to what we want: given the set of points $S_d = \{(x, \vec{\theta}) \in \mathrm{dom}(f)$ :f is multi-discontinuous at $(x, \vec{\theta})\}$, we wish to show $\mu_{n+1}(N^r(S_d)) = 0$. By using $(x', \vec{\theta})$ as the $n+1 = m$ dimensions of Lemma 3, we trivially get this result. Specifically, the endpoints of the 1D intervals unioned by $N^r$ become functions $a, b$ in Lemma 3, and $\mu_{n+1}(N^r(S_d))$ is less than or equal to sum of measures over a finite number of choices of intermediate values $H(g_i), H(g_j)$ with $g_i \in e_a, g_j \in e_a$ of the set $A$ of points where $g_i, g_j$ evaluate to zero, and $\nabla g_i, \nabla g_j$ are linearly independent, so we use Lemma 3 with $G_1 = g_i, G_2 = g_j$, and we choose $U$ as follows. We choose any measurable open set $U'$ containing $S_d$, and use $U = U' \setminus D_0$, where $D_0$ is the set of points in $\mathrm{dom}(f)$ s.t. $g_i = g_j = 0$ and $\nabla g_i, \nabla g_j$ are linearly dependent: this leaves only the desired above points $A$ in the zero set for Lemma 3. Here $U$ can be shown open by considering that if $p \in U$ then (a) $g_i \neq 0$ or $g_j \neq 0$ or (b) $g_i = g_j = 0$ and $\nabla g_i, \nabla g_j$ are linearly independent: in case (a) we can show the neighborhood around $p$ exists in $U$ directly, and in case (b) we can choose a Jacobian in Lemma 3 with nonzero determinant, which is continuous wrt $(x, \vec{\theta})$, so we can likewise show the neighborhood around $p$ exists. Each of the measures in the finite sum over choices of $g_i, g_j$ is zero, so $\mu_{n+1}(N^r(S_d)) = 0$. ■

Because we show correctness of our approximation by comparing with a reference pre-filtered gradient, our proof also involves computing the reference. We show if a function evaluation $f(x, \vec{\theta})$ is either C-simple or D-simple, it can be locally expanded into the following form to allow easy computation of the reference gradient.

Lemma 5. *Local expansion: a function $f_1 \in e_c$ is either C-simple or D-simple at $(x, \vec{\theta}) \Rightarrow \exists f_2 = a + b \cdot H(c)$ with $a, b, c \in e_a$ and $\exists \epsilon > 0$ s.t. within the set $S = [x - \alpha\epsilon, x + \beta\epsilon] \times \vec{\theta}$, there is at most one discontinuity of $f_1$, and the function value of $f_1, f_2$ are identical within $S$ except when at the discontinuity, and their pre-filtered gradients are identical within $S$. If $f_1$ is C-simple at $(x, \vec{\theta})$ then $b = 0$.*

*Proof sketch:* the existence of $\epsilon$ can be justified because discontinuities are isolated on the 1D $x$ axis (Lemma 6). The local expansion can be obtained by recursively evaluating step functions that are *not* discontinuous at $f(x, \vec{\theta})$ into 0 or 1 and merging step functions that are discontinuous into the same form. This is only possible without multi-discontinuity, and is guaranteed by the point being C-simple or D-simple. Additionally, computing the reference gradient in the local expansion form requires applying Equation 1. ■

To accurately characterize the set of programs our compilers can differentiate with first-order correctness, we further define $\bar{e}_b, \bar{e}_c, \bar{e}_d$ as $e_b, e_c, e_d$ excluding three exceptions: discontinuity degeneracy, part dependency on the sampling axis, and discontinuities with roots of order 3 or above in Definition 13 - 15. Except for pathological programs, these occur rarely in practice. For clarity and to emphasize their pathological nature, we presented these exceptional programs as being removed from the grammar. However, since these 3 properties can be analyzed in a pointwise manner, if desired, our main theorem also holds on such programs if $\mathrm{dom}(f)$ is chosen such that it does not contain any points with these properties.

Definition 13. *A program $f \in e_c$ has discontinuity degeneracy at $(x, \vec{\theta})$ if $f$ is $C^0$ continuous, but by static analysis the compiler classifies $f$ as being not $C^0$ continuous. For instance,* min() *is statically classified correctly, but the composition of a $C^0$ continuous function with a discontinuous function can give a discontinuity degeneracy: e.g. $f(x) = (2H(x) + x - 1)^2$: this can be simplified by a human to $f(x) = \{(x-1)^2,\ if x < 0,\ otherwise\ (x+1)^2\}$, which is $C^0$, not $C^1$, but the compiler identifies it as being not $C^0$ due to the Heaviside step.*

Definition 14. *A program $f \in e_c$ has part dependency on the sampling axis $x$ if for some intermediate value $h(g)$ where $h$ is a unary atomic function, $g$ is not statically continuous and statically depends on $x$, and $g$ is continuous at $(x, \vec{\theta})$, and $\partial g/\partial x$ exists, but $\partial g/\partial x$ is locally zero. For instance, $f(x, \theta) = \sin(\theta + \theta \cdot H(x))$. This can result in non-first order correct gradients for certain non-Dirac parameters, which are not the main focus of this paper.*

Definition 15. *A program $f \in e_c$ has a discontinuity with a $p$th order root along the sampling axis $x$ if for some intermediate value $H(g)$ with $g \in e_a$, $g$ has a $p$th order zero along $x$, i.e. $\partial^j g/\partial x^j = 0$ for $j = 0, \dots, p-1$ and $\partial^{p+1} g/\partial x^{p+1} \neq 0$. For instance, $f(x, \theta) = H((x+\theta)^3)$ has a discontinuity with 3rd order root at $x + \theta = 0$ because at those points $g = (x+\theta)^3$ has $g, \partial g/\partial x, \partial^2 g/\partial^2 x = 0, \partial^3 g/\partial x^3 = 6$.*

The above three definitions are only applied for programs in $\bar{e}_b$ and $\bar{e}_c$ ($e_b \subset e_c$ so the definitions also can be applied to $e_b$). Because discontinuities in $e_d$ are more difficult to characterize and our correctness claim in Theorem 1 does not guarantee anything about $\bar{e}_d$, we do not consider such programs.

## A.2 Existence of $\epsilon_f$ and $\epsilon_i^r$ in Theorem 1

We now characterize the $\epsilon_f$ and $\epsilon_i^r$ used in Theorem 1 and briefly justify its existence. We show a certain $\epsilon_f$ exists that has strong properties that are needed next in Appendix A.3 for the proof by induction over subsets of our DSL. More specifically, in Appendix A.3 we will next use the $\epsilon_f$ and $\epsilon_i^r$ with the below properties in the proof by induction of the first-order correctness properties (including kernel sizes that depend on $\epsilon_f$ and $\epsilon_i^r$) for appropriate subsets of our DSL on C-simple and D-simple sets. We first establish a lemma that we will use to obtain the first property.

Lemma 6. Isolated discontinuities: *Given a function $f \in \bar{e}_c$ evaluated at $(x, \vec{\theta}) \in \text{dom}(f)$, $\exists \epsilon > 0$ s.t. $\forall x' \in [x - \alpha\epsilon, x) \cup (x, x + \beta\epsilon]$, $f$ is continuous at $(x', \vec{\theta})$ along the x axis.*

*Proof sketch:* because $f$ is constructed from our DSL, it only consists of a finite number ($N$) of $H(g_i)$ with the property that there are no other Heaviside functions that depend on $H(g_i)$. Without loss of generality, we will only discuss the $N = 1$ case: $H(g)$ is the only source of discontinuity. The $N > 1$ case can be generalized by following the $N = 1$ proof, then take the intersection of the continuous intervals associated with each $H(g_i)$: this can allow us to recursively cover all of the functions in $\bar{e}_c$.

We first discuss when $f$ is discontinuous wrt $x$ at $(x, \vec{\theta})$. Because $f \in \bar{e}_c$, the discontinuity $H(g)$ satisfies either $g \in e_a$ being real analytic or $g \in e_b$ being piece-wise constant. We prove by induction and start with the base case $g \in e_a$. Because $f$ is discontinuous at $(x, \vec{\theta})$ wrt $x$, $\exists \epsilon' > 0$ such that discontinuity can be sampled $\forall \epsilon \in (0, \epsilon']$: $H(g(x + \alpha\epsilon, \vec{\theta})) \neq H(g(x, \vec{\theta}))$ or $H(g(x - \alpha\epsilon, \vec{\theta})) \neq H(g(x, \vec{\theta}))$. Therefore $g$ is not locally zero (wrt $x$). Because real analytic functions that are not locally zero have isolated zeros in 1D (sampling axis $x$), $\exists \epsilon > 0$ s.t. $(x, \vec{\theta})$ is the only zero for $g(x', \vec{\theta})$ within the interval $x' \in [x - \alpha\epsilon, x + \beta\epsilon]$, so we have the desired result for the case $g \in e_a$. In subsequent discussion, we refer to "within the interval" as meaning fixing $\vec{\theta}$ and consider the 1D restriction of g to the x axis. Now we have this is equivalent to $H(g)$ is continuous wrt x within the interval $[x - \alpha\epsilon, x) \cup (x, x + \beta\epsilon]$. We now prove the induction step of $g \in e_b$ and assume if $g$ is discontinuous wrt $x$, $\exists \epsilon > 0$ such that $g$ is continuous wrt $x$ in the interval $[x - \alpha\epsilon, x) \cup (x, x + \beta\epsilon]$. Because $g$ is piece-wise constant, discontinuities wrt $x$ for $H(g^+) \neq H(g^-)$ can only be sampled when $g$ itself is discontinuous wrt $x$. Therefore $g$ is continuous wrt $x$ in the interval $[x - \alpha\epsilon, x) \cup (x, x + \beta\epsilon]$ implies $H(g)$ is continuous wrt $x$ in the interval $[x - \alpha\epsilon, x) \cup (x, x + \beta\epsilon]$.

Now we discuss the case $f$ is continuous wrt $x$ at $(x, \vec{\theta})$. If $f \in e_a$, the conclusion is trivially true. We therefore still assume $f$ has intermediate values of the form $H(g)$. And we prove by induction similarly. For the base case $g \in e_a$, $H(g)$ is continuous along $x$ either indicates $g$ is locally zero wrt $x$ or $g(x, \vec{\theta}) \neq 0$. If $g$ is locally zero wrt $x$, $x$, $\epsilon$ can be chosen appropriately based on the size of the locally zero interval such that $g$ stays zero. If $g(x, \vec{\theta}) \neq 0$, because $g$ is real analytic wrt $x$, it is locally Lipschitz continuous wrt $x$, which justifies a local interval $x' \in [x - \alpha\epsilon, x + \beta\epsilon]$ such that $g(x', \vec{\theta}) \neq 0$. For the induction step where $g \in e_b$, because discontinuities wrt $x$ are isolated and $g$ is piece-wise constant, $\exists \epsilon > 0$ s.t. both $[x - \alpha\epsilon, x)$ and $(x, x + \beta\epsilon]$ are piece-wise constant, therefore $f$ is continuous wrt $x$ in the interval ■.

Lemma 7. *$\forall f \in \bar{e}_c$ that is either C-simple or D-simple at $(x, \vec{\theta}) \in \text{dom}(f)$, $\exists \epsilon_f(x, \vec{\theta}) > 0$ such that $\forall \epsilon \in (0, \epsilon_f]$ all of the following are satisfied:*

- *$\forall x' \in [x - (\alpha + \beta)\epsilon, x) \cup (x, x + (\alpha + \beta)\epsilon]$ such that $(x', \vec{\theta}) \in \text{dom}(f)$, we have $f$ is continuous at $(x', \vec{\theta})$ along $x$. This is a stronger property than the RHS of Lemma 5 so it is compatible with local expansions, and also for C-simple locations lets us when showing absolutely first-order correct to have no discontinuities in the neighborhood.*
- *$\forall g$ that is the intermediate value of $f$ and is not locally zero wrt $x$, $g(x + \beta\epsilon) \neq g(x - \alpha\epsilon)$. This lets our composition rule exclude the zero denominator.*
- *There are certain expressions $e$ that are used in our lemmas that evaluate to a nonzero value at $(x, \vec{\theta})$ and do not depend on $\epsilon$ which are derived from some intermediate values in the program $f$. For these, $1/(e + O(\epsilon)) = 1/e + O(\epsilon)$. This lets us rewrite Taylor expansions into the desired form for first-order correctness.*

*Proof sketch:* The existence of $\epsilon_f$ in the first requirement is a direct application of the isolated discontinuities property of Lemma 6. For the second requirement, if $g$ is discontinuous at $(x, \vec{\theta})$, according to Lemma 5 we have $g = a + b \cdot H(c)$. Because both $a, b$ are real analytic, $\epsilon_f$ doesn't exist implies $b(x, \vec{\theta}) = 0$, which is discontinuity degeneracy and is excluded from $\bar{e}_c$, therefore raising a contradiction. If $g(x, \vec{\theta})$ is continuous, then local expansion reduces to $g = a$ where $a \in e_a$. If $\frac{\partial a}{\partial x} \neq 0$, because $\frac{\partial a}{\partial x}$ is real analytic and locally Lipschitz continuous, $\exists \epsilon > 0$ such that $a$ is monotonic in the local region, therefore the requirement is satisfied. If $\frac{\partial a}{\partial x} = 0$, the requirement is violated only when $f$ is symmetric along the sampling axis $x$, which is excluded from C-simple points, therefore raising a contradiction. The third requirement is valid because in our proof, $O(\epsilon)$ is always used to express polynomials of $\epsilon$ with all other terms being locally bounded. Therefore if $\epsilon_f$ does not exist, it the means $O(\epsilon)$ involves a multiplication of $\epsilon$ with an unbounded term and contradicts how $O(\epsilon)$ is constructed in the proof.

Note in the first requirement, the local neighborhood is larger than the kernel support when pre-filtering at $(x, \vec{\theta})$. This is because we state relatively first-order correctness in a dilated set $D_i^r$, and when we pre-filter near the boundary of $D_i^r$, we still need the kernel support to be within $\text{dom}(f)$ and include only one discontinuity ■.

Once $\epsilon_f$ is defined, the existence of $\epsilon_i^r(x', \vec{\theta})$ can be easily justified using the remapping in Definition 11: $\epsilon_i^r(x', \vec{\theta}) = r\epsilon_f(\tau(x', \vec{\theta}), \vec{\theta})$.

## A.3 First-order Correctness Proof by Induction

In this subsection, we show that Theorem 1 can be proved by induction on different operators. We first report our conclusions for the base and each induction step. We then give an example of proving one of the induction steps. The other induction proof steps can be carried out similarly: we have carried them out but in the interests of space do not report all of the steps of them.

*A.3.1 Conclusions from Proofs by Induction.* We first report that our approximation is absolutely first-order correct for three base cases in Lemma 8. Because the base cases are continuous, $D_i^r$ is a null set. We therefore do not discuss relatively first-order correct.

Lemma 8. *Given $f$ in the following form, $\forall(x, \vec{\theta}) \in \mathrm{dom}(f)$ and $\forall\epsilon > 0$, $f$ is absolutely first-order correct.*

$$f(x, \vec{\theta}) = C, \text{ constant } C \in \mathbb{R}$$

$$f(x, \vec{\theta}) = x$$

$$f(x, \vec{\theta}) = \theta_i$$

We now report the induction results for unary operators $U \in \{H, h\}$ where $H$ denotes the Heaviside step function, and $h$ denotes continuous atomic functions. Because in Theorem 1, absolutely and relatively first-order correct are claimed for different set of points, their induction steps are different. As shorthand, for a function $f$ and $r \in (0, 1]$, we use first-order correct at $(x, \theta) \in D_i^r$ for intermediate value $g$ of $f$ to mean if $g$ is discontinuous wrt $\theta_i$ at $\tau(x, \vec{\theta})$, then it is relatively first order correct, and if $g$ is continuous wrt $\theta_i$ at $\tau(x, \vec{\theta})$, then it is absolutely first-order correct.

Lemma 9. *Given a function $f = U(g)$ with $f, g \in \bar{e}_c$, $\forall(x, \vec{\theta}) \in C_i$ that are C-simple and $\forall\epsilon \in (0, \epsilon_f(x, \vec{\theta})]$, if $g$ is absolutely first-order correct, then $f$ is absolutely first-order correct.*

Lemma 10. *Given a function $f = U(g)$ with $f, g \in \bar{e}_c$, $\forall r \in (0, 1]$, $\forall(x, \vec{\theta}) \in D_i^r$ that are D-simple, if $g$ is first-order correct for $\epsilon = r\epsilon_f(\tau(x, \vec{\theta}), \vec{\theta})$, then $f$ is relatively first-order correct for the same $\epsilon$.*

We now report the induction results for two binary operators $\{+, \cdot\}$: we use the symbol $\oplus$ in the next two lemmas to represent either addition or multiplication.

Lemma 11. *Given a function $f = g \oplus h$ with $f, g, h \in \bar{e}_c$, $\forall(x, \vec{\theta}) \in C_i$ that are C-simple and $\forall\epsilon \in (0, \epsilon_f(x, \vec{\theta})]$, if $g, h$ are absolutely first-order correct, then $f$ is absolutely first-order correct.*

Lemma 12. *Given a function $f = g \oplus h$ with $f, g, h \in \bar{e}_c$, $\forall r \in (0, 1]$, $\forall(x, \vec{\theta}) \in D_i^r$ that are D-simple, if $g, h$ are first-order correct for $\epsilon = r\epsilon_f(\tau(x, \vec{\theta}), \vec{\theta})$, then $f$ is relatively first-order correct for same $\epsilon$.*

As discussed in Section 4.1, the program set $e_a$ is statically continuous, therefore for any $f \in e_a$, $C_i = \mathrm{dom}(f)$ is always C-simple, and is therefore absolutely first-order correct. Because $f \in \bar{e_b}$ is piece-wise constant, both our approximation and reference gradients are 0 for $(x, \vec{\theta}) \in C_i$ with $\epsilon \in (0, \epsilon_f]$, and is therefore absolutely first-order correct. The almost everywhere results of Lemma 2 and Lemma 4 for the C-simple and D-simple properties in $C_i$ and $D_i^r$, respectively, combined with the induction proof on C-simple and D-simple points likewise immediately lead to the almost everywhere results in Theorem 1 for $\bar{e}_b$ and $\bar{e}_c$.

*A.3.2 Induction Proof Example.* In this section, we give an example proof for Lemma 10 with the unary operator $U = H$ and assuming $g$ is also discontinuous wrt $x$ at $\tau(x, \vec{\theta})$. Other cases or other lemmas in Section A.3.1 can all be proved similarly.

Because $f \in \bar{e}_c$, based on the DSL construction, the input arguments to step functions can either be piece-wise constant ($e_b$) or continuous ($e_a$). Therefore because $g$ is also discontinuous wrt $x$ at $\tau(x, \vec{\theta})$, we know $g \in e_b$, and can be locally expanded as $g = a_g + b_g \cdot H(c)$ with $a_g, b_g$ being constant, $c \in e_a$. Accordingly, $f$ can be expanded as $f = H(a_g + b_g \cdot H(c)) = \mathrm{sign}(b_g) \cdot H(c)$. The reference gradient is computed as follows. For simplicity, we denote $x_d$, the first component of $\tau(x, \vec{\theta})$ as the discontinuity location.

$$\begin{aligned}
\frac{\partial \hat{f}}{\partial \theta_i} &= \frac{\partial}{\partial \theta_i} \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} \mathrm{sign}(b_g) \cdot H(c) dx' \\
&= \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} \mathrm{sign}(b_g) \delta(c) \frac{\partial c}{\partial \theta_i} dx' = \frac{\mathrm{sign}(b_g) \frac{\partial c}{\partial \theta_i}}{(\alpha+\beta)\epsilon |\frac{\partial c}{\partial x}|} |_{x_d} \\
&\quad \text{Denominator nonzero: discontinuities with roots of order } n \geq 2 \\
&\quad \text{are excluded from } D_i \ (n = 2) \text{ or } \bar{e}_c \ (n \geq 3) \text{ respectively.}
\end{aligned} \tag{7}$$

Similarly, the reference gradient of $g$ can be computed.

$$\begin{aligned}
\frac{\partial \hat{g}}{\partial \theta_i} &= \frac{\partial}{\partial \theta_i} \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} (a_g + b_g \cdot H(c)) dx' \\
&= \frac{1}{(\alpha+\beta)\epsilon} \frac{b_g \frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|} |_{x_d} \\
&\quad a_g, b_g \text{ are constants}
\end{aligned} \tag{8}$$

We now apply our gradient approximation and start from the LHS of Equation 3 to show its RHS. For simplicity, we assume $H(c^+) = 1$ and $H(c^-) = 0$. The opposite case can be easily proved similarly.

$$\begin{aligned}
\frac{\frac{\partial_O f}{\partial \theta_i}}{\frac{\partial \hat{f}}{\partial \theta_i}} &= \frac{\frac{\partial_O g}{\partial \theta_i} / |g^+ - g^-|}{\frac{\mathrm{sign}(b_g) \frac{\partial c}{\partial \theta_i}}{(\alpha+\beta)\epsilon |\frac{\partial c}{\partial x}|} |_{x_d}} \\
&\quad \text{(Applying our rule to numerator and using Eq 7 for denominator.)} \\
&= \frac{\frac{\partial_O g}{\partial \theta_i} \frac{\partial \hat{g}}{\partial \theta_i} / (|g^+ - g^-| \frac{\partial \hat{g}}{\partial \theta_i})}{\frac{\mathrm{sign}(b_g) \frac{\partial c}{\partial \theta_i}}{(\alpha+\beta)\epsilon |\frac{\partial c}{\partial x}|} |_{x_d}} = \frac{\frac{\partial \hat{g}}{\partial \theta_i} (1 + O(\epsilon) / |g^+ - g^-|}{\frac{\mathrm{sign}(b_g) \frac{\partial c}{\partial \theta_i}}{(\alpha+\beta)\epsilon |\frac{\partial c}{\partial x}|} |_{x_d}} \\
&\quad \text{Apply Equation 3 to } g. \\
&= \frac{\frac{1}{(\alpha+\beta)\epsilon} \frac{b_g \frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|} |_{x_d} (1 + O(\epsilon)) / |g^+ - g^-|}{\frac{\mathrm{sign}(b_g) \frac{\partial c}{\partial \theta_i}}{(\alpha+\beta)\epsilon |\frac{\partial c}{\partial x}|} |_{x_d}} \\
&\quad \text{Using Equation 8} \\
&= \frac{\mathrm{sign}(b_g) \cdot b_g}{|g^+ - g^-|} + O(\epsilon) = \frac{\mathrm{sign}(b_g) \cdot b_g}{|a_g + b_g - a_g|} + O(\epsilon) \\
&\quad \text{Assuming } H(c^+) = 1. \text{ Denominator nonzero because} \\
&\quad \text{discontinuity degeneracy is excluded from } \bar{e}_c. \\
&= 1 + O(\epsilon)
\end{aligned} \tag{9}$$

## B MOTIVATION FOR FUNCTION COMPOSITION RULE

In this section, we further motivate our choice of the function composition rule in Table 2. For function compositions $h \circ g$, this rule applies to atomic continuous functions $h$. The compiler chooses between two equations to avoid numerical instability while differentiating discontinuous functions. Similar to multiplication, directly applying the AD rule to discontinuities leads to incorrect results. For example, the same function $f = H(x+\theta) \cdot H(x+\theta)$ we discussed for multiplication can also be expressed as $f = H(x+\theta)^2$, which can be viewed as applying a square function $(\cdot)^2$ to $H(x+\theta)$. Naively applying the AD function composition rule to this function combined with our Heaviside step gradient rule results in the following.

$$\frac{\partial_{AD} f}{\partial \theta} = 2H(x+\theta)\frac{1}{|2\epsilon|} = \frac{H(x+\theta)}{\epsilon} \neq \frac{\partial \hat{f}}{\partial \theta}$$

The result from using the AD function composition is again incorrect and similarly to multiplication: this is due to AD always being biased to one branch or the other. On the contrary, our composition rule samples on both branches and is robust at discontinuities.

$$\frac{\partial_O f}{\partial \theta} = \frac{H^+ - H^-}{H^+ - H^-}\frac{1}{|2\epsilon|} = \frac{1}{2\epsilon} = \frac{\partial \hat{f}}{\partial \theta}$$

# C DETAILS FOR TERNARY SELECT OPERATOR

## C.1 General Ternary Select Operator

In this section, we discuss branching with inequality conditions that can be written in the form

$$F(p, l, r) = \text{select}(p > 0, l, r) = r + (l - r) \cdot H(p) \tag{10}$$

Branching with complex Boolean expressions will be discussed next in Section C.2. The rule developed in this section can also be used in all inequality and equality comparisons: $p > q$ is equivalent to $p - q > 0$, $p < 0$ can be rewritten as $-p > 0$, and $\text{select}(p \geq 0, l, r)$ is equivalent to $\text{select}(-p > 0, r, l)$. The equality comparison $p == q$ can be written as the Boolean expression $p \geq q \wedge p \leq q$, which can be differentiated by the rules discussed next in Section C.2.

Instead of directly applying the multiplication rule in Table 2, we design a specialized rule for branching that uses a similar register space as if the gradient is approximated by AD. The specialized rule utilizes the fact that our pre-filtering kernels $U[-\Delta x, 0]$ and $U[0, \Delta x]$ are always between the evaluation location and a neighboring location on the sampling grid, therefore for an intermediate value $g$, $g(x) = g^+$ or $g(x) = g^-$ always holds. For simplicity, we denote the samples at two ends of the prefiltering kernel as $g, g_n$, where $g = g(x)$ and $g_n$ is evaluated at the neighoring location at the other end of the kernel support. The gradient rule for the efficient ternary select operator is shown in Equation 11.

$$\frac{\partial_O F}{\partial \theta_i} = (l_n - r_n)\frac{\partial_k H(p)}{\partial \theta_i} + \text{select}(p > 0, \frac{\partial_k l}{\partial \theta_i}, \frac{\partial_k r}{\partial \theta_i}) \tag{11}$$

Note Equation 11 is equivalent to differentiating $F$ in its right hand side form in Equation 10, but applying a simplified multiplication rule as in Equation 12. This results in an identical first-order correctness property as differentiating $F$ using the original multiplication rule, which can be proved using a similar induction step.

$$\frac{\partial (g \cdot h)}{\partial \theta_i} = h\frac{\partial g}{\partial \theta_i} + g_n\frac{\partial h}{\partial \theta_i} \tag{12}$$

For special operators such as min() and max(), although they are also expanded using a ternary select, because the compiler can statically identify they are $C^0$ continuous, their gradients through the branching always use the AD gradient rule.

## C.2 Boolean Conditions for Ternary Select

This section discusses specialized rules for branching conditions in $F = \text{select}(B, l, r)$ such that $B$ can be decomposed into the Boolean expressions of $n$ inequality clauses: $C_i = c_i > 0, i = 1, ..., n$.

Similar to sampling floating point values at two ends of the filtering kernel, we can also sample Boolean values such as $B^+, B^-$. Disagreeing Boolean samples indicates the presence of discontinuity, such as when $B^+ \oplus B^-$ evaluates to true, where $\oplus$ indicates XOR. When a discontinuity is sampled, our single discontinuity assumption allows us to infer that every discontinuous Boolean clause depends on the same underlying floating point valued function. Therefore, if we further sample every Boolean clause used in $F$ and identify two different clauses $C_i, C_j$ disagree simultaneously: $(C_i^+ \oplus C_i^-) \wedge (C_j^+ \oplus C_j^-)$ evaluates to true, then we assume $C_i$ and $C_j$ are equivalent at the current location. Our rule will traverse and sample each Boolean clause $C_i$ in an arbitrary order, and replace $\frac{\partial H(p)}{\partial \theta_i}$ in Equation 11 with $\frac{\partial H(c_i)}{\partial \theta_i}$ for the first clause $C_i$ where a discontinuity is sampled.

# D DETAILS FOR IMPLICITLY DEFINED GEOMETRY

In shader programs, a common way to define geometry is to encode it as an implicit surface, or the zero set of some mathematical function, and iteratively estimate ray-geometry intersections through methods like ray marching [Perlin and Hoffert 1989] or sphere tracing [Hart 1996]. While ray marching and sphere tracing loops themselves are programs, and can be differentiated using rules introduced in Section 2, this usually results in a long gradient tape because the number of loop iterations can be arbitrarily large. As an alternative, we can bypass the root finding process and directly approximate the gradient using the implicit function theorem. Our rule is motivated by [Yariv et al. 2020]: we extend their result for differentiating points that lie on the zero set of the implicit geometry to differentiating discontinuities caused by object silhouettes or interior edges. Similarly, [Li et al. 2020] apply the implicit function theorem to differentiate discontinuities caused by 2D line strokes. Their result, however, is limited to a specific type of function in 2D ($n$-th order polynomials). [Gargallo et al. 2007] also develop a gradient rule for visibility change to the implicitly defined surface. Their derivation only applies to C1 continuous geometry (Appendix D.1) and does not handle discontinuities caused by the intersection of surfaces (Appendix D.2). Unlike previous methods, our rule can differentiate the discontinuities generated by implicit geometries represented as arbitrary signed distance functions.

We assume the geometry is implicitly defined by the signed distance function that depends on 3D locations $\vec{\boldsymbol{p}}$ and scene parameters $\vec{\boldsymbol{\theta}}$: $f(\vec{\boldsymbol{p}}(x, y), \vec{\boldsymbol{\theta}}) = 0$. The 3D locations further depends on image coordinates $(x, y)$, as they are on the rays casting from the camera to the geometry: $\vec{\boldsymbol{p}} = \vec{\boldsymbol{o}}(x, y) + t \cdot \vec{\boldsymbol{d}}(x, y)$ where $o, d, t$ are camera origin, ray direction, and distance from camera to geometry respectively. Because our implementation uses $x, y$ as sampling axes, we will discuss differentiating the geometry discontinuities by pre-filtering along the $x$ axis and asuming $y$ is a fixed constant. Given arbitrary

$\vec{\theta}$, the discontinuity location $x_d(\vec{\theta})$ along $x$ axis can be defined as for a local neighborhood around $x_d$, $x < x_d$ and $x > x_d$ evaluates to different branches of the geometry. For silhouettes, this indicates the Boolean on whether the ray has hit the geometry is evaluated differently; and for interior edges, this corresponds to $f$ evaluates to different branches at different sides of $x_d$. Therefore, the discontinuities can be represented as $H(x - x_d)$. In the forward pass, the user specifies the SDF using a RaymarchingLoop primitive and the compiler automatically expands a ray marching loop to approach the zero set of the SDF. The value of $x_d$ is never explicitly computed in the program. In the backward pass, the discontinuity is sampled by evaluating Boolean conditions described above, and back-propagation is carried out by computing $\frac{\partial x_d}{\partial \theta_i}$. The compiler will classify the cause of the discontinuity based on whether the camera ray intersects a locally C1 smooth geometry or not, and applies the implicit function theorem to each case described in Appendix D.1 and D.2.

### D.1 Camera ray intersecting locally C1 smooth geometry.

In this case, the discontinuity is implicitly defined by the ray hitting the zero set of the function $f$ while perpendicular to the normal direction of $f$ at the intersection.

$$f(\vec{p}, \vec{\theta}) = 0 \tag{13a}$$

$$< \frac{\partial f}{\partial \vec{p}}, \vec{d} > = 0 \tag{13b}$$

We can now differentiate wrt an arbitrary parameter $\theta_i$ on both sides of Equation 13a.

$$\frac{\partial f}{\partial \theta_i} + < \frac{\partial f}{\partial \vec{p}}, \frac{\partial \vec{o}}{\partial \theta_i} + \frac{\partial \vec{o}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + \vec{d}\frac{\partial t}{\partial \theta_i} >= 0 \tag{14}$$

Equation 14 can be simplified by inserting Equation 13b.

$$\frac{\partial f}{\partial \theta_i} + < \frac{\partial f}{\partial \vec{p}}, \frac{\partial \vec{o}}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial \theta_i} > + < \frac{\partial f}{\partial \vec{p}}, \frac{\partial \vec{o}}{\partial x_d} + t\frac{\partial \vec{d}}{\partial x_d} > \frac{\partial x_d}{\partial \theta_i} = 0 \tag{15}$$

Rearranging Equation 15 results in Equation 16.

$$\frac{\partial x_d}{\partial \theta_i} = -\frac{\frac{\partial f}{\partial \theta_i} + < \frac{\partial f}{\partial \vec{p}}, \frac{\partial \vec{o}}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial \theta_i} >}{< \frac{\partial f}{\partial \vec{p}}, \frac{\partial \vec{o}}{\partial x_d} + t\frac{\partial \vec{d}}{\partial x_d} >} \tag{16}$$

### D.2 Camera ray intersecting C1 discontinuous geometry.

Many implicit functions are only C0 continuous. For example, constructive solid geometry (CSG) operators such as union or intersection use max or min to combine different implicit functions, causing the resulting function to be C0 continuous. These operations can generate silhouette or interior edges whenever two smooth surfaces intersect. For example, a box can be viewed as the intersection of multiple implicitly defined half-spaces. In this section, we assume the intersection is generated by two C1 continuous surfaces $f_0$ and $f_1$. Efficiently determining $f_0, f_1$ is discussed in Appendix D.3.

Our derivation starts with assuming the ray is at the zero set for both $f_0$ and $f_1$.

$$\begin{aligned} f_0(\vec{p}, \vec{\theta}) &= 0 \\ f_1(\vec{p}, \vec{\theta}) &= 0 \end{aligned} \tag{17}$$

We now differentiate wrt $\theta_i$ to both equations in Equation 17 .

$$\begin{aligned} &\frac{\partial f_0}{\partial \theta_i} + < \frac{\partial f_0}{\partial \vec{p}}, \frac{\partial \vec{o}}{\partial \theta_i} + \frac{\partial \vec{o}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + \vec{d}\frac{\partial t}{\partial \theta_i} >=0 \\ &\frac{\partial f_1}{\partial \theta_i} + < \frac{\partial f_1}{\partial \vec{p}}, \frac{\partial \vec{o}}{\partial \theta_i} + \frac{\partial \vec{o}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + \vec{d}\frac{\partial t}{\partial \theta_i} >=0 \end{aligned} \tag{18}$$

Next we can rearrange Equation 18 to separate out $\frac{\partial t}{\partial \theta_i}$.

$$\begin{aligned} \frac{\partial t}{\partial \theta_i} &= -\frac{\frac{\partial f_0}{\partial \theta_i} + < \frac{\partial f_0}{\partial \vec{p}}, \frac{\partial \vec{o}}{\partial \theta_i} + \frac{\partial \vec{o}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} >}{< \frac{\partial f_0}{\partial \vec{p}}, \vec{d} >} \\ \frac{\partial t}{\partial \theta_i} &= -\frac{\frac{\partial f_1}{\partial \theta_i} + < \frac{\partial f_1}{\partial \vec{p}}, \frac{\partial \vec{o}}{\partial \theta_i} + \frac{\partial \vec{o}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} >}{< \frac{\partial f_1}{\partial \vec{p}}, \vec{d} >} \end{aligned} \tag{19}$$

Because the left hand sides of the two lines in Equation 19 are identical, their right hand side terms should also be identical. We can therefore derive $\frac{\partial x_d}{\partial \theta_i}$ as in Equation 20.

$$\begin{aligned} \frac{\partial x_d}{\partial \theta_i} &= -\frac{a-b}{c-d} \\ a &= (\frac{\partial f_0}{\partial \theta_i} + < \frac{\partial f_0}{\partial \vec{p}}, \frac{\partial \vec{o}}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial \theta_i} >) < \frac{\partial f_1}{\partial \vec{p}}, \vec{d} > \\ b &= (\frac{\partial f_1}{\partial \theta_i} + < \frac{\partial f_1}{\partial \vec{p}}, \frac{\partial \vec{o}}{\partial \theta_i} + t\frac{\partial \vec{d}}{\partial \theta_i} >) < \frac{\partial f_0}{\partial \vec{p}}, \vec{d} > \\ c &= < \frac{\partial f_0}{\partial \vec{p}}, \frac{\partial \vec{o}}{\partial x_d} + t\frac{\partial \vec{d}}{\partial x_d} >< \frac{\partial f_1}{\partial \vec{p}}, \vec{d} > \\ d &= < \frac{\partial f_1}{\partial \vec{p}}, \frac{\partial \vec{o}}{\partial x_d} + t\frac{\partial \vec{d}}{\partial x_d} >< \frac{\partial f_0}{\partial \vec{p}}, \vec{d} > \end{aligned} \tag{20}$$

### D.3 Efficient back-propagation algorithm

One major challenge for efficiently implementing gradients discussed in Section D.1 and D.2 is that at compile time, it is unknown whether the silhouette is caused by a smooth surface, or the intersection of two arbitrary sub-surfaces. For example, if an implicit function $f$ is defined by applying CSG operations to $n$ C1 smooth sub-functions and we assume the discontinuity is caused by an intersection, at compile time we have $\binom{n}{2}$ possible combinations $f_0$ and $f_1$ in Equation 20 as the decision of which two surfaces are intersecting is not determined until run time. Because our backends always take all branches, the naive implementation would have $O(n^2)$ complexity.

In this section, we provide a specialized algorithm based on our observations of implicit scene representations. Note that we make heuristic assumptions to implicit functions that may be violated by a carefully designed counter-example. The algorithm therefore only serves as a heuristic optimization that improves the efficiency in differentiating most implicit scenes. Any implicit function that violates our heuristic assumption can still be back-propagated through their root finding process using the gradient rules described in Section 4.2.

Our compiler makes the assumption that the branching decision in the implicit functions are represented as min or max, and all values being compared with are scaled similarly (e.g. they all represent the Euclidean distance to some primitives). This assumption holds for most implicit scene representations, such as all the signed distance fields and the CSG operators described in [Inigo Quilez 2021].

Our compiler first statically analyzes whether a certain branch depends on an implicitly defined geometry discontinuity and back-propagates accordingly. For geometry discontinuities, the compiler further classifies whether the discontinuity is due to the camera ray is intersecting a C1 smooth geometry (Section D.1) or not (Section D.2). For intersecting surfaces, we dynamically modify the branching condition in the the signed distance function to obtain $f_0$ and $f_1$. The rest of this section discusses each component in detail.

*Sampling geometry discontinuities.* To identify implicitly defined geometries, our DSL introduces a new primitive: RaymarchingLoop. It is defined using camera orientation $\vec{\boldsymbol{o}}$, ray direction $\vec{\boldsymbol{d}}$ and the user-defined signed distance function (SDF) $f$ and outputs a Boolean condition $B$ on whether the loop has converged, as well as the distance $t$ from the camera to geometry. Optionally, the SDF can be defined to output the surface normal and sub-surface labelling using identical branching conditions as the original SDF. In the forward pass, the SDF will be evaluated at $\vec{\boldsymbol{p}} = \vec{\boldsymbol{o}} + t \cdot \vec{\boldsymbol{d}}$ for these values.

In the forward pass, the compiler automatically expands the primitive into a loop that finds the zero set to the implicit function using ray marching. In the backward pass, we sample the silhouette discontinuity by evaluating $B$, and interior edge discontinuity is found by sampling all branch conditions defined in $f$ evaluated at $\vec{\boldsymbol{p}} = \vec{\boldsymbol{o}} + t \cdot \vec{\boldsymbol{d}}$. Because $t$ is the output of the black-box primitive RaymarchingLoop, it is viewed as a discontinuous function $t = select(x > x_d, t^-, t^+)$ whenever a silhouette or interior edge discontinuity is sampled at $x_d$. However, if the interior edge is caused by the camera ray being tangent to a foreground geometry without changing any Boolean conditions in $f$, our method will fail to sample such a discontinuity.

*Classifying geometry discontinuities.* For discontinuities that depend on RaymarchingLoop, the compiler applies Section D.1 if the camera ray is tangent to the geometry and Section D.2 otherwise. We use a simple classification that works well in all our experiments: a ray is tangent to the geometry if $\langle \frac{\partial f}{\partial \vec{p}}, \vec{\boldsymbol{d}} \rangle \leq 0.1$

*Identifying intersecting sub-surfaces.* If the geometry discontinuity is classified as caused by intersection of sub-surfaces $f_0$ and $f_1$, our algorithm modifies the branching condition in $f$ so that evaluating Equation 20 scales linearly to the number of sub-surfaces.

We start by observing that at least one of the sub-surfaces can be easily differentiated by directly applying AD to the original function $f$. If we denote $f_0$ as the sub-surface chosen by the current branching configuration, applying AD to $f$ is equivalent to differentiating $f_0$.

Differentiating the other intersecting sub-surface $f_1$ is more tricky. The compiler modifies the branching conditions computed in $f$, such that the new conditions evaluates to a different branch that represents $f_1$. This is achieved based on another observation: a ray hitting the intersection of $f_0$ and $f_1$ indicates $f$ makes a "close decision". This means for some branch $\min(a, b)$ (or max), the values of $a$ and $b$ must be almost identical: $f$ should still evaluate to 0 if we flip the condition and chooses the other branch and hit $f_1$ instead.

To correctly modify the branching, our compiler iterates through every condition and finds the one $\mathbf{c}^*$ with minimum absolute difference on both sides of the comparison (with $O(n)$ complexity, $n$ being the number of min/max operators). If the condition $\mathbf{c}^*$ evaluates to branch $a$ instead of branch $b$, intuitively we can simply set $a$ to infinity (for min) or negative infinity (for max) so that branch $b$ is forced to be taken. However, this introduces artifacts if $a$ is used elsewhere. We instead invert every condition where $a$ gets chosen.

### D.4 Caveat when combined with Random Variables

The random variables introduced in Section 7.2 cannot be combined with the RaymarchingLoop primitive because key assumptions made for its gradient derivation will be violated by the introduction of the random variables. On one hand, the specialized rule makes the assumption that at the discontinuity of the geometry, the ray is either tangent to the surface or it is on the zero set of two sub-surfaces. These will approximately hold as long as the image resolution is high enough. On the other hand, the random variables generate great variation in the scene parameters, therefore neighboring pixels on different sides of the geometry discontinuity may correspond to 3D points that are actually very far from the silhouette, violating the assumptions made in Appendix D.1 and D.2. For these reasons we always disable the random variables for Dirac parameters that RaymarchingLoop primitive depends upon.

It is possible to combine random variable with implicitly defined geometry by resorting to the general gradient rules introduced in Sections 4.2 and 6.2. Because ray marching loop can have arbitrarily many iterations, the gradient program can be inefficient due to the long tape. Note that for simple geometries, it is also possible to analytically compute ray object intersection.

## E DETAILS FOR QUANTITATIVE METRIC

Because different methods make different prefiltering assumptions, the kernel $K$ for the LHS $\nabla(K * f) = K * (\nabla f)$ of Equation 4 is chosen differently for different methods so all methods give the same desired kernel $K^*$ when considering both the prefiltering internal to the method and the prefiltering associated with $K$.

**Ours:** as stated in Section 5, our gradient is approximating that of a pre-filtered function using a 1D box kernel, whose size is identical to either dimension of $K$. Because $K = K_x * K_y * K_{\vec{\theta}}$ is a separable kernel, assuming our gradient is pre-filtered on the $y$ axis, the integrand can be estimated as follows.

$$(K_x * K_y * K_{\vec{\theta}}) * (\nabla f) = (K_x * (\nabla f)) * K_y * K_{\vec{\theta}} \approx \nabla_O f * K_y * K_{\vec{\theta}}$$

Here $\approx$ indicates only the error made by our approximation. We obtain a similar result $K_x * \nabla_O f * K_{\vec{\theta}}$ when ours is prefiltered along y. In practice, because our approximation adaptively chooses between $x$ and $y$ as sampling axes (Section 6.1), for a given pixel, the integrand is computed by first deciding which axis to prefilter, then sampling along the orthogonal axis to compute the integrand.

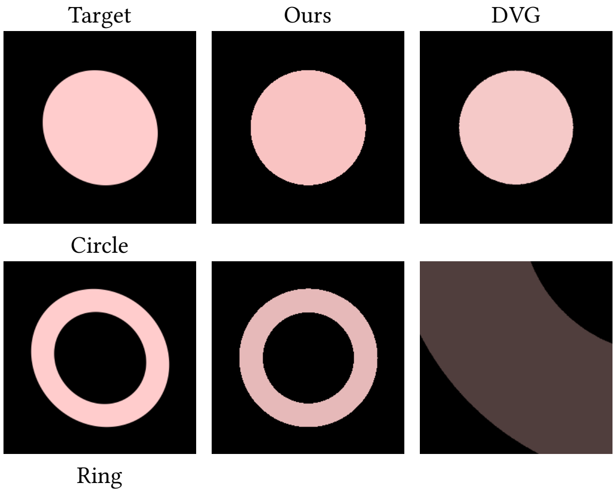


Fig. 14. Comparison of the optimized result for ours and DVG. Because each task restarts with 100 random initializations, here we show for each task, the result whose error corresponds to the median of 100 restarts. The target images are rendered with slightly elliptical shapes to avoid either ours or DVG forming a perfect reconstruction. For the circle shader, both Ours and DVG converge close enough to the target image. But for the ring shader, DVG is unable to converge because of a bug discussed in Figure 15.

**FD and variants:** because FD is approximated based on the original function value, the integrand is directly sampled using $K$.

**TEG:** their compiler handles the pre-filtering of image space as integration, and similar to ours, they resort to sampling for any extra integration after a single integral of Dirac delta functions. To avoid incurring too many samples, pre-filtering in parameter space is handled by Monte-Carlo sampling implemented outside their DSL similar to ours and FD.

Our compiler will randomly sample a unit ray in the parameter space and sample $\vec{\theta}_0, \vec{\theta}_1$ by extending the random ray in both directions to a fixed distance around a center parameter, then integrate along a straight line segment between $\vec{\theta}_0, \vec{\theta}_1$. We always use $10^5$ samples for the RHS, $10^4$ samples for the quadrature of the LHS, and 1 sample for pre-filtering the integrand of the LHS. We find the noisy integrand estimate is smoothed out because the outer integration is sampled so densely. However, for TEG, we use 10 samples for the inner integrand of the LHS because TEG constructs the pre-filtering in image space as a two-dimensional integration, and the symbolic integration of the Dirac delta only eliminates the inner integral. Therefore, the outer integral is sampled by additional quadrature using their implementation of the trapezoid rule, to avoid aliasing and high error that occur with smaller sample counts.

## F ADDITIONAL FIGURES COMPARING WITH DVG

Here we include Figure 14 and 15 comparing ours and DVG, as discussed in Section 8.1.2.

## G DETAILS FOR ROPE EXPERIMENT IN SECTION 8.3.1

We imagine the rope application can be useful in some interactive applications, therefore it involves several simple manual decisions. But the optimization is still carried out automatically to find the parametric representation of the rope.

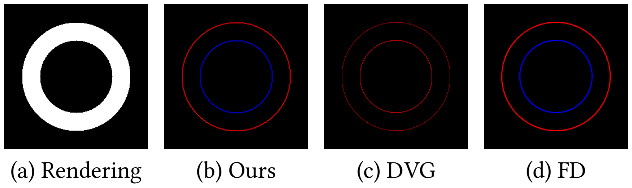


Fig. 15. For a single channel rendering of the ring shader (a), we evaluate the gradient of pixel color wrt the radius parameter and generate per pixel gradient maps for ours (b), DVG (c) and finite difference (FD) (d). Red indicates the gradient is positive and blue indicates negative. Our gradient agrees with FD, while DVG's gradient for the inner circle has opposite direction as ours and FD.

We manually pick 6 key frames for the rope shown as Knot A Figure 12 and 8 key frames for the second rope. Because these animations are expressed as a combination of filled shapes / strokes in html, we further increase the stroke width so that the dark rope edge is more salient. This greatly helps optimizing depth when a rope overlaps with itself, because the filled color is identical for both overlapping pieces, and the edge stroke is the only cue for resolving the depth correctly.

To ensure semantic continuity, our framework optimizes key frames in sequential order, and always initializes the new optimization based on parameters from the previous key frame. For every key frame, our framework first classifies whether a new rope segment should be introduced: at the first key frame or when a new color (representing a different rope) appears in the current frame. If new rope is *not* needed, our framework further decides whether a new segment should be added as appending to the tail of the rope, or subdividing the existing last segment. This is done by randomly sample new spline segments and evaluate their L2 loss with the target image. If the loss is always larger than without adding the new segment, we initialize by subdividing the existing rope, otherwise, we choose 5 sampled configurations with minimum loss as initialization. If a new rope is added, we manually click on the two ends of the new rope and use the coordinates for initialization. This helps both to increase the convergence rate compared to random initialization, as well as indicating the direction of the rope, so that new segments are initialized from the correct end of the rope for the next key frames. At the last key frame, an additional optimization process is applied to search for depth-related parameters only. Because our spline representation is not pixel-wise perfectly reconstructing the target animation frames, the overlapping regions may not correspond exactly. This occasionally lead to the problem that the optimal depth parameters in the L2 sense do not correspond to human intuition, as they may encourage intersection of ropes to lower the loss objective. Therefore, human effort may be involved to reject these optimization results. Finally, the optimized depth parameters will be passed back to all previous optimized key frames to ensure correct layering throughout the animation.