Exploiting Program Representation with Shader Applications

Yuting Yang

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Adam Finkelstein

May 2023

Abstract

Programs are widely used in content creation. For example, artists design shader programs to procedurally render scenes and textures, while musicians construct "synth" programs to generate electronic sound. While the generated content is typically the focus of attention, the programs themselves offer hidden potential for transformations that can support untapped applications. In this dissertation, we will discuss four projects that exploit the program structure to automatically apply machine learning or math transformations as if they were manually designed by domain experts. First, we describe a compiler-based framework with novel math rules to extend reverse mode automatic differentiation so as to provide accurate gradients for arbitrary discontinuous programs. The differentiation framework allows us to optimize procedural shader parameters to match target images. Second, we extend the differentiation framework to audio "synth" programs so as to match the acoustic properties of a provided sound clip. We next propose a compiler framework to automatically approximate the convolution of an arbitrary program with a Gaussian kernel in order to smooth the program for visual antialiasing. Finally, we explore the benefit of program representation in deep-learning tasks by proposing to learn from program traces of procedural fragment shaders – programs that generate images. In each of these settings, we demonstrate the benefit of exploiting the program structure to generalize hand-crafted techniques to arbitrary programs.

Acknowledgements

I would first like to thank my Ph.D. advisor Adam Finkelstein. Thanks for being super supportive of all the ideas I want to explore, and providing invaluable inspiration and advice for my research. I have learned a lot from Adam, especially about how to pitch my project and clearly communicate with others. I am also very grateful to Adam for being a source of encouragement during my low point in my Ph.D. when I struggle with papers being constantly rejected. Your support and understanding are crucial for me to stay enthusiastic to finish this journey.

I would also like to thank my long-term collaborator and previous advisor Connelly Barnes. Connelly is the first person to introduce me to the academic path and has been instrumental in shaping my research skills and interests. He has since become a mentor in my research, career, and life. I really enjoy our times discussing fun math since when we were back in UVA.

My gratitude also goes to my family. Dad, you have always been my role model, and finally, I'm achieving something you aren't: finishing my Ph.D. degree lol. Mom, thanks for being my biggest supporter and always having faith in me. Grandpa, you are the first scientist I know and are the reason I'm interested in academic research. Thanks for enlightening my childhood with your intelligence.

In addition, I would like to thank my Adobe collaborators Andrew Adams and Zeyu Jin, and my Ph.D. committee members Szymon Rusinkiewicz, Felix Heide and Jia Deng. Thanks for your invaluable help with my research. I also want to thank my lab mates for your help and inspiration: Jiaqi Su, Yunyun Wang, and Pranay Manocha. Many thanks to my best friend Zhiheng Zuo for constantly being my emotional support. Thanks to my friends who accompany me through my lows and highs in life: Yan Che, Yuhui Lv, Lin Gong, Deying Kong, Xi Chen, Zhongqiao Gao, Zheng Shi, Yue Tan, Fangyin Wei, Zhan Xu and so on. Lastly, thanks to my dog Xiaolang, I really miss you.

To my dad.

I know you are proud of me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Programs are almost everywhere in our daily lives, such as in our cars, mobile phones, or TVs. In the realm of content creation, program representations are also frequently used as part of the design process. For example, artists design shader programs to procedurally render scenes and textures (Figure 1.1), while musicians construct "synth" programs to generate electronic sound. Even less experienced programmers or non-programmers could also utilize visual design or visual programming environments such as Adobe Substance Designer or Max/MSP to programmatically describe the content creation process.

Typically, the generated content is the focus of attention while the programmatic synthesis process is overlooked and viewed as a black box. For example, if a photographer purchases Adobe Photoshop to retouch their photos, they will only have access to the elements and parameters provided by the software executable, but not the program that compiles to the executable. However, the actual program that describes the algorithm offers hidden potential for transformations that can support untapped applications, such as optimizing the original pipeline, or adding customized features.

Furthermore, even with the program representation, manually modifying the program requires both manual effort as well as domain expertise to prevent bugs and errors. For example, to improve the runtime performance of a program, a domain expert needs to

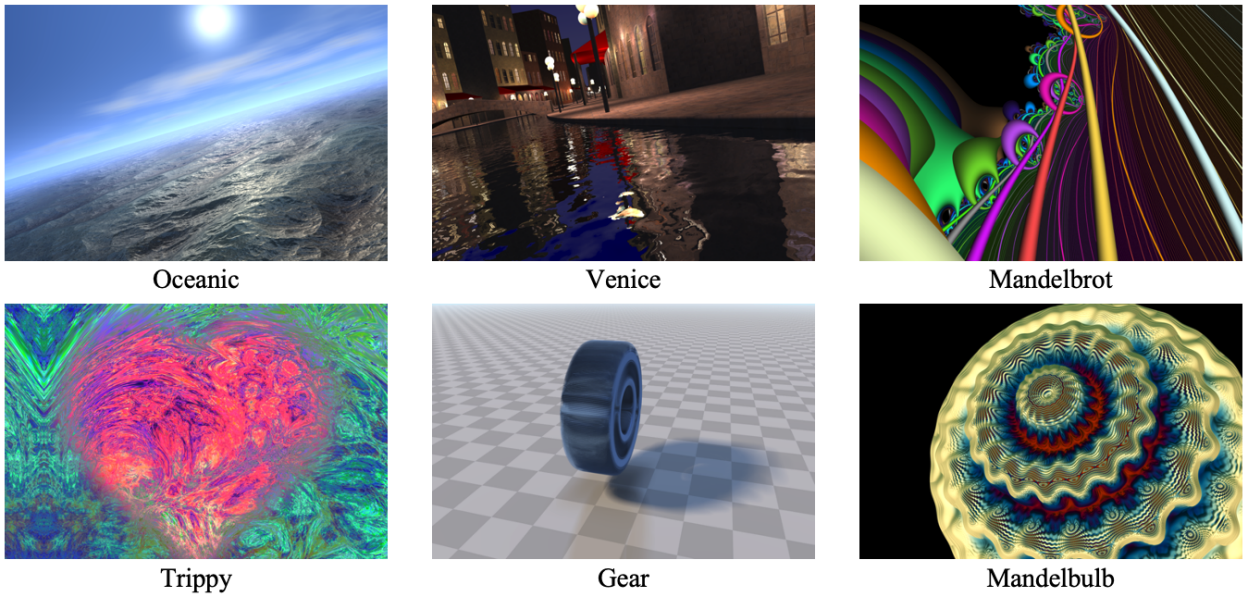


Figure 1.1: Example outputs from shader programs that procedurally render the scenes.

manually make optimization decisions such as which parts of the program can be parallelized, or which memory consumption can be preallocated. The manually optimized program may also suffer from subtle bugs such as race conditions in multithreading, which again requires additional manual effort to identify and fix.

We define a *program transformation* as a process $f \to g$ that constructs the output program g from a given input program f for tasks such as optimizing f in terms of runtime and memory footprint or generating the gradient of f. We further define an *automatic compiler framework* as a pipeline that automatically applies a certain type of program transformation to an *arbitrary* input program where the solution typically resembles the expertise of a human expert but without having to spend the manual labor. In this dissertation, we will explore a variety of tasks and propose automatic compiler frameworks that both exploit the program representation and automate the program transformation at the same time. We further demonstrate the applicability of these frameworks on shader programs.

1.1 Motivation

Researchers have leveraged the program representation to design automatic compiler frameworks for various tasks, such as shader level-of-detail [54], and inverse engineering the shape modeling [81]. More generally, automatic differentiation (AD) analyzes the algebraic computation to generate symbolic gradients for arbitrary programs [12], and TEG further extends AD to integrals over a limited set of discontinuous programs [10]. The program structure also helps researchers to develop automatic frameworks to efficiently schedule the computation [120, 2], as well as translate programs to other programming languages with verified correctness [3].

This section further motivates the benefit of exploiting program representations with a detailed discussion on a concrete example: differentiating a continuous program. We first formally define the transformation, then discuss three different approaches: assuming the program is a black box; manually deriving the solution from the program representation; and finally, automatically applying the transformation through a compiler framework.

Programs that carry out floating point computation can be viewed as math functions. Such programs are frequently used for generating visual and audio content, because images and audio signals are a collection of floating point samples. Differentiating the program can be essential for many tasks such as gradient-based optimization in machine learning. Formally, we define the differentiation transformation in Equation 1.1 for a continuously differentiable input program f. For the motivating examples throughout this chapter, we assume f is a function of n real parameters $\theta_1, ..., \theta_n$.

$$\text{Diff}(f(\vec{\boldsymbol{\theta}})) \coloneqq \nabla f = [\frac{\partial f}{\partial \theta_1}, ..., \frac{\partial f}{\partial \theta_n}] \tag{1.1}$$

Black Box Approach. Finite difference (FD) is the general approach for differentiating a math function without access to the computation process. It simply evaluates the program with a slight offset to the input parameter that we wish to differentiate with respect to.

While being general and easy to compute, it has two major disadvantages. Firstly, FD scales linearly with the number of input parameters. If we wish to differentiate with respect to n parameters, FD requires the black box function to be evaluated $O(n)$ times. This may introduce severe overhead for large n, such as a neural network with millions of parameters. Additionally, there is a dilemma in choosing the amount of the offset (step size) applied to the input parameter. A larger step size oversmoothes the original function and leads to inaccurate gradient approximation, whereas a smaller step size may also amplify noise and introduce numerical instabilities due to floating point precision. As a result, choosing a suitable step size that trades off both concerns can be challenging, and usually requires several steps of trial and error.

Manual Solution. Assuming the function is continuously differentiable, its analytic gradient can be derived using the chain rule and basic calculus. While theoretically correct, manually deriving the solution for hundreds or thousands of lines of a program would be tedious and time-consuming, and is error-prone both during the math derivation as well as coding.

Automatic Framework. Automatic Differentiation (AD) [12] is a popular framework that recursively applies calculus rules to automatically generate a symbolic gradient for an arbitrary program. When computed in reverse mode, which in machine learning contexts is often referred to as backpropagation, the runtime for the AD-generated gradient is agnostic to the number of parameters n, therefore having $O(1)$ time complexity with respect to those parameters. Because of being efficient and automatic, AD is widely used in machine learning applications [13], and is integrated into many libraries, such as JAX [44], TensorFlow [1], and PyTorch [103].

Similar to the AD example, many other program transformation tasks could draw inspiration from their manual solutions as well to develop an automatic compiler framework.

However, straightforwardly generalizing the manual expertise may still be challenging, which will be detailed in the next section.

1.2 Challenges for Automatic Compiler Frameworks

Generally, building an automatic compiler framework requires a set of rules to modify the compute graph of the program that is scalable for arbitrarily complicated programs. For example, the Automatic Differentiation (AD) framework discussed in Section 1.1 utilizes a set of recursive rules generalized from its manual solution: the chain rule in calculus.

Unfortunately, such rules cannot always be easily generalized from manual solutions. This could be because the manual solution heavily relies on the expert's domain expertise and experience, such as how to tile a loop for better memory locality [111]. Therefore, they cannot be easily abstracted into compiler rules. Additionally, for math transformations like integration, simple closed-form solutions may not exist at all for the majority of functions, hence the manual solution may not be applicable even with the presence of an expert.

As a result, a major challenge in developing an automatic compiler framework is to design compiler rules that are more general and have similar performance to the manual solutions. For example, if we can enumerate the space of possible program transformations, we could use a search strategy to find the transformation that minimizes a loss proxy. The loss usually characterizes the goal, such as runtime or memory footprint. As an alternative, approximation rules can also be designed based on the manual solution to trade off scalability with accuracy.

The remainder of this section discusses three program transformation examples where state-of-the-art methods fail to generalize to arbitrary programs. Similar to Section 1.1, we will first present the black box and manual solution for each, followed by a discussion of the challenges of the automatic framework.

Figure 1.2: Target image for Program 1.1. What setting for the parameters (center and radius) best matches this target? An optimization process that searches for these parameters would benefit from an accurate gradient, but conventional AD methods fail in this case.

1.2.1 Automatic Differentiation at Discontinuities

In Section 1.1, we describe automatic differentiation (AD), which is a powerful compiler framework to generate the gradient program for continuously differentiable programs. However, conventional AD methods ignore discontinuities, which are essential in various applications such as visibility tests in rendering, or change of state in physics simulations. Informally, program discontinuities are sudden changes to program outputs caused by infinitesimal changes to program inputs. For example, the if / else branch could cause a discontinuity. More formally, we can consider the program as a math function of all of its real arguments and define continuity in the usual manner from real analysis [139].

The gradient generated by traditional AD usually struggles to optimize parameters controlling the discontinuities. For example, Program 1.1 demonstrates a program that draws a circle. It uses an if / else branch to test whether a certain pixel is inside or outside the circle, then draws the color accordingly. However, if we want to optimize the radius and circle position to best match the target image in Figure 1.2, using the gradient from the AD framework will not work, because its gradients with respect to all three parameters (cx, cy, r) are always zero. Therefore, we need new methods other than the traditional chain rule to correctly differentiate programs with discontinuities.

Program 1.1: A discontinuous program that draws a circle.

```
def circle(cx, cy, r):
    u, v = get_pixel_coord()
    if ((u - cx) ** 2 + (v - cy) ** 2) ** 0.5 < r:
        out = 1
    else:
        out = 0
    return out
```

Mathematically, the jump discontinuity generated by the if /else branching can be characterized as the Heaviside step function, whose gradient is the Dirac delta distribution. Informally in the engineering context, the Dirac delta evaluates to infinity at the discontinuity, and is 0 everywhere else. It is obvious that the Dirac delta is merely an analytic notation and cannot be directly evaluated using a program: it evaluates to infinity at a measure zero discontinuity. Therefore, in engineering, a general approach is to convolve the gradient with a smoothing kernel ϕ to ensure that the gradient at the discontinuity can be sampled as a finite value with nonzero probability. Further, we can typically commute the differentiation and integration operator. [1] In rendering this process is also called pre-filtering. We formally characterize the process of differentiating the pre-filtered program f in Equation 1.2.

$$\text{Diff}_D(f;\phi) \coloneqq \nabla(f * \phi) = [\frac{\partial}{\partial\theta_1}(f * \phi), ..., \frac{\partial}{\partial\theta_n}(f * \phi)] \tag{1.2}$$

Black Box Approach. Finite difference (FD) is still able to account for discontinuities because it relies on multiple evaluation sites, therefore naturally capturing the sudden change in function value. However, as we have discussed in Section 1.1, FD is generally not preferred because of its inefficiency ($O(n)$ runtime) and difficulty in choosing a proper step size. The

[1] In general, commuting is not allowed by the Leibniz integral rule because the integrand is discontinuous. However, in the engineering context, commuting is usually valid when ∇f can be expressed in terms of the Dirac delta functions. Further, we will prove in Section 2.8.4.1 that commuting is valid for the set of programs this dissertation focuses on.

latter becomes especially challenging with the presence of discontinuities: a smaller step size may fail to sample the discontinuities, while a larger step size overly smooths the gradient, or even worse, it may have crossed multiple discontinuities, generating a hard-to-interpret and inaccurate result.

Manual Solution. There are generally two approaches for math experts to manually derive the gradient of the discontinuous program f. Firstly, they could utilize various Dirac delta related properties to analytically convert ∇f into a closed-form expression with the Dirac delta notation [69, 71]. Alternatively, they can cast the problem into an integral over the discontinuous program f. Examples include pre-filtering as in Equation 1.2, or integrating over a 3D hemisphere for physics-based rendering, or integrating over the trajectory path for physics simulation. After that, many math techniques such as the Leibniz integral rule, the Reynold transport theorem, or reparameterization can be applied to remove the discontinuity out of the differentiation operator [76, 156]. However, deriving the gradient using either approach would require a math expert to make program-specific decisions such as which theorems to apply to which part of the equation. This heavily relies on the experts' imagination and domain expertise, and there is no guarantee a good manual solution can be easily found for any arbitrary program.

Automatic Framework. Because the manual solutions require case-by-case decisions, it is not feasible to directly generalize math rules for the compiler as in the traditional AD framework. A naive approach may directly apply the chain rule in AD with the additional notation of the Dirac delta. However, the chain rule easily fails even under simple counterexamples (Section 2.4.2.2). Fortunately, we can summarize scenarios when the chain rule fails and design novel counterparts of the calculus rules that correctly account for the discontinuities. In Chapter 2 we will discuss a framework that allows efficient reverse-mode AD on discontinuous programs.

1.2.2 Automatic Convolution

In Equation 1.2, we introduce the convolution notation but merely use it as an intermediate representation that helps to conveniently evaluate the gradient of discontinuous programs: the convolution does not have to be directly evaluated. Nevertheless, convolution itself is of interest to many applications. Specifically, if ϕ is a known smoothing kernel (such as the Gaussian or boxcar function), its convolution with an arbitrary program f removes sharp or high-frequency textures from f. This could be beneficial for rendering tasks such as antialiasing, or robotics tasks like path smoothing. We formally characterize this process in Equation 1.3.

$$\text{Conv}(f;\phi) \coloneqq f * g = \int_{\mathrm{R}^d} f(\vec{\boldsymbol{\theta}_0} - \vec{\boldsymbol{\theta}})\phi(\vec{\boldsymbol{\theta}})d\vec{\boldsymbol{\theta}} \tag{1.3}$$

Black Box Approach. Without access to the program representation, convolution can still be estimated through numerical integration methods such as Monte Carlo sampling [110]. To evaluate the convolution integral at $\vec{\boldsymbol{\theta}_0}$, it samples $\vec{\boldsymbol{\theta}}$ from the distribution of ϕ and averages the evaluations of $f(\vec{\boldsymbol{\theta}_0} - \vec{\boldsymbol{\theta}})$. In rendering, this is also known as supersampling. However, the sampling approach suffers a tradeoff between accuracy and efficiency: it is noisy at low sample counts, but the runtime scales linearly as the number of samples grows, leading to expensive computation for highly accurate estimates.

Manual Solution. In a few cases, Equation 1.3 may also have closed-form expressions for direct evaluation. However, this requires an expert to manually apply various calculus techniques, such as decomposing the program f into elementary functions, or applying integration by parts. Similar to differentiating a discontinuous program in Section 1.2.1, no general rule has been established for converting the convolution integral to a closed-form expression, resulting in a tedious trial-and-error derivation process. Further, the closed-form

expression may not exist for the majority of programs f, limiting the applicability of manual derivations.

Automatic Framework. Similar to Section 1.2.1, it is difficult to directly generalize a set of compiler rules from the manual solutions. Nevertheless, we will make a connection between the probability theory and the convolution in Equation 1.4, and point out possible approximations it may lead to. Because we assume ϕ in Equation 1.3 is a known non-negative kernel, without loss of generality, we can view ϕ as a probability density function (pdf) because it can always be decomposed into the multiplication of a pdf and a constant scale factor. Equation 1.4 further expands Equation 1.3 and states that when evaluating the convolution integral on $\vec{\boldsymbol{\theta}_0}$, the result can be viewed as the expected value for a random variable $\mathbf{Y} = f(\boldsymbol{\Theta})$, where $\boldsymbol{\Theta}$ is a random variable as well whose probability density is conditioned on $\vec{\boldsymbol{\theta}_0}$: $\text{pdf}_{\boldsymbol{\Theta}}(\vec{\boldsymbol{\theta}}) = \phi(\vec{\boldsymbol{\theta}_0} - \vec{\boldsymbol{\theta}})$. Now that both the input and output of f become random variables $\boldsymbol{\Theta}$ and $\mathbf{Y}$, a natural extension is to view *every* intermediate computation within the program f as a random variable as well. Further, if the kernel ϕ has some nice properties such as being unimodal, we may be able to approximate the intermediate and output random variables simply using their first and second-order statistics (i.e. mean and variance). In Chapter 5, we will present a compiler framework that automatically approximates the convolution integral based on the random variable observation.

$$
\begin{aligned}
\text{Conv}(f; g)(\vec{\boldsymbol{\theta}_0}) &= \int_{\text{R}^d} f(\vec{\boldsymbol{\theta}_0} - \vec{\boldsymbol{\theta}})\phi(\vec{\boldsymbol{\theta}})d\vec{\boldsymbol{\theta}} \\
&= \int_{\text{R}^d} f(\vec{\boldsymbol{\theta}})\phi(\vec{\boldsymbol{\theta}_0} - \vec{\boldsymbol{\theta}})d\vec{\boldsymbol{\theta}} \\
&= \int_{\text{R}^d} f(\vec{\boldsymbol{\theta}})\text{pdf}_{\boldsymbol{\Theta}}(\vec{\boldsymbol{\theta}}; \vec{\boldsymbol{\theta}_0})d\vec{\boldsymbol{\theta}} \\
&= \text{E}[f(\boldsymbol{\Theta})] \\
\text{pdf}_{\boldsymbol{\Theta}}(\vec{\boldsymbol{\theta}}) &= \phi(\vec{\boldsymbol{\theta}_0} - \vec{\boldsymbol{\theta}})
\end{aligned}
\tag{1.4}
$$

1.2.3 Identifying Input Features for Deep Learning

Sections 1.2.1 and 1.2.2 discuss explicit program transformations for the differentiation and convolution tasks: the output program g is constructed by deterministically applying a set of rules to modify the computation process of input program f. As an alternative, a task can be learned by deep learning methods: using a black box neural network as a proxy for the output program g. While most network model parameters are learned by gradient descent, some manual choices have to be made before training, such as network architectures, loss functions, and what input features to feed to the model. Specifically, the input features are often picked to be correlated with an input data generation process f. If f is not a program, but rather a sampling process from the natural images, the input features could simply be the output of f, which samples a natural image from the training set. In other cases where all or part of f can be programmatically described, researchers have explored designing input features based on the computation of f to benefit the learning process. This section discusses techniques that identify these beneficial input features, as well as how they can be automated.

Black Box Approach. As discussed, in this case, the input features can simply be the output of f, such as the RGB image when visual data is sampled by f [57, 47, 104].

Manual Solution. If the data generation process f is known, machine learning researchers could manually design additional input features. Because the output of f is almost always a part of the input features to the learning model, we describe any additional hand-crafted features as *auxiliary input features*. For example, when learning shading or denoising from a traditional rendering pipeline f, features from the G-buffer or the Z-buffer, such as surface normal, albedo, and specularity could be used as auxiliary input features to improve learning performance [92, 132, 21]. Similarly, when learning fluid super-resolution from a low-resolution fluid simulation f [146], the fluid velocity and vorticity computed within f

could be helpful as auxiliary features. Further, the input features could be constructed from additional computations that are not part of the data generation process f, such as the positional encoding in NeRF [82] based methods.

Automatic Framework. The manually designed features vary greatly among applications and domains, and cannot be generalized to arbitrary programs. Nevertheless, we can observe that it is easier to generate a superset of beneficial auxiliary input features. One such example is the program trace of f. In this dissertation we refer to the trace as all the intermediate values computed from executing f. The program trace inherently includes many auxiliary features manually picked by experts in specific applications, such as G and Z buffers for rendering or the internal states for fluid simulation. Even when the auxiliary features are crafted by a separate computation, such as the positional encoding, the program trace may already include similar high-frequency features that could be equivalently beneficial for learning. Because the program trace is a superset of potentially beneficial auxiliary features, it also contains redundant features that are not needed for learning. However, we could defer the decision of picking beneficial features to the learning model, therefore automating the entire pipeline. Chapter 6 describes a framework that explores the idea of using the program trace as auxiliary features and shows its applicability on a variety of learning tasks.

1.3 Contribution

The contributions of this dissertation include three novel compiler tools for automatic program transformation to arbitrary programs, and their applications in the domain of visual and audio programs. First, we present an automatic differentiation framework to differentiate discontinuities and apply it to both shader and audio applications. Second, we develop an automatic convolution framework to approximate the program's convolution with a Gaussian kernel. Third, we extend the compiler framework to deep-learning tasks to help learners explore the additional information from the program trace.

Automatic Differentiation for Discontinuous Programs. Chapter 2 presents a general math framework Aδ that allows reverse-mode AD to discontinuous programs.

Optimizing Discontinuous Shader Applications. We demonstrate that the Aδ framework helps differentiate procedural shader programs where discontinuities are vital for determining object silhouette or visibility. Specifically, in Chapter 3, we optimize unknown parameters of the shader program to match a target image. This allows users to interactively modify and animate the shader, which would otherwise be cumbersome in other representations such as triangle meshes or vector graphics.

Optimizing Discontinuous Audio Synth Applications. We give another example where differentiating discontinuities is important for the optimization task. In Chapter 4, we further extend the Aδ framework to differentiate audio synth programs to match a target sound signal, such as musical instruments or cartoon effects.

Approximate Program Smoothing. Besides differentiation, we also develop a compiler framework for automatic convolution. Chapter 5 presents a compiler framework to approximate the program's convolution with a Gaussian kernel, therefore achieving program smoothing. We demonstrate its applicability on antialiasing procedural shader programs.

Learning from Program Trace. We further demonstrate the program representation can help deep learning approaches as well. Chapter 6 develops a framework that allows learning tasks to automatically collect and learn from the program trace, and investigate its effectiveness under a variety of applications that includes both traditional visual tasks as well as simulation tasks.

The research described in this dissertation has appeared in the following publications:

- Chapters 2 and 3: Aδ: Autodiff for Discontinuous Programs - Applied to Shaders [149]. `https://pixl.cs.princeton.edu/pubs/Yang_2022_AAF/index.php`

- Chapter 4: White-Box Automatic Synthesizer Programming (under review) [151].
- Chapter 5: Approximate Program Smoothing Using Mean-Variance Statistics, with Application to Procedural Shader Bandlimiting [148].https://yyuting.github.io/docs/eg_2018.html
- Chapter 6: Learning from Shader Program Traces [150]. https://pixl.cs.princeton.edu/pubs/Yang_2022_LFS/index.php

Project details can also be found in Yuting Yang's personal website. https://yyuting.github.io/

Chapter 2

Automatic Differentiation for Discontinuous Programs

Over the last decade, automatic differentiation (AD) has profoundly impacted graphics and vision applications — both broadly via deep learning and specifically for inverse rendering. However, traditional AD methods always ignore gradients at discontinuities, instead treating functions as continuous. This raises many challenges for various applications. For example, rendering algorithms intrinsically rely on discontinuities as they are crucial at object silhouettes. And in general, discontinuities are vital 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 chapter instead describes a compiler-based approach to extend reverse mode AD 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 pre-filtered gradient over a box kernel oriented along a 1D sampling axis. We further propose two methods to validate our gradient approximation. We establish a theoretical error bound for a relatively broad

Figure 2.1: Our framework takes as input an arbitrary program expressed by our DSL (§ 2.4.1), and approximates the gradient by pre-filtering a 1D box kernel along sampling axes (§ 2.4.2). Approximations along multiple sampling axes are later combined (§ 2.5.1). Finally, we formally prove our gradient approximation is first-order correct in § 2.7 and 2.8, and we also design a quantitative error metric (§ 2.9) to evaluate any gradient program empirically.

class of programs, and also propose a quantitative error metric to numerically evaluate the approximation error for any program.

2.1 Overview

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 expressed 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, we propose a framework and compiler we call Aδ for automatic differentiation of

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

programs – including proper differentiation of the discontinuities. 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 2.4.1.)

Graphics researchers have derived several application-specific solutions for differentiating Dirac parameters, targeting specialized domains such as spline shapes for vector graphics [71] or triangle meshes rendered in a path tracer [9, 69, 76]. While they address these specific domains, they are not readily adapted to arbitrary functions. TEG [10] 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 functions that are either not invertible or for which the programmer cannot easily write down the inverse.

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, A2 also becomes more applicable for higher sample frequencies because 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 2.3, A2 also allows efficient reverse-mode AD, because gradients of discontinuities with respect to the sampling axis can be easily approximated by finite differences, which can be further transformed into the gradient with respect to other parameters using properties of Dirac delta functions. 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.

Figure 2.1 shows an overview of our framework. This chapter primarily contributes 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. We also propose a novel error metric to quantitatively evaluate the gradient approximation on any program.

2.2 Related Work

Automatic differentiation of parametric discontinuities. TEG [10] 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 the trapezoidal rule. However, the set of programs TEG can correctly handle is restricted. To correctly eliminate Dirac deltas, the discontinuities are limited to be rep-

Table 2.1: Comparison between ours and related work on differentiating discontinuous programs: traditional Auto-Differentiation (AD); finite difference (FD); TEG [10]; differentiable vector graphics (DVG) [71] and differentiable path tracers (DPT) [9, 76, 69]. 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 is 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 2.4.1); 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

resented 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, Chapter 3 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 2.1 compares our method with TEG and other baselines such as traditional AD ([90, 96]) 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 [71], path tracers [157, 69, 9, 76], and other physics-based renderers [159], specialized rules are derived analytically, and algorithms are also designed for efficiently carrying out the computation of these specialized gradients [154, 95, 155]. 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 prefiltering, these methods only differentiate the proxy, and do not attempt to sample discontinuities directly. Soft rasterizer [73] 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 [148], 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 [130], using neural textures or neural implicit 3D representation for differentiable rendering [128, 102, 94, 123, 59], and using NeRF as a surrogate for geometry and reflectance [82, 79]. These representations are inherently differentiable, and leverage all of the recent progress in neural representations, but the resulting representation is a network that is difficult to interpret and manipulate in general ways. In contrast, our approach provides gradients in the original program representation (and does so quickly relative to the training of most neural methods).

2.3 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 θ 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 δ, 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 a smoothing 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: [2]

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

Here we use ϕ to represent the probability density function (pdf) 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 θ, we can differentiate with respect to (wrt) θ 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, or low-dimensional subspaces involving e.g. linear combinations of arbitrary axes. 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 dimensions 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.

[2] We will show in Section 2.8.4.1 that the swapping of differentiation and integration operators is valid for the set of programs investigated in this thesis – this set is described in Section 2.7.2.

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

$$\begin{aligned}\frac{\partial}{\partial\theta}\int H(g(x',\theta))\phi(x-x')dx' &= \int \delta(g(x',\theta))\frac{\partial g}{\partial\theta}\phi(x-x')dx' \\ &= \int \frac{\delta(x'-x_d)\frac{\partial g}{\partial\theta}}{|\frac{dg}{dx}|}\phi(x-x')dx' \\ &= \frac{\frac{\partial g}{\partial\theta}}{|\frac{dg}{dx}|}\,|_{x_d}\,\phi(x-x_d)\end{aligned} \tag{2.1}$$

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 [10], or has to use recursive algorithms to find the exact location of x_d [71]. Unlike previous work, because a box kernel ϕ is piece-wise constant, we can simplify computing $\phi(x-x_d)$ into sampling whether $x - x_d \in [-\epsilon, \epsilon]$.

2.4 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 this section, we extend our DSL to include a ternary `if` or `select` function in Section 2.5.2. In Chapters 3 and 4, we additionally extend the DSL to include application-specific operators: a ray-marching construct for shader programs, and a discrete choice construct for audio synth programs. After presenting the minimal DSL, we will present our gradient rules which can be used to extend typical reverse-mode AD. Finally, we use a toy example to demonstrate the applicability of differentiating discontinuities.

2.4.1 Our Minimal DSL Syntax

We formally define the set of programs expressible in our minimal DSL using the 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, θ 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 θ that expressions of the form of $H(e_d)$ statically depend upon.

2.4.2 Our Gradient Rules

This section formally defines our pre-filtering process, and presents novel gradient rules that approximate the derivatives of the pre-filtered function.

We define a function $f : \text{dom}(f) \to \mathbb{R}$ 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.

Additionally, for evaluation purposes, we define computational singularity, which refers to the function value being undefined for any intermediate node of f. It is formally defined by ruling out every possible case in our DSL syntax that may lead to undefined numbers over the reals.

Definition 1 *A function $f \in e_d$ is computationally singular at $(x, \vec{\boldsymbol{\theta}})$ if any of its intermediate values g satisfies one or more of:*

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

Based on Definition 1, we define $\text{dom}(f)$ to be the set $\{(x, \vec{\boldsymbol{\theta}}) \in \mathbb{R}^{n+1} : f \text{ is not computationally singular at } (x, \vec{\boldsymbol{\theta}})\}$. Note that our framework and implementation also support multidimensional 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 2.5.

Our gradient rules approximate the gradient of the pre-filtered function $\hat{f}$, which is the convolution of f with a box kernel along the sampling axis x.

$$\hat{f}(x, \vec{\boldsymbol{\theta}}; \epsilon) = \frac{1}{(\alpha + \beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} f(x', \vec{\boldsymbol{\theta}}) dx' \tag{2.2}$$

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

Note we only choose the first argument x as the sampling axis for simplicity in our presentation. In general, f could be a function with only the n parameters $\vec{\boldsymbol{\theta}}$, and the sampling axis could be *any* dimension where the sample is placed along, either dependent or independent to the n parameters. But we omit this and explicitly set x for brevity and lucidity.

Table 2.2: 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
$\frac{\partial_k (g+h)}{\partial\theta}$	$\frac{\partial_k g}{\partial\theta} + \frac{\partial_k h}{\partial\theta}$	$\frac{\partial_k g}{\partial\theta} + \frac{\partial_k h}{\partial\theta}$
$\frac{\partial_k (g\cdot h)}{\partial\theta}$	$\frac{1}{2}(h^+ + h^-)\frac{\partial_k g}{\partial\theta} + \frac{1}{2}(g^+ + g^-)\frac{\partial_k h}{\partial\theta}$	$h\frac{\partial_k g}{\partial\theta} + g\frac{\partial_k h}{\partial\theta}$
$\frac{\partial_k h(g)}{\partial\theta}$	$\begin{cases} h'\frac{\partial_k g}{\partial\theta} & \text{if } h(g) \text{ is statically differentiable} \\ \frac{h(g^+)-h(g^-)}{g^+-g^-}\frac{\partial_k g}{\partial\theta} & \text{otherwise}^* \end{cases}$	$h'\frac{\partial_k g}{\partial\theta}$

In our approximation, we locate discontinuities by placing two samples at each end of the kernel support. **Specifically, we denote x^+, x^- as the two ends of the kernel support, and $(\cdot)^+, (\cdot)^-$ as evaluating an expression $(\cdot)$ at each end of the kernel support respectively.** When ϵ is small enough, $(\cdot)^+$ and $(\cdot)^-$ can be viewed as approximating the right and left limits of an expression $(\cdot)$.

Both ours and AD approximate the derivative of functions, with differentiation rules summarized in Table 2.2. We further denote gradient approximations as ∂_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. Section 2.5.2.2 discusses an equivalent but more efficient gradient rule for Boolean operators. Forward mode AD can be carried out by applying Table 2.2 directly. Reverse-mode AD derivative rules can be obtained by replacing in Table 2.2 θ 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.

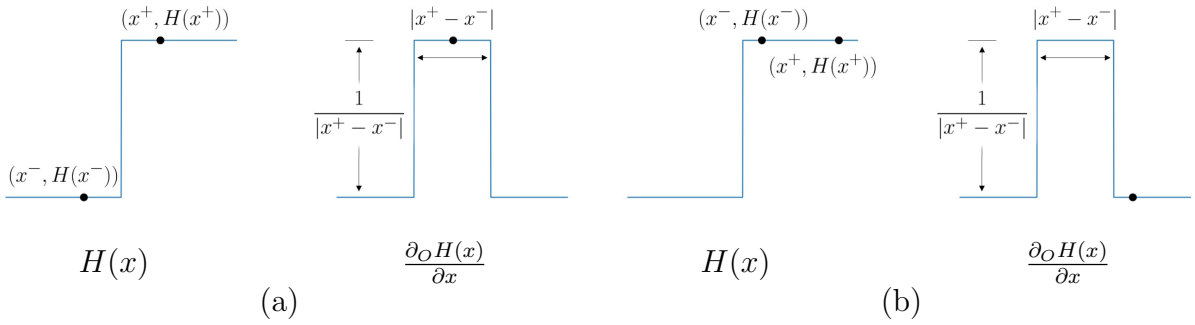


Figure 2.2: Illustration of computing $\frac{\partial_O H(x)}{\partial x}$ using our gradient rule. (a) The discontinuity of $H(x)$ is sampled between x^+ and x^-, therefore the corresponding gradient approximation is nonzero. (b) The discontinuity of $H(x)$ is not sampled, therefore the corresponding gradient approximation is zero.

We now motivate and discuss the gradient rules where ours are different from traditional AD.

2.4.2.1 Heaviside step

We first describe the simplest scenario: differentiating $H(x)$ wrt x using our gradient rule. Figure 2.2 illustrates the computation process. If we use $\theta = x$, $g = x$ in our step function gradient rule in Table 2.2, the approximation result has a piecewise constant box shape as in the right plot of Figure 2.2 (a)(b). Note this box shape also corresponds to the box kernel we use to obtain the pre-filtered function $\hat{f}$ in Equation 2.2. When the two samples x^+ and x^- evaluate to different sides of the discontinuity as in Figure 2.2 (a), the condition $H(x^+) \neq H(x^-)$ is met, therefore our gradient approximation evaluates to the nonzero value $1/|x^+ - x^-|$. Conversely, when x^+ and x^- evaluate to the same side of the discontinuity as in Figure 2.2(b), our gradient approximation evaluates to zero.

We further motivate differentiating the more generic $H(g(x, \theta))$ from Equation 2.1 assuming ϕ is a 1D box kernel with $\alpha = \beta = 1$. Equation 2.1 can be represented using the pre-filtering notation defined in Equation 2.2.

$$
\begin{aligned}
f =& H(g(x,\theta)) \\
\frac{\partial \hat{f}(x,\theta;\epsilon)}{\partial \theta} =& \frac{\frac{\partial g}{\partial \theta}}{|\frac{dg}{dx}|} \mid_{x_d} \phi(x - x_d)
\end{aligned}
\tag{2.3}
$$

Additionally, we can define the 1D box kernel $\phi(x)$ as the following and insert back to Equation 2.3.

$$
\phi(x) = \begin{cases} \frac{1}{2\epsilon} & \text{if } x \in [-\epsilon, \epsilon] \\ 0 & \text{otherwise} \end{cases}
\tag{2.4}
$$

$$
\frac{\partial \hat{f}(x,\theta;\epsilon)}{\partial \theta} = \begin{cases} \frac{1}{2\epsilon} \frac{\frac{\partial g}{\partial \theta}}{|\frac{dg}{dx}|} \mid_{x_d} & \text{if } x_d \in [x - \epsilon, x + \epsilon] \\ 0 & \text{otherwise} \end{cases}
\tag{2.5}
$$

Evaluating Equation 2.5 requires exactly computing the discontinuity x_d, which could be expensive for complicated g. We therefore apply a series of derivations and finally demonstrate the rule in Table 2.2 is a good approximation to Equation 2.5.

We start with the branching condiiton $x_d \in [x - \epsilon, x + \epsilon]$. Because the discontinuity x_d implies $g(x_d, \theta) = 0$, and because we assume ϵ is small enough such that at most one discontinuity exists within the interval $[x-\epsilon, x+\epsilon]$, the branching condition $x_d \in [x-\epsilon, x+\epsilon]$ is equivalent to whether the Heaviside step function $H(g(x,\theta))$ has flipped state between the two ends of the kernel support:

$$
x_d \in [x - \epsilon, x + \epsilon] \equiv H(g^-) \neq H(g^+)
\tag{2.6}
$$

We next discuss evaluating $\partial g / \partial \theta$ at x_d. Because Equation 2.5 is only nonzero for $x_d \in [x - \epsilon, x + \epsilon]$, we only discuss this case. Since Equation 2.1 assumes g is a continuous

function, it is therefore locally Lipschitz continuous (formally proved in Section 2.8). As a result, evaluating g and its derivatives on a neighboring location to x_d always has an error bounded by first-order terms:

$$\begin{aligned}\frac{1}{2\epsilon}\frac{\partial g}{\partial \theta}\mid_{x_d} =& \frac{1}{2\epsilon}\frac{\partial g}{\partial \theta}\mid_{x} + O(x_d - x) \\ =& \frac{1}{2\epsilon}\frac{\partial g}{\partial \theta}\mid_{x} + O(\epsilon) \\ & \text{Because } x_d \in [x-\epsilon, x+\epsilon]\end{aligned} \tag{2.7}$$

Finally, we approximate dg/dx that evaluates at x_d using finite difference. This avoids an extra back-propagation pass to analytically computing dg/dx. Because g^+ and 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. Similarly because of locally Lipschitz continuity, we can bound the approximation with first-order tersm.

$$\frac{dg}{dx} = \frac{g^+ - g^-}{2\epsilon} + O(\epsilon) \tag{2.8}$$

Inserting Equation 2.6 - 2.8 to Equation 2.5 we get the following:

$$\begin{aligned}\frac{\partial \hat{f}(x,\theta;\epsilon)}{\partial \theta} &= \begin{cases} \frac{1}{2\epsilon}\frac{\frac{\partial g}{\partial \theta} + O(\epsilon)}{|\frac{g^+ - g^-}{2\epsilon}| + O(\epsilon)} & \text{if } H(g^-) \neq H(g^+) \\ 0 & \text{otherwise} \end{cases} \\ &= \begin{cases} \frac{\frac{\partial g}{\partial \theta}}{|g^+ - g^-|} + O(\epsilon) & \text{if } H(g^-) \neq H(g^+) \\ 0 & \text{otherwise} \end{cases} \\ & \text{Assuming } g^+ - g^- \neq 0\end{aligned} \tag{2.9}$$

Comparing Equation 2.9 with the $H(g)$ rule in Table 2.2 we conclude that our proposed gradient rule is a first-order approximation to the motivating example in Equation 2.1.

2.4.2.2 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, or 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 may 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 2.3. 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 2.1 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 use our rule to differentiate the Heaviside step function and only use AD's 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 evaluates to 0 or 1, $\frac{\partial_{AD} f}{\partial \theta}$ is always 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 [10] 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.

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

2.4.2.3 Function composition

Our derivative rule applies to atomic continuous functions h to differentiate $h \circ g$. 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 in Section 2.4.2.2 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 similar 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.

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

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.

2.5 Compiler Details

We implement a compiler by extending the reverse-mode automatic differentiation to discontinuous programs by replacing the original gradient rules from calculus with our gradient rules from Table 2.2. As mentioned in Section 2.4.2, our compiler supports functions with multi-dimensional 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 backward 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 within each kernel. Specifically, we average the gradient between $U[-\Delta x, 0]$ and $U[0, \Delta x]$, where Δx 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 that results in a periodic color change with alternating color for each pixel, our compiler can still correctly differentiate

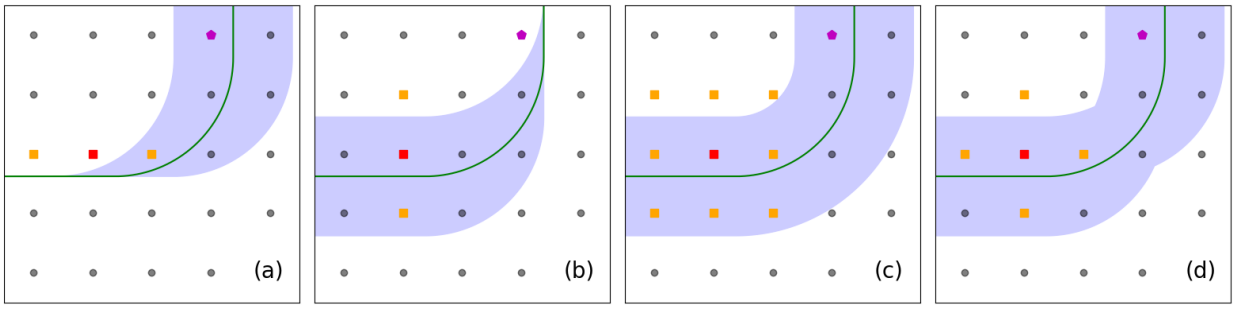


Figure 2.3: Visualizing different options for how to combine multiple sampling axes in 2D. The green line demonstrates a discontinuity, and the blue region indicates evaluation locations where discontinuity can be sampled. Naively choosing either the x (a) or the y axis (b) can result in the discontinuity 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 the 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 that grows exponentially with the number of axes. Our implementation (d) adaptively chooses from available axes, and ensures discontinuities in any orientation can be sampled with nonzero probability.

the program for the the single pixel that has different discontinuities on both sides. 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. We collect the $H(g)$ nodes within the compute graph where g is statically continuous. If more discontinuity is sampled for than one such nodes, the compiler nullifies the contribution from that location by outputting zero gradient.

2.5.1 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 2.4.2 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 2.3(a)(b)). Another approach is to use a multi-dimensional prefiltering kernel (Figure 2.3(c)). However, due to the sifting property of the Dirac delta, integrating against a Dirac delta in n-dimensions with $n > 1$ typically results in an $n - 1$ dimensional integration over the set where the Dirac delta's argument is zero, which can be challenging but can be handled by additional sampling [10] or by recursively finding the intersection between discontinuity and the kernel support [71].

In our implementation, for simplicity, we instead use a separate 1D kernel for each sampling axis and combine gradient approximations from different sampling axes afterward. 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 2.3(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 2.1, a larger value leads to approximate gradients with smaller magnitude, therefore smaller variance. For m sampling axes, our compiler draws $2m + 1$ samples, which can be potentially shared between neighboring locations.

2.5.2 Efficient Ternary Select Operator

Section 2.4.2 discusses that 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 memory usage (although asymptotically the same) in the backward pass, which may lead to performance issues such as register spilling.

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 2.4.1. 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 similar storage to the AD rule, while allowing the first-order correctness property to stay the same as claimed in Section 2.7.

2.5.2.1 General Ternary Select Operator

This section discusses branching with inequality conditions that can be written in the form

$$\begin{aligned} F(p, l, r) =& \text{select}(p \geq 0, l, r) \\ =& r + (l - r) \cdot H(p) \end{aligned} \tag{2.10a}$$

Branching with complex Boolean expressions will be discussed next in Section 2.5.2.2. The rule developed in this section can also be used in all inequality and equality comparisons: $p \geq q$ is equivalent to $p - q \geq 0$, $p \leq 0$ can be rewritten as $-p \geq 0$, and $\text{select}(p > 0, l, r)$ is equivalent to $\text{select}(-p \geq 0, r, l)$. 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 2.5.2.2.

Instead of directly applying the multiplication rule in Table 2.2, we design a specialized rule for branching that uses similar storage as if the gradient is approximated by AD. The specialized rule utilizes the fact that our evaluation location always coincides with one of the endpoints of our pre-filtering kernels. That is, we either have $x = x^+$ for the pre-filtering kernel $U[-\Delta x, 0]$, or $x = x^-$ for $U[0, \Delta x]$. For simplicity, when evaluating a function g at

two ends of the pre-filtering kernel, we always denote them as g and g_n, where $g = g(x)$, and g_n is evaluated at the neighboring location at the other end of the kernel support (i.e. g^- for $U[-\Delta x, 0]$ and g^+ for $U[0, \Delta x]$). The gradient rule for the efficient ternary select operator is shown in Equation 2.11.

$$\frac{\partial_O F}{\partial \theta_i} = (l_n - r_n)\frac{\partial_k H(p)}{\partial \theta_i} + \text{select}(p > 0, \frac{\partial_k l}{\partial \theta_i}, \frac{\partial_k r}{\partial \theta_i}) \tag{2.11}$$

An alternative and potentially more intuitive way to derive Equation 2.11 above is to directly differentiate the `select` definition in Equation 2.10a but using a simplified and biased multiplication rule given in Equation 2.12 below. We denote this biased multiplication gradient rule as δ_T because the reader can only use it to derive the ternary operator `select`, and is different from the multiplication rule we actually used in Table 2.2, which is denoted as ∂_O. Similar to Table 2.2, the gradients for g and h are denoted with ∂_k to emphasize that Equation 2.12 is agnostic to how these two terms are computed. Section 2.8.6.7 will show this new rule for the ternary `select` has identical correctness properties as differentiating F using the original multiplication rule.

$$\frac{\partial_T (g \cdot h)}{\partial \theta_i} = h\frac{\partial_k g}{\partial \theta_i} + g_n \frac{\partial_k h}{\partial \theta_i} \tag{2.12}$$

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, they are always differentiated using AD.

2.5.2.2 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: $C_i = c_i \geq 0, i = 1, ..., n$.

Similar to sampling floating point values at two ends of the filtering kernel, we can also sample Boolean values such as B^+ and B^-. Disagreeing Boolean samples indicate 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 2.11 with $\frac{\partial H(c_i)}{\partial \theta_i}$ for the first clause C_i where a discontinuity is sampled.

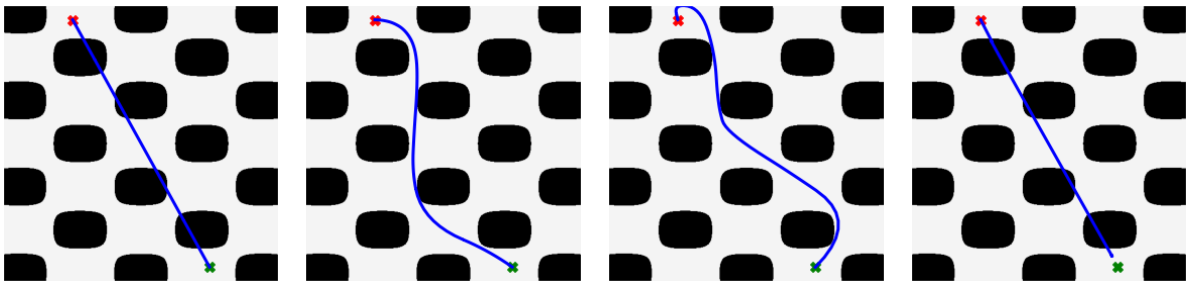
2.6 Toy Application: Path Planning

This section uses a path planning toy example to demonstrate the applicability of our differentiation framework. Chapters 3 and 4 further build two systems that apply our gradient rules to the domains of shader programs and audio synth programs respectively.

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.

We first initialize the 2D trajectory with zero velocity and a fixed start point at time 0. The motion of the 2D trajectory is modeled by 10 segments of piece-wise constant acceleration. Each constant segment is parameterized by its x and y components of the acceleration, as well as the duration of the segment. Therefore, the velocity and position can be represented as piece-wise linear and piece-wise quadratic forms respectively. An L2 loss L_{dist} 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 $L_{repulsion}$ can be further reparameterized



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

Figure 2.4: Path planning task: obstacles we wish to avoid are shaded as black regions, starting and desired ending positions 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 failing to avoid the discontinuous repulsion field. We experimented FD with five different step sizes, and report the one with the lowest error in FD*. We report relative runtime to ours for FD* and AD. While AD has similar runtime, FD* is significantly slower.

as an integral over time, and is approximated by quadrature using 10 samples per constant acceleration segment. We additionally add a third loss L_{fuel} that minimizes the integral of the magnitude of the acceleration over time, which is proportional to the total fuel consumed assuming constant specific impulse: an analogy to propellant in aerodynamics.

We report the optimized trajectory in Figure 2.4 and compare with two baselines: finite difference (FD) and automatic differentiation (AD). Because our gradient rules easily sample the discontinuous repulsion field, our trajectory both avoids all the obstacles, and is energy efficient with optimized L_{fuel} as well. FD, on the other hand, is much slower than ours (17x) and already struggles to find a good step size with this toy example. We run the experiment with FD for 5 different step sizes and only report the variant with the lowest error as FD* in Figure 2.4. While FD* uses a larger step size that allows discontinuity to be easily sampled, this step size is already too large to accurately differentiate the continuous losses, such as L_{fuel}, leaving the trajectory only sub-optimal: it makes unnecessary turns near the starting point. Finally, AD unsurprisingly fails to avoid the obstacles because it is unable to sample the discontinuity in the repulsion field.

2.7 Theoretical Error Bound Claim

This section establishes the theoretical guarantee to our gradient approximations. We first formally define the notion of *first-order correctness* of a gradient approximation. Then we characterize the subset of programs for which ours is first-order correct. Finally, we provide a proof sketch of our claims.

2.7.1 First-Order Correctness Definition

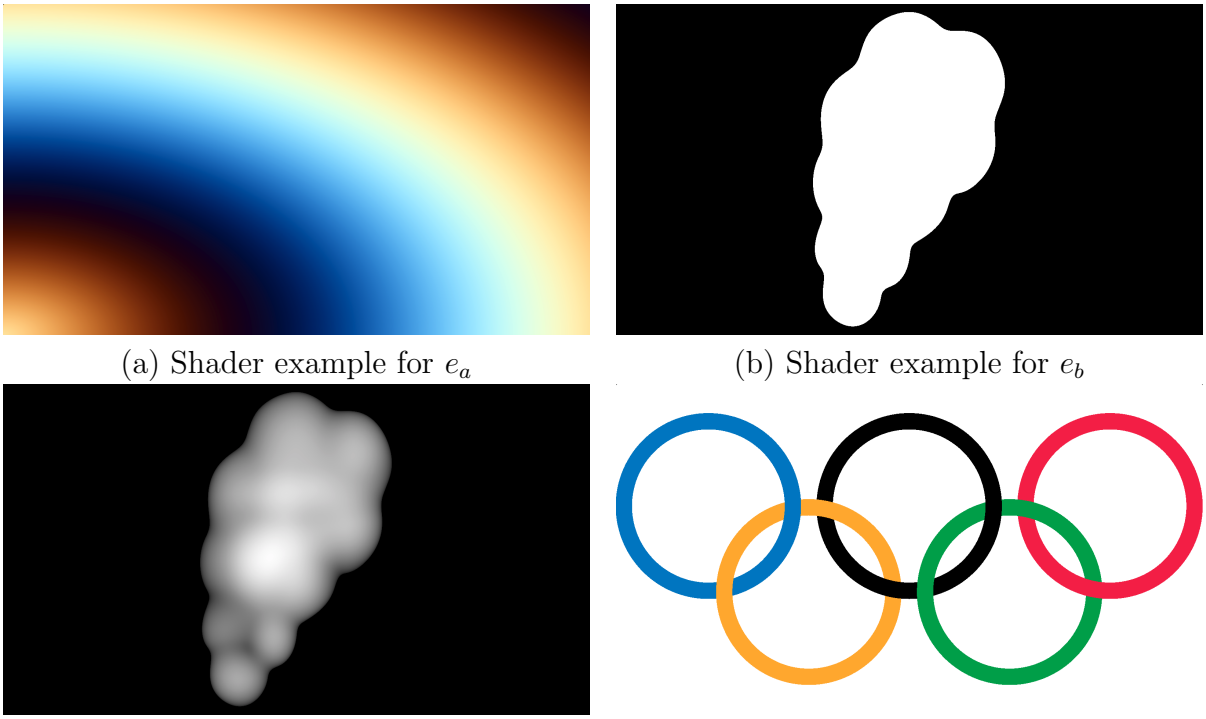
Typically, when approximating a finite valued reference, we characterize the error of an approximation method by it's absolute difference from the reference. However, the reference derivative for pre-filtered discontinuous functions could approach infinity as the kernel size ϵ goes to zero. Therefore we care more about whether the approximation approaches infinity asymptotically with the reference. Because the derivative for continuous and discontinuous functions have varying characteristics, 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 2 *A gradient approximation $\frac{\partial_k f}{\partial \theta_i}$ is absolutely first-order correct for parameter θ_i at $(x, \vec{\boldsymbol{\theta}})$ with kernel size ϵ if*

$$\frac{\partial_k f}{\partial \theta_i}(x, \vec{\boldsymbol{\theta}}; \epsilon) = \frac{\partial \hat{f}}{\partial \theta_i}(x, \vec{\boldsymbol{\theta}}; \epsilon) + O(\epsilon) \tag{2.13}$$

Definition 3 *A gradient approximation $\frac{\partial_k f}{\partial \theta_i}$ is relatively first-order correct for parameter θ_i at $(x, \vec{\boldsymbol{\theta}})$ with kernel size ϵ if*

$$\frac{\partial_k f}{\partial \theta_i}(x, \vec{\boldsymbol{\theta}}; \epsilon) \;/\; \frac{\partial \hat{f}}{\partial \theta_i}(x, \vec{\boldsymbol{\theta}}; \epsilon) = 1 + O(\epsilon) \tag{2.14}$$



(a) Shader example for e_a

(b) Shader example for e_b

(c) Shader example for e_c

(d) Shader example for e_d

Figure 2.5: Shader examples that belong to different subsets of programs.

2.7.2 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 Chapter 3 that ours typically has a 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.

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

$$e_b ::= H(e_a) \mid H(e_b) \mid C \cdot e_b \mid e_b + e_b \mid e_b \cdot e_b \mid f(e_b)$$

$$e_c ::= e_a \mid e_b \mid e_c + e_c \mid e_c \cdot e_c \mid f(e_c)$$

$$e_d ::= e_c \mid H(e_d) \mid e_d + e_d \mid e_d \cdot e_d \mid f(e_d)$$

e_a represents all continuous programs that can be expressed in our DSL. Both ours and AD are correct for this set. For example, a color palette that smoothly changes color according to time and pixel coordinates belongs to this set (Figure 2.5 (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 three pathological cases described in Section 2.7.2.1: discontinuity degeneracy, part dependency on the sampling axis, and discontinuities with roots of order 3 or above. For example, a black and white blob whose shape changes parametrically belongs to e_b (Figure 2.5 (b)).

e_c represents the subset of programs whose discontinuities share similar constraints 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 the same pathological cases as before. For example, a blob whose color is rendered according to pixels' distance to the object boundary belongs to e_c (Figure 2.5 (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, Chapter 3 will empirically show that gradient approximations in this set have low error under a quantitative error metric. For example, an Olympic rings shader with parametric Z ordering can be expressed in the minimal DSL (Figure 2.5 (d)), where the visibility decision involves comparing a continuous value (Z value for the current ring) to a discontinuous value (accumulated Z).

2.7.2.1 Pathological Exceptions

To accurately characterize the set of programs our compilers can differentiate with first-order correctness, we define $\bar{e}_b, \bar{e}_c, \bar{e}_d$ as their original counterparts 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 4, 6 and 7. 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 correctness guarantee also holds on such programs if $\text{dom}(f)$ is chosen such that it does not contain any points with these properties.

Definition 4 *A program $f \in e_c$ has discontinuity degeneracy at $(x, \vec{\boldsymbol{\theta}})$ 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,* [3] *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 $f(x) = \{(x-1)^2,\ \text{if } x < 0,\ \text{otherwise } (x+1)^2\}$, which is C^0, not C^1, but the compiler identifies it as being not C^0 due to the Heaviside step.*

Before defining part dependency, we first start with a preliminary definition: locally zero along the sampling axis in Definition 5. Intuitively, when the function f is not statically constant but evaluates to constant values within a local interval, this may result in undefined values due to dividing by zero when the difference in function evaluations is used in the denominator for our gradient rules (e.g. Heaviside step and function composition in Table 2.2). This may further lead to pathological functions containing intermediate expressions $H(g)$ that always evaluate at the singularity $g = 0$ within a local interval.

Definition 5 *A function f is locally zero along the sampling axis (i.e. x) at $(x, \vec{\boldsymbol{\theta}})$ if $\exists\ \epsilon > 0$ s.t. $f(x', \vec{\boldsymbol{\theta}}) = 0\ \forall\ x' \in (x - \epsilon, x + \epsilon)$ whenever $(x', \vec{\boldsymbol{\theta}}) \in \text{dom}(f)$. Note also that to avoid excessively verbose notation, from here on, when there is such an x' we always implicitly assume $(x', \vec{\boldsymbol{\theta}}) \in \text{dom}(f)$, i.e. we assume we are within the set of points where f is defined.*

Definition 6 *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 $(x, \vec{\boldsymbol{\theta}})$, and $\partial g/\partial x$ exists, but $\partial g/\partial x$ is*

[3] While special operators such as `min`() and `max`() involve comparison and branching, their structure is simple enough for the compiler to identify and statically classify correctly.

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, but they are not the main focus of the proposed approach.*

Definition 7 *A program* $f \in e_c$ *has a discontinuity with a pth order root along the sampling axis* x *if for some intermediate value* $H(g)$ *with* $g \in e_a$, g *has a pth order zero along* x, *i.e.* $\partial^j g/\partial x^j = 0$ *for* $j = 0, \ldots, p-1$ *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 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 Section 2.7.3 does not guarantee anything about $\bar{e}_d$, we do not consider such programs.

2.7.3 First-Order Correctness Conclusions

We begin by defining sets where f is continuous and discontinuous with respect to (wrt) different parameters, then we "dilate" the discontinuous sets for reasons related to the measure theory that we will 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.

Definition 8 *Given a function* f, *the continuous set* C_i *is the set of all points* $(x, \vec{\boldsymbol{\theta}})$ *in* $\mathrm{dom}(f)$ *where* f *is continuous wrt* θ_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 and* $\partial^2 g/\partial x^2$ *evaluate to nonzero). The discontinuous set* $D_i = \mathrm{dom}(f) \setminus C_i$.

Definition 7 excludes 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 9 *The dilated 1D interval* $N^r(x, \vec{\boldsymbol{\theta}})$ *conditioned on some* $\epsilon_f(x, \vec{\boldsymbol{\theta}}) > 0$ *is defined as* $\{(x', \vec{\boldsymbol{\theta}}) \in \mathrm{dom}(f)) : x' \in (x - \beta\epsilon', x + \alpha\epsilon') \;\; \textit{for } \epsilon' = r\epsilon_f(x, \vec{\boldsymbol{\theta}})\}$.

Definition 10 *The dilated discontinuous set* D_i^r *is defined as* $D_i^r = \bigcup_{\forall (x, \vec{\boldsymbol{\theta}}) \in D_i} N^r(x, \vec{\boldsymbol{\theta}})$.

We are now ready to present our correctness results for both the continuous set C_i and dilated discontinuous set D_i^r. We start with the correctness for C_i using absolutely first-order correct defined in Equation 2.13.

Theorem 1 *For our approximation,* $\exists \epsilon_f(x, \vec{\boldsymbol{\theta}}) > 0$ *such that for all parameters* θ_i*, for every kernel size* $\epsilon \in (0, \epsilon_f(x, \vec{\boldsymbol{\theta}}))$*,* $f \in e_a \cup \bar{e}_b$ *is absolutely first-order correct on* C_i*, and almost everywhere on* C_i *for* $\bar{e}_c$.

Theorem 1 trivially holds for e_a and $\bar{e}_b$, and a proof sketch for $\bar{e}_c$ will be presented in Section 2.8. Because e_a programs are statically continuous, our result reduces to AD in the absence of discontinuities, therefore the left-hand side of Equation 2.13 is equivalent to the AD gradient, and trivially approaches the right-hand side as the pre-filtering kernel size ϵ goes to 0. For $\bar{e}_b$, because e_b functions are piecewise constant, its pre-filtered gradient is always 0 at the continuous set C_i. Similarly, our rule also gives 0 for the gradient in those regions because discontinuities are not sampled. We next give some intuition for $\bar{e}_c$. Because $f \in \bar{e}_c$ can be statically discontinuous, our gradient rules may be used instead of

AD. However, because the finite difference approximations made by our rules also approach the AD rules as ϵ goes to 0, on the continuous set C_i, both the left and right-hand side of Equation 2.13 approach the AD result as ϵ goes to 0, and we will show the error between these two is bounded by first-order terms. The choice of ϵ_f is formally described in Section 2.8.5, but intuitively it can be chosen such that the function f is Lipschitz continuous in the region $(x - \epsilon_f(x, \vec{\boldsymbol{\theta}}), x + \epsilon_f(x, \vec{\boldsymbol{\theta}}))$. This ensures the differences between the two evaluation sites on both function values and their gradients are bounded by first-order terms. Our almost everywhere results on C_i for $\bar{e}_c$ excludes measure zero sets of locations that may result in undefined values due to dividing by zero in the denominator for our gradient rules: we show this in Section 2.8 Lemma 3. Note although it is unrelated to the math theory, in practice in our implementation the division safeguard mentioned in Table 2.2 prevents a true division by zero error.

We next present the correctness result for the dilated discontinuous set D_i^r using the relatively first-order correct defined in Equation 2.13.

Theorem 2 *For our approximation,* $\forall r \in (0, 1]$, $\exists \epsilon_f(x, \vec{\boldsymbol{\theta}}) > 0, \tau : D_i^r \to D_i$, *such that for all parameters* θ_i, *for kernel size* $\epsilon_i^r = r\epsilon_f(\tau(x, \vec{\boldsymbol{\theta}}))$, $f \in \bar{e}_b \cup \bar{e}_c$ *is relatively first-order correct almost everywhere in in the dilated discontinuous set* D_i^r *(defined in Definition 10).*

Similar to Theorem 1, a proof sketch for Theorem 2 will be presented in Section 2.8. Because e_a is the set of statically continuous programs, its discontinuous set D_i is empty, therefore is omitted from Theorem 2. Additionally, because $\bar{e}_b \subset \bar{e}_c$, it is equivalent to prove Theorem 2 only for $f \in \bar{e}_c$. The kernel size ϵ_i^r is decided based on the radius r of the discontinuous set D_i^r: this ensures the discontinuity is always sampled with ϵ_i^r. Note that as r varies in $(0, 1]$, ϵ_i^r is proportional to r, so the result on D_i^r holds for a variety of kernel sizes. Also note that τ simply projects any point on the discontinuous set D_i^r back to the discontinuous set D_i: intuitively it projects any point to its closest discontinuity along the sampling axis x. It is needed so that the kernel size is decided based on ϵ_f evaluated at the discontinuity: similar to how the radius of D_i^r is decided. Our almost everywhere results

on D_i^r exclude measure zero sets of locations that have multiple discontinuities: we show this in Section 2.8 Lemma 5. 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 Section 2.8 Lemma 4 and 5).

2.8 Theoretical Error Bound Proof

This section presents the proof sketch for Theorem 1 and 2 when $f \in \bar{e}_c$. The cases for $f \in e_a \cup \bar{e}_b$ are less interesting and are already addressed in Section 2.7.3. The proof is structured as follows. We start in Section 2.8.1 to summarize the important definitions and lemmas for the program sets in Section 2.7.2. With the help of these preliminaries, we then define the set of points that are absolutely or relatively first-order correct as C-simple and D-simple points in Section 2.8.2. We next prove in Section 2.8.3 that C-simple points are almost everywhere in C_i and D-simple points are almost everywhere in D_i^r. Because we compare with the reference pre-filtered gradient in Theorem 1 and 2, we present in Section 2.8.4 a locally equivalent representation for $f \in e_c$ such that the pre-filtered gradient can be easily computed. After that, we construct ϵ_f in Section 2.8.5 to show its existence. And finally, we prove absolutely first-order correct for C-simple points, and relatively first-order correct for D-simple points through induction in Section 2.8.6.

2.8.1 Preliminaries

Section 2.7.2 procedurally defines the program set e_d expressible in our DSL based on program sets e_a, $\bar{e}_b$ and $\bar{e}_c$. Therefore in this section, we will first summarize the important definitions and lemmas for these program sets and briefly justify our claims.

We start with the local boundedness and continuity results for the program set e_a: the statically continuous programs. For simplicity but without loss of generality we always assume the function is evaluated on one given sampling axis x and other tunable parameters $\vec{\boldsymbol{\theta}}$. As a reminder, because the definition of the domain of f made in Section 2.4.2 always excludes computational singularities, our discussion in this section also excludes computational singularities.

Lemma 1 *A function $f \in e_a$ is evaluated at $(x, \vec{\boldsymbol{\theta}}) \in \text{dom}(f) \Rightarrow \exists\ \epsilon > 0$ 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 $f \in e_a$, its derivatives, and its second-order derivatives are all real analytic on the local region. Real analytic therefore leads to local boundedness by the boundedness theorem. Additionally, based on the multivariate mean value theorem, for any two points a, b, $\exists c$ along the line segment of (a, b) such that $|f(a) - f(b)| \leq |\nabla f(c)| \cdot |a - b|$. Because the derivative of f is locally bounded, f is therefore locally Lipschitz continuous. Similar argument could be applied to $\partial f/\partial x$ and $\partial f/\partial \theta$ as well. ■

We next prove that discontinuities are isolated for functions constructed from our DSL. Intuitively, this can be viewed as an analogy to the isolated zeros property for real analytic functions of one variable.

Lemma 2 Isolated discontinuities: *Given a function $f \in \bar{e}_c$ evaluated at $(x, \vec{\boldsymbol{\theta}}) \in \text{dom}(f)$, $\exists \epsilon > 0$ s.t. $\forall x' \in [x - \alpha\epsilon, x) \cup (x, x + \beta\epsilon]$, f is continuous at $(x', \vec{\boldsymbol{\theta}})$ along the x axis.*

Proof sketch: Because $f \in \bar{e}_c$ is constructed from our DSL, it only consists of a finite number (N) of $H(g_i)$ such that no other Heaviside function depends on $H(g_i)$. In other words, this means in the compute graph of f, no additional Heaviside step function node exists on any path between the output node of f and $H(g_i)$. Without loss of generality, we will only discuss the $N = 1$ case: $H(g)$ is the only discontinuity consumed by the output

of f. The $N > 1$ case can be generalized by following the $N = 1$ proof, then taking 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 $(x, \vec{\boldsymbol{\theta}})$. 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. Furthermore, in the $g \in e_b$ case, 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. We can therefore recursively decompose g similar to f until we reach a Heaviside function $H(h)$ such that $h \in e_a$. Based on this observation, we will prove it by induction. The base case proves $H(g)$ is continuous within the intervals $[x - \alpha\epsilon, x) \cup (x, x + \beta\epsilon]$ when $g \in e_a$. The induction step proves the same thing but with $g \in \bar{e}_b$, and assumes g is already continuous within the intervals $[x - \alpha\epsilon, x) \cup (x, x + \beta\epsilon]$. Note here and in the subsequent discussions, we refer to "within the interval" as meaning fixing $\vec{\boldsymbol{\theta}}$ and considering the 1D restriction of the function to the x axis.

Base case: $g \in e_a$. Because f is discontinuous at $(x, \vec{\boldsymbol{\theta}})$ wrt x, $\exists \epsilon' > 0$ such that discontinuity can be sampled $\forall \epsilon \in (0, \epsilon']$: $H(g(x - \alpha\epsilon)) \neq H(g(x + \beta\epsilon))$. Therefore g is not locally zero (wrt x). Additionally, because real analytic functions that are not locally zero have isolated zeros in 1D (sampling axis x), $\exists \epsilon > 0$ $s.t.$ $(x, \vec{\boldsymbol{\theta}})$ is the only zero for $g(x', \vec{\boldsymbol{\theta}})$ within the interval $x' \in [x - \alpha\epsilon, x + \beta\epsilon]$, so we have the desired result for the case $g \in e_a$: $(x, \vec{\boldsymbol{\theta}})$ is the only discontinuity for $H(g)$ within the claimed x' intervals.

Induction step: $g \in \bar{e}_b$ and assuming g is continuous wrt x in the intervals $[x - \alpha\epsilon, x) \cup (x, x + \beta\epsilon]$. As discussed before, because g is piece-wise constant, g being continuous within the intervals $[x-\alpha\epsilon, x)\cup(x, x+\beta\epsilon]$ implies g is constant for each connected intervals: $[x-\alpha\epsilon, x)$ and $(x, x + \beta\epsilon]$. Therefore $H(g)$ is continuous in the intervals $[x - \alpha\epsilon, x) \cup (x, x + \beta\epsilon]$.

Now we discuss f is continuous wrt x at $(x, \vec{\boldsymbol{\theta}})$. 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 this 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 $g(x, \vec{\boldsymbol{\theta}}) \neq 0$. If g is locally zero wrt x, ϵ can be chosen appropriately based on the size of the locally zero interval such that g stays zero. If $g(x, \vec{\boldsymbol{\theta}}) \neq 0$, 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 $g(x', \vec{\boldsymbol{\theta}}) \neq 0$. For the induction step where $g \in e_b$, because we already proved before that discontinuities wrt x are isolated, and because g is piece-wise constant, $\exists \epsilon > 0$ *s.t.* $[x - \alpha\epsilon, x + \beta\epsilon]$ is piece-wise constant, therefore f is continuous wrt x in the interval. ■

We further define two outlier scenarios where our first-order correctness properties cannot be proved everywhere, thus leading to the almost everywhere claims in our Theorem 1 and 2. Note Definition 11 and 12 characterizes when points already in C_i or D_i^r violate the first-order correctness. This is different from Section 2.7.2.1, where pathological exceptions are removed from the programs that we consider in our first-order correctness results. Also note that locally zero is defined in Definition 5.

Definition 11 *A function $f \in e_c$ evaluated at $(x, \vec{\boldsymbol{\theta}})$ is symmetric along the sampling axis x if for some intermediate value g of f, g is not statically continuous, and at $(x, \vec{\boldsymbol{\theta}})$, 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 $(x, \vec{\boldsymbol{\theta}})$.*

Definition 12 *A function $f \in e_c$ is multi-discontinuous at $(x, \vec{\boldsymbol{\theta}}) \in \mathrm{dom}(f)$ 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 $(x, \vec{\boldsymbol{\theta}})$, where $\nabla = [\partial/\partial x, \partial/\partial\theta_1, \ldots, \partial/\partial\theta_n]$.*

2.8.2 C and D-Simple Definitions

With the definitions in Section 2.8.1, we can now characterize the set of points in Definition 13 that we will show are absolutely first-order correct.

Definition 13 *For a function $f \in e_c$, $(x, \vec{\boldsymbol{\theta}}) \in C_i$ is C-simple if at $(x, \vec{\boldsymbol{\theta}})$, 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 as D-simple in Definition 15. 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 $(x', \vec{\boldsymbol{\theta}}) \in N_i^r(x, \vec{\boldsymbol{\theta}})$ back to discontinuous locations $(x, \vec{\boldsymbol{\theta}})$: this corresponds to τ in Theorem 2. Note the existence of the remapping is guaranteed because D_i^r is defined in Definition 10 as a union of $N_i^r(x, \vec{\boldsymbol{\theta}})$.

Definition 14 *$\forall (x', \vec{\boldsymbol{\theta}}) \in D_i^r$, we define a remapping $\tau(x', \vec{\boldsymbol{\theta}}) = (x, \vec{\boldsymbol{\theta}})$ such that $x \in D_i$ and $x' \in (x - \beta\epsilon', x + \alpha\epsilon')$ for $\epsilon' = r\epsilon_f(x, \vec{\boldsymbol{\theta}})$.*

Definition 15 *For a function $f \in e_c$, $(x', \vec{\boldsymbol{\theta}}) \in D_i^r$ is D-simple if $\tau(x', \vec{\boldsymbol{\theta}})$ is not at a multi-discontinuity.*

2.8.3 C and D-Simple Almost Everywhere Proof Sketch

Lemma 3 *For a function $f \in \bar{e}_c$ and a parameter θ_i, C-simple points are almost everywhere in C_i.*

Proof sketch: We justify this using an 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 $f \in \bar{e}_c$ is discontinuous along the sampling axis at $(x, \vec{\boldsymbol{\theta}})$, this implies $g(x, \vec{\boldsymbol{\theta}}) = 0$ for some intermediate value $H(g)$ of f with $g \in e_a \cup e_b$. 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 $h(x, \vec{\boldsymbol{\theta}}) = 0$. If we assume every intermediate value of the form $H(h)$ with $h \in e_a$ and $h(x, \vec{\boldsymbol{\theta}}) = 0$ has the property that h is locally zero wrt x at $(x, \vec{\boldsymbol{\theta}})$, then f would be continuous wrt x at $(x, \vec{\boldsymbol{\theta}})$, therefore

causing a contradiction. So for some intermediate value of the form $H(h)$ with $h \in e_a$ and $h(x, \vec{\boldsymbol{\theta}}) = 0$, h is not locally zero wrt x at $(x, \vec{\boldsymbol{\theta}})$. Note because $h \in e_a$, h is real analytic as a single-variate function wrt x within an open interval containing x intersected with $\text{dom}(f)$. Therefore, if we define $P_d(\vec{\boldsymbol{\theta}})$ as the set of points x *s.t.* $(x, \vec{\boldsymbol{\theta}}) \in C_i$ and are discontinuous along the sampling axis. we can conclude $P_d(\vec{\boldsymbol{\theta}})$ is a subset to the set of points that satisfies $h(x, \vec{\boldsymbol{\theta}}) = 0$ and h *not* locally zero for finitely many real analytic representations h.

Similarly, if f is symmetric along the sampling axis at $(x, \vec{\boldsymbol{\theta}})$, 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 2 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 $\text{dom}(f)$, per Def. 5) 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. Therefore, g and $\partial g/\partial x$ are real analytic single-variate functions within that local region. Additionally, these real analytic functions in the local region can be viewed as one of the finitely many possible local real analytic representations of $\partial g/\partial x$: $H(\cdot)$ can only be evaluated to 2 discrete values: $\{0, 1\}$. Because of these, we can define $P_s(\vec{\boldsymbol{\theta}})$ as the set of points x *s.t.* $(x, \vec{\boldsymbol{\theta}}) \in C_i$ and are symmetric along the sampling axis, and conclude $P_s(\vec{\boldsymbol{\theta}})$ is a subset to the set of points that satisfies $\partial g/\partial x = 0$ and $\partial g/\partial x$ *not* locally zero for finitely many real analytic representations $\partial g/\partial x$.

Given $\vec{\boldsymbol{\theta}}$, we now consider the set of points x s.t. $(x, \vec{\boldsymbol{\theta}}) \in C_i$ and along the sampling axis are symmetric or discontinuous wrt x: $P(\vec{\boldsymbol{\theta}}) = P_d(\vec{\boldsymbol{\theta}}) \cup P_s(\vec{\boldsymbol{\theta}})$. We will prove $P(\vec{\boldsymbol{\theta}})$ is measure zero by constructing its superset $P'(\vec{\boldsymbol{\theta}})$ and instead proving $P'(\vec{\boldsymbol{\theta}})$ is measure zero. $P'(\vec{\boldsymbol{\theta}})$ is constructed as the set of all zeros of all 1D real analytic (wrt x) local representations for the intermediate values g of f and its gradient, $\partial g/\partial x$, excluding points x where these real analytic functions are locally zero. From the previous paragraph, we have justified $P \subset P'$ and there are finitely many such real analytic functions within the program f. 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, therefore $\mu_1(P') = \mu_1(P) = 0$, where μ_1 is the Lebesgue measure on $\mathbb{R}$ for the x axis. We can further define μ_k to refer to the Lebesgue measure in $\mathbb{R}^k$. Recall that we want to prove the measure $\mu_{n+1}(S) = 0$ where S is the set of points that are discontinuous along the sampling axis or symmetric along the sampling axis in C_i, i.e. the union of $P(\vec{\boldsymbol{\theta}})$ for every $\vec{\boldsymbol{\theta}}$. We can replace $\mu_{n+1}(S)$ with the Lebesgue integral $\int 1_S d\mu = \mu_{n+1}(S)$, apply Fubini's theorem considering μ_{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 $\mu_1(P(\vec{\boldsymbol{\theta}})) = 0$. ■

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 is 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 formalize this intuition in a general Euclidean setting in Lemma 4, where the set $N(Z)$ corresponds to the "dilation operation". We will then apply Lemma 4 to prove the almost everywhere result for D-simple points in Lemma 5.

Lemma 4 *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 $N(Z) = \bigcup_{x \in Z} \nu(x)$. Let $\nu(x) = ((x_1 + a(x), x_1 + b(x)) \times \{x_2\} \times \{x_3\} \ldots \times \{x_m\}) \cap U$, $a : U \to \mathbb{R}, b : U \to \mathbb{R}$. Then $\mu_m(N(Z)) = 0$ where μ_m is the Lebesgue measure in $\mathbb{R}^m$.*

Proof: We proceed similarly to Claim 1 of Mityagin [85, 86].

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:

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

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 non-strict 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:

$$J^{i,j} = \begin{bmatrix} \partial G_1/\partial x_i & \partial G_1/\partial x_j \\ \partial G_2/\partial x_i & \partial G_2/\partial x_j \end{bmatrix} \tag{2.15}$$

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 ∇G_1 and ∇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 an $m-2$ dimensional manifold. If $p \in \nu(x)$ then there exists $t \in [0,1]$ such that $p = x + e_1(a(x) + t(b(x) - a(x))$, where $e_1 = [1, 0, 0, \ldots, 0]$, so we can parameterize $N(Q(x))$ by $m-1$ dimensions. Therefore $\mu_m(N(Q(x))) = 0$.

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. Because Z_k is compact, we can choose a finite subcover: choose the cover $Q(x_1), \ldots, Q(x_N)$ associated with points $x_1 \in Z_n, \ldots, x_N \in Z_n$. So $\mu_m(N(Z_n)) = 0$. Additionally, because $Z = \cup_k Z_k$, so by countable subadditivity of measures, $\mu_m(N(Z)) = 0$. [4] ■

[4] Thanks to Boris Mityagin for elucidating details of his proof technique [86] which we followed in our proof.

Lemma 5 *For a function $f \in e_c$ and a parameter θ_i, D-simple points are almost everywhere in D_i^r.*

Proof sketch: Using the 1D dilated intervals from Definition 9, 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 $S_d = \{(x, \vec{\boldsymbol{\theta}}) \in \text{dom}(f) : f \text{ is multi-discontinuous at } (x, \vec{\boldsymbol{\theta}})\}$, we wish to show $\mu_{n+1}(N^r(S_d)) = 0$. We will apply Lemma 4 to prove this. Specifically, we use $(x', \vec{\boldsymbol{\theta}})$ as the $n+1 = m$ dimensions, and the union of the 1D interval endpoints in Definition 9 for N^r (i.e. $x - \beta\epsilon'$ and $x + \alpha\epsilon'$) become the functions a, b in Lemma 4.

We next note that $N_r(S_d)$ can be viewed as the union of a finite number of sets $N_r(S_d^p)$: $N_r(S_d) = \cup_{p=1,\ldots,P} N_r(S_d^p)$. Each S_d^p corresponds to one instance of multi-discontinuity in Definition 12. Specifically, for each p, we choose from intermediate values $H(g_i)$, $H(g_j)$ or f with $g_i \in e_a$ and $g_j \in e_b$, and S_d^p is defined as the set of points where both g_i and g_j evaluate to zero, and ∇g_i, ∇g_j are linearly independent (definition of multi-discontinuity). We will use $G_1 = g_i$ and $G_2 = g_j$ for Lemma 4, and describe the choice of 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 dom(f) s.t. $g_i = g_j = 0$ and $\nabla g_i, \nabla g_j$ are linearly dependent: this leaves only the desired points S_d^p in the zero set for Lemma 4. 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 4 with nonzero determinant, which is continuous wrt $(x, \vec{\boldsymbol{\theta}})$, so we can likewise show the neighborhood around p exists. With all components chosen, we can trivially apply Lemma 4 to get $\mu_{n+1}(N^r(S_d^p)) = 0$. Finally, because $N_r(S_d)$ is the union of finitely many $N_r(S_d^p)$: $N_r(S_d) = \cup_{p=1,\ldots,P} N_r(S_d^p)$, we get $\mu_{n+1}(N^r(S_d)) = 0$. ■

2.8.4 Local Expansion for C and D-Simple Points

Because we show the correctness of our approximation by comparing it with a reference pre-filtered gradient, our proof also involves computing the reference. We show if a function

evaluation $f(x, \vec{\boldsymbol{\theta}})$ 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 6 *Local expansion: a function $f_1 \in e_c$ is either C-simple or D-simple at $(x, \vec{\boldsymbol{\theta}})$ $\Rightarrow \exists f_2 = a + b \cdot H(c)$ with $a, b, c \in e_a$ and $\exists \epsilon > 0$ s.t. within the set $S = [x - \alpha\epsilon, x + \beta\epsilon] \times \vec{\boldsymbol{\theta}}$, 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 $(x, \vec{\boldsymbol{\theta}})$ then $b = 0$.*

Proof sketch: The existence of ϵ can be justified because discontinuities are isolated on the 1D x axis (Lemma 2). The local expansion can be obtained by recursively evaluating step functions that are *not* discontinuous at $f(x, \vec{\boldsymbol{\theta}})$ 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, because the local expansion is identical with the original function f_1 except for measure zero of discontinuous points, their pre-filtered integrals are identical for the same ϵ because integral over a null set is zero. This implies their pre-filtered gradients are identical as well. ■

2.8.4.1 Commuting Differentiation and Integration Operators

With the local expansion, we can easily compute the reference gradient similar to Section 2.3. Note in the equations, we frequently commute the differentiation and integration operators to simplify the computation into integral over the Dirac delta. In general, this commuting is not allowed by the Leibniz integral rule because the integrand is discontinuous. However, in engineering context, this is usually valid when the gradient of the discontinuous function is expressed using the Dirac delta notation. Intuitively, if we first split the integral into multiple integrals over continuous regions and apply Leibniz integral rule to each of them, we get additional terms that differentiate the integral bounds, which corresponds to the discontinuities. If we instead integrate the gradient with Dirac delta notation over

the entire region, integration over the Dirac delta gives the identical result to those extra terms. Therefore the two approaches are equivalent. In this section, we formally prove that commuting the differentiation and integration operators is valid when pre-filtering a local expansion representation with the 1D box kernel.

Lemma 7 *For a function* $f = a + b \cdot H(c)$ *with* $a, b, c \in e_a \Rightarrow \exists \epsilon > 0$ *s.t.*

$$\frac{\partial}{\partial \theta_i} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} f(x', \vec{\boldsymbol{\theta}}) dx' = \int_{x-\alpha\epsilon}^{x+\beta\epsilon} \frac{\partial f(x', \vec{\boldsymbol{\theta}})}{\partial \theta_i} dx' \tag{2.16}$$

Proof: If f is continuous in the interval $(x - \alpha\epsilon, x + \beta\epsilon)$, Equation 2.16 can be trivially proved by applying the Leibniz integral rule. Therefore we focus on the case where discontinuity can be sampled within the interval of the integral. Additionally, because $a \in e_a$, Equation 2.17 is trivially true as well due to Leibniz integral rule. And because additivity holds both for differentiation and integration, our proof for Equation 2.16 can be reduced to Equation 2.18

$$\frac{\partial}{\partial \theta_i} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} a(x', \vec{\boldsymbol{\theta}}) dx' = \int_{x-\alpha\epsilon}^{x+\beta\epsilon} \frac{\partial a(x', \vec{\boldsymbol{\theta}})}{\partial \theta_i} dx' \tag{2.17}$$

$$\frac{\partial}{\partial \theta_i} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} b(x', \vec{\boldsymbol{\theta}}) \cdot H(c(x', \vec{\boldsymbol{\theta}})) dx' \stackrel{?}{=} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} \frac{\partial (b(x', \vec{\boldsymbol{\theta}}) \cdot H(c(x', \vec{\boldsymbol{\theta}})))}{\partial \theta_i} dx' \tag{2.18}$$

Because $c \in e_a$, we can use the isolated discontinuity property in Lemma 2 to choose $\epsilon > 0$ such that only one discontinuity x_d is sampled within the interval of the integral. Without loss of generality, we also assume $H(c)$ evaluates to 0 within the interval $(x - \alpha\epsilon, x_d)$ and evaluates to 1 within the interval $(x_d, x+\beta\epsilon)$. In other words, $H(c^-) = 0$ and $H(c^+) = 1$ when sampling the discontinuity at x_d. We can now rewrite the left-hand side of Equation 2.18 into the following:

$$\begin{aligned}\frac{\partial}{\partial\theta_i}\int_{x-\alpha\epsilon}^{x+\beta\epsilon} b(x',\vec{\boldsymbol{\theta}})\cdot H(c(x',\vec{\boldsymbol{\theta}}))dx' =& \frac{\partial}{\partial\theta_i}\int_{x_d}^{x+\beta\epsilon} b(x',\vec{\boldsymbol{\theta}})dx' \\ & \text{Assuming } H(c)=0 \text{ when } x'\in(x-\alpha\epsilon, x_d) \\ =& -b(x_d,\vec{\boldsymbol{\theta}})\cdot\frac{\partial x_d}{\partial\theta_i}+\int_{x_d}^{x+\beta\epsilon}\frac{\partial b(x',\vec{\boldsymbol{\theta}})}{\partial\theta_i}dx' \\ & \text{Applying Leibniz integral rule}\end{aligned} \tag{2.19}$$

Because x_d represents the discontinuity, its derivative wrt θ_i can be computed using the implicit function theorem by taking the total differentiation wrt θ_i on both sides of $c(x_d,\vec{\boldsymbol{\theta}})=0$.

$$\begin{aligned}&\frac{dc(x_d,\vec{\boldsymbol{\theta_i}})}{d\theta_i}=\frac{\partial c}{\partial x}\frac{\partial x_d}{\partial\theta_i}+\frac{\partial c}{\partial\theta_i}=0 \\ \Rightarrow\quad &\frac{\partial x_d}{\partial\theta_i}=-\frac{\partial c/\partial\theta_i}{\partial c/\partial x}\end{aligned} \tag{2.20}$$

In Equation 2.20, $\partial c/\partial x$ and $\partial c/\partial\theta_i$ represents the partial derivative of c wrt its input argument that aligns with the sampling axis, and its input argument that is the i's entry of θ respectively. We now plug Equation 2.20 into Equation 2.19.

$$\begin{aligned}\frac{\partial}{\partial\theta_i}\int_{x-\alpha\epsilon}^{x+\beta\epsilon} b(x',\vec{\boldsymbol{\theta}})\cdot H(c(x',\vec{\boldsymbol{\theta}}))dx' =& -b(x_d,\vec{\boldsymbol{\theta}})\cdot(-\frac{\partial c/\partial\theta_i}{\partial c/\partial x})+\int_{x_d}^{x+\beta\epsilon}\frac{\partial b(x',\vec{\boldsymbol{\theta}})}{\partial\theta_i}dx' \\ =& b(x_d,\vec{\boldsymbol{\theta}})\cdot\frac{\partial c/\partial\theta_i}{\partial c/\partial x}+\int_{x_d}^{x+\beta\epsilon}\frac{\partial b(x',\vec{\boldsymbol{\theta}})}{\partial\theta_i}dx'\end{aligned} \tag{2.21}$$

We now rewrite the right-hand side of Equation 2.18 and show it is identical to Equation 2.21.

$$\int_{x-\alpha\epsilon}^{x+\beta\epsilon} \frac{\partial(b \cdot H(c))}{\partial\theta_i} dx' = \int_{x-\alpha\epsilon}^{x+\beta\epsilon} (\frac{\partial b(x', \vec{\boldsymbol{\theta}})}{\partial\theta_i} H(c(x', \vec{\boldsymbol{\theta}})) + b(x', \vec{\boldsymbol{\theta}}) \frac{\partial H(c(x', \vec{\boldsymbol{\theta}}))}{\partial\theta_i}) dx'$$

$$= \int_{x_d}^{x+\beta\epsilon} \frac{\partial b(x', \vec{\boldsymbol{\theta}})}{\partial\theta_i} dx' + \int_{x-\alpha\epsilon}^{x+\beta\epsilon} b(x', \vec{\boldsymbol{\theta}}) \frac{\partial H(c(x', \vec{\boldsymbol{\theta}}))}{\partial\theta_i} dx'$$

Assuming $H(c) = 0$ when $x' \in (x - \alpha\epsilon, x_d)$

$$= \int_{x_d}^{x+\beta\epsilon} \frac{\partial b(x', \vec{\boldsymbol{\theta}})}{\partial\theta_i} dx' + \int_{x-\alpha\epsilon}^{x+\beta\epsilon} b(x', \vec{\boldsymbol{\theta}}) \delta(c(x', \vec{\boldsymbol{\theta}})) \frac{\partial c}{\partial\theta_i} dx'$$

$$= \int_{x_d}^{x+\beta\epsilon} \frac{\partial b(x', \vec{\boldsymbol{\theta}})}{\partial\theta_i} dx' + \int_{x-\alpha\epsilon}^{x+\beta\epsilon} b(x', \vec{\boldsymbol{\theta}}) \frac{\delta(c(x - x_d))}{|\partial c/\partial x|} \frac{\partial c}{\partial\theta_i} dx'$$

Applying Dirac delta scaling property

$$= \int_{x_d}^{x+\beta\epsilon} \frac{\partial b(x', \vec{\boldsymbol{\theta}})}{\partial\theta_i} dx' + b(x_d, \vec{\boldsymbol{\theta}}) \frac{\partial c/\partial\theta_i}{|\partial c/\partial x|}$$

Applying Dirac delta sifting property

$$= \int_{x_d}^{x+\beta\epsilon} \frac{\partial b(x', \vec{\boldsymbol{\theta}})}{\partial\theta_i} dx' + b(x_d, \vec{\boldsymbol{\theta}}) \frac{\partial c/\partial\theta_i}{\partial c/\partial x}$$

$\partial c/\partial x > 0$ because $H(c^-) = 0$ and $H(c^+) = 1$ (2.22a)

Comparing Equation 2.21 and 2.22a, we conclude that the left and right-hand sides of Equation 2.18 are indeed identical, therefore Equation 2.16 in Lemma 7 also holds. ■

2.8.5 Existence of ϵ_f in Theorem 1 and 2

We now characterize the ϵ_f used in Theorem 1 and 2 and briefly justify its existence. We show a certain ϵ_f exists that has strong properties needed next in Section 2.8.6 for the proof by induction over subsets of our DSL.

Lemma 8 *$\forall f \in \bar{e}_c$ that is either C-simple or D-simple at $(x, \vec{\boldsymbol{\theta}}) \in \text{dom}(f)$, $\exists \epsilon_f(x, \vec{\boldsymbol{\theta}}) > 0$ 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 $(x', \vec{\boldsymbol{\theta}}) \in \text{dom}(f)$, we have f is continuous at $(x', \vec{\boldsymbol{\theta}})$ along x. This is a stronger property than the RHS of Lemma 6:*

it is compatible with local expansions, and additionally, for C-simple locations, it lets us avoid any discontinuities in the neighborhood when showing absolutely first-order correct.

- $\forall g$ *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 denominators.*

- *There are certain expressions* e *that are used in our lemmas that evaluate to a nonzero value at* $(x, \vec{\boldsymbol{\theta}})$ *and do not depend on* ϵ *which are derived from some intermediate values in the program* f. *For these, we limit* ϵ *to be small enough so that* $1/(e + O(\epsilon)) = 1/e + O(\epsilon)$. *This lets us rewrite Taylor expansions into the desired form for first-order correctness.*

Proof sketch: The existence of ϵ_f in the first requirement is a direct application of the isolated discontinuities property of Lemma 2. For the second requirement, if g is discontinuous at $(x, \vec{\boldsymbol{\theta}})$, according to Lemma 6 we have $g = a + b \cdot H(c)$. Because both a and b are real analytic, ϵ_f doesn't exist implies $b(x, \vec{\boldsymbol{\theta}}) = 0$, which is discontinuity degeneracy and is excluded from $\bar{e}_c$, therefore raising a contradiction. If $g(x, \vec{\boldsymbol{\theta}})$ 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 ϵ with all other terms being locally bounded. Therefore if ϵ_f does not exist, it means $O(\epsilon)$ involves the multiplication of ϵ 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 $(x, \vec{\boldsymbol{\theta}})$. 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 ϵ_f is defined, the existence of $\epsilon_i^r(x', \vec{\boldsymbol{\theta}})$ can be easily justified using the remapping in Definition 14: $\epsilon_i^r(x', \vec{\boldsymbol{\theta}}) = r\epsilon_f(\tau(x', \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$.

2.8.6 First-Order Correctness Proof by Induction

This section proves Theorem 1 and 2 by induction on different operators. We start with functions that are constants or only depend on x or one of θ_i as the base case. After that, we show inductions steps on *every* operator our DSL includes.

In this section, whenever $x \notin D_i$, x_d and $\tau(x, \vec{\boldsymbol{\theta}})$ are always used interchangeably to represent the discontinuity of interest. For simplicity, we always assume in the presence of a discontinuity x_d, $H(c^+) = 1$ and $H(c^-) = 0$ similar to Section 2.8.4.1. The opposite case can be proved under the identical process.

As a shorthand, for a function f and $r \in (0, 1]$, our usage for first-order correct at (x, θ) for intermediate value g of f is based on context: if g is discontinuous wrt θ_i at $(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$, then it means relatively first order correct, and if g is continuous wrt θ_i at $(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$, then it means absolutely first-order correct. Additionally, we will denote $C_i(f), D_i(f), D_i^r(f)$ as C_i, D_i, D_i^r specific to function f, respectively.

2.8.6.1 Base Case

We start our proof with the base case, where function f is either constant, or depends only on x or one of θ_i.

Lemma 9 *Given f in the following form, $\forall(x, \vec{\boldsymbol{\theta}}) \in \mathrm{dom}(f)$ and $\forall\epsilon > 0$, f is absolutely first-order correct.*

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

Note for the functions in Lemma 9, D_i^r is always a null set. Therefore it is vacuous to discuss the relatively first-order correctness properties on D_i^r.

Proof: The first two cases are trivial: both ours and the reference gradient result in 0. We only need to prove $\frac{\partial_O \theta_i}{\partial \theta_i}$ is correct by comparing both sides of Equation 2.13.

$$\begin{aligned} \frac{\partial_O \theta_i}{\partial \theta_i} =& 1 \\ \frac{\partial \hat{\theta}_i}{\partial \theta_i} =& \frac{\partial}{\partial \theta_i} \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} \theta_i dx' \\ =& \frac{\partial}{\partial \theta_i} \theta_i = 1 \end{aligned} \tag{2.23}$$

Equation 2.13 therefore holds because its left and right-hand sides are equal. ■

2.8.6.2 Induction on Addition

We first present the result for absolutely first-order correct, followed by relatively first-order correct.

Lemma 10 *Given a function $f = g + h$ with $f, g, h \in \bar{e}_c$, $\forall(x, \vec{\boldsymbol{\theta}}) \in C_i$ that are C-simple for f and $\forall\epsilon \in (0, \epsilon_f(x, \vec{\boldsymbol{\theta}})]$, if g, h are absolutely first-order correct, then f is absolutely first-order correct.*

Proof: The reference gradient of f can be computed as follows:

$$\begin{aligned}\frac{\partial \hat{f}}{\partial \theta_i} =& \frac{\partial}{\partial \theta_i}\frac{1}{(\alpha+\beta)\epsilon}\int_{x-\alpha\epsilon}^{x+\beta\epsilon}(g+h)dx' \\ =& \frac{\partial}{\partial \theta_i}\frac{1}{(\alpha+\beta)\epsilon}\int_{x-\alpha\epsilon}^{x+\beta\epsilon} g dx' + \frac{\partial}{\partial \theta_i}\frac{1}{(\alpha+\beta)\epsilon}\int_{x-\alpha\epsilon}^{x+\beta\epsilon} h dx' \\ =& \frac{\partial \hat{g}}{\partial \theta_i} + \frac{\partial \hat{h}}{\partial \theta_i}\end{aligned} \tag{2.24}$$

Because $(x, \vec{\boldsymbol{\theta}})$ is C-simple, discontinuity is excluded, we have $(x, \vec{\boldsymbol{\theta}}) \in C_i(g)$ and $(x, \vec{\boldsymbol{\theta}}) \in C_i(h)$ as well.

$$\begin{aligned}\frac{\partial_O f}{\partial \theta_i} =& \frac{\partial_k g}{\partial \theta_i} + \frac{\partial_k h}{\partial \theta_i} \\ =& \frac{\partial \hat{g}}{\partial \theta_i} + \frac{\partial \hat{h}}{\partial \theta_i} + O(\epsilon)\end{aligned} \tag{2.25}$$

Using Equation 2.13 to g, h

Comparing Equation 2.24 and 2.25, we conclude Equation 2.13 holds for f. ■

Lemma 11 *Given a function $f = g + h$ with $f, g, h \in \bar{e}_c$, $\forall r \in (0, 1], \forall (x, \vec{\boldsymbol{\theta}}) \in D_i^r$ that are D-simple for f, if g, h are first-order correct for $\epsilon = r\epsilon_f(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$, then f is relatively first-order correct for the same ϵ.*

Proof: We split the proof into two cases based on whether either or both of g, h are discontinuous at $(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$. Note Lemma 11 discusses D-simple points, therefore we do not need to consider the case when both g and h are continuous, because that implies f is continuous at $(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$, leaving D_i^r being a null set. Because we prove by induction, we are agnostic to how the gradient to g and h are approximated as long as they are correct under the definition. Therefore, we use ∂_k to denote that it can be either from ours (O), AD, or any other approximation.

Case 1: g is discontinuous at $(\tau(x,\vec{\boldsymbol{\theta}}),\vec{\boldsymbol{\theta}})$, but h is continuous at $(\tau(x,\vec{\boldsymbol{\theta}}),\vec{\boldsymbol{\theta}})$ (the same reasoning also goes with the inverse). Based on the premise in Lemma 11, we have g is relatively first-order correct and h is absolutely first-order correct. We will start with the left-hand side of Equation 2.14 and show it is equivalent to the right-hand side.

$$
\begin{aligned}
\frac{\frac{\partial_O f}{\partial\theta_i}}{\frac{\partial\hat{f}}{\partial\theta_i}} =& \frac{\frac{\partial_k g}{\partial\theta_i}+\frac{\partial_k h}{\partial\theta_i}}{\frac{\partial\hat{g}}{\partial\theta_i}+\frac{\partial\hat{h}}{\partial\theta_i}} \\
=& \frac{\frac{\partial_k g}{\partial\theta_i}+\frac{\partial\hat{h}}{\partial\theta_i}+O(\epsilon)}{\frac{\partial\hat{g}}{\partial\theta_i}+\frac{\partial\hat{h}}{\partial\theta_i}} \\
& \text{Using Equation 2.13 to } h \\
=& \frac{\frac{\partial_k g}{\partial\theta_i}/\frac{\partial\hat{g}}{\partial\theta_i}+\frac{\partial\hat{h}}{\partial\theta_i}/\frac{\partial\hat{g}}{\partial\theta_i}+O(\epsilon)}{1+\frac{\partial\hat{h}}{\partial\theta_i}/\frac{\partial\hat{g}}{\partial\theta_i}} \\
=& \frac{1+O(\epsilon)+\frac{\partial\hat{h}}{\partial\theta_i}/\frac{\partial\hat{g}}{\partial\theta_i}+O(\epsilon)}{1+\frac{\partial\hat{h}}{\partial\theta_i}/\frac{\partial\hat{g}}{\partial\theta_i}} \\
& \text{Using Equation 2.14 to } g \\
=& 1+O(\epsilon)
\end{aligned}
\tag{2.26}
$$

Case 2: Both g,h are discontinuous at $(\tau(x,\vec{\boldsymbol{\theta}}),\vec{\boldsymbol{\theta}})$. This implies both g,h are relatively first-order correct. Similarly, we start with the left-hand side of Equation 2.14.

$$
\begin{aligned}
\frac{\frac{\partial_O f}{\partial \theta_i}}{\frac{\partial \hat{f}}{\partial \theta_i}} =& \frac{\frac{\partial_k g}{\partial \theta_i} + \frac{\partial_k h}{\partial \theta_i}}{\frac{\partial \hat{g}}{\partial \theta_i} + \frac{\partial \hat{h}}{\partial \theta_i}} \\
=& \frac{\frac{\partial_k g}{\partial \theta_i} / (\frac{\partial \hat{g}}{\partial \theta_i} \cdot \frac{\partial \hat{h}}{\partial \theta_i}) + \frac{\partial_k h}{\partial \theta_i} / (\frac{\partial \hat{g}}{\partial \theta_i} \cdot \frac{\partial \hat{h}}{\partial \theta_i})}{1 / \frac{\partial \hat{h}}{\partial \theta_i} + 1 / \frac{\partial \hat{g}}{\partial \theta_i}} \\
=& \frac{(1 + O(\epsilon)) / \frac{\partial \hat{h}}{\partial \theta_i} + (1 + O(\epsilon) / \frac{\partial \hat{g}}{\partial \theta_i}}{1 / \frac{\partial \hat{h}}{\partial \theta_i} + 1 / \frac{\partial \hat{g}}{\partial \theta_i}} \\
& \text{Using Equation 2.14 to } g, h \\
& \text{Denominator nonzero because discontinuity degeneracy excluded} \\
=& 1 + O(\epsilon)
\end{aligned}
\tag{2.27}
$$

Combining both cases, we conclude f is always relatively first-order correct. ■

2.8.6.3 Induction on Multiplication

Lemma 12 *Given a function $f = g \cdot h$ with $f, g, h \in \bar{e}_c$, $\forall (x, \vec{\boldsymbol{\theta}}) \in C_i$ that are C-simple for f and $\forall \epsilon \in (0, \epsilon_f(x, \vec{\boldsymbol{\theta}})]$, if g, h are absolutely first-order correct, then f is absolutely first-order correct.*

Proof: Because $(x, \vec{\boldsymbol{\theta}})$ is C-simple, f, g and h are all continuous at $(x, \vec{\boldsymbol{\theta}})$. Therefore, based on Lemma 6, we can apply local expansion to g and h and rewrite them into the following:

$$
\begin{aligned}
g \equiv & a_g, \qquad a_g \in e_a \\
h \equiv & a_h, \qquad a_h \in e_a
\end{aligned}
\tag{2.28}
$$

Because $f = g \cdot h$, f can also be rewritten using the local expansion of g and h.

$$
f = g \cdot h = a_g \cdot a_h \tag{2.29}
$$

We next compute the reference pre-filtered gradient for f.

$$\begin{aligned}\frac{\partial \hat{f}}{\partial \theta_i} =& \frac{\partial}{\partial \theta_i} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} a_g a_h dx' \\ =& \int_{x-\alpha\epsilon}^{x+\beta\epsilon} (a_g \frac{\partial a_h}{\partial \theta_i} + a_h \frac{\partial a_g}{\partial \theta_i}) dx'\end{aligned} \tag{2.30}$$

Finally, we can compute our gradient approximation.

$$\begin{aligned}\frac{\partial_O f}{\partial \theta_i} =& \frac{1}{2}(h^+ + h^-)\frac{\partial_k g}{\partial \theta_i} + \frac{1}{2}(g^+ + g^-)\frac{\partial_k h}{\partial \theta_i} \\ =& \frac{1}{2}(a_h^+ + a_h^-)\frac{\partial_k g}{\partial \theta_i} + \frac{1}{2}(a_g^+ + a_g^-)\frac{\partial_k h}{\partial \theta_i} + O(\epsilon) \\ =& a_h \frac{\partial_k g}{\partial \theta_i} + a_g \frac{\partial_k h}{\partial \theta_i} + O(\epsilon)\end{aligned}$$

a_g, a_h are locally Lipschitz continuous

$$= a_h \frac{\partial \hat{g}}{\partial \theta_i} + a_g \frac{\partial \hat{h}}{\partial \theta_i} + O(\epsilon) \tag{2.31a}$$

Using Equation 2.13 to g, h

$$= \frac{1}{(\alpha+\beta)\epsilon}(a_h \int_{x-\alpha\epsilon}^{x+\beta\epsilon} \frac{\partial a_g}{\partial \theta_i} dx' + a_g \int_{x-\alpha\epsilon}^{x+\beta\epsilon} \frac{\partial a_h}{\partial \theta_i} dx') + O(\epsilon)$$

Applying Equation 2.2, swapping integral operator in pre-filtering with differentiation

$$= \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} [a_h \frac{\partial a_g}{\partial \theta_i} + a_g \frac{\partial a_h}{\partial \theta_i}] dx' + O(\epsilon)$$

a_g, a_h being locally Lipschitz continuous

$$= \frac{\partial \hat{f}}{\partial \theta_i} + O(\epsilon)$$

Applying Equation 2.30 (2.31b)

Comparing Equation 2.31b with Equation 2.13, we conclude that f is absolutely first-order correct. ■

Lemma 13 *Given a function $f = g \cdot h$ with $f, g, h \in \bar{e}_c$, $\forall r \in (0,1], \forall (x, \vec{\boldsymbol{\theta}}) \in D_i^r$ that are D-simple for f, if g, h are first-order correct for $\epsilon = r\epsilon_f(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$, then f is relatively first-order correct for the same ϵ.*

Proof: We first apply Lemma 6 to locally expand g and h into the following:

$$\begin{aligned} g =& a_g + b_g \cdot H(c_g) \quad a_g, b_g, c_g \in e_a \\ h =& a_h + b_h \cdot H(c_h) \quad a_h, b_h, c_h \in e_a \end{aligned} \tag{2.32}$$

Note because $f \in \bar{e}_c$, multi-discontinuity is excluded. This means if both g and h are discontinuous at $(x_d, \vec{\boldsymbol{\theta}})$, ∇c_g and ∇c_h will be linearly dependent. While c_g and c_h are not necessarily linearly dependent, in our proof, we never evaluate the function values of c_g and c_h, but only evaluate their derivatives in the form $\frac{\partial c/\partial \theta}{\partial c/\partial x}$. Therefore for brevity, we will assume $c = c_g = c_h$. Based on the local expansion of g and h, we can rewrite f into the following.

$$f = g \cdot h = a_g a_h + (a_g b_h + b_g a_h + b_g b_h) \cdot H(c) \tag{2.33}$$

We next compute the reference pre-filtered gradient for f. Note because Lemma 13 discusses D-simple sets, this implies the discontinuity x_d is already sampled within the interval $(x - \alpha\epsilon, x + \beta\epsilon)$.

$$\begin{aligned} \frac{\partial \hat{f}}{\partial \theta_i} =& \frac{\partial}{\partial \theta_i} \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} [a_g a_h + (a_g b_h + b_g a_h + b_g b_h) \cdot H(c)] dx' \\ =& \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} (\frac{\partial a_g a_h}{\partial \theta_i} + \frac{\partial (a_g b_h + b_g a_h + b_g b_h)}{\partial \theta_i} H(c)) + dx' \\ &+ \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} (a_g b_h + b_g a_h + b_g b_h) \delta(c) \frac{\partial c}{\partial \theta_i} dx' \\ =& \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} (\frac{\partial a_g a_h}{\partial \theta_i} + \frac{\partial (a_g b_h + b_g a_h + b_g b_h)}{\partial \theta_i} H(c)) dx' \end{aligned}$$

$$+\frac{(a_g b_h + b_g a_h + b_g b_h)\frac{\partial c}{\partial \theta}}{(\alpha+\beta)\epsilon|\frac{\partial c}{\partial x}|}\ |_{x_d} \tag{2.34a}$$

Applying sifting and scaling properties to the Dirac delta

$$=\frac{1}{(\alpha+\beta)\epsilon}\int_{x-\alpha\epsilon}^{x+\beta\epsilon}\frac{\partial a_g a_h}{\partial \theta_i}dx' + \frac{1}{(\alpha+\beta)\epsilon}\int_{x_d}^{x+\beta\epsilon}\frac{\partial(a_g b_h + b_g a_h + b_g b_h)}{\partial \theta_i}dx'$$
$$+\frac{(a_g b_h + b_g a_h + b_g b_h)\frac{\partial c}{\partial \theta}}{(\alpha+\beta)\epsilon|\frac{\partial c}{\partial x}|}\ |_{x_d}$$

Assuming $H(c^-) = 0$ and $H(c^+) = 1$

$$=\frac{1}{(\alpha+\beta)\epsilon}(\frac{\partial a_g a_h}{\partial \theta_i}(\alpha+\beta)\epsilon + \frac{\partial(a_g b_h + b_g a_h + b_g b_h)}{\partial \theta_i}(x+\beta\epsilon - x_d))$$
$$+\frac{(a_g b_h + b_g a_h + b_g b_h)\frac{\partial c}{\partial \theta}}{(\alpha+\beta)\epsilon|\frac{\partial c}{\partial x}|}\ |_{x_d} + O(\epsilon) \tag{2.34b}$$

a_g, a_h, b_g, b_h and their gradients are locally Lipschitz continuous

$$=\frac{1}{(\alpha+\beta)\epsilon}(O(\epsilon)+O(\epsilon)) + \frac{(a_g b_h + b_g a_h + b_g b_h)\frac{\partial c}{\partial \theta}}{(\alpha+\beta)\epsilon|\frac{\partial c}{\partial x}|}\ |_{x_d} + O(\epsilon)$$

Using $x_d \in (x-\alpha\epsilon, x+\beta\epsilon)$

a_g, a_h, b_g, b_h and their gradients are locally bounded

$$=\frac{1}{(\alpha+\beta)\epsilon}(\frac{(a_g b_h + b_g a_h + b_g b_h)\frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|}\ |_{x_d} + O(\epsilon) + O(\epsilon^2))$$
$$=\frac{1}{(\alpha+\beta)\epsilon}(\frac{(a_g b_h + b_g a_h + b_g b_h)\frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|}\ |_{x_d} + O(\epsilon)) \tag{2.34c}$$

Similarly, we can compute the reference pre-filtered gradient for g (the same also holds for h). When g is continuous, its pre-filtered gradient can be trivially obtained by expanding Equation 2.2, therefore we focus on the case when g is discontinuous at $(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$.

$$\frac{\partial \hat{g}}{\partial \theta_i} = \frac{\partial}{\partial \theta_i}\frac{1}{(\alpha+\beta)\epsilon}\int_{x-\alpha\epsilon}^{x+\beta\epsilon}(a_g + b_g \cdot H(c))dx'$$
$$=\frac{1}{(\alpha+\beta)\epsilon}(\int_{x-\alpha\epsilon}^{x+\beta\epsilon}(\frac{\partial a_g}{\partial \theta_i} + \frac{\partial b_g}{\partial \theta_i}H(c))dx' + \frac{b_g\frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|}\ |_{x_d}) \tag{2.35a}$$

Scaling and sifting property for Dirac delta

$$
\begin{aligned}
=&\frac{1}{(\alpha+\beta)\epsilon}(\int_{x-\alpha\epsilon}^{x+\beta\epsilon}\frac{\partial a_g}{\partial\theta_i}dx'+\int_{x_d}^{x+\beta\epsilon}\frac{\partial b_g}{\partial\theta_i}dx'+\frac{b_g\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\mid_{x_d})\\
&\text{Assuming } H(c^-)=0 \text{ and } H(c^+)=1\\
=&\frac{1}{(\alpha+\beta)\epsilon}(\frac{\partial a_g}{\partial\theta_i}(\alpha+\beta)\epsilon+\frac{\partial b_g}{\partial\theta_i}(x+\beta\epsilon-x_d)+\frac{b_g\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\mid_{x_d})+O(\epsilon)\\
&\text{Gradients for } a_g, b_g \text{ are locally Lipschitz continuous}\\
=&\frac{1}{(\alpha+\beta)\epsilon}(O(\epsilon)+O(\epsilon)+\frac{b_g\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\mid_{x_d}+O(\epsilon^2))\\
&\text{Using } x_d\in(x-\alpha\epsilon,x+\beta\epsilon) \text{ and gradients for } a_g, b_g \text{ are locally bounded}\\
=&\frac{1}{(\alpha+\beta)\epsilon}(\frac{b_g\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\mid_{x_d}+O(\epsilon)) \qquad (2.35b)
\end{aligned}
$$

We are ready to derive the relatively first-order result for f. Similar to the addition in Section 2.8.6.2, we split into two cases, only one of g or h is discontinuous or both are discontinuous. We similarly use ∂_k for the inductive gradient for g or h to denote that we are agnostic to how they are computed.

Case 1: g is discontinuous at $(\tau(x,\vec{\boldsymbol{\theta}}),\vec{\boldsymbol{\theta}})$, but h is continuous at $(\tau(x,\vec{\boldsymbol{\theta}}),\vec{\boldsymbol{\theta}})$ (the same reasoning also goes with the inverse). This indicates $b_g\neq 0$ and $b_h=0$.

$$
\begin{aligned}
\frac{\frac{\partial_O f}{\partial\theta_i}}{\frac{\partial\hat{f}}{\partial\theta_i}}=&(\alpha+\beta)\epsilon\frac{\frac{1}{2}(a_h^+ + a_h^-)\frac{\partial_k g}{\partial\theta_i}+\frac{1}{2}(g^+ + g^-)\frac{\partial_k h}{\partial\theta_i}}{\frac{b_g a_h\frac{\partial c}{\partial\vec{\theta}}}{|\frac{\partial c}{\partial x}|}\mid_{x_d}+O(\epsilon)}\\
&\text{Using Equation 2.34c and } b_h=0\\
=&(\alpha+\beta)\epsilon\frac{\frac{1}{2}(a_h^+ + a_h^-)\frac{\partial_k g}{\partial\theta_i}+\frac{1}{2}(g^+ + g^-)\frac{\partial\hat{h}}{\partial\theta_i}+O(\epsilon)}{\frac{b_g a_h\frac{\partial c}{\partial\vec{\theta}}}{|\frac{\partial c}{\partial x}|}\mid_{x_d}+O(\epsilon)}\\
&h \text{ is absolutely first-order correct}\\
=&\frac{a_h\frac{\partial_k g}{\partial\theta_i}(\alpha+\beta)\epsilon+\frac{1}{2}(g^+ + g^-)\frac{\partial\hat{a_h}}{\partial\theta_i}(\alpha+\beta)\epsilon+O(\epsilon)}{\frac{b_g a_h\frac{\partial c}{\partial\vec{\theta}}}{|\frac{\partial c}{\partial x}|}\mid_{x_d}+O(\epsilon)}\\
&\text{Using } b_h=0 \text{ and } a_h \text{ is locally Lipschitz continuous}
\end{aligned}
$$

$$= \frac{a_h \frac{\partial_k g}{\partial \theta_i}(\alpha+\beta)\epsilon + O(\epsilon)}{\frac{b_g a_h \frac{\partial c}{\partial \theta}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d} + O(\epsilon)} \tag{2.36a}$$

Using a_h, $\partial a_h/\partial\theta_i$ and g are locally bounded

$$= \frac{a_h \frac{\partial_k g}{\partial \theta_i}(\alpha+\beta)\epsilon}{\frac{b_g a_h \frac{\partial c}{\partial \theta}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d}} + O(\epsilon) \tag{2.36b}$$

Nonzero denominator: $x \in D_i \rightarrow \dfrac{\partial c}{\partial \theta_i} \neq 0$

Discontinuities with roots of order $n \geq 2$ are excluded from $D_i(n=2)$ and $\bar{e}_c \rightarrow \dfrac{\partial c}{\partial x} \neq 0$

Discontinuity degeneracy is exclude from $\bar{e}_c \rightarrow b_g a_h \neq 0$

$O(\epsilon)$ in the denominator can be removed per the third requirement of Lemma 8

$$= \frac{(\alpha+\beta)\epsilon a_h \frac{\partial_k g}{\partial \theta_i}\frac{\partial \hat{g}}{\partial \theta_i} / \frac{\partial \hat{g}}{\partial \theta_i}}{\frac{b_g a_h \frac{\partial c}{\partial \theta}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d}} + O(\epsilon)$$

$$= \frac{(\alpha+\beta)\epsilon a_h (1+O(\epsilon)) \frac{1}{(\alpha+\beta)\epsilon}\left(\frac{b_g \frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d} + O(\epsilon)\right)}{\frac{b_g a_h \frac{\partial c}{\partial \theta}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d}} + O(\epsilon)$$

Using Equation 2.14 and 2.35b to g

$$= \frac{a_h(x, \vec{\boldsymbol{\theta}})(1+O(\epsilon))}{a_h(x_d, \vec{\boldsymbol{\theta}})} + O(\epsilon)$$

$$= 1 + O(\epsilon)$$

Using a_h is locally Lipschitz continuous

Case 2: Both g, h are discontinuous at $(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$. This indicates $b_g \neq 0$ and $b_h \neq 0$.

$$\frac{\frac{\partial_O f}{\partial \theta_i}}{\frac{\partial \hat{f}}{\partial \theta_i}} = (\alpha+\beta)\epsilon \frac{\frac{1}{2}(h^+ + h^-)\frac{\partial_k g}{\partial \theta_i} + \frac{1}{2}(g^+ + g^-)\frac{\partial_k h}{\partial \theta_i}}{\frac{(a_g b_h + b_g a_h + b_g b_h)\frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d}} + O(\epsilon)$$

Using Equation 2.34c

$$= (\alpha+\beta)\epsilon \frac{\frac{1}{2}(a_h^+ + a_h^- + b_h^+)\frac{\partial_k g}{\partial \theta_i} + \frac{1}{2}(a_g^+ + a_g^- + b_g^+)\frac{\partial_k h}{\partial \theta_i}}{\frac{(a_g b_h + b_g a_h + b_g b_h)\frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d}} + O(\epsilon)$$

Moving $O(\epsilon)$ from denominator to numerator: similar to Equation 2.36b

Assuming $H(c^-) = 0$ and $H(c^+) = 1$

$$=(\alpha+\beta)\epsilon\frac{(a_h+\frac{b_h}{2}+O(\epsilon))\frac{\partial_k g}{\partial\theta_i}+(a_g+\frac{b_g}{2}+O(\epsilon))\frac{\partial_k h}{\partial\theta_i}}{\frac{(a_g b_h+b_g a_h+b_g b_h)\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\mid_{x_d}}+O(\epsilon)$$

a_g, b_g, a_h, b_h are locally Lipschitz continuous

$$=(\alpha+\beta)\epsilon\frac{(a_h+\frac{b_h}{2}+O(\epsilon))\frac{\partial_k g}{\partial\theta_i}\frac{\partial\hat{g}}{\partial\theta_i}/\frac{\partial\hat{g}}{\partial\theta_i}}{\frac{(a_g b_h+b_g a_h+b_g b_h)\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\mid_{x_d}}+(\alpha+\beta)\epsilon\frac{(a_g+\frac{b_g}{2}+O(\epsilon))\frac{\partial_k h}{\partial\theta_i}\frac{\partial\hat{h}}{\partial\theta_i}/\frac{\partial\hat{h}}{\partial\theta_i}}{\frac{(a_g b_h+b_g a_h+b_g b_h)\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\mid_{x_d}}+O(\epsilon)$$

$$=\frac{(a_h+b_h/2+O(\epsilon))(1+O(\epsilon))(\frac{b_g\partial c/\partial\theta_i}{|\partial c/\partial x|}\mid_{x_d}+O(\epsilon))}{\frac{(a_g b_h+b_g a_h+b_g b_h)\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\mid_{x_d}}$$

$$+\frac{(a_g+b_g/2+O(\epsilon))(1+O(\epsilon))(\frac{b_h\partial c/\partial\theta_i}{|\partial c/\partial x|}\mid_{x_d}+O(\epsilon))}{\frac{(a_g b_h+b_g a_h+b_g b_h)\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\mid_{x_d}}+O(\epsilon)$$

Using Equation 2.14 and 2.35b to g, h

$$=\frac{(a_h+\frac{b_h}{2})\mid_x b_g\mid_{x_d}+(a_g+\frac{b_g}{2})\mid_x b_h\mid_{x_d}}{(a_g b_h+b_g a_h+b_g b_h)\mid_{x_d}}+O(\epsilon)$$

$$=\frac{(a_h+\frac{b_h}{2})b_g+(a_g+\frac{b_g}{2})b_h}{(a_g b_h+b_g a_h+b_g b_h)}\mid_{x_d}+O(\epsilon)$$

a_g, b_g, a_h, b_h are locally Lipschitz continuous

$$=1+O(\epsilon)$$

Combining both cases, we conclude that f is always relatively first-order correct. ■

2.8.6.4 Induction on Heaviside Step Function

Lemma 14 *Given a function $f = H(g)$ with $f, g \in \bar{e}_c$, $\forall(x, \vec{\boldsymbol{\theta}}) \in C_i$ that are C-simple for f and $\forall\epsilon \in (0, \epsilon_f(x, \vec{\boldsymbol{\theta}})]$, if g is absolutely first-order correct, then f is absolutely first-order correct.*

Proof: Lemma 14 is trivially true because, for C-simple points, both our approximation and reference gradient are zero. ■

Lemma 15 *Given a function $f = H(g)$ with $f, g \in \bar{e}_c$, $\forall r \in (0,1], \forall (x, \vec{\boldsymbol{\theta}}) \in D_i^r$ that are D-simple for f, if g is first-order correct for $\epsilon = r\epsilon_f(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$, then f is relatively first-order correct for the same ϵ.*

Proof: We prove by two cases, whether g is continuous at $(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$ or not. In both cases, we first need to locally expand g by applying Lemma 6.

$$g = a_g + b_g \cdot H(c_g) \qquad a_g, b_g, c_g \in e_a \tag{2.38}$$

Case 1: g is continuous at $(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$. This indicates $b_g = 0$. We first compute the reference pre-filtered gradient for $\hat{f}$.

$$\begin{aligned}
\frac{\partial \hat{f}}{\partial \theta_i} =& \frac{\partial}{\partial \theta_i} \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} H(a_g) dx' \\
=& \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} \delta(a_g) \frac{\partial a_g}{\partial \theta_i} dx' \\
=& \frac{\frac{\partial a_g}{\partial \theta_i}}{(\alpha+\beta)\epsilon |\frac{\partial a_g}{\partial x}|} \Big|_{x_d}
\end{aligned} \tag{2.39}$$

With Equation 2.39, we are now ready to expand the left-hand side of Equation 2.14.

$$\begin{aligned}
\frac{\frac{\partial_O f}{\partial \theta_i}}{\frac{\partial \hat{f}}{\partial \theta_i}} =& \frac{\frac{\partial_k g}{\partial \theta_i} / |g^+ - g^-|}{\frac{\frac{\partial a_g}{\partial \theta_i}}{(\alpha+\beta)\epsilon |\frac{\partial a_g}{\partial x}|} \Big|_{x_d}} \\
& \text{Using Equation 2.39} \\
=& \frac{(\frac{\partial \hat{g}}{\partial \theta_i} + O(\epsilon)) / |a_g^+ - a_g^-|}{\frac{\frac{\partial a_g}{\partial \theta_i}}{(\alpha+\beta)\epsilon |\frac{\partial a_g}{\partial x}|} \Big|_{x_d}} \\
& \text{Using Equation 2.13 and } b_g = 0 \\
=& \frac{\frac{\partial \hat{a_g}}{\partial \theta_i} / |a_g^+ - a_g^-|}{\frac{\frac{\partial a_g}{\partial \theta_i}}{(\alpha+\beta)\epsilon |\frac{\partial a_g}{\partial x}|} \Big|_{x_d}} + O(\epsilon)
\end{aligned}$$

Using $b_g = 0$

$$= \frac{\frac{\partial a_g}{\partial \theta_i} \mid_{x_d} / |a_g^+ - a_g^-|}{\frac{\frac{\partial a_g}{\partial \theta_i}}{(\alpha+\beta)\epsilon|\frac{\partial a_g}{\partial x}|} \mid_{x_d}} + O(\epsilon)$$

Expanding the $\hat{a_g}$ using Equation 2.2 and using $\frac{\partial a_g}{\partial \theta_i}$ is locally Lipschitz continuous

$$= \frac{|\frac{\partial a_g}{\partial x}| \mid_{x_d}}{\frac{|a_g^+ - a_g^-|}{(\alpha+\beta)\epsilon}} + O(\epsilon)$$

$$= \frac{|\frac{\partial a_g}{\partial x}| \mid_{x_d}}{\frac{|\frac{\partial a_g}{\partial x} \mid_{x_d}((\alpha+\beta)\epsilon + O(\epsilon^2))|}{(\alpha+\beta)\epsilon}} + O(\epsilon)$$

First order Taylor expansion on a_g around x_d

$$= \frac{|\frac{\partial a_g}{\partial x}| \mid_{x_d}}{|\frac{\partial a_g}{\partial x}| \mid_{x_d} + O(\epsilon)} + O(\epsilon)$$

$$= 1 + O(\epsilon) \tag{2.40a}$$

Moving $O(\epsilon)$ from denominator to numerator: similar to Equation 2.36b

Case 2: g is discontinuous at $(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$. This indicates $b_g \neq 0$. 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 $\tau(x, \vec{\boldsymbol{\theta}})$, 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 $f = H(a_g + b_g \cdot H(c)) = \text{sign}(b_g) \cdot H(c)$. The reference gradient is computed as follows.

$$\begin{aligned}
\frac{\partial \hat{f}}{\partial \theta_i} =& \frac{\partial}{\partial \theta_i} \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} \text{sign}(b_g) \cdot H(c) dx' \\
=& \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} \text{sign}(b_g) \delta(c) \frac{\partial c}{\partial \theta_i} dx' \\
=& \frac{\text{sign}(b_g) \frac{\partial c}{\partial \theta_i}}{(\alpha+\beta)\epsilon|\frac{\partial c}{\partial x}|} \mid_{x_d}
\end{aligned} \tag{2.41}$$

Similarly, the reference gradient of g can be computed.

$$\begin{aligned}\frac{\partial \hat{g}}{\partial \theta_i} =& \frac{\partial}{\partial \theta_i} \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} (a_g + b_g \cdot H(c)) dx' \\ =& \frac{1}{(\alpha+\beta)\epsilon} \frac{b_g \frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|} \mid_{x_d} \\ & a_g, b_g \text{ are constants}\end{aligned} \tag{2.42}$$

We now apply our gradient approximation and start from the LHS of Equation 2.14 to show its RHS.

$$\begin{aligned}\frac{\frac{\partial_O f}{\partial \theta_i}}{\frac{\partial \hat{f}}{\partial \theta_i}} =& \frac{\frac{\partial_O g}{\partial \theta_i} / |g^+ - g^-|}{\frac{\text{sign}(b_g)\frac{\partial c}{\partial \theta_i}}{(\alpha+\beta)\epsilon|\frac{\partial c}{\partial x}|} \mid_{x_d}} \\ & \text{Using Equation 2.41} \\ =& \frac{\frac{\partial_O g}{\partial \theta_i} \frac{\partial \hat{g}}{\partial \theta_i} / (|g^+ - g^-| \frac{\partial \hat{g}}{\partial \theta_i})}{\frac{\text{sign}(b_g)\frac{\partial c}{\partial \theta_i}}{(\alpha+\beta)\epsilon|\frac{\partial c}{\partial x}|} \mid_{x_d}} \\ =& \frac{\frac{\partial \hat{g}}{\partial \theta_i}(1 + O(\epsilon)/|g^+ - g^-|}{\frac{\text{sign}(b_g)\frac{\partial c}{\partial \theta_i}}{(\alpha+\beta)\epsilon|\frac{\partial c}{\partial x}|} \mid_{x_d}} \\ & \text{Using Equation 2.14 to } g. \\ =& \frac{\frac{1}{(\alpha+\beta)\epsilon} \frac{b_g \frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|} \mid_{x_d} (1 + O(\epsilon))/|g^+ - g^-|}{\frac{\text{sign}(b_g)\frac{\partial c}{\partial \theta_i}}{(\alpha+\beta)\epsilon|\frac{\partial c}{\partial x}|} \mid_{x_d}} \\ & \text{Using Equation 2.42} \\ =& \frac{\text{sign}(b_g) \cdot b_g \mid_{x_d}}{|g^+ - g^-|} + O(\epsilon) \\ =& \frac{\text{sign}(b_g) \cdot b_g}{|a_g + b_g - a_g|} + O(\epsilon) \\ & \text{Assuming } H(c^-) = 0 \text{ and } H(c^+) = 1. \\ =& 1 + O(\epsilon)\end{aligned} \tag{2.43a}$$

$b_g \neq 0$ because discontinuity degeneracy is excluded from $\bar{e}_c$.

Combining both cases, we can now claim f is relatively first-order correct. ■

2.8.6.5 Induction on Generic Function Composition

Lemma 16 *Given a function $f = h(g)$ with $f, g \in \bar{e}_c$ and h being a continuous unary function, $\forall(x, \vec{\boldsymbol{\theta}}) \in C_i$ that are C-simple for f and $\forall\epsilon \in (0, \epsilon_f(x, \vec{\boldsymbol{\theta}})]$, if g is absolutely first-order correct, then f is absolutely first-order correct.*

Proof: Similar to previous proofs, we can first apply local expansion to g and compute f from it. Because $(x, \vec{\boldsymbol{\theta}}) \in C_i$, g is continuous at $(x, \vec{\boldsymbol{\theta}})$ and therefore $b_g = 0$.

$$\begin{aligned} g =& a_g \quad a_g \in e_a \\ f =& h(a_g) \end{aligned} \tag{2.44}$$

Because our compiler applies static analysis to g and chooses between one of the two composition rules, we will discuss them separately.

Case 1: $h(g)$ is statically differentiable. This implies g is statically differentiable as well.

$$\begin{aligned} \frac{\partial_O f}{\partial\theta_i} =& h'(g)\frac{\partial_k g}{\partial\theta_i} \\ =& h'(g)\frac{\partial\hat{g}}{\partial\theta_i} + O(\epsilon) \end{aligned} \tag{2.45a}$$

Using Equation 2.13 to g and $h'(g)$ bounded

$$= \frac{1}{(\alpha+\beta)\epsilon} h'(g) \int_{x-\alpha\epsilon}^{x+\beta\epsilon} \frac{\partial g}{\partial\theta_i} dx' + O(\epsilon)$$

Expanding $\hat{g}$ using Equation 2.2

$$= \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} h'(g)\frac{\partial g}{\partial\theta_i} dx' + O(\epsilon)$$

Using $g = a_g$ is locally Lipschitz continuous

$$
\begin{aligned}
&= \frac{1}{(\alpha+\beta)\epsilon}\int_{x-\alpha\epsilon}^{x+\beta\epsilon}\frac{\partial h(g)}{\partial \theta_i}dx' + O(\epsilon) \\
&= \frac{\partial \hat{f}}{\partial \theta_i} + O(\epsilon)
\end{aligned}
$$

Case 2: $h(g)$ is not statically differentiable.

$$
\begin{aligned}
\frac{\partial_O f}{\partial \theta_i} &= \frac{h(g^+) - h(g^-)}{g^+ - g^-}\frac{\partial_k g}{\partial \theta_i} \\
&= \frac{h'(g)\frac{\partial^n g}{\partial x}[(\alpha+\beta)\epsilon]^n + O(\epsilon^{n+1})}{\frac{\partial^n g}{\partial x}[(\alpha+\beta)\epsilon]^n + O(\epsilon^{n+1})} \cdot \frac{\partial_k g}{\partial \theta_i}
\end{aligned}
$$

Applying Taylor expansion and assuming $\dfrac{\partial^k g}{\partial x} = 0 \;\forall\; k < n$

$n > 0$ exists because $\bar{e}_c$ excludes part dependency on x

$$
= h'(g)\frac{\partial_k g}{\partial \theta_i} + O(\epsilon)
$$

$O(\epsilon)$ in the denominator can be removed per the third requirement of Lemma 8

and because symmetric along the sampling axis is excluded from C_i

$$
= h'(g)\frac{\partial \hat{g}}{\partial \theta_i} + O(\epsilon) \tag{2.46a}
$$

Using $h'(g)$ is bounded and applying Equation 2.13 to g

$$
= \frac{\partial \hat{f}}{\partial \theta_i} + O(\epsilon)
$$

Following the proof in case 1 starting from Equation 2.45a

Combining both cases, we can now claim f is absolutely first-order correct. ■

Lemma 17 *Given a function $f = h(g)$ with $f, g \in \bar{e}_c$ and h being a continuous unary function, $\forall r \in (0,1], \forall (x, \vec{\boldsymbol{\theta}}) \in D_i^r$ that are D-simple for f, if g is first-order correct for $\epsilon = r\epsilon_f(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$, then f is relatively first-order correct for the same ϵ.*

Proof: Because h is a continuous unary operator, $(\tau(x,\vec{\boldsymbol{\theta}}),\vec{\boldsymbol{\theta}})$ is discontinuous on $f = h(g)$ indicates it's discontinuous on h. Therefore, when applying local expansion to g as in Equation 2.47, we have $b_g \neq 0$ and $\frac{\partial c}{\partial \theta_i} \neq 0$.

$$\begin{aligned} g =& a_g + b_g \cdot H(c) \quad a_g, b_g, c \in e_a \\ f =& h(a_g) + (h(a_g + b_g) - h(a_g)) \cdot H(c) \end{aligned} \tag{2.47}$$

Next, we can compute the reference pre-filtered gradient for f.

$$\begin{aligned} \frac{\partial \hat{f}}{\partial \theta_i} =& \frac{\partial}{\partial \theta_i} \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} [h(a_g) + (h(a_g+b_g) - h(a_g))H(c)]dx' \\ =& \frac{1}{(\alpha+\beta)\epsilon} \int_{x-\alpha\epsilon}^{x+\beta\epsilon} (\frac{\partial h(a_g)}{\partial \theta_i} + \frac{\partial (h(a_g+b_g) - h(a_g))}{\partial \theta_i} H(c)dx' + \frac{h(a_g+b_g) - h(a_g)\frac{\partial c}{\partial \theta_i}}{(\alpha+\beta)\epsilon|\frac{\partial c}{\partial x}|} \mid_{x_d} \end{aligned}$$

Applying scaling and sifting property to the Dirac delta

$$\begin{aligned} =& \frac{1}{(\alpha+\beta)\epsilon} (\frac{\partial h(a_g)}{\partial \theta_i}(\alpha+\beta)\epsilon + \frac{\partial (h(a_g+b_g) - h(a_g))}{\partial \theta_i} \cdot x + \beta\epsilon - x_d + O(\epsilon)) \\ &+ \frac{h(a_g+b_g) - h(a_g)\frac{\partial c}{\partial \theta_i}}{(\alpha+\beta)\epsilon|\frac{\partial c}{\partial x}|} \mid_{x_d} \end{aligned}$$

Gradients of $h(a_g)$ and $h(a_g+b_g)$ are locally Lipschitz continuous

$$= \frac{1}{(\alpha+\beta)\epsilon}(O(\epsilon) + O(\epsilon) + O(\epsilon)) + \frac{h(a_g+b_g) - h(a_g)\frac{\partial c}{\partial \theta_i}}{(\alpha+\beta)\epsilon|\frac{\partial c}{\partial x}|} \mid_{x_d}$$

Gradients of $h(a_g)$ and $h(a_g+b_g)$ are locally bounded

$$= \frac{1}{(\alpha+\beta)\epsilon}(\frac{h(a_g+b_g) - h(a_g)\frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|} \mid_{x_d} + O(\epsilon)) \tag{2.48a}$$

We are now ready to expand the left-hand side of Equation 2.14.

$$\frac{\frac{\partial_O f}{\partial \theta_i}}{\frac{\partial \hat{f}}{\partial \theta_i}} = \frac{\frac{h(a_g^+ + b_g^+) - h(a_g^-)}{a_g^+ + b_g^+ - a_g^-} \frac{\partial_k g}{\partial \theta_i} \frac{\partial \hat{g}}{\partial \theta_i} / \frac{\partial \hat{g}}{\partial \theta_i}}{\frac{1}{(\alpha+\beta)\epsilon}(\frac{h(a_g+b_g) - h(a_g)\frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|} \mid_{x_d} + O(\epsilon))}$$

Using Equation 2.48a and assuming $H(c^-) = 0$, $H(c^+) = 1$

$$= \frac{\frac{h(a_g^+ + b_g^+) - h(a_g^-)}{a_g^+ + b_g^+ - a_g^-} \frac{1}{(\alpha+\beta)\epsilon} \left(\frac{b_g \frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|} \Big|_{x_d} + O(\epsilon) \right)(1 + O(\epsilon))}{\frac{1}{(\alpha+\beta)\epsilon} \left(\frac{h(a_g + b_g) - h(a_g) \frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|} \Big|_{x_d} + O(\epsilon) \right)}$$

Using Equation 2.35b and Equation 2.14 on g

$$= \frac{\left(\frac{h(a_g + b_g) - h(a_g)}{b_g + O(\epsilon)} \frac{b_g \frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|} \right) \Big|_{x_d}}{\frac{h(a_g + b_g) - h(a_g) \frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|} \Big|_{x_d}} + O(\epsilon)$$

Moving $O(\epsilon)$ from denominator to numerator: similar to Equation 2.36b

a_g, b_g are locally Lipschitz continuous

$$= \frac{\left(\frac{h(a_g + b_g) - h(a_g)}{b_g} \frac{b_g \frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|} \right) \Big|_{x_d}}{\frac{h(a_g + b_g) - h(a_g) \frac{\partial c}{\partial \theta_i}}{|\frac{\partial c}{\partial x}|} \Big|_{x_d}} + O(\epsilon)$$

Moving $O(\epsilon)$ in the $b_g + O(\epsilon)$ denominator to numerator: Equation 2.36b and $b_g \neq 0$

$$= 1 + O(\epsilon) \tag{2.49a}$$

Based on Equation 2.49a, we can conclude f is relatively first-order correct. ■

2.8.6.6 Induction Conclusions

As discussed in Section 2.4.1, the program set e_a is statically continuous, therefore for any $f \in e_a$, $C_i = \text{dom}(f)$ 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 $(x, \vec{\boldsymbol{\theta}}) \in C_i$ with $\epsilon \in (0, \epsilon_f]$, and is therefore absolutely first-order correct. The almost everywhere results of Lemma 3 and Lemma 5 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 $\bar{e}_c$ and in Theorem 2 for $\bar{e}_b$ and $\bar{e}_c$.

2.8.6.7 First-order Correctness Proof for Ternary Select in Section 2.5.2.1

The ternary select operator introduced in Section 2.5.2.1 is not a primitive operator defined in our DSL (Section 2.4), but rather an extension that improves the efficiency of the gradient program. Nevertheless, in this section, we also use a similar induction step to show it has equivalent first-order correctness properties.

Lemma 18 *Given a function $f = select(p > 0, l, r)$ with $f, p, l, r \in \bar{e}_c$, $\forall (x, \vec{\boldsymbol{\theta}}) \in C_i$ that are C-simple and $\forall \epsilon \in (0, \epsilon_f(x, \vec{\boldsymbol{\theta}})]$, if p, l, r are absolutely first-order correct, then f is absolutely first-order correct.*

Proof: as discussed in Section 2.5.2.1, our ternary rule for the select operator $f = select(p > 0, l, r)$ is equivalent to differentiating the expansion $f = r + (l - r) \cdot H(p)$ using a biased multiplication rule instead (Equation 2.12). Because computations for all the other operators (addition and Heaviside step function) are proven to be first-order correct inductively in Section 2.8.6, we only need to prove that the multiplication $(l - r) \cdot H(p)$ is absolutely first-order correct when differentiated using ∂_T in Equation 2.12 instead. To reuse part of the proof from Section 2.8.6.3, we will change our notations and denote $g = l - r$ and $h = H(p)$.

Because f is C-simple at $(x, \vec{\boldsymbol{\theta}})$, it implies that g and h are C-simple and therefore continuous at the point. We could apply local expansion and rewrite the two terms into the following. Note because $h = H(p)$ and it is continuous, it can only evaluate to either 0 or 1.

$$\begin{aligned} g =& a_g \quad a_g \in e_a \\ h =& a_h \quad a_h \in \{0, 1\} \end{aligned} \tag{2.50}$$

$$\frac{\partial_T g \cdot h}{\partial \theta_i} = h \frac{\partial_k g}{\partial \theta_i} + g_n \frac{\partial_k h}{\partial \theta_i}$$

$$=a_h\frac{\partial \hat{g}}{\partial \theta_i} + a_g\frac{\partial \hat{h}}{\partial \theta_i} + O(\epsilon) \tag{2.51a}$$

Using Equation 2.13 and Equation 2.50 and a_g is locally Lipschitz continuous

$$=\frac{\partial g \hat{\cdot} h}{\partial \theta_i} + O(\epsilon)$$

Following the proof in Section 2.8.6.3 starting from Equation 2.31a

We can now conclude $g \cdot h$ is absolutely first-order correct, therefore inductively f is absolutely first-order correct. ■

Lemma 19 *Given a function $f = select(p > 0, l, r)$ with $f, p, l, r \in \bar{e}_c$, $\forall r \in (0, 1], \forall (x, \vec{\boldsymbol{\theta}}) \in D_i^r$ that are D-simple, if p, l, r are first-order correct for $\epsilon = r\epsilon_f(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$, then f is relatively first-order correct for the same ϵ.*

Proof: similar to Lemma 18, we only need to prove the multiplication $(l - r) \cdot H(p)$ is relatively first-order correct when differentiated using ∂_T in Equation 2.12. We adopt the similar notation $g = l - r$ and $h = H(p)$, and apply local expansion as follows. Note a_h, b_h are constants because $h = H(p)$ is piecewise constant. Also note we use c to represent the discontinuity in g and h to be consistent with the notations in Section 2.8.6.3. However, if $h = H(p)$ is discontinuous at $(\tau(x, \vec{\boldsymbol{\theta}}), \vec{\boldsymbol{\theta}})$, this implies ∇p and ∇c are linearly dependent because $f \in \bar{e}_c$.

$$\begin{aligned} g =& a_g + b_g \cdot H(c) \quad && a_g, b_g, c \in e_a \\ h =& a_h + b_h \cdot H(c) \quad && a_h, b_h \text{ are constants} \end{aligned} \tag{2.52}$$

The reference pre-filtered gradient of $g \cdot h$ is already computed in Equation 2.34c. Similar to Section 2.8.6.3, we split our relatively first-order proof into two cases: only one of g or h is discontinuous or both are discontinuous.

Case 1: g is discontinuous, but h is continuous at $(\tau(x,\vec{\boldsymbol{\theta}}),\vec{\boldsymbol{\theta}})$ (the same reasoning also goes with the inverse). This indicates $b_g \neq 0$ and $b_h = 0$.

$$\frac{\frac{\partial_T g\cdot h}{\partial\theta_i}}{\frac{\partial \hat{g\cdot h}}{\partial\theta_i}} = (\alpha+\beta)\epsilon \frac{a_h \frac{\partial_k g}{\partial\theta_i} + g_n \frac{\partial_k h}{\partial\theta_i}}{\frac{b_g a_h \frac{\partial c}{\partial\vec{\theta}}}{|\frac{\partial c}{\partial x}|}\big|_{x_d} + O(\epsilon)}$$

Using Equation 2.34c and $b_h = 0$

$$= (\alpha+\beta)\epsilon \frac{a_h \frac{\partial_k g}{\partial\theta_i} + g_n \frac{\partial \hat{a}_h}{\partial\theta_i} + O(\epsilon)}{\frac{b_g a_h \frac{\partial c}{\partial\vec{\theta}}}{|\frac{\partial c}{\partial x}|}\big|_{x_d} + O(\epsilon)}$$

Using Equation 2.13 on h, $b_h = 0$ and g is locally bounded

$$= \frac{(\alpha+\beta)\epsilon a_h \frac{\partial_k g}{\partial\theta_i} + O(\epsilon)}{\frac{b_g a_h \frac{\partial c}{\partial\vec{\theta}}}{|\frac{\partial c}{\partial x}|}\big|_{x_d} + O(\epsilon)}$$

Using a_h is constant

$$= 1 + O(\epsilon)$$

Following the proof in Section 2.8.6.3 starting from Equation 2.36a

Case 2: both g, h are discontinuous at $(\tau(x,\vec{\boldsymbol{\theta}}),\vec{\boldsymbol{\theta}})$. This indicates $b_g \neq 0$ and $b_h \neq 0$. Additionally, without loss of generality, we assume $H(c_n) = 0$ and $H(c) = 1$. The inverse can be proved under the same reasoning.

$$\frac{\frac{\partial_T g\cdot h}{\partial\theta_i}}{\frac{\partial \hat{g\cdot h}}{\partial\theta_i}} = (\alpha+\beta)\epsilon \frac{(a_h+b_h)\frac{\partial_k g}{\partial\theta_i} + a_{gn}\frac{\partial_k h}{\partial\theta_i}}{\frac{(a_g b_h + b_g a_h + b_g b_h)\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\big|_{x_d} + O(\epsilon)}$$

Using Equation 2.34c and assuming $H(c_n) = 0$, $H(c) = 1$

$$= (\alpha+\beta)\epsilon \frac{(a_h+b_h)\frac{\partial_k g}{\partial\theta_i} + a_{gn}\frac{\partial_k h}{\partial\theta_i}}{\frac{(a_g b_h + b_g a_h + b_g b_h)\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\big|_{x_d}} + O(\epsilon)$$

$O(\epsilon$ in the denominator can be moved to the numerator similar to Equation 2.36b

$$=(\alpha+\beta)\epsilon\frac{(a_h+b_h)\frac{\partial_k g}{\partial\theta_i}+(a_g+O(\epsilon))\frac{\partial_k h}{\partial\theta_i}}{\frac{(a_g b_h+b_g a_h+b_g b_h)\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d}}+O(\epsilon)$$

a_g is locally Lipschitz continuous

$$=(\alpha+\beta)\epsilon\frac{(a_h+b_h)\frac{\partial_k g}{\partial\theta_i}\frac{\partial\hat{g}}{\partial\theta_i}/\frac{\partial\hat{g}}{\partial\theta_i}}{\frac{(a_g b_h+b_g a_h+b_g b_h)\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d}}+(\alpha+\beta)\epsilon\frac{(a_g+O(\epsilon))\frac{\partial_k h}{\partial\theta_i}\frac{\partial\hat{h}}{\partial\theta_i}/\frac{\partial\hat{h}}{\partial\theta_i}}{\frac{(a_g b_h+b_g a_h+b_g b_h)\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d}}+O(\epsilon)$$

$$=(\alpha+\beta)\epsilon\frac{(a_h+b_h)(1+O(\epsilon))\frac{1}{(\alpha+\beta)\epsilon}(\frac{b_g\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d}+O(\epsilon))}{\frac{(a_g b_h+b_g a_h+b_g b_h)\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d}}$$

$$+(\alpha+\beta)\epsilon\frac{(a_g+O(\epsilon))(1+O(\epsilon))\frac{1}{(\alpha+\beta)\epsilon}(\frac{b_h\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d}+O(\epsilon))}{\frac{(a_g b_h+b_g a_h+b_g b_h)\frac{\partial c}{\partial\theta_i}}{|\frac{\partial c}{\partial x}|}\Big|_{x_d}}+O(\epsilon)$$

Using Equation 2.14 and 2.35b to g, h

$$=\frac{a_h b_g+b_h b_g+a_g b_h}{a_g b_h+b_g a_h+b_g b_h}\Big|_{x_d}+O(\epsilon)$$

a_g, b_g, a_h, b_h are locally Lipschitz continuous

$$=1+O(\epsilon)$$

Combining both cases we can conclude $g \cdot h$ is relatively first-order correct with ∂_T. Therefore inductively f is relatively first-order correct. ■

2.9 Quantitative Error Metric

Section 2.7 mathematically shows the approximation error for some subsets of programs is $O(\epsilon)$. However, the first-order correctness result does not apply to *every* program, and we cannot use it to compare two methods that are both first-order correct, such as ours and TEG. Therefore, we also wish to numerically evaluate the gradient approximations.

One possibility is to compute the L^1 or L^2 distance between our gradient and the reference gradient. However, analytically writing down the reference gradient requires substantial human effort and may not be feasible depending on the complexity of the program, which is

exactly the motivation for developing approximate techniques to differentiate discontinuities. Therefore, a common approach is to use the finite difference as the proxy for the reference gradient [69]. 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. $\lim_{\epsilon\to 0} G_\epsilon(x)$ and $\lim_{\epsilon\to 0} G_{2\epsilon}(x)$. Here we use $G_\sigma(x)$ to identify the distribution for the Gaussian PDF wrt x with standard deviation σ. These distributions have nonzero L^1 and L^2 distance between each other even as $\epsilon \to 0$: $\lim_{\epsilon\to 0}(G_\epsilon(x) - G_{2\epsilon}(x)) \neq 0$. This behavior is undesirable because for two different gradient approximations with different limiting "shapes" (e.g. $G_\epsilon(x)$ and $G_{2\epsilon}(x)$), they would wrongly be considered to have nonzero error even if they both give precisely correct derivatives as limits in the distribution theory. Additionally, finite difference with finite step size and sample count introduces its own approximation error (Section 3.4).

We, therefore, avoid computing a reference gradient directly, and propose a quantitative metric according to 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, when the program parameters move from $\vec{\boldsymbol{\theta}}_0$ to $\vec{\boldsymbol{\theta}}_1$ along a differentiable curve Θ, the correct gradient should always satisfy Equation 2.55, 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).

$$\int_\Theta \nabla f(\vec{\boldsymbol{\theta}}) d\vec{\boldsymbol{\theta}} = f(\vec{\boldsymbol{\theta}}_1) - f(\vec{\boldsymbol{\theta}}_0) \tag{2.55}$$

Additionally, because the gradient theorem requires a continuously differentiable function, we need to pre-filter the discontinuous program f before applying Equation 2.55. In practice, the desired pre-filter is an m+n-dimensional box filter in both the sampling axes space and the parameter space: $\Psi^* = U[-1,1]^m \times U[-\xi,\xi]^n$. Ψ^* is a separable kernel that can be decomposed into a kernel $U[-1,1]^m$ in the sampling axes space, and a kernel $U[-\xi,\xi]^n$ in the parameter space. For example, if f is a shader program that outputs images, the sam-

pling axes space is the image space, and therefore $m = 2$. The $U[-1, 1]^2$ kernel smooths out a 3x3 region centered at the current pixel. In the parameter space, ξ is chosen such that the diagonal length of 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 can replace f in Equation 2.55 with the pre-filtered smooth function to obtain our final gradient theorem result.

$$\int_{\Theta} \nabla(\Psi^* * f(\vec{\boldsymbol{\theta}}))d\vec{\boldsymbol{\theta}} = \Psi^* * f(\vec{\boldsymbol{\theta}}_1) - \Psi^* * f(\vec{\boldsymbol{\theta}}_0) \tag{2.56}$$

We use the L1 difference between LHS and RHS in Equation 2.56 as our error metric.

$$L = |(\Psi^* * f(\vec{\boldsymbol{\theta}}_1) - \Psi^* * f(\vec{\boldsymbol{\theta}}_0)) - \int_{\Theta} \Psi_k * \nabla_k f(\vec{\boldsymbol{\theta}})d\vec{\boldsymbol{\theta}}| \tag{2.57}$$

The first term in Equation 2.57 corresponds to the RHS in Equation 2.56, and is estimated by sampling the $m + n$-dimensional box filter Ψ^* around $\vec{\boldsymbol{\theta}}_0$ and $\vec{\boldsymbol{\theta}}_1$. The second term in Equation 2.57 corresponds to the LHS in Equation 2.56, and is estimated by quadrature using the midpoint rule. Note we adopt a similar notation as in Section 2.4.2 and use ∇_k to denote *any* gradient approximation method. Also, note that we swap the convolution and the differentiation operator and use a different pre-filtering kernel Ψ_k other than Ψ^* in Equation 2.56. This is because different gradient approximations already make different pre-filtering assumptions, such as the 1D pre-filtering approximation made by our gradient. Therefore, a different kernel Ψ_k is chosen for *each* gradient approximation method such that $\Psi^* = \Psi_k * \phi_k$ where ϕ_k is the pre-filtering approximation *already* made internally by the method. We will discuss the choice of Φ_k for three gradient approximation methods: ours (O), finite difference (FD), and TEG. We will use the shader program to compare these methods in Chapter 3.

Ours: as discussed in Section 2.7, our gradient is approximating that of a pre-filtered function using a 1D box kernel $U[-1,1]$ along one of the m sampling axes. Recall that $\Psi^* = U[-1,1]^m \times U[-\xi,\xi]^n$ is a separable kernel. $U[-1,1]^m$ can be further separated into the convolution of m 1D box kernels along different sampling axes: $U[-1,1]^m = \phi_1 * ... * \phi_m$ where $\phi_i \sim U(\{0\}^{i-1} \times [-1,1] \times \{0\}^{m-i})$ is the 1D box kernel along the ith sampling axis. If we assume our gradient is pre-filtered along the ith sampling axis, we can pick $\Psi_O = \phi_1 * ... * \phi_{i-1} * \phi_{i+1} * ... * \phi_m * U(\{0\}^m \times [-\xi,\xi]^n)$ to estimate $\nabla(\Psi^* * f)$ as follows.

$$\nabla(\Psi^* * f) = \nabla(\phi_j * \Psi_O * f) = \Psi_O * \nabla(\phi_j * f) \approx \Psi_O * \nabla_O f$$

Here $\approx$ indicates ∇_O is an approximation that potentially introduces errors. In practice, because our approximation adaptively chooses between different sampling axes (Section 2.5.1), for a given evaluation at $(\vec{\boldsymbol{x}}, \vec{\boldsymbol{\theta}})$, $\Psi_O * \nabla_O f$ is computed by first deciding which axis to prefilter, then sampling along the other orthogonal axes to estimate the convolution.

FD and variants: because FD is approximated based on the original function value, we directly use $\Psi_{FD} = \Psi^*$.

TEG: because TEG users define their own integrals, Ψ_{TEG} needs to be carefully chosen so as to eliminate the pre-filtering kernels already defined in the TEG language. For example, when differentiating discontinuous shader programs that output images, the TEG program can be defined as an integral that pre-filters the image space with $U[-1,1]^2$. [5] In this case, Ψ_{TEG} can simply be the box kernel in the parameter space: $\Psi_{TEG} = U[-\xi,\xi]^n$.

[5] TEG does not handle analytic integration. After their compiler removes a single-dimensional integral over the Dirac delta using the sifting property, the rest of the integral dimensions will be approximated by sampling.

2.10 Summary and Discussion

This chapter proposes a mathematical framework to differentiate program discontinuities that are usually ignored by traditional AD frameworks. We first extend the reverse-mode backpropagation by replacing the traditional calculus gradient rules with a set of novel gradient rules that can correctly approximate the gradient of a discontinuous program pre-filtered with a 1D box kernel (Section 2.4). We further generalize our differentiation framework to various programming patterns in Section 2.5, such as allowing discontinuities of interest to be sampled from multiple dimensions, as well as improving the efficiency of the gradient for branching operators and complicated Boolean predicates. Finally, we propose two methods to verify the accuracy of our approximation. Mathematically, we formally establish the definition of correctness (Section 2.7), and prove that for a subset of programs, our approximation error is bounded by a first-order term with respect to the size of the pre-filtering kernel (Section 2.8). Numerically, we also design a novel quantitative error metric based on the gradient theorem (Section 2.9) to evaluate the approximation error for any program.

Our differentiation framework still has several limitations, which invite future exploration. Firstly, we choose the 1D box kernel as our pre-filtering to avoid evaluating the kernel exactly at the discontinuity. However, this prevents us from rigorously connecting to the distributional derivative because they require an infinitely smooth test function (i.e. pre-filtering kernel in our case). Future work could better establish our approximation error compared with distributional derivatives by extending our framework to pre-filtering with infinitely smooth compact kernels. Alternatively, one could make a connection to the theory of distributions by considering the distributional derivatives applied to non-smooth test functions as being well-defined at points where all smooth, nonzero, and compactly supported test functions that converge to the given non-smooth test function result in the same integral, and then attempting to make the connection to the reference gradient and thus to our result (which is already proved first order correct wrt the reference gradient) at such well-defined points. Secondly, 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 of the parameter space, we imagine our gradient rules could further be generalized to an arbitrary choice of sampling axis, such as a linear combination of the parameters. Section 3.3 demonstrates one such example, where we stochasticaly sample within the parameter space.

Chapter 3

Optimizing Discontinuous Shader Applications

This chapter demonstrates applying the Aδ differentiation math framework (Chapter 2) to the domain of procedural shader programs, where the image coordinate axes x and y naturally become our sampling axes. Procedural shader programs use floating point operations to flexibly describe a scene and render the visual appearance. For example, many beautiful shader programs are shared through the community `shadertoy.com`, such as the ones shown in Figure 1.1. The parameters for these shaders are carefully tuned by the programmers for best visual appearance. However, the manual tuning process is tedious and difficult, especially if we want to tune the virtual scene to exactly match a target. One approach to address this challenge is to automatically optimize the parameters through gradient descent, which requires the shader programs to be differentiable. While differentiable rendering is an active research field, most of the methods focus on differentiating specific types of discontinuities, such as affine transformation for triangle meshes [69], or polynomial curves for vector graphics [71]. The expressiveness of these specialized pipelines comes from their massive number of parameters (i.e. triangle vertices or curve control points), but the resulting scene representation is usually cumbersome and difficult to interpret. In contrast, a general program

(a) Target (b) Hand-chosen (c) Configurations

(d) Optimization (e) Modified-d (f) Hybrid-b/d

Figure 3.1: 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 project page `https://pixl.cs.princeton.edu/pubs/Yang_2022_AAF/index.php`.

representation allows the programmer to use their own judgment to abstract the scene, and is usually more compact and interpretable. For example, expressing a circle programmatically using its analytic equation is much easier to understand than approximating it using multiple segments of the Bezier curve. Therefore, this chapter utilizes Aδ's ability to differentiate arbitrary programs to optimize unknown shader program parameters to match the shader rendering with target reference images that we found on the Web. The optimization does so more effectively and quickly than using baseline methods such as finite difference, and even for programs outside the set we can formally prove first-order correct. At convergence, we also show the program representation allows us to further modify and animate the scene much more easily, Figure 3.1 shows two such examples. Additionally, while our implementation optimizes parameters from a shader program, the proposed approach can also be generalized for non-programmers. For example, non-programmers can build graph representations using tools such as Adobe Substance Designer, and the graphs can be parsed and differentiated as program representations as well.

This chapter describes the framework that carries out the optimization task for procedural shader programs. We also use the shader application to investigate how to improve the efficiency of the gradient program both algorithmically and by generating highly efficient GPU code. We first extend the Aδ DSL with another shader-specific primitive RaymarchingLoop to efficiently differentiate the raymarching loop that bypasses backpropagating every iteration of the loop (Section 3.1). Next, we describe our compiler implementation that generates the gradient program to multiple backends: TensorFlow and PyTorch for fast prototyping and debugging, and Halide with an optional auto-scheduler for efficiency (Section 3.2.1). After that, we address an optimization challenge when the discontinuity is rarely sampled by introducing random variables to our optimization process (Section 3.3). And finally, we validate our framework in Section 3.4 and compare it with baselines. Our code is available at: `https://github.com/yyuting/Adelta`.

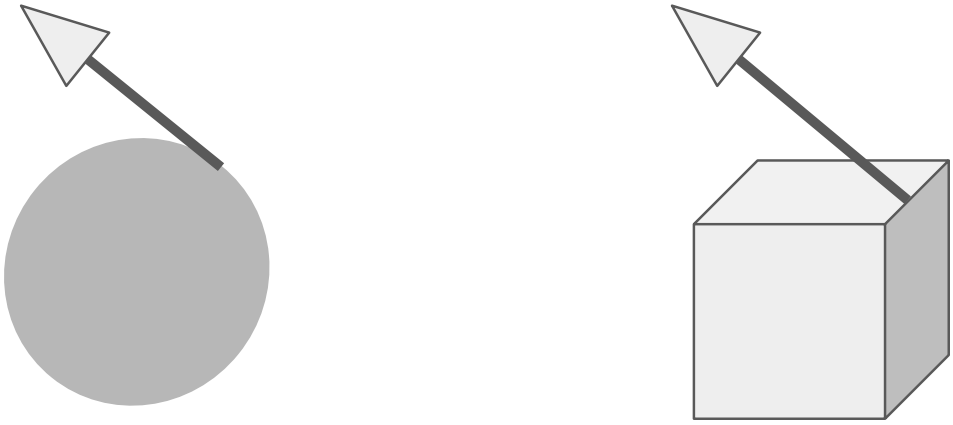
3.1 Implicitly Defined Geometry

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 [108] or sphere tracing [53]. While ray marching and sphere tracing loops themselves are programs, and can be differentiated using rules introduced in Section 2.2, this usually results in a long gradient tape because the number of loop iterations can be arbitrarily large. As an alternative, this section introduces a novel gradient approximation based on the implicit function theorem that bypasses the root-finding loop iterations.

Our rule is motivated by [152]: 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, [71] applies 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). [46] also develops a gradient rule for visibility change to the implicitly defined surface. Their derivation only applies to C1 continuous geometry (Section 3.1.2) and does not handle discontinuities caused by the intersection of surfaces (Section 3.1.3). Unlike previous methods, our rule can differentiate the discontinuities generated by implicit geometries represented as arbitrary signed distance functions.

3.1.1 The RaymarchingLoop Primitive

We extend our DSL with an additional primitive RaymarchingLoop. The programmer specifies the signed distance function (SDF) f that represents the geometry using the RaymarchingLoop primitive and the compiler automatically expands a ray marching loop to approach the zero set of the SDF. Specifically, the 3D location $\vec{\boldsymbol{p}}$ is defined implicitly from the scene



(a) C1 Continuous Geometry (§3.1.2) (b) C1 Discontinuous Geometry (§3.1.3)

Figure 3.2: Illustrating two scenarios for camera-ray intersection: (a) intersecting a locally C1 smooth geometry and (b) intersecting a locally C1 discontinuous geometry.

parameters $\vec{\boldsymbol{\theta}}$ and the SDF f:

$$f(\vec{\boldsymbol{p}}(x,y),\vec{\boldsymbol{\theta}}) = 0 \tag{3.1}$$

The 3D locations further depend on image coordinates (x, y) that restrict the 3D location $\vec{\boldsymbol{p}}$ lying on a ray casting from the camera origin o to the direction $\vec{\boldsymbol{d}}(x, y)$ with distance t:

$$\vec{\boldsymbol{p}} = \vec{\boldsymbol{o}}(x,y) + t \cdot \vec{\boldsymbol{d}}(x,y) \tag{3.2}$$

Without loss of generality, we will derive our gradient assuming x is our sampling axis and keep y fixed. Given arbitrary $\vec{\boldsymbol{\theta}}$, the geometry discontinuity can be sampled if both evaluation sites x^- and x^+ 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 evaluating to different branches of the CSG operation. For both cases, the discontinuity at x_d can be represented as $H(x - x_d)$. Note x_d is never explicitly computed, we simply use the pair of evaluation sites x^- and x^+ to sample its existence.

When differentiating the RaymarchingLoop primitive, the key is to approximate $\partial x_d/\partial\theta_i$. The compiler first samples x_d by evaluating the Boolean predicates within f, then classifies the cause of 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 Section 3.1.2 and 3.1.3.

3.1.2 Camera Ray Intersecting Locally C1 Continuous Geometry

In this case, the discontinuity is implicitly defined by the ray tangentially intersecting the zero set of the SDF f, such as in Figure 3.2(a) . The SDF f is C1 continuous within a neighborhood of the intersection, such that the derivative of f exists. The camera ray's direction is perpendicular to the normal direction of the geometry at the intersection.

$$f(\vec{\boldsymbol{p}}, \vec{\boldsymbol{\theta}}) = 0 \tag{3.3a}$$

$$< \frac{\partial f}{\partial \vec{\boldsymbol{p}}}, \vec{\boldsymbol{d}} > = 0 \tag{3.3b}$$

We can now differentiate wrt an arbitrary parameter θ_i on both sides of Equation 3.3a.

$$\frac{\partial f}{\partial \theta_i} + < \frac{\partial f}{\partial \vec{\boldsymbol{p}}}, \frac{\partial \vec{\boldsymbol{o}}}{\partial \theta_i} + \frac{\partial \vec{\boldsymbol{o}}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + \vec{\boldsymbol{d}}\frac{\partial t}{\partial \theta_i} >= 0 \tag{3.4}$$

Equation 3.4 can be simplified by inserting Equation 3.3b.

$$\frac{\partial f}{\partial \theta_i} + < \frac{\partial f}{\partial \vec{\boldsymbol{p}}}, \frac{\partial \vec{\boldsymbol{o}}}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial \theta_i} > + < \frac{\partial f}{\partial \vec{\boldsymbol{p}}}, \frac{\partial \vec{\boldsymbol{o}}}{\partial x_d} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial x_d} > \frac{\partial x_d}{\partial \theta_i} = 0 \tag{3.5}$$

Rearranging Equation 3.5 results in Equation 3.6.

$$\frac{\partial x_d}{\partial \theta_i} = -\frac{\frac{\partial f}{\partial \theta_i} + < \frac{\partial f}{\partial \vec{\boldsymbol{p}}}, \frac{\partial \vec{\boldsymbol{o}}}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial \theta_i} >}{< \frac{\partial f}{\partial \vec{\boldsymbol{p}}}, \frac{\partial \vec{\boldsymbol{o}}}{\partial x_d} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial x_d} >} \tag{3.6}$$

Because we assume f is locally C1 continuous, the derivatives on the right hand side of Equation 3.6 are computed using traditional AD rules. The raymarching dependent parameters, such as f, t, and $\vec{\boldsymbol{p}}$ represent their corresponding evaluations from the last raymarching iteration of the forward pass.

In our implementation, the compiler identifies every discontinuous branching $g = \texttt{select}(c, a, b)$ where c depends on the RaymarchingLoop primitive, and back-propagates g as if the discontinuity is caused by the two samples x^+ and x^- lying on two sides of the sihoulette intersection x_d: $g = \texttt{select}(x - x_d, a, b)$, which could be easily differntiated using $\partial x_d / \partial x_i$ computed from Equation 3.6.

3.1.3 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, such as the example in Figure 3.2(b). The derivation in this section assumes we already know the intersection is caused by the two C1 continuous surfaces f_0 and f_1. Section 3.1.4 further discusses how to efficiently identify f_0 and f_1.

We start by defining the intersection as the set of points on the zero set for both f_0 and f_1.

$$\begin{aligned} f_0(\vec{\boldsymbol{p}}, \vec{\boldsymbol{\theta}}) &= 0 \\ f_1(\vec{\boldsymbol{p}}, \vec{\boldsymbol{\theta}}) &= 0 \end{aligned} \tag{3.7}$$

We now differentiate wrt θ_i to both equations in Equation 3.7 .

$$\begin{aligned}
\frac{\partial f_0}{\partial \theta_i} + &< \frac{\partial f_0}{\partial \vec{\boldsymbol{p}}}, \frac{\partial \vec{\boldsymbol{o}}}{\partial \theta_i} + \frac{\partial \vec{\boldsymbol{o}}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + \vec{\boldsymbol{d}}\frac{\partial t}{\partial \theta_i} >= 0 \\
\frac{\partial f_1}{\partial \theta_i} + &< \frac{\partial f_1}{\partial \vec{\boldsymbol{p}}}, \frac{\partial \vec{\boldsymbol{o}}}{\partial \theta_i} + \frac{\partial \vec{\boldsymbol{o}}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + \vec{\boldsymbol{d}}\frac{\partial t}{\partial \theta_i} >= 0
\end{aligned} \tag{3.8}$$

Next, we can rearrange Equation 3.8 to separate $\frac{\partial t}{\partial \theta_i}$.

$$\begin{aligned}
\frac{\partial t}{\partial \theta_i} &= -\frac{\frac{\partial f_0}{\partial \theta_i} + < \frac{\partial f_0}{\partial \vec{\boldsymbol{p}}}, \frac{\partial \vec{\boldsymbol{o}}}{\partial \theta_i} + \frac{\partial \vec{\boldsymbol{o}}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} >}{< \frac{\partial f_0}{\partial \vec{\boldsymbol{p}}}, \vec{\boldsymbol{d}} >} \\
\frac{\partial t}{\partial \theta_i} &= -\frac{\frac{\partial f_1}{\partial \theta_i} + < \frac{\partial f_1}{\partial \vec{\boldsymbol{p}}}, \frac{\partial \vec{\boldsymbol{o}}}{\partial \theta_i} + \frac{\partial \vec{\boldsymbol{o}}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial x_d}\frac{\partial x_d}{\partial \theta_i} >}{< \frac{\partial f_1}{\partial \vec{\boldsymbol{p}}}, \vec{\boldsymbol{d}} >}
\end{aligned} \tag{3.9}$$

Note the left-hand sides of both equations in Equation 3.9 are identical. This implies their right-hand sides are identical as well. After rearranging the terms, we derive $\frac{\partial x_d}{\partial \theta_i}$ as in Equation 3.10.

$$\frac{\partial x_d}{\partial \theta_i} = \frac{(\frac{\partial f_1}{\partial \theta_i} + < \frac{\partial f_1}{\partial \vec{\boldsymbol{p}}}, \frac{\partial \vec{\boldsymbol{o}}}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial \theta_i} >) < \frac{\partial f_0}{\partial \vec{\boldsymbol{p}}}, \vec{\boldsymbol{d}} > - (\frac{\partial f_0}{\partial \theta_i} + < \frac{\partial f_0}{\partial \vec{\boldsymbol{p}}}, \frac{\partial \vec{\boldsymbol{o}}}{\partial \theta_i} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial \theta_i} >) < \frac{\partial f_1}{\partial \vec{\boldsymbol{p}}}, \vec{\boldsymbol{d}} >}{< \frac{\partial f_0}{\partial \vec{\boldsymbol{p}}}, \frac{\partial \vec{\boldsymbol{o}}}{\partial x_d} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial x_d} >< \frac{\partial f_1}{\partial \vec{\boldsymbol{p}}}, \vec{\boldsymbol{d}} > - < \frac{\partial f_1}{\partial \vec{\boldsymbol{p}}}, \frac{\partial \vec{\boldsymbol{o}}}{\partial x_d} + t\frac{\partial \vec{\boldsymbol{d}}}{\partial x_d} >< \frac{\partial f_0}{\partial \vec{\boldsymbol{p}}}, \vec{\boldsymbol{d}} >} \tag{3.10}$$

3.1.4 Efficient Backpropagation

One major challenge for efficiently implementing the gradient discussed in Section 3.1.2 and 3.1.3 is to efficiently identify the cause of the discontinuity at runtime, i.e. whether the ray is tangential to a smooth surface, or it reaches the intersection of two arbitrary sub-surfaces. Consider the example when an implicit function f is defined by applying CSG operations to n C1 smooth sub-functions and assume the discontinuity is caused by sub-surface intersection. Statically writing out Equation 3.10 results in enumerating $\binom{n}{2}$ possible combinations of f_0 and f_1 because we do not know which two surfaces may intersect until runtime. Therefore the naive implementation would have $O(n^2)$ complexity.

This section discusses a more efficient algorithm based on our observations for implicit scene representations. Note that we make heuristic assumptions about 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 iterative root-finding process using the gradient rules described in Section 2.4.2.

Our compiler assumes that the branching decision in the implicit functions is always represented as min or max, and all values being compared are scaled similarly (e.g. they all represent the Euclidean distance in the scene space). This assumption holds for most implicit scene representations, such as all the signed distance fields and the CSG operators described in [56].

We define geometry discontinuity as any intermediate nodes $g = H(h)$ of f where g depends on the RaymarchingLoop primitive. The compiler identifies every geometry discontinuity and back-propagates accordingly (Section 3.1.4.1). We further classify at runtime whether the discontinuity is caused by tangential ray intersection or not (Section 3.1.4.2). For discontinuity caused by intersecting sub-surfaces, we further dynamically modify the branching condition in the SDF f to evaluate f_0 and f_1 at runtime (Section 3.1.4.3).

3.1.4.1 Sampling geometry discontinuities

We use the newly introduces primitive RaymarchingLoop to identify geometry discontinuities. It is defined using the camera orientation $\vec{\boldsymbol{o}}$, ray direction $\vec{\boldsymbol{d}}$ and the user-defined SDF f. Its output includes a Boolean condition B on whether the raymarching loop has diverged, as well as the distance t from the camera to the geometry intersection. Optionally, the SDF can define the surface normal and sub-surface labeling that shares the branching with the original SDF. This allows the RaymarchingLoop primitive to output surface normals or surface labeling evaluated at $\vec{\boldsymbol{p}} = \vec{\boldsymbol{o}} + t \cdot \vec{\boldsymbol{d}}$ as well.

In the forward pass, the compiler automatically expands the primitive into a loop that iteratively finds the zero set to the implicit function using ray marching.

In the backward pass, we sample the silhouette discontinuity by evaluating B. The interior edge discontinuity is sampled based on whether any branching condition defined in f evaluates to different values for the two evaluation sites. Recall that any geometry discontinuity sampled at x_d will be represented as $H(x - x_d)$ where x_d is never explicitly computed, but its gradient can be computed implicitly. Therefore, in the backward pass, the compiler backpropagates any geometry discontinuity $H(h)$ as if it was in the form $H(x - x_d)$. Further, we also apply a similar representation to the output of the RaymarchingLoop primitive, which does not necessarily depend on explicit Heaviside step function or `select` operators. For example, the distance t from the camera to geometry can be represented as $t = \texttt{select}\ (x > x_d, t^-, t^+)$ when the geometry discontinuity is sampled. However, note this sampling method would fail to sample one type of discontinuity: when an interior edge is caused by a tangential ray to the foreground geometry without changing any Boolean conditions within f.

3.1.4.2 Classifying geometry discontinuities

For geometry discontinuities, the compiler applies Section 3.1.2 if the camera ray is tangent to the geometry and Section 3.1.3 otherwise. We use a simple classification that works well in all our experiments: a ray is tangent to the geometry if $\langle \partial f / \partial \vec{\boldsymbol{p}}, \vec{\boldsymbol{d}} \rangle \leq 0.1$.

3.1.4.3 Identifying intersecting sub-surfaces

If the compiler classifies the geometry discontinuity as being caused by sub-surface intersection between f_0 and f_1, our algorithm modifies the branching condition in f so that evaluating Equation 3.10 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 a different branch that leads to 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 intersect with f_1 instead.

To correctly modify the branching, our compiler iterates through every branching condition within f and finds the one condition $\mathbf{c}^* = \min(a, b)$ (or max) with minimum absolute difference between the two sides of the comparison a and b (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, we invert every condition where a is chosen to divert f to the other sub-surface f_1.

3.2 Backend Implementation

Because procedural shader programs are usually evaluated over a regular pixel grid, where the workload is embarrassingly parallel, it is important to allow the gradient program fully utilize the GPU. Our compiler outputs a gradient program to three 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 3.2.1 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

Table 3.1: Our compiler provides a simplified Halide scheduling space for a program f. For an explanation of each choice refer to Section 3.2.1.

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}

the system's API. For example, although DVG [71] 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 fourth 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`.

3.2.1 Halide Scheduling

A key challenge for generating an efficient reverse-mode gradient program is to manage the large number of intermediate values computed in the forward pass (i.e. gradient tape). Unfortunately, most state-of-the-art Halide autoschedulers focus on the forward pass only, such as efficiently scheduling the nested image pipelines and utilizing scheduling choices such as tiling and fusion [91, 2, 120]. While Li et al. [70] proposed some scheduling options for the gradient program of image pipelines, their work focuses on efficiently scheduling convolutional scattering and gathering operations (e.g. convolution and its gradient). But in our case, because procedural shader programs compute independently per pixel, our scheduling bottleneck is instead the limited GPU register count per thread, which causes the trade-off between avoiding register spilling and minimizing memory I/O in the presence of long gradient tapes.

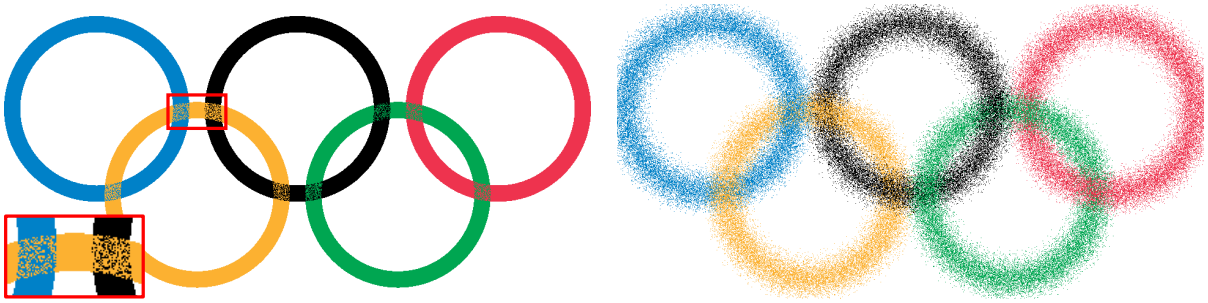
On one hand, assuming unlimited register space, inlining the entire program into one kernel without intermediate checkpointing gives the optimal performance as memory access is minimized. On the other hand, assuming unlimited memory, writing to and reading from memory for every intermediate computation 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. However, 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 potentially leads to register spilling) or checkpointing (which requires 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 [27, 50]. However, generalizing those methods to arbitrary compute graphs 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 an arbitrary compute graph with variable terminal nodes, which is NP-hard [140]. It is nontrivial to adopt classic graph cuts literature (e.g. [14]) to this problem due to the interaction with the complex engineering of the lower-level register allocator and the hardware register space limits.

Therefore, this section describes heuristic-based scheduling options summarized in Table 3.1. 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 later in Section 3.4. For each intermediate value in the forward pass, we decide on checkpointing vs recomputation and encode the list of checkpoint nodes in the logged_trace. For the space of splitting the backward pass into sub-programs, we reduce it to a combination of the discrete choices in Table 3.1. Because the gradient wrt non-Dirac parameters can be computed with traditional AD, it usually results in a smaller gradient kernel, which can be optionally computed in a separate reverse-mode AD without any Aδ rules (separate_cont = True). Additionally, since this traditional AD sub-program is typically smaller, it may have extra register space for recomputing values from the forward pass and save some memory I/O from checkpointing. Therefore, instead of reading checkpointing values from the logged_trace, the gradient to non-Dirac parameters reads checkpoints from cont_logged_trace, which is a strict subset of the 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 2.5). Therefore, we can compute the gradient approximation to each sampling axis in a different sub-program by setting separate_sample = axis, resulting two sub-programs, each computing the left and right pre-filtering kernel for either the x or y axis. Alternatively, we can separate the approximation to each pre-filtering kernel into different parts with separate_sample = kernel, resulting four sub-programs, each responsible for computing the gradient from one of the pre-filtering 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 forward computation from each node to its checkpointed children. Based on the model, we iteratively add nodes to a potential checkpoint list using 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 checkpointing list



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

Figure 3.3: 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.

before every iteration of the greedy search. The list is then combined with discrete options in Table 3.1 to search for a schedule with the 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 = [] 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 the best runtime.

3.3 Random Variables for Sampling Discontinuities

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 2.5(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 interlocking pattern is wrong in Figure 2.5(d).

We propose to solve this problem by introducing auxiliary random variables. Conceptually, we can think of the random variables as extending the space where we sample discontinuities from only the 2D spatial coordinates to a higher-dimensional space that includes both the 2D spatial parameters and the Dirac parameters. For each Dirac parameter (with exception of those discussed next in Section 3.3.1), 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 3.3(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 3.3(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 ϵ 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 and 2 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 (Section 3.1) because assumptions made in its gradient derivation can be violated by random variables. We explain this further in Section 3.3.1. 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.

3.3.1 Caveat with the RaymarchingLoop Primitive

Our gradient rules introduced in Section 2.4.2 and 2.5.2 can easily generalize to random variables introduced in Section 3.3 because the random generation process can be viewed as a high-frequency black box function that we never need to differentiate through. All of our rules about sampling the discontinuities stay valid.

However, the random variables 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 assumes 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 Section 3.1.2 and 3.1.3. For these reasons we always disable the random variables for Dirac parameters that the RaymarchingLoop primitive depends upon.

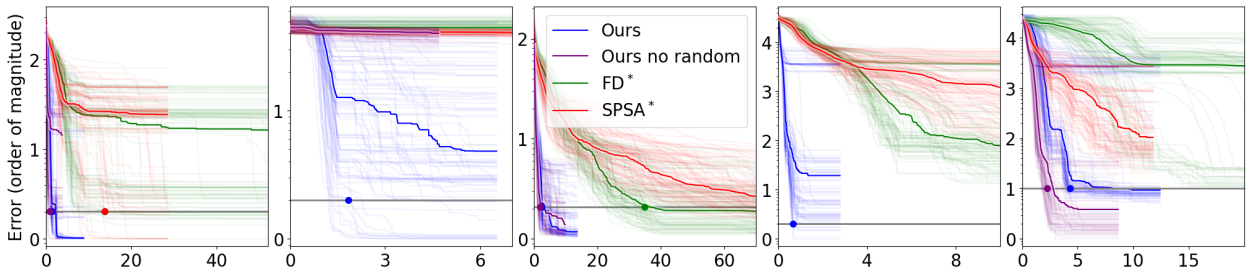
It is possible to combine random variables with implicitly defined geometry by resorting to the general gradient rules introduced in Sections 2.4.2 and 2.5.2. Because the ray marching loop can have arbitrarily many iterations, the gradient program can be inefficient due to the long tape. Note that for simple geometries, it is also possible to analytically compute ray object intersection and therefore avoid the loop iterations.

3.4 Evaluation and Results

3.4.1 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 3.3) 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.



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

Figure 3.4: Runtime-Error plot for five optimization tasks, comparing Ours (with and without random variables) with FD* and SPSA*. 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 an 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. Note we select FD* and SPSA* based on the minimal error from among 10 and 30 different variants, as discussed in Section 3.4.1. Grey horizontal lines denote the success threshold, while circles on grey lines mark the median success time as in Table 3.2.

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 3.4. Based on this definition of success, we compute two metrics: median success time and expected time to success, and report these in Table 3.2. The median success time is

Table 3.2: Time metrics in seconds comparing how fast ours, ours without random variables (O/wo) and baselines converge, as discussed in Section 3.4.1. 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	×

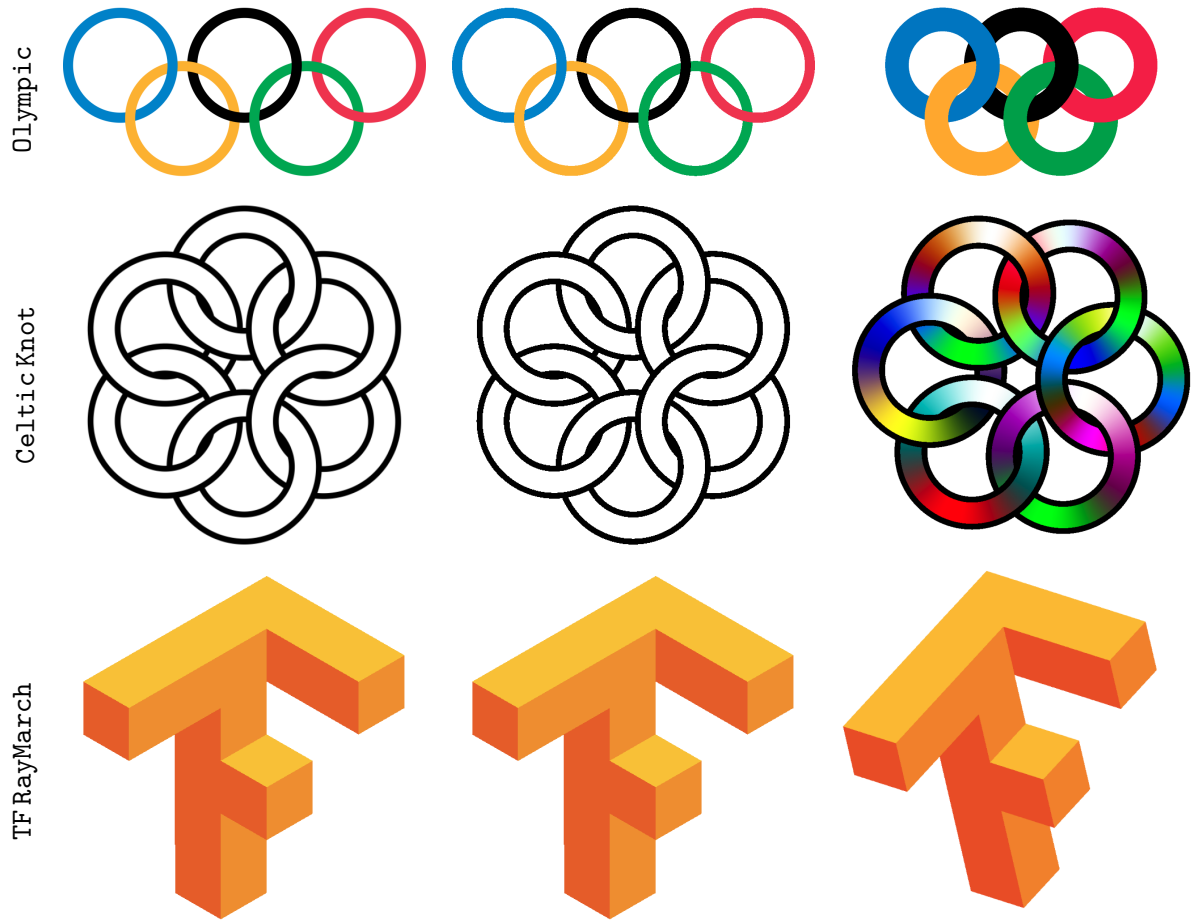
the median time taken for a restart to reach the success threshold across all restarts that succeed. These values are plotted as colored circles on the grey line in Figure 3.4. Expected time to success, on the other hand, evaluates given sufficient restarts, the mean time until the optimization finally converges. It is computed by repeatedly sampling with replacement from the 100 restarts and accumulating their runtime until a sampled restart converges.

Because of the arbitrary composition patterns present in the shader programs, none of the shaders presented in this section can 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 [125]. 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 include 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.

3.4.1.1 Shader: `OlympicRings`

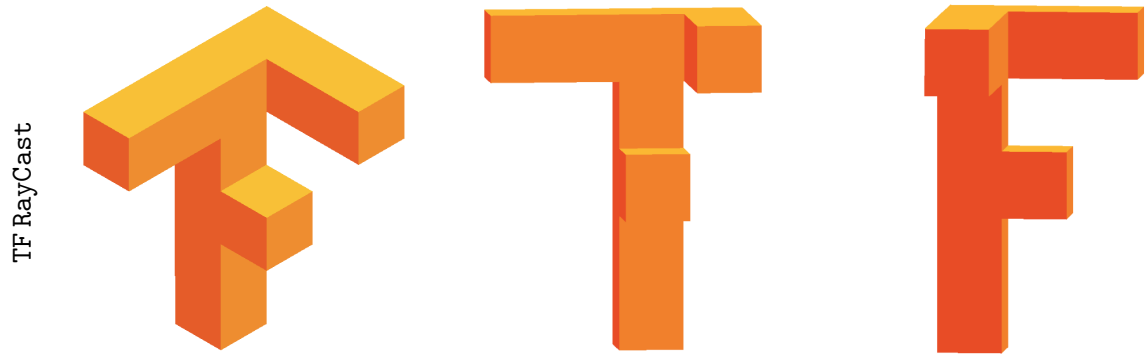
We design a shader to express the Olympic logo [1] shown in Figure 3.5(a)-top. The shader uses more rings than are necessary (10) to avoid being stuck in a local minimum. Each

[1]The Olympic rings are the exclusive property of the International Olympic Committee (IOC).



(a) Target (b) Optimization (c) Modified

Figure 3.5: Target, optimization, and modified results for three shaders discussed in Section 3.4.1: `OlympicRings`, `CelticKnot` and `TFRayMarch`.



(a) Optimization (b) Modified (c) Modified

Figure 3.6: Optimization and modifications for the shader `TFRayCast`, using target from Figure 3.5(a). Section 3.4.1.5 discusses the differences between `TFRayCast` and `TFRayMarch`.

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. In DVG, each connected component of the ring can still be represented by filled shapes. But as we discussed before, such parameterization does not have semantic meaning, increasing the difficulty for future animation and modification.

We run the optimization task with 100 restarts and report our result with the minimum error as Figure 3.5 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 3.4(a). Each restart is plotted using the transparent line and the median across all restarts is plotted as a solid line. 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 3.2. 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 our 5 rings are 0.5x and 1.6x compared to ours 10 rings, respectively. This is because the simpler shader has faster runtime, but converges less often.

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 minima, we apply an EliminateIfUnused() program annotation outside the computation for each ring so the compiler can automatically use 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 3.5 (c) 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.

3.4.1.2 Shader: `CelticKnot`

We modify the ring shader from Section 3.4.1.1 to match the target image for the Celtic Knot in Figure 3.5(a). The Z ordering of the rings is 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 parametric stroke width, and white elsewhere.

The black and white target image [5] brings an extra challenge to the optimization task, because the shader can no longer rely on a 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 3.3. 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 3.5. It correctly locates the rings, and correctly models the interlocking pattern. For the convergence plot in Figure 3.4 (b), ours converges at a lower rate than (a) because of the optimization challenges discussed, but significantly outperforms ours without random, FD*, and SPSA*, which do not converge at all. This is also reflected in Table 3.2. To confirm that the scales of random variables 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 3.4 (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.

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 an 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 3.5 shows an example of coloring the optimized Celtic Knot shader. With our frame-

work, 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.

3.4.1.3 Shader: `SIGGRAPH`

In this section, we explore a 3D shader that can be used to reconstruct the SIGGRAPH logo as in Figure 3.1(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 3.1(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 [152](Section 3.1). 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 randomness. Therefore, we do not associate random variables with *any* parameters that the RaymarchingLoop primitives depend on.

Our optimization is almost identical to the target image and is reported in Figure 3.1(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 3.2 and Figure 3.4(c).

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 3.4.1.2 and add Perlin noise [107] bump mapping to the geometry (Figure 3.1(e)). As an alternative, we can also utilize the displacement mapping and lighting model authored by Inigo Quilez in the original `SIGGRAPH` shader (Figure 3.1(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 3.1(f).

3.4.1.4 Shader: `TF RayMarch`

Similar to Section 3.4.1.3, we author a 3D ray-marching shader that can be used to reconstruct the Tensorflow logo [2] shown in Figure 3.5(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 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 3.5 along with a modified novel view rendering. The convergence is reported in Figure 3.4(d).

[2] TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc.

3.4.1.5 Shader: `TF RayCast`

This section discusses an alternative program representation that directly uses ray-box intersection to attempt to match the same TensorFlow target image as in Section 3.4.1.4. All other components stay the same as the `TF RayMarch` variant.

Unlike the raymarching loop, which is differentiated using special rules discussed in Section 3.1, ray-casting shader programs are differentiated using the general gradient rules in Section 2.4.2, so random variables can be applied to every Dirac parameter. As can be seen in Figure 3.4(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 3.6 we show our optimization result (a), and two novel view renderings where the letters T (b) and F (c) are recognizable.

3.4.1.6 Evaluating Gradient Approximation Quality

We also quantitatively evaluate the quality of the gradient approximation using the error metric described in Section 2.9:

$$\begin{aligned} L =& |(\Psi^* * f(\vec{\boldsymbol{\theta}}_1) - \Psi^* * f(\vec{\boldsymbol{\theta}}_0)) - \int_\Theta \Psi_k * \nabla_k f(\vec{\boldsymbol{\theta}}) d\vec{\boldsymbol{\theta}}| \\ =& |\text{RHS} - \text{LHS}| \end{aligned} \tag{3.11}$$

Recall this error metric evaluates how much the gradient approximation ∇_k violates the gradient theorem, where the first and second term corresponds to the right-hand side (RHS) and the left-hand side (LHS) of the gradient theorem (Equation 2.56) respectively.

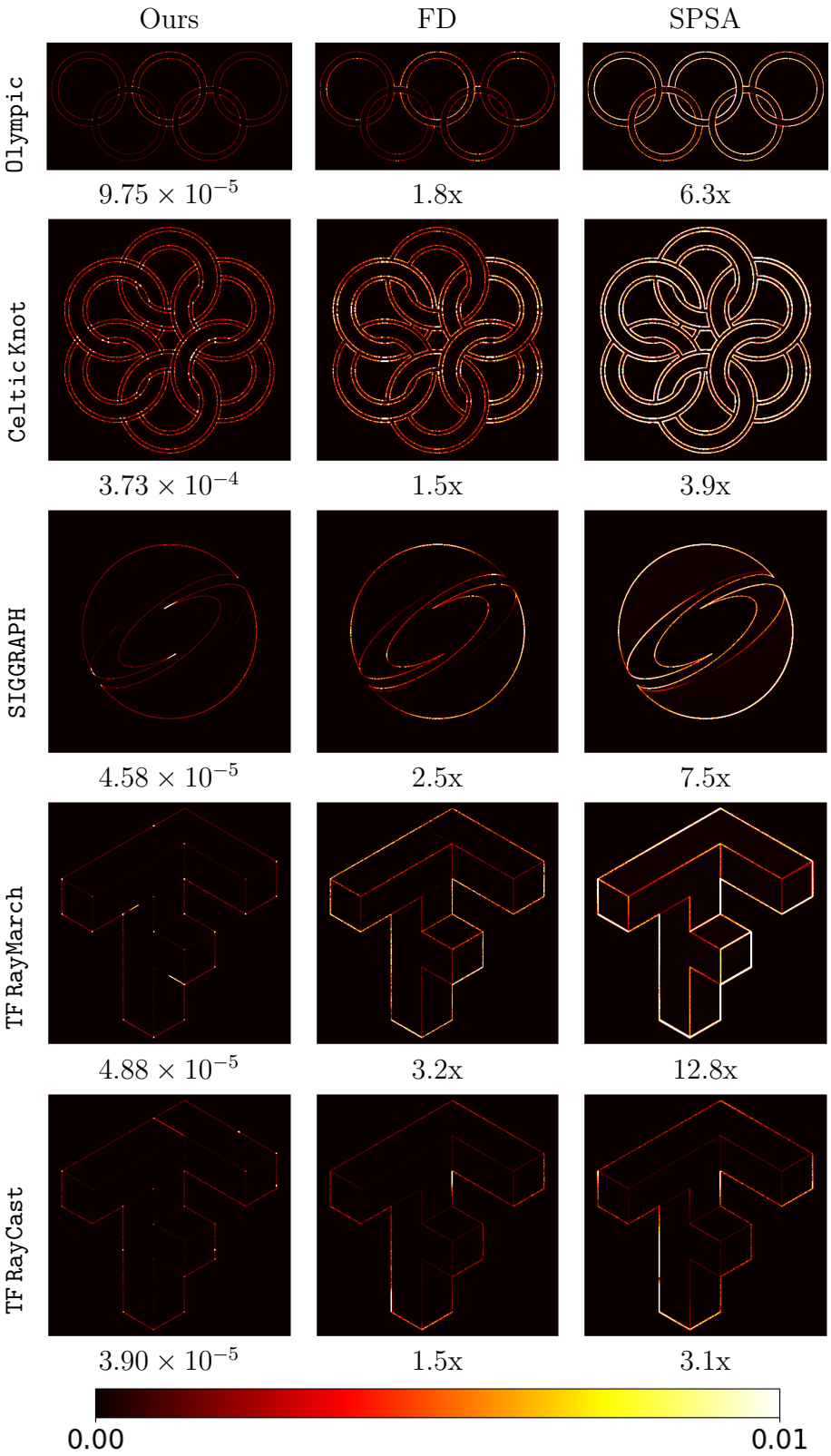


Figure 3.7: Quantitative error metric for ours and baselines. Below each method, we report the mean error from all pixels. For baselines, we report relative error compared to ours.

We evaluate the metric for five of the shader programs in Section 3.4.1.1 - 3.4.1.5. Because these shader programs output 3-channel RGB color for each pixel, we evaluate the metric over a scalar function f that sums up the pixel values in all three color channels. For each shader program, their optimized parameters are used as the center point $\vec{\boldsymbol{\theta}}$ in the parameter space and we randomly sample a unit ray originating from $\vec{\boldsymbol{\theta}}$. The integral endpoints $\vec{\boldsymbol{\theta}}_0, \vec{\boldsymbol{\theta}}_1$ in Equation 2.57 are sampled by extending the random ray in both directions to a fixed distance around the center point $\vec{\boldsymbol{\theta}}$. We then use the straight path from $\vec{\boldsymbol{\theta}_0}$ to $\vec{\boldsymbol{\theta}_1}$ as our integral path Θ in the LHS.

We always use 10^5 samples to evaluate the pre-filtering with Ψ^* for the RHS, 10^4 samples for the quadrature over Θ in the LHS, and 1 sample for the pre-filtering with Ψ_k in the LHS integrand. We find the noisy integrand estimate is smoothed out because the outer integration is sampled so densely.

The metric is evaluated without random variables (Section 3.3) 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.

Figure 3.7 visualizes the quantitative error for every pixel within the image and compares ours with baselines finite difference (FD) and SPSA. Because FD is always slower than ours in the optimization task (Section 3.4.1.1 - 3.4.1.5), it is evaluated at 1 sample per pixel. For SPSA, the number of samples for the gradient approximation is chosen so that its runtime per iteration in the optimization task is comparable to ours. To best benefit baselines, both FD and SPSA result is chosen as the lowest error from among five different step sizes (10^{-i} for $i \in \{1, 2, 3, 4, 5\}$). In all examples, ours outperforms both baselines by a large margin. Note for `SIGGRAPH`and `TF RayMarch`, the shaders render the 3D geometry using sphere tracing and is differentiated using a different rule from Section 3.1 whose accuracy depends on that of the iterative sphere tracer as well. Nevertheless, ours still has a low error by a large margin compared to the baselines.

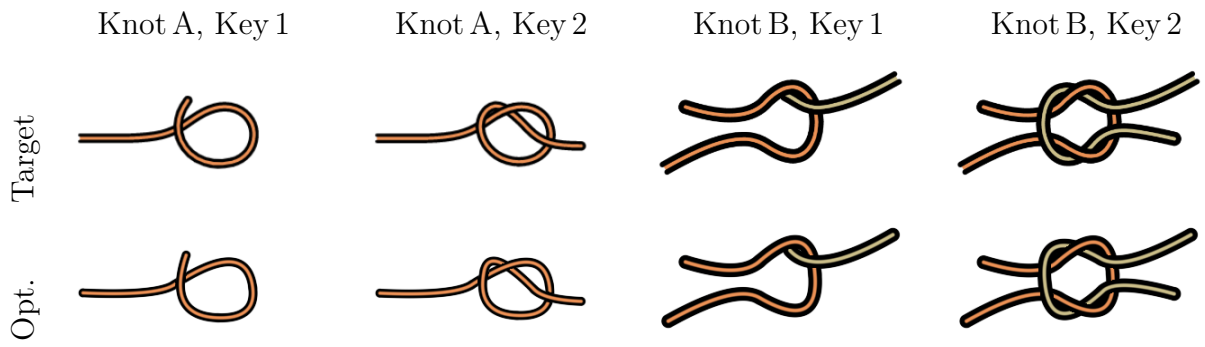


Figure 3.8: Optimizations (below) matching animation keyframes (above) for two knot-tying examples. We manually pick target key frames from the original animation, as described in Section 3.4.2.

3.4.2 Optimizing Animation Sequence

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 3.8. 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. We imagine the mathematical representation of the rope can potentially help with applications such as teaching a robot to tie the knot.

In our implementation, we manually pick and optimize keyframes in the animation sequence and increment the number of Bezier segments by 1 for each keyframe. Figure 3.8 demonstrates our reconstruction of two keyframes for each of the two different knots. 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.

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.

We manually pick 6 keyframes for the rope shown as Knot A Figure 3.8 and 8 keyframes 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 keyframes in sequential order, and always initializes the new optimization based on parameters from the previous keyframe. For every keyframe, our framework first classifies whether a new rope segment should be introduced: at the first keyframe or when a new color (representing a different rope) appears in the current frame. If a 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 sampling new spline segments and evaluating 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 indicate the direction of the rope, so that new segments are initialized from the correct end of the rope for the next keyframes. At the last keyframe, 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 leads to the problem that the optimal depth parameters in the L2 sense do not correspond to human intuition, as they may encourage

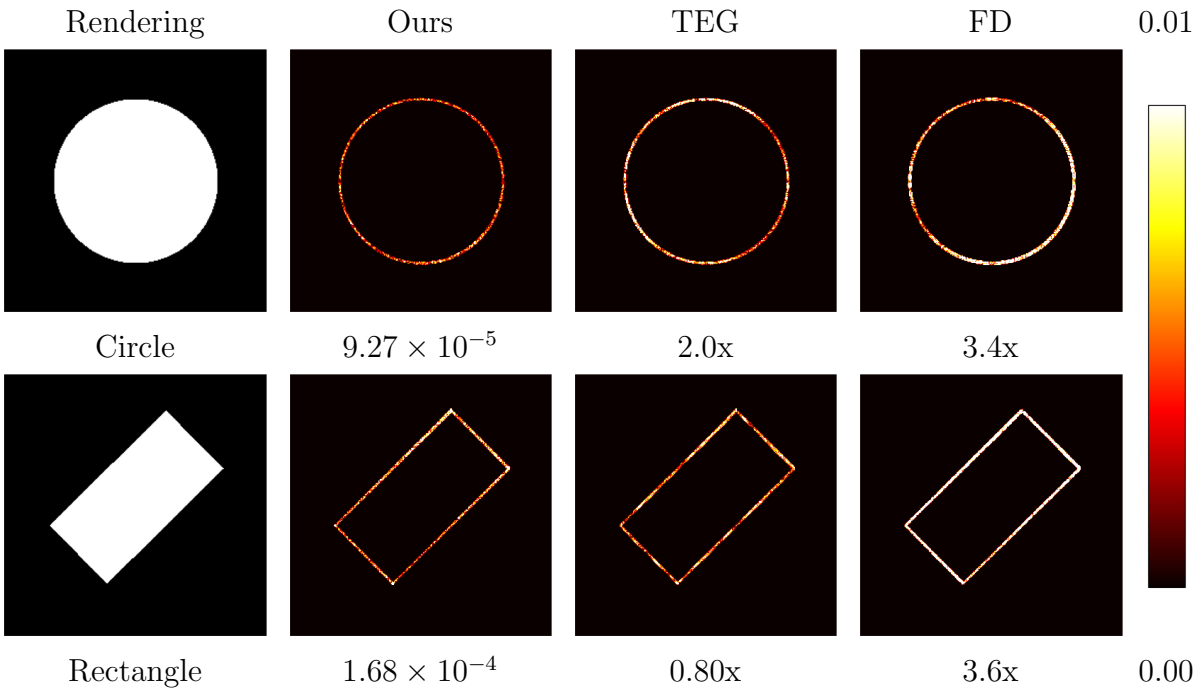


Figure 3.9: We compare ours, TEG [10] and finite difference (FD) using a circle and a rectangle shader. The quantitative errors are reported below as the mean error from all pixels, with baselines relative to ours. FD is evaluated at 1 sample per pixel and is chosen as the lowest error from among five different step sizes (10^{-i} for $i \in \{1, 2, 3, 4, 5\}$).

the intersection of ropes to lower the loss objective. Therefore, the human effort may be involved to reject these optimization results. Finally, the optimized depth parameters will be passed back to all previously optimized keyframes to ensure correct layering throughout the animation.

3.4.3 Simple Shader Comparison with Related Work

This section compares our method with TEG [10] and differentiable vector graphics [71]. Because these baselines are less expressive than ours, the comparison is limited to simple shader programs that are expressible in both their framework and ours.

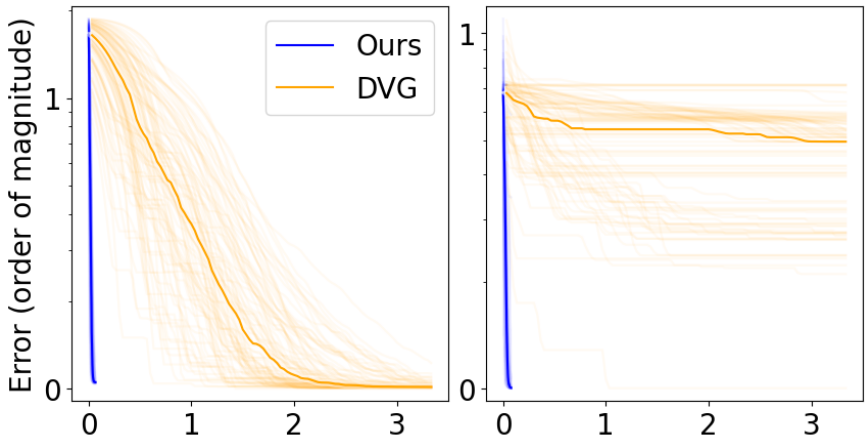
3.4.3.1 Comparison with TEG

We compare with TEG [10] 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 the quantitative error metric as in Section 3.4.1.6 and report in Figure 3.9. Because TEG constructs the pre-filtering in the image space as a two-dimensional integral, and because the Dirac delta only removes the inner integral, the remaining outer integral still needs to be evaluated using the trapezoid rule. We evaluate the remaining integral using 10 samples because otherwise, it causes aliasing and high error. Note that such implementation allows TEG to access more samples than ours and FD.

Because TEG can correctly differentiate both shaders, a low quantitative error is expected, and the error is solely caused by the approximation that TEG makes using its quadrature rule for pre-filtering the 2D kernel. The error is larger in the circle shader because the integral bounds need to be remapped between the Cartesian and polar systems. There are three possible sources of error for ours: sampling error for the 2D pre-filtering; first order residual error in our gradient approximation ($O(\epsilon)$ term in Definition 3 and 2); 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.

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 3.4.1 because rasterization on the CPU is slow.



(a) `Circle` (b) `Ring`

Figure 3.10: Comparing the performance of ours and differentiable vector graphics [71] for two optimization tasks – using 100 restarts with random initialization and plot axes as in Figure 3.4. 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 Figure 3.12 for details.

3.4.3.2 Comparison with Differentiable Vector Graphics

We compare with Differentiable vector graphics (DVG) [71] 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 into PyTorch and does not provide an API for efficiently extracting per-pixel gradient maps for arbitrary parameters, our comparison is focused on comparing performance in a gradient-based optimization task.

The reference images 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 the 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 2×2 samples instead.

For the circle shader (Figure 3.10(a)), both ours and DVG easily converge to images that are similar to the reference image (Figure 3.11), but DVG is much slower. For the ring

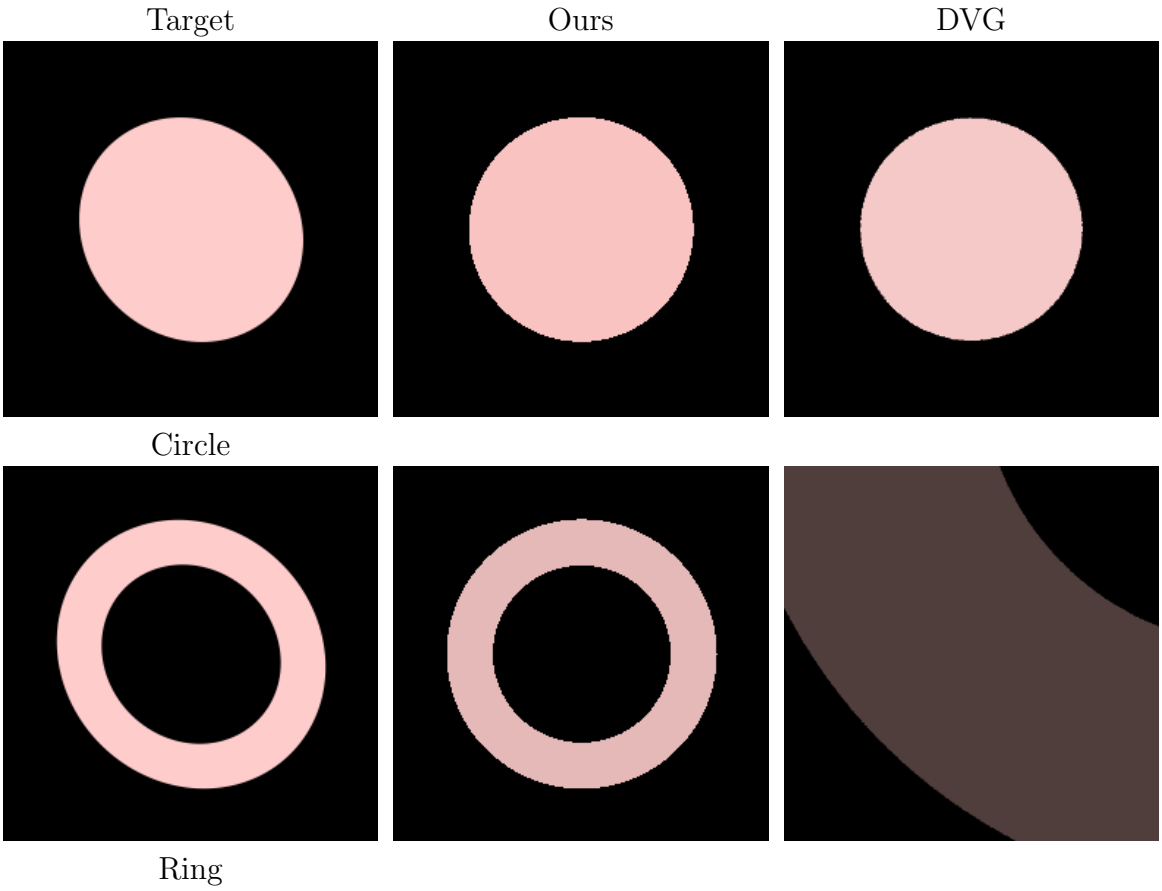
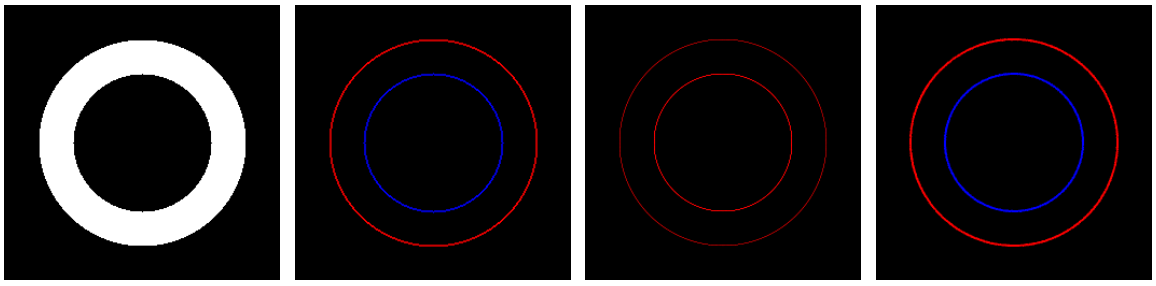


Figure 3.11: Comparison of the optimized result for ours and DVG. Because each task restarts with 100 random initializations, here we show for each task, the result whose error corresponds to the median of 100 restarts. The target images are rendered with slightly elliptical shapes to avoid either ours 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 3.12.



(a) Rendering (b) Ours (c) DVG (d) FD

Figure 3.12: 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 the opposite direction as ours and FD.

shader, however, DVG fails to converge for most of the restarts (Figure 3.10(b)). We suspect the non-convergence is caused by a bug in their code that generates incorrect gradients (Figure 3.12). However, even after disregarding the bug, DVG is slower than our method and can not handle parametric Z ordering as in Figure 3.5.

3.5 Summary and Discussion

In summary, this chapter applies the Aδ differentiation framework introduced in Chapter 2 to the application of interactive editing and animating illustrations represented as shader programs. Our method has several limitations, which offer potential avenues for future work.

First, our current implementation 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, front-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.

Finally, 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 [82], so that both foreground and background objects are involved in the differentiation.

Chapter 4

Optimizing Discontinuous Audio Synth Applications

This chapter adapts and extends the Aδ differentiation framework (Chapter 2) to audio synthesizers. Synthesizers are electronic musical instruments that generate waveforms through compositions of modules such as frequency modulation, filters, noise, and envelope. They are powerful and expressive, but come at the cost of difficulty in control. A typical synthesizer includes hundreds of parameters, and manually tuning every one of them is both tedious and time-consuming even for experts. Automatic synthesizer programming addresses this challenge to automatically search for a set of patch connections and parameter settings to generate audio that best matches a given target sound. Such parameter optimization tasks benefit from access to accurate gradients. However, typical audio synthesizers incorporate components with discontinuities – such as sawtooth or square waveforms, or a categorical search over discrete parameters like a choice among such waveforms – that thwart conventional automatic differentiation (AD). Some of the discontinuities, such as the discontinuous waveforms can be easily differentiated through our Aδ framework, while others, such as the categorical discrete parameters, require specialized gradient rules inspired by the random variables introduced in Section 3.3. Therefore, this chapter utilizes and extends the Aδ

differentiation rules to directly differentiate the synthesizer as a white box program, and thereby optimize its parameters using gradient descent. We evaluate our framework using a generic FM synthesizer with ADSR, noise, and IIR filters, adapting its parameters to match a variety of target audio clips. Our method outperforms baselines in both quantitative and qualitative evaluations.

4.1 Overview

Synthesizers provide musicians and sound designers with creative flexibility, diverse audio characteristics, and convenient production capabilities. However, the versatility of synthesizers also poses challenges in terms of control, because manually searching over numerous parameters to seek a particular type of sound requires expertise, time, and effort. Automatic synthesizer programming addresses these challenges by automating this search process to find a set of parameters that best match a given target sound. Formally, given a synthesizer f with parameters $\vec{\boldsymbol{\theta}}$ and a target sound T, the search problem seeks the optimal parameters $\vec{\boldsymbol{\theta}}^*$ that minimize some loss function L between the synthesizer output and the target.

$$\vec{\boldsymbol{\theta}}^* = \text{argmin}_{\vec{\boldsymbol{\theta}}}\, L(f(\vec{\boldsymbol{\theta}}), T) \tag{4.1}$$

In principle, a straightforward solution to Equation 4.1 would differentiate L with respect to the parameters $\vec{\boldsymbol{\theta}}$, and then minimize L by gradient descent. However, in practice, typical synthesizers f contain discontinuous oscillators, like square or sawtooth waveforms, and discrete categorical parameters, such as choosing different waveforms and modules, that thwart traditional automatic differentiation (AD) approaches.

Audio researchers have developed several workarounds that avoid directly differentiating f. For example, zeroth order optimizations such as genetic algorithms [153, 127] approximately solve Equation 4.1 at the expense of greater computation and potential artifacts from failures near local minima. Alternatively, Equation 4.1 may be approximated by deep learn-

ing approaches. For example, the synthesizer f can be approximated via a differentiable neural proxy [80, 20], or the entire argmin mapping can be approximated by a parameter prediction network [67, 11]. However, the flexibility of deep learning approaches for synthesizers is constrained, as data collection and training are typically limited to specific synthesizers with fixed parameter choices, making it impractical to directly apply trained models to arbitrary synthesizers.

As discussed in Chapter 2 and 3, a variety of methods in computer graphics have been developed recently to approximate the gradient for image generation processes that contain discontinuities, including our own Aδ framework introduced in Chapter 2. Therefore we build on the Aδ gradient rules to differentiate audio synthesizers as well.

This chapter presents an optimization framework to directly differentiate the pre-filtered synthesizer output, and solve Equation 4.1 via gradient descent. We adapt and extend the math in Aδ to differentiate discontinuous and discrete synthesizer components. We also introduce heuristic methods for better convergence. The approach finds synthesizer parameters that better match the target than baseline methods by qualitative and quantitative measures. Moreover, our framework allows musicians to incorporate domain expertise to flexibly modify and fine-tune synthesizer components.

4.2 Related Work

Researchers have explored a variety of techniques to automatically search for optimal synthesizer parameters without having to explicitly differentiate the synthesizer. Genetic algorithm approaches [153, 127] mutate and crossover variants to search over the entire program space for arbitrary synthesizers, but are at the cost of excessive compute and have difficulty accurately converging to local minimums without the guidance of the gradient. On the other hand, deep learning models can be used to directly predict the synthesizer parameters [118, 11, 67]. However, they heavily rely on the annotated datasets of synthesizer presets,

therefore cannot be flexibly generalized to *any* synthesizer. Similarly, each trained model can only be used for one particular synthesizer patching, greatly limiting the flexibility of the method. Unlike learning-based methods, we do not need any dataset, and can flexibly differentiate *any* white-box programs for easy finetuning and parameter transfer between synthesizer patchings. Additionally, our gradient descent framework allows us to converge better than GA.

Alternatively, synthesizers can be defined by differentiable functions, therefore allowing optimal parameters learned through gradient descent. For example, neural audio synthesis methods use black-box neural networks to generate audio samples [99, 38]. The neural proxies can be combined with continuous synthesizer components as well, such as DDSP methods that incorporate digital signal processing modules [37, 20], and DWTS methods with learnable wavetables [117]. However, because these methods use continuous proxies, they usually do not match the exact parameterization of complicated discontinuous synthesizers, therefore cannot be flexibly used to control conventional synthesizers. Additionally, the neural modules introduce nontrivial inference overhead and are less efficient than traditional synthesizers. Unlike the differentiable neural proxies, our method directly differentiates the white box program that emulates a traditional synthesizer. Therefore, it optimizes semantically meaningful synthesizer parameters.

4.3 Method

This section demonstrates our gradient-based optimization pipeline. We first describe in Section 4.3.1 our approach to differentiating the synthesizer. Next, we present our loss function choices in Section 4.3.2 and finally discuss how to explore the multi-modality and avoid local minimums in Section 4.3.3.

4.3.1 Approximating the Gradient

This section focuses on the customized gradient introduced to our framework. This includes differentiating the discontinuities (Section 4.3.1.1 - 4.3.1.2), workaround to avoid zero gradients (Section 4.3.1.3), and efficiently differentiating IIR filters (Section 4.3.1.4). The rest of the program can be easily differentiated using traditional AD.

4.3.1.1 Differentiating Discontinuous Waveforms

We view both the square and sawtooth wave as periodic compositions of the Heaviside step functions, which evaluate to 1 on the one side, and 0 on the other side. The discontinuity can be differentiated by applying the gradient rules proposed by Aδ [149], which approximates the gradient as if the discontinuous signal is convolved with a 1D box filter along the time dimension. Note the Aδ approach is more accurate than differentiating a naively smoothed discontinuity with arbitrary linear or sigmoid transitions, especially when discontinuities are composited. For example, the composition of discontinuous modulation and carrier signals in an FM synthesizer.

4.3.1.2 Differentiating Discrete Categorical Choices

The Aδ gradient rules help differentiate discontinuities that can be easily sampled along the time dimension, such as the discontinuous waveforms in Section 4.3.1.1. However, the challenge remains for the discrete categorical choices, because they rarely interact with the continuously sampled time dimension, therefore the discontinuity cannot be easily sampled.

This section proposes a stochastic approach to differentiate the discrete parameters. Specifically, we define a categorical node g as taking input from a discrete parameter x

with potential choices A, B, ..., and outputs to a floating point value:

$$g(x;\vec{\boldsymbol{\theta}}) = \begin{cases} g_A(\vec{\boldsymbol{\theta}}) & \text{if } x == A \\ g_B(\vec{\boldsymbol{\theta}}) & \text{if } x == B, \\ ... \end{cases} \tag{4.2}$$

g_A, g_B are floating point functions associated with choices A, B respectively. For example, this could be the waveform equations such as sine and square.

Our stochastic approach views the discrete parameter x as a discrete random variable $\mathcal{X}$ with different samples X at different time steps. Therefore $g(\mathcal{X};\vec{\boldsymbol{\theta}})$ becomes a random variable as well. Throughout this section, we will use lowercase letters (e.g. x) for the synthesizer parameters that need to be optimized, calligraphic uppercase letters (e.g. $\mathcal{X}$) for its corresponding random variables, and regular uppercase letters (e.g. X) for sampled values from the random variable. Note when $\mathcal{X}$ has close to zero variance, it will consistently sample the same choice for every time step, therefore X can be viewed as a constant identical to x. We further model $g(\mathcal{X};\vec{\boldsymbol{\theta}})$ similarly to an argmax operator, where each potential choice A, B, ... is associated with a "score" random variable, and the output of g corresponds to the choice with the highest "score". Specifically, the "score" for choice A is modeled as $\mathcal{Y}_A = \mu_A + \sigma_A \cdot \mathcal{U}$, where μ_A, σ_A are the mean and standard deviation, and $\mathcal{U}$ is a uniform random variable with zero mean and unit variance. For any two neighboring samples with disagreeing categorical choices A and B, we view the inconsistency as a discontinuous branching conditioned on whether the sampled "score" Y for choice A is greater than B or not: $g = \text{select}(Y_A > Y_B, g_A, g_B)$. By forming the discontinuity this way, the gradient wrt $\mu_A, \sigma_A, \mu_B, \sigma_B$ can be easily computed with the Aδ gradient rules on the time domain. At convergence, the variance to every "score" variable should be reduced to a small value such that the categorical choice is sampled consistently.

Our stochastic gradient rule works best when there is a high correlation between the functions associated with each choice g_A, g_B, etc. Intuitively, this allows g_A, g_B, ... to form a convex hull for the sampled output $g(X; \vec{\boldsymbol{\theta}})$, therefore reducing the variance of the gradient estimation. Therefore when differentiating categorical waveform choices, we align the phase of the wave functions such that their correlation is maximized.

4.3.1.3 Avoiding Dead Gradient Caused by Clamping

Many synthesizer parameters have constraints on their values, such as the period for ADSR stages should be nonnegative, and the filters' cutoff frequencies should be within a valid range to avoid singularities. A typical strategy for optimizing these constrained parameters in an unconstrained problem is to clamp the parameters: taking the min and max against their upper and lower bounds. However, clamping introduces another challenge for optimization: once the parameter is out of bounds and clamped, the gradient wrt the parameter becomes zero. For example, $\frac{\partial \max(\theta,0)}{\partial \theta} = 0$ whenever $\theta < 0$. An analogy is the "dead neuron" for ReLU activations in neural networks.

We propose a heuristic workaround that avoids constrained parameters getting stuck at out-of-bound values. Specifically, we design the following customized gradient for the min or max operator f that compares with a constant C, and assume the gradient wrt f is already computed as dL/df.

$$\begin{aligned} f =& \min(\theta, C) \\ \frac{\partial L}{\partial \theta} =& \text{select}(\theta < C, \frac{dL}{df}, \max(\frac{dL}{df}, 0)) \end{aligned} \tag{4.3a}$$

The gradient for the max operator is similar to Equation 4.3a, but $<$ and max are replaced by $>$ and min respectively. Note this is only a heuristic workaround for reverse-mode AD, and can not be used for forward-mode AD because it requires computing dL/df before differentiating f. Intuitively, the customized gradient in Equation 4.3a will push the out-of-

bound parameter θ back to its valid range whenever the gradient wrt f wishes to bring the clamped value back to valid.

4.3.1.4 Efficient IIR Filter Back-propagation

Infinite impulse response (IIR) filters are widely used in synthesizers to flexibly control the timbre. However, differentiating the IIR filter introduces performance challenges because each output value at a certain time step recurrently depends on every input/output value in previous steps, and naively unrolling the gradient in the time domain is computationally expensive. We, therefore, avoid the complex dependency in the time domain by applying the filter in the frequency domain similar to [80]. During optimization, we only differentiate the spectrogram of the synthesized audio multiplied by the spectrogram of the unfiltered audio with the frequency response of the filters. Because most popular filters (e.g. Biquad, Butterworth) used in synthesizers already have closed-form solutions for their frequency responses, requiring frequency domain proxy does not affect the expressiveness of our approach.

4.3.2 Loss Function

This section discusses the optimization loss function L. Unlike supervised deep-learning methods that could rely on loss in the parameter space at the cost of collecting the preset dataset, our optimization pipeline can only rely on spectral and time domain losses. However, finding the ideal loss that is consistent with human perception is challenging for several reasons. Firstly, standard losses such as spectrogram (log mel) L2 only work best when the distances between two signals are within the just noticeable difference (JND), but this is rarely the case during our optimization, as we always start with random initial guesses, and the synthesizer may never reach JND to an out-of-domain target signal. Furthermore, although deep perceptual metrics have been developed for speech signals, generalizing them to synthesizers introduces extra noise as well.

We propose a heuristic combination of several different losses to approximate the perceptual similarity. The intuition is that the gradient to the majority of the losses should agree with human perception even if a few of them are noisy. Besides the standard losses, we additionally include the 1D Wasserstein distance along the frequency dimension because of its wide applicability in matching distributions. Our final optimization loss is a weighted combination of the Wasserstein distance, L2, log mel L2, and a deep feature distance from the wav2clip model [143]. The weights are chosen such that each component has a relatively equal contribution. For the losses that work on a spectrogram (L2, log mel L2, and Wasserstein), we use three different window sizes (512, 1024, 2048) with 75% overlap between windows. The deep feature loss also uses the same window and hop sizes, but for efficiency, we stochastically evaluate the model using one of the window sizes per iteration. Additionally, because the deep feature model takes time domain signal as input, we need to apply inverse STFT to the spectrogram because of the frequency domain IIR approximation described in Section 4.3.1.4.

4.3.3 Identifying Perceptually Similar Results

Gradient-based optimizations may converge to a variety of local minimums with very different perceptual similarities to the target sound. Our framework runs multiple random restarts to avoid getting stuck at local minimums. However, we do not yet have a quantitative metric that reliably characterizes the perceptual similarity for synthesizers. While we use our weighted loss in Section 4.3.2 to provide a gradient for the optimization, its absolute value does not precisely correspond to perceptual similarity either: perceptually dissimilar results may still have lower error than similar results, and manual selection is still needed to filter out the bad results. Therefore, we further propose an automated process that uses early termination to avoid wasting compute at local minimums, and also a mechanism to identify good quality results after convergence.

Our early termination strategy is a generalization to the intuition that good initial starting points have ahigher probability of converging into good results. We generalize the heuristic to arbitrary iteration within the optimization process and terminate the optimizations with bad results at the end of a sequence of predetermined iterations. Additionally, because the weighted loss in Section 4.3.2 cannot reliably characterize perceptual similarity, we rely on the Pareto ranking on multiple losses to identify bad results. Specifically, we terminate optimizations whose Pareto rank on every non-deep-learning loss in Section 4.3.2 is higher than ceil(max_rank), where max_rank is the maximum Pareto rank for the current population. In our implementation, the early termination is checked for every 100 iterations starting at iteration 200, and we run every optimization until full convergence and simulate the early termination.

Besides early termination, we also note that when the optimization result is already close to the target at convergence, its loss metrics calculated from a larger window size would better resemble perceptual similarity. Specifically, bad results usually have a much large L2 error. We therefore further filter out the converged result whose L2 loss on the spectrogram with window size 2048 is 2x higher than the lowest among all results, and finally rank the remaining results based on the weighted sum of Wasserstein, L2, and log mel L2 on the same spectrogram.

4.4 Validation

This section validates our proposed framework by optimizing the parameters of an FM synthesizer to match various audio signals for musical instruments and special sound effects. All the targets are downloaded from the web and are therefore out of domain. We first describe our FM synthesizer in Section 4.4.1 and evaluation setup in Section 4.4.2, then compare our method with two baselines through a user study (Section 4.4.3). We also show the optimization convergence in Section 4.4.4. Finally, Section 4.4.5 demonstrates the

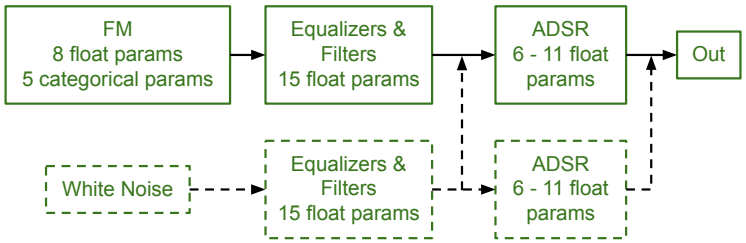


Figure 4.1: Summary for the synthesizer model used in our experiments. Dashed boxes and arrows corresponds to optional components whose connection is decided per target.

flexibility of our framework with a case study that finetunes the optimization result to a modified synthesizer.

4.4.1 Synthesizer Model

We implement an FM synthesizer in PyTorch to leverage its automatic differentiation (AD) framework. The gradient discussed in Section 4.3.3 is implemented as the cutomized backward pass, and regular AD is used for the rest of the computation (e.g. ADSR, STFT).

Our FM synthesizer structure is summarized in Figure 4.1. It has one carrier signal modulated by the weighted sum of four different signals. Each signal, including the carrier and modulation, is parameterized with a categorical choice from the four base waveforms: sin, square, triangle, and sawtooth. Each modulation signal is further parameterized by ratio and index, which controls the frequency and the magnitude of the modulation respectively. The FM signal will further be filtered by three Biquad equalizers (low shelf, high shelf, and peak) and a pair of Butterworth low and high pass filters. After that, the filtered signal will be multiplied by an ADSR model with an optional choice for exponential release and amplitude modulated (AM) attack, decay, and sustain. Finally, filtered white noise can be optionally added either by sharing the original ADSR or with a different set of tunable ADSR parameters. Note the optional configurations are manually decided for each target signal based on their different audio characteristics. The overall model includes 40 - 70 parameters.

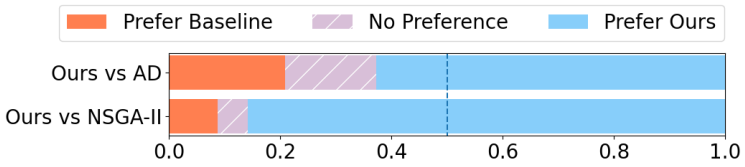


Figure 4.2: Listening test distribution aggregated for every worker and every target audio. For each worker and each target, we compare the average ratings between ours and two baselines (AD and NSGA-II). We denote our rating that is smaller than, equal to, or greater than that of the baseline as "prefer baseline", "no preference", and "prefer ours" respectively.

4.4.2 Evaluation Setup

We compare with two baselines: gradient-based optimization but with traditional AD, and zeroth order optimization with genetic algorithm NSGA-II. The AD baseline uses the same optimization framework described in Section 4.3, except that the gradient described in Sections 4.3.1.1 - 4.3.1.3 is replaced by traditional AD. The zeroth order baseline does not require any gradient, and instead uses the genetic algorithm NSGA-II [33] to search over the parameter space. Note because NSGA-II is a multi-objective algorithm, it directly finds Pareto optimal solutions to the various loss functions in Section 4.3.2 without having to compute their weighted sum as in gradient-based optimization.

We use 16 different target sounds, including 12 musical instruments and 4 special sound effects listed in Figure 4.3.

For ours and AD, we run the experiment with 100 random restarts for a maximum of 2000 iterations per restart. Note that because of the early termination described in Section 4.3.3, the actual number of iterations per restart varies. We additionally supply the NSGA-II with a reasonable sample range to the parameters, and run the algorithm with 100 population size and 2000 generations.

4.4.3 Listening Test

Because no perceptual loss exists for comparing synthesizers and musical instruments, we rely on the user study to qualitatively compare results for ours and baselines. For each

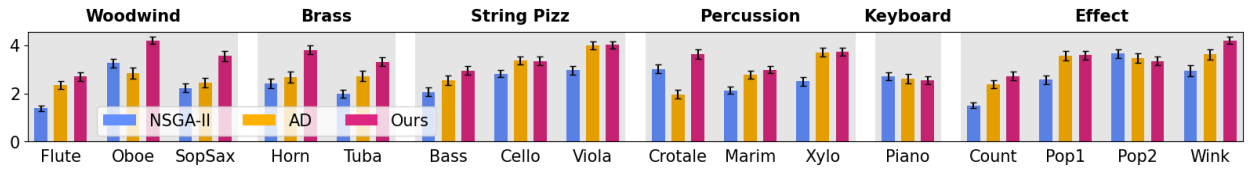


Figure 4.3: Listening test rating for each of the 16 target audios grouped into six categories. Error bars correspond to 2SEM (standard error of mean). To save space we shorten the following names: Marim(ba), Xylo(phone), and Count(down).

method and target sound, we pick the top 4 results based on the ranking from the last paragraph of Section 4.3.3, generating a total of 12 samples per target for all three methods: ours, AD, and NSGA-II. We then ask workers on Mechanical Turk to rate how similar each result is to the target audio. Each worker is asked to rate all 12 samples for two different targets. We further embed four validation samples to filter out careless ratings: two that are intentionally corrupted from the two targets to be worse than any of the 12 corresponding optimization results, and two other samples that are identical to targets randomly chosen from all 16 targets. Therefore each worker rates $2 \times 12 + 4 = 28$ samples. The user study is repeated so that each instrument is rated by 30 different workers.

Figure 4.2 demonstrates the distribution of the user study aggregated across every worker and every target. Our method outperforms both baselines by a larger margin, but AD is preferred more than NSGA-II. We further compute the p-value for the following null hypothesis: our average rating per user per instrument is smaller than or equal to that of the baseline. The p-value for the AD baseline is 2e-8, and for the NSGA-II baseline is 3e-61.

We additionally report in Figure 4.3 the rating for each of the target audio. Ours performs best when the FM synthesizer is a good emulation of the underlying instrument that generates the target, such as for woodwind or brass categories. AD has similar ratings to ours more frequently than NSGA-II, which is consistent with Figure 4.2 where AD is preferred over NSGA-II. Note in all cases when baselines have similar or higher ratings than ours, the rating difference is always within the error bar, indicating the rating preference is not statistically significant. We could further characterize the cases where ours and baselines

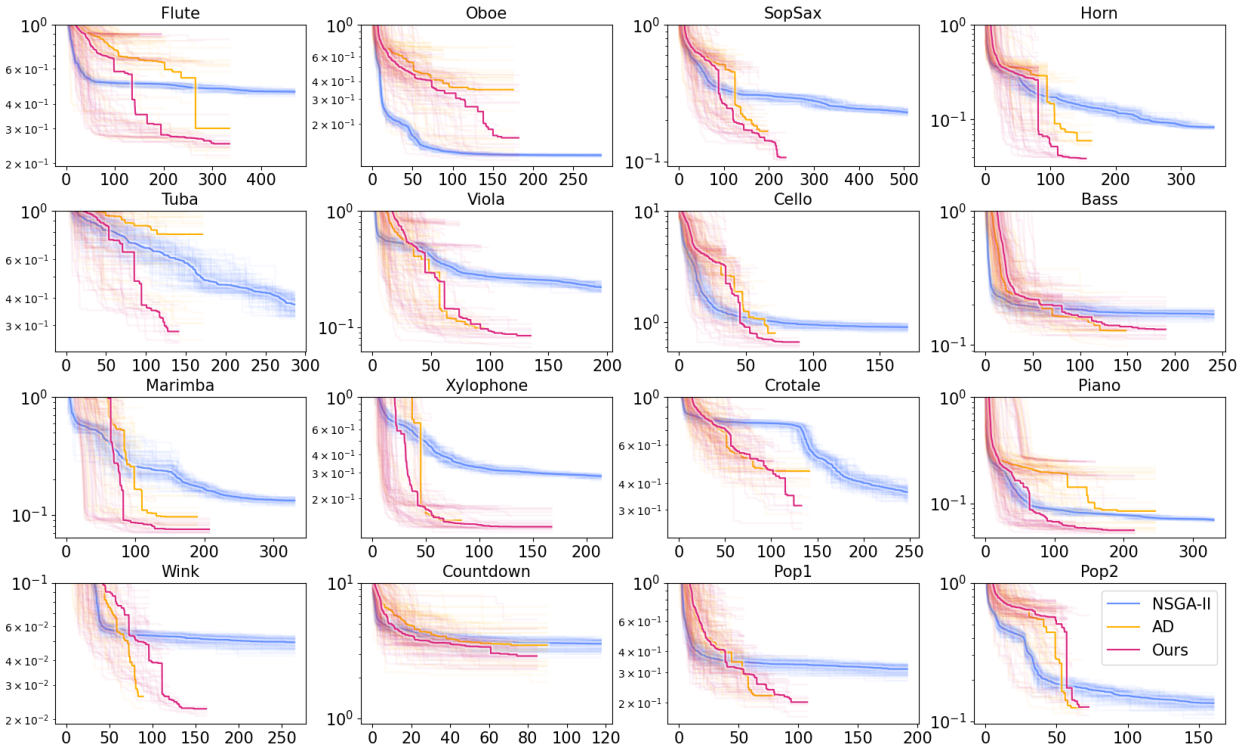


Figure 4.4: Comparing the convergence of ours and baselines for the 100 random restarts of all 16 optimization tasks. The x-axis reports simulated time: the number of function evaluations scaled with the actual runtime for each method. The y-axis reports the weighted loss used for the gradient-based methods (ours and AD). For gradient-based methods, each transparent line corresponds to a restart. For NSGA-II, each transparent line plots the loss value for the kth population at each generation, where k is between 1 and 100. The median within all restarts (all populations for NSGA-II) at a given time is shown as the solid line.

have similar ratings into two scenarios. The first one is when the target is less challenging, and can be easily reconstructed by various local minimums, such as Pop1 and Pop2. Therefore, AD is likely to converge as long as one of the local minimums is close enough to any of the 100 random restarts. Similarly, it is also easier for NSGA-II to find the local minimum through mutation. The second scenario is when the FM synthesizer cannot nicely emulate the instrument, such as for Piano. Therefore none of the methods can converge close enough to the target, resulting in similarly low ratings.

4.4.4 Optimization Convergence

This section discusses the optimization convergence for ours and baselines. This would demonstrate how frequently each method would converge in the optimization. Figure 4.4 demonstrates the convergence for all 16 optimization tasks. In all of the plots, the 100 populations for NSGA-II converge similarly because bad results are removed at the end of each generation. Unlike genetic algorithms, the 100 optimizations for both ours and AD have diverging performances because gradient-descent only explores the local parameter space and may be stuck at a local minimum. The early termination described in Section 4.3.3 conservatively removes some of the local minimums, but more importantly reduces the number of evaluations toward the end of the optimization because fewer restarts are still active. Typically, the convergence plot is consistent with the listening test result in Figure 4.3, with the exception of Oboe, where NSGA-II converges to the lowest error, but its listening test performs worse than ours. But this is simply due to the choice of weights that combine multiple losses into one scalar: NSGA-II converges to lower Wasserstein error and higher L2 and log mel L2 loss, therefore it is *not* Pareto superior to ours and AD.

4.4.5 Finetune Case Study

This section uses the Xylophone target as a case study to demonstrate that our white box method can be easily combined with user expertise to modify the synthesizer components and finetune the parameters flexibly.

Similar to all other target audios, the Xylophone target is initially approximated by the synthesizer model described in Section 4.4.1. It uses filtered white noise with independent ADSR to model the strike at the beginning of the sound. However, the optimization result for this synthesizer model is not ideal, specifically, the beginning of the audio sounds very different from the target. This can be verified by Figure 4.5, which compares the spectrogram for the first 0.07s of the sound between the target (a) and the optimization(b): the optimization has a longer attack stage than the target.

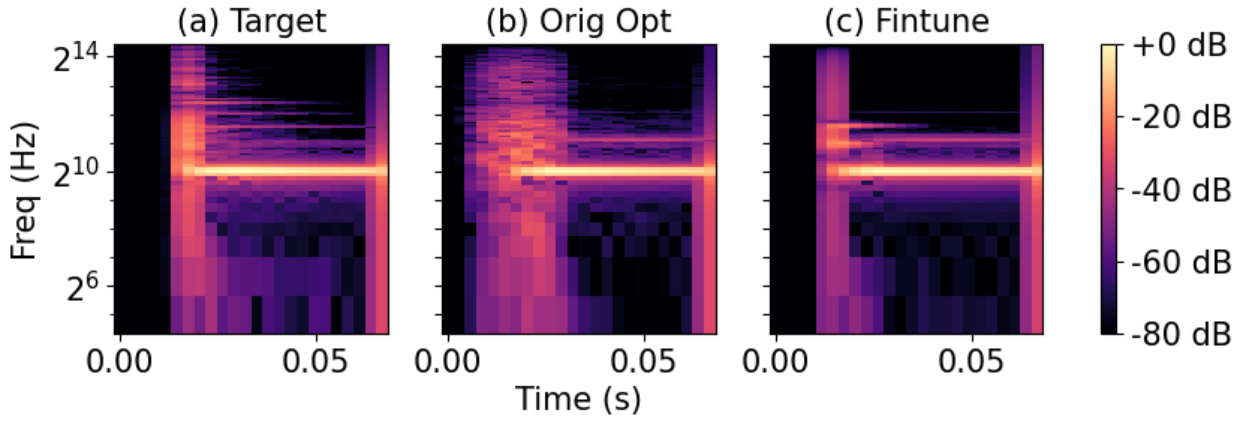


Figure 4.5: Visualizing the spectrogram for the Xylophone target (a), original optimization result (b) using filtered white noise described in Section 4.4.1, and finetune result (c) using an impulse component described in Section 4.4.5. The spectrogram is computed with window size 512 and hop size 128.

We ask a synthesizer expert to identify the potential cause of the inconsistency: instead of using filtered white noise, the beginning of the audio may be better approximated by an impulse with IIR filters. We, therefore, use the following impulse component to replace the filtered white noise component in our synthesizer. We first manually calibrate the starting time of the Xylophone sound within the target audio clip, and set the impulse at that location. Similar to the white noise, the impulse is also filtered by three Biquad equalizers (low shelf, high shelf, and peak) and a pair of low and high-pass Biquad filters. Note because the impulse signal is not static, we have to optimize the IIR parameters in the time domain rather than the frequency domain as in Section 4.3.1.4. Therefore we avoid using any Butterworth filters mentioned in Section 4.4.1 for a faster backward pass. Because the original optimization nicely approximates the target except at the beginning, we only compute the loss for the first 2048 samples, and keep all the FM-related parameters fixed to only optimize the newly added IIR parameters for the impulse, the scale of the impulse, and the original ADSR parameters that are initialized with their previously optimized values. To better characterize the filtered impulse signal, we also use smaller spectrogram window sizes: 128, 256, and 512 with 75% overlap. Figure 4.5(c) shows the spectrogram of the finetune

result that indeed better matches the attack stage of the target. Perceptually it also sounds similar to the target, please refer to our project page for audio results.

Note the finetune process described in this section cannot be easily adopted by deep learning methods, because they would require re-collecting a new dataset and re-training the model for any change in the synthesizer models. Unlike them, because we directly optimize the white-box programs, we can flexibly change the synthesizer components and reuse any parameters from previous optimizations that are still relevant.

4.5 Summary and Discussion

The framework described in this chapter only searches for synthesizer parameters, and leaves patch connections fixed. Nevertheless, the gradient rules described in Section 4.3.1 provide a potential solution. It could be easily extended to optimize binary connection decisions, therefore the general patch connection could be optimized if viewed as compositions of binary choices.

Additionally, because no perceptual loss for musical instruments and synthesizers exists, our framework has to rely on a combination of various loss terms (Section 4.3.2) together with a Pareto rank based early termination strategy to improve convergence. Future work on perceptual similarity could greatly simplify our pipeline.

In conclusion, this chapter proposes to find synthesizer parameter settings that best match a given target sound by directly differentiating the white-box synthesizer program. We adapt and extend recent methods from differentiating rendering to differentiate the discontinuous and discrete components of the synthesizer, and design an optimization pipeline to solve the problem through gradient descent. We validate our method through user studies on Mechanical Turk, where our result is preferred over baselines by a large margin. We further demonstrate the benefit of differentiating white-box programs through a case study, where we can flexibly modify and finetune synthesizer components.

Chapter 5

Approximate Program Smoothing

Chapter 2 introduces a math framework Aδ that extends traditional reverse-mode AD and differentiates arbitrary discontinuous programs. The approximate gradient relies on the idea of pre-filtering: the approximation is equivalent to first convolving the discontinuous program with a smoothing kernel before the differentiation. The integral introduced by the convolution, therefore, avoids the "naked" Dirac delta caused by differentiating the discontinuity, allowing them to be evaluated within the integral. While the Aδ gradient can be conceptually viewed as differentiating a continuously smoothed program (i.e. pre-filtered discontinuous program) using traditional calculus, automatically carrying out the symbolic convolution is challenging. Therefore, Aδ avoids explicitly applying the convolution, and directly develops math rules that approximate pre-filtering and differentiation at the same time instead. Nevertheless, symbolically smoothing a program by convolving with a smoothing kernel allows many interesting applications such as antialiasing in rendering. Therefore, this chapter takes one step into automating the process of program smoothing by approximating the program's convolution with a Gaussian kernel. The framework is then validated under the application of antialiasing procedural shader programs.

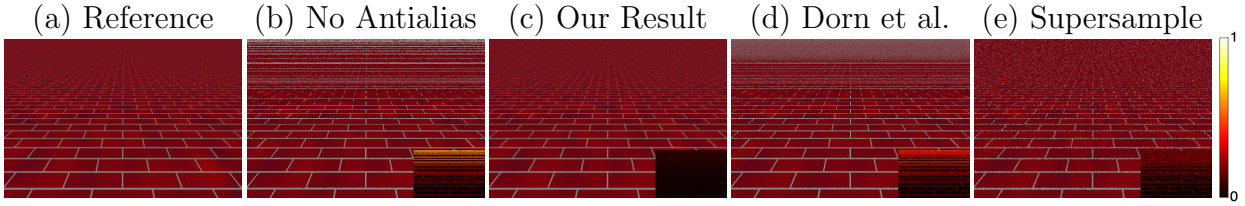


Figure 5.1: We design a novel compiler framework for smoothing programs. Here we show how our smoothing framework can be applied to bandlimiting (antialiasing) procedural shader programs. (a) is the ground truth result for a brick shader, estimated by using 1000 samples; (b) is the aliased result due to naively evaluating the original shader program; (c) is our result; (d) is the result of previous work [35] and (e) is supersampling, chosen to use comparable run-time as our result. The L^2 errors are reported in sRGB color space, with the inset heatmap depicting per-pixel L^2 error. For each method, We report the runtime and L^2 error relative to the naive aliased result (b). Our result has significantly less error, noise, and aliasing than other approaches.

5.1 Overview

In many contexts, functions that have aliasing or noise could be viewed as undesirable. This chapter develops general compiler-driven machinery to approximately smooth arbitrary programs, and thereby suppress aliasing or noise. We then apply this machinery to bandlimit procedural shader programs. To motivate our approach concretely by an application, we first discuss how procedural shaders may be bandlimited, and then return to our smoothing compiler.

Procedural shaders are important in rendering systems, because they can be used to flexibly specify material appearance in virtual scenes [4]. In this work, we focus on purely procedural shaders that do not contain texture lookups or other references to buffers. One visual error that can appear in procedural shaders is *aliasing*. *Aliasing* artifacts occur when the sampling rate is below the Nyquist limit [31]. There are two more conventional approaches used to reduce such aliasing: supersampling and mipmapping. We discuss these before discussing our smoothing compiler.

Supersampling increases the spatial sampling rate, so that the output value for each pixel is based on multiple samples. The sampling rate can either be uniform across the image, or it

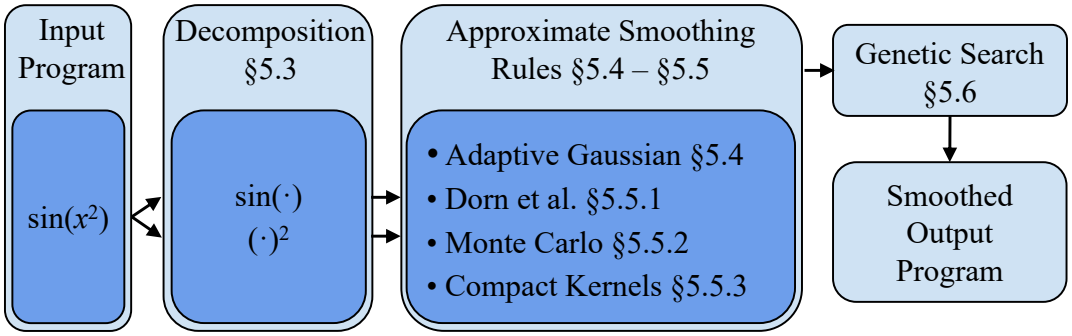


Figure 5.2: An overview of our compiler framework. The key components are: approximation rules, quality improvements, and genetic search. In approximation rules (Section 5.4 - 5.5), a variety of approximation methods are implemented to smooth the input function. All smoothed program variants are selected through a genetic search (Section 5.6), which finds a Pareto frontier that optimally trades off program running time and error.

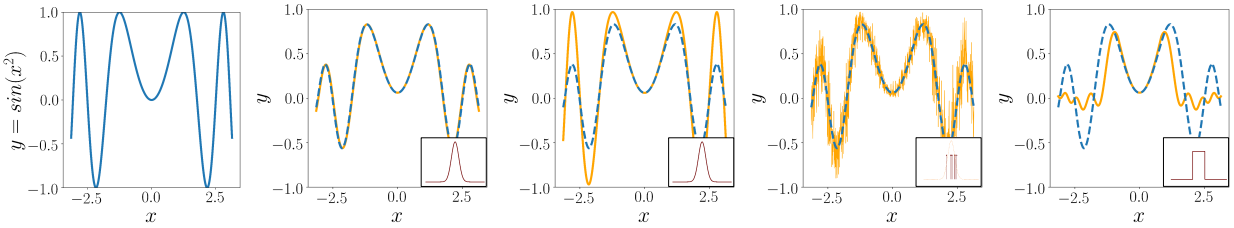
can also be chosen adaptively based on measurements such as local contrast [34, 83, 51, 84]. This approach in the limit recovers the ground truth image, but can be time-consuming due to requiring multiple samples per pixel.

Mipmapping typically stores precomputed integrals in mipmaps [141]. A similar approach can also store precomputation into summed-area tables [32]. It offers the benefit of accurate solutions with a constant number of operations, provided that the shading function can be spatially tiled or otherwise represented on a compact domain. However, in practice, many interesting shaders do not tile, so this limits the applicability of the method. Further, mipmapping increases storage requirements and may replace inexpensive computations with more expensive memory accesses. This approach is not practical for functions of more than two or three variables because memory costs scale exponentially.

Other than the conventional approaches, an alternative strategy is to construct a bandlimited variant of the shading function by symbolic integration. This can be expressed by convolving the shading function with a low-pass filter [97]. Exact analytic band-limited formulas are known for some specialized functions such as noise functions [65]. In most cases, however, the shader developer must manually calculate the convolution integral. However, frequently the integrals cannot be solved in closed form, which limits this strategy.

We take a different approach than most previous work, by using a compiler framework to smooth an input program. We show an overview of this process in Figure 5.2. Our goal is to efficiently smooth an arbitrary input function represented as a program, by accurately approximating its convolution with a Gaussian kernel. This convolution could be multidimensional: for shader programs, the dimension is typically 2D for spatial coordinates. We would also like the output program to be as efficient as possible. The compiler takes the program as input, and decomposes it into atomic parts whose bandlimited solutions are easier to obtain (Section 5.3). We then connect different smoothed atomic parts using their mean and variance statistics. Specifically, each intermediate value in the computation is treated as a random variable under a certain probability distribution, which can be modeled using their mean and variance. A key insight in our work is that modeling the random variable with their second-order variance statistic allows us to derive more accurate smoothing rules than previous work [35] that only considers first-order mean statistics. Intuitively, the second-order statistic (variance) of the input variables corresponds to the smoothing kernel size used in the convolution, and the first-order statistic (mean) of the output variables corresponds to the smoothed function value as if the original program was convolved with the smoothing kernel defined by the input random variables. Therefore, we can smooth arbitrary programs that operate over floating-point numbers. Naturally, our approach can be applied to bandlimit shader programs, by taking input of a shader with potential aliasing, and producing an output bandlimited approximation that convolves the original program with a Gaussian kernel.

A key component in our framework is the set of smoothing rules used to approximate mean and variance statistics for different atomic parts. We design a novel adaptive Gaussian approximation rule (Section 5.4) that accurately handles multivariate Gaussian distributed inputs. It is exact for a larger class of functions than previous work, and is accurate to the second power of the standard deviation for functions with certain analytic properties. The previous work of Dorn et al. [35] also provides one such approximation rule. It can



(a) Input (b) Adaptive, (Section 5.4) (c) Dorn et al., (Section 5.5.1) (d) MC sampling, (Section 5.5.2) (e) Compact, (Section 5.5.3)

Figure 5.3: A visual example of our approximate smoothing rules. (a) The input program is the function $y = f(x) = \sin(x^2)$. This program is further decomposed as the composition of two atomic parts that are easy to smooth: sin() and x^2. The "ground truth" that correctly smoothes the program ($\hat{f}(x)$) is shown in blue dashed curves in subfigures (b-e). It is evaluated by supersampling with a very high sample rate. The orange lines in subfigures (b-e) approximate the ground truth convolution by using different approximation rules: (b) adaptive Gaussian approximation (Section 5.4); (c) Dorn et al. [35] (Section 5.5.1); (d) Monte Carlo sampling approximation with 8 samples (Section 5.5.2); (e) compactly supported kernels approximation (Section 5.5.3). The dark red subplots in (b-e) give an abstract illustration of actual smoothing kernels used in each method. We use a standard deviation of $\sigma = 0.25$ for all input distributions.

be viewed as a simplified version of our adaptive Gaussian rule with less accurate modeling for the second-order variance statistic. We integrate it into our framework and improve its accuracy. We also explain a class of programs for which it gives exact results (Section 5.5.1). We further adopt Monte Carlo sampling (Section 5.5.2) to our framework. This guarantees that our approximation can resort to a known acceptable result in the worst case. For our last approximation rule, we discuss how compactly supported kernels (Section 5.5.3) can be used for parts of the computation that may be undefined for a certain region. Figure 5.3 illustrates these four different rules by applying each of them to a simple 1D function. In particular, note the previous work of Dorn et al. [35] performs poorly when the frequency of the function changes across spatial coordinates (Figure 5.3 c). This happens often for shaders because of foreshortening: frequency changes occur as a texture becomes distant from the camera.

While our primary goal is to accurately approximate the program's convolution with a Gaussian kernel, the approximation needs to be efficient as well. The various smoothing

rules introduced in Figure 5.3, therefore, provide us some freedom in exploring the space of different smoothing approximations to trade-off efficiency and accuracy. Specifically, we use a genetic search algorithm to apply different smoothing rules to individual and connected groups of atomic parts. The search algorithm finds Pareto-optimal smoothing variants that optimally trade off running time and approximation error (Section 5.6).

We apply this framework to the problem of automatically bandlimiting procedural shader programs in Section 5.7. We evaluate our method on a variety of geometries and complex shaders, including shaders with parallax mapping, animation, and spatially varying statistics. We compare the performance with Dorn et al. [35] and supersampling. Our framework gives a wider selection of band-limited programs with less error than Dorn et al. [35], and is frequently an order of magnitude faster than supersampling for comparable errors.

5.2 Related work

Mathematics and smoothing. Smoothing a function is beneficial in domains such as optimizing non-convex or non-differentiable objectives [93, 24, 23]. In numerical optimization, this approach is sometimes known as the continuation method or mollification [40, 39, 145]. In our framework, we model the smoothing process on the input program by relating the statistics of each variable, and applying a variety of approximations to smooth the program. Our idea of associating a range with each intermediate value of a program is conceptually similar to interval analysis [89]. Chaudhuri and Solar-Lezama [22] developed a smoothing interpreter that uses intervals to reason about smoothed semantics of programs. The homogeneous heat equation with initial conditions given by a nonsmoothed function results in a smoothing process, via convolution with its Green's function, the Gaussian. Thus, connections can be made between convolution with a Gaussian and result for the heat equation, such as Łysik [78] and the Hamilton-Jacobi-based Proximal [100].

Procedural shader antialiasing. The use of *antialiasing* to remove sampling artifacts is important and well-studied in computer graphics. The most general and common approach is to numerically approach the band-limited signal using supersampling [6]. Stochastic sampling [34, 31] is one effective way to achieve this. The sampling rate can be effectively lowered if it is adaptively chosen according to the contrast of the pixel [34, 83, 51, 84]. In video rendering, samples from previous frames can also be reused for computation efficiency [147]. An alternative to sample-based *antialiasing* is to create a band-limited version of a procedural shader. This can be a difficult task because analytically integrating the function is often infeasible. There are several practical approaches [36] that approximate the band-limited shader functions by sampling. This includes clamping the high-frequency components in the frequency domain [97], and producing lookup tables for static textures using mipmapping [141] and summed area tables [32].

Like our work, and unlike most other work in this area, Dorn et al. [35] use a compiler-driven technique to approximate a smoothing convolution by decomposing an arbitrary input program into atomic parts that we know how to individually smooth. Like our work, Dorn et al. uses a genetic search to select between these rules. The method of Dorn et al. performs poorly when a function changes in frequency across the spatial coordinates (as shown in Figure 5.3(b)). This happens frequently in shader programs because of foreshortening: the convolution integral is across screen-space pixels, therefore even stationary textures will have frequency changes as they become more distant from the camera. We adapt Dorn et al. as one of the approximation rules into our framework with two improvements: better standard deviation estimates and the collection of a Pareto frontier of smoothed programs instead of one single output program. Unlike Dorn et al. [35], which models only mean statistics, our framework flexibly incorporates both mean and variance statistics. We also use several approximations that have higher accuracy, which can better model textures that change in spatial frequency due to foreshortening. Our framework is general and can apply to arbitrary programs: we simply explore shaders as an example application.

Heuristic search over programs. Genetic algorithms and genetic programming (GP) are general machine learning strategies that use an evolutionary methodology to search for a set of programs that optimize some fitness criterion [63]. In computer graphics, Kensler and Shirley [61] demonstrated that genetic algorithms could be used to optimize ray-triangle intersection routines. Sitthi-Amorn et al. [121] described a GP approach to the problem of automatic procedural shader simplification. Other researchers have also investigated automatic shader simplification by heuristic search methods that simplify programs [98, 105, 54], and by jointly modifying shaders and geometry [133]. Brady and colleagues [15] showed how to use GP to discover new analytic reflectance functions. We use a similar approach as [121] to automatically generate the Pareto frontier of approximately smoothed functions.

5.3 Decomposition and Associated Notation

This section provides preliminaries to prepare the derivation of approximation rules introduced in Section 5.5. Section 5.3.1 first describes how the input program is decomposed into atomic parts. Next, Section 5.3.2 defines math notations associated with these atomic parts.

5.3.1 Decomposing the Input Program into Atomic Parts

Most input programs lack a closed-form solution for their convolution with a Gaussian kernel. We, therefore, decompose the computation graph into atomic parts that individually have known closed-form solutions. We then compute the approximate mean and variance statistics for each part, and substitute the mean and variance that are output from one group of compute nodes as the inputs for any subsequent compute nodes. We now elaborate on how this is done.

Our compiler-based framework assumes the input program has a compute graph, where each node represents a floating-point computation, and the graph is a directed acyclic graph (DAG). This compute graph is constructed directly by the programmer using atomic op-

erations such as addition, multiplication, and trigonometric functions. We use lower-case letters such as x and y to represent real values (scalars) in the input program. These can be either input, output, or intermediate values. We use corresponding capital letters such as X and Y to represent random variables associated with the scalars in the compute graph. In our implementation, we assume the random variables associated with the program input are independently Gaussian distributed. This is not limiting because dependencies such as a joint Gaussian distribution can be easily created by a linear transformation. For each node X in the computation, we use μ_X to denote its mean and σ_X^2 for its variance. Note we use these random variables as a helpful conceptual device to determine statistics, but in most cases, we never explicitly sample from these distributions (except for Monte Carlo sampling in Section 5.5.2). Our compiler then carries mean and variance computations forward through the compute graph, using the various approximate smoothing rules of Section 5.5, and collects the smoothed output value by taking the mean value of the output variable.

As an example, for shader bandlimiting, the input variables are the 2D screen coordinate (x, y), with associated random variables, X and Y. For the random variables, the means are the pixel positions, $\mu_X = X$, and $\mu_Y = y$, and the standard deviations are $\sigma_X = \sigma_Y = 0.5$, i.e. half a pixel, to suppress aliasing beyond the Nyquist rate. Our compiler then gathers the mean of the output random variables to obtain the bandlimited color.

5.3.2 Math Notation For Smoothing a Single Atomic Part

This section defines the notation we will use for smoothing a single atomic part of a program. This can be done by either using convolutions or random variables, in two equivalent notations. First, we note that throughout the paper, we use bold to indicate vectors and matrices. In some cases, we might consider the case where a random variable is scalar-valued, which we could denote as X, and then we might later consider the case of a random vector, which we might similarly denote as $\mathbf{X}$. To avoid confusion between these similar symbols, in this situation we first indicate in the text whether the variable is a scalar or vector quantity.

We now present our smoothing operator. Assume we are smoothing a function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, which maps inputs $\mathbf{x}$ to outputs $\mathbf{y}$. We use the $\hat{}$ operator to denote smoothing using convolution, so the smoothed function is $\hat{\mathbf{f}}(\mathbf{x}, \mathbf{\Sigma})$, defined as:

$$\begin{aligned}\hat{\mathbf{f}}(\mathbf{x}, \mathbf{\Sigma}) &= (\mathbf{f} * G)(\mathbf{x}) \\ &= \int_{\mathbb{R}^n} \mathbf{f}(\mathbf{x} - \mathbf{u}) G(\mathbf{u}, \mathbf{\Sigma}) d^n\mathbf{u} \\ &= \int_{\mathbb{R}^n} \mathbf{f}(\mathbf{u}) G(\mathbf{x} - \mathbf{u}, \mathbf{\Sigma}) d^n\mathbf{u}\end{aligned} \tag{5.1}$$

In Equation 5.1, $G(\mathbf{u}, \mathbf{\Sigma})$ is the smoothing kernel that is used to smooth the original function $\mathbf{f}(\mathbf{x})$, $\mathbf{\Sigma}$ is a covariance matrix associated with the kernel (more precisely, $\mathbf{\Sigma}$ is the covariance matrix of the random vector with a probability density function given by the kernel G), and the convolution is over the first variable of each function. To more explicitly identify the kernel as being G, we can also use the notation $\hat{\mathbf{f}}^G(\mathbf{x}, \mathbf{\Sigma})$. For isotropic kernels, which have the same standard deviation σ for all dimensions, we also use $\hat{\mathbf{f}}(\mathbf{x}, \sigma^2)$ as shorthand for $\hat{\mathbf{f}}(\mathbf{x}, \mathbf{I}\sigma^2)$, where $\mathbf{I}$ is the identity matrix. The convolution kernel $G(\mathbf{x}, \mathbf{\Sigma})$ can be any non-negative kernel that integrates over $\mathbb{R}^n$ to one. This allows us to interpret the kernel also as a probability density function. In our framework, we frequently use the Gaussian kernel, which we conveniently center at the origin:

$$G(\mathbf{u}, \mathbf{\Sigma}) = \frac{1}{\sqrt{(2\pi)^n |\mathbf{\Sigma}|}} \exp\left(-\frac{1}{2}\mathbf{u}^{\mathrm{T}} \mathbf{\Sigma}^{-1} \mathbf{u}\right) \tag{5.2}$$

In the case of a 1D Gaussian kernel, this simplifies to the more familiar form:

$$G(u, \sigma^2) = 1/\sqrt{2\pi\sigma^2} \exp[-u^2/(2\sigma^2)] \tag{5.3}$$

In the case of an isotropic Gaussian distribution with covariance $\mathbf{\Sigma} = \mathbf{I}\sigma^2$, where $\mathbf{I}$ is the identity matrix, this simplifies to:

$$G(\mathbf{u}, \sigma^2) = (2\pi\sigma^2)^{-n/2} \exp\left(-\left\|\mathbf{u}\right\|^2 / (2\sigma^{2n})\right) \tag{5.4}$$

If $\mathbf{f}(\mathbf{x})$ happens to be a shader program, then as is discussed in [35], $\hat{\mathbf{f}}(\mathbf{x}, \mathbf{\Sigma})$ is simply a band-limited version of the same procedural shader function.

We now show the connection between the convolution of Equation 5.1 and the random variables associated with a program's computations. We assume that in the input program, an intermediate scalar random value Y is computed by applying a scalar-valued function f to an input random vector $\mathbf{X}$ (in $\mathbb{R}^k$), i.e., $Y = f(\mathbf{X})$. If the probability density function of $\mathbf{X}$ is $g_{\mathbf{X}}$, then by the law of the unconscious statistician, μ_Y is:

$$\mu_Y = E[f(\mathbf{X})] = \int_{\mathbb{R}^k} f(\mathbf{u}) g_{\mathbf{X}}(\mathbf{u}) d^k\mathbf{u} \tag{5.5}$$

As an example, if the input random vector $\mathbf{X}$ is normally distributed as $\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}_{\mathbf{X}}, \mathbf{\Sigma}_{\mathbf{X}})$, then Equation 5.5 becomes:

$$\begin{aligned} \mu_Y &= \int_{\mathbb{R}^k} f(\mathbf{u}) G(\mathbf{u} - \boldsymbol{\mu}_{\mathbf{X}}, \mathbf{\Sigma}_{\mathbf{X}}) d^k\mathbf{u} \\ &= (f(\mathbf{u}) * G(\mathbf{u}, \mathbf{\Sigma}_{\mathbf{X}}))(\boldsymbol{\mu}_{\mathbf{X}}) \\ &= \hat{f}(\boldsymbol{\mu}_{\mathbf{X}}, \mathbf{\Sigma}_X) \end{aligned} \tag{5.6}$$

Thus, we find that we can switch between two equivalent notations. In "convolution notation," we can write $\mu_Y = \hat{f}(\boldsymbol{\mu}_{\mathbf{X}}, \mathbf{\Sigma}_{\mathbf{X}})$. This is the same as using "random variable notation" and writing $E[f(\mathbf{X})]$. This gives some intuition for how we can either use convolutions or expectations of random variables to smooth programs. One detail is that if the input random vector $\mathbf{X}$ has a different probability density function, we need to convolve with a different kernel. For example, we also consider box filters in this paper. Our framework

provides different methods to approximate the mean and variance terms. We describe them in the following section.

5.4 Adaptive Gaussian Approximation

Using the machinery from the previous section, we can now decompose the input program into atomic parts, and represent these as a directed acyclic graph (DAG). Each part accepts one or more inputs, which are modeled as Gaussian distribution according to mean and variance statistics, and outputs a variable, which is also modeled as Gaussian distribution. This section discusses our novel adaptive Gaussian approximation rule used to compute the mean and variance of the output variable. Section 5.5 further presents other approximation rules integrated into our framework that allow different trade-offs between efficiency, accuracy, and noise.

The adaptive Gaussian approximation models the input variables to a compute node as multivariate Gaussian distributions. It also approximates the output of the node as Gaussian by collecting its mean and variance statistics. Therefore, this approximation rule models the correlations between variables, allowing the variance of the output Gaussian distribution to adapt to the input as well as previous computations the input variable depends on.

Consider the function $f : \mathbb{R}^n \to \mathbb{R}$ and a jointly Gaussian distributed random vector $\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{X}, \boldsymbol{\Sigma}_\mathbf{X})$, we wish to approximate the mean and variance statistic for the scalar-valued output random variable $Y = f(\mathbf{X})$. We approximate μ_Y by convolving the function f with a Gaussian kernel:

$$\mu_Y = \hat{f}(\mu_\mathbf{X}, \boldsymbol{\Sigma}_\mathbf{X}) \tag{5.7}$$

This is the same as the result we derived in Equation 5.6. Here $\hat{f}(\mu_\mathbf{X}, \boldsymbol{\Sigma}_\mathbf{X})$ is computed from its definition in Equation 5.1. We will first discuss the more common case when $f : \mathbb{R} \to \mathbb{R}$ is a univariate function in Section 5.4.1, followed by a multivariate case in

Section 5.4.2. Finally, we will discuss the conditions when this approximation rule is exact, and establish error bounds when the rule is not exact in Section 5.4.3.

5.4.1 Smoothing Univariate Functions

Section 5.5.4 Table 5.1 summarizes the commonly used univariate functions and their corresponding smoothed form $\hat{f}$. This includes polynomials, reciprocal, sine, cosine, tangent, hyperbolic trigonometric functions, exponent, Heaviside step, fract, floor, and ceiling, and the squares of these functions. For example, if $y = \sin(x)$, and we are using a Gaussian kernel, then we can use Equation 5.7 and look up in Table 5.1 to obtain $\mu_Y = \sin(\mu_X)\exp(-\sigma_X^2/2)$. Note here we use σ_X because, for unary functions, the input X is a scalar as well.

The standard deviation is determined based on the definition of the variance of Y:

$$\begin{aligned} \sigma_Y^2 &= E[Y^2] - E[Y]^2 \\ &= \widehat{f^2}(\boldsymbol{\mu_X}, \boldsymbol{\Sigma_X}) - \hat{f}^2(\boldsymbol{\mu_X}, \boldsymbol{\Sigma_X}) \end{aligned} \tag{5.8}$$

5.4.2 Smoothing Multivariate Functions

Our compiler supports several binary functions, including the standard addition, multiplication, etc., as well as a ternary `select` operator that supports branching.

We first discuss the binary functions. Suppose a binary function $f(a, b)$ takes scalar inputs a, b and the associated random variables are A and B, respectively. We make the assumption that A and B are distributed according to a bivariate Gaussian:

$$A, B \sim \mathcal{N}\left(\begin{bmatrix}\mu_A \\ \mu_B\end{bmatrix}, \begin{bmatrix}\sigma_A^2 & \rho\sigma_A\sigma_B \\ \rho\sigma_A\sigma_B & \sigma_B^2\end{bmatrix}\right) \tag{5.9}$$

σ_A and σ_B are standard deviations of A and B. These can be determined directly by applying the approximation rules to the input nodes A and B. ρ is the correlation term between A and B, Section 5.4.2.1 details how ρ can be decided.

The mean and standard deviation for the binary functions of addition ($A+B$), subtraction ($A-B$) and multiplication ($A \cdot B$) can be derived from these assumptions based on properties of the Gaussian distribution [109]:

$$\begin{aligned}
\mu_{(A\pm B)} =& \mu_A \pm \mu_B \\
\sigma^2_{(A\pm B)} =& \sigma^2_A + \sigma^2_B \pm 2\rho\sigma_A\sigma_B \\
\mu_{(A\cdot B)} =& \mu_A\mu_B + \rho\sigma_A\sigma_B \\
\sigma^2_{(A\cdot B)} =& \mu^2_A\sigma_B + \sigma_A\mu^2_B + 2\rho\mu_A\mu_B\sigma_A\sigma_B + \sigma^2_A\sigma^2_B(1+\rho^2)
\end{aligned} \tag{5.10}$$

The binary function division $f(a,b) = a/b$ is always reduced to multiplication and a function composition: $a \cdot b^{-1}$. Here, $g(b) = b^{-1}$ is a univariate function with a singularity at $b = 0$. Technically, the mean and variance, therefore, do not exist if the Gaussian kernel is used. We work around this singularity by approximating using a compact kernel with finite support (Section 5.5.3).

The modulo function, $f_{\text{mod}}(a,b) = a\%b$, can be rewritten as $f_{\text{mod}}(a,b) = b \cdot \text{fract}(a/b)$. Here, $\text{fract}(x)$ is the fractional part of x. We make the simplifying assumption that the second argument b of mod is an ordinary (non-random) variable (so $\sigma_B = 0$), to obtain:

$$\begin{aligned}
\mu_{\text{mod}} =& \mu_B \cdot \widehat{\text{fract}}(\frac{\mu_A}{\mu_B}, \frac{\sigma^2_A}{\mu^2_B}) \\
\sigma^2_{\text{mod}} =& \mu^2_B \cdot \widehat{\text{fract}^2}(\frac{\mu_A}{\mu_B}, \frac{\sigma^2_A}{\mu^2_B}) - \mu^2_{mod}
\end{aligned} \tag{5.11}$$

Comparison functions ($>, \geq, <, \leq$) are approximated by converting them to univariate functions including the Heaviside step function $H(x)$. As an example, the function greater than ($>$) can be rewritten as $f_>(a,b) = H(a-b)$. This reduces to the univariate case in Section 5.4.1.

Finally, our compiler also supports the ternary `select`(a, b, c) function, which returns b if a is non-zero, otherwise c. We follow Dorn et al. [35] to approximate this as a linear interpolation based on binary operators: $\text{select}(a, b, c) = a \cdot b + (1 - a) \cdot c$.

5.4.2.1 Determining ρ for Bivariate Gaussian Distributions

For binary functions, we approximate the input random variables A and B as bivariate Gaussian with correlation coefficient ρ (Equation 5.9). In general, it is difficult to determine ρ, because evaluating ρ exactly involves an integral over the entire subtrees of the computation. In our framework, we provide three options to approximate ρ: (1) Assume ρ is zero; (2) Assume ρ is a constant for each node that is estimated by sampling; and (3) Estimate ρ based on a simplified assumption that the given nodes are affine functions of the inputs.

Estimate constant ρ by sampling. In the preprocessing stage, we use n samples to approximate ρ of two random variables A and B. The samples drawn from these two distributions are represented as a_i and b_i with the corresponding sample means $\overline{a}$ and $\overline{b}$. Thus, ρ can be estimated by:

$$\rho = \frac{\sum_{i=1}^{n}(a_i - \overline{a})(b_i - \overline{b})}{\sqrt{\sum_{i=1}^{n}(a_i - \overline{a})^2}\sqrt{\sum_{i=1}^{n}(b_i - \overline{b})^2}} \tag{5.12}$$

Estimate ρ by an affine assumption. We assume for the binary function f, its input variables a and b are affine transformations of the entire program's input variables x_1, ..., x_n. a and b can therefore be expressed as:

$$a = a_c + \sum_{i=1}^{n} a_i x_i, \qquad b = b_c + \sum_{i=1}^{n} b_i x_i \tag{5.13}$$

In Equation 5.13, a_c and b_c does not affect the computation of ρ and therefore will be ignored. a_i and b_i are coefficients of the affine transformation, and are approximated by

computing the gradient of a and b with respect to the inputs x_i via automatic differentiation:

$$a_i = \frac{da}{dx_i}$$
$$b_i = \frac{db}{dx_i}$$

Finally, we can express ρ as:

$$\rho = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} b_i^2}} \tag{5.14}$$

Note that the resulting ρ estimate for Equation 5.14 is exact if the nodes are actually affine with respect to the inputs, and otherwise is accurate to second order in σ.

We explored these different rules in our genetic search (Section 5.6). In practice, we find that simply using rule (1), $\rho = 0$ typically gives good results already. If the other rules (2) and (3) are also included, minor quality improvements are gained, but these rules are used relatively rarely by our genetic search process of Section 5.6.

5.4.3 Discussion on Approximation Accuracy

One natural question we may ask is, *for what functions does the adaptive Gaussian approximation rule result in the exact answer?* More precisely, we wish to determine a class of programs when recursively applying the adaptive Gaussian rule to each atomic part, the mean of the output node matches Equation 5.1 exactly. Because any affine transformation of a multidimensional Gaussian results in another multidimensional Gaussian, this approximation rule gives the exact smoothing result for any atomic function $f(\mathbf{A}x)$ that we know smoothed $\hat{f}$ for (i.e. in Table 5.1), where $\mathbf{A}$ is any affine transformation, and $\mathbf{x}$ is vector of input variables that are Gaussian distributed. Additionally, because $E[X+Y] = E[X]+E[Y]$ holds for any random variables, and $E[X \cdot Y] = E[X] \cdot E[Y]$ holds for any independent random variables X and Y, we can further extend our class of exact programs to any linear combination of $f(\mathbf{A}x)$ whose smoothed result is exact, as well as separable products with independent input variables. For example, the adaptive Gaussian rule gives exact result for

$g(x, y, z) = ((2x + y)^2 + \cos(y - 2x))z^2$ because exact smoothing results are available for polynomials and cosine.

A second question we can ask is: *if the answer is not exact, to what order is the result accurate?* Suppose for simplicity that the input variables are independent and Gaussian distributed, each with a standard deviation of σ. By using Green's function [7] on the convolution of Equation 5.1 , we can find a Taylor expansion for the function $\hat{f}(\mathbf{x}, \sigma^2)$ in terms of $f(\mathbf{x})$:

$$\hat{f}(\mathbf{x}, \sigma^2) = f(\mathbf{x}) + \frac{1}{2}\sigma^2\nabla^2 f(\mathbf{x}) + \frac{1}{(2!)2^2}\sigma^4\nabla^4 f(\mathbf{x}) + \dots \tag{5.15}$$

The derivation of Equation 5.15 assumes that f is *real analytic* on $\mathbb{R}^n$, and can be extended to a holomorphic function on $\mathbb{C}^n$, so that all the derivatives exist, and the Taylor series has an infinite radius of convergence [137]. This class of functions includes polynomials, sines, cosines, and compositions of these. The function should also be bounded by exponentials: the precise conditions are discussed by Łysik [78]. These properties could hold for some shader programs, but even if they do not hold for an entire program, they often hold for program sub-parts.

For a univariate function, we show our approximation is accurate up to σ^2 terms for a single function composition. Multiple function compositions can be proved similarly via induction.

Suppose we wish to approximate the composition of two functions: $f(x) = f_2(f_1(x))$, where $f_1, f_2 : \mathbb{R} \to \mathbb{R}$. Assume the input random variable is $X_0 \sim \mathcal{N}(x, \sigma^2)$: the Gaussian kernel centered at x. The output from f_1 is an intermediate value in the computation: we can represent this with another random variable $X_1 = f_1(X_0)$. We can similarly represent the output of f_2: $X_2 = f_2(X_1) = f_2(f_1(X_0)) = f(X_0)$.

We apply Equation 5.8 and Equation 5.15 to $X_1 = f_1(X_0)$ and derive the following mean and standard deviation for X_1.

$$\begin{aligned}
\mu_{X_1} &= \hat{f}_1(x, \sigma^2) \\
&= f_1(x) + \frac{1}{2}\sigma^2 f_1^{''}(x) + \mathcal{O}(\sigma^4) \\
\widehat{f_1^2}(x, \sigma^2) &= f_1^2(x) + \frac{1}{2}\sigma^2 \frac{\partial^2}{\partial x^2}(f_1^2(x)) + \mathcal{O}(\sigma^4) \\
&= f_1^2(x) + \frac{1}{2}\sigma^2 (2 f_1 f_1^{''} + 2(f_1^{'})^2)(x) + \mathcal{O}(\sigma^4) \\
\sigma_{X_1}^2 &= \widehat{f_1^2}(x, \sigma^2) - \hat{f}_1(x, \sigma^2)^2 \\
&= \sigma^2 (f_1^{'})^2(x) + \mathcal{O}(\sigma^4)
\end{aligned} \tag{5.16}$$

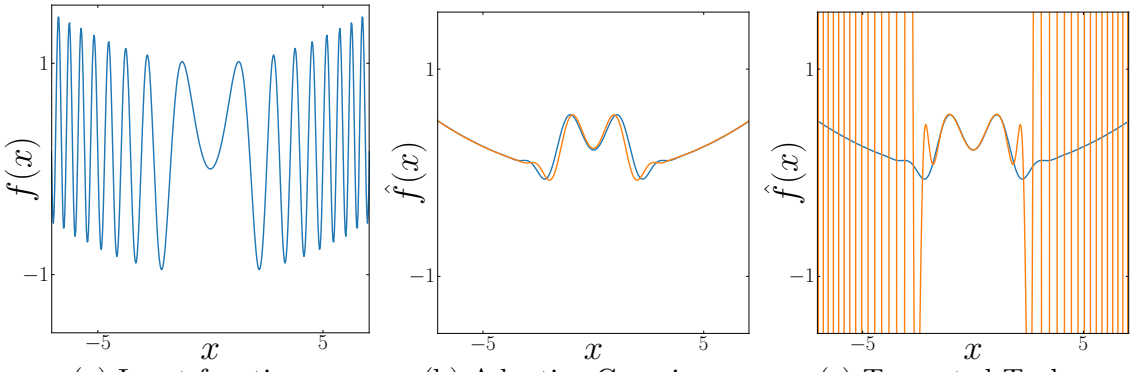
Our rule approximate X_1 as a Gaussian distribution with mean and variance from Equation 5.16: $\mathcal{N}(\mu_{X_1}, \sigma_{X_1}^2)$. We similarly compute μ_{X_2} based on Equation 5.15, Equation 5.16, and repeated Taylor expansion in σ around $\sigma = 0$.

$$\begin{aligned}
\mu_{X_2} =& \hat{f}_2(\hat{f}_1(x, \sigma^2), \sigma_{X_1}^2) \\
=& f_2(f_1(x) + \frac{1}{2}\sigma^2 f_1^{''}(x) + \mathcal{O}(\sigma^4)) + \frac{1}{2}\sigma_{X_1}^2 f_2^{''}(\hat{f}_1(x, \sigma^2)) + \mathcal{O}(\sigma_{X_1}^4) \\
=& f(x) + \frac{1}{2}\sigma^2 f_2^{'}(f_1(x)) f_1^{''}(x) + \frac{1}{2}\sigma^2 f_2^{''}(f_1(x))(f_1^{'})^2(x) + \mathcal{O}(\sigma^4) \\
=& f(x) + \frac{1}{2}\sigma^2 f^{''}(x) + \mathcal{O}(\sigma^4)
\end{aligned} \tag{5.17}$$

Comparing Equation 5.17 with Equation 5.15, we conclude our approximation agrees up to the second order term in the Taylor expansion.

This property can be similarly proved via induction for multiple function compositions. We, therefore, conclude that for functions with certain analytic properties, the adaptive Gaussian rule is accurate to σ^2.

Note there are also other second-order accurate approximations, such as simply truncating the Taylor expansion in Equation 5.15 to use only the first and second terms. Figure 5.4 gives



(a) Input function

(b) Adaptive Gaussian approximation (§5.4) $\sigma = 0.5$

(c) Truncated Taylor expansion (Equation 5.15) $\sigma = 0.5$

Figure 5.4: A comparison of different approximation techniques. (a) The input function $f(x) = \sin(x^2) + \frac{1}{100}x^2$. The ground truth correctly band-limited functions $\hat{f}(x)$ are shown in blue in subfigures (b-c). These were determined by sampling at a high sample rate. (b) Our adaptive Gaussian approximation (Section 5.4) is shown in orange and compared against the ground truth in blue. The approximation is good. (c) A truncated Taylor expansion with 10 terms does not result in smoothing.

an example that our adaptive Gaussian approximation is still more accurate. The truncated Taylor expansion results in amplifying high frequencies, instead of attenuating them.

5.5 Additional Approximate Smoothing Rules

This section discusses several other approximation rules used to compute the mean and variance of the output variable. They allow different trade-offs between efficiency, accuracy, and noise. Different rules are visualized in Figure 5.3.

5.5.1 Approximation of Dorn et al. 2015

We integrate the approximation rule described in Dorn et al. [35] as one of our approximation options. Similar to our adaptive Gaussian rule (Section 5.4), Dorn's rule also approximates function smoothing by convolving with a Gaussian kernel. Suppose an intermediate scalar variable y is computed from another scalar variable x, and the associated random variables are Y and X, respectively, where $Y = f(X)$. Then μ_Y is identical to the univariate case of

Equation 5.7 in the Gaussian adaptive rule:

$$\mu_Y = \hat{f}(\mu_X, \sigma_X^2) \tag{5.18}$$

Different from the adaptive Gaussian rule, Dorn's rule makes simplified assumptions to determine σ_Y: the output σ is a linear combination of the axis-aligned input σs in each dimension. They are determined by simple rules such as σ for addition and subtraction is the sum of input σs, and σ for multiplication or division is the product or quotient, respectively, of the input σs. In all other cases, including function calls, the output σ is the average of the non-zero σs of all the inputs.

We further make two improvements to Dorn et al. [35] before integrating their rule into our framework. The first improvement gives better standard deviation estimates, and the second collects a Pareto frontier. For the standard deviations (known as "sample spacing" in Dorn et al. [35]), we detect the case of multiplication or division by a constant and adjust the standard deviation accordingly (i.e. $\sigma_{aX} = |a|\sigma_X$). This improvement helps give more accurate estimates of the standard deviations and therefore reduces the artifact seen in Dorn et al. [35] Figure 5(c), where their approximation has substantially wrong variances. Our second improvement is to collect not just a single program variant with the least error, but instead a Pareto frontier of program variants that optimally trade off running time and error. This process is described later in Section 5.6.

5.5.1.1 Discussion on Approximation Accuracy

Similar to Section 5.4.3, we can also ask: *for what class of functions does the improved Dorn et al. [35] approximation result in the exact answer?* Even for linear functions, Dorn's approximation gives incorrect variance. For example, Dorn's estimate incorrectly approximates $\text{Var}(X - X)$ as $(2\sigma_X)^2$ when it should be zero. However, if a Gaussian distributed input variable is multiplied by or added to a constant, this rule results in the correct mean

and variance. Therefore, this rule gives exact smoothing results for linear combinations or separable products of functions $f(ax+b)$ where the smoothing result $\hat{f}$ is exact, and a, b are constant. For example, Dorn's rule is exact for $g(x,y,z) = ((2x)^2 + \cos(y))z^2$ because exact smoothing results are available for polynomials and cosine. Note this is a smaller class of programs than our adaptive Gaussian rule presented earlier because the argument to f is only univariate affine $(ax+b)$ rather than multivariate affine.

5.5.2 Monte Carlo Sampling

We adapt Monte Carlo stochastic sampling [29, 34] to our framework. The compiler first identifies each largest connected sub-graph of the compute graph where the nodes within sub-graphs are specified to use Monte Carlo sampling. For each sub-graph f, we then identify their inputs, $X_1, \ldots, X_m$. These could either be input random variables to the entire input program, or intermediate variables calculated as the output from other parts of the compute graph. For simplicity, here we assume these inputs are independent Gaussian distributions based on their specified means μ_{X_i} and standard deviations σ_{X_i}. For each output of the sub-graph f: $Y = f(X_1, ..., X_m)$, we compute its mean and standard deviation by sampled estimators:

$$\begin{aligned} \mu_Y =& \frac{1}{n}\sum_{i=1}^{n} f(\mu_{X_1} + \mathcal{N}_{i,1}\sigma_{X_1}, ..., \mu_{X_m} + \mathcal{N}_{i,m}\sigma_{X_m}) \\ \sigma_Y^2 =& \frac{1}{n}\sum_{i=1}^{n} f^2(\mu_{X_1} + \mathcal{N}_{i,1}\sigma_{X_1}, ..., \mu_{X_m} + \mathcal{N}_{i,m}\sigma_{X_m}) - \mu_Y^2 \end{aligned} \tag{5.19}$$

Here, each $\mathcal{N}_{ij}$ is a random number independently drawn from a normal distribution $\mathcal{N}(0,1)$, and n is the number of samples. We also experimented with applying Bessel's correction [124] to correct the bias in variance that occurs for small sample counts n. But we find in practice, it does not have a significant improvement on the result for our system.

The approximation converges to the ground truth for large sample numbers, and the output program simplifies to supersampling [29] when the entire input program is approximated

using the Monte Carlo sampling. The error of the Monte Carlo sampling σ_M is estimated as [41]:

$$\sigma_M \approx \frac{\sigma_Y}{\sqrt{n}} \tag{5.20}$$

Here, σ_Y is the standard deviation computed from Equation 5.19 and n is the number of samples. This approximation rule becomes more accurate in the limit of large sample numbers.

5.5.2.1 Optional Denoising

For small sample count, Monte Carlo approximation can be noisy. A variety of techniques have been developed to filter such noise [60, 8, 116]. Specifically, we implement the non-local means denoising [16, 17] with Laplacian pyramid [74] for optional denoising. We find that aesthetically appealing denoising results can be obtained using a three-level Laplacian pyramid, with a patch size of 5, a search radius of 10, and the denoising parameter h is 10 for the lower resolutions, and searched over or set by the user for the finest resolution. We allow our genetic search process (Section 5.6) to search over a variety of denoising parameters for the best result. However, because the denoising algorithm incurs time overhead, it usually does not optimally trade off accuracy and efficiency, therefore being only rarely chosen. Therefore, in our current setup, denoising is typically specified manually.

5.5.3 Compactly Supported Kernels Approximation

Because the Gaussian kernel has infinite support, it cannot be used on functions with undefined regions. For example, $\sqrt{x}$ is only defined on non-negative x, and its convolution with a Gaussian using Equation 5.1 does not exist. However, even if an input program contains such functions as sub-parts, the entire program may still have a well-defined result, so smoothing should still be possible for such programs. To handle this case, we introduce compactly supported kernels.

Results for certain compactly supported kernels can be obtained by using repeated convolution [55] of boxcar functions. This is because such kernels approximate the Gaussian by the central limit theorem [135]. In our framework, we use box and tent kernels to approximately smooth functions with undefined values. Because the convolution with a box kernel is easier to compute, this approximation can also be used when the Gaussian convolution does not have a closed-form solution. Table 5.1 lists smoothing results for commonly used functions with the box kernel.

When integrating against a function that has an undefined region, it is important to make sure that the integral range does not intersect with the undefined regions. We solve this by adapting the kernel size based on where this integral is evaluated at. As a result, technically the integral is no longer a convolution, because it is not shift-invariant. When evaluating the integral at x, we first compute the distance r from x to the function's nearest undefined point. If the convolution kernel's original half-width is h, we will rescale it to $\min(h, \lambda r)$ where λ is a constant less than one. In practice, we use $\lambda = 0.5$.

We can further utilize this truncation mechanism to better model functions such as $\text{fract}(x) = x - \lfloor x \rfloor$, which have many discontinuities. Clearly, fract() is discontinuous whenever x is at an integer value. If we naively input a distribution that spans a discontinuity, such as $X \sim \mathcal{N}(0, 0.1^2)$, into fract(), the output $Y = \text{fract}(X)$ becomes bimodal, with some values close to zero, and others close to one. Directly computing the mean of this bimodal distribution results in 0.5, which is far away from either of the two modes. This may result in a poor approximation, which can show up in tiled pattern shaders (where fract is used for tiling) as a bias towards the center of the tile's texture. One potential fix would be to stochastically sample either mode. However, this introduces sampling noise. Instead, we truncate the filter at the discontinuity when the original kernel support is smaller than a truncation constant T_1 (in practice, we use $T_1 = 1/4$). When the original kernel support is above a larger truncation constant T_2 (we use $T_2 = 1/2$) we do not truncate. In between kernel sizes T_1 and T_2 we rescale the kernel size linearly between these endpoints.

5.5.4 Summary of Atomic Function Smoothing

Table 5.1 summarizes functions and their corresponding convolutions with box and Gaussian kernels. These are needed for the approximations we developed previously. This table can be viewed as an extension of the table presented in Dorn et al. [35]. In particular, for each function $f(x)$, we smooth both $f(x)$ as well as $f^2(x)$ (e.g. if we report $\cos(x)$ then we also report $\cos^2(x)$). This is needed to determine the standard deviations output by a given compute stage for the adaptive Gaussian approximation rule of Section 5.4.

In the remainder of the section, we discuss the derivation of two types of functions in Table 5.1: the polynomials x^n and the periodic functions.

Table 5.1: A table of univariate functions, and their corresponding bandlimited result, using a box kernel B and a Gaussian G. The box kernel is the PDF of the uniform random variable $U[-\sqrt{3}\sigma, \sqrt{3}\sigma]$. The Gaussian kernel is the PDF of the random variable $\mathcal{N}(0, \sigma^2)$. Each random variable has a standard deviation σ. We define $\text{sinc}(x) = \sin(x)/x$, and the Heaviside step function $H(x)$ is 0 for $x \leq 0$ and 1 for x positive. Note that functions with undefined regions, such as x^p for negative or fractional p have σ limited as described in Section 5.5.3.

Function $f(x)$	Box kernel $\hat{f}^B(x, \sigma^2)$	Gaussian kernel $\hat{f}^G(x, \sigma^2)$
$x^p, p \neq -1$	$\frac{1}{\sqrt{12}\sigma(p+1)}\left[(x+\sqrt{3}\sigma)^{p+1} - (x-\sqrt{3}\sigma)^{p+1}\right]$	$He_p^{[-\sigma^2]}(x)$
x^{-2}	$(x^2 - 3\sigma^2)^{-1}$	
x^{-1}	$\frac{1}{\sqrt{12}\sigma}\log\left\lvert\frac{x+\sqrt{3}\sigma}{x-\sqrt{3}\sigma}\right\rvert$	
x	x	x
x^2	$x^2 + \sigma^2$	$x^2 + \sigma^2$
x^3	$x^3 + 3x\sigma^2$	$x^3 + 3x\sigma^2$
x^4	$x^4 + 6x^2\sigma^2 + \frac{9}{5}\sigma^4$	$x^4 + 6x^2\sigma^2 + 3\sigma^4$
x^5	$x^5 + 10x^3\sigma^2 + 9x\sigma^4$	$x^5 + 10x^3\sigma^2 + 15x\sigma^4$
$\sqrt{x}$	$\frac{1}{3\sqrt{3}\sigma}\left[(x+\sqrt{3}\sigma)^{3/2} - (x-\sqrt{3}\sigma)^{3/2}\right]$	
$\sin(x)$	$\sin(x)\,\text{sinc}(\sqrt{3}\sigma)$	$\sin(x)e^{-\frac{\sigma^2}{2}}$
$\cos(x)$	$\cos(x)\,\text{sinc}(\sqrt{3}\sigma)$	$\cos(x)e^{-\frac{\sigma^2}{2}}$
$\tan(x)$	$\frac{-1}{\sqrt{12}\sigma}\log\left\lvert\frac{\cos(x+\sqrt{3}\sigma)}{\cos(x-\sqrt{3}\sigma)}\right\rvert$	
$\sinh(x)$	$\frac{1}{\sqrt{12}\sigma}(\cosh(x+\sqrt{3}\sigma) - \cosh(x-\sqrt{3}\sigma))$	$\frac{1}{2}(e^{x+\frac{1}{2}\sigma^2} - e^{-x+\frac{1}{2}\sigma^2})$
$\cosh(x)$	$\frac{1}{\sqrt{12}\sigma}(\sinh(x+\sqrt{3}\sigma) - \sinh(x-\sqrt{3}\sigma))$	$\frac{1}{2}(e^{x+\frac{1}{2}\sigma^2} + e^{-x+\frac{1}{2}\sigma^2})$

(Continued on next page)

Table 5.1: (continued)

Function $f(x)$	Box kernel $\hat{f}^B(x,\sigma^2)$	Gaussian kernel $\hat{f}^G(x,\sigma^2)$
$\tanh(x)$	$\frac{1}{\sqrt{12}\sigma}(\log(\cosh(x+\sqrt{3}\sigma)) - \log(\cosh(x-\sqrt{3}\sigma)))$	
$\sinh^2(x)$	$\frac{1}{8\sqrt{3}\sigma}(-4\sqrt{3}\sigma + \sinh(2\sqrt{3}\sigma - 2x) + \sinh(2\sqrt{3}\sigma + 2x))$	
$\cosh^2(x)$	$\frac{1}{8\sqrt{3}\sigma}(4\sqrt{3}\sigma + \sinh(2\sqrt{3}\sigma - 2x) + \sinh(2\sqrt{3}\sigma + 2x))$	
$\tanh^2(x)$	$\frac{1}{2\sqrt{3}\sigma}(2\sqrt{3}\sigma - \tanh(\sqrt{3}\sigma - x) - \tanh(\sqrt{3}\sigma + x))$	
e^x	$\frac{1}{\sqrt{12}\sigma}\left(e^{x+\sqrt{3}\sigma} - e^{x-\sqrt{3}\sigma}\right)$	$e^{x+\frac{1}{2}\sigma^2}$
$\sin^2(x)$	$\frac{1}{2} - \frac{1}{2}\cos(2x)\operatorname{sinc}(\sqrt{12}\sigma)$	$\frac{1}{2} - \frac{1}{2}\cos(2x)e^{-2\sigma^2}$
$\cos^2(x)$	$\frac{1}{2} + \frac{1}{2}\cos(2x)\operatorname{sinc}(\sqrt{12}\sigma)$	$\frac{1}{2} + \frac{1}{2}\cos(2x)e^{-2\sigma^2}$
$\tan^2(x)$	$\frac{1}{\sqrt{12}\sigma}\left(\tan(x+\sqrt{3}\sigma) - \tan(x-\sqrt{3}\sigma)\right) - 1$	
$H(x)$	$\begin{cases} 0 & x \le -\sqrt{3}\sigma \\ \frac{x}{2\sqrt{3}\sigma} + \frac{1}{2} & -\sqrt{3}\sigma \le x \le \sqrt{3}\sigma \\ 1 & x \ge \sqrt{3}\sigma \end{cases}$	$\frac{1}{2}(1 + \operatorname{erf}\frac{x}{\sqrt{2}\sigma})$
$\operatorname{fract}(x)$	$\frac{1}{\sqrt{48}\sigma}(\operatorname{fract}^2(x+\sqrt{3}\sigma) + \lfloor x+\sqrt{3}\sigma \rfloor - \operatorname{fract}^2(x-\sqrt{3}\sigma) - \lfloor x-\sqrt{3}\sigma \rfloor)$	
$\operatorname{fract}^2(x)$	$\frac{1}{\sqrt{108}\sigma}(\operatorname{fract}^3(x+\sqrt{3}\sigma) + \lfloor x+\sqrt{3}\sigma \rfloor - \operatorname{fract}^3(x-\sqrt{3}\sigma) - \lfloor x-\sqrt{3}\sigma \rfloor)$	
$\lfloor x \rfloor$	$x - \widehat{\operatorname{fract}}(x)$	
$\lfloor x \rfloor^2$	$\widehat{x^2} + \widehat{\operatorname{fract}^2}(x) - F(x+\sqrt{3}\sigma) + F(x-\sqrt{3}\sigma)$ where $F(x) = 2(\frac{\lfloor x \rfloor}{3} + \frac{\lfloor x \rfloor(\lfloor x \rfloor - 1)}{4} + \frac{\lfloor x \rfloor \widehat{\operatorname{fract}^2}(x)}{2} + \frac{\widehat{\operatorname{fract}^3}(x)}{3})$	
$\lceil x \rceil$	$x + \widehat{\operatorname{fract}}(-x)$	
$\lceil x \rceil^2$	$\widehat{\lfloor -x \rfloor^2}$	

Polynomials. The bandlimiting result for x^n is derived from the property of generalized Hermite polynomial $He_n^{[\alpha]}(x)$: the nth noncentral moment of a Gaussian distribution X with expected value μ and variance σ is a generalized Hermite polynomial [138]:

$$He_n^{[\sigma]}(\mu) = \sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} \frac{n!}{(n-2k)!k!}(-2)^{-k}\mu^{n-2k}\sigma^k \tag{5.21}$$

Periodic Functions. We also derive a convenient formula that gives the bandlimited result for any periodic function if its integral within a single period is known. We extend the analysis of fract() made by Dorn et al. [35] to any periodic function. We use Heckbert's

technique of repeated integration [55] to derive the convolution of a periodic function with a box kernel.

Specifically, we assume the periodic function $f(x)$ has a period T and its first and second integrals within one period are also known. These are denoted as $F_p(x)$ and $F_{p2}(x)$, respectively.

$$\begin{aligned} F_p(x) &= \int_0^x f(u)du \\ F_{p2}(x) &= \int_0^x F_p(u)du \\ x &\in [0, T) \end{aligned} \tag{5.22}$$

Using Equation 5.22, we derive the first and second integral of $f(x)$ for any x as follows.

$$\begin{aligned} F(x) &= \int_0^x f(u)du \\ &= \left(\left\lfloor \frac{x}{T} \right\rfloor + 1\right) \cdot F_p(T) - \int_{x - T \cdot \left\lfloor \frac{x}{T} \right\rfloor}^{T} f(u)du \\ &= \left(\left\lfloor \frac{x}{T} \right\rfloor + 1\right) \cdot F_p(T) - F_p(T) + F_p\left(x - T \cdot \left\lfloor \frac{x}{T} \right\rfloor\right) \\ &= \left\lfloor \frac{x}{T} \right\rfloor \cdot F_p(T) + F_p\left(x - T \cdot \left\lfloor \frac{x}{T} \right\rfloor\right) \end{aligned} \tag{5.23}$$

$$\begin{aligned} F_2(x) &= \int_0^x F(u)du \\ &= \int_0^x \left\lfloor \frac{u}{T} \right\rfloor \cdot F_p(T)du + \int_0^x F_p(u - T \cdot \left\lfloor \frac{u}{T} \right\rfloor)du \\ &= F_p(T) \cdot T \sum_{i=0}^{\left\lfloor \frac{x}{T} \right\rfloor - 1} i + \left(x - T \left\lfloor \frac{x}{T} \right\rfloor\right) \cdot \left\lfloor \frac{x}{T} \right\rfloor \cdot F_p(T) + \left\lfloor \frac{x}{T} \right\rfloor \cdot F_{p2}(T) + F_{p2}\left(x - T \left\lfloor \frac{x}{T} \right\rfloor\right) \\ &= F_p(T) \cdot \left(\frac{T \cdot (q-1) \cdot q}{2} + (x - T \cdot q) \cdot q\right) + F_{p2}(T) \cdot q + F_{p2}(x - T \cdot q) \end{aligned}$$

Here, $q = \left\lfloor \frac{x}{T} \right\rfloor$. (5.24)

Using Heckbert's result, the convolution of the periodic function $f(x)$ with a box kernel that has support $[-\sqrt{3}\sigma, \sqrt{3}\sigma]$ (corresponding to a uniform kernel with standard deviation σ) is:

$$\hat{f}(x, \sigma) = \frac{F(x + \sqrt{3}\sigma) - F(x + \sqrt{3}\sigma)}{2\sqrt{3}\sigma} \tag{5.25}$$

The convolution of the periodic function $f(x)$ with a tent kernel that has support $[-\sqrt{6}\sigma, \sqrt{6}\sigma]$ (corresponding to a uniform kernel with standard deviation σ) is:

$$\hat{f}(x, \sigma) = \frac{F_2(x + \sqrt{6}\sigma) - 2 \cdot F_2(x) + F_2(x - \sqrt{6}\sigma)}{6\sigma^2} \tag{5.26}$$

5.6 Genetic Search

This section describes the genetic search algorithm that automatically assigns approximation rules to each computation node. The algorithm finds the Pareto frontier of approximation choices that optimally trade off the running time and error of the program.

We developed this genetic search because it allows users to explore the trade-off between the efficiency and accuracy of the smoothed program. Although developers can manually assign approximation rules, we found this to be a time-consuming process that can easily overlook beneficial approximation combinations. This is because the search space for the approximations is combinatoric.

Our genetic search closely follows the method of Sitthi-Amron et al. [121]. We adopt their fitness function and tournament selection rules, and we use the same method to compute the Pareto frontier of program variants that optimally trade off running time and error with ground truth, which is precomputed with a high sample count at randomly sampled pixel coordinates.

We start with "decent initial guesses." For each approximation rule, we create a program variant where the rule is applied to all the expression nodes. For such initial guesses, we also apply single-point cross-over. The cross-over operation partitions the program into two

parts separated by an arbitrary node, assigns approximation rules from one variant to the first part of the program, and rules from another variant to the other part. Next, we employ cross-over and mutation operations to explore the search space. The mutation step chooses a new approximation rule, and with equal probability, assigns this new rule to 1, 2, or 4 adjacent expression nodes in depth-first order. As an alternative, with equal probability, the new approximation rule can also be assigned to the whole subtree of an arbitrary node. We use tournament selection to select program variants for mutation and crossover. Our tournament selection works by randomly sampling 4 program variants from the population, eliminating variants that are not Pareto optimal, and then randomly choosing a remaining program with optimal running time and error.

For the Monte Carlo sampling approximation, during initialization and mutation, we select sample counts with equal probability from the set $\{2, 4, 8, 16, 32\}$. For the determination of correlation coefficients described in Section 5.4, we pick with equal probability one of the three options.

5.7 Evaluation

We author 21 shaders and apply them to three geometries to provide a more challenging and realistic benchmark than Dorn et al. [35], which is only evaluated on planar geometry with relatively simple shaders. Our shaders include the Phong lighting model, animation, spatially varying statistics, and parallax mapping. Specifically, we implement the "safer mapping" formula from Section 4.1.4 of Szirmay-Kalos and Umenhoffer [126] for parallax mapping. Our 21 shaders were produced by combining 7 *base shaders* with 3 choices for parallax mapping: none, bumps, and ripples. In Table 5.2, we describe our base shaders, the choices for parallax mapping, and the corresponding program complexity. We apply the shaders on 3 different geometries: an infinite plane and two curved geometries: sphere and hyperboloid. Each shader program is tuned independently on each of the geometries.

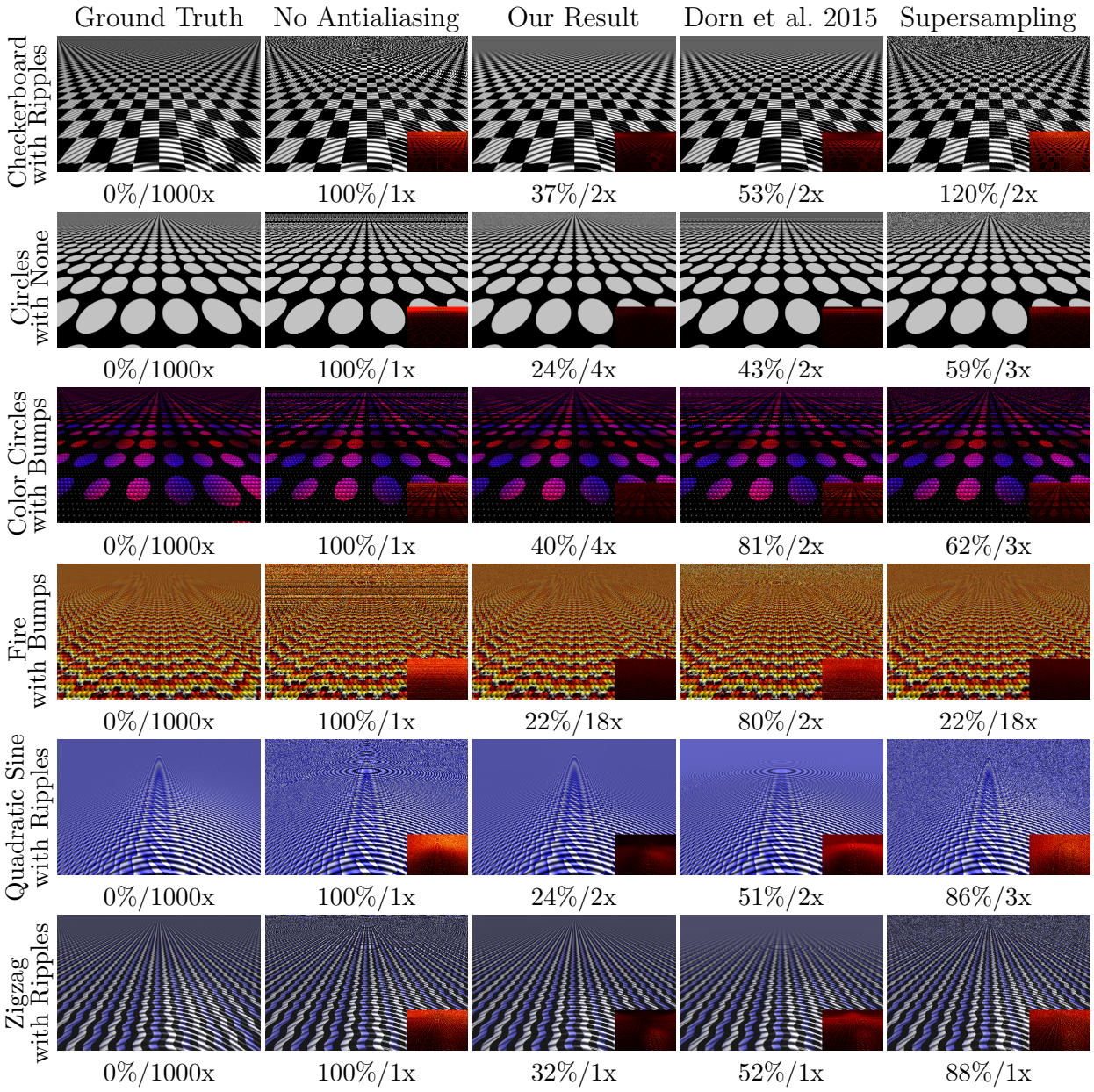


Figure 5.5: Selected result images for 6 shaders on an infinite plane. Reported below each shader are L^2 error and run-time relative to the naive no antialiasing baseline as in Figure 5.1. Please zoom in to see aliasing and noise patterns in the different methods. Program variants with comparable time were selected: see Section 5.7.1 for more details. Note that the amount of aliasing and error for our result is significantly less than Dorn et al. [35]. We typically have significantly less error and noise than the comparable supersampled results. Please zoom in to see details.

Table 5.2: A table of our 21 shaders. At the top, we list our 7 *base shaders*, which are each combined with 3 different choices for parallax mapping, listed at the bottom. We also report the number of non-comment lines and expressions in each program fragment.

Shader	Lines	Exprs	Description
Base shaders			
Bricks	38	192	Bricks with noise pattern
Checkerboard	20	103	Greyscale checkerboard
Circles	16	53	Tiled greyscale circles
Color circles	26	164	Aperiodic colored circles
Fire	49	589	Animating faux fire
Quadratic sine	26	166	Animating sine of quadratic
Zigzag	24	224	Colorful zigzag pattern
Parallax mappings			
None	0	0	No parallax mapping
Bumps	21	203	Spherical bumps
Ripples	23	178	Animating ripples

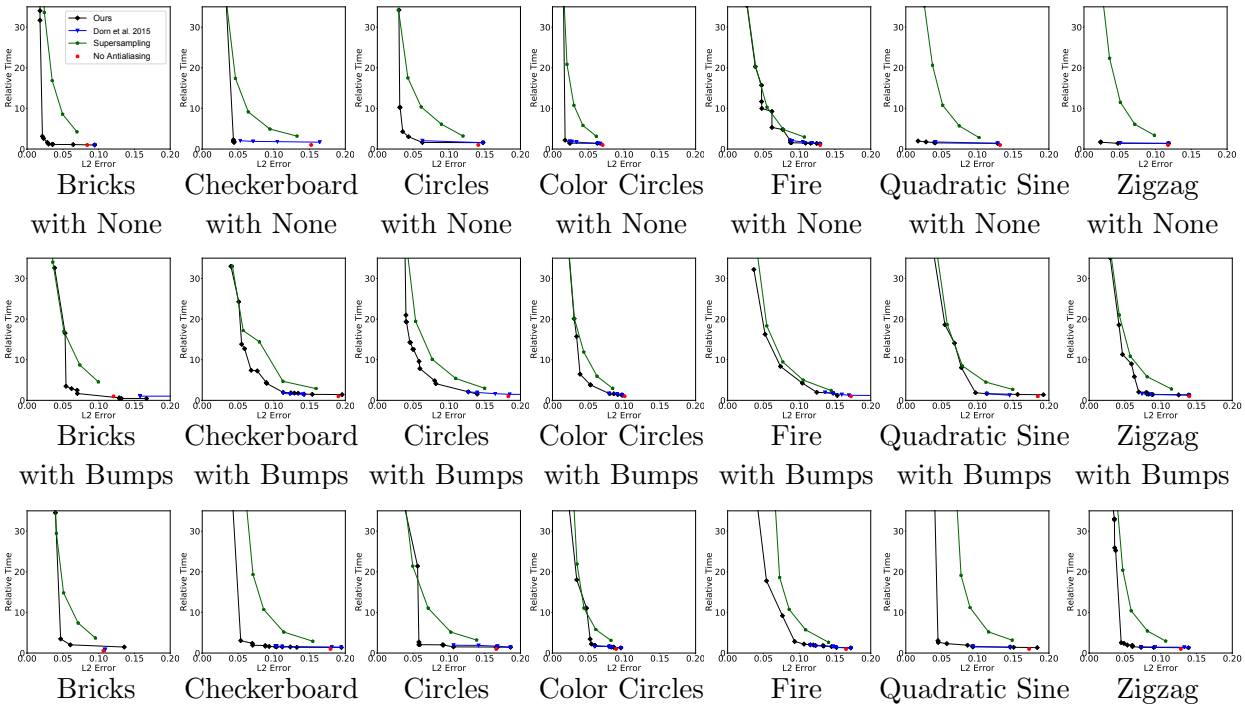


Figure 5.6: Time versus error plots for the 21 shaders defined in this the paper, and applied to planar geometry. Here we show the Pareto frontier of program variants that optimally trade off running time and L^2 error. We show results for our method, Dorn et al. [35], supersampling with varying numbers of samples, and the input shader without antialiasing. Note that our approach typically has significantly less error than Dorn et al. [35] and is frequently an order of magnitude faster than supersampling for comparable error.

We evaluate on an Intel Core i7 6950X 3 GHz (Broadwell), with 10 physical cores (20 hyperthreaded), and 64 GB DDR4-2400 RAM. All shaders are evaluated on the CPU using parallelization. The tuning of each shader took between 1 and 6 hours of wall clock time, with 1 to 3 hours for planar geometry. However, we note that good program variants are typically available after minutes to low tens of minutes, and most of the remaining tuning time is spent making slight improvements to the best individuals. Please see Section 5.7.3 for results available after tuning for 10 minutes. Also, our tuner is intentionally a research prototype that is not particularly optimized: it could be significantly faster if the code generator were optimized, it was parallelized more effectively, cached more redundant computations, or targeted the GPU. We found that getting the code generation and math details right was challenging, so we only target CPU code for simplicity in our prototype.

5.7.1 Planar Geometry

We first evaluate shaders on an infinite plane. Figure 5.1 and 5.5 show 7 of our shaders, including one result for each base shader. The result for our method is selected by a human choosing for each shader a program variant that has a sufficiently low error. Dorn et al. [35] typically cannot reach sufficiently low errors to remove the aliasing, so we simply select the program variant from Dorn et al. that reaches the lowest error. The supersampling result is selected such that its runtime is the most similar to ours. Note for supersampling, the times relative to no antialiasing do not exactly match the sample count due to cache effects and variations in the running time depending on exactly where samples intersect geometry.

Figure 5.6 presents the time versus error plots for the Pareto frontiers associated with all 21 shaders. Note that Dorn et al. typically has a significantly higher error, which manifests in noticeable aliasing. Also, note that the supersampling method frequently takes an order of magnitude more time for equal error.

Statistics for the approximations used are presented in Table 5.3. Note that a rich variety of approximation strategies are used: all approximation choices are selected for different

Table 5.3: Statistics of which approximations were chosen for different shaders on an infinite plane. We show statistics for the 7 program variants for the shaders presented in Figure 5.1 and Figure 5.5. We also show aggregate statistics over all 21 shaders, with each shader's contribution weighted equally. We report aggregate statistics from the entire Pareto frontier, as well as for each shader choosing only the slowest, fastest, or median speed program variant. Our results show that a rich variety of our different approximation rules are needed for the best performance.

Shader	Dorn et al. [35]	Adaptive Gaussian	Monte Carlo Sampling	None
Bricks w/ None	28%	0%	30%	29%
Checkerboard w/ Ripples	66%	34%	0%	1%
Circles w/ None	4%	21%	71%	4%
Color Circles w/ Bumps	8%	47%	44%	0%
Fire w/ Bumps	1%	7%	33%	60%
Quadratic sine w/ Ripples	13%	80%	0%	8%
Zigzag w/ Ripples	0%	91%	1%	8%
All shaders (Pareto frontier)	29%	15%	25%	30%
All shaders (fastest time)	13%	10%	0%	77%
All shaders (median time)	20%	19%	49%	13%
All shaders (slowest time)	10%	27%	49%	14%

programs (compactly supported kernel is only applied to functions with singularities, and is therefore not part of the search). Adaptive Gaussian and Monte Carlo sampling are important for high accuracy but less so for fast running time, as indicated in the bottom row of Table 5.3. In contrast, the Dorn et al. approximation is mainly useful when a fast running time is desired in exchange for higher error. For the correlation term discussed in Section 5.4.2.1, nearly all approximations for programs on the Pareto frontier prefer the simple choice of $\rho = 0$ when aggregated across all 21 shaders. Assuming equal weight for each shader, we find 87% of program variants prefer $\rho = 0$, whereas only 4% use ρ as a constant, and 6% use ρ estimated based on the affine assumption. We conclude that for shader programs, the simple choice of $\rho = 0$ in most cases suffices.

Note that our brick shader (shown in Figure 5.1) gives poor results for the method of Dorn et al. [35], while in their paper, a brick shader with a similar appearance shows good results. This is because their brick shader is implemented such that the tiling can be

bandlimited independently, while our implementation intentionally uses the more challenging function fract() to exercise our compiler.

5.7.2 Evaluation for Curved Geometry

This section evaluates shaders on two curved geometries: sphere and hyperboloid. Variables such as surface normal have more complicated distributions on curved geometries, while in planar geometry (Section 5.7.1), they are just constants. Because of this, shaders are tuned separately on each of the geometries.

Results for 7 of the shaders are presented in Figure 5.7, including one result for each base shader. The program variant shown in the result is chosen similarly as in Section 5.7.1.

5.7.3 Short Tuning

Figure 5.8 shows 5 results for short tuning where each shader is only tuned only for a limited time. Specifically, we report the first available result at the end of a generation after the tuner has run for 10 minutes. We compare the results of short tuning with full tuning: the tuner is run by default for 20 generations. Similarly, our short tuning result shown in Figure 5.8 is chosen with sufficiently low error.

5.8 Summary and Discussion

This chapter proposes a general compiler framework that smoothes an arbitrary program over the floats by approximating its convolution with a Gaussian kernel. We present several different approximations with different accuracy and efficiency trade-offs. We apply the framework to automatically bandlimit procedural shader programs and demonstrate our framework has substantially better error than Dorn et al. [35] even after our improvements, and is frequently an order of magnitude faster than supersampling. While this chapter focuses on the application of procedural shaders, the proposed smoothing framework is indeed gen-

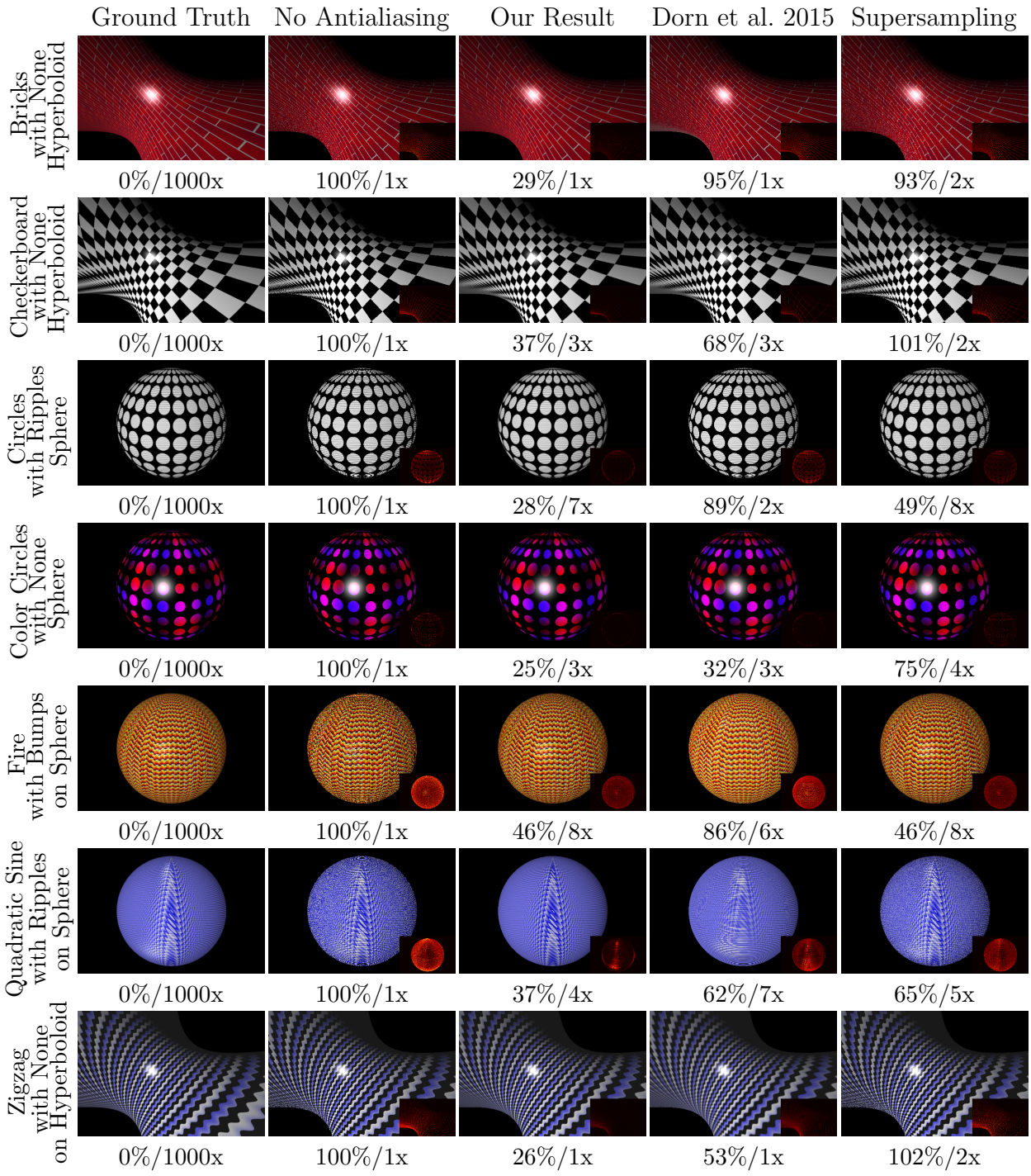


Figure 5.7: Selected result images for 7 shaders on curved geometries. Reported below each shader are L^2 error and run-time relative to no antialiasing as in Figure 5.1. Please zoom in to see details.

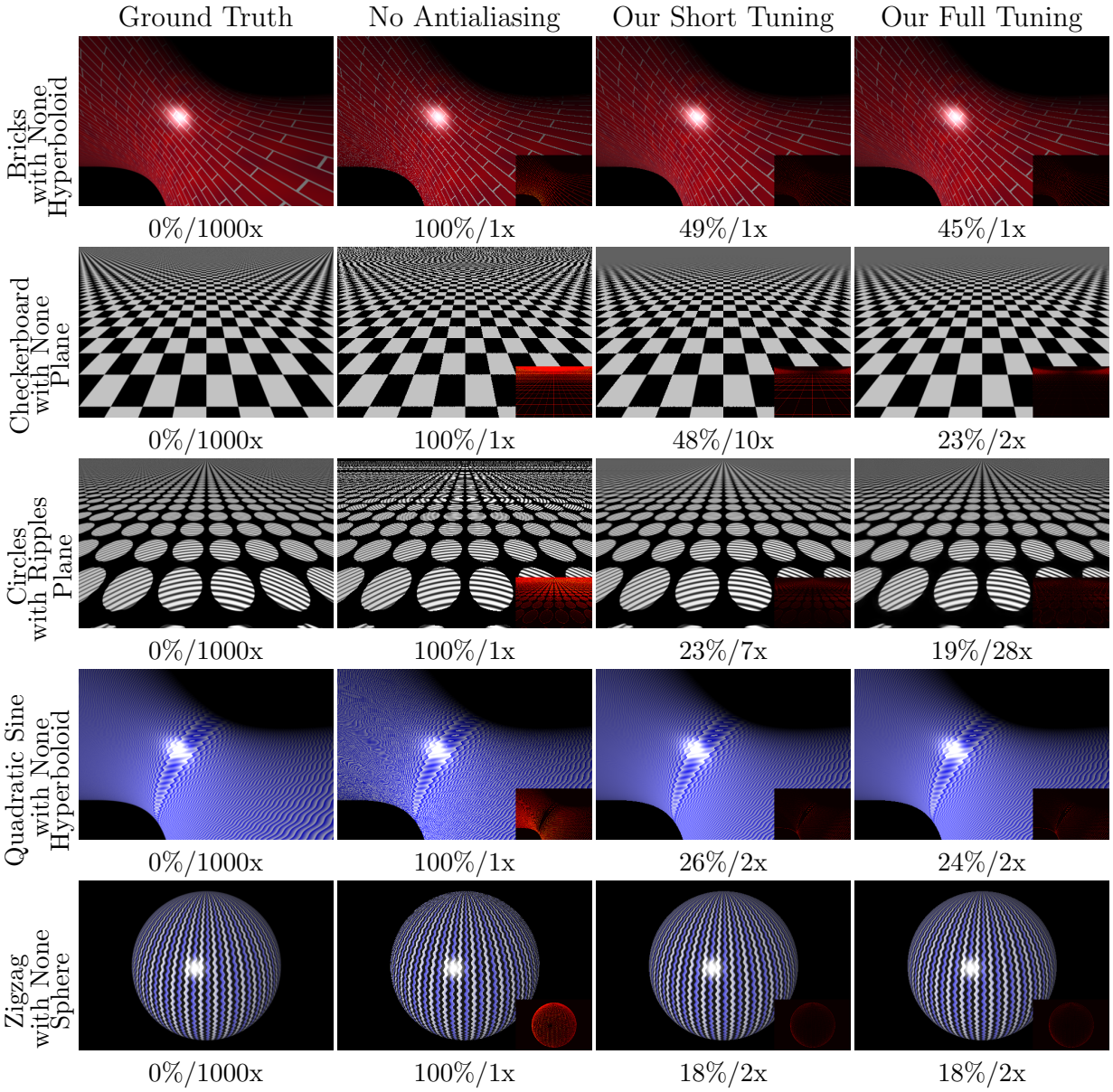


Figure 5.8: Results for short tuning. Shaders are tuned for about 10 minutes ("Our Short Tuning") and compared to tuning for 20 generations ("Our Full Tuning"). Note that many aliasing patterns can be reduced after short tuning. Please zoom in to see details.

eral, and we believe it could be useful for other problems in graphics, mathematics, and other disciplines. Our source code is available at `https://github.com/yyuting/approximate_program_smoothing`.

The proposed framework still has several limitations, which invite future exploration. First, our smoothing result may contain small amounts of residual aliasing or biases when the Gaussian distribution assumption is violated. Future work may extend to approximation rules with more general assumptions on the variable distribution. Second, for curved geometries, highly aliased regions are rarely sampled because they usually occupy small pixel regions in the rendering. This may cause the search algorithm biased toward areas with less aliasing. This could be addressed by adaptive sampling strategies that favor more challenging regions when preparing the ground truth. Finally, the exponential search space in our genetic search limits our capability of generalizing to production shaders. Static approximation choices are needed to scale to significantly more complicated shaders.

Chapter 6

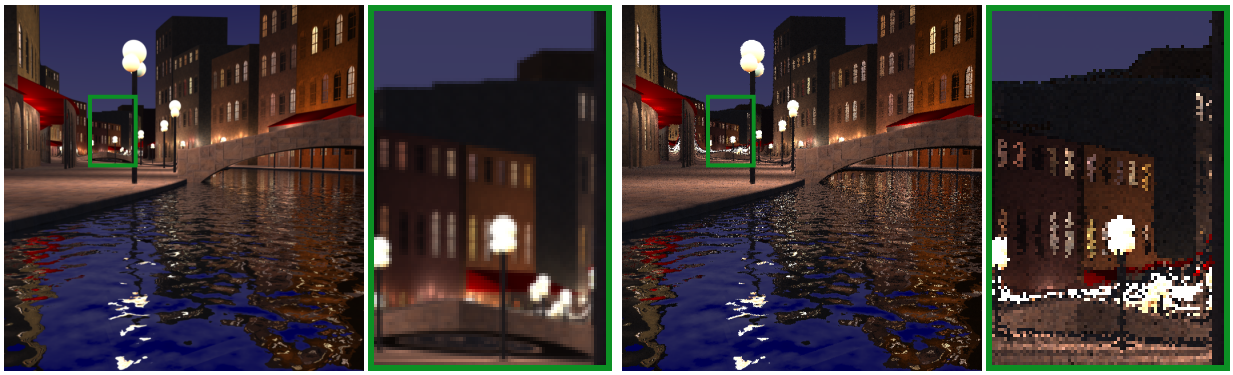
Learning from Program Traces

The frameworks described in Chapters 2 and 5 both utilize a set of compiler rules to deterministically mutate an input program to generate an output program that fulfills some tasks, such as differentiating a discontinuous program, or smoothing a program. On the other hand, deep learning methods are popular alternatives especially when the task cannot be expressed programmatically, or when the program representation is expensive. Because the network architecture is fixed during training, and only the model parameters are updated, the deep learning approach are also referred to as "black box" proxies in contrast to "white box" program representations.

Nevertheless, the network model itself being a "black box" does not mean the data generation process is ignored entirely. In fact, a good deep-learning solution usually benefits from observations of the underlying process. For example, understanding whether the task is a global or local transformation helps design the network architecture, such as deciding its receptive field and whether a skip connection is needed. Similarly, while deep learning in graphics and vision typically uses color images as network inputs, they can also be augmented by features such as depth or surface normals if the input image is generated from a traditional rendering pipeline. These auxiliary features are typically manually picked based on domain expertise and cannot be easily generalized to arbitrary applications or programs. However, a

(a) Reference

(b) Simplified Input



`Venice`: 1x speedup, 0% error

1400x speedup, 7.4x error

(c) RGBx Baseline

(d) Our Result

1300x speedup, 1x error (baseline)

1300x speedup, 70% error

Figure 6.1: Learning to extrapolate from the partial computation of a procedural shader called `Venice`. The reference solution (a) results from a full computation at 1000 samples per pixel (SPP). A simplified version of the shader provides an approximate solution (b) using only 1 SPP and less computation per sample (72% of the original compute; 1400x speedup overall). Our RGBx baseline method (c) learns to approximate the reference well for much of the image, based on only the RGB output of the simplified shader as well as a few hand-picked auxiliary features – but exhibits artifacts in the distance (obvious in the zooms boxed in green). This paper shows that difficult learning tasks like this can benefit from relying on not just the RGBx features but also the *program trace* (a record of the intermediate values computed at every pixel, in this case, that of the simplified shader) – producing a more faithful approximation of the reference. Percent (%) denotes the mean perceptual error [158] relative to that of the RGBx baseline, averaged over the test set.

key property shared by these auxiliary features is that their input image is typically generated by a program, and the program trace is usually a superset of the chosen auxiliary features. Therefore, this chapter instead explores using the entire program trace to augment the deep model's input features.

Specifically, we collect the intermediate values computed at program execution, and these data form the input to the learned model. We investigate this learning task for a variety of applications: our model can learn to predict a low-noise output image from shader programs that exhibit sampling noise; this model can also learn from a simplified shader program that approximates the reference solution with less computation, as well as learn the output of postprocessing filters like defocus blur and edge-aware sharpening. Finally, we show that the idea of learning from program traces can even be applied to non-imagery simulations of flocks of boids. Our experiments on a variety of shaders show quantitatively and qualitatively that models learned from program traces outperform baseline models learned from RGB color augmented with hand-picked shader-specific features like normals, depth, and diffuse and specular color. We also conduct a series of analyses that show certain features are important within the trace: these coincide with intuitively important aspects of the program. The important features can help select a good subset of trace features for learning, and even learning from a small subset of the trace already outperforms the baselines. We finally show that multiple shaders can be learned together with a shared denoising network and a lightweight shader-specific encoder.

6.1 Overview

Deep learning applications in graphics and vision typically work on images encoded as pixels in RGB color space. For images with 3D scenes, researchers have also explored augmenting the RGB data with hand-picked features like depth or surface normals [21, 132]. These

auxiliary features are picked based on domain expertise, and vary for different applications or programs.

This chapter instead proposes augmenting the data from which a neural network learns with the *program trace*. In software engineering, a trace refers to the record of all states that a program visits during its execution [42, 66], including all instructions and data. We explore this idea in the context of procedural shader programs, like the one shown in Figure 6.1. The sequence of instructions tends to be similar from pixel to pixel, so we rely on just the intermediate values for learning, referring to these as the "program trace."

Shader programs can be used to flexibly construct complex and even fantastical appearances by combining sequences of mathematical operations to create texture patterns, produce lighting, perturb surface normals to produce effects such as bump mapping, apply noise functions, or determine ray intersections with procedurally generated geometry [4]. A range of example shaders may be seen throughout this paper; many more examples are available from websites such as `shadertoy.com`. Note that while the example shaders appearing here are simpler than those typical of production or games, they embody the key features that appear in production-level shaders.

Since the fragment shader program operates independently per pixel, we can consider the full program trace as a vector of values computed at each pixel – a generalization from simple RGB. Since there are many pixels (program traces) per image, and potentially many computed images, this provides a rich source of data from which to learn. Graphics and vision researchers have explored learning algorithms for input-output image pairs with a few auxiliary feature buffers, such as those of Vogels et al. [132] on removing sampling noise, and Xie et al. [146] on fluid super-resolution. Such features are *manually* identified by an expert on a per-shader basis. Moreover, the extent to which these auxiliary features help learning depends on the choice of features, the particular shader, and the learning goal. We believe other shader-specific information useful to the learner remains hidden within the program execution, and that a learning process could *automatically* identify and leverage that

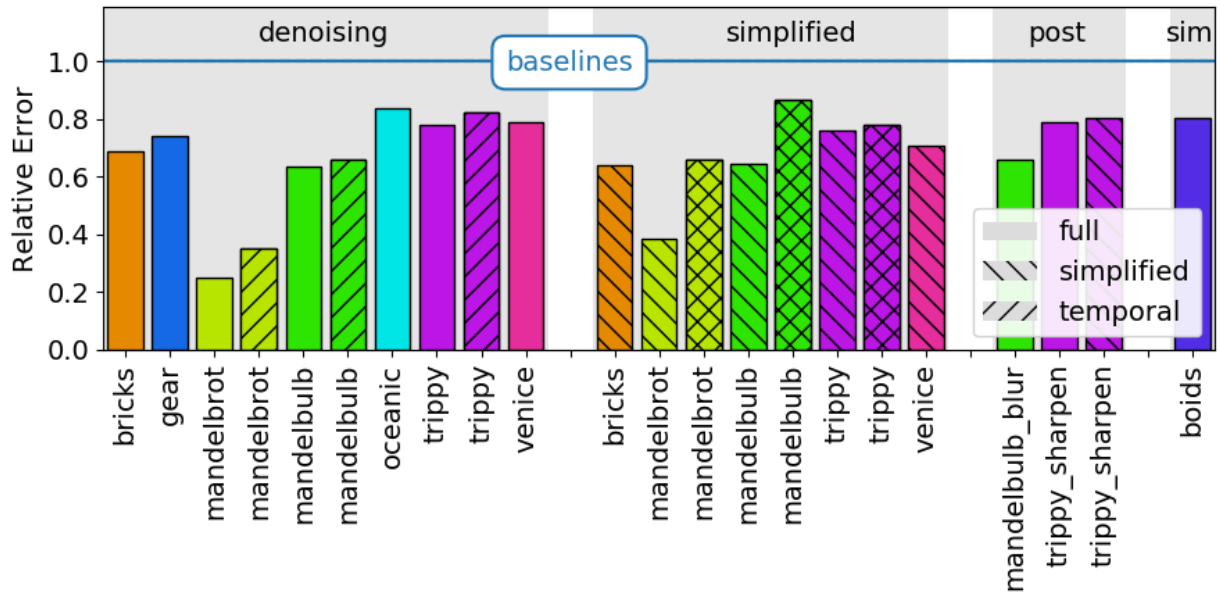


Figure 6.2: Four-application summary: **denoising**, reconstruction from **simplified** shaders, learned **post**-processing effects, and **sim**ulation. The vertical axis shows (in every example) improved perceptual error compared to each application's strongest baseline: RGBx for denoising, simplified, and post; and I/O for simulation (Section 6.5). Hatching directions denote the use of simplified shaders and/or temporally coherent models.

information. Thus, we propose a learning-based approach that utilizes all of the information produced during the execution of a shader program. The learner could automatically identify which features are useful, obviating the need for manual feature selection amid an experimental process.

Intuitively, learning tasks that extrapolate from a partial computation to predict the result of a full computation may benefit from learning from program traces. To illustrate its applicability, we introduce four applications. Three of them work from pixel data: learning to predict low-noise output, learning to reconstruct full computation from a program with partial computation, and learning the output of a postprocessing filter. The fourth application shows that the idea of learning from program traces can be applied to non-imagery data: it learns to simulate the position and velocity of a flock of "boids" [113], which emulate flocking behavior similar to that of birds or fish. The performance of these applications is summarized in Figure 6.2. In most of our experiments, we train a separate model for

each shader on each application. Scene-specific learning is commonly used in recent work on novel view synthesis [122, 128, 129, 82]. Section 6.6.3 describes how a single network can be trained over *multiple* shaders.

The primary contribution of this chapter is the idea that a shader program trace can be used as a feature vector for machine learning. Nevertheless, it is neither obvious how to use such a feature, nor that it would help in any particular application. Thus, a secondary contribution is to introduce a framework for learning from program traces, and demonstrate that it outperforms baseline methods in several applications. The third contribution is to investigate the relative importance of individual trace features, and how the input trace size across various trace subsampling strategies can affect the performance of the model. Our code is available at: `https://github.com/yyuting/learning_from_program_trace`.

6.2 Related Work

Program traces in machine learning. Program traces have proven helpful in malware detection [25], program induction [112], and program synthesis [28, 58]. Researchers have also explored using partial execution or partial rendering to synthesize graphics programs [45] or infer parameters for procedural models [114]. Instead of developing specialized learning models for a particular application, we explore a generic architecture that can learn over a range of applications. Nevertheless, this work focuses on learning from program traces for shaders, which enjoy certain unique properties such as an emphasis on pixel outputs and an enormous degree of parallelism. We also investigate network models and training schemes, and analyze the importance of individual trace features, as well as the effectiveness of several trace subsampling strategies.

Features for deep learning on imagery data. Researchers have explored a variety of features beyond simple RGB as inputs to learned functions. Nalbach et al. [92] have made a comprehensive exploration of such features as part of a deferred shading pipeline.

Xie et al. [146] consider auxiliary features including flow velocity and vorticity when learning density super-resolution for fluid simulation. Positional encoding [82] augments the input by applying high-frequency functions to coordinate features, but does not benefit our tasks (Section 6.5.7). To our knowledge, our paper is the first to propose augmenting such features with the full program trace. The benefits are that the trace of a program that computes such manually picked features inherently includes them, as well as other potentially useful information; moreover, the extent to which various features are useful for a particular application and shader are discovered automatically by the learning process.

Feature space reduction. In deep networks, an overly large feature space can exhaust memory, increase training time, or even make learning tasks harder. Researchers have explored methods to reduce the feature space by pruning whole convolutional filters [68, 77, 88]. In our method, we focus mostly on reducing the input feature space because the dimension of the program trace can be large. We use a method similar to that of Molchanov et al. [87] to evaluate the importance of each trace input and show a trade-off between the runtime and visual fidelity as we change our feature reduction strategies. Our experimental results suggest that without prior execution or learning, we could find no subsampling strategy that consistently outperforms a simple uniform subsampling. However, if the shader is allowed the overhead of learning a model over the full program trace, we can select important trace features from the learned model.

Remove sampling noise. One of our applications addresses Monte Carlo noise reduction in low-budget rendering. One strategy for low-noise rendering involved carefully distributing the samples, see Zwicker et al. [160] for a survey. Another method uses symbolic compilation techniques to analytically approximate the integral that produces the smoothed shader [35, 148], but is hard to scale to complicated shaders such as the one shown in Figure 6.1. On the other hand, learning-based algorithms train regressors such as neural networks to predict the rendering. The input to networks is usually augmented with auxiliary features [21, 132,

48]. Unlike previous work, our approach gathers customized information per shader, and is orthogonal to learning-based denoising network design in the sense that it can be combined with an existing network.

Shader simplification. As the complexity of the shader program grows, it is common to apply lossy optimization to obtain programs that only approximate the original program but with better runtime performance [54, 121, 133]. We show experiments that explore how the trace from the simplified programs can provide information that helps to recover the missing details in the target shader. Thies et al. [128] identify a similar task where they learn novel view synthesis from a coarse proxy geometry. (The `Venice` example shown in Figure 6.1 is particularly reminiscent of that work.) Nevertheless, to our knowledge this is the first paper to propose the application of learning from a simplified shader program to restore details in the original program; and we show that using the program trace can help in this application.

Neural networks for image processing. Researchers have investigated a variety of learning-based methods for image processing tasks, such as image enhancement and filtering [47, 70, 72, 144]. Our postprocessing application demonstrates that the proposed method is also helpful when learning these imagery operations as postprocessing filters.

Learning simulation programs. High-quality simulation usually executes the program over many tiny time steps, which is expensive. Researchers have developed reinforcement learning-based methods [52, 62] to replace the program entirely, or execute the program at a lower spatial resolution and learn a super-resolution model [136]. Our approach instead learns from the program's execution trace on a larger time step, and corrects the output as if the program is executed for multiple smaller steps.

6.3 Compiler and Preprocessing

This section introduces a compiler that can collect traces from shader programs. It translates shader programs from a domain specific language to TensorFlow code that logs the trace (Section 6.3.1). To stay within the hardware memory budget, the compiler also restricts the trace length to an arbitrary size cap (Section 6.3.2). Collected program traces are further preprocessed before learning (Section 6.3.3). Section 6.4 describes the learning process in detail.

6.3.1 Compiler and Program Traces

Our compiler takes as input an arbitrary procedural shader program written in a domain specific language (DSL) and translates it to a TensorFlow (TF) program that outputs a rendered image as well as a collected program trace. We embed the DSL in Python, which allows us to use Pythonic features such as operator overloading. We also include common shader operations such as trigonometric functions, dot, and cross products. For simplicity, we assume the shader program manipulates numerical scalars or vectors of known size. We handle branching by computing both branches of conditionals. Likewise, loops are unrolled to the maximum possible number of iterations: this limit is set by the programmer for each loop. These are not fundamental limitations of the approach, as we experimented with emulating branching and variable-length loops by writing dummy values of zero to traces in the branch/iteration not executed, and this gives visually and quantitatively identical results to our current approaches (Section 6.5.6). These policies permit us to express the trace of any shader as a fixed-length vector of the computed scalar values, regardless of the pixel location.

6.3.2 Feature Vector Reduction

Large program traces can produce unnecessarily large feature vectors from which learning becomes unwieldy, or worse, exhausts memory. Loop unrolling is a common contributor to large traces, because the program trace would be scaled by the number of iterations. The remainder of this section describes several strategies for reducing the size of the feature vector. All these strategies described in this section can be reused when targeting a different language, e.g. OpenGL Shading Language (GLSL) or CUDA.

Compiler optimizations. Since such features would be redundant in the learning network, the compiler omits constant values, duplicate nodes in the compute graph, and neighboring nodes that differ only by a constant addition or multiplication. The compiler also identifies common built-in functions and iterative improvement loops to eliminate highly correlated trace features.

Built-in functions (e.g., `sin`) should typically be treated as a black box. Our DSL provides widely used shader operations such as noise functions and a normal computation functor. The compiler logs only the return values of such built-in functions, not the intermediate values found when computing them. This is a natural choice since in principle one could trace down to a very low level such as including details about the microarchitecture, but we believe that learning will gain the most benefit if it occurs at a similar abstraction level as used by the programmer.

An iterative improvement loop repeatedly improves an approximate result to obtain a more accurate result [119]. A commonly used iterative improvement pattern in shader prototyping is a ray marching loop that computes the distance from the camera to objects in the scene. Because each iteration computes a more accurate approximation than the previous iterations, the final iteration is the most informative. Therefore, the compiler will only log the trace from the final iteration of such loops. We automatically handle common cases of iterative improvement loops found in shaders by classifying loops based on pattern

matching: the output of the loop is either iterative additive or can be written as a parametric form of the iterative additive variable, formally defined in Definition 19. For a loop variable X at iteration n, we will denote its value as X_n

Definition 16 *A loop variable X is iterative additive if it matches the following pattern or its equivalent forms:*

$$X_n = X_{n-1} + Z \tag{6.1}$$

Here Z can be any arbitrary variable.

Definition 17 *A variable Y is dependent on an iterative additive variable X if it matches the following pattern or its equivalent forms:*

$$Y_n = select(cond, Y_{n-1}, f(X_n, X_{n-1}, C)) \tag{6.2}$$

Here, cond is an arbitrary Boolean variable, f is an arbitrary function, and C is a variable computed outside the loop, i.e. C can be viewed as constant inside the loop.

Definition 18 *A loop variable X is an output variable if for any iteration n, its value X_n is used outside the loop.*

Definition 19 *A loop is classified as an iterative improvement loop if all of its output variables are either iterative additive or are dependent on an iterative additive variable.*

We also investigate several other strategies inspired by previous work on loop perforation [119] and image perforation [75]. In our case, however, we always run the full computation, but simply select a subset of those computations as input to the learning task, as follows.

Uniform feature subsampling. The most straightforward strategy is to subsample the vector by some factor n, retaining only every n^{th} trace feature as ordered in a depth-first traversal of the compute graph. This approach tends to work well in our experiments, and we speculate that it does so because nearby nodes in the compute graph tend to be related by simple computations and thus are redundant.

Other sampling schemes. We explored a variety of other schemes to reduce the feature vector length, including "clustering" based on statistical correlation, "loop subsampling" that logs features from every kth loop execution; "first or last" which only collects features from either the first or last iteration of a loop; and "mean and variance" summarize the statistics of a variable over all loop iterations. Yet none outperformed the above straightforward scheme consistently enough to justify their use in our subsequent experiments.

These options are combined as follows. We first apply compiler optimizations, then subsample the features with a subsampling rate that makes the trace length most similar to a fixed target length. For all experiments, we target a length of 200, except where specifically noted such as in the simulation example. After compiling and executing the shader, we have for every pixel: a vector of dimension $\mathcal{N}$: the number of recorded intermediate values in the trace.

6.3.3 Whitening the Collected Trace

We preprocess the traces to rescale the data to a fixed range. Intermediate values in computed shader programs can vary over a large range: resulting in values such as 10^{30}, $\pm\infty$, or *not a number* (NaN), even when most values of this shader computation remain near zero. This can happen, for instance, near object silhouettes where textures have a high frequency in image space. The extreme values could cause a standard whitening technique to fail entirely, due to say undefined mean or standard deviation where values such as $\pm\infty$ or NaN are present. Even if only finite trace values are observed at training time, standard whitening may focus

too much on extreme values such as 10^{30}, resulting in meaningful data (e.g. between $[-1, 1]$) being mapped to a very small range, and at test time, extreme values such as $\pm\infty$ or NaN can still produce non-finite floating point values that are problematic for inference.

We thus develop a whitening method for shader program traces. We first clamp extreme values by collecting the statistics for the intermediate values' distribution at training time. For each intermediate value, we first decide whether its distribution merits clamping. If we detect that the distribution has only a small number of finite, discrete values (10 or fewer), we do not apply clamping to the corresponding intermediate value. For the rest of the intermediate values, we first discard infinite values and then find from their distributions the lowest and the highest pth percentiles, denoted P_0 and P_1, and use these to compute clamping thresholds. Next, we clamp all values to the range $[P_0 - \gamma(P_1 - P_0), \quad P_1 + \gamma(P_1 - P_0)]$. We also set NaN values to the low end of this range. Empirically, we found in our experiments that $p = 5$ and $\gamma = 2$ work well, and we use these values for all results. Finally, for each intermediate feature, we rescale the clamped values to the fixed range [-1,1], and record the corresponding scale and bias used. In both training and testing, the collected program traces are used directly by applying the same precomputed scale and bias, but the values will be clamped to the range [-2, 2] to allow data extrapolation.

We evaluated the effectiveness of scaling and clamping on the denoising task (Section 6.5.1) with `Mandelbrot`. If trained without clamping, the model will diverge to NaN even before the first iteration finishes, while training without whitening results in 12x worse perceptual error compared to our full method. These results indicate that our data preprocessing is essential in our pipeline.

6.4 Network Architecture and Training Details

This section briefly summarizes the training details in our experiments. For all of our imagery applications, we selected a basic architecture described in Section 6.4.1. Nevertheless, our

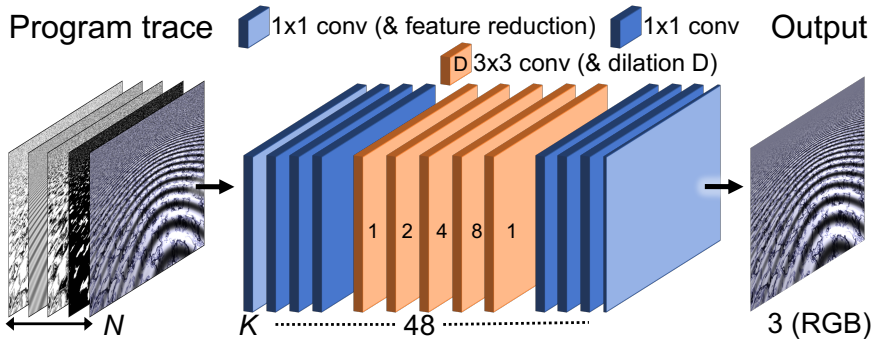


Figure 6.3: Network architecture used in our experiments. The input from the program trace has $\mathcal{N}$ channels. The output layer has three channels for color images. The first feature reduction layer has $\mathcal{K}$ channels. We use $\mathcal{K} = 48$ in our method. When training the baseline method, $\mathcal{K}$ will be increased to a larger value to match the total number of trainable weights to be the same as training with the program trace at the maximum length. All other intermediate layers have 48 channels. The input feature maps are first analyzed by four 1x1 convolutional layers, followed by five 3x3 convolutional layers with dilation rates of $1, 2, 4, 8, 1$ respectively. Finally, four additional 1x1 convolutional layers are applied and output a three-channel image. Note that the first and last convolutional blocks indicated in lighter blue each reduce the number of channels (from $\mathcal{N}$ to 48, and from 48 to 3, respectively).

method could be coupled with any deep learning architecture. This section thus serves as an example of how to select a network architecture and carry out training.

6.4.1 Network Architecture

Our experiments use a dilated convolutional neural network depicted in Figure 6.3, similar to that of Chen et al. [26]. This network architecture is used directly in our denoising (Section 6.5.1) and post-processing (Section 6.5.3) applications, and also serves as a generator model in other applications and scenarios, each of which relies on a GAN model: conditional spatial GAN [57] for learning from a simplified shader (Section 6.5.2) and temporal GAN [134] for learning temporally coherent sequences (Section 6.5.4). We prefer to keep a single network architecture for consistency and ease of experimentation across all applications except boids, to demonstrate our core idea of learning from shader program traces is beneficial across many applications, although more specialized architectures could be beneficial for certain applications like denoising (e.g. [48]). Details about the GAN models are discussed

in Section 6.4.3. The boids simulation relies on a fully-connected architecture described in Section 6.5.5.

6.4.2 Loss Functions

We use a combination of pixel-wise color loss L_c and perceptual similarity loss L_p to encourage network output to be similar to the ground truth during training: $L_b = L_c + \alpha L_p$. The parameter α is a weight that balances between the color and perceptual loss terms. We fix $\alpha = 0.04$ for all of our experiments. This value was chosen to roughly balance the magnitude of the gradients due to L_c and L_p during back-propagation. The color term L_c is simply the standard L_2 loss on the RGB image. The other loss term L_p uses the learned image perceptual dissimilarity metric of Zhang et al. [158]. This section describes the basic loss L_b used in training. Additional details about the GAN losses can be found in Section 6.4.3, and the loss used in the boids simulation is described in Section 6.5.5.

6.4.3 Details for GAN Models

Our spatial GAN model is a conditional GAN, where the conditional labels are the RGB channels of the 1 SPP rendering from the shader program, denoted as c_x. Because c_x is already part of the program trace, we directly use the model from Figure 6.3 as our generator, and the generator's output is naturally conditioned on c_x. We then train the model to match the ground truth denoted as c_y. Additionally, we used a patchGAN architecture similar to that of Isola et al. [57] with receptive field 34×34 as our discriminator D.

Our temporal GAN model uses a similar architecture as the spatial GAN with modifications following [134]. The generator is conditioned on imagery from three consecutive frames: the current predicted frame and the two previous ones. This involves five 3-channel images as conditional labels: shader RGB output from all three frames plus the generator's output from the two previous frames. Because neither the shader output nor the generator output from the previous two frames is part of the program trace for the current frame, we modified

the generator architecture in Figure 6.3 to concatenate the additional four conditional label images *after* the feature reduction layer. The rest of the architecture remains unchanged. We use the same discriminator architecture as for our spatial GAN, but it takes an input of sequences of frames and their corresponding conditional labels.

We now introduce the variation on the basic loss function that incorporates the GAN loss. First, we add GAN loss for some applications (the simplified shaders of Section 6.5.2 and the temporally coherent sequences of Section 6.5.4). We use a modified cross-entropy loss [49] for both spatial and temporal GAN models. Our spatial GAN model is conditioned on the RGB channels of the shader program c_x to approximate the distribution of the ground truth c_y, while our temporal GAN loss is applied to sequences $\widetilde{c_x}$ and $\widetilde{c_y}$. The training objective (that we minimize) for generator L_G and loss for spatial discriminator L_{D_S} can be expressed as:

$$\begin{aligned} L_G = &\quad L_b - \beta\, \mathbb{E}_{c_x} \log(D_S(G(c_x), c_x)) \\ L_{D_S} = &- \mathbb{E}_{c_x, c_y} \log(D_S(c_y, c_x)) \\ &- \mathbb{E}_{c_x} \log(1 - D_S(G(c_x), c_x)) \end{aligned} \tag{6.3}$$

Similarly, the training objective on temporal sequences for generator L_G and temporal discriminator L_{D_T} can be expressed as:

$$\begin{aligned} L_G = &\quad L_b - \beta\, \mathbb{E}_{\widetilde{c_x}} \log(D_T(G(\widetilde{c_x}), \widetilde{c_x})) \\ L_{D_T} = &- \mathbb{E}_{\widetilde{c_x}, \widetilde{c_y}} \log(D_T(\widetilde{c_y}, \widetilde{c_x})) \\ &- \mathbb{E}_{\widetilde{c_x}} \log(1 - D_T(G(\widetilde{c_x}), \widetilde{c_x})) \end{aligned} \tag{6.4}$$

The parameter β is a weight that balances between the GAN loss and the regular color and perceptual loss. In all our experiments with GAN loss, we fix $\beta = 0.05$ to roughly balance

the magnitude of gradients from all loss terms. Note in Equation 6.4 we did not include spatial discriminators for simplicity. But it is possible to combine both Equation 6.3 and Equation 6.4. For example, in Section 6.5.4, we trained both discriminators to produce a temporally coherent model for simplified shaders.

We also skip the back-propagation on the GAN loss for any mini-batch with constant color to avoid training instability.

6.4.4 Generating the Dataset

Our experiments generate the dataset from 800 images for training, 80 images for validation, and 30 images for testing (each 960×640). Although this training set size is small relative to typical deep-learning tasks, we address this concern in Section 6.4.5. The training images are generated with random camera poses, while testing images are divided into two groups: 20 *similar* distance images with camera pose sampled from the same distribution as the training set, as well as 10 *different* distance images that are closer or farther than the training set. For some shaders, (`Trippy Heart`, `Mandelbrot`, `Mandel-bulb`, `Venice` and `Oceanic`), a periodic time parameter also changes the shader appearance, which is sampled from the same distribution for both training and testing datasets.

We find it beneficial to further divide the training and validation set into tiles. One advantage is that certain features in the shader may be visually salient to humans, so we can emphasize such features to ensure they are learned well. In principle, this could be accomplished with automatic saliency models (e.g. [64, 19, 30]). However, off-the-shelf saliency models are trained for natural imagery whereas our shaders are non-photorealistic, and therefore we combine both a saliency model [30] and a traditional Laplacian pyramid representation to robustly and automatically select salient tiles. Another benefit of tiled training is that it reduces memory, and it also accelerates convergence, because we can use larger mini-batches with more varied content within the same GPU memory to obtain a gradient estimator with a lower mean squared error.

We sample training and validation tiles as follows. We first generate saliency maps for each of our 800 training images and 80 validation images using Cornia et al. [30]. Saliency models usually incorporate a center bias that tends to give lower saliency scores to pixels closer to image boundaries. This behavior is not ideal for our framework because our training images are generated from randomly sampled camera poses so that salient content could appear anywhere in the image. Therefore, we run the saliency model on images with an extended field of view (each 1280×960) where the center patches of size 960×640 are our original training images. This allows every pixel in the original training dataset to be away from image boundaries to avoid center bias in the resulting saliency maps.

We then subdivide each of the training and validation images into six 320×320 tiles. For each tile, we estimate its intensity on low, middle, and high frequencies by taking the average over its first, third, and fifth level of the Laplacian pyramid [18]. Together with the average saliency score, these four metrics can be combined to robustly sample salient and interesting tiles for learning.

Next, we use identical sampling rules to sample one-quarter of the sampling budget from each of the four metrics. For each metric, we rank the tiles according to their associated score and only sample from the tiles whose score is within the top 25% nonzero scores. The score of the qualified tiles will further be normalized to [0, 1], and each tile will be sampled with a probability proportional to the normalized score.

Apart from the rules described above, we find it helpful to also include a small portion of constant color tiles in the training dataset, e.g. the black background in `Bricks` Figure 6.4. These uninformative and constant color tiles can be easily selected from a low color variance threshold. Although some salient tiles already contain both informative and uninformative regions, they are usually close to object silhouettes and could still pose challenges when extrapolating to uninformative regions far away from the object. This is because the trace can vary greatly at different noninformative regions (e.g. if the pixel is not hitting any object, its negative distance to objects can still vary by a lot).

We sample a total of 1200 tiles for training and 120 tiles for validation. If the shader does not contain constant color tiles, all of the sampling budgets will be used to equally sample from the 4 saliency metrics described above. Otherwise, only 95% of the sampling budget will be sampled from saliency, and another 5% will be sampled from low color variance tiles. Testing still relies on 30 full images.

6.4.5 On the Fly Training

In training, we generate input program traces on the fly each time one is needed, rather than loading pre-computed traces from the disk. There are two benefits to this approach. First, precomputed traces are large, and it is typically faster to re-compute the trace, as opposed to loading it from a disk. Second, each time a trace is generated, we use a new randomly sampled sub-pixel location for evaluating the trace for any given pixel (a common strategy to reduce aliasing). Therefore, the input traces will generally have different values in each epoch even though we use the same ground truth solution. This approach helps the network avoid overfitting.

6.5 Evaluation

This section describes experiments evaluating our method for various applications and scenarios: denoising pixel shaders (Section 6.5.1), learning to reconstruct simplified shaders (Section 6.5.2), learning postprocessing effects (Section 6.5.3), and learning non-imagery simulation programs (Section 6.5.5). We also discuss learning temporal coherence in Section 6.5.4. The architecture and training scheme in these applications include fully connected networks, traditional CNNs, and GANs, demonstrating our method's wide applicability to various deep learning models. We report LPIPS, SSIM, and PSNR for all applications in Table 6.1, with a performance summary shown in Figure 6.2: in all cases, our method outperforms the strongest baseline.

For image processing applications, we choose a variety of challenging combinations of shaders and geometries. The `Bricks` shader relies on simplex noise [106]. The `Trippy Heart`, `Mandelbrot`, and `Mandel-bulb` shaders rely on iterative fractals. The shaders `Mandel-bulb`, `Gear`, `Oceanic`, and `Venice` construct complex 3D procedural geometry rendered by ray marching over a signed distance field [142]. The shaders `Bricks` and `Venice` extract contents from texture maps. We adapted shaders `Oceanic`, `Trippy Heart`, `Mandel-bulb`, and `Venice` from shaders with the same names at the website `shadertoy.com`, by the authors *Frankenburgh*, *Cras*, *EvilRyu*, and *reinder*, respectively, while `Gear` is adapted from the shader "primitives" by author *Iq*; and the boids and fluid simulations described in Section 6.5.5 were adapted from "Simple Boids" by *Saduras* and "Chimera's Breath" by *nimitz*.

Our implementation is trained on a single GPU. For consistent timing in evaluation, we use a 4-core Intel Xeon E5-2620 v4 2.10 GHz CPU with a single Nvidia GForce RTX 2080 Ti GPU across all models. During training, we always train 400 epochs for models without a GAN and 800 epochs for models with a GAN. Timing results reported throughout appear as speedup relative to ground truth. The actual shader runtime ranges from 30ms to 21s with a median of 1.4s for full computation i.e. non-simplified shaders (Section 6.5.1 and 6.5.3), and from 20ms to 6s with a median of 80ms for partial computation (Section 6.5.2). Inference time ranges from 70ms to 0.2s with a median of 90ms. These shaders are relatively slow because they are implemented as computational graphs in TensorFlow. They could be greatly accelerated through engineering a GLSL or CUDA implementation. Note the shader's runtime is invariant to whether program traces are collected or not, therefore it is not a limitation to our proposed method. In all cases, we select the model at the epoch with the lowest validation loss. For imagery learning tasks (Section 6.5.1, 6.5.2, 6.5.3), the model trains on a dataset of 1200 tiles with 320×320 resolution, and 120 validation tiles in the same resolution. Testing includes 30 full-size images with a resolution of 640×960. Please refer to Section 6.4.4 for further details regarding our training. All experiments presented in this

section are trained per shader. We also demonstrate in Section 6.6.3 that multiple shaders can be trained together with a shared network and a lightweight shader-specific encoder.

Our strongest baseline is RGBx. It uses the same network and training as ours, but with the input features consisting of RGB color plus manually picked auxiliary features that are commonly used for learning with shader programs. We use normal, depth, diffuse, and specular color whenever these terms are explicitly represented in the program. These correspond to auxiliary features used in recent denoising papers [21, 132]. Because the RGBx baseline generally has fewer input channels compared to our method, we increase the number of channels in the first convolutional layer of the baseline model such that the number of trainable weights matches that of our model. Unlike our *automatic* method, RGBx requires additional *manual* expertise to pick auxiliary features for every shader program. An automatic baseline that resembles ours would be RGB, which uses only RGB color without any auxiliary features. However, RGBx always outperforms RGB, so we only compare with RGBx.

6.5.1 Denoising Fragment Shaders

Here we describe the application of removing sampling noise. Our goal is to approximate a low noise reference image collected using 1000 samples per pixel (SPP). Our method is evaluated using 1 SPP, drawn from a Gaussian spatial distribution with a standard deviation of 0.3.

We evaluate our method and compare it against two baselines. The first baseline is RGBx described before. Our second baseline is supersampling. Supersampling draws several samples at each pixel, evaluates the shader to obtain RGB colors for each sample, and takes the mean of the colors. We supersample by choosing a constant sample budget per pixel to achieve approximately the same run time as ours, including the overhead for neural network inference.

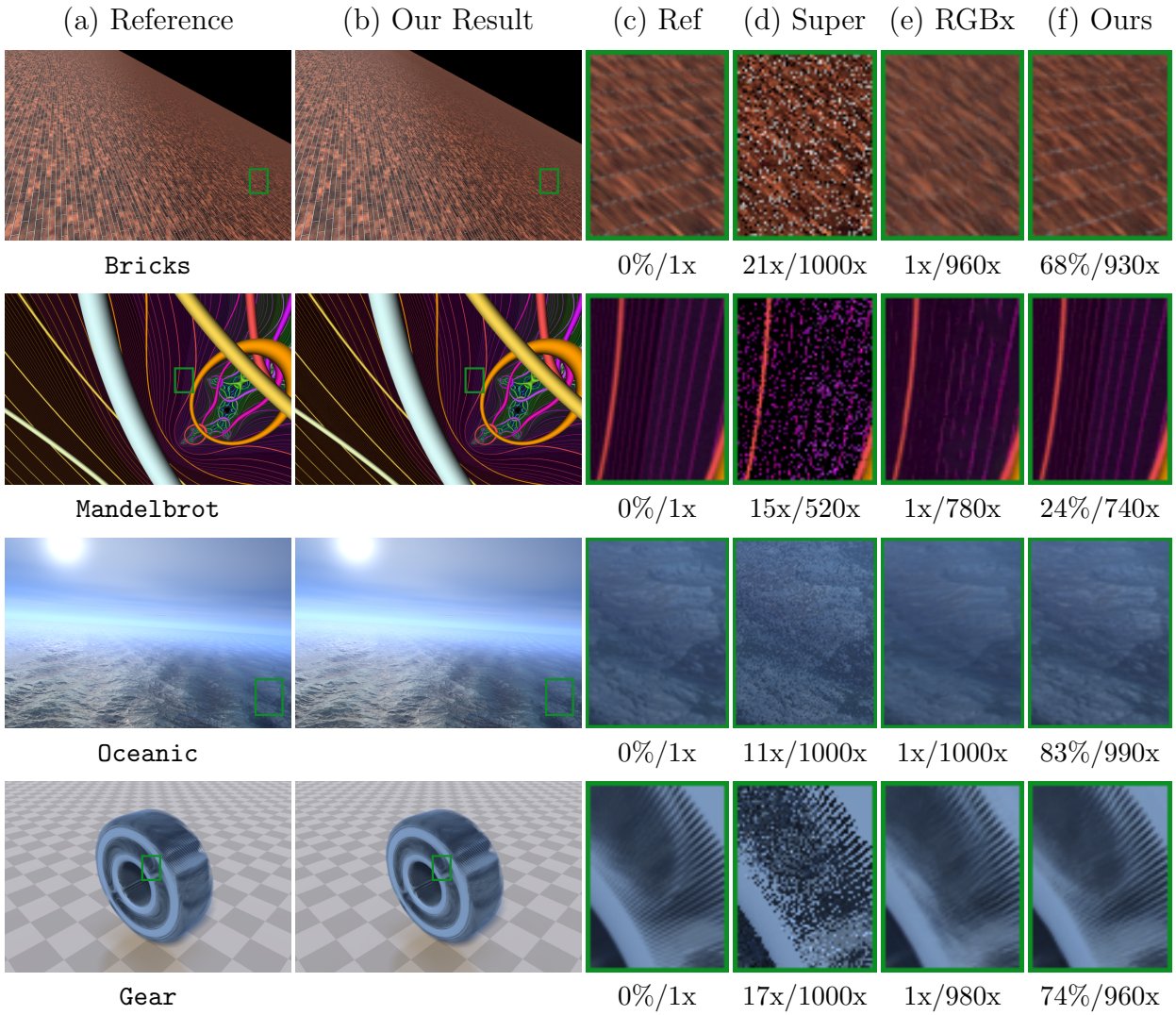


Figure 6.4: Learning to reduce sampling noise in procedural shaders. The reference low-noise solution (a) relies on 1000 samples per pixel (SPP). Our method (b) approximates the reference well at only 1 SPP. Zooming into the region boxed in green (c, f) reveals approximation error, which compares favorably with the two baselines: (d) supersampling (super) where the number of samples is chosen to have comparable run-time as ours, and (e) RGBx. Errors and speedups are reported as in Figure 6.1. Our method better covers both the orientation and high-frequency detail than the baselines. `Bricks` and `Mandelbrot` are gamma corrected to emphasize visual differences. Sample counts chosen in supersampling (d) are: 1 SPP (`Bricks`), 2 SPP (`Mandelbrot`), 1 SPP (`Oceanic`), and 1 SPP (`Gear`).

Table 6.1: Error statistics for applications in Sections 6.5 and 6.6.3. Errors reported as LPIPS [158] / SSIM / PSNR.

	Shader	RGBx	Ours
Denoising	`Bricks`	0.0141 / 0.981 / 36.68	**0.0097** / **0.987** / **38.29**
	`Gear`	0.0173 / 0.986 / 38.90	**0.0127** / **0.988** / **39.86**
	`Mandelbrot`	0.0235 / 0.973 / 36.07	**0.0059** / **0.986** / **38.55**
	`Mandel-bulb`	0.0185 / 0.962 / 32.14	**0.0118** / **0.975** / **34.18**
	`Oceanic`	0.0403 / 0.961 / 33.69	**0.0339** / **0.966** / **34.51**
	`Trippy Heart`	0.0696 / 0.856 / 26.30	**0.0543** / **0.886** / **27.27**
	`Venice`	0.0309 / 0.965 / 32.76	**0.0242** / **0.973** / **33.83**
Simplified	`Bricks`	0.0624 / 0.922 / 25.57	**0.0398** / **0.936** / **29.96**
	`Mandelbrot`	0.1111 / 0.826 / 27.04	**0.0430** / **0.949** / **31.15**
	`Mandel-bulb`	0.0932 / 0.812 / 25.22	**0.0600** / **0.856** / **26.85**
	`Trippy Heart`	0.2412 / 0.520 / 18.55	**0.1824** / **0.629** / **20.95**
	`Venice`	0.0404 / 0.957 / 31.50	**0.0285** / **0.965** / **32.72**
Post	Blur	0.0126 / 0.978 / 35.81	**0.0082** / **0.985** / **37.62**
	Sharpen	0.0881 / 0.833 / 24.26	**0.0693** / **0.868** / **25.31**
	Simp Sharpen	0.2675 / 0.477 / 17.20	**0.2154** / **0.587** / **19.35**
Shared	`Gear`	0.0251 / 0.984 / 38.16	**0.0173** / **0.986** / **38.66**
	`Mandelbrot`	0.0546 / 0.801 / 28.02	**0.0165** / **0.933** / **32.60**
	`Mandel-bulb`	0.0423 / 0.861 / 28.02	**0.0298** / **0.906** / **30.19**
	`Trippy Heart`	0.1048 / 0.815 / 19.99	**0.0755** / **0.857** / **23.88**

Training for 400 epochs typically takes between 6 and 32 hours. However, the `Oceanic` shader is slower, and takes about 7 days to train. Note that all shaders are trained using the same process over an identical architecture with a similar number of input channels; therefore the great variation in training time derives primarily from the cost of sampling from shader programs, not from learning.

In terms of the arithmetic average over all shaders, our method has a relative perceptual error of 67% compared to the RGBx baseline. A different baseline, Supersampling, is consistently worse than RGBx, with relative perceptual error ranging from 3x to 21x compared to RGBx (Table 6.2). We believe the dramatic improvements in relative perceptual error of our method over the baselines corresponds with the qualitatively better reconstruction of high-frequency details that we observe in the renderings (Figure 6.4c-f).

Table 6.2: Error statistics for denoising (Section 6.5.1) for our method and the supersampling baseline. We report LPIPS perceptual error [158], SSIM, and PSNR. The numbers reported are averaged across the entire test dataset. Our error metric is already reported in Table 6.1, we include it here for a clear comparison with Supersampling. The error metric of the RGBx baseline is also reported in Table 6.1 and is omitted here to save space. To enable a direct comparison with RGBx, we additionally report for both ours and supersampling their LPIPS perceptual error relative to that of the RGBx baseline (% or ×).

Shader	Ours	Supersampling
`Bricks`	**0.0097(68%) / 0.987 / 38.29**	0.2985(21×) / 0.839 / 24.14
`Gear`	**0.0127(73%) / 0.988 / 39.86**	0.2935(17×) / 0.880 / 24.71
`Mandelbrot`	**0.0059(24%) / 0.986 / 38.55**	0.3502(15×) / 0.746 / 23.46
`Mandel-bulb`	**0.0118(63%) / 0.975 / 34.18**	0.1580(8.5×) / 0.895 / 23.98
`Oceanic`	**0.0339(84%) / 0.966 / 34.51**	0.4314(11×) / 0.780 / 23.81
`Trippy Heart`	**0.0543(77%) / 0.886 / 27.27**	0.2178(3.1×) / 0.767 / 22.42
`Venice`	**0.0242(78%) / 0.973 / 33.83**	0.2893(9.4×) / 0.853 / 23.73

6.5.2 Reconstructing Simplified Shaders

We also explore a more challenging task: learning to reconstruct the appearance of a shader from its simplified variant. Shader simplification is commonly used as a lossy optimization that improves runtime while approximating the output of the original program. However, simplified programs often lose texture or geometry detail as compared with the original. For example, the simplified versions of `Mandelbrot` and `Mandel-bulb` shown in Figure 6.5d look obviously different from their original counterparts in Figure 6.5c. We, therefore, propose an application that learns to recover the denoised output of the original shader from the traces of the simplified shader program sampled at 1 SPP. To our knowledge, this paper is the first to propose this learning task.

We use two different techniques to simplify the shader programs: genetic programming simplification [121] (on `Bricks`) and loop perforation [119] (on all other shaders). Because the model needs to synthesize unseen texture, we use a spatial discriminator for this application, described in Section 6.4.3. Training for 800 epochs takes between 10 and 60 hours. Similar to the denoising application, the great variation in training time mostly comes from generating

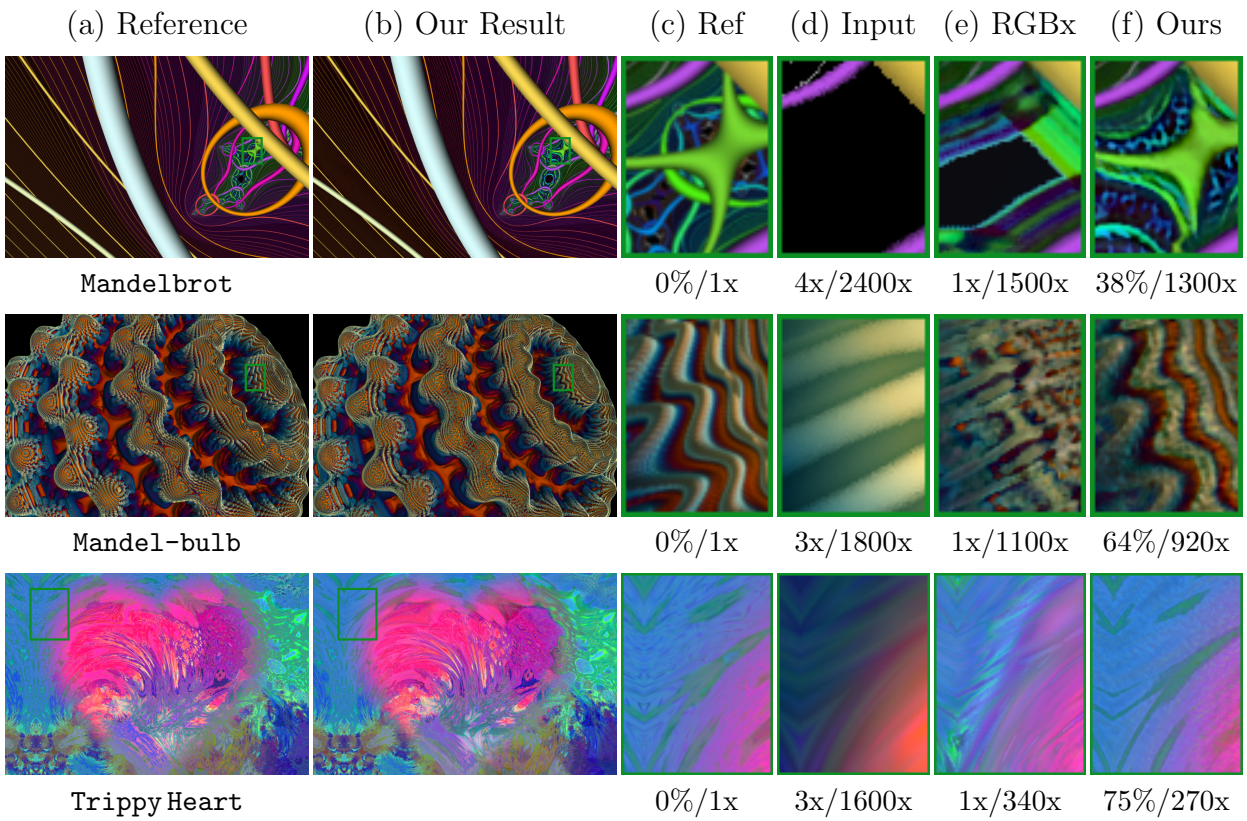


Figure 6.5: Learning from simplified shaders `Mandelbrot`, `Mandel-bulb` and `Trippy Heart`. Errors and speedups are reported as in Figure 6.1. In `Mandelbrot` our method better reconstructs missing regions due to oversimplification in the input. In `Mandel-bulb` our method better recovers the orientation of the texture. In `Trippy Heart` ours correctly recovers the color.

input samples from the shader. Our method has on average 62% perceptual error compared to the RGBx baseline.

6.5.3 Postprocessing Filters

Our method can be useful for learning not only denoising, but also applying additional image-space postprocessing filters. We implement two postprocessing filters on the CPU: an edge-aware sharpening filter [101] and defocus blur [115]. The network learns simultaneously to denoise and apply the postprocessing filter on the GPU. Figure 6.6 shows learning a defocus blur filter on `Mandel-bulb`, and learning a sharpening filter on simplified `Trippy Heart`. Our

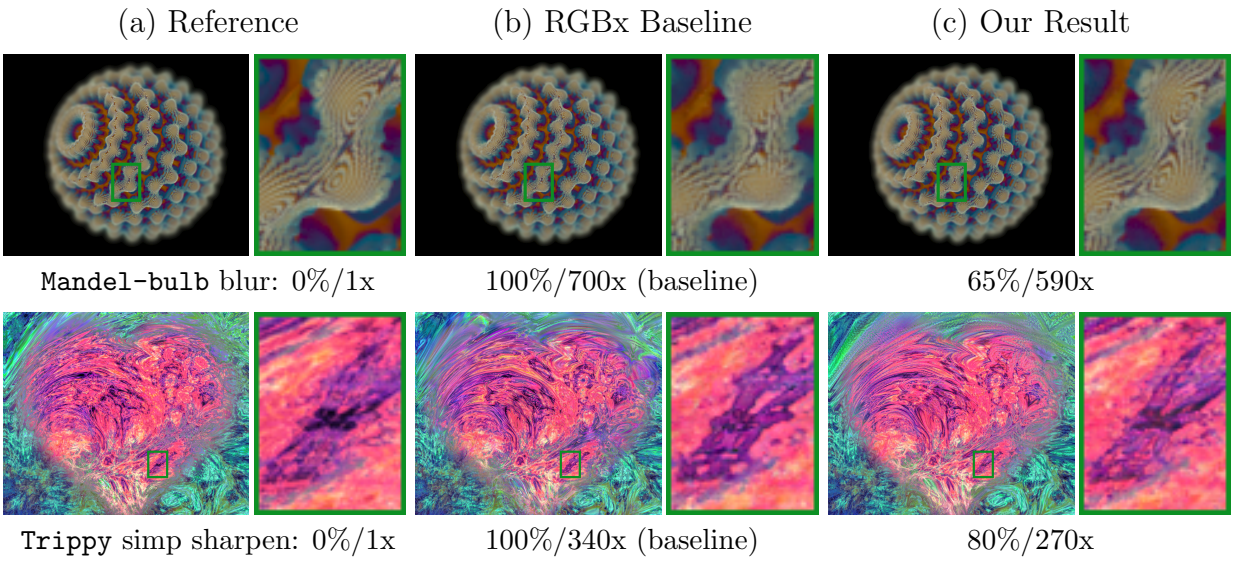


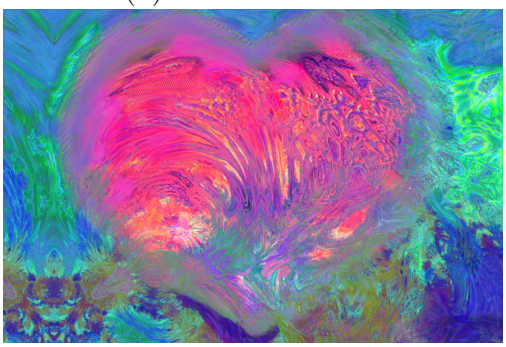
Figure 6.6: Learning postprocessing effects: defocusing blur for `Mandel-bulb` and sharpening for `Trippy Heart`. The reference solution (a) shows the result of a postprocessing filter applied to a low-noise shader rendering sampled at 1000 SPP. Both RGBx baseline (b) and our method (c) approximate the reference at 1 SPP. Our method recovers more faithfully the thin structure in `Mandel-bulb` and the color pattern in `Trippy Heart`. We report relative perceptual error speedup as in Figure 6.1. `Mandel-bulb` is gamma corrected so it can be viewed comfortably on darker displays.

Table 6.3: Error statistics for learning temporally coherent sequences. The metrics reported are similar as in Table 6.1. The temporal application is trained both on shaders with full computation and simplified shaders with partial computation (simp). For each experiment, we generate a 30 frames sequence and compute the error with respect to ground truth using the last frame. The reported numbers are averaged across 30 different sequences.

Shader	RGBx	Ours
`Mandelbrot`	0.0104 / 0.980 / 37.16	**0.0037** / **0.989** / **39.75**
`Mandelbrot` simp	0.1049 / 0.898 / 27.51	**0.0693** / **0.929** / **28.93**
`Mandel-bulb`	0.0213 / 0.959 / 32.06	**0.0140** / **0.971** / **33.56**
`Mandel-bulb` simp	0.1194 / 0.780 / 23.49	**0.1035** / **0.788** / **24.25**
`Trippy Heart`	0.0665 / 0.864 / 26.61	**0.0546** / **0.884** / **27.17**
`Trippy Heart` simp	0.2295 / 0.563 / 19.10	**0.1788** / **0.637** / **21.10**

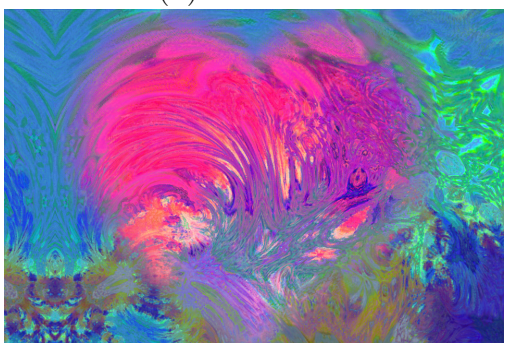
approach reproduces the complex effect more faithfully, as compared to RGBx, and the average relative perceptual error for ours is 74% of that of RGBx.

(a) RGBx Baseline



`Trippy Heart`: 100%

(b) Our result

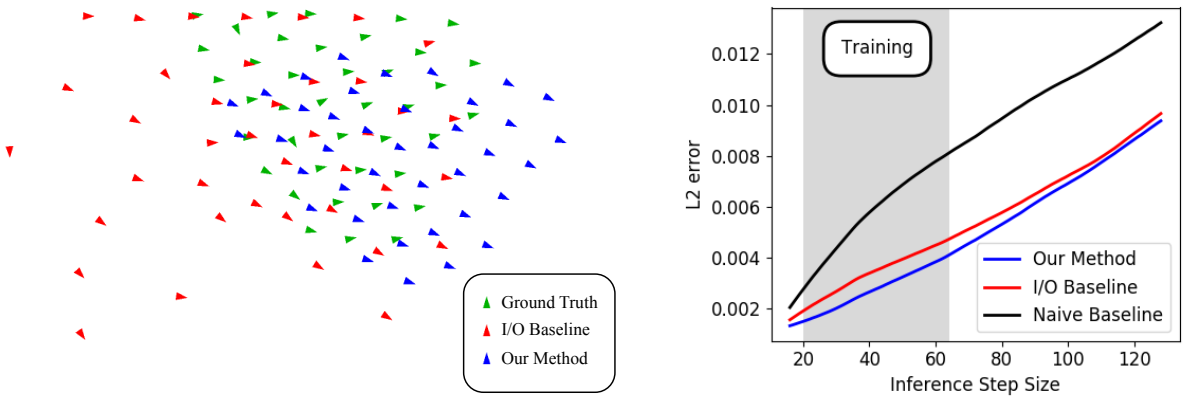


77%

Figure 6.7: Learning temporally coherent sequences for `Trippy Heart` with the same ground truth as in Figure 6.5. We report relative perceptual error compared to the RGBx baseline. Both RGBx (a) and ours (b) are the 90th frame of a synthesized temporally coherent sequence. Note how our method generalizes well to long sequences whereas the RGBx baseline presents obvious artifacts such as color residual from previous frames near the silhouette of the heart.

6.5.4 Training Temporally Coherent Sequences

Temporal coherence in a graphics or vision context refers to there being a strong correlation between each frame and the next. Training only on individual images can introduce temporal incoherence for rendered video. One straightforward fix would be to apply a temporal filter to the output sequences to blur out the noise. Alternatively, we implemented a temporal discriminator to directly train temporally coherent sequences using a training scheme similar to that of Wang et al. [134]. Each frame in a sequence is synthesized and conditioned on two previous frames. In training, frames are synthesized in groups of six consecutive frames, relying on eight-frame ground truth sequences to be able to bootstrap the initial frame. We train temporally coherent sequences both for the task of denoising and learning from simplified programs, and compare with an RGBx baseline as in Sections 6.5.1 & 6.5.2. A summary of quantitative error is shown in Table 6.3. In all cases, ours outperforms the RGBx baseline, and produces a more temporally coherent sequence than their non-temporal counterparts (Sections 6.5.1 & 6.5.2) while retaining similar visual quality in still images. We additionally verify that the temporal models generate more temporally stable sequences



(a) Visualization (b) Error Analysis

Figure 6.8: (a) Visualization for flocks of boids. Both the I/O baseline (red) and our method (blue) start from the same initial state and have taken 80 inference steps with step size 20. Our mean position as well as flocking behavior is more faithful to the ground truth (green). (b) We plot the average error as a function of step size, where training ranges from step size 20 to 64 (gray). Ours consistently outperforms both I/O and a more naive baseline, in the training range and beyond it.

by computing the perceptual loss of 2 adjacent frames. For each of the 30 test sequences, we use the last two frames of the length 30 sequence and average the score across ten renders with different random seeds. We then average the score across the test dataset and compare our temporal and our non-temporal models. In all cases, the temporal model has a lower error between adjacent frames. The temporal models have 94% perceptual error relative to the non-temporal models on average and 80% in the best case. Our supplementary video does not present temporally coherent animation as a separate application, but rather shows this training scheme in the denoising and simplification applications. Figure 6.7 shows an example where our method generalizes better to longer sequences than the RGBx baseline. Our result correctly learns both temporal coherence as well as the complicated structure in each individual frame, whereas the RGBx baseline introduces additional color artifacts in the output. The video shows even longer sequences (180 frames).

6.5.5 Learning to Approximate Simulation

Departing from learning from procedural pixel shader programs, we also explore learning to predict the future for shader programs that perform simulations. We study two simulations: a flock of boids and a fluid simulation.

6.5.5.1 Boids Simulation

Our first example simulates a flock of "boids" [113] which emulate the flocking behavior of birds or fish. Each boid has a 4-vector state representing 2D position and velocity. For a flock of K boids, the simulation program takes the input of a $K \times 4$ tensor that represents each individual boid's initial state, then updates the state based on repulsion and alignment forces. The updated state then becomes the input to the next simulation step, and so forth. The interaction between boids forms a complex flocking behavior that is difficult to predict. We run the ground truth simulation using a small δ step size: 2×10^6 steps with $\delta = \frac{1}{600}$s, targeting 20δ per frame at 30fps. During training, we further augment the data by randomly permuting boid indices. The learning task is to correct the simulation output from a larger time step $m \cdot \delta$ in order to approximate the boids' states as if the simulation ran m times for step size δ. (We train with $m \in [20, 64]$.) We compare our method with two baselines: a naive baseline that directly takes the larger step simulation without any correction, and an input/output (I/O) baseline that uses the input and the output of the larger step simulation as the input to a neural network.

The learning model is a combination of 1D convolution layers with 3 fully connected layers. The input to the network has size $B \times \mathcal{N}$ where B represents the number of boids (40 in our experiments) and $\mathcal{N}$ represents either the length of the program trace in our method or 8 for the I/O baseline. We first reduce the dimensionality of the trace to $\mathcal{K}$ using a 1D convolution with kernel size one, followed by 3 additional 1D convolutions with kernel size one and 48 output channels. This is an analogy to the 2D feature reduction layer and 1x1 convolutions described in Figure 6.3, where $\mathcal{K} = 48$ for our method and $\mathcal{K} = 1173$ for the

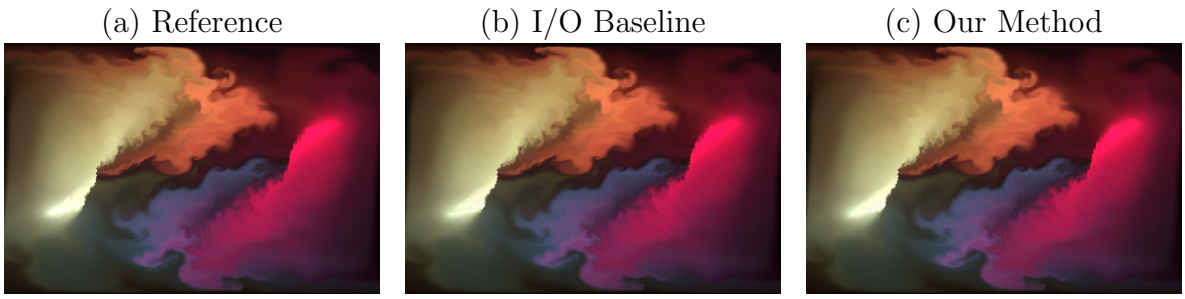


Figure 6.9: An example of fluid simulation where our method (c) gives a very similar result as the I/O baseline (b). This indicates our method may not be advantageous for simple learning tasks where the baseline is already good enough to reconstruct the reference (a). The relative perceptual error compared to the I/O baseline is reported below each image.

I/O baseline to match the number of trainable weights in both models. We then flatten the $B \times 48$ tensor as an input to a 3-layer fully connected network, where each layer has 256 hidden neurons, and an output fully connected layer with the number of neurons being $B \cdot 4$, representing the output state for each boid. For reported results, we use $B = 40$ and log every program trace from the boids program. We choose a larger program trace length than for the pixel shaders because the simulation considers all pairwise interactions between boids, and a larger program trace budget better captures these interactions.

We learn a four-channel state (2D position and velocity) for each boid, rather than RGB. Therefore we use only L_2 loss on these coordinates after separately normalizing position and velocity over the training set.

Figure 6.2 and 6.8 shows that ours always outperforms baselines numerically and visually, even when the step size is extrapolated outside the training range ($m \in [16, 128]$). The supplemental video shows that ours recovers individual boids' interaction behaviors more faithfully with a step size of $m = 20$, while the I/O baseline mainly learns the average position and velocity for the entire flock but fails to recover a reasonable distance between the boids.

6.5.5.2 Fluid Simulation

Although our method is beneficial in all the previously described experiments, we also find a null result for our second simulation example: a 2D fluid simulation. The state of the simulation on a 2D grid can be viewed as a 7D feature: 3D for the RGB color of the fluid and 4D for internal states: velocity, density, and vorticity. The simulation takes an input of the 7-channel fluid state, solves the Navier-Stokes equation with a hard-coded external force to compute the new internal state, then applies color advection on image space to output the new 7D state. The color advection step controls the trade-off between how fast the fluid propagates and how accurate the simulation is. We simulate with step size δ as the ground truth. The learning task is to run the simulation at a coarser step 10δ, and predict the intermediate states in between the 10 steps as if they were run at the fine scale simulation with step size δ.

We use the same architecture as in Section 6.5.1 for this task and compare our method with an I/O baseline that takes the initial and output fluid states as learning features. While our method is marginally numerically better than the baseline (ours has 92% L2 error and 96% perceptual error compared to the baseline), the visual quality of the two methods is almost identical. We hypothesize that this learning task is not suitable for our method because it is relatively simple and lacks a complicated hidden state: the neural network can easily approximate solving the Navier-Stocks equation given initial and output states. Additionally, because the fluid states change slowly even after 10 simulation steps, the network can easily hallucinate a reasonable correction using the initial state as a good starting point, therefore, the baseline features already suffice. Figure 6.9 shows both the baseline and our method can reasonably approximate the reference with almost identical results.

6.5.6 Branching and Loop Emulation

As discussed in Section 6.3.1, our compiler currently handles conditional execution by simply evaluating both branches and unrolling loops to their maximum possible iteration. Variable-length loops are handled using a user-given compile-time pragma specifying a ceiling on the possible number of loop iterations: it is common to have such ceilings on iteration counts in shader programs because of the need to maintain consistent shading performance. Values from unused iterations are replaced with the values from the final computed iteration. We made these choices because they are much easier to implement in TensorFlow. However, in a practical application, shaders would typically be compiled to code that takes either branch or exits the loop early based on a termination condition. Therefore, we experiment to determine what would have been the effect of handling branches and loops the traditional way. For branching, we simply wrote dummy values of zero to traces in the branch not taken. We applied such branch emulation to a shader called `TextureMaps` which—similar to aspects of `Venice` in Figure 6.1—uses a conditional statement to select a texture based on whether a ray has hit a plane. For loops, we wrote zero values to traces after the loop termination condition is met, and applied the emulation to `Mandelbrot`. In both cases, we found that the emulation gives results that are visually and quantitatively identical to our compiler's implementation.

6.5.7 Comparison with Positional Encoding

Positional encoding [82] can be viewed as a general method to augment input to learning that is agnostic to the input data's generation process. It applies high-frequency functions to positional features such as 3D coordinates. Because many shaders involve computing intermediate values that vary spatially in ways that cannot easily be captured via positional encoding, and some of them will be important to the learner, we believe our method offers an improvement over positional encoding for most shaders and most applications. To evaluate this hypothesis, we tested two applications (denoising in Section 6.5.1 and simplification

in Section 6.5.2) × two shaders (`Mandelbrot` and `Trippy Heart`), adding explicit positional encoding features as described by [82] to both the RGBx baseline and our method. On average across the four cases, we found that the addition of positional encoding features did not measurably change PSNR values. Moreover, the addition of positional encoding features increased the perceptual error of both RGBx and ours by 4% on average. Therefore, we verified our hypothesis that in the context of our applications and shaders, learning does not benefit from positional encoding.

6.6 Trace Analysis

This section presents a series of analyses that help to understand how program traces are beneficial for learning. We start by analyzing which trace features are contributing the most to a learned model. Based on trace importance, we then investigate which subset of the trace can be used for learning. We empirically find that if one cannot afford to first execute and learn from the full shader trace, then the Uniform subsampling used throughout Section 6.5 always gives a reasonable performance, and we were not able to find any strategy that consistently outperforms Uniform. However, if one can train an additional initial network that first uses the full program trace, then we can do better than Uniform, using a strategy that we call Oracle that selects important features. Finally, we show that multiple shaders can be trained together with a shared denoising network and a lightweight shader-specific encoder.

6.6.1 Which Trace Features Matter in a Learned Model?

We characterize the importance of the trace features by quantifying the change in training loss when removing each of the trace inputs. Inspired by Molchanov et al. [87, 88], we used the first-order Taylor expansion to approximate the importance of each input trace feature. Specifically, for a model trained with loss L and trace length $\mathcal{T}$, the importance score Θ of

the input trace feature $\mathbf{z}^l$ $(l = 1, ..., \mathcal{T})$ with image dimension $M \times N$ across K examples is:

$$\Theta(\mathbf{z}^l) = \frac{1}{K} \sum_{k=1}^{K} |\frac{1}{M \cdot N} \sum_{m=1}^{M} \sum_{n=1}^{N} \frac{\partial L}{\partial \mathbf{z}^l_{m,n}} \cdot \mathbf{z}^l_{m,n}| \tag{6.5}$$

We evaluate Equation 6.5 on the denoising model for two shaders: `Bricks` and `Mandelbrot`. Only a small fraction of the trace results in a very high importance score. We manually inspect what the top 10% most important trace features represent and verified that the learned importance corresponds to human intuition. For example in `Bricks`, we found the most important traces include features that determine the distance to the nearest brick edges and the Boolean condition that decides whether the pixel is inside the mortar: this helps prevent edges from being broken. In `Mandelbrot`, we found the trace that controls the complex number computation for almost every iteration is among the most important features: a breakdown of such information at each level could help the model to better denoise between nearby structures.

6.6.2 Which Subset of the Trace to Use for Learning?

As discussed in Section 6.3.2, program traces can be arbitrarily long, and we could input only a subset of the trace for efficient learning and inference, such as the Uniform subsampling used in Section 6.5. Therefore, a natural question to ask is: given a fixed input trace length budget, what subsets of the program trace are good for learning? The best way to answer this question is to enumerate all possible subsets of the program trace and train a separate model for each. However, for a shader program that has $\mathcal{T}$ traces before subsampling and a fixed input budget $\mathcal{N}$, this strategy will introduce combinatoric $\binom{\mathcal{T}}{\mathcal{N}}$ learning tasks, which are intractable.

To investigate how different subsets of the trace could practically affect learning, we propose subsampling strategies we call Oracle and Opponent. Both the Oracle and Opponent strategies are based on the feature importance score (Section 6.6.1) from a Full Trace model

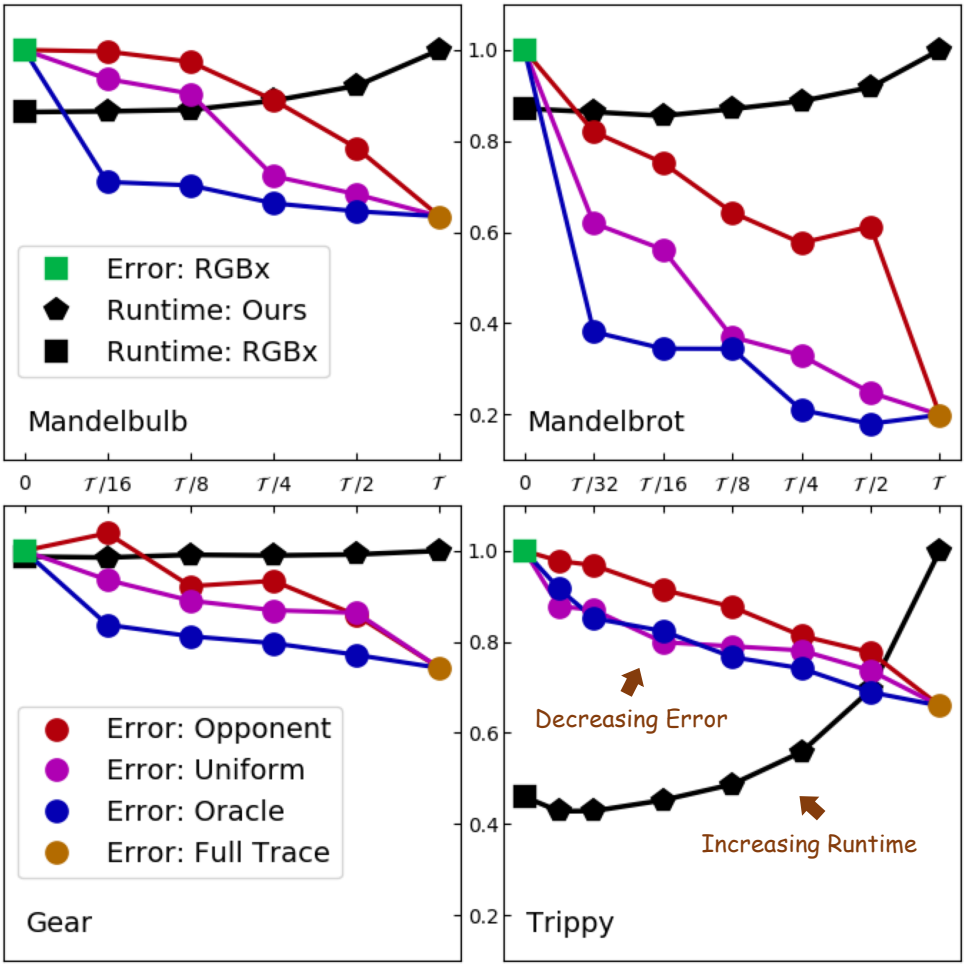


Figure 6.10: Error vs. Time trade-off for Opponent, Uniform, and Oracle subsampling strategies with varying trace length. For each shader with $\mathcal{T}$ program traces, we subsample the trace such that the actual input trace length $\mathcal{N}$ is equal to $\mathcal{T}/2$, $\mathcal{T}/4$, $\mathcal{T}/8$, etc., and the x-axis shows each model's relative trace ratio compared to the full program trace. The circle shows perceptual error relative to the RGBx baseline (green square) for Opponent (red), Uniform (purple), Oracle (blue), and Full Trace (gold). The black pentagon shows the relative inference time (including shader program runtime and network inference) for each $\mathcal{N}$ relative to the runtime of the Full Trace model. Similarly black square shows relative inference time for the RGBx baseline. Note that the x-axis is on a log scale in relative trace length compared to the Full Trace model, therefore although the relative error plot appears linear as $\mathcal{N}$ increases, the actual performance improvement is faster at the beginning of the plot: adding only a few traces quickly improves performance.

trained with all of the program trace. Oracle always chooses the traces that have the highest importance scores, while the Opponent always chooses the ones with the lowest scores. In an analogy to the lottery ticket hypothesis [43], we hypothesize that the Oracle exploits a winning "lottery ticket" found within the Full Trace model, and selects out the relevant trace subset: a "lottery ticket trace." The Opponent likewise selects losing tickets.

To better understand the trade-offs associated with the subsampled trace length, we experimented with varying trace lengths using Opponent, Uniform, and Oracle subsampling and compare them with the RGBx baseline, as shown in Figure 6.10. For each shader, the trace is subsampled by a relative sample budget compared to the full program trace length $\mathcal{T}$ (e.g. $\mathcal{N}= \mathcal{T}/2$, $\mathcal{T}/4$). Under a fixed budget $\mathcal{N}$, in most cases, the inference error decreases in the ordering of Opponent, Uniform, and Oracle. This corresponds to our intuition because Oracle selects traces that are beneficial to training based on prior knowledge from the Full Trace model, and similarly Opponent selects traces that are unimportant based on the same prior knowledge.

We provide statistical evidence for our findings when investigating trade-offs between different subsampling strategies and subsampling budgets described in Section 6.6.2. Our first null hypothesis makes the following assumption on the performance between Uniform and Oracle subsampling: the ratio of relative error between Uniform and Oracle (μ_0) is less than or equal to 1. This hypothesis has a p-value of $p_0 = 7.2 \times 10^{-4}$. Similarly, we propose another null hypothesis regarding the performance between Opponent and Uniform subsampling: the ratio of relative error between Opponent and Uniform (μ_1) is smaller than or equal to 1, which has a p-value $p_1 = 5.9 \times 10^{-3}$. If we choose a significance level of 0.05 and apply Bonferroni correction over the 2 hypotheses, we have both $p_0 < 0.025$ and $p_1 < 0.025$, indicating significant evidence that Oracle outperforms Uniform ($\mu_0 > 1$) and Uniform outperforms Opponent ($\mu_1 > 1$). These statistics are computed using all possible $\mathcal{N}$ available for all 4 shaders: $\mathcal{N} \in [\mathcal{T}/2, \mathcal{T}/4, \mathcal{T}/8, \mathcal{T}/16]$.

To summarize, our hypotheses that Uniform outperforms Oracle, and Opponent outperforms Uniform each has p-values 7.2×10^{-4} and 5.9×10^{-3}, respectively. These are smaller than a threshold of 0.025 determined by correcting the traditional p threshold of 0.05 for the two comparisons, so we conclude that the ordering *Oracle outperforms Uniform outperforms Opponent* is significant.

It is also worth noting that even when $\mathcal{N}$ is small (e.g. the leftmost two data points in the plots correspond to $\mathcal{N}$ below 50), the extra information from the program trace can still substantially reduce the relative perceptual error without significant extra cost in inference time. Because the x-axis in the plot is on a log scale, the actual performance gain would have a more drastic slope starting from RGBx to a small $\mathcal{N}$. Additionally, the current comparison is advantageous to RGBx as its learning capacity matches that of the Full Trace model as discussed in Section 6.5, which is more capacity than any of the subsampled models in Figure 6.10.

In practice, subsampling strategies can be chosen based on resources allowed for training and inference. If there is no limit at all, training a model with the Full Trace can always give the best performance. If $\mathcal{N}$ is only limited by inference time, but extra cost and memory can be permitted during training, one could use the Oracle strategy. However, when training also becomes a practical concern, our results suggest that without actually learning from the full trace in advance, there may not be a single subsampling strategy that could consistently outperform all others, as discussed in Section 6.3.2. Thus, Uniform subsampling provides an effective proxy that follows the performance of Oracle, and always outperforms the worst-case scenario Opponent.

6.6.3 Can Multiple Shaders be Learnt Together?

This section explores whether part of the model can be shared across shaders with the same task. Because program traces are unique per shader, we propose to train a separate shallow

encoder for each of the shaders, followed by a task-specific model shared across shaders. The setup is similar to the source-aware encoder by Vogels et al. [132].

Four shaders (`Mandelbrot`, `Mandel-bulb`, `Gear`, and `Trippy Heart`) are trained together for the denoising task. The encoder consists of four 1x1 convolutional layers, where the first layer outputs $\mathcal{K}$ channels and the rest output 48 channels. In our method, $\mathcal{K} = 48$ while in the RGBx baseline $\mathcal{K}$ varies similarly as in Section 6.5. The encoder is identical to the four 1x1 convolutions that analyze the input program trace in Figure 6.3. The 48-dimensional encoding then inputs to a shared denoising network, whose architecture is identical to Figure 6.3 excluding the four initial 1x1 convolutions. All four shaders use Uniform subsampling to bring their $\mathcal{N}$ to closest to 200. Training alternates between the 4 shaders after each epoch, and each shader is trained for 400 epochs.

We report the error statistics for the shared model in Table 6.1. Ours has on average 60% perceptual error compared to the RGBx baseline. Although one might expect this experiment to benefit the RGBx baseline as the RGBx features are more similar, in fact, ours outperforms RGBx in all cases.

6.7 Summary and Discussions

This chapter proposes the idea of learning from shader program traces. It evaluates this idea in a range of learning contexts: denoising, simplified shaders, postprocessing filters, and simulation. We describe a compiler that can produce program traces suitable for learning, as well as practical considerations like how to handle large traces and how to process the trace data to make it amenable to learning. Our method is agnostic to the learning architecture, loss function, and training process; however, we also discuss a particular set of these that worked well in our experiments. We evaluate our method on a range of shaders, over which it compares favorably with baselines. We also analyze which features are important in the trace, and explain how one can select subsets of the trace for learning.

Links to our code, data, and supplemental video may be found on our project page: `https://pixl.cs.princeton.edu/pubs/Yang_2022_LFS/`

Our method has several limitations, which offer potential avenues for future work. First, as with many neural network-based approaches, the inference time is not negligible. For example, for the denoising task, simple, fast shaders can draw sufficiently many samples via supersampling to outperform inference. Likewise for the simplified shader tasks, one could use the time budget for network inference to instead compute multiple samples of the original more expensive shader. Future research might address this by developing specialized networks that are more efficient for inference, along similar lines as the method of Gharbi et al. [47]. Another limitation of our TensorFlow implementation is that the operation of collecting program traces and concatenating them into one single tensor is not particularly efficient, and is a major cause of the inference time increasing with the trace length (Figure 6.10). Additionally, TensorFlow is efficient for deep learning models, but less efficient for shader computations. For example, for shaders with complex BRDFs, branching generated by varying numbers of ray bounces per pixel may become a major bottleneck in TF. We believe further engineering efforts could alleviate these bottlenecks, for example by using compiled GLSL as a frontend renderer before the inference process.

The experiments described in this chapter were performed using computer graphics shaders. Future work could explore how well the ideas introduced herein generalize to other kinds of programs that can rely on (and tolerate) approximate solutions, for example, those relying on stochastic algorithms or Markov-like decision processes.

Chapter 7

Conclusion and Future Work

Program representations include rich information both semantically and algorithmically. While domain experts can leverage the program structure to manually design a better solution for various tasks, there is a nontrivial gap to automating the process to arbitrary programs. This dissertation takes steps to bridge the gap in three important tasks. We first extend the traditional AD framework to automatically differentiate arbitrary discontinuous programs (Chapter 2) and demonstrate its application in both procedural shader programs (Chapter 3) and audio synthesizers (Chapter 4). Second, we explore automatic convolution that smoothes a program by approximating its convolution with a Gaussian kernel (Chapter 5), and show it can be used for bandlimiting procedural shaders. Finally, we also demonstrate that if the input to a deep learning model is programmatically generated, the model could benefit from learning from the entire program trace (Chapter 6). We verify that the additional auxiliary input features help a variety of visual and simulation learning tasks.

Based on this dissertation, I envision the following research directions to be promising:

Differentiating Discrete Parameters. Discrete parameter examples include categorical choices that set the program into different modes, and integer-valued hyperparameters for loop iteration or filter size. This type of parameter usually controls discontinuities on its

own, and do not interact with continuously sampled parameters such as pixel coordinates and time. As a result, the associated discontinuities are rarely sampled, therefore obtaining their gradients using our Aδ rule is challenging. We argue perturbation should be applied to these discrete parameters along the sampling axes for the discontinuities to be efficiently sampled. For example, our audio synthesizer application (Chapter 4) proposes one such solution: we apply random noise to the categorical choices such that different choices can be sampled at different times. Future work could explore more general differentiation rules such as the ones we propose in Section 2.4.2 for correctly differentiating the discrete parameters.

Differentiating Numerical Approximations. Many math or physics problems do not have closed-form solutions, and are usually approximated numerically in programs. For example, the FFT of a shader rasterization is a discrete approximation to the Fourier transform of the original continuous image defined by the shader program. Similarly, the ray object intersection found by a raymarching loop is a numerical root-finding approximation as well. While AD or Aδ can be applied to the numerical approximation programs, the generated gradient may amplify the noise within the approximation, such as those caused by phase offset or the exact location of the discretization. As an alternative, one may develop a specialized gradient for each numerical problem by directly differentiating the underlying math representation *before* the numerical approximation. This may achieve better gradient accuracy, as well as potential improvement in runtime. As an example, Section 3.1 differentiates the implicitly defined geometry that bypasses the raymarching iterations, generating a more accurate gradient with less memory footage and faster runtime.

End-to-End Optimization with Combinations of White-Box and Black-Box Modules. Deep learning proxies are widely used to replace modules within traditional pipelines such as for rendering or camera. However, the neural pipeline usually has limited white-box components [131] because many traditional white-box algorithms are non-differentiable. We imagine the Aδ framework can be used in combination with traditional AD to allow users to

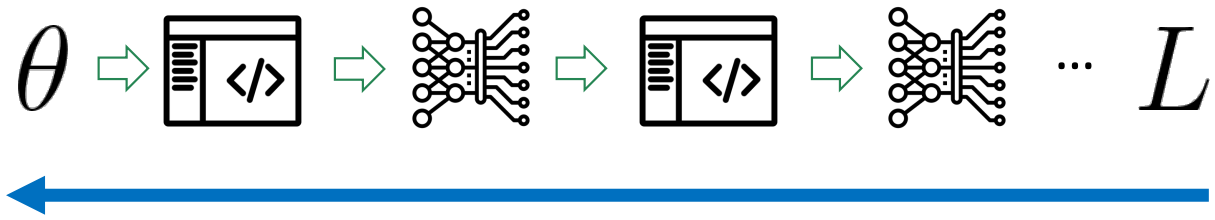


Figure 7.1: Rendering or camera pipeline can be emulated using both black-box neural proxies and white-box programs. They can be differentiated end-to-end (indicated by the blue arrow) using both AD and Aδ differentiating frameworks.

freely emulate a pipeline with both neural proxies and discontinuous white-box programs as in Figure 7.1. Therefore the composition can both easily inverse engineer black-box components through data-driven neural proxies, as well as enjoy the flexibility and interpretability provided by the white-box algorithms, leading to great potential for combining recent advances in neural proxies [131, 128] and classic algorithms.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] 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. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4), jul 2019.

[3] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically translating image processing libraries to halide. *ACM Trans. Graph.*, 38(6), nov 2019.

[4] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 2008.

[5] Alexander Panasovsky. Celtic. `https://thenounproject.com/icon/celtic-1975448/`, 2018.

[6] Anthony A Apodaca, Larry Gritz, and Ronen Barzel. *Advanced RenderMan: Creating CGI for motion pictures*. Morgan Kaufmann, 2000.

[7] M Baker and S Sutlief. Green's functions in physics version 1. 2003.

[8] Steve Bako, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, Tony DeRose, and Fabrice Rousselle. Kernel-predicting convolutional networks for denoising monte carlo renderings. *ACM Transactions on Graphics (TOG) (Proceedings of SIGGRAPH 2017)*, 36(4), July 2017.

[9] Sai Bangaru, Tzu-Mao Li, and Frédo Durand. Unbiased warped-area sampling for differentiable rendering. *ACM Trans. Graph.*, 39(6):245:1–245:18, 2020.

[10] Sai Bangaru, Jesse Michel, Kevin Mu, Gilbert Bernstein, Tzu-Mao Li, and Jonathan Ragan-Kelley. Systematically differentiating parametric discontinuities. *ACM Trans. Graph.*, 40(107):107:1–107:17, 2021.

[11] Oren Barkan, David Tsiris, Ori Katz, and Noam Koenigstein. Inversynth: Deep estimation of synthesizer parameter configurations from audio signals. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 27(12):2385–2396, 2019.

[12] Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1):171–190, 2000. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[13] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

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

[15] Adam Brady, Jason Lawrence, Pieter Peers, and Westley Weimer. genbrdf: Discovering new analytic brdfs with genetic programming. *ACM Transactions on Graphics (TOG)*, 33(4):114, 2014.

[16] Antoni Buades, Bartomeu Coll, and J-M Morel. A non-local algorithm for image denoising. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 60–65. IEEE, 2005.

[17] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. Non-local means denoising. *Image Processing On Line*, 1:208–212, 2011.

[18] Peter Burt and Edward Adelson. The laplacian pyramid as a compact image code. *IEEE Transactions on communications*, 31(4):532–540, 1983.

[19] Zoya Bylinskii, Tilke Judd, Ali Borji, Laurent Itti, Frédo Durand, Aude Oliva, and Antonio Torralba. Mit saliency benchmark, 2019.

[20] Franco Caspe, Andrew McPherson, and Mark Sandler. Ddx7: Differentiable fm synthesis of musical instrument sounds. *arXiv preprint arXiv:2208.06169*, 2022.

[21] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph.*, 36(4):98:1–98:12, July 2017.

[22] Swarat Chaudhuri and Armando Solar-Lezama. Smoothing a program soundly and robustly. In *International Conference on Computer Aided Verification*, pages 277–292. Springer, 2011.

[23] Bintong Chen and Xiaojun Chen. A global and local superlinear continuation-smoothing method for p 0 and r 0 ncp or monotone ncp. *SIAM Journal on Optimization*, 9(3):624–645, 1999.

[24] Bintong Chen and Naihua Xiu. A global linear and local quadratic noninterior continuation method for nonlinear complementarity problems based on chen–mangasarian smoothing functions. *SIAM Journal on Optimization*, 9(3):605–623, 1999.

[25] Li Chen, Salmin Sultana, and Ravi Sahita. Henet: A deep learning approach on intel® processor trace for effective exploit detection. *CoRR*, abs/1801.02318, 2018.

[26] Qifeng Chen, Jia Xu, and Vladlen Koltun. Fast image processing with fully-convolutional networks. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[27] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[28] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

[29] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1):51–72, January 1986.

[30] Marcella Cornia, Lorenzo Baraldi, Giuseppe Serra, and Rita Cucchiara. A deep multi-level network for saliency prediction. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 3488–3493, 2016.

[31] Franklin C Crow. The aliasing problem in computer-generated shaded images. *Communications of the ACM*, 20(11), 1977.

[32] Franklin C Crow. Summed-area tables for texture mapping. *ACM SIGGRAPH computer graphics*, 18(3):207–212, 1984.

[33] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[34] Mark AZ Dippé and Erling Henry Wold. Antialiasing through stochastic sampling. *ACM Siggraph Computer Graphics*, 19(3), 1985.

[35] Jonathan Dorn, Connelly Barnes, Jason Lawrence, and Westley Weimer. Towards automatic band-limited procedural shaders. In *Computer Graphics Forum*, volume 34, pages 77–87. Wiley Online Library, 2015.

[36] David S Ebert. *Texturing & modeling: a procedural approach*. Morgan Kaufmann, 2003.

[37] Jesse Engel, Lamtharn Hantrakul, Chenjie Gu, and Adam Roberts. Ddsp: Differentiable digital signal processing. *arXiv preprint arXiv:2001.04643*, 2020.

[38] Jesse Engel, Cinjon Resnick, Adam Roberts, Sander Dieleman, Mohammad Norouzi, Douglas Eck, and Karen Simonyan. Neural audio synthesis of musical notes with wavenet autoencoders. In *International Conference on Machine Learning*, pages 1068–1077. PMLR, 2017.

[39] Yuri M Ermoliev and Vladimir I Norkin. On nonsmooth and discontinuous problems of stochastic systems optimization. *European Journal of Operational Research*, 101(2):230–244, 1997.

[40] Yuri M Ermoliev, Vladimir I Norkin, and Roger JB Wets. The minimization of semicontinuous functions: mollifier subgradients. *SIAM Journal on Control and Optimization*, 33(1):149–167, 1995.

[41] AE Feiguin. Monte carlo error analysis, 2011. [Online; accessed 22-May-2017].

[42] Peter H Feiler and Watts S Humphrey. Software process development and enactment: Concepts and definitions. In *Software Process, 1993. Continuous Software Process Improvement, Second International Conference on the*, pages 28–40. IEEE, 1993.

[43] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[44] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.

[45] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. *CoRR*, abs/1804.01118, 2018.

[46] Pau Gargallo, Emmanuel Prados, and Peter Sturm. Minimizing the reprojection error in surface reconstruction from images. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8, 2007.

[47] Michaël Gharbi, Jiawen Chen, Jonathan T Barron, Samuel W Hasinoff, and Frédo Durand. Deep bilateral learning for real-time image enhancement. *ACM Transactions on Graphics (TOG)*, 36(4):118, 2017.

[48] Michaël Gharbi, Tzu-Mao Li, Miika Aittala, Jaakko Lehtinen, and Frédo Durand. Sample-based monte carlo denoising using a kernel-splatting network. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[49] Ian Goodfellow. NIPS 2016 tutorial: Generative adversarial networks. *CoRR*, abs/1701.00160, 2017.

[50] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. *CoRR*, abs/1606.03401, 2016.

[51] Toshiya Hachisuka, Wojciech Jarosz, Richard Peter Weistroffer, Kevin Dale, Greg Humphreys, Matthias Zwicker, and Henrik Wann Jensen. Multidimensional adaptive sampling and reconstruction for ray tracing. In *ACM Transactions on Graphics (TOG)*, volume 27, page 33. ACM, 2008.

[52] Carsten Hahn, Thomy Phan, Thomas Gabor, Lenz Belzner, and Claudia Linnhoff-Popien. Emergent escape-based flocking behavior using multi-agent reinforcement learning. *CoRR*, abs/1905.04077, 2019.

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

[54] Yong He, Tim Foley, Natalya Tatarchuk, and Kayvon Fatahalian. A system for rapid, automatic shader level-of-detail. *ACM Transactions on Graphics (TOG)*, 34(6):1–12, 2015.

[55] Paul S Heckbert. Filtering by repeated integration. In *ACM SIGGRAPH Computer Graphics*, volume 20, pages 315–321. ACM, 1986.

[56] Inigo Quilez. Distance functions. `https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm`, 2021.

[57] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 5967–5976. IEEE, 2017.

[58] N. Y. Jeppu, T. Melham, D. Kroening, and J. O'Leary. Learning concise models from long execution traces. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[59] Chiyu Max Jiang, Avneesh Sud, Ameesh Makadia, Jingwei Huang, Matthias Nießner, and Thomas A. Funkhouser. Local implicit grid representations for 3d scenes. *CoRR*, abs/2003.08981, 2020.

[60] Nima Khademi Kalantari, Steve Bako, and Pradeep Sen. A machine learning approach for filtering monte carlo noise. *ACM Trans. Graph.*, 34(4):122, 2015.

[61] Andrew Kensler and Peter Shirley. Optimizing ray-triangle intersection via automated search. In *Interactive Ray Tracing, IEEE Symp., 2006*. IEEE.

[62] Seung Wook Kim, Yuhao Zhou, Jonah Philion, Antonio Torralba, and Sanja Fidler. Learning to Simulate Dynamic Environments with GameGAN. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2020.

[63] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[64] Matthias Kummerer, Thomas SA Wallis, Leon A Gatys, and Matthias Bethge. Understanding low-and high-level contributions to fixation prediction. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4789–4798, 2017.

[65] Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. Procedural noise using sparse gabor convolution. In *ACM Transactions on Graphics (TOG)*, volume 28, page 54. ACM, 2009.

[66] James R. Larus. Efficient program tracing. *Computer*, 26(5):52–61, 1993.

[67] Gwendal Le Vaillant, Thierry Dutoit, and Sébastien Dekeyser. Improving synthesizer programming from variational autoencoders latent space. In *Proceedings of the 24th International Conference on Digital Audio Effects (DAFx20in21)*, September 2021.

[68] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *CoRR*, abs/1608.08710, 2016.

[69] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 37(6):222:1–222:11, 2018.

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

[71] Tzu-Mao Li, Michal Lukáč, Gharbi Michaël, and Jonathan Ragan-Kelley. Differentiable vector graphics rasterization for editing and learning. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 39(6):193:1–193:15, 2020.

[72] Yijun Li, Jia-Bin Huang, Narendra Ahuja, and Ming-Hsuan Yang. Joint image filtering with deep convolutional networks. *arXiv preprint arXiv:1710.04200*, 2017.

[73] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. Soft rasterizer: A differentiable renderer for image-based 3d reasoning. *CoRR*, abs/1904.01786, 2019.

[74] Yan-Li Liu, Jin Wang, Xi Chen, Yan-Wen Guo, and Qun-Sheng Peng. A robust and fast non-local means algorithm for image denoising. *Journal of computer science and technology*, 23(2):270–279, 2008.

[75] Liming Lou, Paul Nguyen, Jason Lawrence, and Connelly Barnes. Image perforation: Automatically accelerating image pipelines by intelligently skipping samples. *ACM Transactions on Graphics (TOG)*, 35(5):153, 2016.

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

[77] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[78] Grzegorz Łysik. Mean-value properties of real analytic functions. *Archiv der Mathematik*, 98(1):61–70, 2012.

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

[80] Naotake Masuda and Daisuke Saito. Synthesizer sound matching with differentiable dsp. In *ISMIR*, pages 428–434, 2021.

[81] Elie Michel and Tamy Boubekeur. Dag amendment for inverse control of parametric shapes. *ACM Transactions on Graphics*, 40(4):173:1–173:14, 2021.

[82] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

[83] Don P Mitchell. Generating antialiased images at low sampling densities. In *ACM SIGGRAPH 1987*, volume 21. ACM, 1987.

[84] Don P Mitchell. Spectrally optimal sampling for distribution ray tracing. In *ACM SIGGRAPH 1991*, volume 25. ACM, 1991.

[85] Boris Mityagin. The zero set of a real analytic function. *arXiv preprint arXiv:1512.07276*, 2015.

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

[87] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. *CoRR*, abs/1906.10771, 2019.

[88] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *CoRR*, abs/1611.06440, 2016.

[89] Ramon E Moore. *Methods and applications of interval analysis*. SIAM, 1979.

[90] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. 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*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[91] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4), jul 2016.

[92] Oliver Nalbach, Elena Arabadzhiyska, Dushyant Mehta, Hans-Peter Seidel, and Tobias Ritschel. Deep shading: Convolutional neural networks for screen-space shading. 36(4), 2017.

[93] Yu Nesterov. Smooth minimization of non-smooth functions. *Mathematical programming*, 103(1):127–152, 2005.

[94] Michael Niemeyer, Lars M. Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3d representations without 3d supervision. *CoRR*, abs/1912.07372, 2019.

[95] Merlin Nimier-David, Sébastien Speierer, Benoît Ruiz, and Wenzel Jakob. Radiative backpropagation: An adjoint method for lightning-fast differentiable rendering. *ACM Trans. Graph.*, 39(4), jul 2020.

[96] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: A retargetable forward and inverse renderer. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 38(6), December 2019.

[97] Alan Norton, Alyn P Rockwood, and Philip T Skolmoski. Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space. In *ACM SIGGRAPH*, volume 16, pages 1–8. ACM, 1982.

[98] Marc Olano, Bob Kuehne, and Maryann Simmons. Automatic shader level of detail. In *Proceedings of 2003 ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware*. Eurographics, 2003.

[99] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[100] Stanley Osher, Howard Heaton, and Samy Wu Fung. A hamilton–jacobi-based proximal operator. *Proceedings of the National Academy of Sciences*, 120(14):e2220469120, 2023.

[101] Sylvain Paris, Samuel W. Hasinoff, and Jan Kautz. Local laplacian filters: Edge-aware image processing with a laplacian pyramid. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 68:1–68:12, 2011.

[102] Jeong Joon Park, Peter Florence, Julian Straub, Richard A. Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. *CoRR*, abs/1901.05103, 2019.

[103] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[104] Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, and Alexei A Efros. Context encoders: Feature learning by inpainting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2536–2544, 2016.

[105] Fabio Pellacini. User-configurable automatic shader simplification. In *ACM Transactions on Graphics 2005*, volume 24. ACM, 2005.

[106] Ken Perlin. Noise hardware. *Real-Time Shading SIGGRAPH Course Notes*, 2001.

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

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

[109] Kaare Brandt Petersen, Michael Syskind Pedersen, et al. The matrix cookbook. *Technical University of Denmark*, 7:15, 2008.

[110] William H. Press, Saul a. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in Fortran 77: the Art of Scientific Computing. Second Edition*, volume 1. 1996.

[111] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[112] S. Reed and N. D. Freitas. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2016.

[113] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *SIGGRAPH Computer Graphics*, 21(4):25–34, July 1987.

[114] Daniel Ritchie, Anna Thomas, Pat Hanrahan, and Noah Goodman. Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29, pages 622–630. Curran Associates, Inc., 2016.

[115] Przemyslaw Rokita. Fast generation of depth of field effects in computer graphics. *Computers & Graphics*, 17(5):593–595, 1993.

[116] Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. Adaptive rendering with non-local means filtering. *ACM Transactions on Graphics*, 31(6):195, 2012.

[117] Siyuan Shan, Lamtharn Hantrakul, Jitong Chen, Matt Avent, and David Trevelyan. Differentiable wavetable synthesis. In *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4598–4602. IEEE, 2022.

[118] Jordie Shier, George Tzanetakis, and Kirk McNally. Spiegelib: An automatic synthesizer programming library. In *Audio Engineering Society Convention 148*. Audio Engineering Society, 2020.

[119] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 124–134. ACM, 2011.

[120] Savvas Sioutas, Sander Stuijk, Luc Waeijen, Twan Basten, Henk Corporaal, and Lou Somers. Schedule synthesis for halide pipelines through reuse analysis. *ACM Trans. Archit. Code Optim.*, 16(2), apr 2019.

[121] Pitchaya Sitthi-amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Trans. Graph.*, 30:152, 12 2011.

[122] Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Nießner, Gordon Wetzstein, and Michael Zollhöfer. Deepvoxels: Learning persistent 3d feature embeddings. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 2019.

[123] Vincent Sitzmann, Michael Zollhöfer, and Gordon Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations. *CoRR*, abs/1906.01618, 2019.

[124] Stephen So. Why is the sample variance a biased estimator? *Griffith University, Tech. Rep., 09*, 2008.

[125] J.C. Spall. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE Transactions on Automatic Control*, 37(3):332–341, 1992.

[126] László Szirmay-Kalos and Tamás Umenhoffer. Displacement mapping on the gpu—state of the art. In *Computer Graphics Forum*, volume 27, pages 1567–1592. Wiley Online Library, 2008.

[127] Kıvanç Tatar, Matthieu Macret, and Philippe Pasquier. Automatic synthesizer preset generation with presetgen. *Journal of New Music Research*, 45(2):124–144, 2016.

[128] Justus Thies, Michael Zollhöfer, and Matthias Nießner. Deferred neural rendering: Image synthesis using neural textures. *CoRR*, abs/1904.12356, 2019.

[129] Justus Thies, Michael Zollhöfer, Christian Theobalt, Marc Stamminger, and Matthias Nießner. Image-guided neural object rendering. In *International Conference on Learning Representations*, 2020.

[130] Ethan Tseng, Felix Yu, Yuting Yang, Fahim Mannan, Karl St. Arnaud, Derek Nowrouzezahrai, Jean-Francois Lalonde, and Felix Heide. Hyperparameter optimization in black-box image processing using differentiable proxies. *ACM Transactions on Graphics (TOG)*, 38(4), 7 2019.

[131] Ethan Tseng, Yuxuan Zhang, Lars Jebe, Cecilia Zhang, Zhihao Xia, Yifei Fan, Felix Heide, and Jiawen Chen. Neural photo-finishing. *ACM Transactions on Graphics (TOG)*, 41(6), 2022.

[132] Thijs Vogels, Fabrice Rousselle, Brian McWilliams, Gerhard Röthlin, Alex Harvill, David Adler, Mark Meyer, and Jan Novák. Denoising with kernel prediction and asymmetric loss functions. *ACM Transactions on Graphics (TOG)*, 37(4):124, 2018.

[133] Rui Wang, Xianjin Yang, Yazhen Yuan, Wei Chen, Kavita Bala, and Hujun Bao. Automatic shader simplification using surface signal approximation. *ACM Transactions on Graphics (TOG)*, 33(6):1–11, 2014.

[134] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Guilin Liu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. Video-to-video synthesis. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[135] William M Wells. Efficient synthesis of gaussian filters by cascaded uniform filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (2):234–239, 1986.

[136] Maximilian Werhahn, You Xie, Mengyu Chu, and Nils Thuerey. A multi-pass GAN for fluid flow super-resolution. *CoRR*, abs/1906.01689, 2019.

[137] Wikipedia. Entire function, 2017. [Online; accessed 2017-05-20].

[138] Wikipedia. Hermite polynomials — wikipedia, the free encyclopedia, 2017. [Online; accessed 20-May-2017].

[139] Wikipedia contributors. Continuous function — Wikipedia, the free encyclopedia, 2023. [Online; accessed 12-April-2023].

[140] Wikipedia contributors. Minimum cut — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Minimum_cut&oldid=1145437520`, 2023. [Online; accessed 18-April-2023].

[141] Lance Williams. Pyramidal parametrics. In *Acm siggraph computer graphics*, volume 17, pages 1–11. ACM, 1983.

[142] Jamie Wong. Ray marching and signed distance functions, 2016. Accessed: 2019-01-12.

[143] Ho-Hsiang Wu, Prem Seetharaman, Kundan Kumar, and Juan Pablo Bello. Wav2clip: Learning robust audio representations from clip. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2022.

[144] Huikai Wu, Shuai Zheng, Junge Zhang, and Kaiqi Huang. Fast end-to-end trainable guided filter. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1838–1847, 2018.

[145] Zhijun Wu. The effective energy transformation scheme as a special continuation approach to global optimization with application to molecular conformation. *SIAM Journal on Optimization, 1996*, 6(3), 1996.

[146] You Xie, Erik Franz, Mengyu Chu, and Nils Thuerey. tempogan: A temporally coherent, volumetric gan for super-resolution fluid flow. *ACM Transactions on Graphics (TOG)*, 37(4):95, 2018.

[147] Lei Yang, Diego Nehab, Pedro V Sander, Pitchaya Sitthi-amorn, Jason Lawrence, and Hugues Hoppe. Amortized supersampling. In *ACM Transactions on Graphics (TOG)*, volume 28, page 135. ACM, 2009.

[148] Y. Yang and C. Barnes. Approximate program smoothing using mean-variance statistics, with application to procedural shader bandlimiting. *Comput. Graph. Forum*, 37(2):443–454, 2018.

[149] Yuting Yang, Connelly Barnes, Andrew Adams, and Adam Finkelstein. Aδ: Autodiff for discontinuous programs - applied to shaders. In *SIGGRAPH, to appear*, August 2022.

[150] Yuting Yang, Connelly Barnes, and Adam Finkelstein. Learning from shader program traces. In *Eurographics*, April 2022.

[151] Yuting Yang, Zeyu Jin, Connelly Barnes, and Adam Finkelstein. White-box automatic synthesizer programming. In *under review*, 2023.

[152] Lior Yariv, Yoni Kasten, Dror Moran, Meirav Galun, Matan Atzmon, Basri Ronen, and Yaron Lipman. Multiview neural surface reconstruction by disentangling geometry and appearance. *Advances in Neural Information Processing Systems*, 33, 2020.

[153] Matthew John Yee-King, Leon Fedden, and Mark d'Inverno. Automatic programming of vst sound synthesizers using deep networks and other techniques. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(2):150–159, 2018.

[154] Tizian Zeltner, Sébastien Speierer, Iliyan Georgiev, and Wenzel Jakob. Monte carlo estimators for differential light transport. *ACM Trans. Graph.*, 40(4), jul 2021.

[155] Cheng Zhang, Zhao Dong, Michael Doggett, and Shuang Zhao. Antithetic sampling for monte carlo differentiable rendering. *ACM Trans. Graph.*, 40(4):77:1–77:12, 2021.

[156] Cheng Zhang, Bailey Miller, Kai Yan, Ioannis Gkioulekas, and Shuang Zhao. Path-space differentiable rendering. *ACM Trans. Graph.*, 39(4):143:1–143:19, 2020.

[157] Cheng Zhang, Zihan Yu, and Shuang Zhao. Path-space differentiable rendering of participating media. *ACM Trans. Graph.*, 40(4):76:1–76:15, 2021.

[158] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR*, 2018.

[159] Yang Zhou, Lifan Wu, Ravi Ramamoorthi, and Ling-Qi Yan. Vectorization for fast, analytic, and differentiable visibility. *ACM Trans. Graph.*, 40(3), jul 2021.

[160] Matthias Zwicker, Wojciech Jarosz, Jaakko Lehtinen, Bochang Moon, Ravi Ramamoorthi, Fabrice Rousselle, Pradeep Sen, Cyril Soler, and S-E Yoon. Recent advances in adaptive sampling and reconstruction for monte carlo rendering. In *Computer Graphics Forum*, volume 34, pages 667–681. Wiley Online Library, 2015.