# Differentiable Vector Graphics Rasterization for Editing and Learning

TZU-MAO LI, MIT CSAIL
MICHAL LUKÁČ, Adobe Research
MICHAËL GHARBI, Adobe Research
JONATHAN RAGAN-KELLEY, MIT CSAIL

(a) vector sculpting (b) painterly rendering (c) image vectorization (d) seam carving (e) generative modeling

Fig. 1. We introduce a differentiable rasterizer for vector graphics that bridges the raster and vector domains through backpropagation. Differentiable rasterization enables many novel vector graphics applications. (a) Interactive editing that locally optimizes for image-space metrics, such as opacity, under geometric constraints. (b) A new painterly rendering technique by fitting random Bézier curves to a target image. (c) Improving state of art image vectorization result. (d) Editing vector graphics using potentially non-differentiable raster image processing operators, such as seam carving [Avidan and Shamir 2007] for image retargeting. (e) Training a variational autoencoder [Kingma and Welling 2014] to generate vector MNIST digits [LeCun et al. 1998] and adding stylized strokes as postprocessing. Images courtesy of wikipedia user Daderot and Eric Guinther, and freesvg.org user OpenClipart.

We introduce a differentiable rasterizer that bridges the vector graphics and raster image domains, enabling powerful raster-based loss functions, optimization procedures, and machine learning techniques to edit and generate vector content. We observe that vector graphics rasterization is differentiable after pixel prefiltering. Our differentiable rasterizer offers two prefiltering options: an analytical prefiltering technique and a multisampling anti-aliasing technique. The analytical variant is faster but can suffer from artifacts such as conflation. The multisampling variant is still efficient, and can render high-quality images while computing unbiased gradients for each pixel with respect to curve parameters.

We demonstrate that our rasterizer enables new applications, including a vector graphics editor guided by image metrics, a painterly rendering algorithm that fits vector primitives to an image by minimizing a deep perceptual loss function, new vector graphics editing algorithms that exploit well-known image processing methods such as seam carving, and deep generative models that generate vector content from raster-only supervision under a VAE or GAN training objective.

Authors' addresses: Tzu-Mao Li, MIT CSAIL, tzumao@mit.edu; Michal Lukáč, Adobe Research, lukac@adobe.com; Michaël Gharbi, Adobe Research, mgharbi@adobe.com; Jonathan Ragan-Kelley, MIT CSAIL, jrk@csail.mit.edu.


0730-0301/2020/12-ART193
https://doi.org/10.1145/3414685.3417871

CCS Concepts: • **Computing methodologies** → **Rendering**; *Shape modeling*; *Image manipulation*.

Additional Key Words and Phrases: vector graphics, differentiable rendering, image vectorization

**ACM Reference Format:**
Tzu-Mao Li, Michal Lukáč, Michaël Gharbi, and Jonathan Ragan-Kelley. 2020. Differentiable Vector Graphics Rasterization for Editing and Learning. *ACM Trans. Graph.* 39, 6, Article 193 (December 2020), 15 pages. https://doi.org/10.1145/3414685.3417871

## 1 INTRODUCTION

Vector graphics compactly define images using mathematical primitives such as 2D points and curves in a resolution-independent fashion. They are ubiquitous in print, animation, web design, and user interfaces. However, current methods to edit or create vector graphics are arguably trailing behind those designed for raster images. For instance, automatic generation of vector graphics typically requires specialized algorithms that trace image edges and fit curves [Hertzmann 1998; Selinger 2003], and vector editing often calls for specialized solvers tailored to specific geometric primitives [Sun et al. 2007; Yang et al. 2015; Zhao et al. 2018].

In contrast, image processing and machine learning have created powerful and general tools such as convolutional neural networks to manipulate raster images. These methods can extract and transfer image style [Gatys et al. 2016], learn to map images from one domain to another [Isola et al. 2017], or synthesize novel images by learning

from large image collections [Goodfellow et al. 2014; Kingma and Welling 2014].

One could consider a workflow where vector graphics are first rasterized [Batra et al. 2015; Ganacim et al. 2014; Kilgard and Bolz 2012] then post-processed by a raster-domain algorithm. Unfortunately, this is a one-way process that discards the structure implicit in the vector graphic, and the only way back to the vector domain is by re-tracing the raster result, which produces a graphic with a dramatically different structure. This means that it is not possible to truly apply raster-based algorithms to vector graphics this way.

We seek to create a differentiable rasterizer, to bridge the raster and vector domains so that raster-based algorithms and loss functions can be used to modify or synthesize vector content. The common preconception that rasterization is not differentiable has led to the development of differentiable neural approximations to rasterization [Huang et al. 2019; Nakano 2019; Zheng et al. 2019]. Unfortunately, these are often limited to only a few primitives, are costly to train, and generalize poorly beyond their training dataset.

Building on recent work in differentiable 3D rendering [Li et al. 2018], we show that such approximations are unnecessary; when properly accounting for pixel prefiltering (anti-aliasing), the rasterized image *is* differentiable with respect to curve and other vector parameters. Based on this observation, we build a *differentiable* rasterizer that produces correct, high-quality rasterizations of general vector content (in the forward pass) and can automatically compute gradients with respect to the input vector parameters (backward pass).

This crucial issue of anti-aliasing requires special treatment, since automatically differentiating anti-aliasing algorithms does not produce correct or efficient gradient code. We develop two distinct anti-aliasing implementations that each enable differentiable rasterization. The first approach uses Monte Carlo sampling (e.g., [Ganacim et al. 2014; Kilgard and Bolz 2012]) to compute a high-quality *reference* solution that converges to the correct raster image. It has reasonable performance, but requires incoherent memory access, and it is stochastic. The second formulation is slightly faster and deterministic, but can sometimes be less accurate or suffer from artifacts such as *conflation* [Kilgard and Bolz 2012]. It uses an approximate analytical prefilter based on the signed distance to the closest curve (e.g., [Nehab and Hoppe 2008]).

In the Monte Carlo variant, we express a pixel's color as the sum of integrals over object areas and apply the Reynolds transport theorem [Reynolds et al. 1903] — a high-dimensional generalization of the Leibniz integral rule — to differentiate the object boundary parameters by sampling the boundaries. We properly account for discontinuities introduced by point sampling, e.g., when rendering a step edge, which automatic differentiation techniques do not handle. While previous differentiable Monte Carlo rendering methods (e.g., [Li et al. 2018]) focused on triangle meshes, our method works with arbitrary curves and strokes, including Bézier curves and ellipses.

In the analytical prefiltering variant, we can compute gradients by automatic differentiation. However, we must take special care for cubic Bézier curves, since the distance between a point and a cubic curve does not have a closed-form solution, and we need to differentiate an iterative polynomial root solver. We use the implicit function theorem for efficient differentiation of the iterative solver.

Both anti-aliasing strategies are embarrassingly parallel. We provide our CUDA implementation together with a PyTorch interface that enables our differentiable rasterizer to be used within larger differentiable programs.

Differentiable rasterization enables gradient-based optimization across the vector and raster domains. With this, we cast a wide range of novel vector graphics problems into a unified optimization framework: minimizing a rendering loss on the raster output, either by directly updating vector graphics parameters, or by optimizing a neural network that generates vector graphics (Figure 1). We develop a new interactive vector graphics editing system that can simultaneously optimize geometric and raster metrics, improve existing image vectorization methods by fine-tuning the results to better match the target image, apply raster image processing filters such as seam carving [Avidan and Shamir 2007] to vector graphics, and train generative models for vector graphics — both variational autoencoders and generative adversarial networks — which are supervised only with raster images.

Our work focuses on differentiating the rasterization procedure with respect to continuous parameters of the vector shapes (vertex positions, colors, stroke width, etc). Discrete changes to the *topology* of vector graphics are important , but they remain beyond the scope of this paper.

To summarize, we make the following contributions:

- We identify the key criteria for a general-purpose differentiable rasterization algorithm for vector graphics, including polynomial and rational polynomial curves, transparency, occlusion, constant and gradient fill, and strokes.
- We show how traditional rasterization algorithms are not directly differentiable in this new setting. We propose a differentiable anti-aliasing algorithm using multisampling. We resolve the discontinuities by explicitly sampling the shape boundaries and deriving the gradients for curves using the Reynolds transport theorem.
- We demonstrate an alternative approximate, deterministic solution through analytical prefiltering that is slightly faster at the cost of artifacts such as conflation.
- We unify and formulate a wide range of vector graphics editing and learning tasks as gradient-based optimization problems where a vector graphic is made to match a target image via a loss on its rasterization (Figure 1). We show that the rasterizer can be used in complex and practical optimization and learning tasks, including vector graphics editing, image vectorization, and learning to synthesize vector graphics.

We open-source our code at https://github.com/BachiLi/diffvg.

## 2 RELATED WORK

### 2.1 Vector graphics creation and editing

We briefly review previous work on creating and editing vector graphics. Due to the lack of a general differentiable rasterizer, previous techniques often either focused on the *geometry* of the vector graphics, or converted the raster images to a convenient geometric

representation. Some specialized methods for fitting vector graphics to raster images exist, but they do not handle general vector graphics, which include occlusion and transparency.

*Image vectorization methods.* Most methods that generate vector graphics from raster images first segment images into regions, and fit Bézier curves [Lecot and Levy 2006; Selinger 2003; Xia et al. 2009], diffusion curves [Orzan et al. 2008; Xie et al. 2014] and gradient meshes [Sun et al. 2007] to the region boundaries. These techniques provide a good initial guess for image vectorization. We show that we can improve the results by optimizing the continuous parameters of the generated vector graphics to minimize a raster-space rendering loss using our differentiable rasterizers (§ 6.3).

Techniques based on differentiating rasterization exist for diffusion curves [Zhao et al. 2018], gradient meshes [Sun et al. 2007], and closed Bézier curves [Yang et al. 2015]. Compared to these methods, we significantly extend the range of vector graphics that can be differentiated through the rasterizer, including occlusion, transparency, rational polynomials, and strokes.

*Vector graphics editing.* Research on user interfaces for vector graphics dates back to Sketch Pad [Sutherland 1964], which allowed users to impose geometric constraints on the vector graphics when editing them. Briar [Gleicher 1992] and Lillicon [Bernstein and Li 2015] explored similar ideas but extended the editors to address different constraints and editing. Our interactive editor (§ 6.2) builds on top of this work, and allows the user to put constraints on the rendering in addition to the geometry, increasing the expressivity of the editor.

*Generating vector graphics using neural networks.* Deep learning models have been proposed for vector graphics. Ha and Eck [2018] and Lopes et al. [2019] trained generative recurrent models using a sketch database. Azadi et al. [2018] and Yue et al. [2019] generate new fonts given a few example styles.

Deep learning has also been used for solving image vectorization. Ellis et al. [2018] combined program synthesis and recurrent neural networks to infer a graphics program that generates a target sketch image. Ganin et al. [2018] instead trained a model-free reinforcement learning agent on natural images using a drawing simulator, combined with an adversarial loss.

*Neural approximations of rasterization.* Some recent work uses neural networks to smoothly approximate the rasterization operation [Huang et al. 2019; Nakano 2019; Zheng et al. 2019]. These neural approximations usually assume a specific vector graphics configuration (e.g., a single constant color quadratic stroke) and a fixed resolution. These techniques were necessary because rasterization was considered to be non-differentiable. We show that with careful derivation, it *can* be made differentiable, and the approximation is no longer required.

Differentiable rasterization allows us to use a general *rendering loss* and propagate image gradients to the vector representation of such models, in contrast to the task-specific position based losses or model-free reinforcement learning approaches used in previous work. This allows us, for example, to train vector graphics-generating variational autoencoders and generative adversarial networks directly on raster images (§ 6.3).

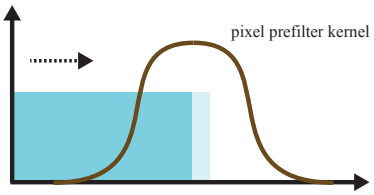



Fig. 2. Pixel prefiltering using a convolution makes a discontinuous function smooth. We illustrate this using a box function in 1D. To rasterize this discontinuous box function with anti-aliasing, we convolve the box with a kernel centered at the pixel center. Moving the box causes a continuous change of the area under the filtering kernel, and therefore, of the anti-aliased signal.

## 2.2 Vector graphics rasterization

Previous work has studied efficient algorithms and data structures to rasterize closed curves and strokes. Much of it focused on anti-aliasing (e.g., [Duff 1989; Fabris and Forrest 1997; Manson and Schaefer 2013]). Recent work explored the efficient parallelization of vector graphics rasterization on graphics hardware (e.g., [Batra et al. 2015; Kilgard and Bolz 2012; Kokojima et al. 2006; Li et al. 2016; Loop and Blinn 2005]).

Nehab and Hoppe [2008] and Ganacim et al. [2014] proposed data structures to efficiently compute the shaded color given an arbitrary point on the image. Our boundary sampling procedure (§ 5) requires the same random access pattern and can therefore benefit from these data structures.

## 2.3 Differentiable rendering

Differentiating 3D rendering has drawn attention in both computer graphics and vision research recently (e.g., [de La Gorce et al. 2011; Kato et al. 2018; Liu et al. 2018, 2019; Loper and Black 2014]). Our gradient derivation is inspired by the work of Li et al. [2018], but it extends it using the Reynolds transport theorem [Reynolds et al. 1903] to simplify the derivation and generalize to curves and strokes.

Reynolds's transport theorem has also been shown to be useful for differentiable rendering in Li's thesis [2019] and the recent work of Zhang et al. [2019] on differentiating volume rendering. We focus our derivation on 2D vector graphics, which significantly simplifies the theory.

# 3 OVERVIEW

We want to bridge the gap between vector and raster graphics, by differentiating through the vector rasterization process and enabling gradient-based optimization.

The key insight that makes this possible is that, while vector shapes — represented mathematically as indicator functions — are not directly differentiable, applying anti-aliasing smoothes out the discontinuities. This makes the anti-aliased rasterized image differentiable with respect to the vector shape parameters (Figure 2).

Special care is needed when differentiating through rasterization as different anti-aliasing strategies call for different gradient implementations. Depending on the anti-aliasing technique we choose, we evaluate the gradient with respect to shape parameters either by

Fig. 3. Our method supports vector graphics formed by polynomial and rational polynomial curves, including polygons, quadratic and cubic Bézier curves, circles and ellipses, with colored strokes, fills, and gradients.

integrating the shape boundaries to account for the discontinuities, or by analytically integrating the integrand, which requires making simplifying assumptions.

Our differentiable rasterizer is useful both for direct optimization of vector graphics, and to train neural networks that generate vector graphics. It supports polynomial and rational curves, stroking, transparency, occlusion, and gradient fills (Figure 3).

In the following, first, we present a vector graphics rendering model that is suitable for differentiable rasterization. We then discuss how we differentiate through this model. Finally, we demonstrate several applications enabled by our differentiable rasterizer.

## 4 RENDERING MODEL AND DESIGN CHOICES

Our goal is to have an accurate, high-quality solution that covers as many common vector graphics representations as possible, while being fully differentiable. Differentiation and optimization introduce unique challenges to rendering models:

- We show that prefiltering, or anti-aliasing can make rasterization differentiable. However, care has to be taken when differentiating the anti-aliasing algorithm.
- We *should* avoid methods based on converting curves into polylines or converting filled curves into polygons or triangle meshes, since such conversion is *not* differentiable — a small change to the curve parameters can cause a topological change on the generated polylines or meshes.
- We need to be particularly careful about introducing approximations to the rendering model, as all the intermediate shapes need to be rendered accurately during iterative optimization.

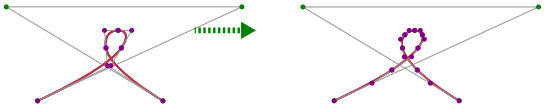
We first describe the vector graphics primitives we support. Then we discuss how vector graphics are rasterized and introduce the mathematical model of pixel prefiltering, which is crucial to differentiation. In Section 5, we present and analyze two differentiable rasterization algorithms that solve the pixel prefiltering.

### 4.1 Vector primitives

Our primitives are based on the SVG (Scalable Vector Graphics) standard. We support SVG *paths* (with linear, quadratic or cubic segments[1]), ellipses, circles, and rectangles, but any curve with a parametric form is compatible with our method. The curves can be either open or closed (Figure 3).

Each curve can have a *fill color* and/or a *stroke color* (fill color has no effect on open curves). Our implementation assumes round caps for the strokes, but adding new cap styles is straightforward. Colors

[1]Our current implementation converts arcs to cubic curves [Cridge 2015], but extending our method to handle arcs is possible.



(a) quadratic approximation of a cubic curve

(b) a control point moved → more subdivisions

Fig. 4. Some rasterizers approximate cubic curves with quadratic curves by recursively splitting the curves until quadratic curves can accurately approximate the subdivided cubic curves. This enables efficient computation for stroke rasterization. Unfortunately, this is not suitable in differentiable rasterization, since moving the control points could generate more or less quadratic curves, making the operation not differentiable. Here we demonstrate a simple curve splitting scheme guided by an error bound [Colomitchi 2006]. Moving the top-right control point to the right would generate extra quadratic curves.

can be solid, linear gradient, or radial gradient, and contain RGB and alpha channels.

### 4.2 Vector graphics rasterization

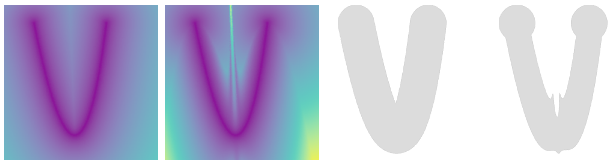
To understand the differentiation of vector graphics, we first briefly review how rasterization is typically implemented [Nehab and Hoppe 2008] and how differentiable rasterization demands certain forward rendering models. A raster image is a 2D grid sampling over the space of the vector graphics scene $f(x, y; \Theta)$, where $\Theta$ contains the curve parameters. We first define the scene function $f$, then we will describe how to use $f$ to compute a color for each pixel.

Given a 2D location $(x, y) \in \mathbb{R}^2$, we first find all the filled curves and strokes that overlap with the location. We then sort them with a user-specified order and compute the color using alpha blending [Porter and Duff 1984].

To find the curves overlap with the 2D location, for each filled curve, we compute the *winding number* at the location by tracing a ray to the right of $(x, y)$ and count the number the intersections. We determine whether the point is inside the curve based on the *fill rule*: the even-odd fill rule defines all points with an odd winding number as inside, and the non-zero fill rule includes all points with a non-zero winding number. For each stroke, we compute the distance between the point and the closest point on the curve. If it is smaller than half the stroke width, the point is inside the curve stroke.

Intersecting a ray with a polynomial curve with degree $N$ for winding number computation boils down to solving the roots of a polynomial with degree $N$. On the other hand, computing the closest distance between a point and the polynomial curve requires solving the roots of a polynomial $p(t)$ with degree $2N - 1$, since it amounts to solving $\arg\min_t (p(t) - q)^2$ where $q = (x, y)$.

For cubic curve strokes, which are ubiquitous in practice, we need to solve a 5th order polynomial that does not admit a closed-form solution. Most rasterizers resolve this by either approximating cubic curves using quadratic curves by splitting the curves using the de Casteljau algorithm [Batra et al. 2015; Kilgard and Bolz 2012], or converting the stroke boundary into a filled curve [Ganacim et al. 2014; Kilgard 2020; Nehab 2020]. Some approaches approximate the



(a) reference distance (ours) (b) approximate distance [Nehab and Hoppe 2008] (c) reference stroke (ours) (d) approximate stroke [Nehab and Hoppe 2008]

Fig. 5. A popular technique approximates the distance between a point and a polynomial curve by converting the curve into an implicit function, and approximates the neighborhood using a Taylor expansion [Loop and Blinn 2005; Nehab and Hoppe 2008]. While this method produces accurate distance in the proximity of the curve and is very efficient, it is not suitable for differentiable rasterization, since we need to process strokes with arbitrary width. Here we compare the approximate distance of quadratic curves proposed by Nehab and Hoppe [2008] with the actual distance. The approximation shows clear artifacts when the stroke radius is large.

distance by converting the strokes into an implicit function [Loop and Blinn 2005; Nehab and Hoppe 2008].

None of these approaches are suitable for differentiable rasterization. Adaptive curve subdivision is not differentiable, since a small parameter change on the cubic curve can cause the subdivision procedure to generate more or fewer quadratic segments (Figure 4). Similarly, the *parallel curve* of a cubic stroke is a 10th order polynomial [Farouki and Neff 1990] and requires even more subdivisions. The distance approximation is also not preferable, since they are only accurate in the proximity of the curves. During optimization, we could generate curves with large stroke width that breaks these approximations (Figure 5).

Instead, we directly solve the polynomial using bisection and the Newton-Raphson method [Press et al. 2007]. The initial guess of the iterative solver is obtained from isolator polynomials [Sederberg and Chang 1994] – the real roots of a 5th order polynomial equation can be isolated by the roots of a linear equation and a 3rd order polynomial. Appendix A details the implementation. Efficiently differentiating the iterative solver is not trivial. We discuss the differentiation in Section 5.2.

Similarly, the distance between a point and an ellipse also requires iterative root solving, where a robust solution is detailedly discussed by Eberly [2011]. Our implementation does not support ellipse strokes yet, but the way we handle cubic curve strokes can be used for ellipse strokes .

### 4.3 Pixel prefiltering and anti-aliasing

The scene function $f$ is not differentiable with respect to curve parameters, due to the inside-outside test. However, we show that common anti-aliasing techniques make the pixel color differentiable.

$f$ is not band-limited due to discontinuities, so rasterization, a 2D sampling over $f$, is prone to *aliasing*. To avoid aliasing, instead of computing $f$ at a single location for each pixel, we can *prefilter* the scene $f$ over a convolution kernel $k$ with support $A$ to make it band-limited:

$$I(x,y) = \iint_A k(u,v) f(x-u, y-v; \Theta) du dv. \tag{1}$$

We can then sample $I(x,y)$ at the pixel location to compute an alias-free image.

For differentiable rasterization we are interested in $\frac{\partial I}{\partial \Theta}$. Importantly, the integration over the filter support means we only care about the *average color* of a pixel, instead of the point evaluation at the center. The curve movements lead to a continuous change of the average color, making $I(x,y)$ differentiable (Figure 2).

Even though the pixel integral is differentiable, evaluating the derivatives of the integral requires extra care. In the next section, we propose two strategies for computing the pixel integrals $I$ and its derivatives $\frac{\partial I}{\partial \Theta}$. We then discuss the advantages and disadvantages of the two.

## 5 DIFFERENTIABLE RASTERIZATION

The pixel integral (Equation 1) does not have a closed-form solution in general. Inspired by previous approaches, we propose two ways to solve the pixel integral. The first strategy computes the integral using Monte Carlo integration. It generates a high-quality reference solution and has reasonable performance. The second strategy approximates the integral by assuming the scene function $f$ to be a simplified configuration such that the integral has an analytical solution. It has the benefit of being deterministic and slightly faster, at the cost of being an approximate solution, which sometimes manifests as artifacts on the raster image.

### 5.1 Monte Carlo sampling

To evaluate the pixel integral and its derivatives, our first approach is to discretize the pixel integral using Monte Carlo sampling:

$$\begin{aligned} I(x,y) &= \iint_A k(u,v) f(x-u, y-v; \Theta) du dv \\ &\approx \frac{1}{N} \sum_i^N k(u_i, v_i) f(x-u_i, y-v_i; \Theta), \end{aligned} \tag{2}$$

where the samples $u_i, v_i$ can be either points on a uniform grid, independent samples, stratified samples, or blue noise samples [Banterle et al. 2012; Cook 1986; Dippé and Wold 1985].

Unfortunately, because of geometric discontinuities, $f$ is not differentiable with respect to most scene parameters, so we cannot exchange the integral and differential operator to express $\frac{\partial I}{\partial \Theta}$ as an integral that could be similarly discretized.

To resolve this, we take inspiration from 3D differentiable rendering techniques [Li et al. 2018] and rework the integral by removing the derivative operator outside of the integral, and discretizing the reworked integrals. While previous 3D differentiable rendering methods focused on triangle meshes and 3D radiative transfer equations [Li et al. 2018; Zhang et al. 2019], we present a simpler derivation tailored to 2D vector graphics that work with curved boundaries.

Our goal is to compute the gradients of $I$ with respect to the parameters $\Theta$:

$$\begin{aligned}\frac{\partial I(x,y)}{\partial \Theta} &= \frac{\partial}{\partial \Theta}\iint_A k(u,v) f(x-u, y-v; \Theta) du dv \\ &= \frac{\partial}{\partial \Theta}\iint_A g(u,v) du dv,\end{aligned} \tag{3}$$

where we use $g(u,v)$ to represent the multiplication of the scene function $f$ and kernel $k$ for brevity. Without loss of generality, we assume the kernel $k$ is continuous.

We partition the domain $A$ into multiple sub-regions $A_i$ such that all the discontinuities of $g$ are at the integration boundaries (Figure 6a):

$$\frac{\partial}{\partial \Theta}\iint g(u,v) du dv = \sum_i \frac{\partial}{\partial \Theta}\iint_{A_i(\Theta)} g(u,v) du dv. \tag{4}$$

The partition is done for the mathematical derivation only, and we do not explicitly clip the curves to partition the space.

To simplify the problem, let us consider the case in 1D of an individual integral. We can rewrite the integral when the boundaries depend on the parameter $\Theta$:

$$\frac{\partial}{\partial \Theta}\int_0^{b(\Theta)} g(u) du = \int_0^{b(\Theta)} \frac{\partial g(u)}{\partial \Theta} du + g\left(b\left(\Theta\right)\right) \frac{\partial b(\Theta)}{\partial \Theta}. \tag{5}$$

This equation is often called the Leibniz's integral rule. The first integral is responsible for the differentiation of color and transparency, while the second term is responsible for the change of the boundaries. We can estimate the derivative integral using Monte Carlo sampling, and add the second correction term to take boundary change into account.

The same idea can be generalized to 2D using Reynolds transport theorem [Reynolds et al. 1903], whose definition is precisely the following equation:

$$\begin{aligned}&\frac{\partial}{\partial \Theta}\iint_{A_i(\Theta)} g(u,v) du dv = \\ &\iint_{A_i(\Theta)} \frac{\partial}{\partial \Theta} g(u,v) du dv + \int_{\partial A_i(\Theta)} \left(\frac{\partial p(t)}{\partial \Theta} \cdot n(t)\right) g(p(t)) dt,\end{aligned} \tag{6}$$

where $\partial A_i$ is the boundary of area $A_i$, $n(t)$ is the outward-pointing normal of the boundary, and $p(t)$ is a 2D vector representing the points on the boundary (e.g., for a quadratic segment of a filled shape, $p(t) = (1-t)^2 p_0 + 2(1-t)t p_1 + t^2 p_2$ and $n(t)$ is the normalized 2D vector perpendicular to the tangent $p'(t)$). We refer to readers to Flanders's article [1973] for a concise proof of Reynolds' theorem.

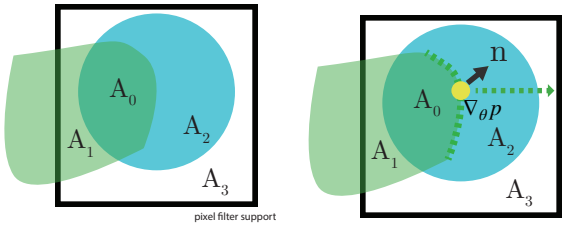
Compared to the 1D case, the boundary correction term now becomes a 1D curve over the integration domain boundary $\partial A_i$. It measures the expansion speed of the boundary with respect to the differentiating parameter $\Theta$. Figure 6b illustrates the intuition. Our method correctly handles complex occlusions between multiple primitives, since for the vector graphics primitives we consider (Section 4.1), all discontinuities happen on the primitive boundaries. Reynolds transport theorem handles the boundary changes by integrating over them.

To estimate Equation 6 over all shapes, we separately apply two Monte Carlo estimators for the two integrals. Estimating the first integral is similar to standard vector graphics rendering, but instead of outputting the color, we accumulate the gradients to the corresponding color parameters. To estimate the second boundary integral, we allocate $N$ samples for each pixel (we usually set $N = 4$). We first randomly pick a curve and a point on it. If the curve is part of a stroke, we randomly offset the point on one side of the normal by the half of the stroke width, or we randomly sample one of the caps of the stroke. Notice that each point on the boundary $p(t)$ is associated with two sub-regions $A_i$. We compute the scene function $f$ at the two sides of the sampled point, and propagate the gradients to the corresponding parameter $\Theta$:

$$\begin{aligned}&\sum_i \int_{\partial A_i(\Theta)} \left(\nabla_\Theta p \cdot n\right) f\left(x, y; \Theta\right) dx dy \approx \\ &\frac{1}{N}\sum_j \frac{\left(\nabla_\Theta p_j \cdot n_j\right)\left(f(p_j + \epsilon n_j) - f(p_j - \epsilon n_j)\right)}{P(p_j|c)P(c)},\end{aligned} \tag{7}$$

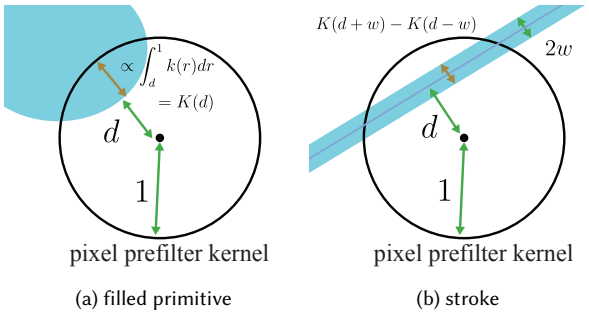
where $\epsilon$ is a small number (we use $10^{-4}$ pixels for all renderings), and $P(p_j|c)P(c)$ is the probability density of picking the curve $c$ and point $p_j$ on the curve or stroke boundaries. We sample on the curve using the parametric form. For a curve $p(t)$, the probability density can be computed by the inverse Jacobian of the transformation $(P(p_j|c) = \left\|\frac{\partial p(t)}{\partial t}\right\|^{-1})$.

The Monte Carlo estimation handles the occlusion correctly. If we sample a point on a curve that is occluded, the point will land on a continuous region. Thus $\lim_{\epsilon\to 0} f(p_j + \epsilon n_j) - f(p_j - \epsilon n_j) = 0$ due to the definition of continuity, and the sample's contribution is 0.




(a) partition (b) sample boundary

Fig. 6. We can handle the discontinuities, occlusion, and transparency with multiple shapes correctly using Monte Carlo sampling when differentiating vector graphics rendering. To achieve this, (a) we first partition the pixel filter support into disjoint regions such that the discontinuities are at the boundaries of the disjoint regions $A_i$. (b) Assuming we want to take the derivative of the green shape moving right ($\nabla_\theta p$), we integrate over the change at the boundaries (the dashed green line), to take the increase of the green area and the decrease of the blue area into account. The infinitesimal change at the yellow point $p$ is defined by the dot product of the normal $n$ and the x-axis. We derive the gradients through the Reynolds transport theorem [Reynolds et al. 1903] and generalize previous differentiable rendering approaches [Li et al. 2018] to handle shapes with arbitrary boundaries.




(a) filled primitive (b) stroke

Fig. 7. Our differentiable analytical prefiltering method is based on previous approaches that approximate radial filtering using the (signed) distance $d$ to the curve boundaries (e.g., [Gupta and Sproull 1981; Nehab and Hoppe 2008; Turkowski 1982]). We approximate the 2D integral using an 1D integral on the line formed by pixel center and the closest point on the curve boundary. (a) For filled primitives, we approximate by analytically integrating a filter kernel $k$ over the line segment inside the primitive. (b). For strokes, we approximate by taking a line segment whose length is the stroke width $w$

## 5.2 Analytical prefiltering

Our second, alternative, approach to differentiating rasterization is to approximate the pixel integral by simplifying the vector graphics configuration. We adapt a common approach that makes use of signed distance field. We show that efficiently differentiating through the signed distance field requires extra care.

We are interested in methods that can approximate general vector graphics without the need to convert the original curves into another form. Our method is based on approaches that approximate the 2D anti-aliasing integral using an 1D integral based on the signed distance between pixel center and the closest point on the curve [Fabris and Forrest 1997; Gupta and Sproull 1981; Loop and Blinn 2005; Nehab and Hoppe 2008; Turkowski 1982]. These methods can be easily applied to general vector graphics as long as we can compute the distance between the point and the curve.

*Analytical approximation.* We approximate the anti-aliasing integral by analytically integrating a radially symmetric filter kernel $k(r)$ with support 1 ($k(r) = 0$ for $|r| \geq 1$) along a line, formed by the pixel center and the closest point on the curve boundary. This analytical approximation $K(d)$ would represent the *coverage* of the shape inside the pixel filter. We also want our analytical coverage approximation $K(d)$ to satisfy the normalization constraints: $K(1) = 0, K(-1) = 1$. Therefore we define $K(d) = C_0 + C_1 \int_d^1 k(r)dr$, where the constants $C_0$ and $C_1$ can be solved by solving the normalization constraint.

For filled curves we approximate the color at $(x, y)$ as

$$K\left(d_i(x,y)\right) f_i(x,y) = \alpha_i(d_i(x,y)) f_i(x,y), \tag{8}$$

and for strokes we approximate the color as

$$\begin{aligned}&\left(K(|d_i(x,y)| + w) - K\left(|d_i(x,y)| - w\right)\right) f_i(x,y) = \\ &\alpha_i(d_i(x,y), w) f_i(x,y),\end{aligned} \tag{9}$$

where $d_i(x, y)$ is the closest signed distance between the pixel center and the curve $i$ (negative $d_i$ means $(x, y)$ is inside of a closed curve), $w$ is the stroke half-width, and $f_i$ is a differentiable function that represents the color assigned to the curve $i$ (constant, linear or radial gradient), evaluated at $(x, y)$. We then loop over all curves in a user-specified order for alpha blending. The alpha, or the opacity, for each shape is multipled by $\alpha_i$ to approximate occlusion between shapes. Following the suggestion of Nehab and Hoppe, we use a parabolic kernel $k(r) = \frac{4}{3}(1 - r^2)$ and $K(r) = \frac{1}{2} + \frac{1}{4}\left(r^3 - 3r\right)$.

*Differentiation.* The prefiltering contribution (Equations 8 and 9), along with the alpha blending makes rasterization differentiable with respect to the curve parameters (notice the dependency to the signed distance $d$, which in turn depends on the curve parameters). However, the computation of the signed distance $d$ to a cubic curve requires solving a 5-th order polynomial equation (Section 4.2).

Differentiating the iterative root solver using the standard reverse-mode automatic differentiation [Griewank and Walther 2008] requires memorizing all intermediate steps, which is inefficient. Instead, we apply the implicit function theorem [Rio Branco de Oliveira 2012]. For a $n$th-order polynomial $p(t; \mathbf{c}) = 0$ with a variable $t \in \mathbb{R}$ and coefficients $\mathbf{c} \in \mathbb{R}^n$, we want to know how changing the coefficients $\mathbf{c}$ can change the root $t$, i.e. $\frac{\partial t}{\partial \mathbf{c}}$. The implicit function theorem gives us:

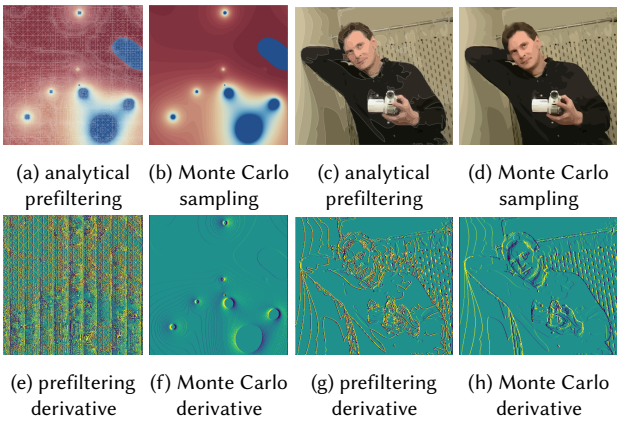
$$\frac{\partial t}{\partial \mathbf{c}} = -\frac{1}{\frac{\partial p(t,\mathbf{c})}{\partial t}} \frac{\partial p(t,\mathbf{c})}{\partial c}. \tag{10}$$

Therefore, we only need the root $t$ in order to calculate the derivatives, where the root can be found by the iterative solver. This is a known technique in automatic differentiation literatures and has been applied in bi-level optimization [Bell and Burke 2008]. Once we have the derivative $\frac{\partial t}{\partial \mathbf{c}}$, we can differentiate the prefiltering contribution using the chain rule: $\frac{\partial \hat{k}}{\partial \mathbf{c}} = \frac{\partial \hat{k}}{\partial d} \frac{\partial d}{\partial t} \frac{\partial t}{\partial \mathbf{c}}$.

*Remarks.* Some works noticed the relation between the pixel integral and the curve boundaries using Green's theorem [Catmull 1978; Duff 1989; Manson and Schaefer 2013] and derived analytical solutions by integrating over curve boundaries. This is still an approximation due to several reasons. The boundary of a polynomial curve stroke is not necessarily a closed polynomial curve (the *parallel curve* of a cubic curve is a 10-th order polynomial, but the cap at the stroke endpoints make it not a polynomial). Shading with linear gradients that have multiple stops, radial gradients, or even textures complicates the analytical solution. Finally, all these methods require clipping a polynomial or rational polynomial curve against each other and the pixel filter support, which makes their implementation and the differentiation complex.

## 5.3 Discussion

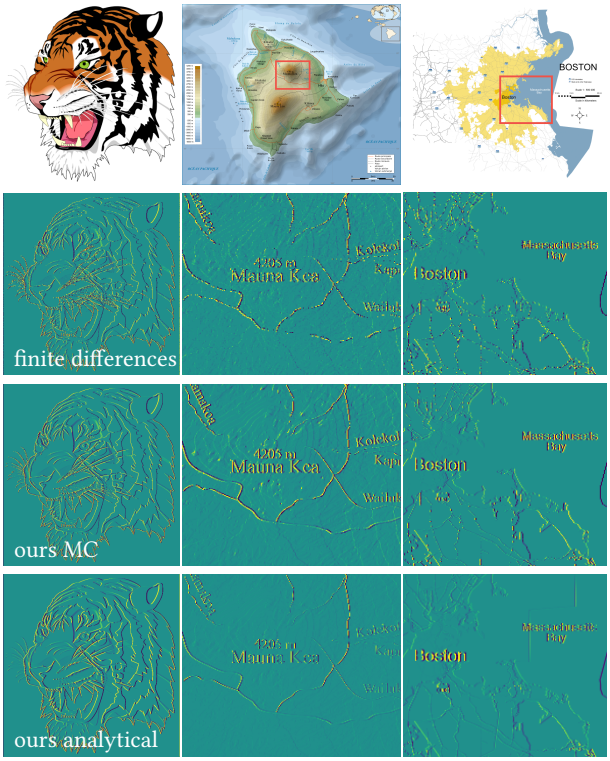
Here we compare the two anti-aliasing strategies qualitatively. Overall, the Monte Carlo sampling produces higher-quality images since it does not make assumptions on the scene configuration. The analytical prefiltering approach is prone to the well-known *conflation* artifacts when parallel edges exist (Figure 8). Modern vector graphics rasterization engines running on GPUs mostly rely on multisampling [Batra et al. 2015; Ganacim et al. 2014; Kilgard and Bolz 2012] to avoid conflation artifacts. We discuss more about the differentiability of parallel edges in Appendix B.



(a) analytical prefiltering (b) Monte Carlo sampling (c) analytical prefiltering (d) Monte Carlo sampling

(e) prefiltering derivative (f) Monte Carlo derivative (g) prefiltering derivative (h) Monte Carlo derivative

Fig. 8. Rendering with analytical prefiltering provides a way to make the rendering process differentiable, but produces undesirable artifacts. The top row shows scenes with shared edges between primitives. The bottom row shows the derivative for each pixel color with respect to the whole scene moving right. Analytical prefiltering methods usually have to make numerous assumptions about the scene (here we implemented Nehab and Hoppe's method [2008] based on analytical convolutions with half-planes). This results in the notorious *conflation* artifacts. In contrast, our Monte Carlo sampling generates pleasing results. The vector graphics are taken from the benchmark used by Ganacim et al. [2014] and Kilgard and Bolz [2012].




(a) tiger (b) Hawaii (c) Boston

Fig. 9. We compare the gradients of finite differences over Monte Carlo sampling, our Monte Carlo boundary sampling, and our analytical prefiltering approach. The figure visualizes the derivative of pixel color with respect to all primitives moving right, using benchmark images from Ganacim et al. [2014] that contain thousands of segments to tens of thousands of segments with complex occlusion. Our boundary sampling matches the finite differences reference closely, while the analytical prefiltering gradient is largely the same, but slightly different due to the different image formulation model. For the Boston scene (c), the analytical prefiltering generates conflation artifacts due to the parallel edges in the scene. We used 16 samples per pixel were used to compute the Monte Carlo sampling, and 1 sample per pixel for the analytical prefiltering. Typically when doing gradient-based optimization we use 4 samples per pixel for Monte Carlo sampling. Zoom in to compare the images in detail.

On the other hand, our boundary sampling (Section 5.1) creates incoherent memory access because the evaluation location is decided at random. This has several consequences: Firstly, this makes the gradient computation *stochastic*, and can have impact to the convergence rate of optimization. Secondly, we need to be able to query the scene function $f$ in a random-access manner [Nehab and Hoppe 2008], which requires efficient data structures for sorting the curve geometry. Thirdly, the incoherent memory access pattern makes naive implementation less efficient on modern GPU due to warp divergence. Our current implementation sorts the evaluation locations by their Z-order to create coherent access, however for large images the cost of sorting is non negligible. Finally, the boundary sampling complicates selective evaluation of the pixel colors: consider a pixel-wise loss function, we can stochastically evaluate the loss function and gradients by randomly choosing the pixel locations we want to evaluate, and this significantly speeds up the gradient descent procedure [Azinović et al. 2019]. Unfortunately, boundary sampling makes selective evaluation difficult: we need to sample the intersection of the boundaries and the selected pixels' filter support area.

In practice, we prefer the Monte Carlo formulation for our applications because it is less prone to artifacts and behaves closer to modern GPU rasterizers. Unless otherwise specified, for most of our applications we opted for the Monte Carlo sampling strategy. The analytical prefiltering approach can be used when we want determinism in gradient computation, or when it is desirable to mimic the behavior of an existing non-differentiable rasterizer (notably the WebKit browser engine [2]). The analytical prefiltering also offers the capability to evaluate samples on a random access basis without rendering the whole image.

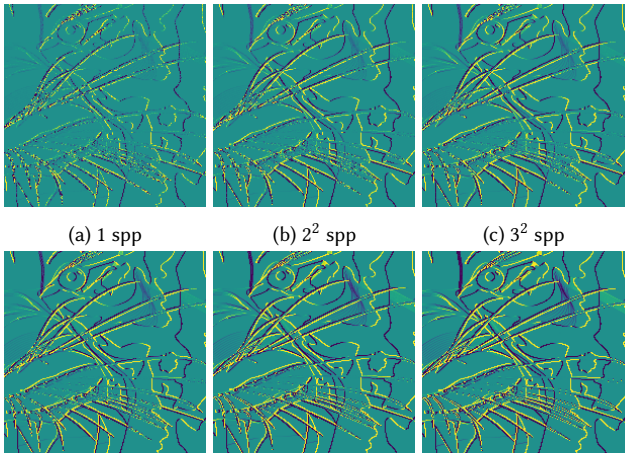
[2]https://webkit.org/

## 6 APPLICATIONS AND EVALUATION

We implement our rasterizer in C++/CUDA with a PyTorch interface [Paszke et al. 2019]. We use a bounding volume hierarchy with axis-aligned bounding boxes of curve segments to accelerate the winding number and distance queries. For the boundary sampling, we sort the samples according to Z-order before evaluating the contribution of each sample to decrease thread divergence. The bounding volume hierarchy is built on CPU using a single thread. We sort the bounding volumes of the primitives by the 2D Morton codes of the box center [Lauterbach et al. 2009]. For the bounding

Table 1. We measure the performance of our implementation of our differentiable rasterizer using a set of standard benchmarks [Ganacim et al. 2014; Kilgard and Bolz 2012]. We denote the time spent on the forward pass as *fwd*, which includes the preprocessing time to copy the curve data from PyTorch, build the acceleration structure, and render an image. We denote the time spent on the backward pass for computing gradient as *rev*. The preprocessing time is typically negligible, except for *contour*, which takes around 0.15 seconds. Monte Carlo sampling is slightly more costly due to the need to sample multiple times per pixel. Analytical prefiltering is slightly faster, while each sample is more expensive due to the requirement of the exact signed distance calculation.

| | *tiger* (495 × 510)<br>2532 curves | *Boston* (761 × 682)<br>27945 curves | *contour* (692 × 692)<br>188336 curves | *Hawaii* (1555 × 1281)<br>52946 curves | *mcseem2* (684 × 594)<br>10224 curves |
|---|---|---|---|---|---|
| Monte Carlo sampling (2x2) | fwd:0.060s/rev:0.222s | fwd:0.074/rev:0.266s | fwd:0.213s/rev:0.176s | fwd:1.117s/rev:3.371s | fwd:0.059s/rev:0.162s |
| Monte Carlo sampling (4x4) | fwd:0.271s/rev:0.598s | fwd:0.197s/rev:0.862s | fwd:0.293s/rev:0.472s | fwd:3.235s/rev:11.173s | fwd:0.129s/rev:0.437s |
| Analytical prefiltering (1x1) | fwd:0.098s/rev:0.0992s | fwd:0.413s/rev:0.402s | fwd:0.195s/rev:0.044s | fwd:1.287s/rev:1.267s | fwd:0.050s/rev:0.049s |



(a) 1 spp (b) $2^2$ spp (c) $3^2$ spp

(d) $4^2$ spp (e) $8^2$ spp (f) $16^2$ spp

Fig. 10. We study the variance of our Monte Carlo sampling using the tiger scene in Fig. 9. As in the usual Monte Carlo integration, the variance of our image decreases linearly as the number of the samples grow. We typically use $2^2$ samples per pixel (spp), which works well for various tasks ranging from optimizing a complex vector graphic to training a neural network.

volumes of the segments within a path, we sort them by the y-axis of the box center.

We show timing analysis of our methods in Table 1. In general we achieve interactive performance for vector graphics with moderate complexity (such as the *tiger* scene in Figure 9). As we will show in Section 6.2, we can use our rasterizer inside an interactive vector graphics editor. The analytical prefiltering approach is slightly faster than the Monte Carlo sampling, since it only requires one number of samples per pixel.

We believe performance optimization [Levien 2020], and incorporating more advanced acceleration data structures such as shortcut trees [Ganacim et al. 2014], could significantly speed up the rendering, but the current performance is already sufficient for nontrivial applications.

## 6.1 Evaluation

*Gradient comparison.* We compare the gradients generated by both of our methods and central finite differences on various vector graphics scenes ranging from thousands of segments to tens of thousands of segments. The results are shown in Figure 9 and the supplementary material. The boundary sampling can handle complex occlusion occurred in the complex vector graphics and generates results close to finite differences. On the other hand, apart from the conflation artifact case in Figure 8, the analytical prefiltering gradients behave similarly to both the boundary sampling and the finite differences. While the boundary sampling usually requires multiple samples per pixel to reduce the variance, the analytical prefiltering approach can produce stable output with one sample per pixel.

Our approaches are significantly more efficient than finite difference when the number of variables to differentiate with is higher than five. The computation time of finite differences grows linearly with the number of variables, while our approach remains constant.

We analyze the variance of our Monte Carlo sampling in Figure 10. Like typical Monte Carlo estimators, the variance decreases linearly as the number of samples grows. We typically use $2 \times 2$ samples per pixel, and it works well across a wide range of tasks, from optimizing thousands of curves to training a neural network, as we will demonstrate below.

We apply our Monte Carlo differentiable vector graphics rasterizer to a wide variety of tasks. Unless specified otherwise, we use the Adam algorithm [Kingma and Ba 2015] to perform optimization. The learning rates are tuned for each application, and we found that having separate learning rates for color and points is crucial due to their different ranges. All images, SVGs, and optimization videos are included in the supplementary material.

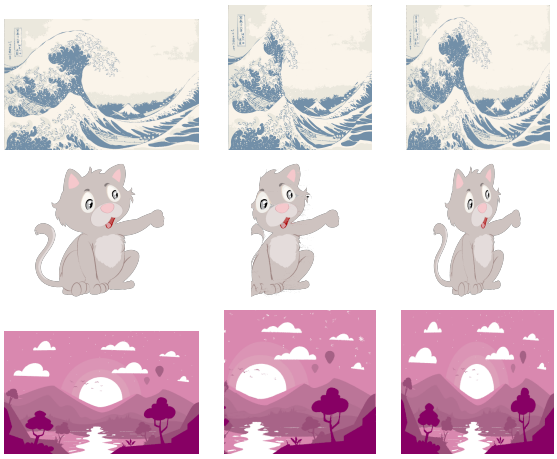
## 6.2 Optimization-based Vector Editing

By and large, current vector graphics editing tools are based around directly manipulating the geometry control points, or applying local or global affine transformations to them. More sophisticated editing tools based on a variety of geometric algorithms exist [Bernstein and Li 2015; Chugh et al. 2016; Sutherland 1964], but there is a deep divide between these and typical editing operations used on raster representations, such as brushes and filtering. Currently, there is no general-purpose way to bridge this gap and gain the ability to interpret these raster operations on the vector domain non-destructively.

Our approach opens an avenue to achieve this. With differentiable rasterization, we can define these editing operations as image-space losses and then backpropagate them to optimize the parameters

(a) (b)

Fig. 11. Examples of our brush-based editing using image losses. (a) shows the result (below) of locally optimizing opacity to sculpt the note while preserving parallelism of line segments (shown above). (b) shows the result (below) of optimization of a block of text characters (above) to locally increase opacity according to a pixel stencil while preserving the shape of characters by optimizing text "'boldness"'. We recommend readers viewing this document electronically zoom in to see the difference more clearly.



Initialization Seam Carving result Naive scaling

Fig. 12. We show the results of our vector Seam Carving. An initial graphic is shown next to the retargeted result and a naively rescaled graphic for reference. Images courtesy of freesvg.org user OpenClipArt and vecteezy.com users Vectorbox Studio and Graphics RF.

of the vector graphics representation. This way of editing is non-destructive in that it preserves the complete structure of the original graphic, which can be both semantically meaningful and convenient for other types of editing operations. While this optimization only finds local optima and cannot manipulate discrete parameters such as the topology of the shape, this is a desirable behavior in practice because the edit results in a slight modification of the input rather than an entirely new graphic.

*Interactive brush-based editing.* We may directly impose a loss on e.g., the color of pixels under a brush, which corresponds to brushing in a raster image. Optimizing the geometry for this loss causes the shape to deform in a way that facilitates the sculpting of vector shapes, as shown in Figure 11. We found that an operation that increases or decreases the opacity of the pixels to be useful. More involved pixel statistics can be used as well, such as imposing losses on pixel Laplacians to formulate a smoothing/sharpening operator. Losses backpropagated from the raster representation can easily be combined with geometric losses common in vector graphics [Bernstein and Li 2015] to enforce the preservation of desirable geometric properties like node smoothness or line parallelism. These simple brushing operators are fast enough for interactive editing. We use stochastic gradient descent (without momentum) as the optimizer since we found that the momentum introduced by Adam can hurt the interactive editing experience.

*Vector seam carving.* Many more advanced image filters and editing algorithms can also be transferred to the vector domain with this optimization strategy. To illustrate this, we have implemented a vector retargeting operator that changes the aspect ratio of a vector graphic using the seam carving algorithm in the raster domain [Avidan and Shamir 2007]. The original seam carving algorithm anisotropically scales an image while preserving structure or important content. At every step, until the desired aspect ratio is reached, it removes the "seam" that minimizes some cost functional (e.g., image gradients).

Our idea is to use the standard raster seam carving algorithm, but to reshape a vector graphic. We take an existing vector graphic, render it to a raster image using our differentiable rasterizer, apply one step of the seam carving algorithm on the raster image, and optimize the vector graphics to match the altered raster image using an $L_2$ loss with 10 gradient descent steps. Figure 12 illustrates the results. Any iterative raster image processing algorithm which makes reasonably small steps could in theory be used with the same scheme.

## 6.3 Vector Graphics Generation

Raster image generation using optimization and differentiable techniques is a well-established research area. Thanks to our rasterizer, we can now also achieve this in the vector domain . The most straightforward way to accomplish this is to iteratively optimize some initial vector shapes to match a target image.

*Painterly rendering.* For example, we may initialize the graphic with a randomly distributed set of primitives and optimize to match a target in the $L_2$ sense, as shown in Figure 13. This achieves

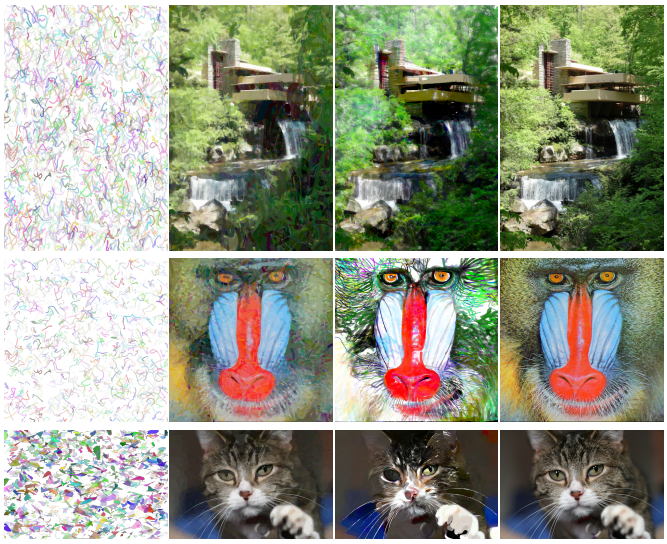



Fig. 13. Our renderer allows us to develop a novel painterly rendering algorithm by fitting randomly initialized curves to target images, optimizing control point positions, stroke widths, color, and opacity . The first column shows the curve initialization. The second column shows the results of minimizing $L^2$ distance between the rendering and the target. The third column shows the results of minimizing a deep-learning-based perceptual loss [Zhang et al. 2018]. The fourth column shows the target images. We found that the perceptual loss optimization usually results in more abstract, stylized images, while the $L^2$ loss usually delivers more photorealistic images. The first two rows are initialized with random strokes with 1 to 3 cubic Bézier curves per stroke. The third row is initialized with random closed Bézier curves with 3-5 segments per shape. Images courtesy of wikipedia user Daderot and David Corby, and USC-SIPI image database.

by optimization results similar to what *painterly rendering* algorithms [Hertzmann 1998, 2003] do by employing heuristics, or more recently reinforcement learning [Ganin et al. 2018; Nakano 2019]. More interestingly, we can use a deep feature-based perceptual loss [Zhang et al. 2018] for the same purpose, because our differentiable rasterizer can backpropagate the loss all the way back to the graphic elements. In practical terms, we have found that the $L_2$ loss generates more photorealistic graphics, while those created through the perceptual loss are more stylized and abstract. Other differentiable perceptual metrics can also be used [Artusi et al. 2019; Wang et al. 2004]. For all images, we use the same learning rates (1 for control points, 0.1 for stroke widths, and 0.01 for color) and run for 500 iterations.

*Image vectorization.* A plethora of image tracing algorithms exist that approximate a raster image with a vector graphic. Usually, these are based on trying to match curves to the edges detected in the pictures to discover shapes, and then coloring them post-hoc. While these produce empirically good results, they are not even locally optimal due to various simplifying assumptions that had to be made about the nature of the data. We can easily show – and fix – this by using such a traced image as an initialization for our optimization.

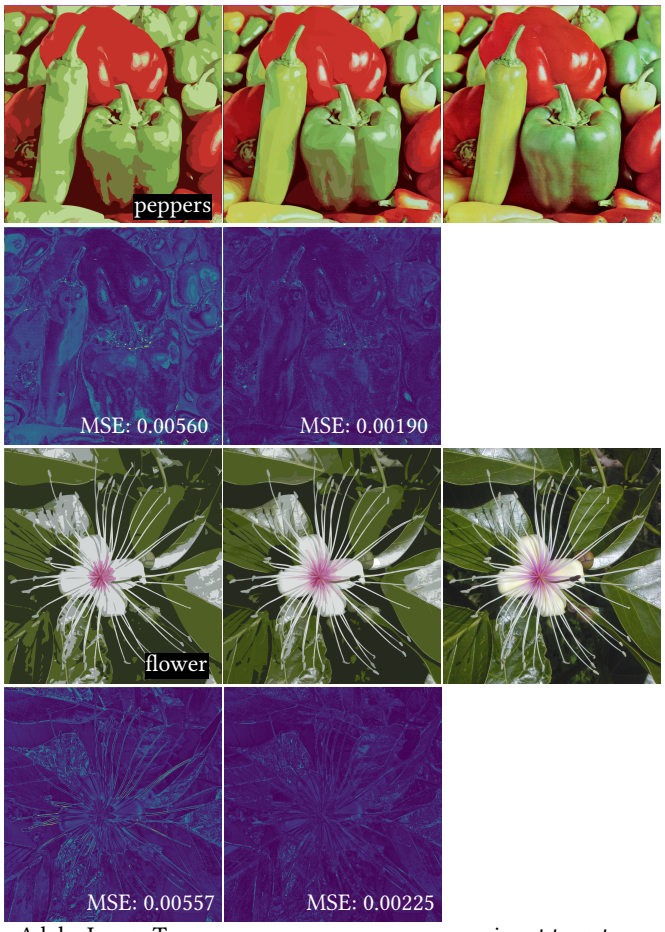


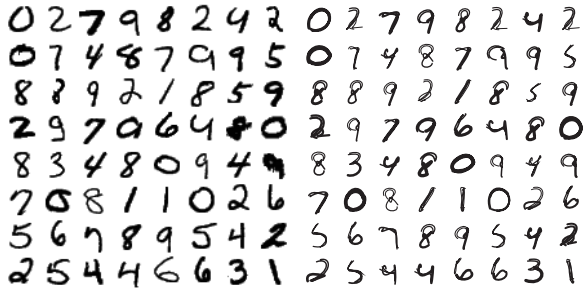

Fig. 14. Our rasterizer can be used to refine image vectorization results. Given an input raster and the output of the corresponding vectors from Adobe Image Trace, we optimize all the vector parameters, including color and control points, with respect to an $L^2$ loss. This significantly improves fidelity, both numerically and visually. Even rows show the absolute difference between the rendering and the target, and the mean squared error (MSE). Our results better represent gradual color changes, such as the thin, long, pepper on the left in *peppers*, the white reflection on the right, and the leaf at the bottom-left in *flower*. Images courtesy of wikipedia user Eric Guinther and USC-SIPI image database.

By taking the result of the tracing algorithm implemented in Adobe Illustrator, and then optimizing it to best match the target image in the $L_2$ sense, we achieve a roughly $2.5x$ reduction in the mean square error and 4 dB improvements on the peak signal-to-noise ratio, as shown in Figure 14 and Table 2. Our improvement is achieved by tweaking the control point positions and fill colors of the trace, without introducing additional complexity to the representation. For all images, we use the same learning rates (2 for control points and 0.02 for color) and run for 250 iterations.

*Deep generative modeling.* Our rasterizer can be used as a differentiable operator within a neural network — i.e., with other

Table 2. We refine the image vectorization results of Adobe Image Trace on a dataset of 5 images. Each image sees significant improvements in various error metrics. On average we achieve 2.5× reduction on mean square error and 4 dB improvement on peak signal-to-noise ratio. We tried both our Monte Carlo method (§5.1) and the analytical prefiltering method (§5.2) on this task. The analytical prefiltering method has slightly lower error, most likely due to the absence of noise, which leads to faster convergence in gradient descent.

| | MSE | PSNR | SSIM |
|---|---|---|---|
| Adobe Image Trace | 0.00712 | 21.84 | 0.7318 |
| ours (Monte Carlo) | 0.00316 | 25.54 | 0.8287 |
| ours (analytical prefiltering) | 0.00265 | 26.22 | 0.8494 |



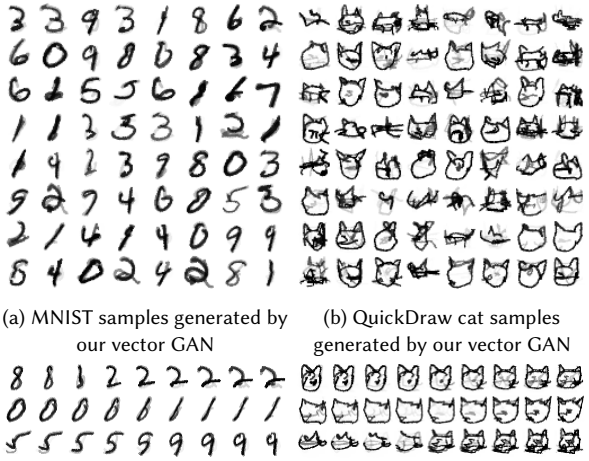
(a) random set of target images (b) our vector VAE outputs

Fig. 15. We trained a variational autoencoder to encode MNIST digits a into vector outputs. This enables us to sample new digits that are resolution-independent b. Zooming in on the digital version of the paper reveals the superior sharpness of our vector output.

differentiable components either preceding or following it. This allows us to adapt existing generative models for raster images into a counterpart that outputs a vector representation. For demonstration purposes, we have selected two models.

The first is a variational autoencoder (VAE) [Kingma and Welling 2014], which we trained to produce vector MNIST digits [LeCun et al. 1998]. The encoder is a convolutional network. It processes the grayscale raster digits, and encodes them into a 20-D latent vector. The decoder transforms a latent code (which after training can be sampled from a 20-D normal distribution) into the points positions, stroke width, and opacity parameters of a pair of two-segment Bézier paths. These paths are then rasterized using our method to produce an output image that we can directly compare to the raster input. We train this network using an $L_2$ loss between the ground truth image and the rasterized output, and a Kullback-Liebler divergence that encourages the latent vectors to be normally distributed. Figure 15 shows a random sample of vector outputs and the ground-truth MNIST reference images. Figure 1e shows how vector outputs enable resolution-independent renderings, and even *a posteriori* stylization of the digits.

Our second model demonstrates that the differentiable rasterizer works equally well in the know-to-be-challenging generative adversarial training context [Goodfellow et al. 2014]. For this example, we trained a similar decoder architecture to minimize the Wasserstein



(a) MNIST samples generated by our vector GAN (b) QuickDraw cat samples generated by our vector GAN

(c) Latent space interpolation (d) Latent space interpolation

Fig. 16. Our rasterizer can also be used in a Generative Adversarial Networks framework to train vector-generating models from a dataset of raster images. Here we show random samples from models trained on the MNIST dataset a and the 'cats' subset of the QuickDraw dataset b. We also show the result of interpolating pairs of samples in the latent space of each model c and d.

GAN with the Gradient Penalty objective [Gulrajani et al. 2017]. In this example, the discriminator is a convolutional network that either takes as input an image from the training dataset, or a generated vector graphics rasterized with our algorithm, and tries to classify the image as real or fake. In Figure 16, we show some rasterized vector graphics produced by these models trained on two datasets — MNIST and the more complex sketches of cats from QuickDraw [Ha and Eck 2018]. We also show examples of interpolations of random latent vectors in the GAN's latent space. The training took about 12 hours for each model. For more details of the architecture and training, see Appendix C.

Thanks to our differentiable rasterization, we can train generative models with only image-based supervision, without requiring any supervision on the curves themselves like previous approaches [Ha and Eck 2018; Lopes et al. 2019]. Neither do we need to train a neural network for approximating the rasterization process, which usually fixes and limits the output resolution and only works on a single type of vector primitive [Huang et al. 2019; Nakano 2019; Zheng et al. 2019].

## 7 LIMITATIONS AND FUTURE WORK

*Optimizing topologies.* Our method shares the same limitation as other gradient-based optimization methods. Our renderer only gives gradients for continuous parameters like control point positions, color values, and transformation parameters. It cannot create gradients for discrete decisions, such as adding or removing shapes or path segments, rearranging the rendering order, or changing shape types. As such, applications presented above are limited to fixed topologies, determined either heuristically, or by preserving the topology of the input. It is possible to use a recurrent or reinforcement learning model to handle the discrete decisions [Ha and Eck

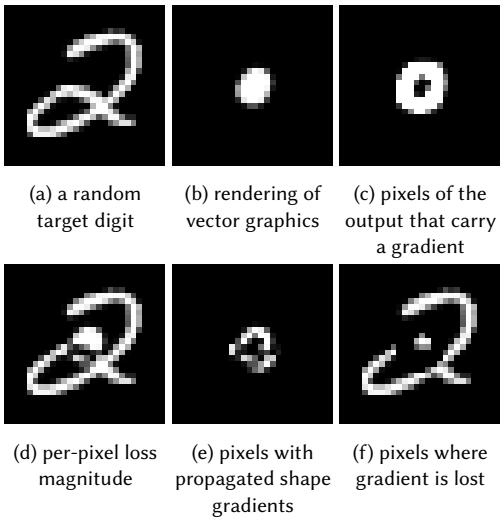



Fig. 17. In color rendering, shape gradients are only propagated along boundaries (c, d), so gradient information is lost in non-boundary pixels (f).

2018; Lopes et al. 2019; Zhu et al. 2018], while additionally benefiting from being able to compute gradients on continuous parameters.

*Sparsity of gradients.* In Reynolds' formulation, the backpropagation of gradients from image space to shape parameters depends on the shape boundaries. In practice, this means that information from pixels not incident to the boundaries is lost (Figure 17). In some cases, this may induce local minima that lead to degenerate geometries, or even lead to an unrecoverable configuration where the shape is not rendered on the canvas at all (e.g., due to the entire shape having moved off-canvas where no gradient signal is available). This can be mitigated for shape parameters by using larger pixel filters, or by relying on the analytical prefiltering formulation instead. However, rendering with larger pixel filters is more expensive, while prefiltering does not address all such local minima.

*More vector graphics primitives.* Our current implementation does not support some advanced primitives like diffusion curves [Orzan et al. 2008] and gradient meshes [Sun et al. 2007]. In their original form, diffusion curves require differentiating a Poisson equation solver. However, the ray tracing formulation of diffusion curves [Bowers et al. 2011] would be relatively simple to integrate. Differentiating gradient mesh rendering requires differentiating their tessellation into triangle meshes. Gradient meshes also contain discontinuities when a patch *folds over* onto itself [Adobe Inc. 2006].

## 8 CONCLUSION

We have presented a differentiable rasterizer for vector graphics capable of converting vector data to a raster representation, while facilitating backpropagation between the two domains. It facilitates both direct optimization of vector data based on raster criteria, and seamless integration of vector graphics components into deep learning models which rely heavily on image-space convolution. We hope that this new technique will enable new ways of learning and optimizing visual data across representations.

## ACKNOWLEDGMENTS

The main author is funded by DARPA under award HR0011-20-9-0017, from the PAPPA program. We thank Andrew Adams and Gilbert Bernstein for their helpful comments. We thank people who provided the graphics used in the figures: wikipedia users Daderot (fallingwater), Eric Guinther (flower), and David Corby (kitty), freesvg.org user OpenClipart (The Wave), and vecteezy.com users Vectorbox Studio and Graphics RF (cat and landscape).

## REFERENCES

Adobe Inc. 2006. PDF Reference, six edition. https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdf_reference_archive/pdf_reference_1-7.pdf.

Alessandro Artusi, Francesco Banterle, Alejandro Moreo, and Fabio Carrara. 2019. Efficient Evaluation of Image Quality via Deep-Learning Approximation of Perceptual Metrics. *IEEE Trans. Image Process.* 29 (2019), 1843–1855.

Shai Avidan and Ariel Shamir. 2007. Seam Carving for Content-Aware Image Resizing. *ACM Trans. Graph. (Proc. SIGGRAPH)* 26, 3 (2007), 10.

Samaneh Azadi, Matthew Fisher, Vladimir G Kim, Zhaowen Wang, Eli Shechtman, and Trevor Darrell. 2018. Multi-content GAN for few-shot font style transfer. In *Computer Vision and Pattern Recognition*. 7564–7573.

Dejan Azinović, Tzu-Mao Li, Anton Kaplanyan, and Matthias Nießner. 2019. Inverse Path Tracing for Joint Material and Lighting Estimation. In *Computer Vision and Pattern Recognition*.

Francesco Banterle, Massimiliano Corsini, Paolo Cignoni, and Roberto Scopigno. 2012. A low-memory, straightforward and fast bilateral filter through subsampling in spatial domain. In *Comput. Graph. Forum*, Vol. 31. Wiley Online Library, 19–32.

Vineet Batra, Mark J. Kilgard, Harish Kumar, and Tristan Lorach. 2015. Accelerating Vector Graphics Rendering Using the Graphics Hardware Pipeline. *ACM Trans. Graph. (Proc. SIGGRAPH)* 34, 4 (2015), 146:1–146:15.

Bradley M Bell and James V Burke. 2008. Algorithmic differentiation of implicit functions and optimal values. In *Advances in Automatic Differentiation*. Springer.

Gilbert Louis Bernstein and Wilmot Li. 2015. Lillicon: Using Transient Widgets to Create Scale Variations of Icons. *ACM Trans. Graph. (Proc. SIGGRAPH)* 34, 4 (2015).

John C Bowers, Jonathan Leahey, and Rui Wang. 2011. A ray tracing approach to diffusion curves. 30, 4 (2011), 1345–1352.

Edwin Catmull. 1978. A Hidden-Surface Algorithm with Anti-Aliasing. *Comput. Graph. (Proc. SIGGRAPH)* 12, 3 (1978), 6–11.

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. 51, 6 (2016), 341–354.

Adrian Colomitchi. 2006. Approximating cubic Bezier curves by quadratic ones. http://caffeineowl.com/graphics/2d/vectorial/cubic2quad01.html#pseudoQuadDist. Accessed: 2020-05-21.

Robert L. Cook. 1986. Stochastic Sampling in Computer Graphics. *ACM Trans. Graph.* 5, 1 (1986), 51–72.

Joe Cridge. 2015. Approximating Arcs Using Cubic Bézier Curves. https://www.joecridge.me/content/pdf/bezier-arcs.pdf

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

Mark A. Z. Dippé and Erling Henry Wold. 1985. Antialiasing through Stochastic Sampling. *Comput. Graph. (Proc. SIGGRAPH)* 19, 3 (1985), 69–78.

Tom Duff. 1989. Polygon scan conversion by exact convolution.. In *International Conference On Raster Imaging and Digital Typography*. 154–168.

David Eberly. 2011. Distance from a point to an ellipse, an ellipsoid, or a hyperellipsoid. *Geometric Tools, LLC* (2011).

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-drawn Images. In *Advances in Neural Information Processing Systems*. Curran Associates Inc., 6062–6071.

A. E. Fabris and A. R. Forrest. 1997. Antialiasing of Curves by Discrete Pre-Filtering. In *SIGGRAPH*. ACM, 317–326.

Rida T Farouki and C Andrew Neff. 1990. Algebraic properties of plane offset curves. *Computer Aided Geometric Design* 7, 1-4 (1990), 101–127.

Harley Flanders. 1973. Differentiation under the integral sign. *The American Mathematical Monthly* 80, 6 (1973), 615–627.

Francisco Ganacim, Rodolfo S. Lima, Luiz Henrique de Figueiredo, and Diego Nehab. 2014. Massively-parallel Vector Graphics. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 33, 6 (2014), 229:1–229:14.

Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals. 2018. Synthesizing Programs for Images using Reinforced Adversarial Learning. In *International Conference on Machine Learning*. 1666–1675.

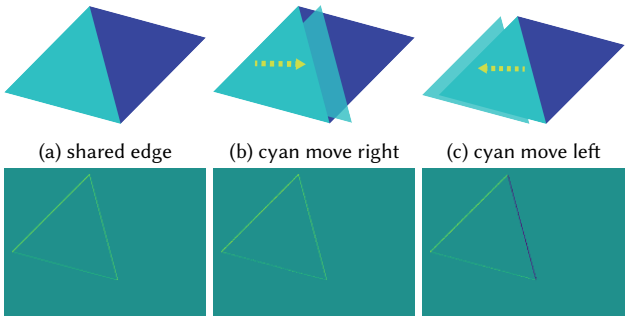
Leon A Gatys, Alexander S Ecker, and Matthias Bethge. 2016. Image style transfer using convolutional neural networks. In *Computer Vision and Pattern Recognition*.

IEEE, 2414–2423.
Michael Gleicher. 1992. Briar: A Constraint-Based Drawing Program. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM.
Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in Neural Information Processing Systems*. 2672–2680.
Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (second ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.
Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. 2017. Improved training of wasserstein gans. In *Advances in Neural Information Processing Systems*. 5767–5777.
Satish Gupta and Robert F. Sproull. 1981. Filtering Edges for Gray-Scale Displays. *Comput. Graph. (Proc. SIGGRAPH)* 15, 3 (1981), 1–5.
David Ha and Douglas Eck. 2018. A Neural Representation of Sketch Drawings. In *International Conference on Learning Representations*.
Aaron Hertzmann. 1998. Painterly rendering with curved brush strokes of multiple sizes. In *SIGGRAPH*. ACM, 453–460.
Aaron Hertzmann. 2003. A survey of stroke-based rendering. *IEEE Comput. Graph. Appl.* 4 (2003), 70–81.
Zhewei Huang, Wen Heng, and Shuchang Zhou. 2019. Learning to paint with model-based deep reinforcement learning. In *International Conference on Computer Vision*. 8709–8718.
Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. 2017. Image-to-image translation with conditional adversarial networks. In *Computer Vision and Pattern Recognition*. 1125–1134.
Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. 2018. Neural 3D Mesh Renderer. In *Computer Vision and Pattern Recognition*. IEEE, 3907–3916.
Mark J. Kilgard. 2020. Polar Stroking: New Theory and Methods for Stroking Paths. *ACM Trans. Graph. (Proc. SIGGRAPH)* 39, 4 (2020).
Mark J. Kilgard and Jeff Bolz. 2012. GPU-accelerated Path Rendering. *ACM Trans. Graph. (Proc. SIGGRAPH)* 31, 6 (2012), 172:1–172:10.
Diederick P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.
Diederik P Kingma and Max Welling. 2014. Auto-encoding variational Bayes. In *International Conference on Learning Representations*.
Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. 2017. Self-normalizing neural networks. In *Advances in Neural Information Processing Systems*. 971–980.
Yoshiyuki Kokojima, Kaoru Sugita, Takahiro Saito, and Takashi Takemoto. 2006. Resolution Independent Rendering of Deformable Vector Objects Using Graphics Hardware. In *ACM SIGGRAPH 2006 Sketches*. ACM, 118.
Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH construction on GPUs. *Comput. Graph. Forum (Proc. Eurographics)* 28, 2 (2009), 375–384.
Gregory Lecot and Bruno Levy. 2006. Ardeco: Automatic Region DEtection and COnversion. (2006).
Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
Raph Levien. 2020. Fast 2D rendering on GPU. https://raphlinus.github.io/rust/graphics/gpu/2020/06/13/fast-2d-rendering.html Accessed: 2020-08-24.
Rui Li, Qiming Hou, and Kun Zhou. 2016. Efficient GPU Path Rendering Using Scanline Rasterization. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 35, 6 (2016), 228:1–228:12.
Tzu-Mao Li. 2019. *Differentiable Visual Computing*. Ph.D. Dissertation. Massachusetts Institute of Technology. Advisor(s) Durand, Frédo.
Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. 2018. Differentiable Monte Carlo Ray Tracing through Edge Sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 6 (2018), 222:1–222:11.
Hsueh-Ti Derek Liu, Michael Tao, and Alec Jacobson. 2018. Paparazzi: Surface Editing by Way of Multi-view Image Processing. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 6 (2018), 221:1–221:11.
Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. 2019. Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning. *International Conference on Computer Vision* (2019).
Charles Loop and Jim Blinn. 2005. Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph. (Proc. SIGGRAPH)* 24, 3 (2005), 1000–1009.
Matthew M. Loper and Michael J. Black. 2014. OpenDR: An Approximate Differentiable Renderer. In *European Conference on Computer Vision*, Vol. 8695. ACM, 154–169.
Raphael Gontijo Lopes, David Ha, Douglas Eck, and Jonathon Shlens. 2019. A Learned Representation for Scalable Vector Graphics. In *International Conference on Computer Vision*.
Josiah Manson and Scott Schaefer. 2013. Analytic rasterization of curves with polynomial filters. *Comput. Graph. Forum (Proc. Eurographics)* 32, 24 (2013), 499–507.
Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. 2018. Spectral Normalization for Generative Adversarial Networks. In *International Conference on Learning Representations*.
Reiichiro Nakano. 2019. Neural Painters: A learned differentiable constraint for generating brushstroke paintings. *arXiv preprint arXiv:1904.08410* (2019).
Diego Nehab. 2020. Converting Stroked Primitives to Filled Primitives. *ACM Trans. Graph. (Proc. SIGGRAPH)* 39, 4 (2020).
Diego Nehab and Hugues Hoppe. 2008. Random-access Rendering of General Vector Graphics. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 27, 5 (2008), 135:1–135:10.
Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, and David Salesin. 2008. Diffusion Curves: A Vector Representation for Smooth-shaded Images. *ACM Trans. Graph. (Proc. SIGGRAPH)* 27, 3 (2008), 92:1–92:8.
Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*. 8024–8035.
Thomas Porter and Tom Duff. 1984. Compositing digital images. (1984), 253–259.
William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes 3rd Edition: The Art of Scientific Computing* (3 ed.). Cambridge University Press, USA.
Osborne Reynolds, Arthur William Brightmore, and William Henry Moorby. 1903. *The sub-mechanics of the universe*. Vol. 3. University Press.
Oswaldo Rio Branco de Oliveira. 2012. The Implicit and the Inverse Function theorems: easy proofs. *arXiv preprint arXiv:1212.2066* (2012).
Thomas W Sederberg and Geng-Zhe Chang. 1994. Isolator polynomials. In *Algebraic Geometry and Its Applications*. Springer, 507–512.
Peter Selinger. 2003. Potrace: a polygon-based tracing algorithm. http://potrace.sourceforge.net/potrace.pdf
Jian Sun, Lin Liang, Fang Wen, and Heung-Yeung Shum. 2007. Image Vectorization Using Optimized Gradient Meshes. *ACM Trans. Graph. (Proc. SIGGRAPH)* 26, 3 (2007).
Ivan E. Sutherland. 1964. Sketch Pad: a Man-machine Graphical Communication System. In *Proceedings of the SHARE Design Automation Workshop*. 6.329–6.346.
K. Turkowski. 1982. Anti-Aliasing through the Use of Coordinate Transformations. *ACM Trans. Graph.* 1, 3 (1982), 215–234.
Zhou Wang, Alan C Bovik, Hamid R Sheikh, Eero P Simoncelli, et al. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Trans. Image Process.* 13, 4 (2004), 600–612.
Tian Xia, Binbin Liao, and Yizhou Yu. 2009. Patch-Based Image Vectorization with Automatic Curvilinear Feature Alignment. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* (2009).
Guofu Xie, Xin Sun, Xin Tong, and Derek Nowrouzezahrai. 2014. Hierarchical Diffusion Curves for Accurate Automatic Image Vectorization. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 33, 6 (2014), 230:1–230:11.
Ming Yang, Hongyang Chao, Chi Zhang, Jun Guo, Lu Yuan, and Jian Sun. 2015. Effective clipart image vectorization through direct optimization of bezigons. *IEEE Trans. Vis. Comput. Graph.* 22, 2 (2015), 1063–1075.
Gao Yue, Guo Yuan, Lian Zhouhui, Tang Yingmin, and Xiao Jianguo. 2019. Artistic Glyph Image Synthesis via One-Stage Few-Shot Learning. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 38, 6 (2019).
Cheng Zhang, Lifan Wu, Changxi Zheng, Ioannis Gkioulekas, Ravi Ramamoorthi, and Shuang Zhao. 2019. A differential theory of radiative transfer. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 38, 6 (2019), 227.
Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. 2018. The Unreasonable Effectiveness of Deep Features as a Perceptual Metric. In *Computer Vision and Pattern Recognition*. 586–595.
Shuang Zhao, Frédo Durand, and Changxi Zheng. 2018. Inverse diffusion curves using shape optimization. *IEEE Trans. Vis. Comput. Graph.* 24, 7 (2018), 2153–2166.
Ningyuan Zheng, Yifan Jiang, and Dingjiang Huang. 2019. StrokeNet: A Neural Painting Environment. In *International Conference on Learning Representations*.
Chenyang Zhu, Kai Xu, Siddhartha Chaudhuri, Renjiao Yi, and Hao Zhang. 2018. SCORES: Shape Composition with Recursive Substructure Priors. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 6 (2018), 14.

## A CLOSEST DISTANCE BETWEEN A POINT AND A CUBIC BÉZIER CURVE

Given a cubic curve $p(t) = (1-t)^3 p_0 + 3(1-t)^2 t p_1 + 3(1-t)t^2 p_2 + t^3 p_3$ and a point $q \in \mathcal{R}^2$, we want to compute the closest distance between them. We write down the squared distance as $\left(p(t^\star) - q\right)^2$ where $t^\star$ is the closest point to $q$ on the curve $p$:

$$t^\star = \arg\min_t \left(p(t) - q\right)^2 . \tag{11}$$



(a) shared edge (b) cyan move right (c) cyan move left

(d) our MC derivative (e) forward difference (f) backward difference

Fig. 18. Rasterization of two primitives with their edges exactly coinciding a is not differentiable with respect to the vertex positions. Suppose we want to differentiate with respect to the cyan triangle's horizontal translation parameter. When it moves right b, it reduces the coverage of the blue triangle, making the color brighter. When the cyan triangle moves left c, it reveals more white background, making the color darker. Therefore forward and backware differences produce different results (e and f), making the function non-differentiable. Our method outputs the derivative defined by one-side of the limit, where the cyan triangle occludes the blue triangle d.

To solve for $t^\star$, we take the derivative of the squared distance with respect to $t$ and set it to zero:

$$2\left(p(t)-q\right)\cdot p'(t) = \rho(t) = 0. \tag{12}$$

This is a 5th order polynomial ($p$ is 3rd order and $p'$ is 2nd order) and does not have a closed-form solution. To solve for all real roots of the 5th order polynomial, we use a standard hybrid Newton-Raphson bisection method [Press et al. 2007]. This requires us to identify the intervals of all the real roots. To achieve this, we use the technique of *isolator polynomial* [Sederberg and Chang 1994]. Given a polynomial $\rho(t)$ (in our case a 5th order polynomial) and its derivative $\rho'(t)$, the long division of them yields two lower-order polynomials $a(t)$ and $b(t)$:

$$a(t) = \rho(t) - b(t)\rho'(t). \tag{13}$$

It turns out the real roots of $a$ and $b$ *isolate* the roots of $\rho$. To see this, consider any two adjacent real roots of $\rho$, $r_0$ and $r_1$ (if $\rho$ only has one or zero real root, we know $t$ is bounded by $[0,1]$). We can show that there must be a root of either $a$ or $b$ in $[r_0, r_1]$. Since $\rho(r_0) = \rho(r_1) = 0$, we know $a(r_0)a(r_1) = b(r_0)b(r_1)\rho'(r_0)\rho'(r_1)$. Since $r_0$ and $r_1$ are roots of $\rho$, $\rho'(r_0)\rho'(r_1) \le 0$. Thus either $a(r_0)a(r_1) \le 0$ or $b(r_0)b(r_1) \le 0$. See Sederberg and Chang's paper for remarks on multiple roots.

We can construct a 5th order polynomial $\rho(t) = t^5 + Bt^4 + Ct^3 + Dt^2 + Et + F$ by rearranging Equation 12. By doing a long division between $\rho$ and $\rho'$, we obtain

$$\begin{aligned} a(t) &= \left(\frac{2C}{5} - \frac{4B^2}{25}\right)t^3 + \left(\frac{3D}{5} - \frac{3BC}{25}\right)t^2 + \left(\frac{4E}{5} - \frac{2BD}{25}\right)t + F - \frac{BE}{25} \\ b(t) &= \frac{t}{5} + \frac{B}{25}. \end{aligned} \tag{14}$$

Since $a$ is a cubic polynomial and $b$ is linear, we can solve for all the real roots of $a$ and $b$ between $(0,1)$. This forms at most 5 intervals, and we can then find all roots of $\rho$ within these intervals using the hybrid Newton-Raphson bisection method.

## B PARALLEL EDGES AND NON-DIFFERENTIABILITY OF RASTERIZATION

Rasterization is not technically differentiable in the degenerate case of the parallel edges (Figure 18). In this case, the integral is discontinuous with respect to the vertices' movement: moving the top primitive to one direction reveals the background, and moving it to the other direction occludes the other primitive, thus the two sides of the limit are not the same. Since we only compute the topmost contribution when evaluating the scene function $f$, our method outputs the derivative corresponding to the limit that does not involve the background, ignoring the effect of the background.

In the presence of the parallel edge, our multisampling antialiasing would still output one side of the limit, which we argue is the correct behavior in this situation. Our analytical prefiltering would generate conflation artifacts (Figure 8).

## C NETWORK ARCHITECTURES AND TRAINING DETAILS

The encoder network of our MNIST VAE (§6.3) maps an input image to a latent vector. The encoder is a sequence of convolution layers with leaky ReLU activations (with a negative slope of 0.2). We use 3 convolution layers with $3 \times 3$ filters and progressively increasing feature channels: 64, 128, and 256. In the GAN setting, the latent vector is sampled from a multivariate normal distribution.

In the VAE experiment, our decoder is a fully-connected network with SELU activations [Klambauer et al. 2017]. It consumes a latent vector and outputs the parameters of $N$ cubic Bézier segments: positions, stroke widths, and opacity. We use a hyperbolic tangent as the last activation to obtain positions in $[-1,1]$, and a sigmoid function for stroke widths and opacity in $[0,1]$. The decoder uses 3 fully-connected layers, all with 1024 hidden units.

The generator in both the QuickDraw and MNIST GAN experiments is also a SELU, fully-connected network. It uses 5 fully-connected layers with 32, 64, 128, 256, 256 hidden units respectively. The discriminator has a structure similar to the VAE's encoder, with two modifications: it uses spectral normalization [Miyato et al. 2018] between the convolution layers, and outputs a single scalar instead of a latent vector. It has 8 convolution layers with $3 \times 3$ filters and 64, 128, 128, 256, 256, 512, 512, 512, 512 channels, with a fully-connected network for the final real/fake classification output. All convolutions, except the first two, use spectral normalization.

We use the Adam optimizer [Kingma and Ba 2015], with learning rate set to 0.0001 and $\beta_1 = 0.5, \beta_2 = 0.9$. In the GAN experiments, we schedule the learning rate to decay exponentially with factor 0.9999. The gradient penalty is set to 10, and we clip gradients with norm larger than 1.