Converting stroked primitives to filled primitives

DIEGO NEHAB, IMPA, Brazil

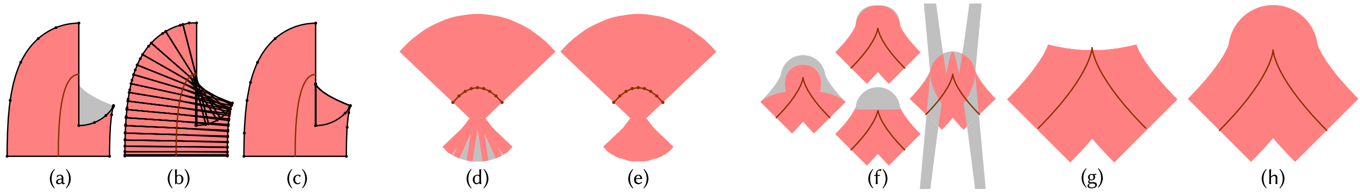

Fig. 1. We analyzed 22 distinct third-party strokers. (a) 14 of them confuse the stroke-to-fill conversion problem with the curve-offsetting problem, and produce incorrect results near high-curvature regions. (b) The remaining 8 do treat high-curvature regions, but offer no accuracy guarantees, and often output too many segments. (c) Our stroker correctly handles these regions using evolutes. (d) All but 2 strokers ignore inner joins between segments, leaving unexpected gaps. (e) Our stroker detects when such joins are visible (even between curved segments) and outputs them. (f) All but 2 strokers fail spectacularly in the vicinity of intra-segment cusps and "almost cusps". (g) Following the standards to the letter produces discontinuous results at cusps. (h) Our stroker robustly detects cusps and "almost cusps" to produce continuous, intuitive results. — input path — output path ■ filled region ■ missing or extraneous region

Vector graphics formats offer support for both filled and stroked primitives. Filled primitives paint all points in the region bounded by a set of outlines. Stroked primitives paint all points covered by a line drawn over the outlines. Editors allow users to convert stroked primitives to the outlines of equivalent filled primitives for further editing. Likewise, renderers typically convert stroked primitives to equivalent filled primitives prior to rendering. This conversion problem is deceivingly difficult to solve. Surprisingly, it has received little to no attention in the literature. Existing implementations output too many segments, do not satisfy accuracy requirements, or fail under a variety of conditions, often spectacularly. In this paper, we present a solution to the stroke-to-fill conversion problem that addresses these issues. One of our key insights is to take into account the evolutes of input outlines, in addition to their offsets, in regions of high curvature. Furthermore, our approach strives to maintain continuity between the input and the set of painted points. Our implementation is available in open source.

CCS Concepts: • **Computing methodologies** → *Graphics file formats*; *Parametric curve and surface models*; *Rasterization.*

Additional Key Words and Phrases: stroke, vector graphics, offset curves

ACM Reference Format:

Diego Nehab. 2020. Converting stroked primitives to filled primitives. *ACM Trans. Graph.* 39, 4, Article 1 (July 2020), 17 pages. https://doi.org/10.1145/3386569.3392392

1 INTRODUCTION

Vector graphics is the standard way of describing scalable visual information, such as pages of text, illustrations, maps, etc. Well-known vector graphics file formats include PS [1999], PDF [2006], SVG [2011], OpenXPS [2009], and CGM [1999]. All these formats closely follow the framework put forth in the seminal work by Warnock and Wyatt [1982]. They define an illustration as a series of *shapes* to be *painted* in order, each one on top of the previous. Each shape-paint combination forms a *layer*. The process mimics silk-screen printing, with shapes working as stencils that restrict the area of the illustration affected by paint. Paints can be constant, vary spatially in gradients, or even consist of arbitrary textures. Shapes can be *filled* or *stroked*. Both types of shapes are defined by *paths* (i.e., piecewise-polynomial outlines). Points are *inside* a filled shape based on their *winding number* relative to the path. (Both *odd* and *non-zero* tests are widely supported.) Stroked shapes, in contrast, consist of the set of points covered when a line of given *width* is drawn over the path.

We must often find a set of outlines defining a shape that, when filled, paints the same set of points as those painted by a given stroked shape. The focus of this paper is this *stroke-to-fill conversion problem* (or simply *stroking*). The fact that filled and stroked shapes define interior points in entirely different ways is inconvenient for vector graphics renderers. Rather than using two distinct algorithms at rendering time, virtually all rendering engines convert stroked shapes to equivalent filled shapes prior to rendering. (The only exception we found is NVpr [Kilgard and Bolz 2012].) A solution to the stroke-to-fill problem is also useful outside the context of rendering: Vector graphics editors allow users to convert stroked primitives to their outlines to enable further editing (e.g., Inkscape's "Stroke to Path" or Adobe Illustrator's "Outline Stroke").

Surprisingly, in this problem's nearly 40 years of history, it has rarely, if ever, been mentioned in the literature. As a consequence, developers of vector graphics rendering engines and editors keep "reinventing the wheel". The approaches followed by Anti-Grain Geometry, the Cairo Graphics Library, Microsoft's Direct2D, Apple's Quartz, Ghostscript, MuPDF, MPVG [Ganacim et al. 2014], Java 2D, the livarot library (used by Inkscape), Qt 5, the OpenVG Reference Implementation, the Skia Graphics Library, and Adobe Illustrator are all different *and* inconsistent with each other.

These implementations can be broadly categorized according to two axes: *flat* vs. *curve-based*, and *local* vs. *global*. *Flat* algorithms operate on piecewise-linear approximations of the input path (i.e., they *flatten* the input). Although this makes the problem easier to solve, an undesirably large number of output segments are needed

Author's address: Diego Nehab, IMPA, Rio de Janeiro, Brazil.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0730-0301/2020/7-ART1 $15.00
https://doi.org/10.1145/3386569.3392392

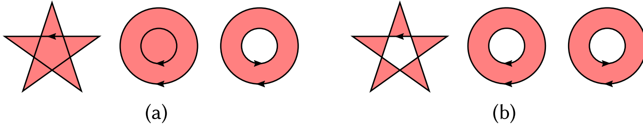

Fig. 2. (a) The non-zero and (b) even-odd winding rules.

to achieve good accuracy. In contrast, *curve-based* algorithms work directly on input curves to produce output paths containing relatively fewer segments. This is particularly advantageous for later editing, or if the target rendering engine can fill shapes bounded by polynomial segments without flattening them first. *Local* algorithms produce multiple output paths. Since these paths potentially overlap, they are typically filled into a stencil buffer as soon as they are produced. When all paths have been filled, the points marked in the stencil are painted over the output image. This prevents any point from being painted twice, which is important for compositing with transparency. In contrast, *global* algorithms produce a single output path that can be rendered directly over the output image. This enables them to omit many of the redundant segments that local algorithms output twice in opposite orientations.

Stroke-to-fill conversion is a deceivingly difficult problem to solve correctly. It is frequently confused with the relatively simpler *curve offsetting problem*, which is merely part of its solution. Another challenge is robustly identifying cusps within the input segments for proper treatment. Even those irregularities that have been marked explicitly (e.g., inner joins between adjacent input segments) are rarely handled correctly in practice. To the best our knowledge, all existing curve-based implementations solve the offsetting problem instead of the harder problem at hand. Furthermore, even the most robust flat implementations fail to meet any accuracy requirements.

This is a sorry state of affairs. The closer we move towards a fully digital future, the more important it becomes to consistently render documents. A complete solution to the stroke-to-fill conversion problem is necessary and long overdue.

Our work makes the following contributions:

- We strengthen the definition of what it means to stroke a path to eliminate poorly-defined corner cases (section 2);
- We propose a strategy for regularizing the input that is robust to intra-segment cusps and "almost cusps" (section 3.1);
- We show how to conservatively identify when inner joins can be omitted, even between curved segments (section 3.2);
- We present the first global curve-based stroker that works correctly near high-curvature regions (section 3.3);
- We discuss 22 different prior implementations (section 4), compare their results against ground truth, and show the improvements brought by our proposal (section 5).

2 PROBLEM STATEMENT

Path definition. A path is a list of outlines $o_1, o_2, \ldots, o_m$. Each outline o_i is a list of connected segments $\alpha_{i,1}, \ldots, \alpha_{i,n_i}$. A segment $\alpha_{i,j} : [0,1] \to \mathbf{R}^2$ is a parametric plane curve. Outlines are connected, so $\alpha_{i,j}(1) = \alpha_{i,j+1}(0)$, for $j \in \{1, \ldots, n_i - 1\}$ and $i \in \{1, \ldots, m\}$. An outline o_i can be *closed* to form a loop. In that case, if $\alpha_{i,n_i}(1) \neq \alpha_{i,1}(0)$, then a linear segment α_{i,n_i+1} with $\alpha_{i,n_i+1}(0) = \alpha_{i,n_i}(1)$ and $\alpha_{i,n_i+1}(1) = \alpha_{i,1}(0)$ is implicitly added to the outline.

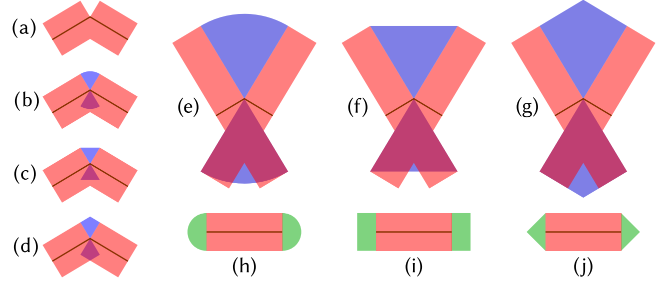

Fig. 3. (a) Stroking segments independently can leave gaps when they meet at an angle. *Joins* close these gaps. (b,c,d) Round, bevel, and miter joins are shown in blue. When the line width is large enough, another gap appears. It must filled by an *inner join*. In (b,c,d), these inner joins (in purple) are completely hidden by the stroked segments. In (e,f,g), they are partially visible in blue. The starts and ends of dashes and outlines can be decorated with *caps*. Round, square, and triangle caps are shown in green in (h,j,i), and butt (omitted) caps in (b,c,d,e,f,g).

Segment definition. Linear segments, parabolic arcs, elliptical arcs, or cubic segments are typically allowed in outlines. A segment $\alpha_{i,j}$ is specified by its Bézier control points $p_{i,j,0}, \ldots, p_{i,j,d_{i,j}} \in \mathbf{R}^2$, where $p_{i,j,k} = \begin{bmatrix} x_{i,j,k} & y_{i,j,k} \end{bmatrix}$, and $d_{i,j} \in \{1, 2, 3\}$. Formally, let

$$b_{k,d}(t) = \binom{d}{k}(1-t)^{d-k}t^k \tag{1}$$

define the kth Bernstein basis polynomial of degree d. Then,

$$\alpha_{i,j}(t) = \begin{bmatrix} x_{i,j}(t) & y_{i,j}(t) \end{bmatrix} = \sum_{k=0}^{d_{i,j}} p_{i,j,k}\, b_{k,d_{i,j}}(t). \tag{2}$$

To represent elliptical arcs or allow projective transformations of segments, *rational* Béziers are needed. In this case, the control points $\tilde{p}_{i,j,0}, \ldots, \tilde{p}_{i,j,d_{i,j}}$, where $\tilde{p}_{i,j,k} = \begin{bmatrix} u_{i,j,k} & v_{i,j,k} & w_{i,j,k} \end{bmatrix} \in \mathbf{R}^3$ and $d_{i,j} \in \{1, 2, 3\}$, define a curve $\tilde{\alpha}_{i,j}(t)$ in the projective plane $\mathbf{RP}^2$:

$$\tilde{\alpha}_{i,j}(t) = \begin{bmatrix} u_{i,j}(t) & v_{i,j}(t) & w_{i,j}(t) \end{bmatrix} = \sum_{k=0}^{d_{i,j}} \tilde{p}_{i,j,k}\, b_{k,d_{i,j}}(t). \tag{3}$$

The segment $\alpha_{i,j}(t)$ is the projection of $\tilde{\alpha}_{i,j}(t)$ to $\mathbf{R}^2$:

$$\alpha_{i,j}(t) = \begin{bmatrix} x_{i,j}(t) & y_{i,j}(t) \end{bmatrix} = \begin{bmatrix} \dfrac{u_{i,j}(t)}{w_{i,j}(t)} & \dfrac{v_{i,j}(t)}{w_{i,j}(t)} \end{bmatrix}. \tag{4}$$

All elliptical arcs are rational quadratics with $w_{i,j,0} = w_{i,j,2} = 1$ and $|w_{i,j,1}| < 1$. Note that standards do not typically support projective transformations, and use ad hoc representations for elliptical arcs.

Filled path definition. The points inside filled paths are chosen by the non-zero or even-odd winding rules. The PDF standard offers:

> *The non-zero winding number rule determines whether a given point is inside a [filled] path by conceptually drawing a ray from that point to infinity in any direction and then examining the places where a segment of the path crosses the ray. Starting with a count of 0, the rule adds 1 each time a path segment crosses the ray from left to right and subtracts 1 each time a segment crosses from right to left. After counting all the crossings, if the result is 0, the point is outside the path; otherwise, it is inside.* (4.4.2)

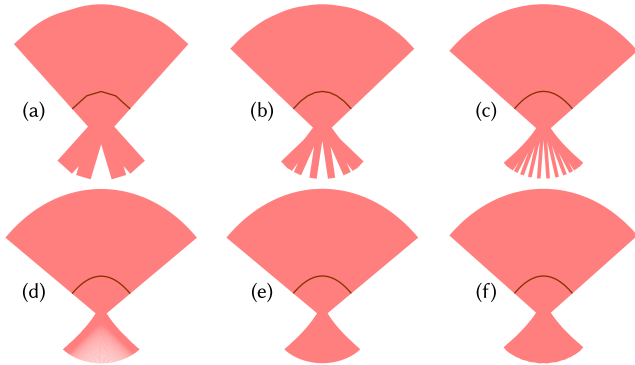

Fig. 4. (a,b,c) Stroked approximation of a quadratic segment by 4, 8, and 16 linear segments. Without inner joins, the gaps are clearly visible. (d) They are still visible at 256 linear segments, and will always remain so. (e) In contrast, stroking the quadratic segment leaves no gaps. (f) Adding inner joins moves (c) much closer to (e). Inner joins are needed if stroking linear approximations are to become, in the limit, equivalent to stroking curves.

The non-zero winding rule (figure 2a) prevents overlapping outlines from cancelling each other, as long as they are consistently oriented. This is key to rendering potentially overlapping text in a single pass and, as we will see, the design of global stroke-to-fill algorithms.

The even-odd winding rule (figure 2b) considers the points for which the number of crossings is odd (rather than non-zero) to be inside. This is easier to render, since it does not require a counter. Most standards support both alternatives.

Stroked path definition. The standards do not completely define how paths are to be stroked. The PDF [2006] standard says:

> *The line width parameter specifies the thickness of the line used to stroke a path. It is a non-negative number expressed in user space units; stroking a path entails painting all points whose perpendicular distance from the path in user space is less than or equal to half the line width.* (4.3.2)

OpenXPS [2009] proposes something equivalent:

> *Contours*[1] *and dashes SHOULD be rendered so that they have the same appearance as if rendered by sweeping the complete length of the contour or dash with a line segment that is perpendicular to the contour and extends with half its length to each side of the contour. All points covered by the sweep of this perpendicular line are part of the dash or contour.* (18.6)

Stroking consecutive segments independently causes a gap to appear whenever they meet at an angle (figure 3a). Most standards allow for a selection of *joins* that close these gaps (figures 3b, 3c, and 3d). *Caps* can also be added to the starts and ends of dashes and outlines (figures 3h, 3i, and 3j). Enabling decorative joins and caps is a key motivation for using perpendicular directions in these definitions.

Unfortunately, the standards make no mention of *inner joins*. In figures 3b, 3c, and 3d, the inner joins (in purple) are completely hidden by the stroked segments. However, this isn't always the case. Figures 3e, 3f, and 3g show an example where they are partially visible (in blue). Although the example seems contrived, it isn't. Without inner joins, stroking a curved segment can produce different results

[1]Here, contour is a synonym for outline.

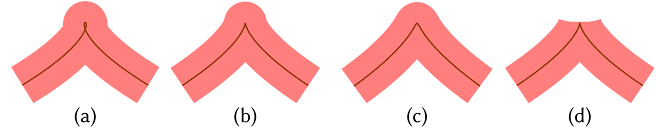

Fig. 5. Arbitrarily small changes to the input path can cause a cusp to appear or disappear. A cubic loop (a) moves through a cusp (b) in its transition to a serpentine (c). Following the standard to the letter results in a gap (d) when stroking an exact cusp. Continuity between input and output—and the principle of least astonishment—requires output (b) instead.

from stroking a piecewise-linear approximation of the same curved segment, no matter how close the approximation. Figures 4a, 4b, and 4c show the stroking of progressively finer piecewise-linear approximations of a parabolic arc. Even at 256 segments (figure 4d), gaps are still visible. They will never disappear. As figure 4e shows, the result of stroking the parabolic arc leaves no such gaps. Adding inner joins makes the approximation behave as expected (figure 4f).

The standards also omit any discussion of cusps within segments. Although rare, cusps can appear in non-degenerate cubics. They are more frequent in parabolic arcs, elliptical arcs, and cubics that have collapsed into lines because of singular geometric transformations. The tangent direction is well defined at the cusps, and therefore so is the perpendicular direction. Yet, as figure 5d shows, the results of stroking a cusp can be surprising. That particular cusp is midway in the transition between the loop in figure 5a and the serpentine in figure 5c. We argue that the result shown in figure 5b is more intuitive, since it maintains continuity between input and output.

To allow for full input-output continuity, joins and caps must be round. Then, the PDF standard can avoid perpendicular distances in its definition and the OpenXPS standard can sweep the outline with a disk, rather than a perpendicular segment. Without these changes, this type of continuity is desirable but not always achievable. For example, it is impossible to eliminate discontinuities when cusps happen at segment endpoints. We need to compromise.

Motivated by the *principle of least astonishment*, we propose to strengthen the standards with two continuity requirements:

(1) The sequence that results from stroking progressively finer piecewise-linear approximations to a curve should converge to the result of stroking the curve itself;
(2) Whenever possible, the result of stroking an outline should be continuous with regard to changes in the input path, the line width, and the dash pattern.

Indeed, users are likely to be surprised whenever an arbitrarily small change in the input path produces a large change in set of painted points. In a similar fashion, they will be disappointed if, no matter how closely a flattened input path approximates a curve, the painted points never approach those painted by stroking the curve itself.

Dashing. Before it is stroked, a *dash pattern* $d_1, d_2, \ldots, d_{2\ell-1}, d_{2\ell}$, where $d_i \in \mathbf{R}_{\geq 0}$ are in units of the line width, $i \in \{1, \ldots, 2\ell\}$, can be applied to the path. This has the effect of cutting the outlines into pieces, or *dashes*, whose *arc-lengths* cycle over the values d_i. The even-numbered dashes are then discarded from the outline. This dashing process can start at an arbitrary *initial phase*. It can continue directly from one disconnected outline to the next, or have this initial *phase reset* between outlines.

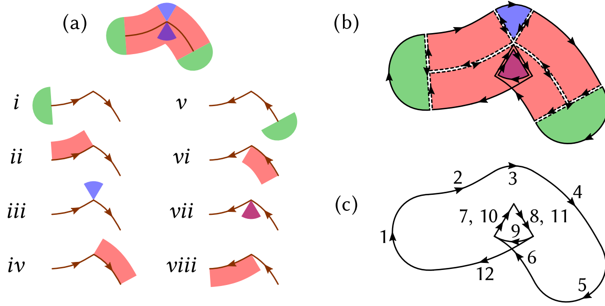

Fig. 6. (a) Local algorithms decompose the stroked region into the union of multiple filled paths. (b) Many segments appear in two different paths, in opposite orientations. (c) Global algorithms combine all these paths together into a single outline that omits these repeated segments.

The stroke-to-fill conversion problem. We can now precisely state the role of a global stroke-to-fill conversion algorithm. Given a line width, join style, cap style, and dash pattern:

> *Find a single output path that, when filled, contains the same points as those that would be painted by stroking the input path.*

Local algorithms solve a simpler problem:

> *Find a list of output paths, the union of which, when filled, contains the same points as those that would be painted by stroking the input path.*

A secondary goal is to produce the smallest possible number of segments in the output for a given target accuracy.

3 STROKE-TO-FILL CONVERSION

Figure 6 shows the operation of an incipient stroking algorithm as it converts an input outline to be stroked into equivalent output paths to be filled. Its underlying principle is to go over the input outline forwards and then backwards, outputting independent filled paths that cover the same points as would be painted by the stroking of each outline piece. Figure 6a shows the resulting decomposition. On the way forward, the stroker outputs filled paths for the cap, the top part of the first input segment, the outer join, and top part of the second input segment. On the way backward, it handles the cap, the bottom part of the second input segment, the inner join, and finally the bottom part of the first input segment.

Many local implementations are variations of this basic idea. For example, the top and bottom parts of stroked segments could be merged into a single filled path. The outer join could also be merged in. The algorithm could detect when the inner join is fully covered by the adjacent segments, and omit the corresponding filled path. The entire process could be completed in a single pass over the input. Setting aside for the moment the problem of producing these pieces, it should be clear that the approach works, at least in the local sense (i.e., with the help of a stencil buffer).

The same idea works globally as long as the pieces are oriented consistently (e.g., clockwise) and the resulting combined path is filled with the non-zero winding rule. In this latter scenario, two output pieces often include the same outline parts, but traversed in opposite directions. Figure 6b shows how these parts cancel each other. The parts that remain can be reconnected into a single output

Table 1. The stream elements in stages (I)nput, (O)utput, (R)egularization, (D)ecoration, (F)orward & backward, and (T)hickening.

Element	Arguments	Stages
begin outline	p	I,O
end open/closed outline	p	I,O
segment	α	I,O
begin regular outline	p, d	R–D
end regular open/closed outline	p, d	R–D
begin/end segment piece	p, d	R–D
segment piece	$\alpha, [a, b]$	R–T
degenerate segment	p_0, d, p_1	R–T
cusp	d_0, p, d_1, ω	R–T
inflection parameter	t	I–T
offset/evolute cusp parameter	t	R–T
join tangent/vertex parameter	t	D
begin/end dash parameter	t	D–T
initial/terminal cap	p, d	D–T
initial/terminal butt cap	p, d	D–T
join	d_0, p, d_1, ω	D–T
inner cusp/join	d_0, p, d_1, ω	D–T
backward initial/terminal cap	p, d	F–T
backward initial/terminal butt cap	p, d	F–T
backward begin/end dash parameter	t	F–T

$p \in \mathbf{R}^2$, d, $d_i \in \mathbf{R}^2$ are tangent directions. $[a, b]$ is the parameter interval for the piece of segment α. ω is the winding number inside an inner cusp or join (see section 3.2).

outline. A typical global implementation follows figure 6c. Caps, segments, and outer joins need only add their exterior boundaries to the output path. Inner joins are more complicated, but still fit into this scheme (see section 3.2). As a result, global implementations produce fewer output segments.

Our implementation. Our stroking algorithm is implemented as a chain of filters. Each filter performs a simple task and forwards its results to the next filter along the chain. The effect of all filters combines to transform the input stream describing a path to be stroked into an output stream containing an equivalent path to be filled. This separation reduces implementation complexity and increases code reuse without incurring the price of a multi-pass algorithm. The high-level decomposition of the chain is as follows:

input path → *regularization* → *decoration* →
→ *forward & backward* → *thickening* → output path

The *regularization* stage (section 3.1) identifies regular segment pieces where the filters that follow can safely compute tangents. The *decoration* stage (section 3.2) transforms the stream into dashed outlines annotated with joins and caps. These outlines do not yet have any “thickness”. The filter chain then follows the blueprint for global algorithms discussed in figure 6. The *forward & backward* filter sends each dash twice down the chain, first traversing it in the forward direction, then backward. The *thickening* stage (section 3.3) completes the conversion by outputting an outline for each join, cap, and thick segment piece, cancelling the parts that would appear in opposite orientations.

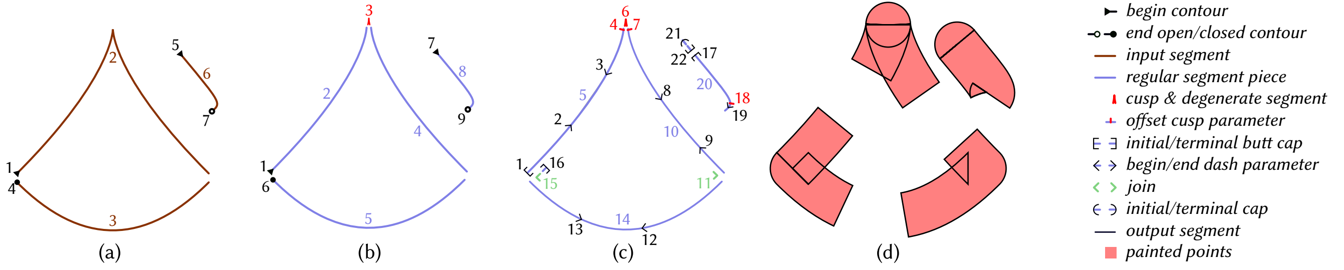

Fig. 7. Overview of algorithm. (a) The input path to be stroked contains a closed outline (1–4) and an open outline (5–7). (b) The regularization stage identifies regular segment pieces (2,4,5,8) and a cusp (3). (c) Decoration finds the parameters where dashes start and end (2,3,8,9,12,13,19), adds caps (1,16,17,21,22) and joins (11,15), and identifies the offset cusp parameters (4,7,18). Due to these additions, segment pieces 2,4,5,8 in (b) appear as 5,10,14,20 in (c). The omitted forward & backward stage traverses each dash in the decorated output in both directions, reversing elements on the way back (e.g., 9,10,11,12,14,ƧƖ,ᔭƖ,ƖƖ,ɘ,0Ɩ). (d) Finally, the dashes are fed one by one into the thickening stage, which outputs the equivalent path to be filled. Certain elements, such as begin/end segment piece markers and evolute cusp parameters, were omitted to avoid overcrowding.

Figure 7 shows the stages and table 1 lists the stream elements. The input stream contains a typical path (section 2). Outlines are delimited by begin and end markers. The end marker tells if the outline is open or closed. The elements between markers describe a connected sequence of segments given by Bézier control points. Regularization identifies regular subintervals within each segment (*segment piece*) and inserts cusps and degenerate segments between them. The segment pieces and outlines are surrounded by begin and end markers that carry the tangents at the endpoints. The stream can contain parameters (e.g. *begin dash parameter*, *offset cusp parameter* etc) that refer to the first segment or segment piece that follows. The decoration stage applies the dashing pattern by attaching the parameters values that begin and end each dash. It adds joins between consecutive segment pieces, differentiates inner/outer cusps and inner/outer joins, and caps the outlines. When traversing the stream backwards, the *forward & backward* stage marks key elements as being reversed, flips between outer joins/cusps and inner joins/cusps, maintains the order of parameters relative to the segments they refer to, and adjusts their arguments appropriately. Finally, the thickening stage outputs a standard path.

Notation and properties. The unit tangent T and normal N to a regular parametric plane curve α are

$$T(t) = \frac{\alpha'(t)}{|\alpha'(t)|} \quad \text{and} \quad N(t) = T(t)^\perp, \text{ where } \begin{bmatrix} x & y \end{bmatrix}^\perp = \begin{bmatrix} -y & x \end{bmatrix}. \tag{5}$$

The signed *curvature* κ and the *radius of curvature* ρ of α are

$$\kappa(t) = \frac{\langle \alpha'(t)^\perp,\ \alpha''(t)\rangle}{|\alpha(t)|^3} \quad \text{and} \quad \rho(t) = \frac{1}{\kappa(t)}. \tag{6}$$

The *evolute* or *central curve* of a curve α (the locus of its centers of curvature) is

$$e(t) = \alpha(t) + \rho(t)\,N(t), \quad \text{and its derivative is} \tag{7}$$

$$e'(t) = \rho'(t)\,N(t). \tag{8}$$

The *offset* or *parallel curve* to a curve α at distance h is

$$\alpha_h(t) = \alpha(t) + h\,N(t), \quad \text{and its derivative is} \tag{9}$$

$$(\alpha_h)'(t) = \big(1 - h\,\kappa(t)\big)\,\alpha'(t). \tag{10}$$

In this paper, the distance h is half the line width used for stroking.

3.1 Regularization

The conversion of strokes to fills requires the approximation of offset curves (section 3.3). This approximation process only works when the offset is smooth. Equation (9) shows that the offset is well defined whenever the normal to the input segments is well defined. Equation (10) indicates it is smooth whenever the input segments are smooth and have bounded curvature. The regularization stage imposes these conditions on the input.

Well-defined normals. The normal direction is defined in terms of the tangent direction. In the interior of a segment α, the tangent direction is given by a slope $r(t) : s(t)$, where r and s are polynomials:

$$r(t) : s(t) = x'(t) : y'(t). \tag{11}$$

If the segment is rational, the $w^2(t)$ in the denominators cancel and

$$r(t) : s(t) = w(t)\,u'(t) - u(t)\,w'(t) : w(t)\,v'(t) - v(t)\,w'(t). \tag{12}$$

The ratios seem undefined when $r(t) = s(t) = 0$. However, by L'Hôpital's rule, we can then compute the slope by the ratio between the factors' derivatives. This process either eventually ends with a well-defined slope, or all derivatives vanish. The latter can only happen when $r(t)$ and $s(t)$ are identically zero. In turn, this implies that x and y are constant, or u and v are multiples of w. Either way, the segment must degenerate to a point.

These degenerate segments do not need to be stroked, but they can wreak havoc in the orientation of joins and caps. They must be eliminated. Once this is done, L'Hôpital's rule enables us to compute perpendicular directions in every other case; at least in theory.

In practice, due to finite precision arithmetic, each computation carries a rounding error. This can cause serious complications. The standard model of arithmetic [Higham 2002, section 2.2] guarantees, for all basic arithmetic operations, that

$$\text{fl}(x \bullet y) = (x \bullet y)(1 + \epsilon), \quad |\epsilon| \le \mathbf{u}, \quad \bullet = +, -, \times, \div. \tag{13}$$

Here, $\text{fl}(x \bullet y)$ denotes the arithmetic operation $x \bullet y$ performed in floating-point. The machine-epsilon is $\mathbf{u} = 2^{-24}$ or 2^{-53}, for single or double precision, respectively. Thus, the evaluation of polynomials r and s, no matter how careful, will always contain errors. In the vicinity of cusps, or near-cusps as those in figure 5, these errors can destroy the accuracy of the slope.

Given a degree-d polynomial r expressed in Bernstein basis:

$$r(t) = \sum_{k=1}^{d} r_k\, b_{k,d}(t), \quad \text{for } t \in [0,1], \tag{14}$$

Jiang et al. [2010] give the following forward error bound for its evaluation by de Casteljau's method:

$$\left|r(t) - \mathrm{fl}\big(r(t)\big)\right| \leq \gamma_{2d}\, \tilde{r}(t), \tag{15}$$

where

$$\gamma_n = \frac{n\mathbf{u}}{1 - n\mathbf{u}} \quad \text{and} \quad \tilde{r}(t) = \sum_{k=1}^{d} |r_k|\, b_{k,d}(t). \tag{16}$$

Their compensated algorithm achieves the smaller error bound of

$$\left|r(t) - \mathrm{fl}\big(r(t)\big)\right| \leq \mathbf{u}\left|r(t)\right| + 2\gamma_{3d}^2\, \tilde{r}(t). \tag{17}$$

(Similar bounds apply to factor s, of course.)

L'Hôpital's rule is valid only at the cusp, not in its general vicinity. There, although $r(t)$ and $s(t)$ must be close to zero, $\tilde{r}(t)$ and $\tilde{s}(t)$ most likely are not. Regardless of the polynomial evaluation algorithm, numerical errors may leave us with no confidence in the slope. This can happen even when there is no cusp, as in figures 5a and 5c.

Therefore, even for non-degenerate segments, there may exist subintervals in the parameter range in which they are *effectively irregular*: the numerical computation of the slope becomes unstable. These subintervals must be marked for special treatment.

From the forward error bounds in (15), we know that

$$\mathrm{fl}\big(r(t)\big) - \gamma_{2d}\, \tilde{r}(t) \leq r(t) \leq \mathrm{fl}\big(r(t)\big) + \gamma_{2d}\, \tilde{r}(t) \quad \text{and} \tag{18}$$

$$\mathrm{fl}\big(s(t)\big) - \gamma_{2d}\, \tilde{s}(t) \leq s(t) \leq \mathrm{fl}\big(s(t)\big) + \gamma_{2d}\, \tilde{s}(t). \tag{19}$$

These inequalities define a rectangle in the r-s coordinate system where the true values for $r(t)$ and $s(t)$ are known to be. The slope uncertainty is the angle θ subtended by this rectangle from the point of view of the origin. We mark as problematic the subintervals for which this angle is larger than a maximum allowed slope error $\theta_{\max}$ (figure 8a). To simplify the problem, we use the angle ϕ subtended by the rectangle's circumcircle as an upper bound to θ (figure 8b). The effectively irregular intervals are those for which:

$$\sin\left(\tfrac{1}{2}\phi\right) = \frac{\gamma_{2d}\sqrt{\tilde{r}^2(t) + \tilde{s}^2(t)}}{\sqrt{\mathrm{fl}\big(r(t)\big)^2 + \mathrm{fl}\big(s(t)\big)^2}} > \sin\left(\tfrac{1}{2}\theta_{\max}\right). \tag{20}$$

Although we cannot solve this inequality exactly, we can obtain meaningful results by solving this polynomial inequality instead:

$$\gamma_{2d}^2\big(\tilde{r}^2(t) + \tilde{s}^2(t)\big) - \sin^2\left(\tfrac{1}{2}\theta_{\max}\right)\big(r^2(t) + s^2(t)\big) > 0. \tag{21}$$

In this work, we replace γ_{2d} with the more conservative $32\mathbf{u}$ and set $\theta_{\max} = 0.5°$. We also prevent the polynomial $\tilde{r}^2(t) + \tilde{s}^2(t)$ from having any roots for $t \in [0,1]$. To do so, we simply enforce that its first and last Bernstein coefficients are greater than $32\mathbf{u}$. This works because the other coefficients are non-negative and all Bernstein basis polynomials are positive in $(0,1)$.

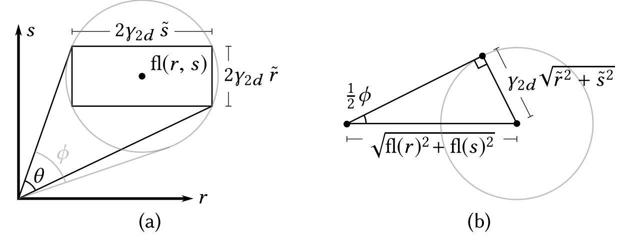

Fig. 8. The computation of tangent directions is subject to uncertainties. (a) The true tangent can fall anywhere in the rectangle, which subtends an angle θ. (b) The angle ϕ subtended by the circumcircle is an upper bound to θ, and is much easier to handle.

Bounded curvature. Looking at equation (10), we see that offset smoothness requires $\alpha'(t) \neq 0$ and a bounded curvature. Outside of subintervals where (21) holds, we already know that $\alpha'(t) \neq 0$. To bound the curvature, we note that, when it is sufficiently large in magnitude, the offset can be approximated by a circular arc. Let τ be the approximation tolerance for the offset. We mark as effectively irregular the subintervals for which

$$\left|\rho(t)\right| < \tau. \tag{22}$$

Root finding. The most robust strategy for identifying irregular intervals is to use interval analysis [Snyder 1992]. However, we obtained satisfactory results by simply finding the roots of (21) and (22) as equations, and then tallying the inequalities at a few parameter values in each subinterval delimited by these roots.

An extremely simple algorithm is very effective in finding all roots (with multiplicity) of polynomials $p \in \mathbf{R}[t]$ of moderate degree for $t \in [a,b]$. Let $t_1, \ldots, t_k$ be the roots of p' in $[a,b]$, and define $t_0 = a$ and $t_{k+1} = b$. Since p is monotonic in each of the intervals $[t_i, t_{i+1}]$, for $i \in \{0, \ldots, k\}$, each interval can contain at most one root of p. Since p is continuous, the intervals that contain roots are those for which $p(t_i)\,p(t_{i+1}) \leq 0$. Within these intervals, we find the root using a fail-safe hybrid between bisection and Newton–Raphson's algorithm [Press et al. 2007, section 9. 4]. Since $\deg(p') = \deg(p) - 1$, this recursive algorithm will eventually reach a linear polynomial, which we solve explicitly. We keep all polynomials in Bernstein basis. The *convex-hull* property of this basis can be used to abort the recursion early when it implies there are no roots in the interval.

Regularization filters. The regularization stage is divided into three filters:

regularization = close outlines ⟶ identify irregularities ⟶
⟶ orient degeneracies

The *close outlines* filter simply adds a linear segment connecting the last and first control points in closed outlines, when needed.

The second filter, *identify irregularities*, is responsible for finding the regular pieces in each segment, connecting them by degenerate segments and cusps, and surrounding them with markers that carry the endpoint tangents. The endpoint tangents are used by the *orient degeneracies* filter that follows.

A degenerate segment is defined by an initial control point, a tangent direction, and a final control point. It behaves like a linear segment, except that its tangent direction is given explicitly, rather

than being computed from control points. A cusp is defined by an initial tangent direction, a control point, and a final tangent direction. It behaves like a round join placed at the given control point.

Let $t'_1, \dots, t'_{k'}$ be the sorted roots of (21) and (22) as equations for a given input segment α. These roots partition the interval $[0, 1]$ into subintervals that may or may not satisfy each of (21) and (22). Those that satisfy either inequality are effectively irregular; the others are regular. Let $t_1, \dots, t_k$ be the boundaries between these effectively regular and irregular subintervals, sorted, and set $t_0 = 0$ and $t_{k+1} = 1$. From these subintervals, the *identify irregularities* filter produces a list of segment pieces that will replace each segment α in the stream.

When $k = 0$ and the interval $[0, 1]$ is effectively irregular, the list contains a single *invalid* degenerate segment $\big(\alpha(0), (0, 0), \alpha(1)\big)$. Otherwise, subintervals are processed in order, for $i \in \{0, \dots, k\}$. Regular subintervals $[t_i, t_{i+1}]$ cause the segment piece $\big(\alpha, [t_i, t_{i+1}]\big)$ to be added to the list, surrounded by the tangents $\big(\alpha(t_i), \alpha'(t_i)\big)$ and $\big(\alpha(t_{i+1}), \alpha'(t_{i+1})\big)$. The first and last subintervals are special when effectively irregular. If t_0 satisfies (21), $[t_0, t_1]$ adds the degenerate segment $\big(\alpha(t_0), \alpha'(t_0), \alpha(t_1)\big)$. Otherwise it adds $\big(\alpha(t_0), \alpha'(t_1), \alpha(t_1)\big)$. Subinterval $[t_k, t_{k+1}]$ works the same way, giving preference to the endpoint tangent at t_{k+1}. Interior subintervals $[t_i, t_{i+1}]$ that are effectively irregular add to the list a cusp $\big(\alpha'(t_i), \alpha(t_i), \alpha'(t_{i+1})\big)$ followed by a degenerate segment $\big(\alpha(t_i), \alpha'(t_{1+1}), \alpha(t_{i+1})\big)$.

In the example of figure 7a, segment 3 contains a cusp. The *identify irregularities* filter replaces it by the sequence 2,3,4 in figure 7b.

After *identify irregularities*, the stream can still contain invalid degenerate segments in the form $\big(p_1, (0, 0), p_2\big)$. The filter *orient degeneracies* follows the algorithm proposed by the SVG [2011] standard to obtain valid tangents directions for these segments:

> *[...] In these cases, the following algorithm is used to establish directionality: to determine the directionality of the start point of a zero-length path segment, go backwards in the path data specification within the current subpath until you find a segment which has directionality at its end point (e.g., a path segment with non-zero length) and use its ending direction; otherwise, temporarily consider the start point to lack directionality. Similarly, to determine the directionality of the end point of a zero-length path segment, go forwards in the path data specification within the current subpath until you find a segment which has directionality at its start point (e.g., a path segment with non-zero length) and use its starting direction; otherwise, temporarily consider the end point to lack directionality. If the start point has directionality but the end point doesn't, then the end point uses the start point's directionality. If the end point has directionality but the start point doesn't, then the start point uses the end point's directionality. Otherwise, set the directionality for the path segment's start and end points to align with the positive x-axis in user space.* (F.5)

To do so, it processes each outline in turn. By visiting the markers that begin and end each segment piece, it propagates the endpoint tangents forward, then backward.

Past these two filters, tangent directions can be safely computed without further concern for numerical stability. In other words, the stroking of the path has been precisely defined.

3.2 Decoration

The decoration stage is divided into two filters:

$$\textit{decoration} = \textit{dash, join \& cap} \rightarrow \textit{classify joins \& cusps}$$

The *dash, join & cap* filter adds markers where caps and joins should be inserted into the stream and applies the dashing pattern. The stroking style defines the dashing pattern, the initial phase, and a potential phase reset between outlines. Before starting, the filter uses the initial phase and the dash pattern to compute the initial dash index and the length needed by the initial dash. As segments are processed, the filter maintains a current dash index and the length needed by the current dash.

Dashing is handled by a procedure invoked as each segment piece arrives (see the listing in appendix A). This procedure first checks whether any part of the segment is visible after dashing. If so, it emits the parameters where dash transitions happen within the piece (if any) and then emits the segment piece itself. Otherwise, it simply consumes the length of the segment piece while advancing the dash pattern. The dash parameter values emitted by the dashing procedure mark where dash caps must appear.

A join is added between every consecutive pair of visible segment pieces (see the listing in appendix B). This is handled by simple procedures invoked upon arrival of elements that begin and end each segment piece. When a segment piece ends, the filter saves its final tangent direction for future reference. When a new segment piece begins, it uses the saved final direction and the newly received initial direction to emit the join. If the style mandates a phase reset, the current dash index and length needed are overwritten with the corresponding initial values whenever a new outline begins.

Outline caps are trickier to handle (also in appendix B's listing). The endpoints of an outline may have been eliminated during the dashing process. In an open outline, caps must be added to the endpoints that remain visible. If an outline is closed and both endpoints are visible, the caps are omitted and a closing join connects the end of the outline to its start (elements 15 and 16 of figure 7c). However, this information is only available when the element that ends the outline is processed. Therefore, when processing the begin outline element, the filter emits an initial butt cap. Then, when the outline's end is processed, after the terminal cap or join are added, a separate outline containing only the initial cap is added, if needed (elements 20 and 21 of figure 7c).

Computing arc lengths. The arc-lengths and matching parameters used in appendix A can only be obtained numerically. An algorithm by Farouki [1997] and generalized by Jüttler [1997] gives excellent results. Given a curve α defined on an interval $[a, b]$, the procedure finds a continuous, monotonic, piecewise linear-rational function $t^*\colon [0, 1] \rightarrow [a, b]$ so the parameterization $\alpha \circ t^*$ has approximately constant speed. It proceeds by minimization:

$$t^* = \underset{t\colon [0,1]\rightarrow[a,b]}{\arg\min} \int_0^1 \left(\left|\tfrac{d}{du}\alpha\big(t(u)\big)\right| - S\right)^2 du \tag{23}$$

where $S = \int_a^b \left|\alpha'(t)\right| dt$ is the arc length of α in $[a, b]$. Surprisingly, given a partition of $[a, b]$ into a fixed number of sub-intervals as input, the minimization problem has a closed form solution for all our input segment types. As suggested by Jüttler [1997], we

use Gaussian quadrature [Press et al. 2007, section 4.6] to obtain the coefficients required by the minimizer, rather than the partial fractions method of Farouki [1997].

Join & cusp classification. The responsibility of the *classify joins & cusps* filter is to compute the parameter ω in the 4-tuples (d_0, p, d_1, ω) that define joins and cusps. This parameter is initially 0.

As shown in figure 3, a join has an outer and an inner part. When optimizing inner joins in section 3.3, it will be necessary to know if the unions of the regions filled when stroking the segments adjacent to the join completely covers the inner join, and even better if their intersection does. This turns out to be a challenging problem to solve, and we are forced to be conservative.

First, we consider the adjacent segments individually. We set ω to 1 if either stroked piece independently covers the inner join (which implies their union also does), and 2 if both stroked pieces independently cover the inner join (equivalently, their intersection does). Otherwise, ω is left at 0. Second, we only study inner round joins. This covers the case of inner bevel joins as well, since they are contained by their round counterparts. (We cannot see any use for inner miter joins.) In summary, we can only set ω to a value other than 0 if we can prove that the stroking of an adjacent segment piece covers the inner join entirely.

Let $\big(\alpha, [a, b]\big)$ be the second segment in the join (d_0, p, d_1) as in figure 9. Within this section, assume an arc-length parameterization for segment α. The perpendicular line segment that sweeps α is

$$\ell(t, s) = \alpha(t) + s\alpha'^{\perp}(t), \quad \text{for } s \in [-h, h]. \tag{24}$$

The region covered by stroking the segment piece $\big(\alpha, [a, t_1]\big)$ is

$$S(a, t_1) = \bigcup_{(t,s)\,\in\,[a,t_1]\times[-h,h]} \ell(t, s), \quad \text{the gray region in figure 9.} \tag{25}$$

Appendix C shows the proof for the following theorem:

Theorem 3.1. *Let t_{tan} be where $\ell(t_{tan}, s)$ first becomes tangent to the radius-h circle centered at $p = \alpha(a)$ (or $+\infty$ when there is no such parameter). If $t_1 \in [a, t_{tan}]$ and $\big|\rho(t_0)\big| > h$ for all $t_0 \in [a, t_1]$, then $S(a, t_1)$ covers the disk region between $\ell(a, s)$ and $\ell(t_1, s)$.*

From this theorem, it follows that:

Corollary 3.2. *Let $t_1 \in [a, b]$ be the such that $\ell(t_1, s)$ intersects the inner round join at a single point. If $\big|\rho(t_0)\big| > h$ for all $t_0 \in [a, t_1]$, then $S(a, t_1)$ covers the inner round join.*

Note the inner join is a convex region. As shown in figure 9, this single point of contact between line segment $\ell(t, s)$ and the inner join can happen when $\ell(t_{vtx}, s)$ goes through one of the inner join's vertices or when $\ell(t_{tan}, s)$ is tangent to the circle.

We can now describe the *classify joins & cusps* filter (which does *not* rely on an arc-length parameterization for α). Assume the join directions d_0 and d_1 have length 1 and refer back to figure 9. Using $p = \alpha(a)$ as the origin, vertex v in the inner join is

$$v = \sigma\, h\, d_0^{\perp}, \quad \text{where } \sigma = \operatorname{sign}\langle d_0^{\perp},\, d_1\rangle. \tag{26}$$

Parameter t_{vtx} for which $\ell(t_{vtx}, s)$ goes through v is a solution to

$$\big\langle \alpha(t) - v,\ \alpha'(t)\big\rangle = 0. \tag{27}$$

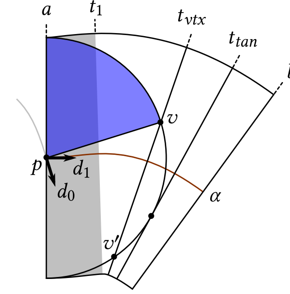

Fig. 9. Identifying if an inner join is covered by an adjacent segment piece.

The other intersection between $\ell(t_{vtx}, s)$ and the circle is

$$v' = v - 2\langle v,\, n\rangle\, n, \quad \text{where } n = \alpha'^{\perp}(t_{vtx})/\big|\alpha'(t_{vtx})\big|. \tag{28}$$

It is outside the inner join if

$$\sigma\,\langle v',\, d_1\rangle > 0 \ \vee\ \sigma\,\langle v',\, d_0\rangle < 0. \tag{29}$$

Parameter t_{tan} is a solution of

$$\big\langle \alpha(t) - p,\ \alpha'(t)/|\alpha'(t)|\big\rangle = \pm h. \tag{30}$$

The listing in appendix D shows how this machinery, together with corollary 3.2, is used to compute the parameter ω for a join. The filter simply counts the number of segment pieces adjacent to the join that completely cover the inner join (0, 1, or 2).

Not shown in the listing, the filter completes the classification by marking as inner joins and cusps those for which $\langle d_0^{\perp},\, d_1\rangle < 0$, and eliminating those for which $\langle d_0^{\perp},\, d_1\rangle = 0 \wedge \langle d_0,\, d_1\rangle > 0$.

3.3 Thickening

Following the blueprints in figures 6 and 7, the results of decoration are traversed forward and then backward and fed to the thickening stage. At this point, the stream consists of segment pieces (with associated parameters), caps, joins, cusps, and degenerate segments. The thickening filter must ensure that the corresponding regions are correctly filled (figure 6a). Many output segments can be omitted because they are common to two elements but appear in opposite directions (figure 6b). The result of thickening is a single output outline, whose interior contains the same points as those that would be painted by stroking the input outline (figure 6c).

As shown in figure 6b, the thickening of segments (*ii* and *iv*) adjacent to an outer join (*iii*) starts and ends at the segment's offset. In contrast, in the case of an inner join (*vii*), the thickened segments (*vi* and *viii*) start from the segment itself. To make the treatment of segments independent of join type, and to allow for the inner-join optimization described in the next paragraph, the additional line segments (7 and 11) that go from the segment endpoint to the offset are handled by the thickening of the inner joins. This leads to the simple convention that the thickening of each element starts at the offset of the element that precedes it, and ends at the offset of the element that follows.

Inner join optimization. Figure 10 shows how the join parameter ω, obtained by the listing in appendix D, described in section 3.2, is used to optimize the number of segments output by inner join elements. When $\omega = 0$, we are unsure if some part of the inner join is visible outside the region already covered by the stroking of its adjacent

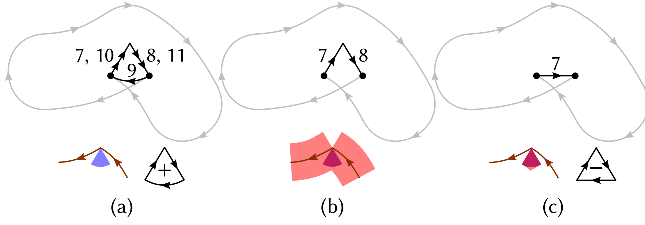

Fig. 10. Inner join optimization. (a) In general, drawing an inner join between two adjacent segment offsets requires 4 linear segments and 1 circular arc. (b) When the union of regions covered by the adjacent segments contains the inner join, we can connect the offsets using only 2 linear segments. (c) When the *intersection* contains the inner join, we can remove an additional triangle and connect the offsets directly with a single linear segment.

segments. Therefore, we are forced to include segments 7–11 of figure 10a. In contrast, when $\omega = 1$, we know that the entire inner join has already been covered. By omitting the closed outline that bounds the join, we can save 3 segments (figure 10b) without altering the set of points inside the output. When $\omega = 2$, we can introduce another copy of the inner join outline, in reverse orientation, and still keep its interior covered in the output. This saves yet another segment per inner join. However, it is even better to add a reverse triangle instead, as this produces the same interior while avoiding the reintroduction of the circular arc (figure 10c).

Thickening segment pieces with curves. Figure 6 seems to suggest that all it takes to stroke a segment piece is to connect its offset to the (reversed) segment by means of two linear segments. Although this is often true, it is *not always* true.

Figure 11a shows a segment piece $\big(\alpha, [s_i, s_{i+1}]\big)$ being thickened. Given a partition of the interval $[s_i, s_{i+1}]$, $\{t_0 = s_i, t_1, \ldots, t_n = s_{i+1}\}$, consider the quadrilaterals

$$\big(\alpha(t_k),\ \alpha_h(t_k),\ \alpha_h(t_{k+1}),\ \alpha(t_{k+1}),\ \alpha(t_k)\big),\ \ k \in \{0, \ldots, n-1\}.$$

Four of these quadrilaterals are shown in figure 11b: two for when the curve is traversed forward, two for when it is traversed backward. When all quadrilaterals are simple (i.e., they do not self-intersect), they are all oriented clockwise. Their shared edges cancel each other, and the area filled by their union is the same as the area filled by the polygon

$$\big(\alpha(t_0),\ \alpha_h(t_0),\ \alpha_h(t_1), \ldots,\ \alpha_h(t_n),\ \alpha(t_n),\ \alpha(t_{n-1}), \ldots,\ \alpha(t_0)\big).$$

Because α and α_h are smooth in the interval, the polyline that connects points in α and the polyline that connects points in α_h both converge to the curves themselves as we refine the partition:

$$\lim_{n\to\infty} \big(\alpha(t_0),\ \alpha(t_1), \ldots, \alpha(t_n)\big) = \big(\alpha, [s_i, s_{i+1}]\big) \quad \text{and} \tag{31}$$

$$\lim_{n\to\infty} \big(\alpha_h(t_0),\ \alpha_h(t_1), \ldots, \alpha_h(t_n)\big) = \big(\alpha_h, [s_i, s_{i+1}]\big). \tag{32}$$

Unfortunately, the quadrilaterals *may* contain self-intersections. The sides $\big(\alpha(t_k), \alpha(t_{k+1})\big)$ and $\big(\alpha_h(t_k), \alpha_h(t_{k+1})\big)$ are parallel in the limit. However, sides $\big(\alpha(t_k), \alpha_h(t_k)\big)$ and $\big(\alpha(t_{k+1}), \alpha_h(t_{k+1})\big)$ can continue to intersect all the way to the limit. In fact, in the limit, they intersect at the center of curvature for α. Looking at figure 11b, we see that, when the curvature is negative, these limiting intersections (the evolute e) and the offset are on opposite sides of segment α. Therefore, we only have to worry when traversing the curve in the positive curvature orientation. Then, the limit quadrilaterals will be simple if and only if $\rho(t) > h$ for $t \in [s_i, s_{i+1}]$.

This is the case shown in figure 11*a*. Since all other parts of the outline cancel out against the output of other elements, the only part left is 1) the offset of the segment piece.

Figure 11c shows the case where $0 < \rho(t) < h$ for $t \in [s_i, s_{i+1}]$. The offset α_h changes its orientation relative to figure 11a. Even though the area shown in gray must be stroked, it is left out when we naively follow the same procedure as we did when $\rho(t) > h$. This is what all curve-based strokers we analyzed do, and this is why they produce incorrect results. They blindingly output only 1) the offset of the segment piece.

Figure 11d explains the issue. The self-intersecting quadrilaterals can each be split into two triangles. Those with sides on the offset have a counter-clockwise orientation; the ones with sides on the segment are clockwise. They cancel out each other where they overlap, exactly over the gray area of figure 11c. The solution is to flip the orientation of the counter-clockwise triangles.

Figure 11e shows the two sets of triangles. Part of the common side between adjacent triangles cancels out. Once a gain, the polyline connecting points $\alpha(t_k)$ converges to $\big(\alpha, [s_i, s_{i+1}]\big)$. The polyline connecting points $\alpha_h(t_k)$ converges to $\big(\alpha_h, [s_i, s_{i+1}]\big)$, but traversed in the opposite direction. Critically, the polyline connecting the vertices c_k, the self-intersections of the quadrilaterals, converges to the evolute e:

$$\lim_{n\to\infty} (c_0,\ c_1, \ldots, c_n) = \big(e, [s_i, s_{i+1}]\big). \tag{33}$$

The regions covered by these triangles can be filled by the outlines shown in figure 11f. When these outlines are put together with the outlines coming from other elements, additional parts cancel out. Figure 11g shows what is left: 1) a linear segment $\big(\alpha_h(s_i), e(s_i)\big)$ that connects the offset to the evolute, 2) the evolute, 3) a linear segment $\big(e(s_{i+1}), \alpha_h(s_{i+1})\big)$ connecting the evolute to the offset, 4) the offset in the opposite direction, 5) linear segment $\big(\alpha_h(s_i), e(s_i)\big)$ again, 6) the evolute again, and 7) linear segment $\big(e(s_{i+1}), \alpha_h(s_{i+1})\big)$ again.

We can finally describe how to thicken a segment piece $\big(\alpha, [a, b]\big)$ using curves. Let $s_1, \ldots, s_m$ be the roots of

$$\rho(s) = h, \quad \text{for } s \in [a, b], \tag{34}$$

sorted in increasing order. Set $s_0 = a$ and $s_{m+1} = b$. These roots partition $[a, b]$ into subintervals. For each of the subintervals $[s_i, s_{i+1}]$, if $0 < \rho(s) < h$, then output the 7 parts shown in figure 11g. Else, output the offset $\big(\alpha_h, [s_i, s_{i+1}]\big)$. This is how the segment pieces in figure 7c are thickened to produce the result in figure 7d.

Approximating offsets and evolutes. In general, these curves are not representable exactly using the set of segment types allowed in output paths. The only exceptions are the offsets of linear segments and circles (which are closed under offsetting operations), and the evolutes of parabolic arcs (which are cubic segments). Elber et al. [1997] recommend a simple method by Tiller and Hanson [1984] for approximating offsets of quadratic curves and a more involved least-squares algorithm by Hoschek [1988] for the general case. This latter method was designed to compute G^1-continuous cubic Bézier approximations to offset curves, but can handle evolutes with trivial modifications.

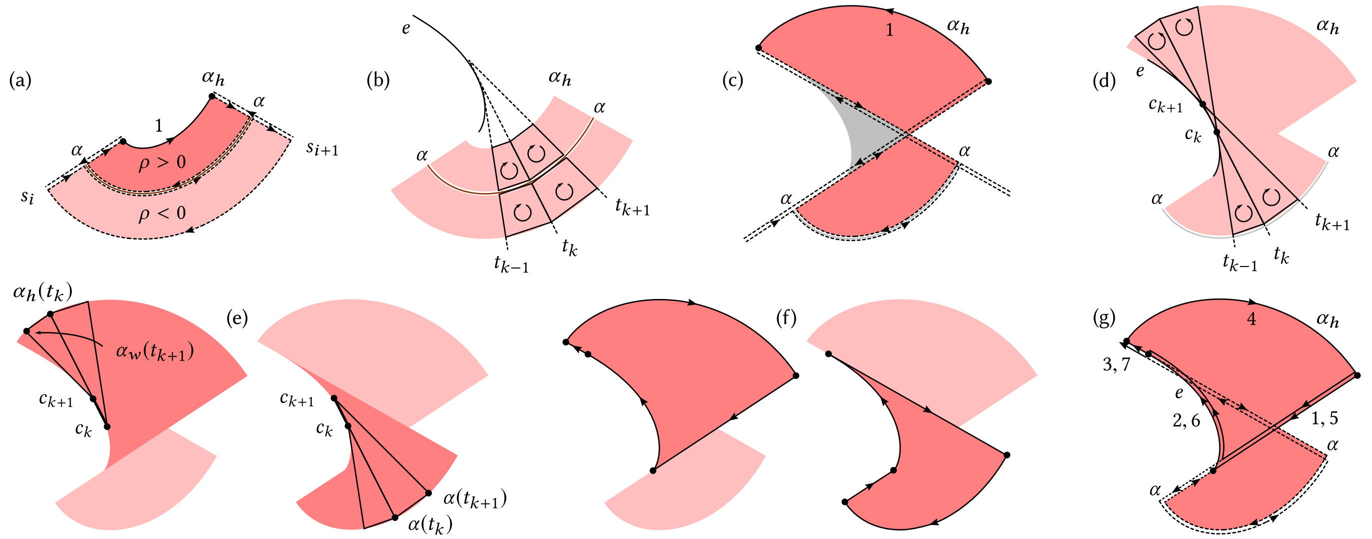

Fig. 11. Thickening segments. (a) When $\rho < 0$ or $\rho > h$ throughout a segment piece $(\alpha, [s_i, s_{i+1}])$, thickening the piece only requires outputting its offset $(a_h, [s_i, s_{i+1}])$. (b) This is because, in the limit, the quadrilaterals between the offsets and the segment in a progressively refined partition $[s_i, \ldots, t_{k-1}, t_k, t_{k+1}, \ldots, s_{i+1}]$ of $[s_i, s_{i+1}]$ do not self-intersect. (c) When $0 < \rho < h$, outputting the offset would leave the gray area missing. (d) This is because the triangles forming the self-intersecting quadrilaterals are traversed in opposite orientations. (e) The solution is to group triangles according to their orientations. (f) In the limit, two regions are filled. (g) When combined, they can still be filled by a single outline, but it now contains 7 parts. It traverses the offset in the opposite orientation, and traverses the evolute $(e, [s_i, s_{i+1}])$ twice.

Consider the approximation of a curve ξ in subinterval $[s_i, s_{i+1}]$, either an offset or an evolute, by a single cubic Bézier γ with control points p_0, p_1, p_2, and p_3. To interpolate endpoints, $p_0 = \xi(s_i)$ and $p_3 = \xi(s_{i+1})$. To enforce tangent directions, $p_1 = p_0 + \lambda_1 \xi'(s_i)$ and $p_2 = p_3 + \lambda_2 \xi'(s_{i+1})$. Thus, each choice of λ_1 and λ_2 defines a potential candidate $\gamma_{\lambda_1,\lambda_2}$ approximation. The method first samples the curve n times to produce $q_j = \xi\big(t_j^{(0)}\big)$, $s_i < t_j^{(0)} < s_{i+1}, j \in \{1, \ldots, n\}$. It then repeatedly alternates between two minimization stages:

$$\lambda_1^{(k)}, \lambda_2^{(k)} = \underset{\lambda_1,\lambda_2}{\arg\min} \sum_{j=1}^{n} \left|\gamma_{\lambda_1,\lambda_2}\big(t_j^{(k-1)}\big) - q_j\right|^2, \quad \text{and} \tag{35}$$

$$t_j^{(k)} = \underset{u\in[a,b]}{\arg\min} \left|\gamma_{\lambda_1^{(k)},\lambda_2^{(k)}}(u) - q_j\right|^2, \quad j \in \{1, \ldots, n\}. \tag{36}$$

The first minimization can be solved by linear least-squares. The second by equating the derivative to zero and solving with Newton–Raphson's method [Press et al. 2007, section 9.4]. Upon convergence, if the approximation is still too distant from the sampled points, ξ is split into two and the procedure is invoked recursively.

As stated, the method is not robust enough. Hoschek [1988] uses an uniform sampling $t_1^{(0)}, \ldots, t_n^{(0)}$ of $[s_i, s_{i+1}]$. This produces points that cluster around the zeros of ξ', causing the optimization to fail to approximate the desired curve. To avoid this problem, we force subdivision whenever the tangents at any two consecutive samples turn by a large angle. Cusps can also cause problems. As can be seen in equation (10), when ξ is the offset, $\xi'(t) = 0$ only if $\rho(t) = h$ or $\alpha'(t) = 0$. Since (s_i, s_{i+1}) is effectively regular and we break subintervals at $\rho(t) = h$, we know that $\xi'(t) \neq 0$ for $t \in (s_i, s_{i+1})$. However, when ξ is the evolute, $\xi'(t) = 0$ when $\rho'(t) = 0$, which could happen inside a subinterval. Therefore, to avoid these parameters, we also split the subintervals at $\rho'(t) = 0$. (Note that, in the process of solving for $\rho(t) = h$, we already solve for $\rho'(t) = 0$.) In other words, we avoid optimizing the approximation of offsets and evolutes across their cusps. We have also tried sampling the curve uniformly *in arc-length*, using the method by Jüttler [1997] described above. At a *significant* performance cost (often $> 10\times$), this makes the approximation robust enough.

Due to these difficulties with Hoschek's method, we recommend approximation by quadratic segments. The method implemented in Skia can be described as follows. First, split all subintervals at inflections. Then, the curve $\big(\xi, [s_i, s_{i+1}]\big)$ is either approximated by the quadratic $\gamma_{s_i,s_{i+1}}$ that has the same endpoints and tangents, or the interval is split in half at $s_m = \frac{1}{2}(s_i + s_{i+1})$, and each half approximated recursively. The quadratic approximation is accepted if all these conditions are met: $\xi'(s_i)$ and $\xi'(s_{i+1})$ are in opposite sides of $\xi(s_{i+1}) - \xi(s_i)$; the tangents at $\xi(s_i)$ and $\xi(s_{i+1})$ intersect at a well-defined point (the middle control-point of the candidate quadratic); the distance $\left|\gamma_{s_i,s_{i+1}}(s_m) - \xi(s_m)\right|$ or the distance between $\xi(s_m)$ and the single intersection between the normal at $\xi(s_m)$ and $\gamma_{s_i,s_{i+1}}$ are small enough. The subdivision continues until acceptance or until either: the subdivision limit is reached; the interval size $|s_{i+1} - s_i|$ is too small; the tangents are parallel and point in the same direction; or the distances from both endpoints to their opposite tangents are small enough. This method works well in practice, but occasionally stops subdivision too early due to the proximity between $\xi(s_m)$ and $\gamma_{s_i,s_{i+1}}$. We made the following modification to the method in Skia. Let s_p be such that $\xi'(s_p)$ is parallel to $\gamma'_{s_i,s_{i+1}}(s_m)$. We check the distance $\left|\gamma_{s_i,s_{i+1}}(s_m) - \xi(s_p)\right|$, instead of $\left|\gamma_{s_i,s_{i+1}}(s_m) - \xi(s_m)\right|$. This makes the approximation more robust to the potentially poor parameterization of ξ. With this modification, the approximation by quadratics is the default option in our implementation.

Thickening caps and outer joins poses no challenge whatsoever, so the algorithm is complete.

4 SURVEY OF EXISTING IMPLEMENTATIONS

To the best of our knowledge, there is no publication that directly attacks the stroke-to-fill conversion problem. Instead, there are myriad undocumented implementations. To address this problem, we present a survey of the current state of the art. Readers interested only in how these implementations compare against each other and against our method can skip directly to section 5.

Several software packages provide the functionality to convert strokes to fills through APIs: Microsoft's Direct2D, the Skia Graphics Library, Apple's Quartz, Anti-Grain Geometry, Java 2D (3 strokers), the livarot library used by Inkscape (2 strokers), and Qt 5. With some effort, we have managed to expose the internal implementations of additional open-source software packages: the OpenVG Reference Implementation, MPVG [Ganacim et al. 2014], MuPDF, the Cairo Graphics Library (3 strokers), and Ghostscript (3 strokers). We automated the process of obtaining renderings and output paths from Adobe Illustrator (2 strokers) using AppleScript. For completeness, we also produced renderings with NVpr [Kilgard and Bolz 2012] even though it does not convert strokes to fills prior to rendering.

In total, we analyzed 22 different third-party implementations. Our objective here is to show the breadth of their diversity, not to present a complete reverse engineering. For each method, we are interested in whether it respects the principle of least astonishment of section 2 with regard to inner joins and intra-segment cusps, on whether it deals with high-curvature areas and how, and on whether it honors accuracy requirements. Since the greatest variety comes from the flat strokers, we start with them.

4.1 Flat algorithms

These strokers flatten the input path during a pre-processing step. Traditionally, a curve is recursively subdivided until the chord that connects its endpoints is a good approximation to the curve [Catmull 1974, chapter 3]. Alternatively, the parameter is progressively advanced, and curve points at consecutive parameters are connected into a polyline [Hain et al. 2005]. All flat strokers we analyzed perform their own flattening. While thickening, they connect points obtained from the offsetting of the flattened input using line segments or circular arcs (possibly approximated by cubics or quadratics).[2] Vertices can be offset perpendicular to the input curve, as in figure 12a, or perpendicular to the linear segments in the approximation, as in figure 12b.

Although it is considerably easier to stroke a polyline than it is to deal with piecewise polynomial curves, the problem is *still* difficult. In flat strokers, the high-curvature issue manifests itself when the radii in figures 12a and b cross each other, as is the case in the figure. This situation must be identified for proper treatment. Many flat strokers fail near intra-segment cusps or completely ignore inner joins. Finally, all flat strokers we evaluated suffer from accuracy problems. Even those that are careful enough to approximate the offset fail to account for the approximation of the evolute.

[2] Oddly, on some strokers, the decision of using arcs is controlled by the active join type. It shouldn't be, since joins are for decoration *between* input segments, and should have no bearing on the approximation of the offset *within* them.

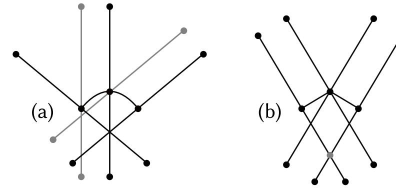

Fig. 12. Flat strokers can offset in the direction of the input curve (a), or in the direction of the piecewise linear approximation (b).

To illustrate the diversity in flat strokers, we will show how they would stroke the flattening of the parabolic arc of figure 12a into the two line segments in figure 12b. The results are shown in figure 13.

Ghostscript 9.26 ships with 3 different strokers, selected by arcane considerations over the target output device. Figure 13 *gs* shows the best one. It is a single-pass algorithm that outputs two outlines per input linear segment. One covers the segment and the outer join to the next segment, while the other covers the visible part of the inner join, if any. Since outlines are oriented consistently, they can be filled globally as a single path with non-zero winding, or streamed with even-odd winding into a stencil, as originally intended. This is a fairly robust stroker. Surprisingly, the *gs compat* stroker produces incompatible results. It outputs one outline to cover the segment and another for the outer join, but ignores inner joins. We have argued here that inner joins are desirable even when the original input consists exclusively of linear segments. When the input contains curves that were flattened by the stroking process, they are unquestionably mandatory. The last stroker, the global *gs fast*, outputs one outline for each input outline. It ostensibly goes over the flattened input forwards and then backwards. When processing each linear segment, it outputs the offset segment and then joins it to the offset of the segment that follows. This fails because the inner offset will be traversed in the opposite orientation to the outer offset whenever the perpendicular radii, which emanate from the control points of the flattened curve, cross each other. None of these strokers attempt to identify intra-segment cusps. However, joining the offsets of adjacent segments with a round join, as *gs* does, is significantly more robust than connecting them with line segments, as *gs compat* and *gs fast* do.

MuPDF 1.14 uses essentially the same algorithm as *gs fast*, and therefore has the same failure modes. It does not generate an output path, but rather a disconnected list of edges that are passed directly to an active-edge-list polygon filler [Newman and Sproull 1979]. This can be done in a single pass over the outlines.

The Cairo Graphics Library 1.17 comes with 3 different strokers. The local *cairo traps* stroker is analogous to *gs compat*, whereas the global *cairo polygon* is similar to *mupdf*. They fail, respectively, for the same reasons. The notable difference here is that *cairo traps* and *cairo polygon* use tangents to the input curve before flattening. The last stroker, *cairo tristrips*, is still a work in progress and ships disabled. It produces two triangles for each segment, which, with proper orientation, can be painted as a single path or streamed into a stencil. It fails only due to the careless treatment of cusps (a problem it shares with the other two algorithms.)

When there is dashing, Inkscape's "Stroke to Path" facility uses livarot library's `Path::Stroke` method. This is a global flat stroker.

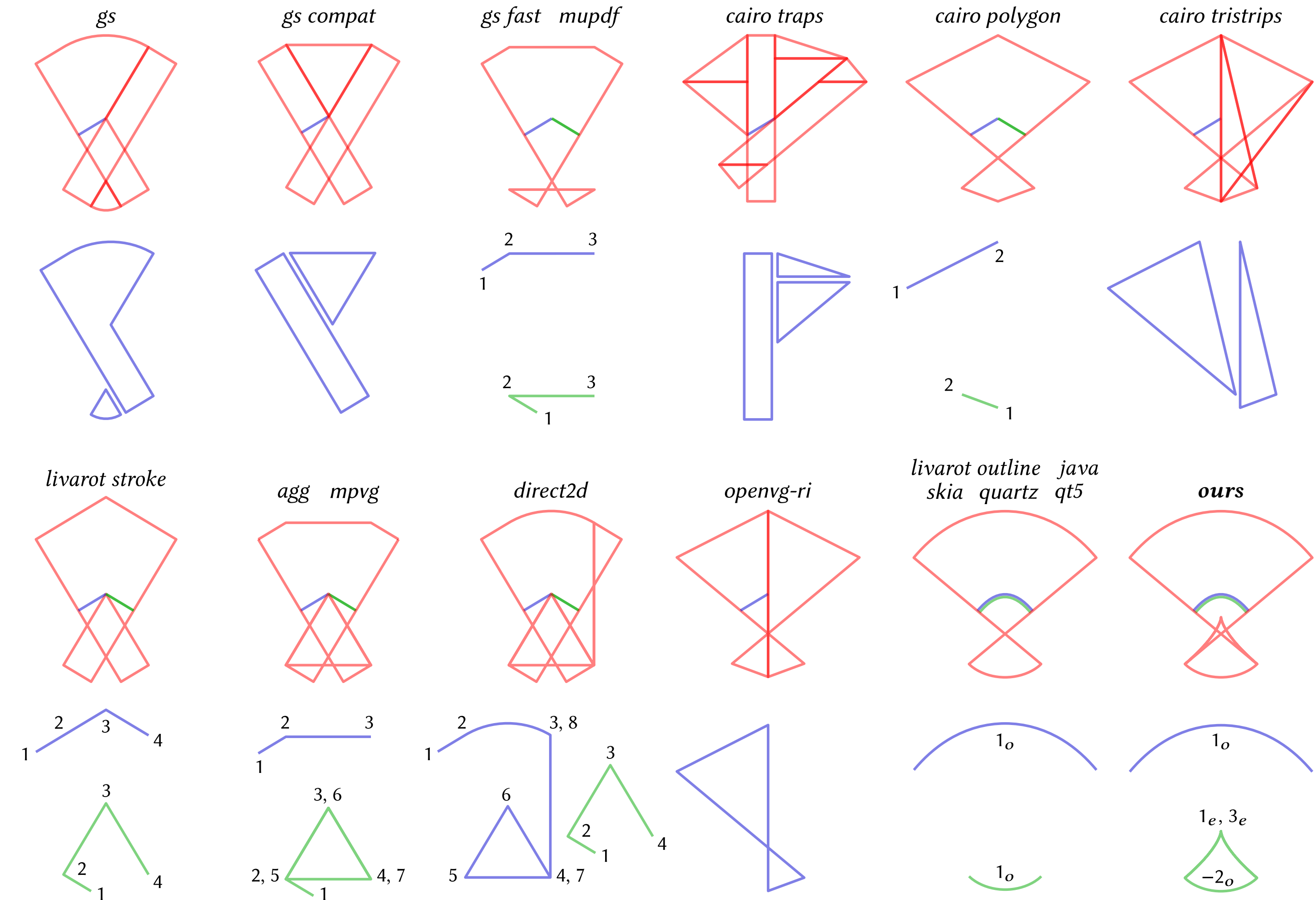

Fig. 13. The wide variety of stroke-to-fill algorithms. Each implementation has two rows. For flat algorithms, the top row shows the result of stroking a parabolic arc flattened into two linear segments. The bottom row shows the output segments generated for each flattened input segment. Local algorithms show the output for a single segment in blue. Global algorithms show the result for the blue segment on the way forward, and for the green segment on the way backward. The two last examples are curve-based, and do not operate on flattened inputs. Existing global algorithms simply traverse the offset on the way forward and backward. Our results show the evolute being traversed twice on the way backward (1_e, 3_e), and the intervening offset reversed (-2_o).

It does not output inner joins, handle crossing radii, or identify and treat intra-segment cusps. Moreover, it connects offsets using the current join type. Inkscape uses additional facilities in the library to post-process these results into a path without self-intersections and where multiple linear segments are approximated by cubic Béziers.

The global strokers offered by Anti-Grain Geometry 2.5, MPVG, and by Direct2D's `ID2D1Geometry::Widen` facility detect and treat the radii crossing at high-curvature regions. For the purpose of figure 13, *agg* and *mpvg* are equivalent. For each input outline, they go back and forth over all approximating linear segments to produce a single output outline. For outer offsets, they behave just like *gs fast* and *mupdf*. However, when generating inner offsets, they detect radii intersections and pivot around the point shared by adjacent segments. This ensures the offset segments are always traversed in a consistent orientation. When filled with the non-zero winding rule, this strategy produces correct results, at least in the limit. The stroker in Direct2D operates in a similar fashion. However, when processing the outer offset on the way forward, it skips to the inner offset to pivot. Then, on the way back, it pivots again at the same vertex when processing the inner offset. Although this seems unnecessary complicated, it is a fairly robust stroker. Its behavior is consistent with it identifying and treating intra-segment cusps and "almost cusps". On the other hand, *agg* and *mpvg* are the only strokers that output inner joins between segments.

The simplest stroker of all is the local algorithm that ships with the OpenVG Reference Implementation. In a single pass, it outputs one quadrilateral per flattened input segment. When the radii cross, these quadrilaterals are not simple polygons. Nevertheless, when streaming into a stencil buffer, it is a conceptually sound method. To turn it into a production stroker, one would have to make the flattening adaptive, add code to identify and treat intra-segment cusps, and output inner joins between segments (and fix few bugs).

When rendering, Adobe Illustrator CS9 seems to employ an excellent flat stroker which we cannot tell is local or global. It treats high-curvature regions properly and is rarely confused by intra-segment cusps, but does not output inner joins between segments.

Table 2. Summary of flat strokers. The first column shows the number of output segments per flattened input segment in the (S)imple case when the radii do not cross; and the (H)ard case when they do. Does the stroker (E)liminate repeated segments? Does it work with crossing-(R)adii? Does it output inner (J)oins? Does it identify intra-segment (C)usps?

Stroker	S/H	E	R	J	C	Stroker	S/H	E	R	J	C
gs	6 / 9	✗	✓	✗	✗	*mupdf*	4 / 4*	✓	✗	✗	✗
gs compat	6 / 6*	✗	✗	✗	✗	*cairo polygon*	2 / 2	✓	✗	✗	✗
cairo traps	10 / 10	✗	✗	✗	✗	*livarot stroke*	6 / 6*	✓	✗	✗	✗
cairo tristrips	6 / 6	✗	✓	✗	✗	*agg*	4 / 8*	✓	✓	✓	✗
openvg-ri	4 / 4	✗†	✓	✗	✗	*mpvg*	4 / 8	✓	✓	✓	✗
gs fast	4 / 4*	✓	✗	✗	✗	*direct2d*	4 / 10	✓	✓	✗	✓

†Needs a stencil. *Depends on join.

A summary of flat stroker characteristics can be found in table 2.

4.2 Curve-based algorithms

All 8 curve-based strokers we analyzed suffer from fundamental problems. None of these strokers output inner joins or identify and treat "almost cusps". They do not even solve the stroke-to-fill problem: they simply connect the offsets of the input segments on the way forward and backward. As we have seen in section 3, this produces incorrect results whenever the radius of curvature of the segment is smaller than half the line width.

Skia's `SkPaint::getFillPath` facility approximates the offsets using quadratic segments only. Java 2D's `BasicStroke` returns a stroked version of a `Shape`. Its implementation in Oracle's JDK 8 uses the Ductus renderer, OpenJDK 8 uses Pisces, and OpenJDK 11 uses Marvin. They use both quadratics and cubics to approximate offsets. All remaining curve-based strokers approximate offsets using cubics. Qt5 offers the facility `QPainterPathStroker::createStroke` and Quartz offers `CGPathCreateCopyByStrokingPath`. Inkscape's "Stroke to Path" uses livarot's `Path::Outline` method when no dashing is needed. Finally, Adobe Illustrator CS9's "Outline Stroke" facility seems to use a different stroker, one that is not nearly as robust as the one used for rendering. It seems as though its results are then subjected to a fill simplification step that eliminates self intersections. The final results are consistent with the stroker being unaware of high-curvature regions and not identifying or treating intra-segment cusps. It does not output inner joins either.

4.3 Distance-based algorithms

Kilgard and Bolz [2012] compute perpendicular distances explicitly to render stroked paths in NVpr. Input paths are first converted to parabolic arcs. The distances to these arcs are found by solving for the roots of cubic polynomials. Then, a conservative hull is obtained for each parabolic arc that needs stroking. These hulls can all be drawn simultaneously into a stencil buffer. If a fragment generated by a hull is too far away from the corresponding stroked segment, it is discarded before modifying the stencil. Joins and caps are similarly drawn into the stencil. In a second phase, the conservative hulls are again drawn into the output image. This time around, fragments selected in the stencil paint the output image and clear their position in the stencil. The quality of results depends on the robustness and accuracy of the approximation of input paths by parabolic arcs, the generation of joins and caps, the root-finding algorithm, and the conservative hull.

Using distances to render strokes is very rare. To the best of our knowledge, NVpr is the only rendering engine with any traction that works in this way. Although it does not solve the stroke-to-fill conversion problem as stated in section 2, it is the most direct application of the definition and therefore is the most straight-forward. This is why we use distances to generate ground-truth renderings.

Ground-truth renderer. To decide whether a given point p should be painted, our ground-truth renderer goes over all segments $\alpha_{i,j}$ and solves for

$$d_{i,j} = \min_{\substack{t \in (0,1) \\ \langle \alpha_{i,j}(t),\ \alpha'_{i,j}(t) \rangle = 0}} \left| p - \alpha_{i,j}(t) \right|. \tag{37}$$

It searches over all roots of $\langle \alpha_{i,j}(t),\ \alpha'_{i,j}(t) \rangle = 0$ for $t \in (0, 1)$. These are polynomials of moderate degree. If $d_{i,j} \le h$ for any segment, it paints the point. (Note that this takes care of intra-segment cusps, since $\alpha'_{i,j}(t) = 0$ at all cusps.) Then, it goes over all joins and caps. If p is inside any of them, it paints the point. This operation is repeated for all pixels in the image, using 16 samples per pixel.

5 RESULTS

Our test harness allows us to stroke paths using any style and any of the strokers we described in the survey. Adobe Illustrator's renderer, NVpr, and the ground-truth renderer produce only raster images. For all other strokers, we obtain the output paths themselves. This allows us not only to render the corresponding filled shapes, but also their outlines. There are too many interesting results to fit within these pages, especially when comparing 22 strokers. We believe the best way to evaluate the robustness of an implementation is to test multiple steps in an animation. Therefore, we generated a variety of animated test cases and produced three animations for each test × implementation combination. These animations show the bare result of stroking a test with a given implementation, the input and output paths, and a comparison against the result produced by the ground-truth renderer. This "animations" dataset offers a mesmerizing glimpse of the workings of each implementation. It is available in supplemental materials.

Even in single-precision, our method produces results virtually indistinguishable from ground truth. We realize that, unlike us, none of the authors of the third-party implementations had the benefit of using our tests to debug their code. However, the errors visible in the animations are not just bugs: they are conceptual problems. Curve-based implementations *must* take evolutes into account. None do. The flat implementations that do consider radii crossings are correct in the limit, but they cannot honor accuracy requirements unless they bound their deviation to the evolute. No implementation does that. Inner joins should be output between segments whenever they are visible. Only two implementations do that, and both are flat (Anti-Grain Geometry's and MPVG's). Finally, all implementations should regularize their inputs. Only two implementations seem to do that, with varying success (Direct2D is flat, and Adobe Illustrator's renderer also likely flat).

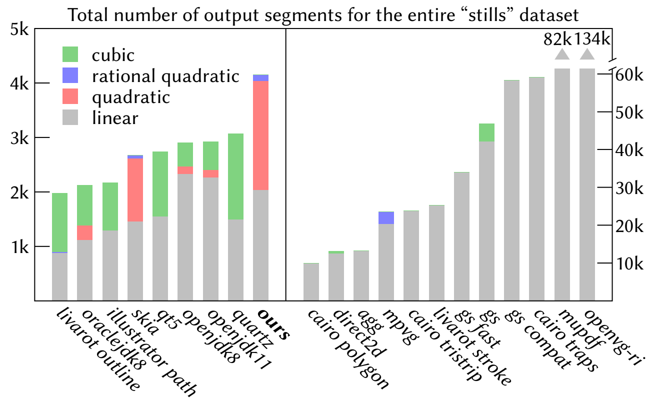

Fig. 14. Our stroker generates more output segments than other global curve-based strokers only where those strokers are broken.

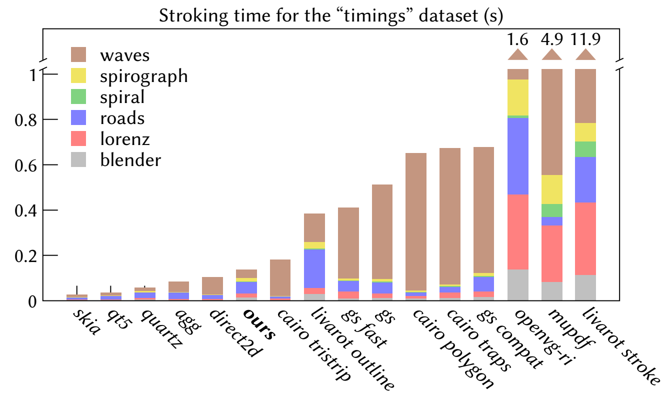

Fig. 15. Our stroker is not particularly fast but also not slow, even with the increased complexity required for its correct behavior.

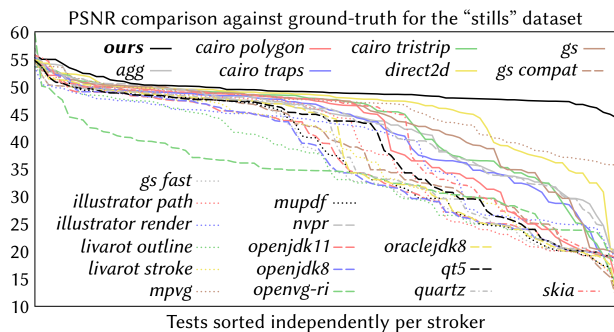

Fig. 16. PSNR values show the robustness of our stroker. Other notable strokers are Direct 2D's and Adobe Illustrator's rendering stroker.

Some quantitative evaluations help position our method among the competing implementations. Since the accuracy targets are not commensurate between different methods, it is difficult to compare the number of segments they output. Nevertheless, global strokers generate fewer segments than local strokers, and curve-based ones fewer than flat ones. Our method generates more segments than other global curve-based strokers only where those other strokers fail to consider the evolutes. Otherwise, we are simply comparing the offsetters. For obvious reasons, difficult cases are overrepresented in our tests. Even so, our stroker generates fewer than 1/3 of the segments output by the most accurate global flat stroker, and 1/5 of the most accurate local flat stroker. Figure 14 shows the total number of segments output for the entire "stills" dataset for each stroker. All results are available as supplemental materials.

We did not find a good way to measure the speed of the strokers in the Java 2D implementations, in Adobe Illustrator, and in NVpr. Figure 15 shows the time the remaining 16 strokers take to process our "timings" dataset (available in supplemental materials). These tests contain between 1 and 3k outlines to be stroked, ranging from 150 to 10k cubic segments of varying curvature. The timings should be taken with a grain of salt, given the small but non-zero overhead involved in translating input and output between our representation and theirs. Our method is ranked 6 among them: not particularly fast or slow. Note that we have not invested much time in optimization.

Figure 16 shows the PSNR comparison against ground-truth for all tests in the "stills" dataset. For each stroker, the lines show PSNR values sorted from best to worst. Although the lines are not comparable for any given x, they are quite informative as a whole. It is clear our stroker behaves better than the alternatives. The PSNR values for each test are available in the supplemental materials.

Figure 17 shows the effect of different components of our algorithm on the final output. Results show the importance of inner joins (section 2), of our regularization stage (section 3.1), of taking the evolute into account when thickening segment pieces (section 3.3), and of simplifying inner-joins (sections 3.2 and 3.3).

6 CONCLUSIONS AND FUTURE WORK

We did not originally set out to work on the stroke-to-fill conversion problem: A stroker was needed for a different project. With such a foundational computer graphics problem, we expected to find a variety of robust open-source implementations, backed by many publications on the topic. Instead, the bibliography search returned nothing specific. None of the dozens of implementations we found seemed to completely solve the problem.

Our curve-based stroker includes several original contributions: the regularization stage, the inner-join simplification between curved segments, and the use of evolutes in high-curvature regions. It is the first stroker that is correct *in principle*. Even before extensive optimization, it is fast enough for most applications. We hope our implementation is adopted and improved upon by the community. There is still challenging work to do on the topic.

Often, stroking part of a segment requires outputting the evolute, even though some other part of the stroked outline (or even of the stroked segment itself) hides its influence. An efficient conservative heuristic that detects such "non-local coverage" would be very useful in avoiding this unnecessary work. The same is true for the inner-join classification procedure of section 3.2, which is also "local".

More work is needed on robust, efficient approximation of curves by cubic or quadratic Béziers, especially when the parameterization speed is at times large and at times close to zero. Again, in this day and age, we expected this to be a "solved problem". This was the most time-consuming and exasperating part of the project.

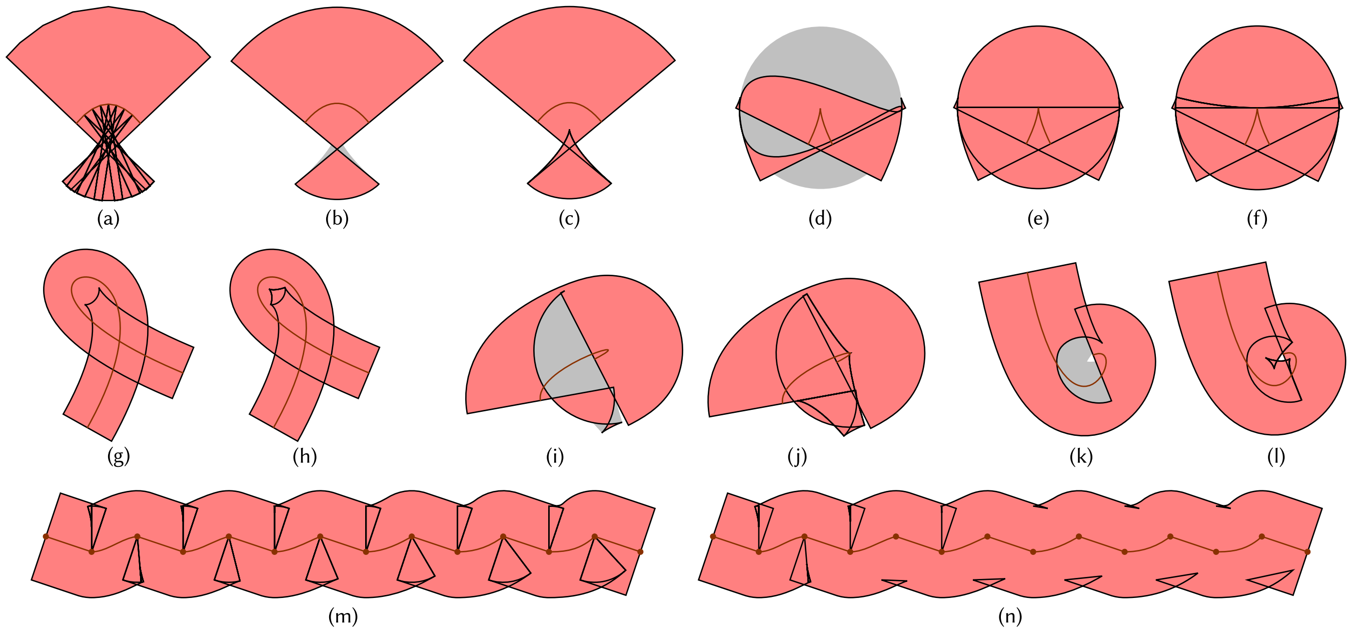

Fig. 17. Highlights of the effect of different components of our algorithm. (a) Using inner joins between segments of the path in figure 4b. (b) The result of blindly connecting offsets, and (c) the correct result that takes evolutes into account. (d) The instability of offset approximations across an intra-segment cusp. (e) How regularization avoids these instabilities. (f) The evolutes are hidden by other parts of the stroked segment, but our algorithm still outputs them. (g) Another example where naive offsetting works because evolutes (h) are hidden non-locally. (i,k) Further examples where offsetting leaves required regions uncovered that are properly filled (j,l) with evolutes. (m) A sequence of curved segments with inner joins like those in figure 10a between them. (n) After optimization, all inner joins were replaced by the simpler ones in figure 10b and c. — input path — output path ■ filled region ■ missing or extraneous region

Someone ought to compile and distribute a benchmark of paths (and corresponding stroke styles) found in real-world illustrations. Another dataset could include synthetic examples covering difficult corner cases. New and existing implementations could then be tested against ground-truth renderings. Most of the tests cases we use were created by Mark Kilgard and bundled with the NVpr demos.

The METAFONT description language [Knuth 1989] allows the stroke width, pen shape, and pen angle to vary along the path. The problem of converting these "generalized" strokes to fills is even more challenging than the problem we solve here.

There is a growing interest in the GPU-accelerated rendering of vector graphics [Dokter et al. 2019; Ganacim et al. 2014; Kilgard and Bolz 2012; Li et al. 2016]. A massively-parallel solution to the stroke-to-fill conversion problem is still missing. Our method could be parallelized using standard techniques (i.e., the count, parallel-scan, process paradigm). The only stage that requires multiple passes is the recursive approximation of offset curves and evolutes. This is a most promising area of future work.

Finally, we would like to investigate the use of interval analysis for subinterval classification.

7 ACKNOWLEDGEMENTS

This work has been funded in part by a research scholarship from CNPq and by an INST grant from FAPERJ. We would like to thank Daniel Yukimura, Juan Carlos Rojas Colunche, and Luiz Henrique de Figueiredo for insightful discussions throughout this project, and the SIGGRAPH reviewers for their comments.

A LISTING FOR DASHING

The dashing procedure is invoked for each segment piece in the stream. It advances the dash pattern, emits the transition parameters, and then, if the piece was at all visible, emits the piece itself.

```
procedure DASH-SEGMENT-PIECE(α, [a, b])
    i ← current dash index
    n ← length needed by current dash
    ℓ ← length of piece [a, b] of α
    v ← odd(i) ∨ n < ℓ              ▹ segment piece is at least partially visible
    while ℓ > 0 do                  ▹ while piece has not been exhausted
        u ← min(ℓ, n)               ▹ most this dash can use from piece
        ℓ ← ℓ − u
        n ← n − u
        if n ≤ 0 then               ▹ dash has been exhausted
            t ← parameter of remaining length ℓ in piece [a, b] of α
            if odd(i) then          ▹ exhausted dash is drawn
                EMIT-TERMINAL-DASH-PARAMETER(t)
            else                    ▹ exhausted dash is discarded
                EMIT-INITIAL-DASH-PARAMETER(t)
            end
            i ← CYCLE-DASH-INDEX(i)
            n ← the length of dash i
        end
    end
    if v then EMIT-SEGMENT-PIECE(α, [a, b]) end   ▹ emit piece if visible
    current dash index ← i
    length needed by current dash ← n
end
```

B LISTING FOR JOIN & CAP

The join and cap procedure adds joins between visible segment pieces, caps loose visible outlines endpoints, and joins the start and end of closed outlines when both endpoints are visible.

```
procedure BEGIN-SEGMENT-PIECE(p, d)
    if odd(current dash index) then
        EMIT-JOIN(last tangent, p, d, 0)
    end
end
procedure CUSP(d0, p, d1, ω)
    last tangent ← d1
    EMIT-CUSP(d0, p, d1, ω)
end
procedure END-SEGMENT-PIECE(p, d)
    last tangent ← d
end
procedure BEGIN-REGULAR-OUTLINE(p, d)
    if phase resets then
        current dash index ← initial dash index
        length needed by current dash ← length needed by initial dash
    end
    begin was visible ← odd(current dash index)
    initial position ← p
    initial tangent ← last tangent ← d
    EMIT-INITIAL-BUTT-CAP(p, d)
end
procedure END-REGULAR-OPEN-OUTLINE(p, d)
    if odd(current dash index) then                    ▷ end visible
        EMIT-TERMINAL-CAP(p, d)
    end
    if begin was visible then
        EMIT-INITIAL-CAP(initial position, initial tangent)
        EMIT-TERMINAL-BUTT-CAP(initial position, initial tangent)
    end
end
procedure END-REGULAR-CLOSED-OUTLINE(p, d)
    if begin was visible ∧ odd(current dash index) then    ▷ both visible
        EMIT-JOIN(d, p, initial tangent)
        EMIT-TERMINAL-CAP(p, initial tangent)
    else if odd(current dash index) then               ▷ only ends visible
        EMIT-TERMINAL-CAP(p, d)
    else if begin was visible then                     ▷ only begins visible
        EMIT-INITIAL-CAP(initial position, initial tangent)
        EMIT-TERMINAL-BUTT-CAP(initial position, initial tangent)
    end
end
```

C PROOF OF THEOREM 3.1

Fix $t_0 \in [a, b]$ and define a reference frame $\big(\alpha'(t_0), \alpha'^{\perp}(t_0)\big)$ with origin at $\alpha(t_0)$ as in figure 18. For the purposes of the proof, assume an arc-length parameterization for α. Let β be the curve α and q be $p = \alpha(a)$ in this new frame:

$$\beta(t) = \big(\alpha(t) - \alpha(t_0)\big) \left[\alpha'(t_0)^T \quad \alpha'^{\perp}(t_0)^T\right], \quad \text{and} \tag{38}$$

$$q = \big(p - \alpha(t_0)\big) \left[\alpha'(t_0)^T \quad \alpha'^{\perp}(t_0)^T\right] = \left[q_x \quad q_y\right]. \tag{39}$$

The perpendicular line segment that sweeps α is

$$\ell(t, s) = \beta(t) + s\beta'^{\perp}(t), \quad \text{for } s \in [-h, h]. \tag{40}$$

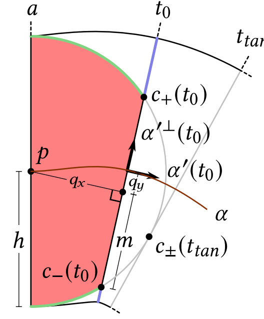

Fig. 18. Inner join coverage proof.

Solving for the s-values of the intersections between $\ell(t, s)$ and the radius-h circle centered at q, we obtain

$$s_\pm(t) = \left\langle q - \beta(t),\ \beta'^{\perp}(t)\right\rangle \pm \sqrt{\left\langle q - \beta(t),\ \beta'^{\perp}(t)\right\rangle^2 + h^2 - \left|q - \beta(t)\right|^2}. \tag{41}$$

In figure 18, the intersection points are

$$c_\pm(t_0) = \ell(t_0, s_\pm(t_0)), \quad \text{so that} \tag{42}$$

$$c_\pm(t_0) = \left[0 \quad q_y \pm m\right], \quad \text{where} \quad m = \sqrt{h^2 - q_x^2}. \tag{43}$$

The region covered by stroking the segment piece $\big(\alpha, [a, t_1]\big)$ is

$$S(a, t_1) = \bigcup_{(t,s) \in [a,t_1] \times [-h,h]} \ell(t, s). \tag{44}$$

The key idea is to show that, if the offsets $\beta_{\pm h}(t_0)$ are outside the disk and $\left|\rho(t_0)\right| > h$, then the following hold:

(1) The intersects $c_\pm(t_0)$ move monotonically towards each other along the circle. I.e., $\mp\left\langle c_\pm(t_0) - q,\ c'_\pm(t_0)\right\rangle > 0$;
(2) The offsets remain outside the disk. In fact, their distances $\left|\beta_{\pm h}(t_0) - c_\pm(t_0)\right|$ to the intersects increase monotonically.

When these conditions hold for all $t_0 \in [a, t_1]$, the perpendicular line segment sweeps the region specified by the theorem.

Substituting,

$$c'_\pm(t_0) = \ell\big(t, s_\pm(t)\big)'\Big|_{t=t_0}, \tag{45}$$

$$= \left[1 - \kappa(t_0)(q_y \pm m) \quad q_x\left(-\kappa(t_0) \pm \frac{1-\kappa(t_0)\,q_y}{m}\right)\right]. \tag{46}$$

When the curvature $\kappa(t_0) = 0$,

$$\mp\left\langle \big(c_\pm(t_0) - q\big)^{\perp},\ c'_\pm(t_0)\right\rangle = \frac{h^2}{m} > 0, \quad \text{and} \tag{47}$$

$$\left(\left|\beta_{\pm h}(t) - c_\pm(t)\right|^2\right)'\Big|_{t=t_0} = \frac{2q_x}{m}\,\mu_\pm > 0, \quad \text{where} \tag{48}$$

$$\mu_\pm = m \pm q_y - h. \tag{49}$$

Inequality (47) holds trivially. To see that inequality (48) also holds, first note that $\mu_\pm < 0$, because the offsets are outside the circle, and therefore $|q_y \pm m| < h$. On the other hand, $q_x < 0$ because

$$q_x|_{t_0=a} = 0, \quad \text{and} \tag{50}$$

$$q'_x = -1 + \kappa(t_0)\,q_y < 0, \quad \text{for all } t_0 \in [a, t_1], \quad \text{since} \tag{51}$$

$$|q_y| < \max|q_y \pm m| < h \quad \text{and} \quad \left|\kappa(t_0)\right| = \frac{1}{|\rho(t_0)|} < \frac{1}{h}. \tag{52}$$

When $\kappa(t_0) \neq 0$,

$$\mp\left\langle \left(c_\pm(t_0) - q\right)^\perp,\ c'_\pm(t_0)\right\rangle = \frac{h^2}{m}\,\eta_\pm > 0, \quad \text{where} \tag{53}$$

$$\left(\left|\beta_{\pm h}(t) - c_\pm(t)\right|^2\right)'\Big|_{t=t_0} = \frac{2q_x}{m}\,(\mu_\pm)\,(\eta_\pm) > 0, \quad \text{and} \tag{54}$$

$$\eta_\pm = 1 - \frac{q_y \pm m}{\rho(t_0)}. \tag{55}$$

Since $|q_y \pm m| < h$ and $\left|\rho(t_0)\right| > h$, it follows that $\eta_\pm > 0$. Therefore, both inequalities (53) and (54) hold.

To complete the proof, note that the offsets start on top on the circle, but must immediately move outside because

$$\left|\beta_\pm(a) - q\right| = h, \quad \left(\left|\beta_\pm(t) - q\right|^2\right)'\Big|_{t=a} = 0, \quad \text{and} \tag{56}$$

$$\left(\left|\beta_\pm(t) - q\right|^2\right)''\Big|_{t=a} = 2\left(1 \mp k(a)\,h\right) > 0. \tag{57}$$

D LISTING FOR CLASSIFY JOINS & CUSPS

To compute the join/cusp ω parameter using corollary 3.2, the two adjacent segment pieces are considered independently. The first segment is reversed so the same procedure can be used for both.

```
procedure GET-JOIN-ω(h, α₀, [a, b], d₀, p, d₁, α₁, [u_i, u_f])
    d₀ ← d₀/|d₀|
    d₁ ← d₁/|d₁|
    ω₁ ← GET-SEGMENT-ω(h, d₀, p, d₁, α₁, [u_i, u_f])
    ω₀ ← GET-SEGMENT-ω(h, −d₁, p, −d₀, REVERSE(α₀), [1 − b, 1 − a])
    return ω₀ + ω₁
end
procedure GET-SEGMENT-ω(h, d₀, p, d₁, α, [a, b])
    t_{|ρ|=h} ← b
    t_tan ← b + 1
    t_vtx ← b + 1                                          ▷ initial sentinels
    t_{|ρ|=h} ←? first root of ρ(t) = ±h in [a, b]        ▷ change only if found
    if |ρ(½(a + t_{|ρ|=h}))| < h then     ▷ curvature bound already violated
        return 0
    end
    t_tan ←? first root of ⟨α(t) − p, α′(t)/|α′(t)|⟩ = ±h in [a, b]
    if t_tan ≤ t_{|ρ|=h} then
        return 1
    end
    σ ← sign ⟨d₀^⊥, d₁⟩
    v ← σ h d₀^⊥
    t_vtx ←? first root of ⟨α(t) − v, α′(t)⟩ = 0 in [a, b]
    if t_vtx ≤ t_{|ρ|=h} then
        n ← α′^⊥(t_vtx)/|α′(t_vtx)|
        v′ ← v − 2⟨n, v⟩n
        if σ ⟨v′, d₁⟩ > 0 ∨ σ ⟨v′, d₀⟩ < 0 then ▷ v′ is outside the join
            return 1
        end
    end
    return 0
end
```

REFERENCES

E. Catmull. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* Ph.D. Dissertation. Dept. Computer Science.

CGM. 1999. *Computer Grahpics Metafile* (2nd ed.). ISO/IEC 8632.

M. Dokter, J. Hladky, M. Parger, D. Schmalstieg, H.-P. Seidel, and M. Steinberger. 2019. Hierarchical Rasterization of Curved Primitives for Vector Graphics Rendering on the GPU. *Computer Graphics Forum* 38, 2 (2019), 93–103. https://doi.org/10.1111/cgf.13622

G. Elber, In-Kwon Lee, and Myung-Soo Kim. 1997. Comparing offset curve approximation methods. *IEEE Computer Graphics and Applications* 17, 3 (1997), 62–71. https://doi.org/10.1109/38.586019

R. T. Farouki. 1997. Optimal parameterizations. *Computer Aided Geometric Design* 14, 2 (1997), 153–168. https://doi.org/10.1016/s0167-8396(96)00026-x

F. Ganacim, R. S. Lima, L. H. de Figueiredo, and D. Nehab. 2014. Massively-Parallel Vector Graphics. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2014)* 36, 6 (2014), 229. https://doi.org/10.1145/2661229.2661274

T. F. Hain, A. L. Ahmad, S. V. R. Racherla, and D. D. Langan. 2005. Fast, precise flattening of cubic Bézier path and offset curves. *Computers & Graphics* 29, 5 (2005), 656–666. https://doi.org/10.1016/j.cag.2005.08.002

N. J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (2nd ed.). SIAM.

J. Hoschek. 1988. Spline approximation of offset curves. *Computer Aided Geometric Design* 5, 1 (1988), 33–40. https://doi.org/10.1016/0167-8396(88)90018-0

H. Jiang, S. Li, L. Cheng, and F. Su. 2010. Accurate evaluation of a polynomial and its derivative in Bernstein form. *Computers & Mathematics with Applications* 60, 3 (2010), 744–755. https://doi.org/10.1016/j.camwa.2010.05.021

B. Jüttler. 1997. A vegetarian approach to optimal parameterizations. *Computer Aided Geometric Design* 14, 9 (1997), 887–890. https://doi.org/10.1016/s0167-8396(97)00044-7

M. J. Kilgard and J. Bolz. 2012. GPU-accelerated Path Rendering. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2012)* 31, 6 (2012), 172.

D. E. Knuth. 1989. *The METAFONTbook.* Addison-Wesley.

R. Li, Q. Hou, and K. Zhou. 2016. Efficient GPU Path Rendering Using Scanline Rasterization. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2016)* 35, 6 (2016). https://doi.org/10.1145/2980179.2982434

W. M. Newman and R. F. Sproull. 1979. *Principles of Interactive Computer Graphics* (2nd ed.). McGraw-Hill, Chapter 16.

OpenXPS. 2009. *Open XML Paper Specification* (first ed.). ECMA-388.

PDF. 2006. *Adobe Portable Document Format, v. 1.7* (sixth ed.). Adobe Systems Incorporated.

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. 2007. *Numerical Recipes: The Art of Scientific Computing* (3rd ed.). Cambridge University Press.

PS. 1999. *PostScript Language Reference* (3rd ed.). Adobe Systems Incorporated.

J. Snyder. 1992. Interval analysis for computer graphics. *Computer Graphics (Proceedings of ACM SIGGRAPH 92)* 26, 2 (1992), 121–130. https://doi.org/10.1145/133994.134024

SVG. 2011. *Scalable Vector Graphics, v. 1.1* (second ed.). W3C.

W. Tiller and E. Hanson. 1984. Offsets of Two-Dimensional Profiles. *IEEE Computer Graphics and Applications* 4, 9 (1984), 36–46. https://doi.org/10.1109/mcg.1984.275995

J. Warnock and D. K. Wyatt. 1982. A Device Independent Graphics Imaging Model for Use with Raster Devices. *Computer Graphics (Proceedings of ACM SIGGRAPH 1982)* 16, 3 (1982), 313–319. https://doi.org/10.1145/800064.801297