Graphical Models 141 (2025) 101295

Contents lists available at ScienceDirect

Graphical Models

journal homepage: www.elsevier.com/locate/gmod

Review Article

GPU-accelerated rendering of vector strokes with piecewise quadratic approximation

Xuhai Chen [a], Guangze Zhang [b], Wanyi Wang [a], Juan Cao [a], Zhonggui Chen [b,*]

[a] School of Mathematical Sciences, Xiamen University, Fujian, 361005, China
[b] School of Informatics, Xiamen University, Fujian, 361005, China

ARTICLE INFO

Keywords:
Curvature
Rendering
Vector graphics
Parallel computing
Adaptive subdivision
Arc-length parameterization

ABSTRACT

Vector graphics are widely used in areas such as logo design and digital painting, including both stroked and filled paths as primitives. GPU-based rendering for filled paths already has well-established solutions. Due to the complexity of stroked paths, existing methods often render them by approximating strokes with filled shapes. However, the performance of existing methods still leaves room for improvement. This paper designs a GPU-accelerated rendering algorithm along with a curvature-guided parallel adaptive subdivision method to accurately and efficiently render stroke areas. Additionally, we propose an efficient Newton iteration-based method for arc-length parameterization of quadratic curves, along with an error estimation technique. This enables a parallel rendering approach for dashed stroke styles and arc-length guided texture filling. Experimental results show that our method achieves average speedups of 3.4× for rendering quadratic stroked paths and 2.5× for rendering quadratic dashed strokes, compared to the best existing approaches.

Contents

* Corresponding author.
E-mail addresses: 19020230157197@stu.xmu.edu.cn (X. Chen), 23020231154255@stu.xmu.edu.cn (G. Zhang), 13758662305@163.com (W. Wang), juancao@xmu.edu.cn (J. Cao), chenzhonggui@xmu.edu.cn (Z. Chen).

https://doi.org/10.1016/j.gmod.2025.101295
Received 26 June 2025; Received in revised form 27 July 2025; Accepted 29 July 2025
Available online 13 August 2025

1. Introduction

Raster and vector graphics are two common forms of digital image representation, each with distinct characteristics and suited for different application scenarios. Raster images consist of discrete pixels, each representing a specific color value. This structure enables rich and detailed color variation, making raster graphics widely used in digital photography, image editing, and printing. However, because image data is stored in a fixed pixel grid, raster graphics often suffer from distortion and jagged edges when scaled up, which degrades visual quality. Additionally, storing color information for every pixel typically requires a large amount of memory. In contrast, vector graphics represent images using geometric primitives such as points and Bézier curves. This resolution-independent representation allows vector graphics to retain sharpness under arbitrary scaling when rendered appropriately. They also require less storage space and offer greater flexibility for editing. These advantages make vector graphics valuable in logo design, icon production, digital painting, and animation.

Fill and stroke are the two most fundamental rendering styles of paths (the basic primitives of vector graphics) [1]. Filling refers to coloring the interior of a region enclosed by a path, using solid colors, gradients, or textures. Stroking involves drawing along the boundary of a path, with the appearance of the contour controlled by parameters such as line width and color. Intuitively, a stroked path represents the area swept out by a brush moving along a curve, with width and color corresponding to the brush size and ink color, respectively. Efficiently rendering resolution-independent fills and strokes is a key challenge in vector graphics rendering.

The problem of GPU-accelerated filled path rendering has been largely solved [2,3], with existing solutions based on implicit equations [4,5] and scanline rasterization [6]. In contrast, efficiently rendering stroked paths is more challenging [2,3]. This is primarily because stroke definition involves computing offset curves of the path boundaries, and the complexity of offset curves increases significantly [7]. As a result, using implicit equations of the original boundary curves to determine pixel inclusion or the scanline based methods cannot be directly applied to stroke regions. Additionally, stroke rendering supports various styles, such as dashed lines and texture fills [8]. These require arc-length parameterization of the original curves. For dashed strokes, each solid segment is specified by its arc length and spacing; rendering requires computing the corresponding start and end parameters on the Bézier curve's domain (i.e., the interval $[0, 1]$). To minimize texture distortion, the curve's aspect ratio should approximate that of the texture, which also requires arc-length-based segmentation — again relying on arc-length parameterization.

Existing stroke rendering methods either approximate strokes as filled regions [2,3,9] or compute perpendicular distances explicitly [10], but none of these approaches achieve optimal performance. In Polar Stroking [9], trapezoids are used to approximate stroked regions; however, a large number is required to accurately represent curved shapes. The method in [2] relies on sequential generation of quadratic outlines, making it unsuitable for GPU implementation [3]. Vello [3] introduces an elegant parallel curve approximation method based on Euler spirals, but requires substantial numerical integration due to the complexity of Euler spirals and handles dashing styles on the CPU. NVpr [10], on the other hand, computes costly perpendicular distances for each pixel within a CPU-generated bounding hull of the stroked quadratic segment.

This paper presents a high-performance, GPU-accelerated method for vector stroke rendering that incorporates efficient arc-length parameterization, addressing the limitations of existing approaches in terms of accuracy, efficiency, and parallelism. In particular, we introduce a GPU-accelerated vector stroke rendering method that uses approximated stroked quadratic segments as the fundamental primitives. Our method combines the efficiency of implicit equation-based rendering [4] with the accuracy of offset curve fitting through curvature-guided adaptive subdivision [11]. In addition, we propose an efficient method for computing arc-length parameterization, enabling highly efficient parallel dashed stroke rendering and texture filling. Our contributions are summarized as follows:

1. We propose a GPU-accelerated rendering algorithm for approximated stroked quadratic regions, based on implicit equations and curve offset approximation. A curvature-guided GPU-accelerated adaptive subdivision method is designed, which reduces the number of segments compared to trapezoids while maintaining accuracy.
2. We introduce an efficient Newton-iteration-based method for computing arc-length parameterization of quadratic Bézier curves, with a fast-converging initial value selection strategy and error estimation. We apply this to dashed stroke rendering by proposing a parallel algorithm for computing the start and end parameters of each solid segment. Furthermore, we construct texture mappings for the approximated stroked quadratic regions and propose an arc-length-guided texture filling method.

2. Related work

2.1. Rendering filled paths

Rendering of filled paths has already been addressed with relatively mature solutions. Loop et al. [4] triangulate the fill path, where the triangulation consists of control triangles that enclose the boundary curves and a Delaunay triangulation [12] of the polygonal interior (excluding the control triangles). Their insight is using the implicit equations of Bézier curves within the control triangles to determine whether a given pixel lies inside the curve. This enables efficient parallel rendering of the curved regions of the fill path. However, the Delaunay triangulation of the polygon is time-consuming. To address this, Kokojima et al. [5] simplified the triangulation step by decomposing the polygon into a series of potentially overlapping triangles and using GPU stencil testing to achieve efficient polygon rendering. This method requires the boundary Bézier curves to be convex, and thus only supports rendering of filled paths composed entirely of quadratic Bézier curves. Kilgard et al. [10] extended this method by using Bézier curve subdivision to break cubic curves into multiple convex subcurves, enabling efficient rendering of filled paths with cubic curve boundaries.

2.2. Rendering stroked paths

In contrast to filled path rendering, efficiently rendering stroked paths presents greater challenges. This is because defining stroked paths involves offsetting the boundary curves, and the complexity of the

offset curves increases significantly. Farouki et al. [7] noted that the degree of an offset of a quadratic curve is six, and for a cubic curve, it becomes ten. This prevents Loop et al.'s method [4], which relies on implicit equations of the original curves, from being directly applied to stroke regions.

Kilgard et al. [10] uses the exact stroked quadratic segments as the rendering primitive for strokes in NVpr. For the quadratic stroke regions, they determine pixel inclusion by computing the distance from each pixel to the quadratic curve. This involves calculating the foot of the perpendicular from the pixel to the curve, which requires solving a cubic equation—resulting in high per-pixel computational cost. Moreover, the method requires precomputing a bounding polygon and its triangulation for each stroke region on CPU, which also entails significant preprocessing overhead. Regarding arc-length parameterization which is required in dashed stroke rendering, Kilgard et al. [10] take advantage of the fact that the arc length of a quadratic Bézier curve can be computed exactly and explicitly. They approximate curves using recursive quadratic Bézier segments and use the cumulative lengths of these segments as an arc-length function. However, efficiently solving the arc-length parameterization for each segment is non-trivial, and their method does not specify how this is achieved. As for texture filling, rendering textures along stroked paths requires constructing a texture mapping over the stroke region. Due to the complexity of offset curves, constructing such a mapping for quadratic stroke regions is also non-trivial, and their method does not offer a solution.

In a subsequent work, Kilgard [9] abandoned the use of quadratic stroke regions as the minimal rendering unit in Polar Stroking. Instead, the work used simpler linear stroke regions (trapezoids). Unlike traditional methods that recursively subdivide curves into straight segments until the approximation error is below a threshold, this method controls the approximation accuracy by limiting the angle between adjacent trapezoidal segments. The subdivision parameters are determined by uniformly sampling in angular space and solving the inverse mapping, thereby avoiding recursive subdivision. Since curves are approximated by piecewise linear segments, the arc-length function is piecewise linear, making arc-length parameterization easy. This simplifies the implementation of dashed strokes and texture-filled strokes. While linear primitives are simple, they cannot accurately capture the curvature of the original curves, resulting in a high number of required segments.

Around the same time as Kilgard's work [9], Nehab [2] contributed valuable insights into the correctness of stroke rendering by addressing critical issues such as evolutes and inner joins through a comprehensive survey of prior methods. The approach approximates input paths using quadratic curves and computes offset curves via the widely adopted method of Tiller et al. [13]. However, due to its reliance on sequential processing to generate watertight outlines and dashed stroke segments, the method is not suitable for GPU implementation [3]. Furthermore, it does not address texture filling within stroke regions.

The current state-of-the-art in stroke rendering, Vello, was introduced by Levien et al. [3]. This method pioneers GPU-based stroke expansion — converting stroked paths into filled regions — and introduces an elegant parallel curve approximation technique based on Euler spirals. A key innovation is the use of invertible error metrics, which reduce excessive subdivision and increase the exploitable parallelism. Euler spirals, along with their offset and evolute curves, offer a well-established relationship between curvature and arc length, making them ideal for error estimation and inversion. However, despite these advantages, a major drawback is the need for numerical integration during the flattening process — i.e., converting offset curves into line or arc segments suitable for scanline-based rendering — which is computationally more expensive than sampling Bézier curves, which only require linear interpolation. Additionally, the dashing style in Vello is still processed on the CPU, which limits performance in certain scenarios.

3. Preliminaries

3.1. Bézier curves

A Bézier curve [14] of degree n is defined by $n+1$ control points $\mathbf{P}_0, \dots, \mathbf{P}_n$ as follows:

$$\mathbf{B}(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i \mathbf{P}_i, t \in [0,1]. \tag{1}$$

These control points determine the curve's shape and direction. Bézier curves are advantageous due to their simplicity in computation, ease of implementation, and good visual intuitiveness. In particular, a quadratic Bézier curve has the expression:

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2t(1-t)\mathbf{P}_1 + t^2\mathbf{P}_2, t \in [0,1]. \tag{2}$$

Its tangent vector $\mathbf{B}'(t)$ forms a linear (degree-1 Bézier) curve using control points $2\mathbf{P}_0\mathbf{P}_1$ and $2\mathbf{P}_1\mathbf{P}_2$:

$$\mathbf{B}'(t) = 2(1-t)\mathbf{P}_0\mathbf{P}_1 + 2t\mathbf{P}_1\mathbf{P}_2, t \in [0,1]. \tag{3}$$

3.2. Rendering based on implicit equations

Define the standard planar quadratic curve by using the control points $(0,0)$, $(0.5,0)$, and $(1,1)$ for a quadratic Bézier curve. Points on this curve can be expressed as

$$\mathbf{B}(t) = (t, t^2)^T, t \in [0,1], \tag{4}$$

which satisfy the implicit equation in uv space:

$$u^2 - v = 0. \tag{5}$$

By evaluating the sign of $u^2 - v$, one can determine whether a point (u, v) lies outside or inside the standard curve. The rendering method based on implicit equations [4] applies an affine transformation to map arbitrary planar quadratic Bézier control triangles onto the standard control triangle. For a pixel $\mathbf{P}$, its transformed coordinate (u, v) in the standard curve space is used to compute $u^2 - v$, and the sign of the result is used to determine whether the pixel lies inside or outside the general quadratic curve. Based on a chosen rule (e.g., only rendering the interior of the quadratic curve), this enables rendering of curved-edge triangles.

3.3. Offset curves and stroke regions

Given a planar curve $\mathbf{g}(t)$, where $t \in [0,1]$, its offset curve at distance d is defined as:

$$\mathbf{O}(t) = \mathbf{g}(t) + d\,\mathbf{n}(t), t \in [0,1], \tag{6}$$

where $\mathbf{n}(t)$ is the unit normal vector to $\mathbf{g}(t)$. The stroke region of width w generated from $\mathbf{g}(t)$ is given by:

$$\boldsymbol{\Omega}(t,u) = \mathbf{g}(t) \pm \frac{w}{2} u\mathbf{n}(t), t, u \in [0,1]. \tag{7}$$

The offset curves are much more complex than the original curves. Existing work often approximate the offset curves to reduce complexity, we refer to [15] for a detailed survey.

3.4. Curvature

For a planar curve $\mathbf{g}(t)$, the curvature at a point $\mathbf{P}$ of $\mathbf{g}(t)$ is defined as the rate of change of the tangent angle at that point. Take a small arc segment Δs along the curve from $\mathbf{P}$ to another point $\mathbf{P}'$ located at a distance Δs. Let the tangent at $\mathbf{P}$ form an angle α with the x-axis, and the tangent at $\mathbf{P}'$ form an angle α'. The change in angle is $\Delta\alpha = \alpha' - \alpha$, and the average rate of change is $\frac{\Delta\alpha}{\Delta s}$. Taking the limit as $\Delta s \to 0$ gives the curvature. Fig. 1 gives a visualization for the quantities involved in the definition of curvature.

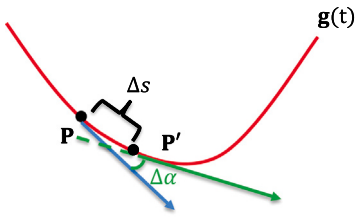


Fig. 1. Visualization of the quantities involved in the definition of curvature.

3.5. Arc-length parameterization

For a planar (continuously differentiable) curve $\mathbf{g}(t)$, the arc-length function is defined as:

$$s(t) = \int_0^t |\mathbf{g}'(\tau)| d\tau, \tag{8}$$

which gives the accumulated arc length along the curve from $t = 0$ to the current t. The inverse function $t = t(s)$ finds the parameter t that results in a cumulative arc length of s. The arc-length parameterization of $\mathbf{g}(t)$ is then the curve $\mathbf{g}(t(s))$, where s is the parameter. In this parameterization, the tangent vector has unit magnitude at all points. This property makes arc-length parameterization widely useful in curve analysis and geometric processing.

4. Accelerated rendering and stylization of stroke paths

In this section, we address the problem of rendering quadratic strokes. Our method also supports general parametric curves through piecewise quadratic approximation. This includes cubic Bézier curves and conic curves, for which established piecewise quadratic approximation techniques can be found in [16,17].

4.1. Approximation and fast rendering of stroke regions for quadratic curves

Among various approaches for approximating offset curves of quadratic curves, the method proposed by Tiller et al. [13] is considered the most accurate [15]. Therefore, this paper adopts their technique to approximate the stroke region of a quadratic Bézier curve.

Specifically, let the control points of a quadratic Bézier curve $\mathbf{B}(t)$ be $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2$, and let $\mathbf{n}_0$ and $\mathbf{n}_2$ denote the unit normal vectors at the endpoints $\mathbf{P}_0$ and $\mathbf{P}_2$, respectively. Given a stroke width of d, the offset curve in the outward direction, $\mathbf{B}(t) + d \cdot \mathbf{n}(t),\ t \in [0,1]$, can be approximated by a quadratic curve $\mathbf{O}_1(t)$ defined by the control points $\mathbf{P}'_0, \mathbf{P}'_1, \mathbf{P}'_2$, where

$$\mathbf{P}'_0 = \mathbf{P}_0 + d \cdot \mathbf{n}_0, \tag{9}$$

$$\mathbf{P}'_2 = \mathbf{P}_2 + d \cdot \mathbf{n}_2, \tag{10}$$

$\mathbf{P}'_1$ is the intersection point of the lines $\mathbf{P}'_0 + \alpha(\mathbf{P}_1 - \mathbf{P}_0), \alpha \in \mathbb{R}$ and $\mathbf{P}'_2 + \alpha(\mathbf{P}_2 - \mathbf{P}_1), \alpha \in \mathbb{R}$. Similarly, the inward offset curve $\mathbf{B}(t) - d \cdot \mathbf{n}(t),\ t \in [0,1]$ is approximated by a quadratic curve $\mathbf{O}_2(t)$ with control points $\mathbf{P}''_0, \mathbf{P}''_1, \mathbf{P}''_2$, where

$$\mathbf{P}''_0 = \mathbf{P}_0 - d \cdot \mathbf{n}_0, \tag{11}$$

$$\mathbf{P}''_2 = \mathbf{P}_2 - d \cdot \mathbf{n}_2, \tag{12}$$

$\mathbf{P}''_1$ is the intersection point of the lines $\mathbf{P}''_0 + \alpha(\mathbf{P}_1 - \mathbf{P}_0)$ and $\mathbf{P}''_2 + \alpha(\mathbf{P}_2 - \mathbf{P}_1)$. Fig. 2 illustrates the geometry of this approximation. The thick green curves above and below represent $\mathbf{O}_1(t)$ and $\mathbf{O}_2(t)$, respectively, while the thick red curve denotes the original curve $\mathbf{B}(t)$. The stroke region is thus approximated by the region enclosed by the two quadratic curves $\mathbf{O}_1(t)$ and $\mathbf{O}_2(t)$, and the two line segments $\mathbf{P}''_0\mathbf{P}'_0$ and $\mathbf{P}''_2\mathbf{P}'_2$.

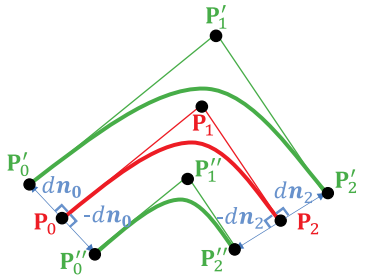


Fig. 2. Visualization of the offset curves approximation. The accurate offset curves of $\mathbf{B}(t)$ are approximated by quadratic curves $\mathbf{O}_1(t)$ and $\mathbf{O}_2(t)$, whose control triangles are $\mathbf{P}'_0\mathbf{P}'_1\mathbf{P}'_2$ and $\mathbf{P}''_0\mathbf{P}''_1\mathbf{P}''_2$, respectively.

Furthermore, we demonstrate that this approximation is highly suitable for parallel GPU rendering. Since the boundaries of the approximated stroke region are also quadratic curves, we render them directly using an implicit equation-based method [4], leveraging the GPU's fragment shader to handle curved triangles.

We propose the following algorithm for rendering the approximated stroke region:

Algorithm 1 Rendering of the Approximate Stroke Region

Input: Control points $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2$ of the quadratic curve; stroke width $w = 2d$

Output: Rendered stroke region

1: Compute the control points of the offset curves: $\mathbf{P}'_0, \mathbf{P}'_1, \mathbf{P}'_2$ and $\mathbf{P}''_0, \mathbf{P}''_1, \mathbf{P}''_2$

2: Use the implicit equations of the offset curves $\mathbf{O}_1(t)$ and $\mathbf{O}_2(t)$ to render the three curved triangles: $\mathbf{P}'_0\mathbf{P}'_1\mathbf{P}'_2$, $\mathbf{P}'_0\mathbf{P}''_0\mathbf{P}'_2$, and $\mathbf{P}''_0\mathbf{P}'_2\mathbf{P}''_2$

Fig. 3 provides an intuitive illustration of this algorithm. Based on Bézier curve geometry, the three triangles, i.e., $\mathbf{P}'_0\mathbf{P}'_1\mathbf{P}'_2$, $\mathbf{P}'_0\mathbf{P}''_0\mathbf{P}'_2$, and $\mathbf{P}''_0\mathbf{P}'_2\mathbf{P}''_2$, fully cover the approximated stroke region. For each pixel covered by $\mathbf{P}'_0\mathbf{P}'_1\mathbf{P}'_2$, we determine whether it lies inside the curve $\mathbf{O}_1(t)$ using $\mathbf{O}_1(t)$'s implicit equation evaluation result and discard the fragments outside $\mathbf{O}_1(t)$ in the fragment shader which is invoked by rasterizing the circumscribed triangle $\mathbf{P}'_0\mathbf{P}'_1\mathbf{P}'_2$. Additionally, when curve $\mathbf{O}_2(t)$ intersects this triangle, we also discard fragments if the evaluation of implicit equation indicates they fall outside $\mathbf{O}_2(t)$. To optimize performance, we perform intersection test between the control triangles of $\mathbf{O}_1(t)$ and that of $\mathbf{O}_2(t)$—i.e., between triangle $\mathbf{P}'_0\mathbf{P}'_1\mathbf{P}'_2$ and $\mathbf{P}''_0\mathbf{P}''_1\mathbf{P}''_2$. If they intersect, the $\mathbf{O}_2(t)$'s implicit equation evaluation is necessary; otherwise, it can be skipped. For the other two triangles, i.e., triangle $\mathbf{P}'_0\mathbf{P}''_0\mathbf{P}'_2$ and $\mathbf{P}''_0\mathbf{P}'_2\mathbf{P}''_2$, since $\mathbf{O}_1(t)$ does not intersect them, only the implicit equation evaluation which indicates if the fragments fall outside $\mathbf{O}_2(t)$ in the fragment shader is required per pixel.

From an implementation standpoint, Step 1 can be executed using the GPU's geometry shader, while Step 2 is handled by the fragment shader [18]. Therefore, the algorithm is highly parallelizable and efficiently implementable on modern GPUs. Although subtractive rasterization typically relies on stencil tests or additional framebuffers, our method avoids these extra steps. Instead, we evaluate implicit equations in the fragment shader to determine whether a fragment lies outside the curve boundary; such fragments are then discarded, effectively achieving the subtractive rasterization effect—without requiring any additional framebuffer.

Regarding approximation accuracy, we take a two-stage approach. First, assuming that the input path is already composed of quadratic curves, we analyze the accuracy of the stroke region approximation in terms of the "turning angle" (to be defined in the following section). Second, a preprocessing step is required to convert general path segments into quadratic curves, ideally in a way that also considers the turning angle (which has not been done to our knowledge). Although

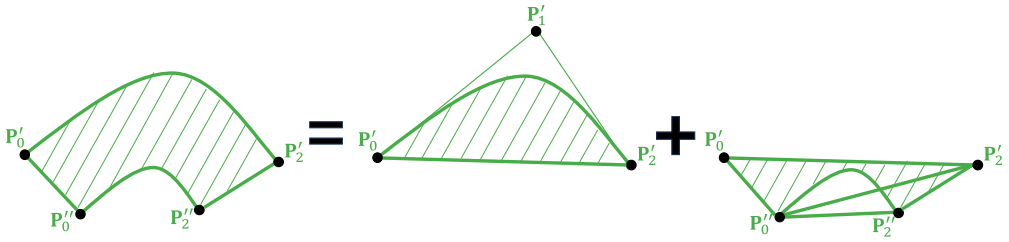


Fig. 3. Representation of the approximated stroke region. The approximated region is decomposed into three triangles $\mathbf{P}'_0\mathbf{P}'_1\mathbf{P}'_2$, $\mathbf{P}'_0\mathbf{P}''_0\mathbf{P}'_2$ and $\mathbf{P}''_0\mathbf{P}'_2\mathbf{P}''_2$ that contain curved features. In all triangles, pixels between $\mathbf{O}_1(t)$ and $\mathbf{O}_2(t)$ should be rendered, which can be achieved by Loop et al. [4]'s implicit equation-based rendering method.

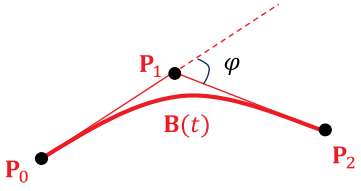


Fig. 4. Turning angle of a quadratic curve $\mathbf{B}(t)$.

a rigorous analysis of this approximation step is beyond the scope of this paper, we plan to extend our method in the future with a formal investigation into how well quadratic curves approximate general path segments with bounded turning angles, which would further enhance the global accuracy and robustness of the method.

4.2. Curvature-guided adaptive subdivision of quadratic curves

This paper uses an approximation method to render the stroked region of quadratic curves, and to ensure sufficient approximation accuracy, we propose a curvature-guided adaptive subdivision algorithm for quadratic curves, as described in Algorithm 2. This method can be efficiently implemented on the GPU in parallel. Since the subdivision is guided by curvature — i.e., more subdivisions are applied to regions with higher curvature — it also aligns with human intuition.

Algorithm 2 Curvature-Guided Adaptive Subdivision for Quadratic Curves

Input: Control points $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2$ of a quadratic Bézier curve; the maximum turning angle φ_0 allowed for each subcurve

Output: Number of subdivisions K; parameter values t_i for subdivision points

1: Compute the turning angle φ of the original curve
2: Determine the number of segments $K = \lceil \varphi/\varphi_0 \rceil$
3: Solve a system of equations based on angles between control vectors to determine t_i

We now explain Algorithm 2 in detail. The key concept introduced here is the **turning angle** of a quadratic curve, defined as the angle between the control vectors $\mathbf{P}_0\mathbf{P}_1$ and $\mathbf{P}_1\mathbf{P}_2$, denoted by φ as shown in Fig. 4. Specifically,

$$\varphi = \arccos \frac{\langle \mathbf{P}_0\mathbf{P}_1, \mathbf{P}_1\mathbf{P}_2 \rangle}{|\mathbf{P}_0\mathbf{P}_1| \cdot |\mathbf{P}_1\mathbf{P}_2|}, \tag{13}$$

where $\langle \cdot, \cdot \rangle$ denotes the dot product. Thus $0 \le \varphi \le \pi$, and from the geometric meaning of curvature (the rate of change of tangent angle), it is easy to see that φ is essentially the integral of curvature over the quadratic curve. Intuitively, a larger φ indicates a more curved segment.

In fact, according to Yzerman [11], the relative error η of the offset approximation method proposed by Tiller et al. [13] (which we adopt here) depends only on the turning angle φ:

$$\eta = \max_{t\in[0,1]} \frac{|\hat{d}(t) - d|}{d} = \frac{2\sin^4(\frac{\varphi}{4})}{\cos(\frac{\varphi}{2})}, \tag{14}$$

where d is half the stroke width, and $\hat{d}(t)$ is the approximate half stroke width at parameter t, i.e., the distance between $\mathbf{O}_i(t)$ and $\mathbf{B}(t)$. Since η increases monotonically with φ in the range $[0, \pi)$, this matches the intuition that flatter curves can be better approximated. When $\varphi = \frac{\pi}{6}$, the error η is about 0.06%, which is already a very good approximation in practice. By setting the maximum turning angle $\varphi_0 \le \frac{\pi}{6}$ for subcurves, we subdivide highly curved segments into multiple flatter subcurves with equal turning angle φ_0, thus achieving adaptive subdivision guided by curvature.

We use a classic subdivision method in which each subcurve is also a quadratic Bézier curve, and adjacent subcurves are joined with G^1 continuity. Let the control points of the ith subcurve be $\mathbf{Q}_{i,0}, \mathbf{Q}_{i,1}, \mathbf{Q}_{i,2}$, then G^1 continuity implies that $\mathbf{Q}_{i,2} = \mathbf{Q}_{i+1,0}$ and $\mathbf{Q}_{i,1}\mathbf{Q}_{i,2} \parallel \mathbf{Q}_{i+1,0}\mathbf{Q}_{i+1,1}$. Since $\mathbf{Q}_{i,0}$ and $\mathbf{Q}_{i,2}$ lie on the original curve, there exist parameters t_i such that $\mathbf{Q}_{i,0} = \mathbf{B}(t_i)$ and $\mathbf{Q}_{i,2} = \mathbf{B}(t_{i+1})$. The middle control point $\mathbf{Q}_{i,1}$ is computed as the intersection point of the two lines: $\mathbf{Q}_{i,0} + \alpha\mathbf{B}'(t_i)$ and $\mathbf{Q}_{i,2} + \alpha\mathbf{B}'(t_{i+1})$, where $\alpha \in \mathbb{R}$.

In the first step, we compute the minimal number of subdivisions $K = \lceil \frac{\varphi}{\varphi_0} \rceil$ required to satisfy the turning angle constraint. The second step determines the parameters t_i by solving the following system:

$$\begin{cases} \angle(\mathbf{P}_0\mathbf{P}_1, \mathbf{Q}_{i,0}\mathbf{Q}_{i,1}) = i \cdot \varphi_0 \\ \angle(\mathbf{Q}_{i,0}\mathbf{Q}_{i,1}, \mathbf{P}_1\mathbf{P}_2) = \varphi - i \cdot \varphi_0 \end{cases} \tag{15}$$

where $\angle(\cdot, \cdot)$ denotes the angle between two vectors. If t_i and t_{i+1} satisfy the equation system with index i and $i+1$, then the turning angle of the ith subcurve is φ_0. We leave the detailed derivation of t_i in Appendix A. Note that this system depends only on the control vectors $\mathbf{P}_0\mathbf{P}_1$ and $\mathbf{P}_1\mathbf{P}_2$, the overall turning angle φ, the angle constraint φ_0, and the subcurve index i, so the computations for different subcurves are independent and suitable for GPU parallelization.

This algorithm is efficiently implemented using GPU tessellation shaders [18]. In the tessellation control shader, the output is configured as a single isoline with K segments (each segment corresponds to one quadratic subcurve). In the tessellation evaluation shader, the angle system above is solved to compute each parameter t. Since the output from the tessellation stage to the geometry shader consists of pairs of endpoints $\mathbf{Q}_{i,0}$ and $\mathbf{Q}_{i,2}$, the geometry shader computes the intermediate control point $\mathbf{Q}_{i,1}$ by line intersection, and the rendering proceeds using the method described in Section 4.1.

Thanks to the curvature-guided subdivision strategy, which explicitly enforces G^1 continuity at the shared endpoints between subdivided quadratic segments, our method achieves smooth visual results within the interior of the curve. However, at endpoints where two distinct quadratic curves are connected, visual gaps may appear if G^1 continuity is not maintained. In practice, these points correspond to "joins" in the SVG specification, which defines explicit styles (e.g., miter, round, bevel) for rendering such junctions. While our current focus is on accurate rendering of continuous stroke regions, supporting such join styles is planned as future work.

4.3. Arc-length parameterization method

This section presents an algorithm for computing the arc-length parameter for quadratic Bézier curves using the Newton iteration method.

The general framework of the arc-length parameter computation based on Newton's method is described in Algorithm 3.

Algorithm 3 Newton-based Arc-Length Parameter Computation

Input: Control points $\mathbf{P}_0$, $\mathbf{P}_1$ and $\mathbf{P}_2$ of a quadratic Bézier curve; target arc length S; maximum allowable error ϵ

Output: A parameter t such that the cumulative arc length $s(t)$ differs from the input S by no more than ϵ

1: Compute the initial guess of t for Newton's iteration
2: Perform Newton iteration to update t until the error tolerance is satisfied

The Newton iteration update rule is given by:

$$t_{n+1} = t_n + \frac{S - s(t_n)}{s'(t_n)}, \quad \forall n \in \mathbb{N}, \tag{16}$$

which quickly approximates the solution to the equation $s(t) - S = 0$. Since $s(t_n)$ can be computed explicitly from the properties of quadratic Bézier curves and $s'(t_n) = |\mathbf{B}'(t_n)|$ is easy to evaluate, this method is highly suitable for arc-length parameterization of quadratic curves. As is well known, the convergence rate of Newton's method strongly depends on the choice of the initial guess, making Step 1 of Algorithm 3 particularly important. We now present a method for selecting an initial value that ensures rapid convergence, along with an associated error estimate.

4.3.1. Initial value selection and error estimation

We define the following quadratic approximation to the arc-length function:

$$\tilde{s}(t) = -\frac{1}{2}s''(0.5)(1-t)t + lt, \tag{17}$$

where $l = s(1)$ is the total arc length of the curve, and $s''(0.5)$ is the second derivative of $s(t) = \int_0^t |\mathbf{B}'(\tau)|d\tau$ evaluated at $t = 0.5$. Given a target arc length $0 \le S \le l$, we determine the initial value t_0 for Newton iteration by solving the quadratic equation:

$$\tilde{s}(t) = S. \tag{18}$$

Since $\tilde{s}(0) = 0$ and $\tilde{s}(1) = l$, the intermediate value theorem guarantees that at least one solution to Eq. (18) exists in the interval $[0, 1]$. If multiple solutions exist, any one may be selected as the initial guess.

Here we briefly explain the intuition behind this initial value selection. Traditional arc-length parameterization often uses the linear function $f(t) = lt$ to approximate $s(t)$, which satisfies interpolation conditions at $t = 0$ and $t = 1$, and is easy to invert to find t from a given S. To improve accuracy while retaining ease of inversion, we add a quadratic correction to the linear approximation: $C \cdot (1-t)t \approx s(t) - lt$. Heuristically choosing $C = -\frac{1}{2}s''(0.5)$ and adding it to the linear term yields the quadratic approximation in Eq. (17). Then we provide an error estimate for this initial value method:

Theorem 1. *Let $0 \le S \le l$, and let $t_0 \in [0, 1]$ satisfy the quadratic equation $\tilde{s}(t_0) = S$. Then the following error bound holds:*

$$|s(t_0) - S| \le \frac{1}{8}\varphi \cdot |\mathbf{B}''(0)|, \tag{19}$$

where φ is the turning angle of the quadratic Bézier curve $\mathbf{B}(t)$.

For the proof of Theorem 1, we refer to Appendix B. Next, we estimate the number of iterations required for Newton's method to converge when using this initial value.

Theorem 2. *Assume the quadratic Bézier curve satisfies the regularity condition:*

$$\frac{1}{2} \le \frac{|\mathbf{P}_0\mathbf{P}_1|}{|\mathbf{P}_1\mathbf{P}_2|} \le 2, \text{ and } \varphi \le \frac{\pi}{6}. \tag{20}$$

Table 1
Upper bound on the number of Newton iterations.

$\lvert B'(0)\rvert$	Max. Iterations Required
100	2.0764
1000	2.3840
10000	2.6374

Then for any given $\epsilon > 0$, the Newton iteration t_{n+1} satisfies:

$$|s(t_{n+1}) - S| \le \epsilon, \forall n + 1 \ge \log_2 \frac{\log_2 \frac{2|\mathbf{B}'(0)|}{\sqrt{5-2\sqrt{3}\epsilon}}}{\log_2 \frac{48}{\pi(5-2\sqrt{3})}}. \tag{21}$$

For the proof of Theorem 2, we refer to Appendix C. Theorem 2 provides an upper bound on the number of Newton iterations required for convergence. This bound depends on $|\mathbf{B}'(0)|$ (which is twice the length of the control edge $\mathbf{P}_0\mathbf{P}_1$) and the target tolerance ϵ: the larger the edge length or the smaller the error tolerance, the more iterations are needed. However, because the bound involves two logarithmic layers, its growth is very slow with respect to $|\mathbf{B}'(0)|$ and ϵ. For instance, when $\epsilon = 0.01$, we compute the upper bounds shown in Table 1. The table demonstrates that with the proposed initial guess, Newton's method **always converges within 3 steps** in most practical cases (e.g., when rendering quadratic curves on a 1920×1080 screen, $|\mathbf{P}_0\mathbf{P}_1| = \frac{1}{2}|\mathbf{B}'(0)|$ typically does not exceed $1920 + 1080 = 3000$).

Theorem 2 assumes that the quadratic curve satisfies certain regularity conditions. The condition $\varphi \le \frac{\pi}{6}$ can be ensured using the curvature-guided adaptive subdivision algorithm proposed in this paper. The ratio condition $\frac{1}{2} \le \frac{|\mathbf{P}_0\mathbf{P}_1|}{|\mathbf{P}_1\mathbf{P}_2|} \le 2$ ensures the control edges are not disproportionately long or short. For general quadratic curves that do not satisfy the control edge ratio condition, we show in Proposition 1 that subdividing the curve at $t = 0.5$ yields two subcurves with improved control edge ratios that more closely fall within the interval $[\frac{1}{2}, 2]$. The detailed proof is provided in Appendix D. Proposition 1 states that if the control edge ratio of a quadratic curve is too small ($A < \frac{1}{2}$), subdividing at $t = 0.5$ increases the ratio in one of the subcurves by a factor of $\frac{4}{3}$ while the other subcurve automatically satisfies the desired range $[\frac{1}{2}, 2]$. Conversely, if the original ratio is too large ($A > 2$), subdivision reduces the ratio in one subcurve by a factor of $\frac{3}{4}$, and again ensures the other falls within the target interval.

Proposition 1. *Assume $\varphi < \frac{\pi}{2}$. Consider subdividing the quadratic Bézier curve at $t = 0.5$. Let A denote the ratio $\frac{|\mathbf{P}_0\mathbf{P}_1|}{|\mathbf{P}_1\mathbf{P}_2|}$ of the quadratic Bézier curve, B, C denote the corresponding ratios in two subdivided curves. If $A < \frac{1}{2}$, then we have $1 \ge B \ge \frac{4}{3}A$ and $\frac{3}{4} \ge C \ge \frac{1}{2}$; If $A > 2$, then $2 \ge B \ge \frac{4}{3}$ and $\frac{3}{4}A \ge C \ge 1$.*

Since the only requirement for applying Proposition 1 is that the turning angle $\varphi < \frac{\pi}{2}$ and because subdivision always reduces the turning angles, the proposition can be recursively applied to the subcurve with out-of-range ratio until satisfy the condition. As a result, we can obtain an upper bound for the subdivision step number n by solving $(\frac{4}{3})^n A \ge \frac{1}{2}$ if $A < \frac{1}{2}$, or $(\frac{3}{4})^n A \le 2$ if $A > 2$. However, we treat this result primarily as a theoretical guarantee and do not apply such subdivision in practice, as empirical evidence indicates that Newton's method converges rapidly even without it.

4.3.2. Dashed line rendering method

The dashed line style is determined by two parameters, l_1 and l_2, where l_1 denotes the length of each solid segment, and l_2 represents the gap length between two solid segments. Specifically, given l_1, l_2, and a path length l, the path consists of $\lceil \frac{l}{l_1+l_2} \rceil$ solid segments. The kth solid segment starts at arc length $k(l_1 + l_2)$ and ends at arc length $\min\{k(l_1+l_2)+l_2, l\}$, for $0 \le k < \lceil \frac{l}{l_1+l_2} \rceil$. To determine the endpoints of

these solid segments, we need to convert arc length S into Bézier curve parameter t using arc-length parameterization. This section applies the proposed arc-length parameterization method to dashed-line rendering to achieve a parallel and efficient rendering approach.

According to the error analysis of the proposed arc-length parameterization method (Theorems 1, 2), the method converges rapidly when the turning angle φ of a quadratic curve satisfies $\varphi \le \frac{\pi}{6}$. Therefore, in conjunction with the curvature-guided adaptive subdivision method for quadratic curves presented in Section 4.2, the dashed-line rendering problem is decomposed into a set of parallel subcurve rendering problems. Specifically (with notation as in Section 4.2), suppose the curve is divided into K subcurves, and the ith subcurve has control points $\mathbf{Q}_{i,0}, \mathbf{Q}_{i,1}, \mathbf{Q}_{i,2}$. For each subcurve, we compute the accumulated arc lengths $s(t_i), s(t_{i+1})$ at the endpoints $\mathbf{Q}_{i,0}, \mathbf{Q}_{i,2}$ of the original curve, from which we obtain the total length $l_{\text{total},i} = s(t_{i+1}) - s(t_i)$ and the upper bound on the number of solid segments as $\lceil \frac{l_{\text{total},i}}{l_1+l_2} \rceil$. Unlike the original curve, the first solid segment's (relative) starting arc length $s_{\text{start},0}$ and its length $l_{\text{solid},0}$ in each subcurve need to be specially handled:

$$s_{\text{start},0} = \begin{cases} 0, & \text{if } m \le l_1, \\ l_1 + l_2 - m, & \text{if } m > l_1, \end{cases} \tag{22}$$

$$l_{\text{solid},0} = \begin{cases} \min\{l_1 - m, l_{\text{total},i} - s_{\text{start},0}\}, & \text{if } m \le l_1, \\ \min\{l_1, l_{\text{total},i} - s_{\text{start},0}\}, & \text{if } m > l_1, \end{cases} \tag{23}$$

where m is the remainder of $s(t_i)$ divided by $l_1 + l_2$, defined as:

$$m = s(t_i) - (l_1 + l_2)\lfloor \frac{s(t_i)}{l_1 + l_2} \rfloor. \tag{24}$$

Next, we compute the starting arc length $s_{\text{start},k}$ and the solid segment length $l_{\text{solid},k}$ for the kth solid segment:

$$s_{\text{start},k} = s_{\text{start},0} + l_{\text{start},0} + l_2 + (k-1)(l_1 + l_2), \tag{25}$$

$$l_{\text{solid},k} = \min\{l_1, l_{\text{total},i} - s_{\text{start},k}\}. \tag{26}$$

The arc length of the kth solid segment's endpoint is given by $s_{\text{end},k} = s_{\text{start},k} + l_{\text{solid},k}$. This information is sufficient to render the dashed segments within each subcurve.

Note that Eqs. (25) and (26) depend only on the segment index k, allowing parallel computation of the start and end parameters of each segment within a subcurve. Additionally, the curvature-guided subdivision of the curve into subcurves (as described in Section 4.2) is also parallelizable, making the proposed dashed-line rendering method highly efficient.

From an implementation perspective, this algorithm leverages the GPU's tessellation shader functionality: the tessellation control shader computes necessary data such as t_i, $s(t_i)$, and $l_{\text{total},i}$; the output primitive is set to K isolines, each subdivided into $2\max_i \lceil \frac{l_{\text{total},i}}{l_1+l_2} \rceil$ segments (the kth isoline's $2i$-th segment corresponds to the ith solid segment of the kth quadratic subcurve), and all subcurves are subdivided into the same number of segments because tessellation shaders require consistent subdivision counts. In the tessellation evaluation shader, the arc-length parameterization method is used together with Eqs. (25) and (26) to compute the Bézier parameter t for each segment. Rendering follows the same approach as in Section 4.2: the geometry shader performs a line-line intersection to compute control points, and rendering is carried out using the method from Section 4.1. Aside from the sequential calculation of t_i, $s(t_i)$, and $l_{\text{total},i}$ for $0 \le i < K = \lceil \frac{\varphi}{\varphi_0} \rceil$, all other steps are parallelized, ensuring high efficiency.

One practical limitation arises from hardware constraints on the tessellation shader. Ideally, we set the tessellation levels gl_TessLevelOuter[0] to $\lceil \frac{\varphi}{\varphi_0} \rceil$ and gl_TessLevelOuter[1] to $2\max_i \lceil \frac{l_{\text{total},i}}{l_1+l_2} \rceil$, in order to subdivide the curve into flatter subcurves and tessellate each into solid segments that form the desired dashed pattern. However, gl_TessLevelOuter is subject to a maximum value, which limits the number of solid segments that can be generated for each subcurve. As a result, rendering long strokes or high-frequency dash patterns may exceed this limit, causing missing segments. To address this, long strokes can be further split into smaller substrokes, each containing a limited number of dashes. These can then be rendered as separate quadratic curves, ensuring compatibility with hardware limits while preserving visual fidelity.

4.3.3. Texture filling method

This section constructs a texture mapping for the approximated stroke region described in Section 4.1, using Newton's method to compute the texture coordinates of each pixel and thereby achieves a texture-filled effect. Additionally, the arc-length parameterization method is applied to enable arc-length-guided texture mapping. We present a framework for texture filling in the approximated stroke region, as described in Algorithm 4.

Algorithm 4 Texture Filling for Approximated Stroke Region

Input: Control points $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2$ of a quadratic Bézier curve; stroke width w; pixel coordinate $\mathbf{P}$; texture image $\mathbf{T}$

Output: Texture coordinates (u, v) of $\mathbf{P}$; the color $\mathbf{C}$ sampled from the texture image

1: Construct the texture mapping $\boldsymbol{\psi}$
2: Use Newton's method to solve $\mathbf{P} = \boldsymbol{\psi}(u, v)$
3: Sample the texture image to obtain $\mathbf{C} = \mathbf{T}(u, v)$

We now explain the first step of the algorithm, namely the construction of the texture mapping. Let $Q = [0,1] \times [0,1]$ be the unit square, and let Ω be the approximated stroke region from Section 4.1. For convenience, we denote $\mathbf{d}_i = \mathbf{P}_i\mathbf{P}'_i = \mathbf{P}''_i\mathbf{P}_i$ for $0 \le i < 3$. The texture mapping $\boldsymbol{\psi} : Q \to \Omega$ is defined as:

$$\begin{aligned} \boldsymbol{\psi}(u,v) =& (1-u)^2(\mathbf{P}_0 + (2v-1)\mathbf{d}_0) + \\ & 2u(1-u)(\mathbf{P}_1 + (2v-1)\mathbf{d}_1) + \\ & u^2(\mathbf{P}_2 + (2v-1)\mathbf{d}_2), \end{aligned} \tag{27}$$

where $(u, v) \in Q$, and for any fixed $v \in [0,1]$, $\boldsymbol{\psi}(\cdot, v)$ represents a quadratic Bézier curve with control points $\mathbf{P}_i + (2v-1)\mathbf{d}_i$ for $0 \le i < 3$. It can be seen that the line segment $v = 0$ is mapped to the approximate offset curve $\mathbf{O}_2(t), t \in [0,1]$, $v = 0.5$ maps to the original curve $\mathbf{B}(t), t \in [0,1]$, and $v = 1$ maps to $\mathbf{O}_1(t), t \in [0,1]$.

In the second step of the algorithm, Newton's method is employed. The initial value Init($\mathbf{P}$) for the iteration is chosen as a piecewise linear function, which is linear within the triangles $\mathbf{P}'_0\mathbf{P}'_1\mathbf{P}'_2$, $\mathbf{P}'_0\mathbf{P}'_2\mathbf{P}''_0$, and $\mathbf{P}''_0\mathbf{P}'_2\mathbf{P}''_2$. It is uniquely defined by the values at the triangle vertices: Init($\mathbf{P}''_0$) = $(0,0)$, Init($\mathbf{P}'_0$) = $(0,1)$, Init($\mathbf{P}'_1$) = $(0.5,1)$, Init($\mathbf{P}'_2$) = $(1,1)$, Init($\mathbf{P}''_2$) = $(1,0)$.

Next, the Newton iteration is:

$$(u_{n+1}, v_{n+1})^T = \boldsymbol{J}_{\boldsymbol{\psi}}^{-1}(\mathbf{P} - \boldsymbol{\psi}(u_n, v_n)) + (u_n, v_n)^T, \tag{28}$$

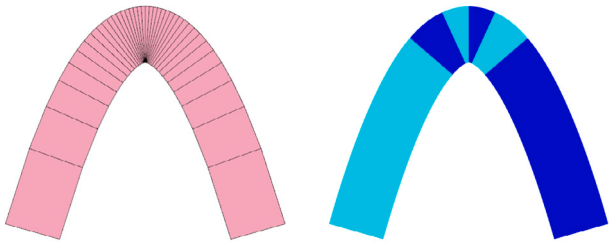
where $\boldsymbol{J}_{\boldsymbol{\psi}}$ is the Jacobian matrix of $\boldsymbol{\psi}$ evaluated at (u_n, v_n), and (u_0, v_0) = Init($\mathbf{P}$). Empirically, just one iteration (i.e., (u_1, v_1)) usually yields satisfactory texture-mapping results.

The Jacobian matrix can be explicitly computed:

$$\begin{aligned} \frac{\partial \boldsymbol{\psi}}{\partial u} =& 2(1-u)(\mathbf{P}_0\mathbf{P}_1 + (2v-1)\mathbf{d}_0\mathbf{d}_1) + \\ & 2u(\mathbf{P}_1\mathbf{P}_2 + (2v-1)\mathbf{d}_1\mathbf{d}_2), \end{aligned} \tag{29}$$

$$\frac{\partial \boldsymbol{\psi}}{\partial v} = 2((1-u)^2\mathbf{d}_0 + 2u(1-u)\mathbf{d}_1 + u^2\mathbf{d}_2). \tag{30}$$

The above texture-filling method performs well when the aspect ratio (length-to-width) of the stroke region is close to 1. However, when there is a significant aspect ratio imbalance, the texture may appear distorted. To address this, we apply the arc-length-guided segmentation method from Section 4.3.2 to divide the curve into subregions with an aspect ratio of 1 by setting $l_1 = w$. Each of these subregions is then texture-mapped using the method above to reduce distortion.



(a) Polar Stroking [9]'s tessellation. (b) Proposed tessellation.

Fig. 5. Comparison of tessellation methods. (a) shows the tessellation result of Polar Stroking [9] which consists of 38 trapezoids. (b) shows our tessellation result which consists of 6 approximated quadratic strokes. The adjacent tessellated segments are colored differently.

However, this segmentation may also generate some regions — especially around high curvature areas — with aspect ratios less than 1. We handle these cases as follows. Let the aspect ratio of a subregion be $\frac{l}{w} < 1$. Then the parameter domain $Q' = [0, \frac{l}{w}] \times [0, 1]$ matches the aspect ratio of the stroke region. Let $(u', v') = (\frac{l}{w}u, v)$ for $(u, v) \in Q$, so that $\boldsymbol{\psi}(u', v')$ becomes a texture mapping from Q' to Ω. Since Q' has the same aspect ratio as Ω, $\boldsymbol{\psi}(u', v')$ introduces less distortion than $\boldsymbol{\psi}(u, v)$. By accordingly adjusting the initialization and Jacobian computation, we solve the equation $\boldsymbol{\psi}(u', v') = \mathbf{P}$ using Newton's method and sample the texture at (u', v') to reduce distortion.

This texture-filling algorithm can be implemented using the fragment shader on the GPU. Moreover, to ensure continuity of texture coordinates at the seams between subregions, the horizontal coordinate u (or u') must be translated by a length $l_o = \frac{s}{w}$, where s is the cumulative arc length of the subcurve's starting point on the original curve.

5. Experimental results and comparisons

All experiments presented in this section were conducted under the same hardware and software environment. The operating system used was Windows 11 Home Edition, with a CPU of AMD Ryzen 7 7840H (3.80 GHz), a GPU of NVIDIA GeForce RTX 4060 Laptop GPU, and 16 GB of memory. The proposed algorithm was implemented using OpenGL (version 4.6) with GPU parallelization.

5.1. Rendering quality of the proposed algorithm

Fig. 5 visually compares the proposed quadratic curve stroke tessellation method with Kilgard's Polar Stroking method [9]. Both are curvature-guided techniques; however, Kilgard's method tessellates the stroke region into multiple trapezoids, while our method approximates the stroke using multiple approximated quadratic curve strokes (as described in Sections 4.1 and 4.2). Because our tessellation primitives are inherently curved, they better represent curved regions and require fewer primitives for the same stroke area. For example, to render a particular stroke, Kilgard's method [9] uses 38 trapezoids with an inter-segment angle of 4°, while our method achieves a similar result with only 6 approximated quadratic curve strokes using a turning angle of approximately 28° per subcurve. Fig. 6 demonstrates the rendering results of strokes with different widths. Notably, our method can accurately render extremely thin strokes as narrow as a single pixel.

Fig. 7 shows the dashed stroke rendering results using our algorithm, and Fig. 8 demonstrates the anti-aliasing performance. Our approach combines the signed-distance-based anti-aliasing method proposed by Loop [4] with 8×MSAA (Multisample Anti-Aliasing) [18] for high-quality rendering.

Table 2
Average rendering time (ms) for solid stroke paths.

w	Skia	NVpr	Polar Stroking	Vello	Ours
100	103	44.9	1982	9.8	**4.6**
50	102	33.5	1994	8.5	**3.0**
10	91	24.3	1996	7.5	**1.4**

Table 3
Average rendering time (ms) for dashed stroke paths.

(l_1, l_2)	Skia	NVpr	Polar Stroking	Vello	Ours
(100,100)	140	24.5	2029	8.9	**3.2**
(50,50)	177	23.7	2121	10.6	**4.1**
(10,10)	445	21.3	2645	17.4	**8.4**

Fig. 9 presents the results of texture filling using our method, which incorporates arc-length information. Fig. 10 visualizes the texture coordinate grids computed per pixel. Using Newton's method with a piecewise linear initialization strategy, our method achieves visually accurate texture coordinates with only one iteration.

5.2. Rendering performance comparison

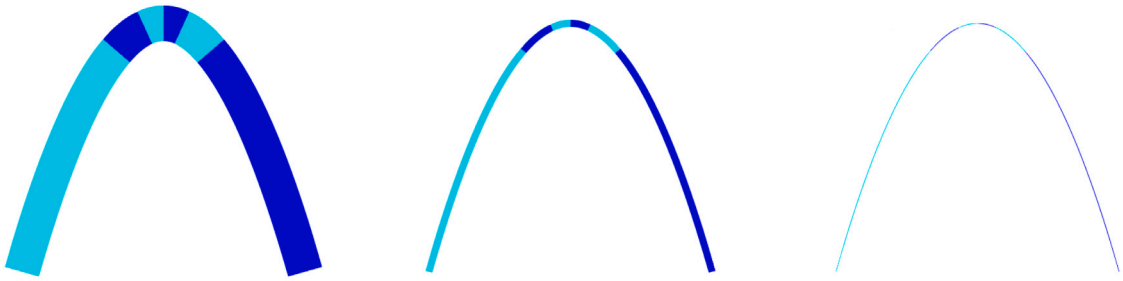
This section compares the rendering performance of our method with that of Skia [19], NVpr [10], Polar Stroking [9] and Vello [3]. Skia (version m115), widely used in applications such as the Chrome browser, performs most of its rendering via CPU texture generation, as revealed by frame captures using RenderDoc. NVpr [10] is a classic GPU-based rendering method built into NVIDIA's GPU driver and tested using the official SDK [20]. The Polar Stroking method [9] was evaluated using the released demo code; however, its curve tessellation is implemented on the CPU, and a GPU implementation is currently unavailable. Levien et al. [3] present GPU stroke expansion algorithms implemented in the Vello rendering library. Their methods are evaluated using the publicly available source code released on GitHub [21–23]. The NVpr, Polar Stroking, Vello and our method all use MSAA for anti-aliasing. For fair comparison, we used 8×MSAA in these tested methods.

Table 2 reports the average rendering times for 2000 quadratic stroke paths of varying widths. Our method achieves speedups of 2.1×, 2.8×, and 5.4× over the best competing method for widths of 100, 50, and 10 pixels, respectively. Table 3 compares the rendering times of 2000 dashed stroke paths with a fixed width of 100 pixels but varying dash patterns. Our method achieves speedups of 2.8×, 2.6×, and 2.1× for dash patterns of (100,100), (50,50), and (10,10), respectively. These results demonstrate the high efficiency of our algorithm.

6. Conclusion and future work

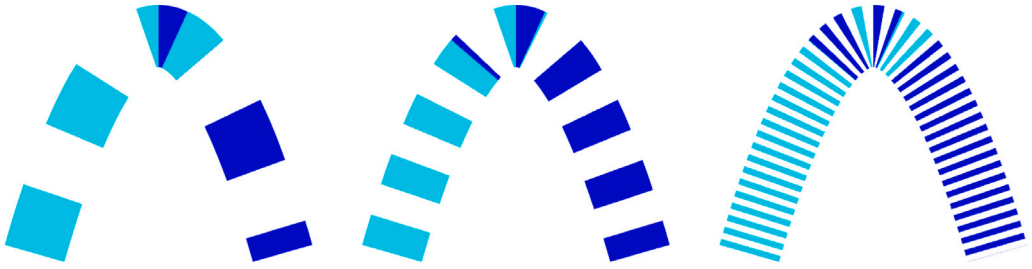
This paper presents a resolution-independent GPU-accelerated algorithm for approximating and rendering quadratic stroke regions based on offset curves approximation and implicit equations. We propose a curvature-guided, adaptive subdivision method that runs on the GPU and requires fewer segments than trapezoidal subdivision while maintaining approximation accuracy. An efficient arc-length parameterization method for quadratic curves is introduced, using Newton iteration with a carefully designed initial guess strategy to ensure fast convergence, along with error estimation. The algorithm is further extended to support dashed stroke rendering, with a parallel computation scheme for determining the start and end parameters of each solid segment. We also construct a stroke region texture mapping technique, and propose an arc-length-guided texture filling method that works in conjunction with our parameterization scheme.

Despite its advantages, the proposed method has some limitations. The parallel rendering of dashed strokes relies on the use of tessellation shaders, which are subject to GPU hardware constraints. For example,



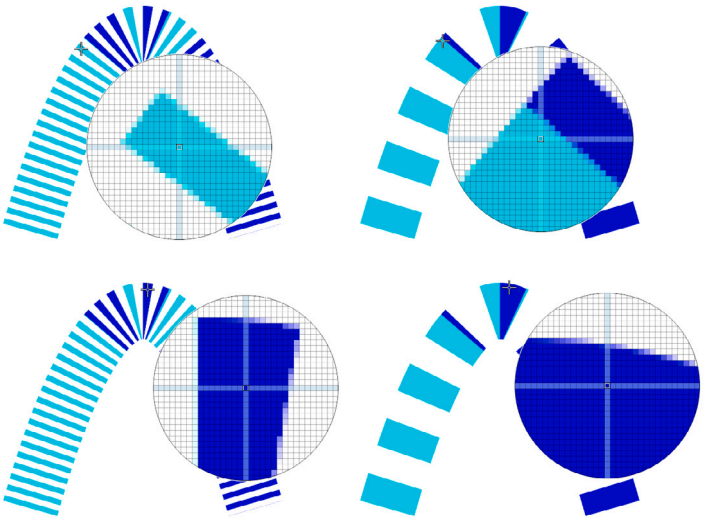
(a) $w = 50$. (b) $w = 10$. (c) $w = 1$.

Fig. 6. Rendering results for strokes of varying widths. (a), (b), (c) show rendered strokes with width of 50 pixels, 10 pixels, 1 pixel, respectively.



(a) $(l_1, l_2) = (100, 100)$. (b) $(l_1, l_2) = (50, 50)$. (c) $(l_1, l_2) = (10, 10)$.

Fig. 7. Dashed stroke rendering results. (a), (b), (c) show rendered dashed strokes (100 pixels width) with varying dash pattern $(l_1, l_2) = (100, 100), (50, 50), (10, 10)$ respectively.

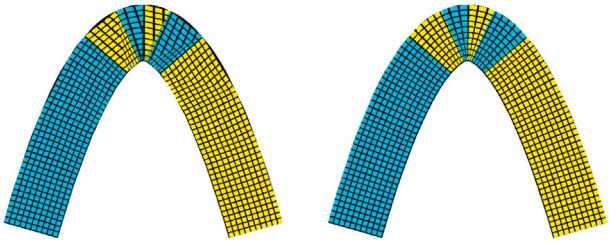


(a) $(l_1, l_2) = (10, 10)$. (b) $(l_1, l_2) = (50, 50)$.

Fig. 8. Anti-aliasing results. (a), (b) show the anti-aliasing effect of the rendered dashed stroke (100 pixels width) with dash pattern $(l_1, l_2) = (10, 10), (50, 50)$, respectively. Note that for pixels on the boundary of rendered regions, the transparency is gradually changed to alleviate the jagged, stair-step appearance of edges.

on an RTX 4060 laptop GPU, the maximum number of solid segments that can be rendered per subcurve is limited to 32 (the maximum gl_TessLevelOuter[i] is 64); exceeding this number may lead to rendering artifacts. Future work may involve splitting long strokes into smaller substrokes with fewer dashes to maintain compatibility with hardware-imposed tessellation limits. To support more complex stroke paths (e.g., cubic Bézier curves) and correct global stroke rendering results described in [2], additional preprocessing, such as approximating general path segments with quadratic curves of bounded curvature, is required. In future work, we plan to extend our method to handle more complex stroke paths, support more stroke styles (e.g., cap and join types), and enable full stroke rendering of SVG path data while

Fig. 9. Texture filling results. The distortion of texture map is minimized by the guidance of arc length.



(a) (u_0, v_0). (b) (u_1, v_1).

Fig. 10. Visualization of texture coordinate isoparameteric curves involved in Newton iteration. (a) shows the isoparameteric curves of the piecewise-linearly interpolated initial guess (u_0, v_0), while (b) shows the isoparameteric curves of (u_1, v_1) which is obtained by the first iteration of the Newton's method. Note that a satisfactory texture filling accuracy is achieved with just 1 Newton iteration.

conducting a detailed error analysis of the bounded-curvature quadratic approximation process.

CRediT authorship contribution statement

Xuhai Chen: Writing – original draft, Visualization, Software, Methodology. **Guangze Zhang:** Visualization, Software, Methodology. **Wanyi Wang:** Visualization, Software, Methodology. **Juan Cao:** Writing – review & editing, Supervision, Project administration, Funding acquisition, Conceptualization. **Zhonggui Chen:** Writing – review & editing, Supervision, Project administration, Methodology, Funding acquisition, Conceptualization.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the author(s) used ChatGPT in order to polish language. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the National Key R&D Program of China (No. 2022YFB3303400), National Natural Science Foundation of China (Nos. 62272402, 62372389), Natural Science Foundation of Fujian Province, China (No. 2024J01513243), Special Fund for Key Program of Science and Technology of Fujian Province, China (No. 2022YZ040011), and Fundamental Research Funds for the Central Universities, China (No. 20720220037).

Appendix A. Derivation of subdivision parameters

In this section, we determine the parameter t_i that uniquely solves the following system:

$$\begin{cases} \angle(\mathbf{P}_0\mathbf{P}_1, \mathbf{Q}_{i,0}\mathbf{Q}_{i,1}) = i \cdot \varphi_0 \\ \angle(\mathbf{Q}_{i,0}\mathbf{Q}_{i,1}, \mathbf{P}_1\mathbf{P}_2) = \varphi - i \cdot \varphi_0 \end{cases} \tag{31}$$

where $\angle(\cdot,\cdot)$ denotes the angle between two vectors.

The first equation in the system has the explicit solution:

$$t_1 = \frac{A\left(A - AD^2 - BC + \sigma_2 + BCD^2\right)}{\sigma_1}, \tag{32}$$

$$t_2 = \frac{A\left(A - AD^2 - BC - \sigma_2 + BCD^2\right)}{\sigma_1}, \tag{33}$$

where

$$A = 2|\mathbf{P}_0\mathbf{P}_1|, \tag{34}$$

$$B = 2|\mathbf{P}_1\mathbf{P}_2|, \tag{35}$$

$$C = \cos\varphi, \tag{36}$$

$$D = \cos(i \cdot \varphi_0), \tag{37}$$

$$\sigma_1 = (A^2 - 2ABC)(1 - D^2) + B^2(C^2 - D^2), \tag{38}$$

$$\sigma_2 = BD\sqrt{(C^2 - 1)(D^2 - 1)}. \tag{39}$$

Among t_1 and t_2, only one satisfies the second equation in the system, and that is the unique solution to the system.

Appendix B. Proof of Theorem 1

Proof. Note that $f(t) = lt$ is the linear Lagrange interpolation of $s(t)$ at $t = 0$ and $t = 1$. By the Lagrange interpolation error formula:

$$s(t) - lt = -\frac{1}{2}s''(\xi)(1 - t)t, \tag{40}$$

for some $\xi \in (0, 1)$ depending on t. That is,

$$s(t) = -\frac{1}{2}s''(\xi)(1 - t)t + lt. \tag{41}$$

Subtracting Eq. (17) from Eq. (41) and taking the absolute value yields:

$$|\tilde{s}(t) - s(t)| = \frac{1}{2}(1 - t)t \cdot |s''(\xi) - s''(0.5)|. \tag{42}$$

We now estimate $|s''(\xi) - s''(0.5)|$. Observe that:

$$s''(t) = \frac{d}{dt}\left(\langle \mathbf{B}'(t), \mathbf{B}'(t)\rangle^{\frac{1}{2}}\right) = \frac{\langle \mathbf{B}''(t), \mathbf{B}'(t)\rangle}{|\mathbf{B}'(t)|} \tag{43}$$

$$= |\mathbf{B}''(t)|\cos\beta(t) = |\mathbf{B}''(0)|\cos\beta(t), \tag{44}$$

where $\beta(t)$ is the angle between $\mathbf{B}''(t)$ and $\mathbf{B}'(t)$. Since $\mathbf{B}(t)$ is a quadratic curve, $\mathbf{B}''(t) \equiv \mathbf{B}''(0)$ is constant.

Thus,

$$|s''(\xi) - s''(0.5)| = |\mathbf{B}''(0)| \cdot |\cos\beta(\xi) - \cos\beta(0.5)| \tag{45}$$

$$\leq |\mathbf{B}''(0)| \cdot (\cos\beta(1) - \cos\beta(0)), \tag{46}$$

$$\leq |\mathbf{B}''(0)| \cdot (\beta(0) - \beta(1)), \tag{47}$$

where (46) follows from the fact that $\cos\beta(t)$ is monotonically increasing on $[0, 1]$ (as $\frac{d}{dt}\cos\beta(t) = \frac{|\mathbf{B}''(0)|}{|\mathbf{B}'(t)|}\sin^2\beta(t) > 0$), and (47) follows from the Mean Value theorem for the cosine function that $\exists\xi_0$, s.t. $|\cos x - \cos y| = |-\sin\xi_0 \cdot (x - y)| \leq |x - y|$.

Finally, from the geometric relation illustrated in Fig. 11, we know:

$$\beta(0) - \beta(1) = \varphi. \tag{48}$$

Combining (42), (47), and (48), we obtain:

$$|\tilde{s}(t) - s(t)| \leq \frac{1}{2}(1 - t)t \cdot |\mathbf{B}''(0)| \cdot \varphi. \tag{49}$$

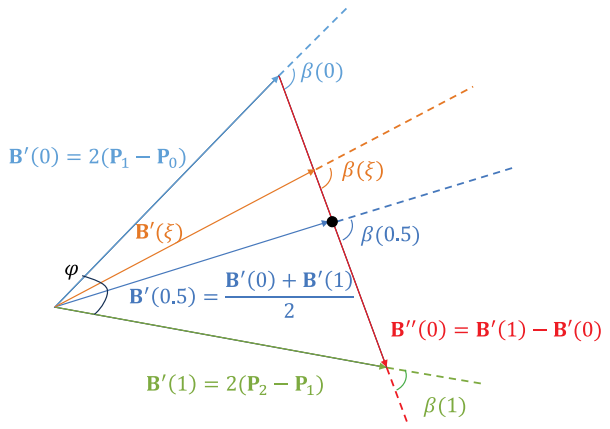


Fig. 11. Geometric relations among the control edges of a quadratic Bézier curve. $\beta(t)$ is the angle between $\mathbf{B}''(t)$ and $\mathbf{B}'(t)$. As $\mathbf{B}(t)$ is quadratic, $\mathbf{B}'(t)$ becomes the linear combination of $\mathbf{B}'(0) = 2\mathbf{P}_0\mathbf{P}_1$ and $\mathbf{B}'(1) = 2\mathbf{P}_1\mathbf{P}_2$. Moreover, $\mathbf{B}''(t) \equiv \mathbf{B}''(0) = \mathbf{B}'(1) - \mathbf{B}'(0)$ is a constant.

Since $\frac{1}{2}(1-t)t$ achieves its maximum value of $\frac{1}{8}$ on $[0,1]$, substituting $t = t_0$ gives:

$$|s(t_0) - S| \le \frac{1}{8}\varphi \cdot |\mathbf{B}''(0)|. \quad \square \tag{50}$$

Appendix C. Proof of Theorem 2

Proof. From the assumed regularity conditions and by geometric analysis (see Fig. 11), we obtain:

$$|\mathbf{B}''(0)| \le \sqrt{5-2\sqrt{3}}|\mathbf{B}'(0)|, \tag{51}$$

$$2 \ge \frac{|\mathbf{B}'(t)|}{|\mathbf{B}'(0)|} \ge \frac{1}{2}, \forall t \in [0,1]. \tag{52}$$

Regarding the error of Newton's method, the classical result [24] states:

$$|t_{n+1} - t^*| = \frac{|s''(c)|}{2|s'(t_n)|}|t_n - t^*|^2, \tag{53}$$

where t^* is the exact solution to $s(t) - S = 0$, and c lies between t_n and t^*. Using Eqs. (44) and (51), we have:

$$|s''(c)| \le |\mathbf{B}''(0)| \le \sqrt{5-2\sqrt{3}}|\mathbf{B}'(0)|. \tag{54}$$

Also, from (52):

$$s'(t_n) = |\mathbf{B}'(t_n)| \ge \frac{1}{2}|\mathbf{B}'(0)|. \tag{55}$$

Substituting (54) and (55) into (53), we get:

$$|t_{n+1} - t^*| \le \frac{\sqrt{5-2\sqrt{3}}|\mathbf{B}'(0)|}{2 \cdot \frac{1}{2}|\mathbf{B}'(0)|}|t_n - t^*|^2 \tag{56}$$

$$= \sqrt{5-2\sqrt{3}}|t_n - t^*|^2, \tag{57}$$

Using mathematical induction:

$$|t_{n+1} - t^*| \le \frac{1}{\sqrt{5-2\sqrt{3}}}\left(\sqrt{5-2\sqrt{3}}|t_0 - t^*|\right)^{2^{n+1}}. \tag{58}$$

Next, we estimate $|t_0 - t^*|$. From Theorem 1, we have:

$$|s(t_0) - s(t^*)| = |s(t_0) - S| \le \frac{\varphi}{8}|\mathbf{B}''(0)|. \tag{59}$$

By the Mean Value Theorem, there exists ξ such that:

$$|s(t_0) - s(t^*)| = s'(\xi)|t_0 - t^*| = |\mathbf{B}'(\xi)\parallel t_0 - t^*|. \tag{60}$$

Combining (59) and (60) gives:

$$|t_0 - t^*| = \frac{|s(t_0) - s(t^*)|}{|\mathbf{B}'(\xi)|} \le \frac{\varphi \cdot |\mathbf{B}''(0)|}{8|\mathbf{B}'(\xi)|}, \tag{61}$$

Applying (51) and (52):

$$|t_0 - t^*| \le \frac{\varphi}{8} \cdot \frac{\sqrt{5-2\sqrt{3}}|\mathbf{B}'(0)|}{2 \cdot \frac{1}{2}|\mathbf{B}'(0)|} \tag{62}$$

$$= \frac{\varphi}{8} \cdot \sqrt{5-2\sqrt{3}}. \tag{63}$$

Substituting this into (58):

$$|t_{n+1} - t^*| \le \frac{1}{\sqrt{5-2\sqrt{3}}}\left(\frac{\varphi}{8} \cdot (5-2\sqrt{3})\right)^{2^{n+1}}, \tag{64}$$

Again, by the Mean Value Theorem, there exists ξ_0 such that:

$$|s(t_{n+1}) - S| = |s(t_{n+1}) - s(t^*)| = |t_{n+1} - t^*| \cdot |\mathbf{B}'(\xi_0)| \tag{65}$$

$$\le 2|\mathbf{B}'(0)| \cdot |t_{n+1} - t^*| \tag{66}$$

$$\le 2|\mathbf{B}'(0)| \cdot \frac{1}{\sqrt{5-2\sqrt{3}}}\left(\frac{\varphi}{8} \cdot (5-2\sqrt{3})\right)^{2^{n+1}} \tag{67}$$

$$\le 2|\mathbf{B}'(0)| \cdot \frac{1}{\sqrt{5-2\sqrt{3}}}\left(\frac{\pi}{48} \cdot (5-2\sqrt{3})\right)^{2^{n+1}}, \tag{68}$$

where the first inequality uses (52), the second uses (64) and the last uses $\varphi \le \frac{\pi}{6}$.

To ensure $|s(t_{n+1}) - S| \le \epsilon$, we simply require the right-hand side of (68) to be no greater than ϵ. Solving yields:

$$n+1 \ge \log_2 \frac{\log_2 \frac{2|\mathbf{B}'(0)|}{\sqrt{5-2\sqrt{3}}\epsilon}}{\log_2 \frac{48}{\pi(5-2\sqrt{3})}}. \quad \square \tag{69}$$

Appendix D. Proof of Proposition 1

Proof. By subdividing the quadratic Bézier curve at $t = 0.5$, we obtain two subcurves. The first one has control points $\mathbf{P}_0, \frac{\mathbf{P}_0+\mathbf{P}_1}{2}, \frac{\mathbf{P}_0+2\mathbf{P}_1+\mathbf{P}_2}{4}$ and the second one has control points $\frac{\mathbf{P}_0+2\mathbf{P}_1+\mathbf{P}_2}{4}, \frac{\mathbf{P}_1+\mathbf{P}_2}{2}, \mathbf{P}_2$. We can therefore calculate the ratio in two subcurves, obtaining $B = 2\frac{|\mathbf{P}_1-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_0|}$ and $C = \frac{1}{2}\frac{|\mathbf{P}_2-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_1|}$.

If $A < \frac{1}{2}$, i.e., $\frac{|\mathbf{P}_1-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_1|} < \frac{1}{2}$, we have

$$|\mathbf{P}_2 - \mathbf{P}_0| \le |\mathbf{P}_1 - \mathbf{P}_0| + |\mathbf{P}_2 - \mathbf{P}_1| \le \frac{3}{2}|\mathbf{P}_2 - \mathbf{P}_1|, \tag{70}$$

then $B = 2\frac{|\mathbf{P}_1-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_0|} \ge 2\frac{|\mathbf{P}_1-\mathbf{P}_0|}{\frac{3}{2}|\mathbf{P}_2-\mathbf{P}_1|} = \frac{4}{3}A$ and $C = \frac{1}{2}\frac{|\mathbf{P}_2-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_1|} \le \frac{1}{2}\frac{\frac{3}{2}|\mathbf{P}_2-\mathbf{P}_1|}{|\mathbf{P}_2-\mathbf{P}_1|} = \frac{3}{4}$.

On the other hand, $\varphi < \frac{\pi}{2}$ implies the triangle $\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2$ is an obtuse triangle, therefore the edge $\mathbf{P}_2 - \mathbf{P}_0$ which corresponds to the obtuse angle is the longest edge. As a result,

$$|\mathbf{P}_2 - \mathbf{P}_0| > |\mathbf{P}_2 - \mathbf{P}_1|, \tag{71}$$

hence $B = 2\frac{|\mathbf{P}_1-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_0|} \le 2\frac{|\mathbf{P}_1-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_1|} = 2A$ and $C = \frac{1}{2}\frac{|\mathbf{P}_2-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_1|} \ge \frac{1}{2}\frac{|\mathbf{P}_2-\mathbf{P}_1|}{|\mathbf{P}_2-\mathbf{P}_1|} = \frac{1}{2}$.

To sum up, if $A < \frac{1}{2}$, then we have $1 \ge 2A \ge B \ge \frac{4}{3}A$ and $\frac{3}{4} \ge C \ge \frac{1}{2}$.

In a similar manner, if $A > 2$, then

$$|\mathbf{P}_2 - \mathbf{P}_0| \le |\mathbf{P}_1 - \mathbf{P}_0| + |\mathbf{P}_2 - \mathbf{P}_1| \le \frac{3}{2}|\mathbf{P}_1 - \mathbf{P}_0|, \tag{72}$$

then $B = 2\frac{|\mathbf{P}_1-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_0|} \ge 2\frac{|\mathbf{P}_1-\mathbf{P}_0|}{\frac{3}{2}|\mathbf{P}_1-\mathbf{P}_0|} = \frac{4}{3}$ and $C = \frac{1}{2}\frac{|\mathbf{P}_2-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_1|} \le \frac{1}{2}\frac{\frac{3}{2}|\mathbf{P}_1-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_1|} = \frac{3}{4}A$.

On the other hand, $\varphi < \frac{\pi}{2}$ implies

$$|\mathbf{P}_2 - \mathbf{P}_0| > |\mathbf{P}_1 - \mathbf{P}_0|, \tag{73}$$

hence $B = 2\frac{|\mathbf{P}_1-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_0|} \le 2\frac{|\mathbf{P}_1-\mathbf{P}_0|}{|\mathbf{P}_1-\mathbf{P}_0|} = 2$ and $C = \frac{1}{2}\frac{|\mathbf{P}_2-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_1|} \ge \frac{1}{2}\frac{|\mathbf{P}_1-\mathbf{P}_0|}{|\mathbf{P}_2-\mathbf{P}_1|} = \frac{1}{2}A$.

To sum up, if $A > 2$, then we have $2 \ge B \ge \frac{4}{3}$ and $\frac{3}{4}A \ge C \ge \frac{1}{2}A \ge 1$. This completes the proof. $\square$

Data availability

Data will be made available on request.

References

[1] W3C, Scalable vector graphics (SVG) 1.1 (second edition), 2011, https://www.w3.org/TR/SVG11/. (Accessed: 2025-05-03).

[2] D. Nehab, Converting stroked primitives to filled primitives, ACM Trans. Graph. 39 (4) (2020) 1–17.

[3] R. Levien, A. Uguray, GPU-friendly stroke expansion, Proc. the ACM Comput. Graph. Interact. Tech. 7 (3) (2024) 1–29.

[4] C. Loop, J. Blinn, Resolution independent curve rendering using programmable graphics hardware, in: ACM SIGGRAPH 2005 Papers, 2005, pp. 1000–1009.

[5] Y. Kokojima, K. Sugita, T. Saito, T. Takemoto, Resolution independent rendering of deformable vector objects using graphics hardware, in: ACM SIGGRAPH 2006 Sketches, 2006, pp. 118–es.

[6] R. Li, Q. Hou, K. Zhou, Efficient GPU path rendering using scanline rasterization, ACM Trans. Graph. 35 (6) (2016) 1–12.

[7] R.T. Farouki, C.A. Neff, Algebraic properties of plane offset curves, Comput. Aided Geom. Design 7 (1–4) (1990) 101–127.

[8] S. Ciao, Z. Guan, Q. Liu, L.-Y. Wei, Z. Wang, Ciallo: GPU-accelerated rendering of vector brush strokes, in: ACM SIGGRAPH 2024 Conference Papers, 2024, pp. 1–11.

[9] M.J. Kilgard, Polar stroking: New theory and methods for stroking paths, ACM Trans. Graph. 39 (4) (2020) 1–15.

[10] M.J. Kilgard, J. Bolz, GPU-accelerated path rendering, ACM Trans. Graph. 31 (6) (2012) 1–10.

[11] F. Yzerman, Precise offsetting of quadratic Bézier curves, 2020.

[12] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, Computational Geometry: Algorithms and Applications, third ed., Springer, 2008.

[13] W. Tiller, E.G. Hanson, Offsets of two-dimensional profiles, IEEE Comput. Graph. Appl. 4 (9) (1984) 36–46.

[14] G. Farin, Curves and Surfaces for CAGD: A Practical Guide, Elsevier, 2001.

[15] G. Elber, I.-K. Lee, M.-S. Kim, Comparing offset curve approximation methods, IEEE Comput. Graph. Appl. 17 (3) (1997) 62–71.

[16] M. Floater, High-order approximation of conic sections by quadratic splines, Comput. Aided Geom. Design 12 (6) (1995) 617–637.

[17] N. Truong, C. Yuksel, L. Seiler, Quadratic approximation of cubic curves, Proc. the ACM Comput. Graph. Interact. Tech. 3 (2) (2020) 1–17.

[18] Khronos Group, OpenGL 4.6 specification, 2022, https://registry.khronos.org/OpenGL/specs/gl/glspec46.core.pdf.

[19] Skia Graphics Library, Skia graphics library, 2025, https://skia.org. (Accessed: 2025-05-03).

[20] M. Kilgard, NVprSDK: NV_path_rendering Examples, 2020, https://github.com/markkilgard/NVprSDK/tree/master/nvpr_examples. (Accessed: 2025-05-03).

[21] Raph Levien and contributors, GPU stroke expansion paper and source code, 2025, https://github.com/linebender/gpu-stroke-expansion-paper. (Accessed: 2025-05-04).

[22] Raph Levien and contributors, Vello: A GPU-based 2D renderer, 2025, https://github.com/linebender/vello. (Accessed: 2025-05-04).

[23] Raph Levien and contributors, Vello_svg: SVG rendering for the Vello GPU 2D renderer, 2025, https://github.com/linebender/vello_svg. (Accessed: 2025-05-04).

[24] E. Süli, D.F. Mayers, An Introduction to Numerical Analysis, Cambridge University Press, 2003.