GPU-friendly Stroke Expansion

Anonymous Author 1
Anonymous Author 2

ABSTRACT

Vector graphics includes both filled and stroked paths as the main primitives. While there are many techniques for rendering filled paths on GPU, stroked paths have proved more elusive. This paper presents a technique for performing stroke expansion, namely the generation of the outline representing the stroke of the given input path. Stroke expansion is a global problem, with challenging constraints on continuity and correctness. Nonetheless, we implement it using a fully parallel algorithm suitable for execution in a GPU compute shader, with minimal preprocessing. The output of our method can be either line or circular arc segments, both of which are well suited to GPU rendering, and the number of segments is minimal. We introduce several novel techniques, including an encoding of vector graphics primitives suitable for parallel processing, and an Euler spiral based method for computing approximations to parallel curves and evolutes.

CCS CONCEPTS

• **Computing methodologies** → **Rendering**; **Parametric curve and surface models**.

KEYWORDS

Vector Graphics, Stroke, Offset Curve, Path Rendering, GPU

1 INTRODUCTION

Rendering of vector graphics documents requires handling both filled and stroked primitives. There is substantial literature on GPU rendering of filled paths, but many fewer published techniques for strokes. It is a more challenging problem, especially for parallel computation, because path segments cannot be processed independently of each other; the *joins* between adjacent path segments depend on context. In addition, the path topology affects the rendered result, in particular whether subpaths are open or closed. In the former case, the endpoints are rendered with *caps*. Joins and caps can have multiple styles, including different styles within the same document.

Stroke expansion is the process of generating an outline that, when filled, produces the *stroke* of a given input path. Informally, a line of the same width as the stroke is swept along the path, normal to it, and also decorated with joins and caps. Though stroke rendering is referenced in many 2D graphics standards, it has lacked a precise formal definition until recent work, primarily Nehab [2020] and Kilgard [2020].

The Nehab [2020] paper gives a comprehensive survey of techniques for stroke expansion, and an algorithm which, for a number of reasons, is only suitable for implementation on CPU (though adapting it to GPU is listed as promising future work). It classifies techniques into *local,* where each path segment generates closed geometry (which may be triangles or other primitives), and *global,* where the overall result is a closed outline of the stroked path, but the partial result from each segment is in general open. Our technique is considered global in this scheme, yet allows independent processing of each path segment.

A number of factors contribute to an algorithm being "GPU-friendly." In addition to simply being able to process the input segments in parallel, such an algorithm also avoids divergent control flow (avoiding the need for explicit subdivision at special events such as inflection points and cusps) and uses robust numerical techniques not subject to particular problems when evaluated using 32-bit floating point numbers.

We propose that the correctness of stroke outlines be divided into *weak correctness* and *strong correctness.* We define strong correctness as the computation of the outline of a line swept along the segment, maintaining normal orientation, combined with stroke caps and joins. Weak correctness, by contrast, only requires the parallel curves of stroke segments, combined with caps and the outer contours of joins. The two notions are equivalent for sufficiently well-behaved input, in particular when the curvature measured at endpoints of path segments does not exceed the reciprocal of the half linewidth. As described in detail in Nehab [2020], very few existing implementations actually implement the strong version, so document authors have become accustomed to not depending on behavior at endpoints. Standards for graphics formats, such as SVG [4], also provide an "out," enabling the weaker behavior. While our current implementation emphasizes weak correctness, we believe it can be extended to the stronger sense, by implementing evolutes and inner join contours.

A great number of rendering techniques for strokes have been proposed. Some are *local,* in that they break the stroke up into smaller closed pieces, the union of which forms an approximation to the stroke. Included in this category is polar stroking [10], which proposes a GPU-friendly subdivision scheme based on an angle step error. Others are based on stroke expansion, generating an explicit outline, which is then filled to produce the stroke rendering. Such techniques are further divided into *flattening* approaches, where the outline is approximated by a polyline, and *curve-based* approaches, exemplified by the approach in Nehab [2020]. In both cases, the outline contains approximations of parallel curves of the input curve segments, and, as proposed in Nehab [2020], their evolutes when targeting strong correctness. For a more detailed survey of existing stroke rendering techniques, the reader is directed to Section 4 of Nehab [2020].

The subproblem of approximating parallel curves has also spawned extensive literature, none of which is entirely satisfying, especially when it comes to algorithms that can be efficiently evaluated on GPU. An example of a reasonably good algorithm that computes flattened parallel curves is Yzerman [2020], as used in the Blend2D rendering library. For producing curved outlines, the Tiller and Hanson [21] algorithm is commonly implemented and cited, but its performance is quite poor when applied to cubic Béziers. The quadratic Bézier version is adequate, and forms the basis of the

algorithm in Nehab [2020]. A variant is also used in Skia [8]; instead of lowering the cubic Bézier to a quadratic approximation and then computing the parallel curve, Skia approximates the parallel curve directly with a quadratic Bézier, then measures the error to determine whether further subdivision is necessary.

This paper presents a comprehensive solution to stroke expansion, well suited to GPU implementation. It can produce both flattened polylines or an approximation consisting of circular arcs. The best choice depends on the capabilities of the path rendering mechanism following stroke expansion, but as we show, outlines consisting of circular arcs have many fewer segments and are correspondingly faster to produce. The algorithm is based on Euler spiral segments as an intermediate representation, with an iterative algorithm based on a straightforward error metric for conversion from cubic Béziers. We have also devised a compact binary encoding of paths, suitable for fully parallel computation of stroke outlines, while requiring minimal CPU-side processing. Our algorithm has been implemented in GPU compute shaders, integrated in a full rendering engine for vector graphics, and shows a dramatic speedup over CPU methods.

2 FLATTENING AND ARC APPROXIMATION OF CURVES

The core problem in stroke expansion is approximating the desired curve by segments of some other curve, usually a simpler one. These segments must be within an error tolerance of the source curve, and ideally close to a minimal number of them. We consider a number of source to target pairs, most importantly cubic Béziers to Euler spirals, and Euler spiral parallel curves to either lines or arcs.

There are generally three approaches to such curve approximation. The most straightforward but also least efficient is "cut then measure," usually combined with adaptive subdivision. In this technique, a candidate approximate curve is produced, then the error is measured against the source curve, usually by sampling a number of points along both curves and determining a maximum error (or perhaps some other error norm). If the error is within tolerance, the approximation is accepted. Otherwise, the curve is subdivided (usually at $t = 0.5$) and each half is recursively approximated. A substantial fraction of all curve approximation methods in the literature are of this form, including Nehab [2020]. The main disadvantage is the cost of computing the error metric. Another risk is underestimating the error due to inadequate sampling; this is a particular problem when the source curve contains a cusp.

The next approach is similar, but uses an *error metric* to estimate the error. Ideally such a metric is a closed-form computation rather than requiring iteration. A good error metric is conservative, yet tight, in that it never underestimates the error (which would allow results exceeding the error bound to slip through), and does not significantly overestimate the error, which would result in more subdivision than optimal.

By far the most efficient approach is an *invertible* error metric. In this approach, the error metric has an analytic inverse, or at least a good numerical approximation. Because the metric is invertible, it can predict the number of subdivisions needed, as well as the parameter value for each subdivision. If the error metric is accurate, then approximation is near-optimal. One example of an invertible error metric is angle step, used in polar stroking (Kilgard [2020]); the number of subdivisions is the total angle subsumed by the curve divided by the angle step size, and the parameter value for each subdivision is the result of solving for a tangent direction. Another widely used invertible error metric is Wang's formula (Goldman [2003], Section 5.6.3), which gives a bound on the flattening error based on the second derivative of the curve. This metric is conservative but works well in practice; among other applications, it is used in Skia for path flattening. One limitation of Wang's method is that it only applies to the flattening of filled outlines; when applied naively to generation of parallel curves, it can undershoot substantially, especially near cusps.

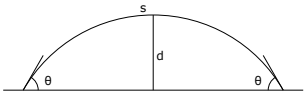

Figure 1: A circular arc segment with notations for angles (θ), arc length (s), and distance to chord (d)

2.1 Error metrics for flattening

The distance between a circular arc segment of length s and its chord, with angle between arc and chord of θ (see Figure 1), is exactly $(1-\cos\theta)\frac{s}{2\theta}$. The curvature is $\kappa = \frac{2\theta}{s}$ (equivalently, $\theta = \frac{\kappa s}{2}$), and this remains constant even as the arc is subdivided. Rewriting, $d = (1 - \cos\frac{\kappa s}{2})\frac{1}{\kappa}$. From this, we can derive a precise, invertible error metric. Subdividing the arc in to n segments, the distance error for each segment is $\frac{1}{\kappa}(1 - \cos\frac{\kappa s}{2n})$. Solving for n, we get:

$$n = \frac{s\kappa}{2\cos^{-1}(1 - d\kappa)}$$

To flatten a finite arc, round up n to the nearest integer. This will cause the error to decrease, so will still be within the error bounds.

Note that the number of subdivisions is proportional to the arc length. Another way of stating this relationship is that the *subdivision density*, the number of subdivisions per unit of arc length, is constant.

The error metric for flattening an arc is exact. It always yields the minimum number of subdivisions needed to flatten the curve, and the flattening error is the least possible given that number of subdivisions. For general curves, an exact error bound is not feasible, and we resort to an approximation. Again the circular arc provides a good example. Applying the small angle approximation $\cos\theta \approx 1 - \theta^2/2$, the approximate distance error is $d = \frac{\kappa s^2}{8n^2}$, and solving for n we get $n = s\sqrt{\frac{\kappa}{8d}}$. Note that this estimate is *conservative,* in that it will always request more subdivision and thus produce a lower error than the exact metric.

We are of course concerned with the flattening of more general curves (ultimately the parallel curve of a cubic Bézier), not simply circular arcs. It is tempting to sample the curvature and plug it into the formula above, but this can both dramatically undershoot the error (predicting that no subdivision is needed at an inflection point) and overshoot it (requiring infinite subdivision at a cusp; and cusps). Apparently, some kind of average curvature is needed.

We propose the following error metric to estimate the maximum distance between an arbitrary curve of length $\hat{s}$ and its chord.

$$d \approx \frac{1}{8}\left(\int_0^{\hat{s}} \sqrt{|\kappa(s)|}ds\right)^2$$

This formula is the same as the approximate error metric for circular arcs, except that instead of a constant curvature value, a norm-like average with an exponent of 1/2 is used (it is not considered a true norm because the triangle inequality does not hold). We chose this particular formulation because it tends to produce invertible error metrics, and because it works well in practice.

This formula also has a meaningful interpretation: the quantity under the integral sign is the subdivision density, and represents the number of subdivisions per unit length of an optimal flattening as the error tolerance approaches zero. In particular, the number of subdivisions is:

$$n = \left(\int_0^{\hat{s}} \sqrt{|\kappa(s)|}ds\right)\sqrt{\frac{1}{8d}}$$

In addition, if the function represented by the integral is invertible, then the corresponding error metric is invertible. Evaluate the function to determine the number of subdivision points, then evenly divide the result, using the inverse of the function to map these values back into parameter values for the source curve being approximated.

2.2 Error metrics for flattening Euler spirals

We choose Euler spiral segments for our intermediate curve representation precisely because their simple formulation in terms of curvature (Cesàro equation) results in similarly simple subdivision density integrals.

An Euler spiral segment is defined by $\kappa(s) = as + b$, or alternatively $\kappa(s) = a(s - s_0)$, where $s_0 = -b/a$ is the location of the inflection point. Applying the above error metric, the subdivision density is simply $\sqrt{|a(s - s_0)|}$. The integral is $\frac{2}{3}\sqrt{a}(s - s_0)^{1.5}$, which is readily invertible.

An immediate consequence is that flattening an Euler spiral by choosing subdivision points $s_i = a \cdot i^{\frac{2}{3}}$ produces a near-optimal flattening, as visualized in Figure 2.

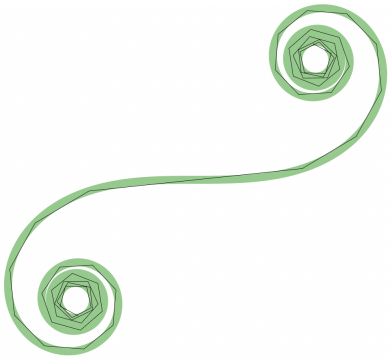

Figure 2: Flattening of an Euler spiral using points spaced by power-law

3 EULER SPIRALS AND THEIR PARALLEL CURVES

It is common to approximate cubic Béziers to some intermediate curve format more conducive to offsetting and flattening. A number of published solutions (Yzerman [2020], Nehab [2020]) use quadratic Béziers, as it is well suited for computation of parallel curves. Even so, this curve has some disadvantages. In particular, it cannot model an inflection point, so the source curve must be subdivided at inflection points.

Like these other approaches, we also use an intermediate curve, but our choice is an Euler spiral. In some ways it is similar to quadratic Béziers – it also has $O(n^4)$ scaling and is best computed using geometric Hermite interpolation – but differs in others. It has no difficulty modeling an inflection point. Further, its parallel curve has a particularly simple mathematical definition and clean behavior regarding cusps.

An Euler spiral segment is defined as having curvature linear in arc length.

The parallel curve of the Euler spiral (also known as "clothoid") was characterized by Wieleitner [1907] well over a hundred years ago [23], and has a straightforward closed-form Cesàro representation, curvature as a function of arc length.

4 FLATTENED PARALLEL CURVES

The geometry of a stroke outline consists of joins, caps, and the two parallel curves on either side of the input path segments, offset by the half linewidth. The joins and caps are not particularly difficult to calculate, but parallel curves of cubic Béziers are notoriously tricky. Analytically, it is a tenth order algebraic curve, which is not particularly feasible to compute directly.

Conceptually, generating a flattened stroke outline consists of computing the parallel curve of the input curve segment followed by *flattening*, the generation of a polyline that approximates the parallel curve with sufficient accuracy (which can be measured as Fréchet distance). However, these two stages can be fused for additional performance, obviating the need to store a representation of the intermediate curve.

Using a subpixel Fréchet distance bound guarantees that the rendered image does not deviate visibly from the exact rendering. Another choice would be uniform steps in tangent angle, as chosen by polar stroking [10]. However, at small curvature, the stroked path can be off by several pixels, and at large curvature there may be considerably more subdivision than needed for faithful rendering.

The limitation of the angle step error metric is shown in Figure 3. The top row shows the use of a distance-based error metric, as is used in our approach, which is visually consistent at varying curvature (in practice, a tolerance value of 0.25 of a pixel is below the threshold of perceptibility). The bottom row shows a consistent angle step, as implemented in polar stroking, but has excessive distance error at low curvature, and excessive subdivision at high curvature. It should be noted, to avoid the undershoot at low curvature, both the Skia [8] and Rive [9] renderers use a hybrid of the Wang and polar stroking error metrics.

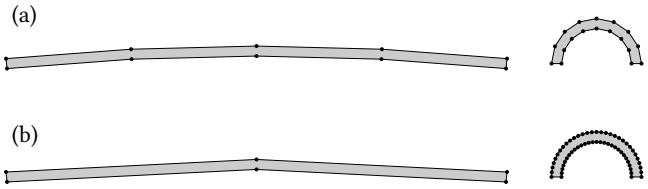

Figure 3: Comparison of error metrics. Top row (a) shows a distance based error metric. Bottom row (b) shows the angle step error metric.

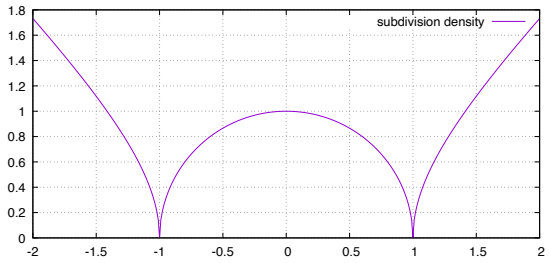

Figure 4: Subdivision density for the parallel curve of an Euler spiral

4.1 Optimal flattening

The optimal flattening of a curve contains the minimum number of subdivisions required to satisfy a maximum error constraint, in which the error is the minimum possible with that number of subdivisions.

For a convex curve, the optimal flattening is tractable to compute, though not necessarily very fast. When a curve is monotonic, error is also monotonic with respect to both subdivision points. Thus, any standard root finding technique can find a subdivision with a given error (bisection is the best known, but the ITP method Oliveira and Takahashi [2020] is better). A similar outer loop with a root finding technique can find the error which minimizes the global error for the entire flattening, which, due to monotonicity, is that for which the error of the final segment equals that of the previous segments. Subdividing at the inflection point adds at most one additional point; it is possible to optimize further, but we do not consider that (though the near-optimal flattening we will later present is not subject to this constraint).

4.2 The subdivision density integral

The subdivision density for the parallel curve of an Euler spiral, normalized so that its inflection point is at -1 and the cusp of the parallel curve is at 1, is simply $1 - \sqrt{|1 - s^2|}$. This function is plotted in Figure 4.

The subdivision density integral for the parallel curve of an Euler spiral is given as follows:

$$f(x) = \int_0^x \sqrt{|u^2 - 1|}du$$

This integral has a closed-form analytic solution:

$$f(x) = \begin{cases} \frac{1}{2}(x\sqrt{|x^2-1|} + \sin^{-1} x) & \text{if } |x| \le 1 \\ \frac{1}{2}(x\sqrt{|x^2-1|} - \cosh^{-1} x + \frac{\pi}{4}) & \text{if } x \ge 1 \end{cases}$$

Values for $x < -1$ follow from the odd symmetry of the function.

4.3 Approximation of the subdivision density integral

The subdivision density integral (Section 4.2) is fairly straightforward to compute in the forward direction, but not invertible using a straightforward closed-form equation. Numerical techniques are possible, but require multiple iterations to achieve sufficient accuracy, so are slower. In this subsection, we present a straightforward and accurate approximation, constructed piecewise from easily invertible functions. If higher flattening quality is desired at the expense of slower computation, this approximation can be used to determine a good initial value for numeric techniques; two iterations of Newton solving are enough to refine this guess to within 32-bit floating point accuracy.

The approximation is given as follows:

$$f_{approx}(x) = \begin{cases} \frac{\sin c_1 x}{c_1} & \text{if } x < 0.8 \\ \frac{\sqrt{8}}{3}(x-1)^{1.5} + \frac{\pi}{4} & \text{if } 0.8 \le x < 1.25 \\ 0.6406x^2 - 0.81x + c_2 & \text{if } 1.25 \le x < 2.1 \\ 0.5x^2 - 0.156x + c_3 & \text{if } x \ge 2.1 \end{cases}$$

$$\begin{aligned} c_1 &= 1.0976991822760038 \\ c_2 &= 0.9148117935952064 \\ c_3 &= 0.16145779359520596 \end{aligned}$$

The primary rationale for the constants is for the approximation to be continuous. The other parameters were determined empirically; further automated optimization is possible but is unlikely to result in dramatic improvement. Further, this approximation is given for positive values. Negative values follow by symmetry, as the function is odd.

The exact integral and the approximation given above are shown in Figure 5. Visually, it is clear that the agreement is close, and in numerical testing the worst case discrepancy between approximate and exact results is approximately 6%.

5 ERROR METRICS FOR APPROXIMATION BY ARCS

The problem of approximating a curve by a sequence of arc segments has extensive literature, but none of the published solutions are quite suitable for our application. The specific problem of approximating an Euler spiral by arcs is considered in Meek and Walton [2004] using a "cut then measure" adaptive subdivision scheme, but their solution has poor quality; it scales as $O(1/n^2)$, while $O(1/n^3)$ is attainable. The result was improved "slightly" by Narayan [2014]. The literature also contains optimal results, namely Maier [2014] and Nuntawisuttiwong and Dejdumrong [2021], but at considerable cost; both approaches claim $O(n^2)$ time complexity. The through-line for all these results is that they are solving a harder problem: adopting the constraint that the generated arc sequence is G^1 continuous. While desirable for many applications,

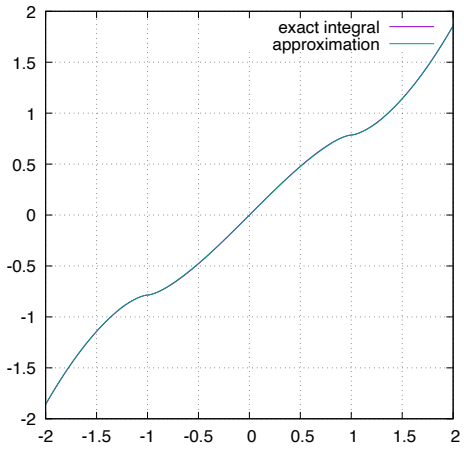

Figure 5: Integral of subdivision density for Euler spiral parallel curve, and its approximation

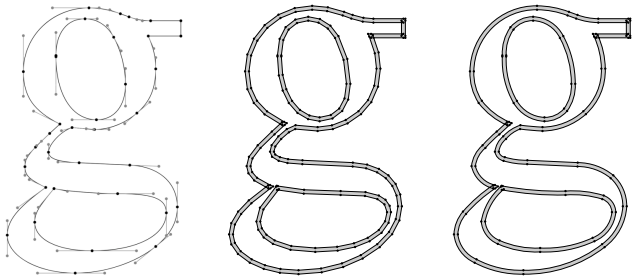

Figure 6: A lowercase 'g' glyph from Nimbus Roman (left, constructed from cubic Bézier segments), its flattening to lines (center), and its approximation by arc segments (right), both with a tolerance of 2.0.

this constraint is not needed for rendering a stroke outline. Even with this constraint relaxed, the angle discontinuities of an arc approximation are tiny compared to flattening to lines.

Our approach is based on a simple error metric, similar in flavor to the one for flattening to line segments. The details of the metric (in particular, tuning of constants) were obtained empirically, though we suspect that more rigorous analytic bounds could be obtained. In practice it works very well indeed; the best way to observe that is an interactive testing tool, which is provided in the supplemental materials.

The proposed error metric is as follows. The estimated distance error for a curve of length $\hat{s}$ is:

$$d \approx \frac{1}{120}\left(\int_0^{\hat{s}} \sqrt[3]{|\kappa'(s)|}ds\right)^3$$

For an Euler spiral segment, $\kappa'(s)$ is constant and thus this error metric becomes nearly trivial. With n subdivisions, the estimated distance is simply $\frac{s^3\kappa'}{120n^3}$. Solving for n, we get $n = s\sqrt[3]{\frac{|\kappa'|}{120d}}$ subdivisions, and those are divided evenly by arc length, as the subdivision density is constant across the curve, just as is the case for flattening arcs to lines.

Remarkably, the approximation of an Euler spiral parallel curve by arc segments is almost as simple as that for Euler spirals to arcs. As in flattening to lines, the parameter for the curve is the arc length of the originating Euler spiral. The subdivision density is then constant, and only a small tweak is needed to the formula for computing the number of subdivisions, taking into account the additional curvature variation from the offset by h (the half line-width). The revised formula is:

$$n = s\sqrt[3]{\frac{|\kappa'|(1 + 0.4|hs\kappa'|)}{120d}}$$

This formula was determined empirically by curve-fitting measured error values from approximating Euler spiral parallel curves to arcs, but was also inspired by applying the general error metric formula to the analytical equations for Euler spiral parallel curve, and dropping higher order terms. A more rigorous derivation, ideally with firm error bounds, remains as future work.

One consequence of this formula is that, since h appears under absolute value, the same arc approximation can be used for both sides of a stroke.

See Figure 6 for a comparison between flattening to a polyline and approximation with arc segments. The arc segment version has many fewer segments at the same tolerance, while preserving very high visual quality.

6 EVOLUTES

In the principled, correct specification for stroking [16], parallel curves are sufficient only for segments in which the curvature does not exceed the reciprocal half-width. When it does, additional segments must be drawn, including evolutes of the original curve. In general, the evolute of a cubic Bézier is a very complex curve, requiring approximation techniques. By contrast, the evolute of an Euler spiral ($\kappa = as$) is another spiral with a simple Cesàro equation, namely $\kappa = -a^{-1}s^{-3}$, an instance of the general result that the evolute of a log-aesthetic curve is another log-aesthetic curve [24].

Flattening this evolute is also straightforward; the subdivision density is proportional to $s^{-0.5}$ where s is the arc length parameter of the underlying Euler spiral (and translated so $s = 0$ is the inflection point). Thus, the integral is $2\sqrt{s}$, and the inverse integral is just squaring. Thus, flattening the evolute of an Euler spiral is simpler than flattening its parallel curve.

The effect of adding evolutes to achieve strong correctness is shown in Figure 7. The additional evolute segments and connecting lines are output twice, to make the winding numbers consistent and produce a watertight outline. All winding numbers are positive, so rendering with the nonzero winding rule yields a correct final render.

7 CONVERSION FROM CUBIC BÉZIERS TO EULER SPIRALS

The Euler spiral segment representation of a curve is useful for computing near-optimal flattened parallel curves, but standard APIs

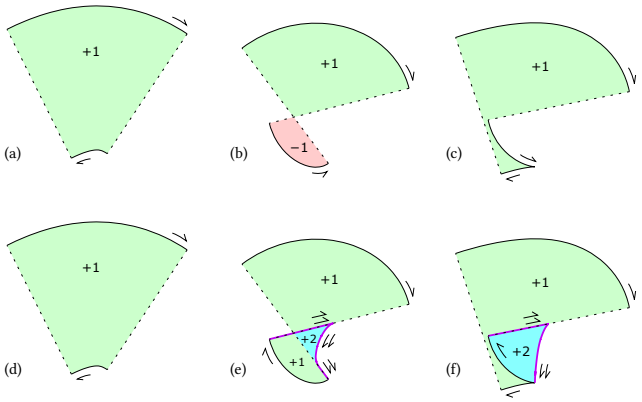

Figure 7: Weakly and strongly correct stroke outlines. The top row shows weakly correct stroke outlines. In (a) the curvature does not exceed the reciprocal half-width, and the stroke is rendered correctly. In (b) the curvature of the bottom outline consistently exceeds the reciprocal half-width, and a section of the outline incorrectly has negative winding number. In (c) there is a cusp. The bottom row shows corresponding strongly correct renders; (d) is the same as (a), while the other two show additional segments for the evolute and connecting lines (in purple). Note that the sections of parallel curve outline with above-threshold curvature are reversed, and that the section enclosed by the evolute has a total winding number of +2.

and document formats overwhelmingly prefer cubic Béziers as the path representation.

Many techniques for stroke expansion described in the literature apply some lowering of cubic Bézier curves to a simpler curve type, more tractable for evaluating parallel curve. Computing parallel curves directly on cubic Bézier curve segments is not very tractable. In particular, the widely cited Tiller-Hanson algorithm [21] performs well for quadratic Béziers but significantly worse for cubics.

A typical pattern for converting from one curve type to another is *adaptive subdivision*. An approximate curve is found in the parameter space of the target curve family. The error of the approximation is measured. If the error exceeds the specified tolerance, the curve is subdivided (typically at $t = 0.5$), otherwise the approximation is accepted. Subdivisions are also indicated at special points; for example, since quadratic Béziers cannot represent inflection points, and geometric Hermite interpolation is numerically unstable if the input curve is not convex, lowering to quadratic Béziers also requires calculation of inflection points and subdividing there. A good example of this pattern is Nehab [2020]. One advantage of Euler spirals over quadratic Béziers is that they can represent inflection points just fine, so it is not necessary to compute those for additional subdivision.

The approach in this paper is another variant of adaptive subdivision, with two twists. First, it's not necessary to actually generate the approximate curve to measure the error. Rather, a straightforward closed-form formula accurately predicts it. The second twist is that, since compute shader languages on GPUs typically don't support recursion, the stack is represented explicitly and the conceptual recursion is represented as iterative control flow. This is an entirely standard technique, but with a clever encoding the entire state of the stack can be represented in two words, each level of the stack requiring a mere single bit.

7.1 Error prediction

A key step in approximating one curve with another is evaluating the error of the approximation. A common approach (used in Nehab [2020] among others) is to generate the approximate curve, then measure the distance, often by sampling at multiple points on the curve. All this is potentially slow, with the additional risk of underestimating the error due to undersampling.

Our approach is different. In short, we perform a straightforward analytical computation to accurately estimate the error. Our approach to the error metric has two major facets. First, we obtain a cubic Bézier which is a very good fit to the Euler spiral, then we estimate the distance between that and the source cubic. Due to the triangle inequality, the sum of these is a conservative estimate of the true Fréchet distance between the cubic and the Euler spiral.

For mathematical convenience, the error estimation is done with the chord normalized to unit distance; the actual error is scaled linearly by the actual chord length.

The cubic Bézier approximating the Euler spiral is one in which the distance of each control point from the endpoint is $d = \frac{2}{3(1+\cos\theta)}$, where θ is the angle of the endpoint tangent relative to the chord. This is a generalization of the standard approximation of an arc. It should be noted that this is not in general the closest possible fit, but it is computationally tractable and has near-uniform parametrization. In general it has quintic scaling; if an Euler spiral segment is divided in half, the error of this cubic fit decreases by a factor of 32. A good estimate of this error is:

$$4.6255 \times 10^{-6}|\theta_0 + \theta_1|^5 + 7.5 \times 10^{-3}|\theta_0 + \theta_1|^2|\theta_0 - \theta_1|$$

The distance between two cubic Bézier segments can be further broken down into two terms; in most cases, the difference in area accurately predicts the Fréchet distance between the curves. Two cubic Béziers with the same area and same tangents tend to be relatively close to each other, but there is some error stemming from the difference in parametrization. Area of a cubic Bézier segment is straightforward to compute using Green's theorem:

$$a = \tfrac{3}{20}(2d_0 \sin\theta_0 + 2d_1 \sin\theta_1 - d_0 d_1 \sin(\theta_0 + \theta_1))$$

The conservative Fréchet distance estimate is 1.55 the absolute difference in area between the source cubic and the Euler spiral approximation. The final term for imbalance is as follows, with $\bar{d}_0$ and $\bar{d}_1$ representing the distance from the endpoints of the Euler approximation and d_0 and d_1 the corresponding distance in the source cubic segment, as in the area calculation above:

$$(0.005|\theta_0 + \theta_1| + 0.07|\theta_0 - \theta_1|)\sqrt{(\bar{d}_0 - d_0)^2 + (\bar{d}_1 - d_1)^2}$$

The total estimated error is the sum of these three terms. We have validated this error metric in randomized testing. For values of θ between 0 and 0.5, and values of d between 0 and 0.6, this estimate is always conservative, and it is also tight: over Bézier curves generated randomly with parameters drawn from a uniform

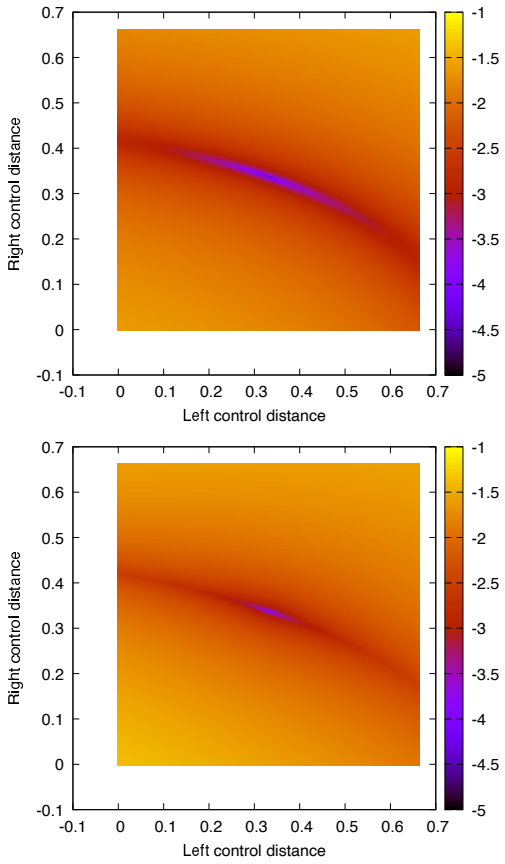

Figure 8: Comparison of measured (top) and approximated (bottom) error for cubic Bézier to Euler spiral conversion. Scale is log (base 10) of the error.

distribution in this range, the mean ratio of estimated to true error is 1.656.

A visualization of combined error metric is shown in Figure 8, comparing measured and approximate error for a slice of the parameter space, fixing the endpoint angles to 0.1 and 0.2, and varying the distance from both endpoints to the control point.

7.2 Geometric Hermite interpolation

Given tangent angles relative to the chord, finding the Euler spiral segment that minimizes total curvature variation is a form of geometric Hermite interpolation. There are a number of published solutions to this problem, involving nontrivial numerical solving techniques: gradient descent [11], bisection [22], or Newton iteration [3]. A more direct approach is to approximate the function as a reasonably low-order polynomial in terms of the endpoint angles. Our approach is to use a 7th order polynomial, which is more precise than the 3rd order polynomial proposed in Reif and Weinmann [2021], at only modest increased cost. The exact coefficients used are presented in Appendix A.

7.3 Unrolled recursion

The entire recursion stack for adaptive subdivision can be represented in two words: dt and scaled_t0. The range is dt * scaled_t0 .. dt * (scaled_t0 + 1). Initial values of 1 and 0 represent the range (0, 1). Pushing the stack, subdividing the range in half, is represented by halving dt and doubling scaled_t0, which leaves the start of the range invariant but its size halved. After accepting an approximation, the next range is determined by incrementing scaled_t0, then repeatedly popping the stack by doubling dt and halving scaled_t0, as long as the latter is odd. This can in fact be achieved without iteration by using the "count trailing zeros" intrinsic.

7.4 Cusp handling

There are two types of cusps that must be handled in the stroke expansion problem. One is when the input cubic Bézier contains a cusp (or near-cusp, which is prone to numerical robustness issues), and one is when the parallel curve contains cusps. This section will describe both in order.

A Bézier curve is expressed in parametric form, and its derivative with respect to the parameter can be zero or nearly so, causing serious numerical problems for many stroke expansion algorithms. In the limit, as the Bézier curve describes a semi-cubical parabola, the curvature can become infinite.

Our approach is to leverage the conversion to Euler spiral segments, which have finite curvature. The geometric Hermite interpolation depends on accurate tangents at the endpoints, which in turn are derived from derivatives. The tangents are not well defined when the derivative is zero, and are not numerically stable when near-zero. Thus, the algorithm has one major mechanism to deal with numerical robustness in these cases: when sampling the cubic Bézier, if the derivative is near zero (as determined by testing against an epsilon threshold), then the derivative is sampled at a slightly perturbed parameter value.

In practice, when rendering a cubic Bézier with a cusp, the region near the cusp is rendered with one Euler spiral segment approximately in shape of the top of a question mark, as shown in Figure 9. Its parallel curve is well defined, and has the correct shape for the outline of the stroke, given of course that the distance is within tolerance (as enforced by the error metric).

As recommended in both Nehab [2020] and Kilgard [2020], the rendering of a cusp with infinite curvature matches that of a near-cusp. The code to detect near-zero derivatives and re-evaluate is a small amount of non-divergent logic, unlike the additional "regularization" pass proposed in Section 3.1 of Nehab [2020] to handle special cases.

7.4.1 Cusps in parallel curves. Even when the input curve is smooth, its parallel curve contains a cusp when the (signed) curvature equals the offset distance (the half-width of the stroke). In published techniques, dealing with these cusps is a nontrivial effort, and involves numerical methods that are not GPU friendly. In particular, detecting locations in the cubic Bézier where the curvature crosses a given quantity is a medium-degree polynomial, and in general requires numerical techniques for root finding. In the approach of Nehab [2020], the root finder not only requires a hybrid Newton/bisection method, which requires iteration, but is also recursive in that it uses the roots of a polynomial of one degree lower as a subroutine. In

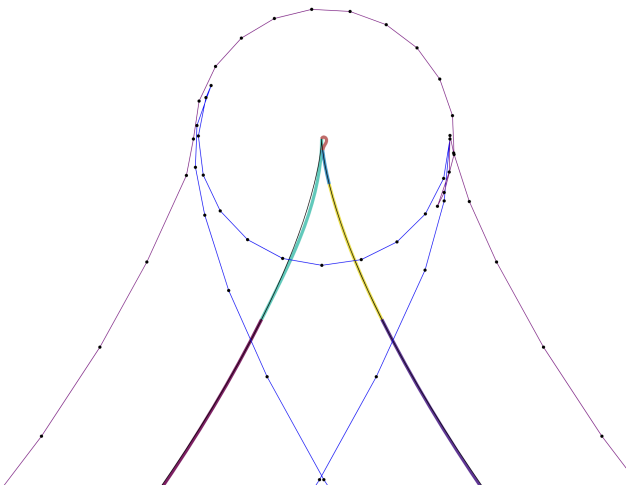

Figure 9: Rendering of a cubic Bézier cusp using Euler spiral segments

general, polynomials up to cubic can be considered GPU-friendly, while finding cusps in cubic Béziers requires a higher degree.

In the simplest case, weak stroke correctness with line segments as the output primitive, no additional work is needed – the flattening algorithm for Euler spiral parallel curves will naturally generate a point within the given error tolerance of the true cusp. However, generation of arc segments and drawing the evolutes needed for strong correctness both require subdivision at the parallel curve cusps.

Fortunately, finding the cusp in the parallel curve of an Euler spiral is a simple linear equation, and there is at most one cusp in any such segment. Euler spirals are thus a good solution for determining cusps as a piecewise linear approximation to curvature.

8 GPU IMPLEMENTATION

We applied the techniques outlined in Section 7 to implement a data-parallel algorithm that can convert stroked 2D Bézier paths into flat geometry suitable for GPU rendering. We focused primarily on the global stroke-to-fill conversion problem in order to generate polygonal outlines for rasterization. Since the Euler spiral approximation can fit any source cubic Bézier and its parallel offset curves, our implementation can be used to render both filled and stroked primitives within a satisfactory error tolerance.

We aimed to satisfy the following the criteria:

(1) The implementation must be able to handle a large number of inputs at real-time rates.
(2) The stroke outlines must meet the strong correctness definition.
(3) The implementation must support the standard SVG stroke end cap (*butt*, *square*, *round*) and join styles (*bevel*, *miter*, *round*).
(4) The CPU must do no work beyond the basic preparation of the input path data for GPU consumption, in order to maximize the exploited parallelism.

One of our artifacts is a GPU compute shader that satisfies these criteria. We coupled this with a simple and efficient data layout for parallel processing. Notably, our implementation is fully data-parallel on input path segments and does not require any computationally expensive curve evaluation on the CPU. We implemented our shader on general-purpose GPU compute primitives supported by all modern graphics APIs, particularly Metal, Vulkan, D3D12, and WebGPU [2]. As such, the ideas presented in this section are portable to various GPU platforms.

The rest of this section describes the design of our GPU pipeline and input encoding scheme.

8.1 Pipeline Design

An SVG path consists of a sequence of instructions (or "verbs"). The *lineto* and *curveto* instructions denote an individual *path segment* consisting of either a straight line or a cubic Bézier. The *moveto* instruction is used to begin a new *subpath* and the *closepath* instruction can be used to create a closed contour (by inserting a line connecting the current endpoint to the start of the subpath). A subpath must form a closed contour if painted as a fill. When painted as a stroke, each subpath must begin and end with a cap according to the desired cap style. In addition, each path segment in a stroked subpath must be connected to adjacent path segments of the same subpath with a join. For any given path, the cap and join styles do not vary across subpaths or individual path segments.

The standard organizational primitive for a compute shader is the *workgroup*. A workgroup can be organized as a 1, 2, or 3-dimensional grid of individual threads that can cooperate using shared memory. Multiple workgroups can be *dispatched* as part of a larger global grid, also of 1, 2, or 3 dimensions.

Our compute shader is arranged as a 1-dimensional grid, parallelized over input path segments. We chose a workgroup size of 256 as it works well over a wide range of GPUs. However, this choice of workgroup size is not particularly important for our algorithm as it does not require any cross-thread communication. Each thread in the global grid is assigned to process a single path segment. We encode the individual path segments contiguously in a GPU storage buffer such that each element has 1:1 correspondence to a global thread ID in the dispatch. This layout easily scales to hundreds of thousands of input path segments, which can be distributed across any number of paths and subpaths. We submit the kernel to the GPU as a single dispatch with as many workgroups as needed to handle the entire input.

Each thread outputs a polyline that fits a subset of the rendered subpath's outer contour, represented by the path segment (see Figure 11). For a filled primitive, a thread only outputs the flattened approximation of the path segment. For a stroke, a thread outputs the polylines approximating multiple curves: a) the two parallel curves on both sides of the source segment, offset by the desired stroke *half-width*, and b) the joins that connect the path segment to the next adjacent segment, and c) evolute patches in regions of high curvature. We also store metadata in our input encoding that marks the position of a path segment within the containing subpath. We use this information to determine whether to output an end cap instead of a join. We encode one additional segment designated as

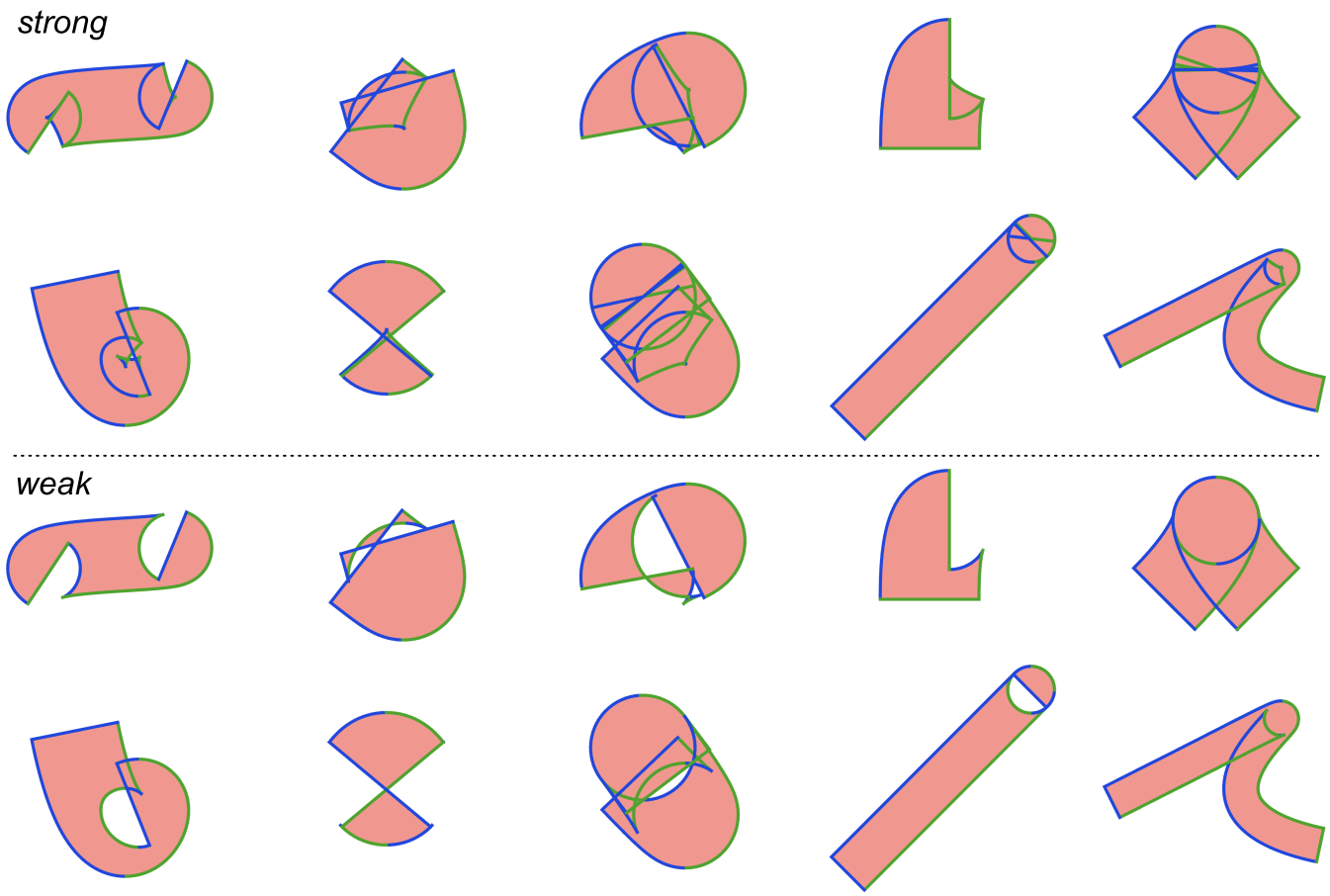

Figure 10: Stroked cubic Béziers exhibiting strong and weak correctness, rendered by our implementation. The contents of the line soup buffer are visualized as green and blue outlines. Segments with a winding direction that points *up* are colored blue while those pointing *down* are colored green. The curves were adapted from the *stills* dataset by Nehab [2020] and the Skia test corpus.

the *stroke cap marker* for every subpath. The thread assigned to the marker segment only outputs a start cap, which completes the stroke outline.

The joins between adjacent segments need to form a continuous outline when combined with the segments. This requires that a thread have access to the tangent vector at the end of its assigned segment *and* the start tangent of the next adjacent segment. We require that the input segments within a subpath are stored in order which simplifies the access to the adjacent segment down to a buffer read at the next array offset.

All input segments first get converted to a cubic Bézier by degree elevation. Our flattening routine (see Algorithm 2) starts by computing the error estimate for geometric Hermite interpolation from the cubic Bézier to an Euler spiral segment. If the error exceeds the desired tolerance threshold, the cubic segment is iteratively subdivided in half and a new error estimate is computed. Once the error satisfies the tolerance, the Euler spiral segment is directly flattened to a polyline by invoking a subroutine that is parameterized by the stroke half-width (see `es_seg_flatten_offset` in Algoritm 2). For a regular fill, this parameter is set to 0. For strokes, the sign of the parameter determines the position of the parallel curve relative to the source segment as well as the winding direction of the output lines.

This procedure is sufficient to generate weakly correct stroke outlines. If the outline contains a cusp – which can be determined by evaluating the curvature variation along the Euler spiral segment – we output an evolute patch (as shown in Figure 7) in order to achieve strong correctness. A selection of renderings produced by our algorithm that demonstrate both strong and weak correctness is shown in Figure 10.

The output of the compute shader is a collection of line segments that gets stored in a GPU storage buffer. Every element in the output contains the viewport coordinates of the two endpoints of a single line segment. Threads reserve storage buffer regions for their output by incrementing an atomic index stored in GPU device memory. A property of the data-parallel structure is that the line segments are unordered. Because the segments are processed in parallel, ordering cannot be guaranteed across path segments. We refer to this output data structure as *line soup*.

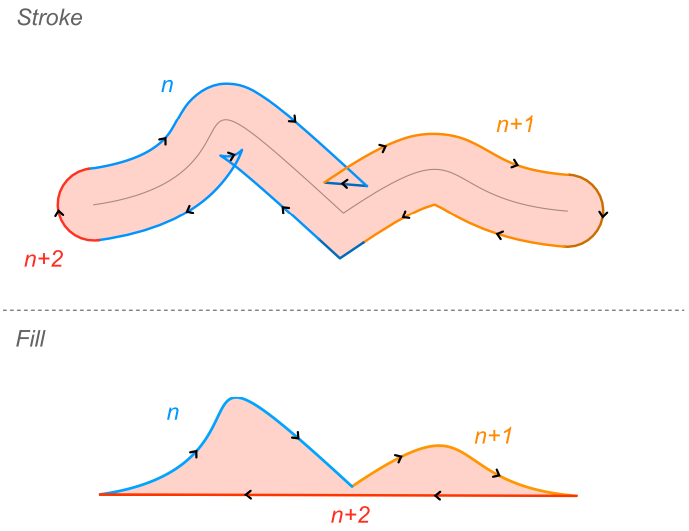

Figure 11: Visualization of contour outlines and the IDs of GPU threads that flatten them. The subpath contains two adjacent cubic Bézier curves rendered as both a stroke and a fill. In the stroked version, thread n flattens the offset curves of the first curve (blue) and the join segment with the miter style (dark blue). Thread $n+1$ flattens the offset curves of the second curve (orange) and the end cap with the round style (dark orange). Thread $n+2$ handles the *subpath end segment* and flattens the start cap (red). When the same path is rendered as a fill, threads n and $n+1$ each flatten the source curves, while thread $n+2$ outputs the *close* segment that joins the end of the subpath to its starting point. The black arrows represent the winding direction of the output segments.

```
1  tid ← global invocation ID;
2  (style, segment) ← readScene(tid);
3  if style is fill then
       // Output source curve (offset = 0)
4      flatten_cubic(segment, 0);
5  else
       // Handle stroked segment
6      if segment is stroke cap marker then
7          if path is open then
8              output start cap;
9          else
10             output nothing;
11         end
12     else
           // Output parallel curves
13         flatten_cubic(segment, style.offset);
14         flatten_cubic(segment, -style.offset);
15         neighbor ← readScene(tid + 1);
16         if neighbor is not stroke cap marker OR path is closed
             then
17             output join;
18         else
19             output end cap;
20         end
21     end
22 end
```

Algorithm 1: The control flow of the compute shader

An application may want to sort the line soup, depending on the requirements of the chosen rendering algorithm. For instance, our own renderer employs tile-based scanline rasterization in a compute shader, with a pipeline architecture that resembles *cudaraster* [12] and *MPVG* [5]. In this design, the line soup elements are spatially sorted by subsequent compute stages into a grid of 16x16 pixel tiles. The final compute stage of our renderer (called *fine rasterizer*) operates at tile granularity and computes pixel coverage for filled polygons outlined by the line soup (Figure 12 illustrates the overall structure of our renderer, the details of which are beyond the scope of this paper).

We also implemented a variant that outputs circular arc segments instead of lines, in which the output entries store curvature in addition to the endpoints. For both primitive types, our renderer also outputs a 32-bit *path ID* used by downstream pipeline stages to identify the path that a segment belongs to, in order to retrieve the correct paint (solid or gradient color) for rendering.

8.2 Input encoding

The input to our pipeline consists of a sequence of paths, each with an associated 2D affine transformation matrix and a *style* data structure containing information about the stroke style or fill rule. We encode paths, transforms, and styles as parallel input streams. Unlike a conventional structure-of-arrays arrangement, we permit a one-to-many mapping across elements between certain streams: each transform and style entry can apply to one or more path entries with an array index greater than or equal to transform or style index.

Paths get encoded as two parallel streams: a *path tag* stream containing an element for every segment in every subpath, and a *path data* stream of point coordinates. Each segment/verb contributes a variable number of points: 1 for *moveto*/*lineto*, 2 for quadratic Béziers, and 3 for cubic Béziers. This variable length stream encoding allows for a highly compact representation of SVG-type content in memory. A GPU thread assigned to a path segment must be able to reference the corresponding element in each stream in order to obtain the right curve coordinates, transform, and style information. This is done by computing a set of stream offsets for every element in the path tag stream.

We compute the offsets on the GPU in a separate compute shader that runs prior to the flattening stage. A path tag consists of 8 bits that store *stream offset increments*, such that the inclusive prefix sums of all increments for every element in the tag stream yield the stream offsets. We call this structure the *tag monoid* and have implemented the computation as a parallel prefix scan.

Table 1 shows the individual fields of the 8-bit path tag. The offset increments for the transform and style streams can be stored in a single bit that is set to 1 only if a path tag marks the beginning of a new transform and/or style entry. We also encode an additional *path bit* in the final segment of all paths, which allows us to access additional streams (such as fill color or gradient parameters) using a per-path offset in later rendering stages.

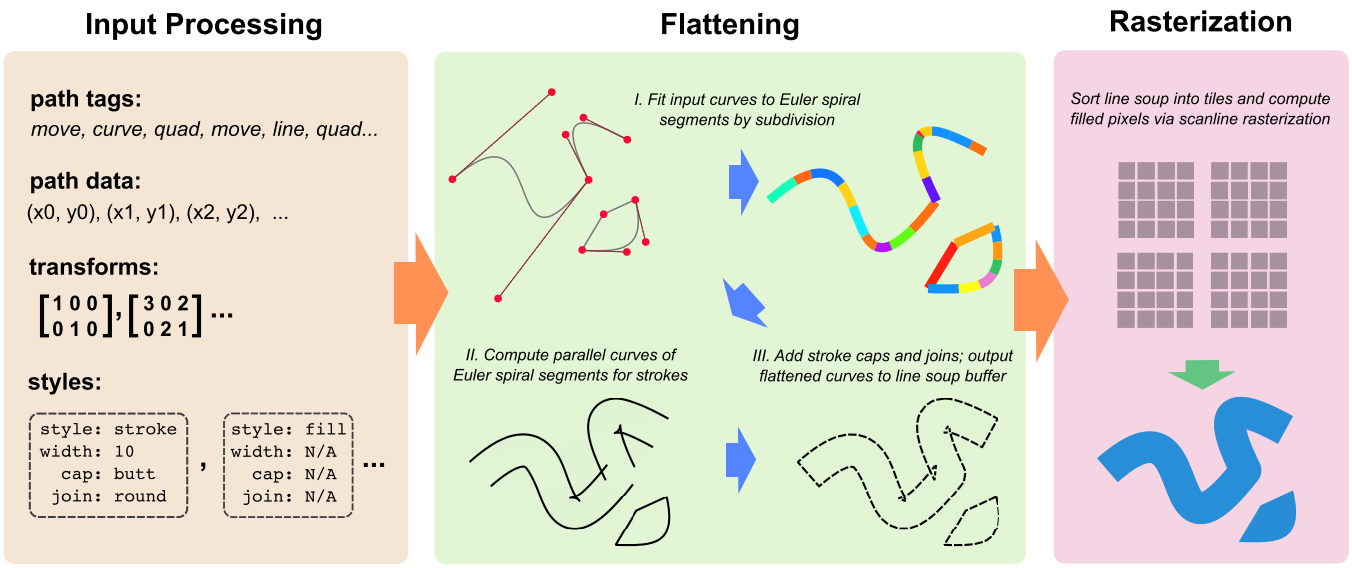

Figure 12: We integrated our curve flattening shader into our own renderer implemented entirely in compute shaders. The flattening stage (described in Section 8) converts stroked paths to filled outlines and flattens them to an unordered soup of line segments. Our renderer spatially sorts the line soup into tiles before computing pixel coverage using a highly parallel scanline rasterizer.

The handling of the subpath bit allows for overlap in coordinates, so that except at subpath boundaries, the first coordinate pair of each segment overlaps with the last coordinate pair of the previous one. During the CPU-side encoding, subpath boundaries are moved from the beginning of a subpath (moveto) to the end. The encoding process inserts an additional line segment into the tag and coordinate streams to close the subpath if the final point does not coincide with the start point. The *subpath end bit* is set to 1 for the final segment of every subpath. For strokes, the subpath end segment represents the *stroke cap marker* defined in Section 8.1. This special segment is assigned a coordinate count of 2, and the corresponding two points in the path data stream encode the tangent vector of the subpath's initial segment, used to correctly draw the start cap or the connecting join in a closed contour.

9 RESULTS

We present two versions of our stroking method: a sequential CPU stroker and the GPU compute shader outlined in Section 8. Both versions can generate stroked outlines with line and arc primitives. We focused our evaluation on the number of output primitives and execution time, using the *timings* dataset provided by Nehab [2020], in addition to our own curve-intensive scenes to stress the GPU implementation. We present our findings in the remainder of this section.

9.1 CPU: Primitive Count and Timing

We measured the timing of our CPU implementation, generating both line and arc primitives, against the Nehab and Skia strokers, which both output a mix of straight lines and quadratic Bézier primitives. The results are shown in Figure 13. The time shown is the total for test files from the *timings* dataset of Nehab [2020], and the tolerance was 0.25 when adjustable as a parameter.

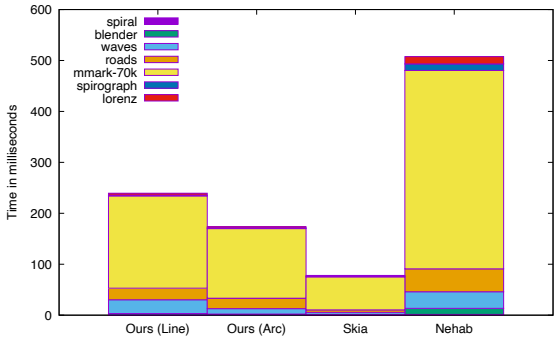

Figure 13: CPU timings for stroke expansion, comparing two versions of our stroker (outputting line and arc primitives) against Skia's curve-to-curve stroker. Measured on an AMD 3970

Similarly to Nehab [2020]'s observations, Skia is the fastest stroke expansion algorithm we measured. The speed is attained with some compromises, in particular an imprecise error estimation that can sometimes generate outlines exceeding the given tolerance. We confirmed that this behavior is still present. The Nehab paper proposes a more careful error metric, but a significant fraction of the total computation time is dedicated to evaluating it.

We also compared the number of primitives generated, for varying tolerance values from 1.0 to 0.001. The number of arcs generated is significantly less than the number of line primitives. The number of arcs and quadratic Bézier primitives is comparable at practical tolerances, but the count of quadratic Béziers becomes more favorable at finer tolerances. The vertical axis shows the sum of output

```
fn flatten_cubic(cubic: Cubic, offset: f32)
    TOL ← Euler spiral fit tolerance;
    // 't0_u' and 'dt' track the recursion stack
    t0_u ← 0u;
    dt ← 1.0;
    last_p ← cubic.p0;
    last_q ← cubic.p1 - cubic.p0;
    loop
        t0 ← f32(t0_u) * dt;
        if dt == 1.0 then
            break;
        end
        t1 ← t0 + dt;
        (next_p, next_q) ← eval_cubic_and_deriv(cubic, t1);
        // Estimate error of geometric
        // Hermite interpolation to Euler spiral
        (th0, th1, chord_len, err) ← fit_cubic_to_es(
            last_p, next_p, last_q, next_q, dt);
        if err <= TOL then
            es ← euler_segment(th0, th1, last_p, next_p);
            // Find location of any cusp in outline.
            // If 0 ≤ t < 1, draw evolute at 't'.
            cusp0 ← es_curvature(es, 0) * offset + chord_len;
            cusp1 ← es_curvature(es, 1) * offset + chord_len;
            t ← cusp0 * cusp1 >= 0 ? 1 : cusp0 / (cusp0 - cusp1);
            if cusp0 >= 0 then
                evolute_finalize();
                es_seg_flatten_offset(es, offset, vec2(0, t));
            end
            if cusp0 < 0 OR t < 1 then
                evolute_start(es, offset);
                range ← cusp0 >= 0 ? vec2(t, 1) : vec2(0, t);
                es_seg_flatten_evolute(es, range);
                es_seg_flatten_offset_reverse(es, offset, range);
            end
            if cusp0 < 0 AND t < 1 then
                evolute_finalize();
                es_seg_flatten_offset(es, offset, vec2(t, 1));
            end
            // Pop the stack
            t0_u ← t0_u + 1;
            shift ← countTrailingZeros(t0_u);
            t0_u ← t0_u ≫ shift;
            dt ← dt * f32(1 ≪ shift);
            last_p ← next_p;
            last_q ← next_q;
        else
            // Error is above the tolerance.
            // Push the stack; Continue subdivision
            t0_u ← t0_u * 2;
            dt ← dt * 0.5;
        end
    end
end
```

Algorithm 2: Outline of the iterative cubic Bézier flattening routine invoked in our compute shader. A working WGSL implementation is provided in the supplemental materials.

Table 1: *Coordinate count* indicates the type of path segment: 1 for lines, 2 for quadratic Béziers, 3 for cubic Béziers. The *subpath end bit* is set on the last segment of a subpath and the *path bit* is additionally set on the last tag of a path. The number of curve control points is given by *coordinate count*, plus 1 if the subpath end bit is set. The number of bytes per coordinate pair is 4 for 16 bit or 8 for 32 bit coordinates.

Path Tag Bits	Monoid Fields
0-1	coordinate count
2	subpath end bit
3	16/32 bit coordinates
4	path bit
5	transform bit
6	style bit

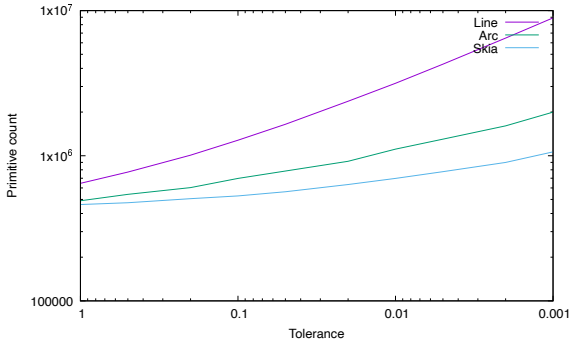

Figure 14: Primitive count for stroke expansion

segment counts for test files from the *timings* dataset, plus mmark-70k (as described in more detail in Section 9.2), and omitting the *waves* example[1].

9.2 GPU: Execution Time

We evaluated the runtime performance of our compute shader on 4 GPU models and multiple form factors: Google Pixel 6 equipped with the Arm *Mali-G78 MP20* mobile GPU, the Apple *M1 Max* integrated laptop GPU, and two discrete desktop GPUs: the mid-2015 NVIDIA GeForce GTX 980Ti and the late-2022 NVIDIA GeForce RTX 4090. We authored our entire GPU pipeline in the *WebGPU Shading Language* (WGSL) and used the *wgpu* [6] framework to interoperate with the native graphics APIs (we tested the M1 Max on macOS via the Metal API; the mobile and desktop GPUs were tested via the Vulkan API on Android and Ubuntu systems).

We instrumented our pipeline to obtain fine-grained pipeline stage execution times using the GPU *timestamp query* feature supported by both Metal and Vulkan. This allowed us to collect measurements for the curve flattening and input processing stages in isolation. We ran the compute pipeline on the *timings* SVG files provided by Nehab [2020]. The largest of these SVGs is *waves* (see rendering *(a)* in Figure 18) which contains 42 stroked paths and a total of 13,308 path segments.

[1]The *waves* example triggers Skia issue https://issues.skia.org/issues/336617138 at finer tolerance.

Table 2: Comparison of input and output segment counts in the more intensive GPU test cases. *Segments* are input path segments. *Lines* and *Arcs* are the output primitives.

Test	Input Segments	Lines	Arcs
waves.svg	13,308	475,855	181,229
mmark-70k	70,000	1,577,705	1,162,117
mmark-120k	120,000	2,709,013	1,994,946
long dash (butt)	503,304	1,672,561	1,672,561
long dash (round)	503,304	2,625,300	1,672,561

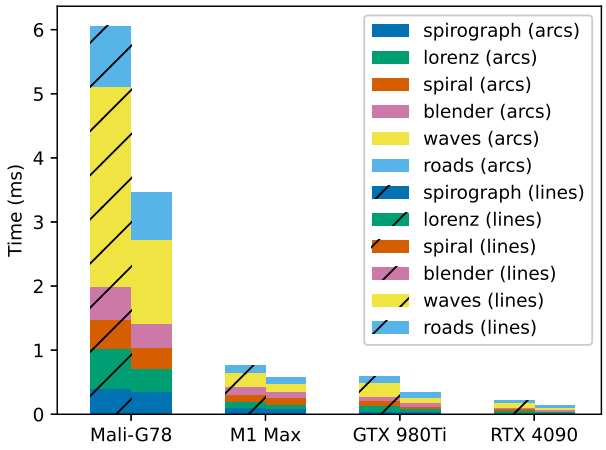

Figure 15: Mean GPU execution times of the compute shader that performs stroke expansion and flattening when run on the Nehab [2020] *timings* data set. The timings are shown in milliseconds. The hatched bars show the timings for the version that outputs line primitives while the solid bars show that for arcs. Mali-G78 (the lowest end GPU we tested) converts *waves* (the largest scene in the set) to lines in 3ms. The compute shader executes at least an order of magnitude faster on the other GPUs.

We authored two additional test scenes in order stress the GPU with a higher workload. The first is a very large stroked path with 500,000 individually styled dashed segments which we adapted from Skia's open-source test corpus. We used two variants of this scene with *butt* and *round* cap styles. For the second scene we adapted the *Canvas Paths* test case from MotionMark [1], which renders a large number of randomly generated stroked line, quadratic, and cubic Bézier paths. We made two versions with different sizes: mmark-70k and mmark-120k with 70,000 and 120,000 path segments, respectively. Table 2 shows the payload sizes of our largest test scenes.

The test scenes were rendered to a GPU texture with dimensions 2088x1600. The contents of each scene were uniformly scaled up (by encoding a transform) to fit the size of the texture, in order to increase the required number of output primitives. As with the CPU timings, we used an error tolerance of 0.25 in all cases. We recorded the execution times across 3,000 GPU submissions for each scene.

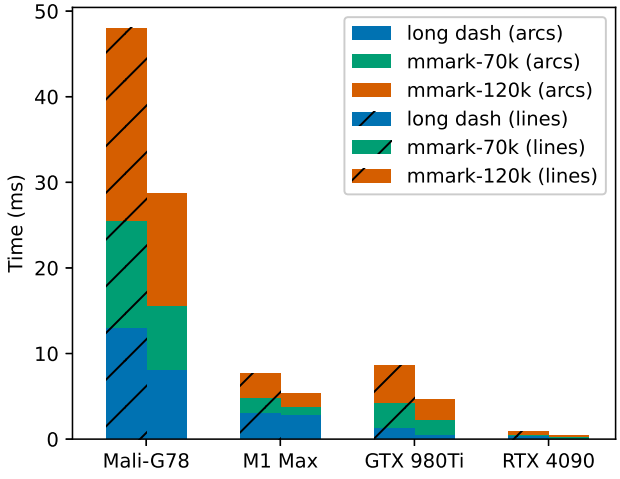

Figure 16: Mean GPU execution times of the compute shader that performs stroke expansion and flattening when run on the stress test scenes. The timings are shown in milliseconds.

Figures 15 and 16 show the execution times for the two data sets. The entire Nehab [2020] timings set adds up to less than 1 ms on all GPUs except the mobile unit. Our algorithm is 14× slower on the Mali-G78 compared to the M1 Max but it is still capable of processing *waves* in 3 ms. We observed that the performance of the kernel scales well with increasing GPU ability even at very large workloads, as demonstrated by the order-of-magnitude difference in run time between the mobile, laptop, and high-end desktop form factors we tested. We also confirmed that outputting arcs instead of lines can lead to a 2× decrease in execution time (particularly in scenes with high curve content, like *waves* and *mmark*) as predicted by the lower overall primitive count.

9.3 GPU: Input Processing

Figure 17 shows the run time of the tag monoid parallel prefix sums with increasing input size. The prefix sums scale better at large input sizes compared to our stroke expansion kernel. This can be attributed to a lack of expensive computations and high control-flow uniformity in the tag monoid kernel. The CPU encoding times (prior to the GPU prefix sums) for some of the large scenes is shown in Table 3. Dashing on the GPU was out of scope, so the dash style applied to the *long dash* scene had to be processed on the CPU and dashes were sequantially encoded as individual strokes. Including CPU-side dashing, the time to encode *long dash* on the Apple M1 Max was approximately 25 ms on average, while the average time to encode the path after dashing was 4.14 ms.

Our GPU implementation targets graphics APIs that do not support dynamic memory allocation in shader code (in contrast to CUDA, which provides the equivalent of *malloc*). This makes it necessary to allocate the storage buffer in advance with enough memory to store the output, however, the precise memory requirement is unknown prior to running the shader.

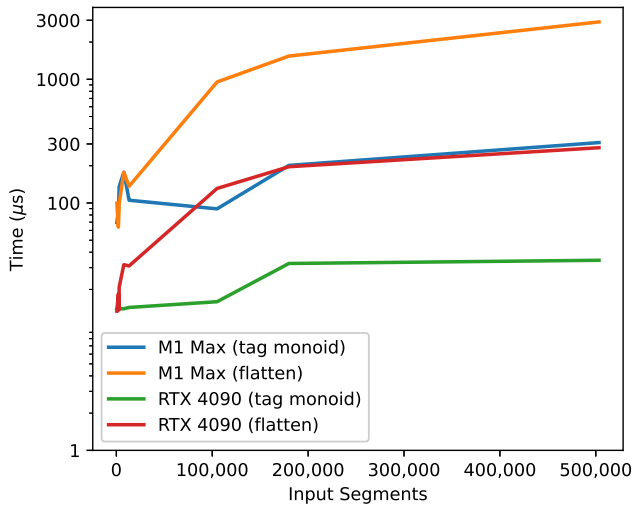

Figure 17: Mean GPU execution time of the tag monoid parallel prefix sums with increasing number of input path segments, measured on Apple M1 Max and NVIDIA RTX 4090. The figure also shows the execution time of the stroke expansion and flattening kernel. The y-axis is shown in logarithmic scale to highlight the similar scaling trend on both GPUs despite the order-of-magnitude difference in performance.

Table 3: Comparison of CPU encoding time with and without size estimation on the largest scenes. All measurements were computed on an Apple M1 Max. The additional cost is negligible when the scene size is modest but tangible with increased complexity.

Test	Encoding	Encoding + Estimation
roads.svg	398.56 μs	618.24 μs
waves.svg	197.53 μs	418.96 μs
mmark-70k	2.69 ms	6.89 ms
mmark-120k	4.28 ms	11.59 ms

We augmented our encoding logic to compute a conservative estimate of the required output buffer size based on an upper bound on the number of line primitives per input path segment. We found that Wang's formula [7] works well in practice, as it is relatively inexpensive to evaluate, though it results in a higher memory estimate than required (see Table 4). For parallel curves, Wang's formula does not guarantee an accurate upper bound on the required subdivisions, which we worked around by inflating the estimate for the source segment by a scale factor derived from the linewidth. We observed that enabling estimation increases our CPU pre-processing time by 2-3× on the largest scenes but the impact is overall negligible when the scene size is modest.

10 CONCLUSIONS

Path stroking has received attention in recent years, with Nehab [2020] and Kilgard [2020] both having proposed a complete theory of correct stroking in vector graphics. While there are a number of implementations of path filling on the GPU, the goal of also implementing stroke expansion on the GPU had not yet been realized. Building upon the theory proposed by Nehab [2020], we implemented an approach to stroke expansion that is GPU-friendly, and achieves high performance on commodity hardware.

We present the Euler spiral as an intermediate representation for a fast and precise approximation of both filled and stroked Bézier paths. Our method for lowering to both line and arc primitives avoids recursion and minimizes divergence, potentially serious problems for parallel evaluation on the GPU. We propose a novel error metric to directly estimate approximation error as opposed to the commonly employed cut-then-measure approaches which often consume the lion's share of computational expense.

Our approach includes an efficient encoding scheme that alleviates the need for expensive CPU pre-computations and unlocks fully GPU-driven rendering of the entire vector graphics model.

Table 4: The estimated and actual *line soup* buffer sizes for scenes rendered at 2088x1600 resolution. Our estimation uses Wang's formula combined with a scaling heuristic based on the linewidth. In the worst case, our buffer size estimator overshoots by a factor 4.

Test	Actual Size	Estimated Size
roads.svg	2.76 MB	8.35 MB
waves.svg	10.89 MB	80.69 MB
mmark-70k	36.10 MB	103.06 MB
mmark-120k	60.01 MB	177.33 MB

11 FUTURE WORK

- The GPU implementation sequentializes some work that could be parallel, and is thus not as well load-balanced as it might be. An appealing future direction is to split the pipeline into separate stages, executed by the GPU as nodes in a work graph [19]. This structure is appealing because load balancing is done by hardware.
- The pre-allocation requirement for bump-allocated line soup buffer is a limitation due to today's graphics APIs. A more accurate and optimized buffer size estimation heuristic is considered future work. It may also be interesting to explore whether graphics APIs can be extended to facilitate GPU-driven scheduling of the compute workload under a bounded memory footprint.
- Many of the error metrics were empirically determined. The mathematical theory behind them should be developed more rigorously, and that process will likely uncover opportunities to fine-tune the technique.
- Dashing stroked paths on the GPU was left out of scope for this paper. Given its parameterization by arc length, we believe the Euler spiral representation is highly suitable for dashing.

REFERENCES

[1] 2021. MotionMark 1.2. https://browserbench.org/MotionMark1.2/about.html
[2] World Wide Web Consortium 2024. *WebGPU.* World Wide Web Consortium. https://www.w3.org/TR/webgpu
[3] Dale Connor and Lilia Krivodonova. 2014. Interpolation of two-dimensional curves with Euler spirals. *J. Comput. Appl. Math.* 261 (2014), 320–332. https://doi.org/10.1016/j.cam.2013.11.009

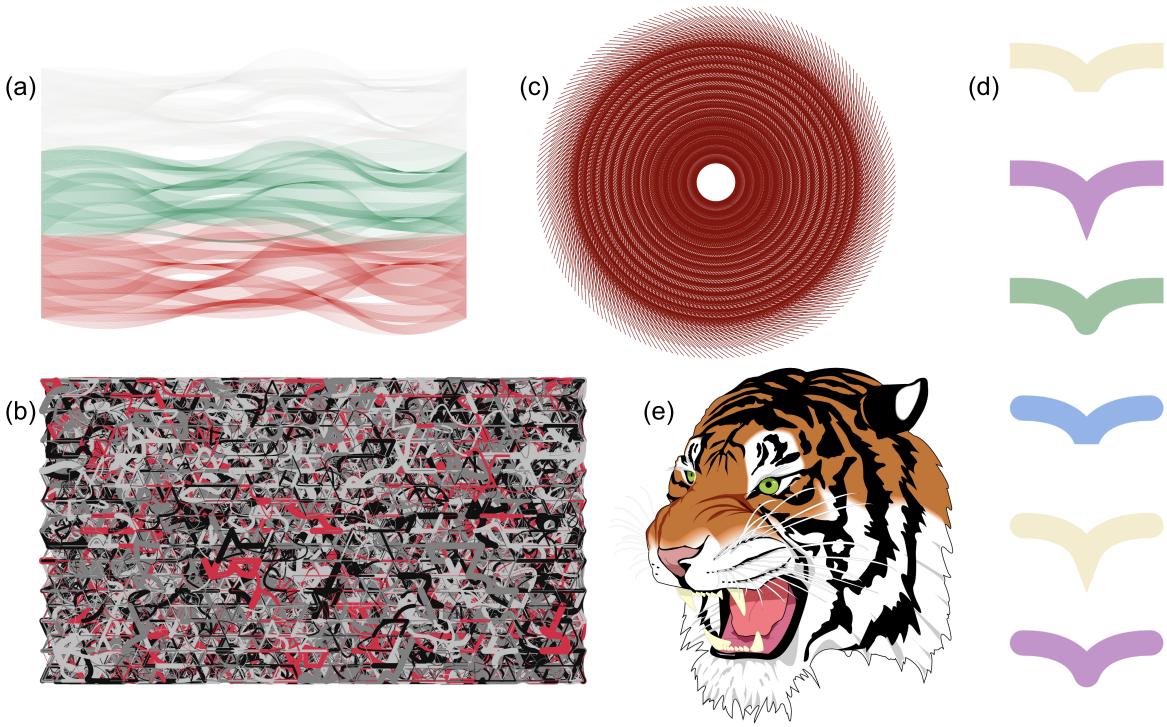

Figure 18: Renderings of some of our test scenes created using our algorithm. All images were rendered on an Apple M1 Max at 2088x1600 resolution using our GPU pipeline. **(a)** *waves.svg* from the Nehab2020 timings data set (3.09 ms); **(b)** The *mmark-120k* scene with 120,000 input segments (19.86 ms); **(c)** The *long path* scene with 500k input segments and round caps (25 ms including CPU-side dashing - the GPU time to render the scene is 9 ms); **(d)**; A simple test scene showcasing various cap and join styles (1.56 ms); **(e)** The famous *Ghostscript Tiger* containing both stroked and filled paths (1.69 ms).

[4] Amelia Bellamy-Royds et al. 2018. *Scalable Vector Graphics (SVG 2) Candidate Recommendation.* World Wide Web Consortium. https://www.w3.org/TR/SVG2/painting.html#StrokeShape

[5] Francisco Ganacim, Rodolfo S. Lima, Luiz Henrique de Figueiredo, and Diego Nehab. 2014. Massively-parallel vector graphics. *ACM Transactions on Graphics* 33, 6 (2014), 1–14. https://doi.org/10.1145/2661229.2661274

[6] The gfx-rs authors. 2024. gfx-rs/wgpu. https://github.com/gfx-rs/wgpu

[7] Ron Goldman. 2003. Chapter 5 - Bezier Approximation and Pascal's Triangle. In *Pyramid Algorithms*, Ron Goldman (Ed.). Morgan Kaufmann, San Francisco, 187–306. https://doi.org/10.1016/B978-155860354-7/50006-4

[8] Google. 2024. Skia. https://skia.org

[9] Rive Inc. 2024. Rive Renderer. https://github.com/rive-app/rive-renderer

[10] Mark J. Kilgard. 2020. Polar Stroking: New Theory and Methods for Stroking Paths. *ACM Trans. Graph.* 39, 4, Article 145 (Aug. 2020), 15 pages. https://doi.org/10.1145/3386569.3392458

[11] Benjamin B. Kimia, Ilana Frankel, and Ana-Maria Popescu. 2003. Euler Spiral for Shape Completion. *International Journal of Computer Vision* 54, 1 (01 Aug 2003), 159–182. https://doi.org/10.1023/A:1023713602895

[12] Samuli Laine and Tero Karras. 2011. High-Performance Software Rasterization on GPUs. *HPG '11: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (2011), 79–88. https://doi.org/10.1145/2018323.2018337

[13] Georg Maier. 2014. Optimal arc spline approximation. *Computer Aided Geometric Design* 31, 5 (2014), 211–226. https://doi.org/10.1016/j.cagd.2014.02.011

[14] D. S. Meek and D. J. Walton. 2004. An arc spline approximation to a clothoid. *J. Comput. Appl. Math.* 170, 1 (2004), 59–77. https://doi.org/10.1016/j.cam.2003.12.038

[15] Smita Narayan. 2014. Approximating Cornu spirals by arc splines. *J. Comput. Appl. Math.* 255, 1 (2014). https://doi.org/10.1016/j.cam.2013.06.038

[16] Diego Nehab. 2020. Converting Stroked Primitives to Filled Primitives. *ACM Trans. Graph.* 39, 4, Article 137 (Aug. 2020), 17 pages. https://doi.org/10.1145/3386569.3392392

[17] Taweechai Nuntawisuttiwong and Natasha Dejdumrong. 2021. An Approximation of Bézier Curves by a Sequence of Circular Arcs. *Information Technology and Control* 50, 2 (2021). https://doi.org/10.5755/j01.itc.50.2.25178

[18] I. F. D. Oliveira and R. H. C. Takahashi. 2020. An Enhancement of the Bisection Method Average Performance Preserving Minmax Optimality. *ACM Trans. Math. Softw.* 47, 1, Article 5 (Dec. 2020), 24 pages. https://doi.org/10.1145/3423597

[19] Amar Patel and Tex Riddell. 2024. *D3D12 Work Graphs.* DirectX Developer Blog. https://devblogs.microsoft.com/directx/d3d12-work-graphs/

[20] Ulrich Reif and Andreas Weinmann. 2021. Clothoid fitting and geometric Hermite subdivision. *Advances in Computational Mathematics* 47, 50 (26 June 2021). https://doi.org/10.1007/s10444-021-09876-5

[21] W. Tiller and E. G. Hanson. 1984. Offsets of two-dimensional profiles. *IEEE Computer Graphics and Applications* 4, 9 (Sept. 1984), 36–46.

[22] D. J. Walton and D. S. Meek. 2009. G1 interpolation with a single Cornu spiral segment. *J. Comput. Appl. Math.* 223, 1 (2009), 86–96. https://doi.org/10.1016/j.cam.2007.12.022

[23] Heinrich Wieleitner. 1907. Die Parallelkurve der Klothoide. *Archiv der Mathematik und Physik* 11 (1907), 373–375.

[24] Norimasa Yoshida and Takafumi Saito. 2012. The Evolutes of Log-Aesthetic Planar Curves and the Drawable Boundaries of the Curve Segments. *Computer-Aided Design and Applications* 9, 5 (2012), 721–731. https://doi.org/10.3722/cadaps.2012.721-731

[25] Fabian Yzerman. 2020. Fast approaches to simplify and offset Bézier curves within specified error limits. https://blend2d.com/research/simplify_and_offset_bezier_curves.pdf

A GEOMETRIC HERMITE INTERPOLATION FOR EULER SPIRAL

This appendix gives the detailed algorithm for Geometric Hermite interpolation, determining Euler spiral segment parameters given the tangent angles at the endpoints relative to the chord.

The Euler spiral is represented by $\kappa(s) = k_0 + k_1 s$, where s ranges from 0 to 1, i.e. the arc length is unit normalized. This segment is scaled, rotated, and translated into place so that its endpoints match the desired locations. Immediately, k_0 can be determined as $\theta_0 + \theta_1$. Therefore, the remaining parameters to compute are k_1 and the ratio of chord length to arc length. The latter *could* be computed from k_0 and k_1 by numerical integration, but it is more efficient to obtain it at the same time as k_1. Setting $k = \theta_0 + \theta_1$ and $\Delta = \theta_1 - \theta_0$, we have:

$$
\begin{aligned}
k_1 = {} & 6\Delta \\
& - \Delta^3/70 \\
& - \Delta^5/10780 \\
& + \Delta^7 \cdot 2.769178184818219 \times 10^{-7} \\
& - k^2\Delta/10 \\
& + k^2\Delta^3/4200 \\
& + k^2\Delta^5 \cdot 1.6959677820260655 \times 10^{-5} \\
& - k^4\Delta/1400 \\
& + k^4\Delta^3 \cdot 6.84915970574303 \times 10^{-5} \\
& - k^6\Delta \cdot 7.936475029053326 \times 10^{-6}
\end{aligned}
$$

Similarly, the formula for the chord to arc length ratio:

$$
\begin{aligned}
c = {} & 1 \\
& - \Delta^2/40 \\
& + \Delta^4 \cdot 0.0003422619048256986 4 \\
& - \Delta^6 \cdot 1.9349474568904524 \times 10^{-6} \\
& - k^2/24 \\
& + k^2\Delta^2 \cdot 0.0024702380951963226 \\
& - k^2\Delta^4 \cdot 3.7297408997537985 \times 10^{-5} \\
& + k^4/1920 \\
& - k^4\Delta^2 \cdot 4.87350869747975 \times 10^{-5} \\
& - k^6 \cdot 3.1001936068463107 \times 10^{-6}
\end{aligned}
$$

Note that the latter quantity is equal to $\mathrm{sinc}(k/2)$ when $\Delta = 0$.

The coefficients were determined by numerical differentiation, using a Newton-style solver as the source of truth. The resulting formulas are extremely accurate over a wide range of inputs.