GPU-friendly Stroke Expansion

Raph Levien
Arman Uguray
Google
San Francisco, CA, USA

ABSTRACT

Vector graphics includes both filled and stroked paths as the main primitives. While there have been many proposed 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 an outline representing the stroke of the given input path, an operation which is considered global, but which we nonetheless implement using a fully parallel algorithm. We introduce several novel techniques, including an encoding of stroked primitives suitable for parallel processing and an Euler spiral based method for computing both flattened parallel curves and approximations to arc segments. Our method produces a near-optimal number of line segments for the filled outline of a stroked cubic Bézier and can be fully evaluated on a GPU with minimal preprocessing.

ACM Reference Format:
Raph Levien and Arman Uguray. 2024. GPU-friendly Stroke Expansion. In . ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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.

TODO: brief description of stroke to fill conversion.

The [**?**] 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 [**?**], 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[**?**], 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.

TODO: the introduction should summarize the paper's contributions and the outline of the rest of the paper. TODO: discussion of prior art.

- Nehab 2020
- Kilgard 2020: Polar stroking (discussion of error metric can move here)
- Yzerman 2020: combined offset + flatten using quadratics
- Blog post of Dragoș?
- Curve-to-curve offsetting? Tiller and Hanson (which is good for quadratics, terrible for cubics), Shape control (pretty good but has robustness issues). Hain (misses error tolerance in cusp case, as described in Yzerman). The "Comparing Offset Curve Approximation Methods" paper is flawed.

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

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

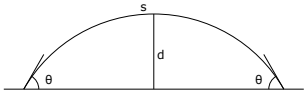

Figure 1: A circular arc segment with notations for angles (θ), arc length (s), and distance to chord (d)

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 [**?**]. 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[**?**]; 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 ([**?**], 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.

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 with 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

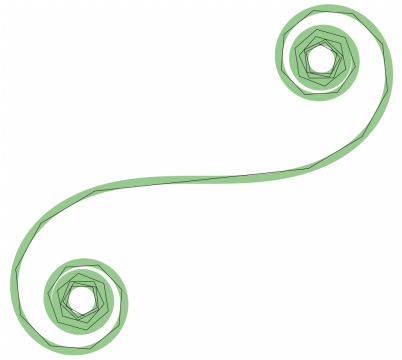

Figure 2: Flattening of an Euler spiral using points spaced by power-law

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 can visually be seen in Figure2.

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 ([**?**], [**?**]) 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 interpoliation – 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 well over a hundred years ago[**?**], and has a straightforward closed-form Cesàro representation, curvature as a function of arc length. TODO: present closed-form solution.

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 segemnts, offset by the half linewidth. The former 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[**?**]. 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. It should be noted, to avoid the undershoot at low curvature, both the Skia[**?**] and Rive[**?**] renderers use a hybrid of the Wang and polar stroking error metrics.

Proposed image: show circular arcs with polar stroking vs near-optimal flattening.

4.1 Optimal flattening

The optimal flattening of a curve, with a maximum error constraint, is as follows. An optimal flattening contains the minimum number of subdivisions required to meet that constraint. Further, 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 [**?**] is better). An 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 Figure3.

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.

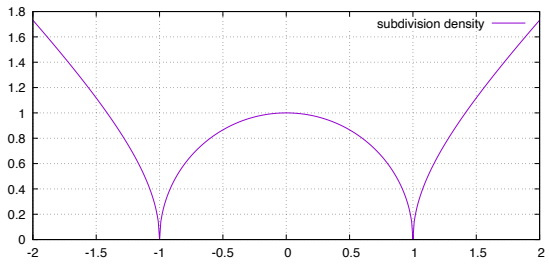

Figure 3: Subdivision density for the parallel curve of an Euler spiral

4.3 Approximation of the subdivision density integral

The subdivision density integral [TODO: probably should use section or equation number reference] 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, based on piecewise 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 4. 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

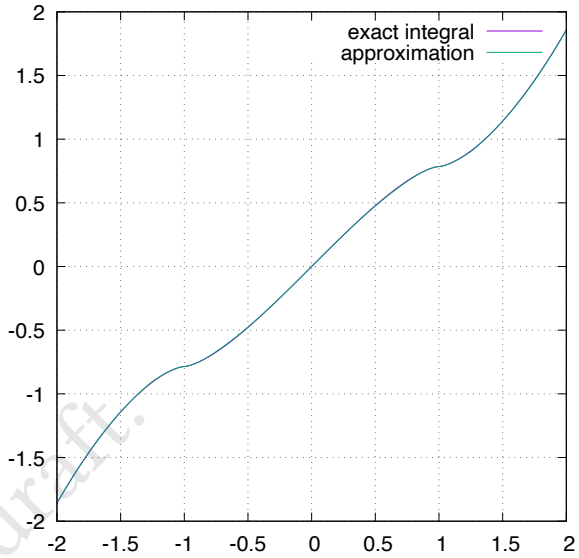

Figure 4: Integral of subdivision density for Euler spiral parallel curve, and its approximation

are quite suitable for our application. The specific problem of approximating an Euler spiral by arcs is considered in [**?**] 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 [**?**]. The literature also contains optimal results, namely [**?**] and [**?**], 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, 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.

TODO: image. This will likely strongly resemble Fig. 8a from Narayan2014.

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 fitting 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.

6 EVOLUTES

In the principled, correct specification for stroking[**?**], 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[**?**].

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.

TODO: figure showing ES, its evolute, and subdivision points for flattening.

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 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[**?**] performs well for quadratic Béziers but significantly worse for cubics.

A typical pattern for converting from one curve type to another is *adaptive subdivision*. (TODO: is there a good general reference for this?) 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[**?**]. 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 [**?**] among others) is to generate the approximate curve, the 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 that with 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:

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

Possible image for this: the heatmap from the cleaner parallel curves blog post.

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:

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

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

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, all involving nontrivial numerical solving techniques: gradient descent[**?**], bisection[**?**], or Newton iteration[**?**]. Our approach is much more direct. We observe that the Euler spiral parameters are a smooth function of the two input parameters, and determine a 2D Taylor's series for this function. Many of the terms are zero because of symmetry. Preserving terms up to 7th order provides excellent accuracy (TODO: quantify more carefully) with modest computational expense.

TODO: A good cite for this is [**?**]. Our solution is similar in flavor, but is more precise at modest cost as it contains more polynomial terms.

TODO: the technique used is a refinement of what was sketched in the Python notebook attached to the "cleaner parallel curves with Euler spirals" blog post. It's basically a 2D Taylor's series. Previously we had the secant method from section 8.2 of my thesis; not sure whether we want to cite that in the submission for anonymity reasons. It's functionally pretty similar to the [**?**] reference.

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 representing 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.

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 weak 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.

Our implementation consists of a single 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[? citation?]. As such, the ideas presented in this section are easily portable.

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. A given subsequence of path segments can be bounded by a *moveto* and/or *closepath* instruction which mark the beginning and end of a *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 correspondance 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.

```
tid ← global invocation ID;
(style, segment) ← readScene(tid);
if style is fill then
    flatten source curve
else
    if you're sad then
        just keep going
    end
end
```

Algorithm 1: The control flow of our compute shader

Each thread outputs a polyline that fits a subset of the rendered subpath's outer contour, represented by the path segment. 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 *halfwidth*, and b) the joins that connect the path segment to the next adjacent segment. 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 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. Notably, this scheme does not cooperation across threads. While we take care to output the *outer* join segment (where the offset curves meet at a reflex angle) we output the inner join as a line segment.

A property of this data-parallel structure is that the output is unordered. The cubic-Bézier-to-Euler-spiral conversion is sequential within a single thread, so it is possible to preserve ordering at the scope of an individual path segment, if desired. However, ordering cannot be guaranteed across path segments. We refer to this output data structure as *line soup*. We believe this structure can be adapted to other primitives, such as circular arc segments or triangles. We incorporated our compute shader into our own massively-parallel GPU rasterizer that spatially sorts the line soup in subsequent binning and tiling stages. As such, maintaining any sort of ordering with respect to the input path data structure was not necessary.

8.2 Input encoding

One contribution of this paper is a highly efficient encoding for paths to be rendered. Goals include conciseness, low cost to encode on the CPU side, and fully parallel processing by the GPU. There are two major aspects to the encoding: a variable size encoding of paths, suitable for both filling and stroking, and an extension to stroking. The input is separated into a *tag stream*, with one byte per path element, and another stream for the coordinate data. The index into coordinate data is computed by a prefix sum from a size value derived from the tag byte.

8.3 Path encoding

The tag byte is a number of bit fields, packed into a byte.

TODO: make this into a diagram.

- 0-1: coordinate count
- 2: subpath end bit
- 3: 16/32 bit coordinates
- 4: path end bit
- 5: transform bit
- 6: line style bit

For most path segments, the coordinate count indicates the type of path segment: 1 for lines, 2 for quadratic Beziers, and 3 for cubic Beziers. The subpath bit is set on the last segment of a subpath, and the path bit is additionally set on the last bit of a path. The latter is not directly interpreted by the path flattening code, but is forwarded to later rendering stages, for example, the path id can be used to select a color or other paint style.

The size is readily inferred from the tag byte: the number of coordinate pairs is the coordinate count plus one if the subpath end bit is set. The nuber of bytes per coordinate pair is 4 for 16 bit coordinates or 8 for 32 bit coordinates.

It is relatively straightforward to convert the standard moveto/lineto/curveto path representation into this encoding. Subpath boundaries are moved from the beginning of a subpath (moveto) to the end. A separate segment must be generated for closepath if the ending point does not coincide with the start point.

The handling of the subpath bit allows for overlap in coordinates, so that, except at subpath boundaries, the first coordinate pair of each path segment overlaps with the last coordinate pair of the previous one. Subpath handling is all computed in the prefix sum; no special handling is needed on the GPU side.

8.4 Algorithm

8.5 Limitations

9 PERFORMANCE

10 DISCUSSION

TODO

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[**?**]. This structure is appealing because load balancing is done by hardware.

- 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.