Solving Distance Queries on Conic Curves

Erik Ruf

October 4, 2011

MSR-TR-2011-110

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Solving Distance Queries on Conic Curves

(Microsoft Research Technical Report MSR-TR-2011-110)

Erik Ruf

October 4, 2011

Abstract

We describe a technique for obtaining the minimum distance vector from a query point to a conic curve segment, and evaluate its performance and precision. Salient features of our approach include a symmetry-based means for guaranteeing sound resuts, a bracket-restricted Newton solver that guarantees convergence, and a query-locality-based optimization for improving performance.

1 Introduction

The family of conic curves forms a useful geometric abstraction that can express multiple varieties of curves, including parabolic, hyperbolic, elliptical and circular arcs. Sequences of such curves, or splines[1], can be used to describe graphical shapes; e.g., a closed spline might describe the inner or outer outline of a glyph, while an open spline, paired with a displacement, could describe a brushed path or a coordinate warp. Compared with their simpler brethren, quadratic Bezier curves, conics add a degree of freedom that enables curvature matching between spline segments. The vX Geometry World library makes extensive use of conic curves.

Making use of such curves in drawing often requires information concerning the distance vector linking an arbitrary point on the plane to the corresponding nearest point on a curve or curve segment. For example, antialiasing evaluates a distance-based attenuation function, while brush effects may require not only a distance value, but also the distance vector itself. In other cases, such as coordinate transformations (warps), the curve parameter value at which the curve intersects the distance vector is also needed.

This document describes our present solution to this distance problem. We begin with our formulation of the distance minimization problem as a zero-finding exercise, and describe how we find a single relevant parameter interval over which iterative techniques can be used. We present two forms of a restricted Newton-based solver, and compare their performance to that of a binary subdivision solver on several metrics. We find that query locality is important, and that we can expect to take advantage of it even in task-parallel and data-parallel implementations.

2 Related Work

2.1 Closed-form solutions

Loop and Blinn [LB05] make use of the rasterization hardware in a GPU to efficiently evaluate the implicit definition of quadratic, conic, and cubic curve segments, thus obtaining an inclusion predicate, but not a distance function. For the specific case of quadratic curves, Nehab and Hoppe [NH08] present a closed-form approximation to the point-curve distance, which has been found to be accurate for points sufficiently near the curve.

[1] Technically, the elements of splines are curve segments. We use the term "curve" for both curves and curve segments when doing so does not introduce ambiguity.

2.2 Iterative solutions

Wang et al. [WKA02] descrives an iterative distance solver for cubic splines. Targeted toward a hardware implementation, it performs a fixed number of rounds of quadratic minimization followed by a fixed number of iterations of a Newton-Raphson technique.

Qin et al. [QMK08] partition quadratic and cubic curve segments into sub-segments, each of which is analyzed separately via a binary subdivision-based search. For each sub-segment analyzed, a correct result is returned if the query point is either outside of the sub-segment or within its minimum radius of curvature. Otherwise, a perpendicular, but not necessarily minimal, intersection point is reported. In the worst case, all sub-segments may have to be analysed (and their distances compared) to find the minimum distance. In Qin's vector graphic rendering example, though, all of the curve sub-segments comprising a graphical object are combined in a statically-constructed grid-based acceleration structure. Thus, for any point query, it is likely that not all of a segment's sub-segments need be processed. For dynamically generated curves, where grid formation is impractical, this optimization would not apply.

3 Algorithms

3.1 Basics

A conic curve can be specified using four control points $p_0 ... p_3$, where the first three points form a Bezier control triangle and p_3 is a point on the curve, restricted to lying within the control triangle.[2] These control points can describe parabolic, hyperbolic, or elliptic curves, which have as degenerate forms points, lines, and circles.[3] We can construct an implicit equation $I(p) = 0$ for the infinite curve, as well as a parameterized formula $F(t)$ such that $F(0) = p_0$, $F(1) = p_2$, and $\forall t, I(F(t)) = 0$. When evaluated on $0 \leq t \leq 1$, F yields points on the portion of the curve lying within the control triangle. When evaluated on the entire real line, F yields the entire implicit curve I for the parabolic and elliptical cases, but in the hyperbolic case, it yields only the arm of the hyperbola that passes through the control triangle. When we speak of a conic curve C, we are referring to this restricted curve.

We are specifically interested in the curve segment $F(t), 0 \leq t \leq 1$, as such segments are employed in the description of glyphs and vector graphic objects. However, much of our analyis relies on properties of the underlying curve.

3.2 Properties of the distance function

Given a nondegenerate conic curve C and a query point q, 1 to 4 points p_i on C will have the property that the vector from q to p_i will be orthogonal to C at p_i, and at least one of the vectors $p_i - q$ will represent the minimum distance from q to the curve. Having multiple minimal distance vectors of the same length is problematic, as it means we can only extract a distance value, but not a curve parameter value, for each query point. Luckily, this situation arises only when q lies exactly on one of the curve's axes of symmetry,[4] in which case we can use the time-honored graphics technique of adjusting q by an infinitesimal offset to move it off the axis. With this case discharged, we can state that every query point q has a single minimum-length distance vector intersecting C, and that vector will be perpendicular to C at the point of intersection. Note that other perpendicular vectors connecting q and C may exist, but will not be of minimum length.

A useful property of symmetric convex curves is that the minimal distance vector from a query point q to a curve C does not cross any of C's axes of symmetry. Furthermore, any other vectors perpendicular to

[2] An alternative formulation, consisting of three control points $p_0 ... p_2$ and an additional scalar $w \geq 0$ will also be used later in the paper.

[3] We will not address the degenerate cases in this document, as finding point-curve distance vectors for such curves can easily be accomplished via analytical means.

[4] All conic curves have either one or two axes of symmetry, which can be obtained analytically by means described in the Appendix.

C and intersecting q must cross a curve axis.[5] Thus, if we use the axes to divide the plane into regions (i.e., two half-planes for single-axis curves, or four quadrants for two-axis curves), any query point q lying within a particular region s will have exactly one perpendicular curve intersection point p inside s, and p will be the curve point closest to q. Thus, a single minimal distance vector can be obtained without the need to find and compare the lengths of multiple curve normals intersecting q.

Unfortunately, these properties hold for *curves*, but not for the curve *segments* in which we are interested. Two cases arise:

- **There is no perpendicular vector from q to the segment.** In such cases, one of the segment endpoints will be the on-segment point closest to q. Only applications requiring endpoint distance information need compute it; other applications (e.g., coordinate warping and transverse brush effects) will find this data irrelevant and will not need to compute it. Our solver simply returns a failure code, allowing the client to decide what to do.
- **Perpendicular vector(s) from q to the segment exist, but they are not minimal with respect to the curve**; i.e., the closest on-curve point is not included in the segment. This means that analyses based on curve properties (e.g., the plane partitioning mentioned above) are sound, but not complete, solutions to the problem of finding perpendicular distance vectors.[6] This leaves the implementor two choices: either find all of the on-curve distance vectors [7] and compare them, or give up and return no distance vector at all.[8]

3.3 The dot-product formulation

We wish to find the Bezier parameter value yielding the smallest query point–curve distance; e.g., we want to find t such that $D(t) = |F(t) - q|$ is minimized. Translating this minimization problem into a zero-finding problem enables the use of techniques such as Newton-Raphson iteration. By observing that all minimal point-curve distance vectors are curve normals, we can restate this problem as follows. We seek to find values of t for which the vector from q to $F(t)$ is perpendicular to the curve tangent $F'(t)$. These values correspond to the zeros of the function $G_q(t) = (F(t) - q) \cdot F'(t)$, which is a fourth-order rational polynomial in t[9] and has 1 to 4 distinct real roots. Finding these roots and choosing the closest one lying on the segment S (i.e., having a parameter value in [0..1]) will yield the minimum perpendicular distance vector, if one exists.

3.4 The iterative solver

We are faced with the task of finding the roots of the fourth order rational polynomial function $G_q(t)$. Given that no direct solution is available, we must approximate the result using an iterative technique.

3.4.1 Choosing solution intervals

Bracket-based iterative zero finding techniques (e.g., subdivision) require an initial interval for which the the function (in this case, $G_q(t)$) has opposite signs when evaluated at the interval endpoints, and is monotonic between the endpoints. Finding such initial intervals can be difficult.

In the particular case of simple quadratic curves (parabolas), we can find suitable intervals by examining the first derivative of the dot product function, namely $G'_q(t)$, which is a quadratic polynomial in t. Each zero of $G'_q(t)$ (easily found via the quadratic formula) indicates either a minimum, maximum, or inflection point in $G_q(t)$). Breaking the real number line at the zero points and intersecting the resulting intervals with

[5] See the Appendix for a proof.

[6] A *sound* algorithm returns only correct results (in this case, a minimal distance vector) but may fail to return a result at all (in this case, a minimal distance vector exists, but it is not returned because a better one exists w.r.t the curve).

[7] This can be difficult; see Section 3.4.1 below.

[8] This failure case is different from the "no perpendicular vector exists" case above. In that case, the client can obtain a correct answer by examining the segment endpoints. This is not true here.

[9] For the specific case of quadratic curves, the dot product function is a cubic polynomial in t.

Given

- a query point q,
- a sorted, possibly empty vector of parameter values $0 < t_i < 1$ at which axes intersect the curve segment, and
- a vector of curve normal vectors v_i corresponding to the t_i

we can determine the appropriate parameter interval on which to solve $G_Q(t) = 0$ as follows:

```
findInterval(q, t, v)
   tMin := 0
   for (i := 1 to # axes)
      if q lies to the left of v[i] then
         return (tMin, t[i])
      else
         tMin = t[i]
   return (tMin, 1)
```

Figure 1: Using saved axis information to determine the initial bracketing interval for solving a query.

the segment interval $[0, 1]$ yields 0–3 intervals to examine. For each interval $[t_{min}, t_{max}]$ such that $G_q(t_{min})$ and $G_q(t_{max}))$ have differing signs, a bracketing search will find a zero. Comparing the resulting distance values for each interval will yield the minimum distance vector.

This works well, in that it finds a correct solution for any query point q in the plane for which a perpendicular distance vector to the curve segment exists. However, it also has disadvantages. First, the derivative $G'_q(t)$ is dependent on q, meaning that we have to repeat the entire interval construction procedure for each query point. Even more significantly, in the general conic case, the derivative $G'_q(t)$ is a rational polynomial whose numerator is of order 5 in t, rendering this interval construction scheme impractical for non-parabolic curves.

Qin [QMK08] attempts to solve this problem by subdividing the curve segment into subsegments for which both the x and y components of the curve tangent are monotonic. As a result, a bisection-based search will find a single curve normal passing through q (and thus yielding a nearest curve point), unless q is interior to the curve and lies beyond the sub-segment's minimum radius of curvature. In this case, more than one perpendicular distance vector may exist, and the algorithm may return a non-minimal one, rendering the algorithm unsound. Qin finds this causes little difficulty in pracice, as the errors occur only for q sufficiently far from the curve that the distance error does not affect the computed pixel value.[10] A secondary form of error arises when an incorrectly large distance value causes the wrong sub-segment to be chosen, which could cause even larger errors, particularly in the parameter value. Two additional issues are not addressed in Qin's paper.

- Because the sub-segment partitioning is based on curve monotonicity with respect to the frame coordinate axes rather than properties of the curve itself (i.e., curve axes), rotating the curve by manipulating its Beizer control points will alter the partition boundaries, potentially resulting in error behavior that varies as the angle of rotation is altered.
- The client has no simple means to determine whether or not his query lies sufficiently close to the curve segment, and thus cannot know when the returned distance vector might be incorrect.

[10]For example, when performing antialiasing, if the minimal distance value yields an attenuation factor of 1, a non-minimal value will also yield a value of 1, so the distance error has no effect. Similarly, erroneous distance values will not effect embossing provided that q lies outside the embossing width.

Our approach is both more conservative and potentially more efficient. We rely on the symmetry properties of conic curves, as described above in Section 3.2. If, for a query point q, we restrict the search space of nearest points (e.g., the range of curve parameter values) to the axis-based half-plane or quadrant where q resides, we can be assured that

- either 0 or 1 perpendicular distance vectors (to the curve segment) will be found, and
- relative to the curve, the computed distance vectors will not change when the query point and curve segment's control points experience the same rotation transformation, and
- any resulting distance vector will be correct (i.e., no shorter distance vector exists), modulo the precision of the bracketing algorithm used.

When a curve segment is created, we find its axis or axes and save a table mapping axis lines to curve parameter ranges. Given a query point, a maximum of two line-side tests yield a parameter range defining the relevant portion (call it the sub-segment) of the curve segment. Figure 1 gives psuedocode for this operation.

Since we know that the dot product function will have at most one root in this range, we can test for the presence of a zero crossing by checking to see that the dot products obtained by evaluating the dot product at both endpoints of the sub-segment have non-equal signs.

What we have done, essentially, is to consider only the closest point on the *curve*, rather than the closest point on the *segment*. This, unfortunately, keeps us from reporting perfectly good distance vectors to on-segment points having a better on-curve (but not on-segment) distance vector. In such cases, [QMK08] would report a distance vector, though not necessarily a minimal one.[11]

3.4.2 Solving an interval

Given a curve parameter interval produced by our axis-based analysis, we know that the dot product function is monotonic over the interval and that it has a single root in the interval, ensuring that a bracket-based iterative solver will be able to find the root. In hope of obtaining a solution more quickly, we would instead like to make use of Newton-Raphson iteration, which usually converges quadratically but may fail to converge entirely. We have found that, for our objective function, non-convergence is a frequent problem in practice; the derivative $G'_p(t)$ can be quite small, resulting in intermediate estimates that fall far outside the original interval.

We address this problem by combining Newton-Raphson with bracketing. On each iteration, a Newton estimate is computed; if it lies within the current bracket interval, we use it; otherwise we revert to an in-interval estimate.[12] This approach will typically perform fewer iterations than the underlying bracket-based analysis would have, but does have to expend extra effort (in the form of a derivative computation and a division) per iteration. An implementation of a similar strategy is given in [PTVF92].

We must also choose a termination criterion. Our algorithm does not minimize the distance function directly; rather, it minimizes the absolute value of a proxy for the distance, namely the dot product of the curve tangent and the point-curve vector at the curve/vector intersection. As changes in this proxy value do not correlate in any simple manner to either the location of the estimated curve point or the query point's distance to it, mapping a desired maximum distance imprecision to a proxy value range is not possible. Similarly, the effect of a change in the estimated parameter value may have widely differing effects on the location of the estimated curve point,[13] and thus the distance value.

Our present approach is to compute the distances between subsequent estimated curve points $d_c = ||F(t_n) - F(t_{n+1})||$ and cease iteration when this distance becomes sufficiently small. The idea here is that

[11] This occurs when the query point and curve segment lie on opposite sides of the curve axis, admitting multiple perpendicular distance vectors, any of which may be found by the subdivision-based search in [QMK08].

[12] Any bracketing strategy can be used to limit the estimate in this manner; we have tried binary subdivision and Regula Falsi and experienced similar results.

[13] In other words, the arc distance along the curve is not a simple function of the curve parameter value.

d_c will always be greater than or equal to the change in the query-curve distance $d_q = ||F(t_n) - q|| - ||F(t_{n+1}) - q||$, so the former can be used to bound the latter. This works, but turns out to be overly conservative for the vast majority of query points.

Near the curve, d_q can be as large as d_c, but as the query point is moved away from the curve, the ratio $\frac{d_q}{d_c}$ becomes much smaller, resulting in excess precision (and excess iteration in achieving it). For example, on our curve segment corpus, we have found that a restriction on d_c such that the true and approximate distances differ by δ near the curve yields an average point-curve distance error of $\frac{\delta}{100}$ (and sometimes even $\frac{\delta}{1000}$) over all query points within a tight rectangular bounding box of the curve segment. We could consider scaling the constraint on d_c depending on the point-curve distance, but would be a recursive invocation of the original problem—and since our intermediate distance estimates are always conservative, the constraint on d_c would be over-damped, resulting in premature termination.

We also note that the degree of precision required will vary across multiple rendered instances of the same curve. For example, a distance value used for antialiasing must become increasingly precise as the number of physical pixels spanned by the curve segment increases.

3.5 Acceleration

The point-curve distance problem has a locality property, in that nearby query points are likely to have nearby closest on-curve points. This suggests that a client (e.g., a rendering algorithm) traversing the query space in a coherent manner (raster lines or rectangular tiles) can benefit by saving the parameter value from one query, and using it as the initial estimate for the next query. Such techniques rely on sequential evaluation of neighbors. Fine-grain parallel clients (e.g., GPU shaders/kernels) that evaluate only a single query per thread would require a separate pass to precompute parameter values at some coarser granularity. We will further examine locality optimizations in Section 4.6.

Another way of improving performance is to weaken the convergence condition of the iterative solver. In particular, if we don't need to know the precise distance to the curve, but only that it is below some threshold, we can cease iteration as soon as the approximate distance falls below the threshold value.

Yet another class of strategies focuses on limiting the space of queries executed. If the client can specify a maximum relevant distance δ, a conservative approximation of the distance function can be used to discard query points for which it can prove that the point-curve distance will exceed δ. Several such methods are described in [Ruf11]. Note that this and the previous case are duals, making use of minimumally- and maximally-relevant distance limits, respectively, and can be usefully combined.

4 Experimental Results

The intent of our experiments was to examine the performance and precision consequences of adding Newton iteration and locality-based acceleration to a basic binary subdivision algorithm. All instances used the dot product minimization form of the problem and the axis-based interval selection technique, both described above. As we were primarily interested in relative performance, we did not implement a GPU-based version of the algorithms, as (modulo SIMD execution issues) performance depends primarily on iteration count. Results concerning precision are of course independent of the target platform as long as the precision of mathematical operations remains the same. Thus, our results are based on a single-threaded C++ implementation, executing on an Intel i7 processor running Windows 7.

We considered comparing our results with an implementation of Qin's technique, but found it difficult to construct a fair comparison, as both the performance and precision of that technique depend on a statically-constructed acceleration structure. Without this structure, Qin's technique must compute and compare multiple distance vectors, and must do so over a larger query space than our region-restricted solver will traverse. Since we are primarily interested in dynamic scenarios involving zooming and/or per-frame changes in curve shape, evaluating our system's performance under a grid scheme seemed a waste of effort. Thus, we will present only results obtained using our technique.

4.1 Curves

Our experiments are based on a corpus of curve segments derived from font descriptions. We extracted 1,000 unique, non-degenerate quadratic curve segments from the Times New Roman TrueType font (beginning with the first ASCII character), and another 1,000 from the MS Mincho TrueType font (beginning with the first Japanese kana character). For each quadratic segment, we generated four hyperbolic segments (by choosing w in $\{2.0, 3.0, 4.0, 5.0\}$) and four elliptical segments (by choosing w in $\{0.8, 0.6, 0.4, 0.2\}$).

For each segment, we saved the bounds of the enclosing glyph's coordinate system, the three control points, described in terms of that coordinate system, and a flag indicating which side of the segment faces outward from the glyph contour to which it belongs. This information was used to drive our experiments. Where needed, precise distance values were obtained by executing our solver configured to return when the iteration-to-iteration difference in the distance value, measured in the curve's coordinate system, becomes less than 10^{-9}.

4.2 Solvers

We first examine the relative performance of Newton iteration and spatially accelerated Newton iteration with respect to a basic subdivision algorithm. We looked at three solution techniques in two scenarios.

The solvers are:

- `subdiv` uses binary subdivision,
- `newt` uses Newton-Raphson iteration, reverting to subdivision when the Newton estimate fails to fall in the current bracket interval, and
- `newt+` is similar to **newton**, but takes its initial estimate from the previously evaluated pixel.[14]

4.3 Performance

4.3.1 Antialiasing example

For outline-based fonts, antialiasing is performed on the outside of each curve segment. A distance-based transparency value is computed for all pixel positions lying within a single pixel width of the segment, on the appropriate side.

We analyze each of the 18,000 curve segments in our corpus over a variety of glyph height values, expressed in screen pixels, ranging from 16 to 2048 pixels in height. Each height value defines a coordinate transformation between pixel and glyph coordinates. We map each segment's bounding rectangle to screen space, pad it by one pixel in each direction, and traverse it, querying the corresponding position in glyph space. The termination condition is set such that a maximum imprecision of 0.2 in the distance value is obtained.

The absolute execution times for each glyph height and solver, and the corresponding relative execution times (with respect to the subdivision solver) are shown in Figure 2. The absolute execution time scales with the problem size (square of the glyph height) as expected. For the two smallest sizes, the Newton techniques are slightly slower than subdivision, presumably due to greater per-iteration costs. At these sizes,the precision criterion can often be met in a small number of iterations, making the faster convergence of the Newton-based techniques irrelevant.

[14] By default, we evaluate the pixels of the bounding region in raster-scan order. See Section 4.6 for an analysis of tiled evaluation.

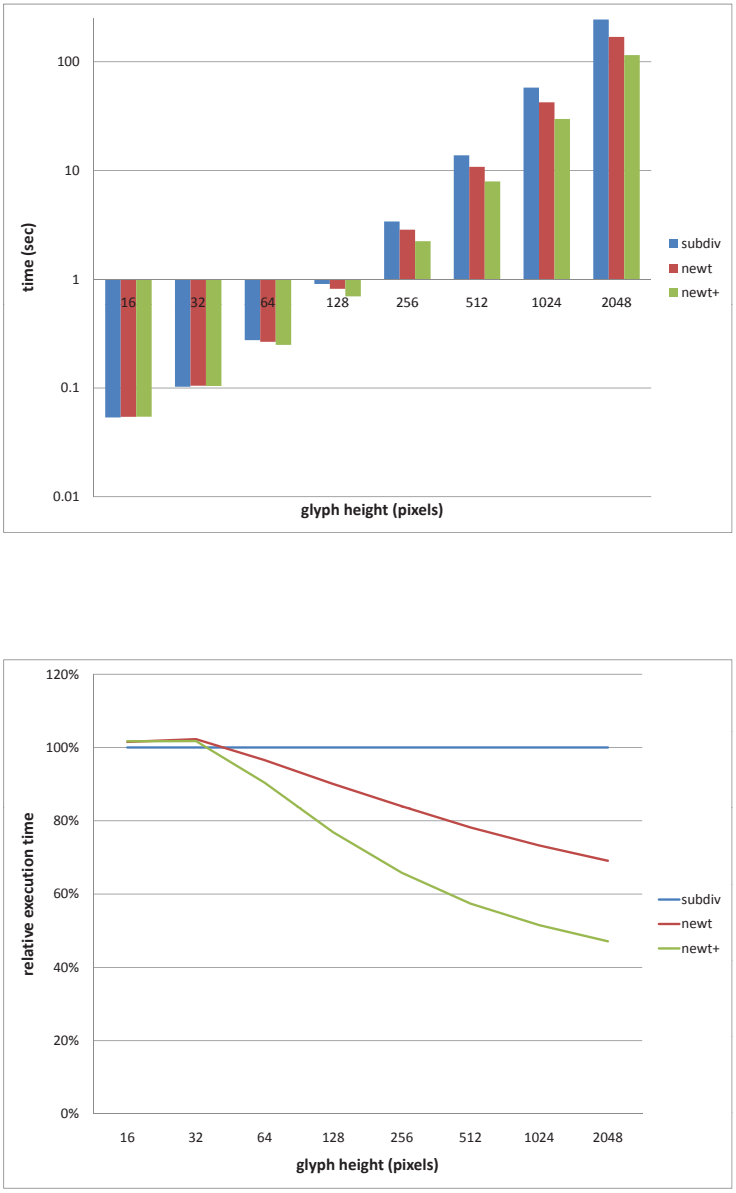
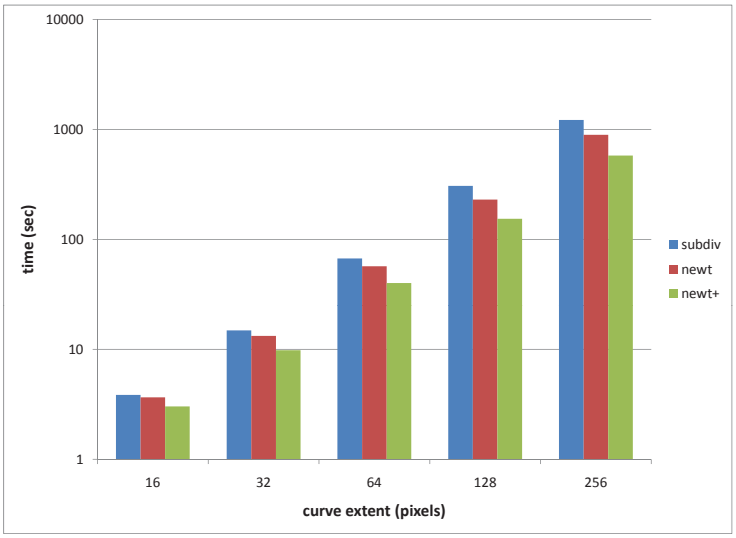


Figure 2: Antialising example: absolute and relative (subdivision=1.0) execution times of solvers at varying glyph heights.



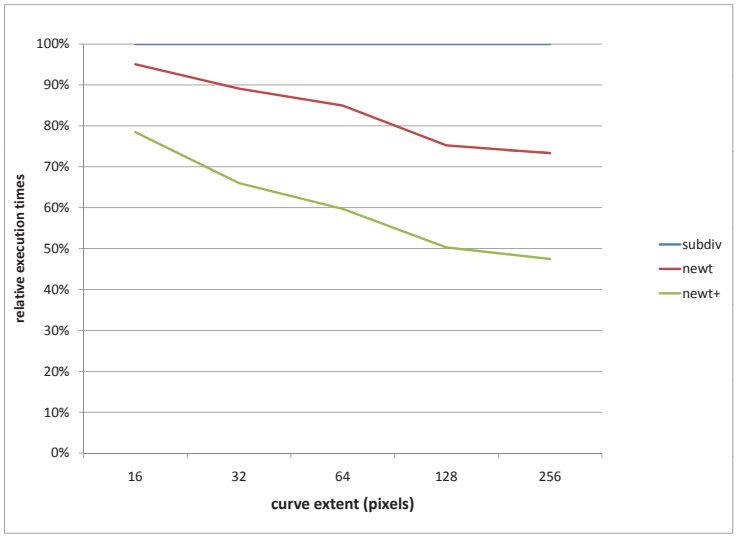


Figure 3: Brush example: absolute and relative (subdivision=1.0) execution times of solvers at varying sampling block sizes. The value on the x-axis denotes the number of pixels allocated to the larger of the x and y extents of the curve segment.

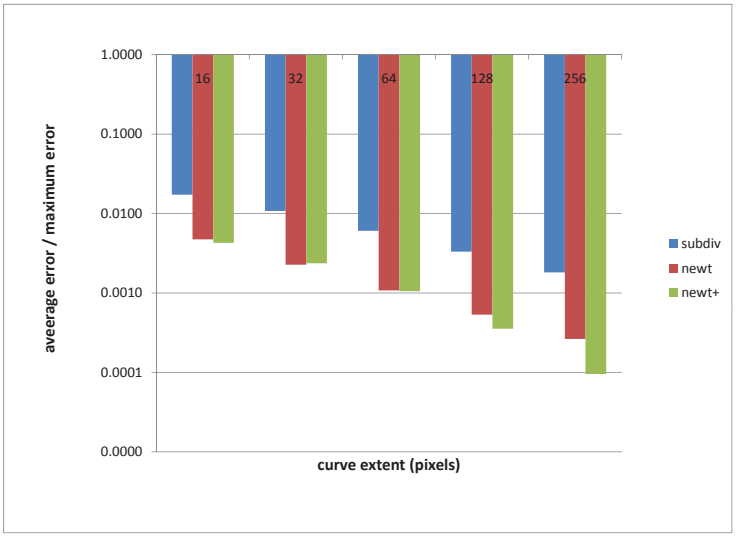


Figure 4: Average error as a fraction of maximum error.

4.3.2 Brush example

For a brushed path of width $2d$, sample points within d of a curve segment require a precise distance computation to determine the transverse parameter value. This example simulates the computations required to render brushed curves with a half-width that is 10% of the curve's larger dimension. Given a specified size value s, we scale each segment to s pixels in the vertical or horizontal dimension, depending on its aspect ratio, and compute a tight bounding rectangle. This rectangle is then padded by 10% in each direction, ensuring that all pixels within a 10% half-width of the segment will be queried.

Figure 3 shows the absolute and relative performance values for our three solvers over a variety of values for the size value s. Broadly speaking, the results are similar to those in the antialiasing example, in that both Newton-based algorithms yield better performance as the problem size is increased. Like the antialias example, execution time scales roughly linearly with problem size (in this case, the size of the bounding area of the segment, plus padding), as does the performance improvement achieved by the Netwon-based solvers.

For the sake of brevity, the remainder of our analysis will be confined to the brush case only.

4.4 Precision

We analyzed the precision of our solvers by comparing their output to that of a solver set to terminate when the pixel distance between the approximate curve points produced by adjacent iterations falls below 10^{-9}. This allowed us to measure the distance error at each pixel location for all input/solver combinations in the brush example. To obtain a result correct to within $\frac{1}{4}$ pixel for all sample points, we found that we needed to set the termination constraint (i.e., the maximum distance allowed between successive estimated curve points, measured in curve space) to $0.1\frac{d_{curve}}{d_{pixel}}$.

While all solvers did attain the desired precision at all sample points, this is achieved in an overly confservative manner. Figure 4 shows that the average error is 2–4 orders of magnitude smaller than the maximum error. This suggests that a better convergence criterion might allow us to perform fewer iterations

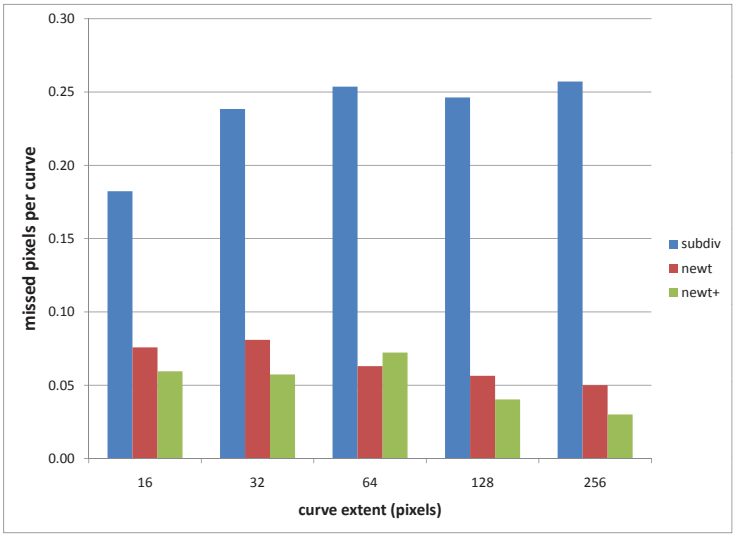


Figure 5: Average number of missed pixels per curve drawn.

to meet any particular precision goal. That said, we don't know how much precision changes as a function of iteration count–given Newton's quadratic convergence, multiple digits of precision may be gained in a single iteration. Thus, it is possible that eliminating a single iteration could push a large fraction of queries beyond the desired precision threshold. A quick experiment in which we forced the `newt+` solver to terminate after 3 iterations (the average iteration count), only on queries reusing a neighboring parameter value,[15] yielded maximum distance errors of 2–9 pixel widths, which is clearly unacceptable.

We can also examine precision in terms of a client-specific metric. In the brush example, the distance solver is used to implement an inclusion test, in which pixels found to lie within a fixed distance (±10% of the larger of the vertical and horizontal extents of the curve segment) of the brush curve are deemed to be part of the brush. Since the true distance is always less than the returned distance, inclusion test will find more on-brush pixels as the error is reduced. Figure 5 shows the average number of on-brush pixels omitted (per curve) for various curve sizes and solvers. For example, a client using the `newt` solver on size-256 curves will, on average, fail to include a single pixel in one of every 20 curves evaluated. Given that the maximum error of all queries is limited to .25 pixels, any omitted sample would necessarily lie in the outermost quartile of the 1-pixel-wide antialiasing region, and thus, if included, would have been rendered as mostly transparent anyway.

In essence, this is an inverse measure of the average precision obtained for sample points just inside the maximumum relevant distance (brush half-width) from the curve, rather than for all sample points. Given that both the relevant sample area[16] and the precision requirement scale roughtly linearly with the curve extent, we would expect to see roughly constant miss counts across problem sizes, and we do indeed see less than 2x scaling as the problem is scaled 16x. It's also not surprising that the Newton-based techniques do better at larger problem sizes, as Figure 4 showed an apparent correllation between curve extent and improvement over worst-case precision. The difference between the subdivision-based solver's behavior here

[15]The hypothesis being that only the initial, unprimed query would require an above-average amount of iteration, while preprimed query points should converge very quickly.

[16]This largely a function of the brush's perimeter, rather than its area.

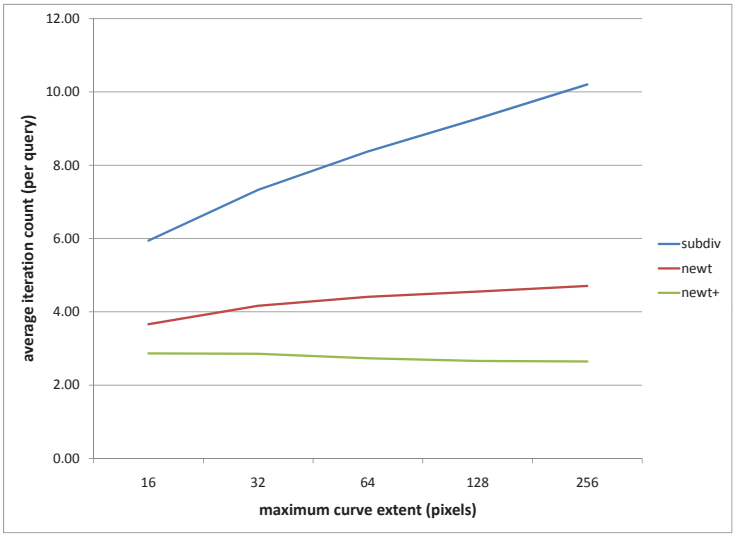


Figure 6: Average iterations per query as a function of curve extent.

and that shown in Figure 4 is not something we are able to explain at the moment.

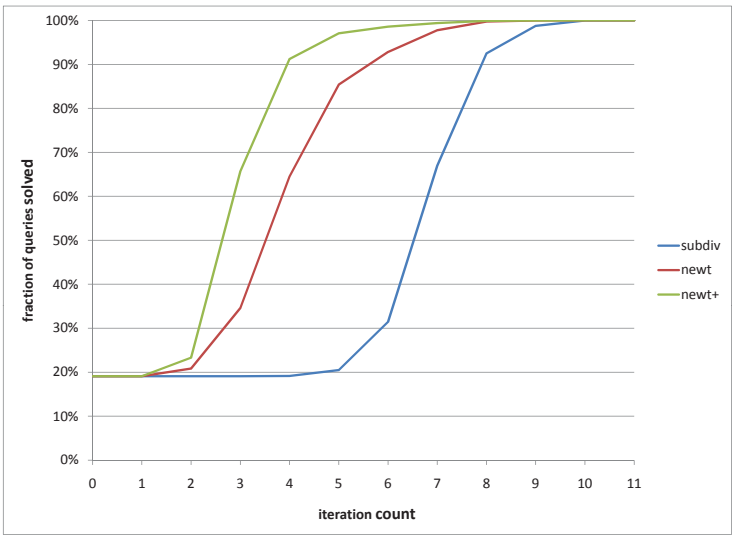
4.5 Iterations and Iteration Costs

We are also interested in the number of iterations required for our various solvers to converge. This statistic is interesting for two reasons:

1. the number of iterations performed is the main problem-size-dependent component in overall performance, and
2. a sufficiently small maximum iteration count might allow us to implement the solver as combinational logic; e.g., in an FPGA.

First, we must clarify what our iteration counts mean. At each pixel, our algorithm first determines whether a perpendicular distance to the curve exists, and only invokes the iterative solver if this is the case. This computation is not considered to be part of an "iteration." At this point, the initial estimated parameter value is constructed, and the solver enters a loop that computes the dot product value for the estimated parameter, and either returns successfully, or computes a new parameter value and recurses. Our iteration counter is incremented each time the loop's body is executed.[17] Because the loop's termination criterion operates by comparing a property derived from two parameter estimates (the distance between their corresponding curve points), the loop (if entered) will always recurse at least once, yielding a minimum iteration count of 2 when a perpendicular distance exists.

[17]This differs from typical nomenclature, in which each recursive loop invocation increments the iteration counter. Because the majority of the loop body's computation (e.g., computing the approximate curve point, its distance from the query point, the resulting vector's dot product with the curve tangent, and the distance between the previous and current curve points) takes place even when the loop does not recurse, we feel that a measure based on how many times this code is executed will better model the solver's execution time than a recursion count would.



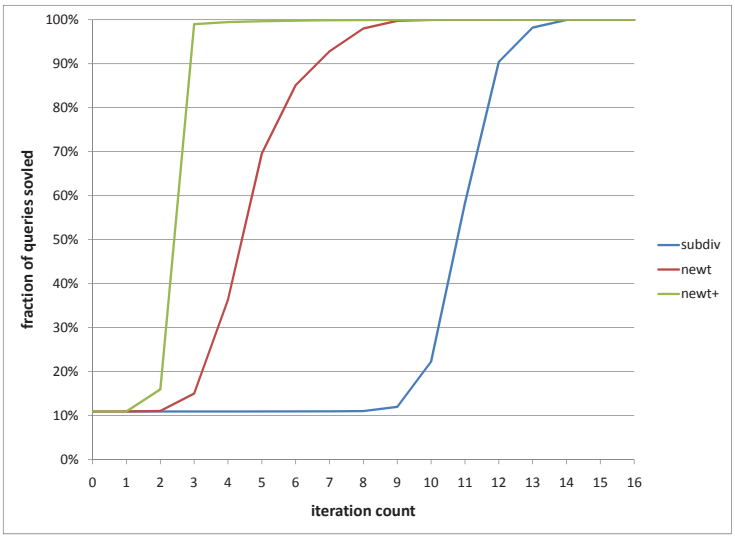


Figure 7: Cumulative fraction of queries solved as a function of iteration count for the brush example at size 16 (upper chart) and size 256 (lower plot). For example, in the upper chart, the `newt+` solver solves 90% of queries within 4 iterations, while while the `newt` and `subdiv` solvers solved only 60% and 20%, respectively.

Figure 6 shows average iteration counts for our solvers at all problem sizes. Recalling that each increment in problem size (i.e., doubling the ratio $\frac{d_{pixel}}{d_{curve}}$) doubles the required precision, we can examine the iteration count as a function of precision. For subdivision, the iteration count is roughly linear in brush size, and thus logarithmic w.r.t. precision requirements. The Newton-based solver is less affected by precision, growing by only 1.1 iterations as precision requirements are raised 8x. Interestingly, the locality optimization (`newt+`) actually results in lower iteration counts for larger problem sizes. As the neghboring sample points (in curve space) grow nearer, the utility of using their results as the starting point for iterative search more than overcomes the costs of meeting the additional precision requirement.[18]

Figure 7 the shows the cumulative fraction of queries solved as a function of iteration count for two problem sizes. We see that 10-20% of queries do not iterate at all, as no perpendicular distance vector exists for those query points.[19] The maximum iteration count at which 100% of queries are solved is diffcult to read from the charts—in the upper chart, it is 11 iterations for all solvers, while in the lower chart, newt requrires 16 iterations, while the others require only 15. Over all problem sizes, the maximum iteration count grows with problem size, while the difference in this value between the three servers is never more one iteration.

On the size-16 example, 90% of queries are solved in 4 or fewer iterations by the `newt+` solver, while subdivision requires as many as 8 iterations to cover the same fraction of queries. Once again, we see the benefits of the locality optimizaion in the `newt+` solver, which improves coverage from 15% to 98% at the cost of only one addition iteration. We also see that, although all solvers have similar maximum iteration counts, the vast majority (99%) of queries require significantly fewer iterations than the maximum. For a software implementation, the existence of a long, narrow tail of query positions requiring these additional iterations doesn't really matter, as they account for such a small fraction of the operations performed. For hardware implementations, however, avoiding these outliers could have a large impact, allowing the amount of circuitry (and associated latency) to be reduced by as much as a factor of 4 for the `newt+` solver on the size-256 example, Unfortunately, we have thus far been unable to characterize the query regious where the outlying query counts are achieved, leaving us unable to develop alternate means for evaluating them.

To judge the overall effect of iteration count improvement on performance, consider the following. In Figure 7, `newt+` achieves a 2-3x reduction in averate iteration counts, but its performance improvement (shown in Figure 3) is at most a factor of 2. Several factors may apply here. First, we have multiple costs which are independent of iteration count: the per-curve setup time for the solver (e.g., axis analysis, construction of parameters for curve, dot product, and derivative formals) and the per-pixel setup times (computing and comparing the signs of the point-endpoint dot products). These costs are incurred even in zero-iteration cases. Second, the Newton-based solvers perform more calculation per iteration, as they must perform an additional derivative computation, as well as an additional comparison for checking to see if the Newton estimate is in range.

We attempted to quantify these factors by re-executing the benchmarks with various bits of code (e.g., the iterative portion of the solver, or all per-query computations) disabled, and came to the following conclusions:

- the per-curve setup time has no visible effect on execution time, even for the size-16 case, which makes relatively few (approximately 256) queries, and
- the per-pixel setup time is significant, accounting for 17–36% of overall execution time. Its effect varies across solvers (more significant for more efficient solvers) and across problem sizes (larger problems require more iterations by all solvers), and
- the per-iteration cost of the Newton-based solvers is approximately 150% of the subvision solver, while the `newt+` solver is 0-4% more efficient than plain Newton..[20]

[18] We do not know if this effect scales to very large curves; e.g., zooming a single brush to fill a wall-sized display. We haven't tested this case yet because our single-CPU test harness would an impractically large amount of time to evaluate it.

[19] The fraction of such query points is smaller in the size-256 example because the padding required to contain the brushed curve becomes a smaller fraction of the overall region to be sampled as the problem size increases.

[20] We believe that this is primarily due to a sharply decreased need for subdivision-based correction of out-of-bounds Netwon

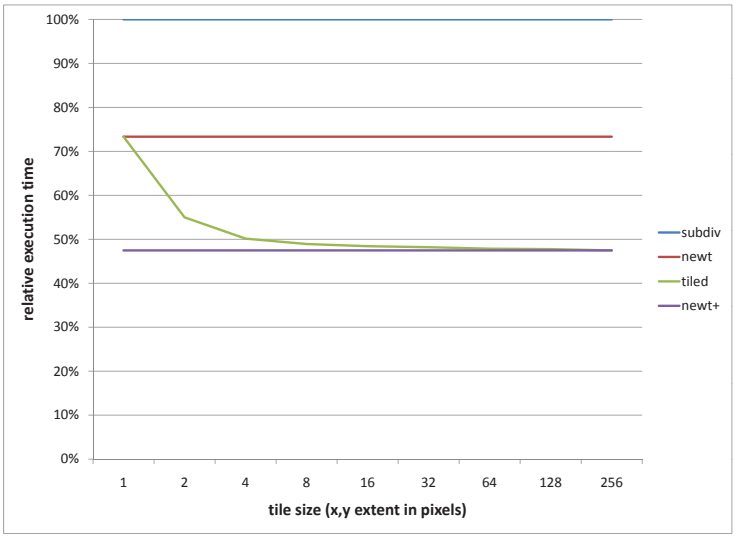


Figure 8: Relative execution time as a function of tile size, for task-parallel evaluation of the 256-pixel-wide brush scenario.

Across our problem sizes, the locality optimization yields average iteration counts of 3 or below, indicating that (on average) the initial estimate is usually revised twice. Expecting a sufficiently precise answer after one revision may be a bit of a stretch. Even if this is achieved, the high iteration-independent costs mean that we will experience diminishing returns. We estimate that reducing the average iteration count to 2 would yield at most a 23% reduction in execution time.

4.6 Tiling

We performed experments to examine how the `newt+` solver's use of locality is affected by two forms of tiling.

4.7 Tiling for task parallelism

To take advantage of task parallelism, we seek to divide the query space into multiple regions, or tiles, and serially process each tile in a distinct thread of execution. As we are still able to process neighboring pixels in series, we expect that the `newt+` solver should continue to perform well. The primary increase in cost will come from the need to evaluate each tile's first pixel without benefit of a prior neighbor computation. This is easily amortized for large tile sizes, but may be a problem with smaller tiles. Figure 8 shows that we can retain most of the performance benefits of `newt+` at tile sizes that are small relative to the query domain. In this example, most of the performance benefit of reuse is achieved even at a tile size of 4, or $\frac{4^2}{256^2} = 0.24\%$ of the query space.

estimates in the locality-optimized solver (`newt+`), which reduces the number of such cases by up to two orders of magnitude in some cases (but zero in others). Any definitive correlation seems impossible, as the convergence behavior of Newtonian methods is known to be fractal in nature.

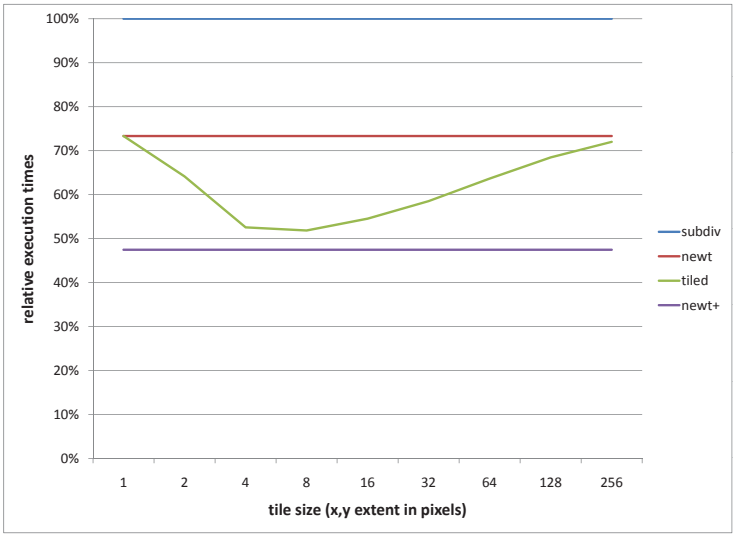


Figure 9: Relative execution times as a function of tile size, for data-parallel evaluation of the 256-pixel-wide brush scenario.

4.8 Tiling for stream parallelism

Stream parallelism, as implemented on the GPU, gains most of its benefits by limiting communication between independent processing elements. If we assign a GPU thread to each query point, all queries are processed in parallel, and no query can take advantage of its neighbor's results. We can partially address this issue through the use of a multipass algorithm. The first pass, which uses one thread per tile, executes a distance query on each tile's center point, and saves the result. The second pass, which uses one thread per query point, initializes its solver with the saved result from the corresponding tile. This initial estimate is thus less precise than that obtained from the preceding query in a serial implementation, but is still better than a non-locality-optimized solver's inital guess. Figure 9 shows the performance effects of various tile sizes. We see a tradeoff: small tiles give better precision in Pass 2, at the cost of more precomputation in Pass 1, while large tiles perform less precomputation but must operate with less-precise initial values. We observe a "sweet spot" favoring smaller tiles.

Note that in both the task-parallel and stream-parallel experiments, we are using sequential execution to determine the total work performed at various tile sizes, without modeling the degreee of parallellism available to perform that work. Your mileage may vary.

5 Conclusion and Future Work

We have described an approach to solving distance problems on conic curves. Important features of our technique include

- **axis-based decomposition** of curves enables a sound solution that evaluates a single relevant portion of the curve segment,
- **restricted Newton iteration** guarantees convergence,

- **reuse of solutions** improves performance, and
- **tiling** preserves performance in task- and stream-parallel implementations.

One path for future work involves implementation. The CPU implementation can stand some tuning, and a single-pass GPU implementation should be straightforward (modulo many details). The data in Figure 9 suggests that a multipass GPU implementation could take advantage of locality, but it remains to be seen if the improved performance due to locality will repay the costs associated with an additional pass.

Making the algorithm practical for a fixed-iteration-count hardware implementation will require CPU-based precomputation of estimates that are both sufficiently precise (to obtain a precise result after only a few "cleanup" iterations on the hardware) and sufficiently general (to fit within the available configuration bandwidth). Implementing such a scheme will likely require a better understanding of the distance formulation's convergence behavior than we have at the moment.

References

[Far89] Gerald Farin. Curvature continuity and offsets for piecewise conics. *ACM Transactions on Graphics*, 2(1):89–99, 1989.

[Hea97] Thomas L. Heath. *Archimedes: Works*. Cambridge University Press, 1897. 234–252.

[LB05] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. *ACM Transactions on Graphics*, pages 1000–1009, 2005.

[NH08] Diego Nehab and Hugues Hoppe. Random-access rendering of general vector graphics. *ACM Transactions on Graphics*, 27(5), 2008.

[Pav83] Theo Pavlidis. Curve fitting with conic splines. *ACM Transactions on Graphics*, pages 1–31, 1983.

[Pra85] Vaughan Pratt. Techniques for conic splines. *ACM Transactions on Graphics*, 19(3):151–159, 1985.

[PTVF92] William H. Press, Saul A. Teukolsky, Willam T. Vettering, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition edition, 1992.

[QMK08] Zheng Qin, Michael D. McCool, and Craig Kaplan. Precise vector textures for real-time 3d graphics. *ACM Transactions on Graphics*, 27(5), 2008.

[Ruf11] Erik Ruf. An inexpensive bounding representation for offsets of quadratic curves. In *High Performance Graphics*, 2011.

[WKA02] Hongling Wang, Jospeh Kearney, and Kendall Atkinson. Robust and efficient computation of the closest point on a spline curve. *Curve and Surface Design*, pages 397–405, 2002.

Appendix

5.1 Finding curve axes

This section describes in more detail how the axis-based partition of the space of query points is derived from the Bezier control triangles describing the curve segment. The client provides the control triangle $p_0 ... p_2$ along with either a sharpness factor $w \geq 0$ or a fourth point p_3 lying on the curve segment. In the latter

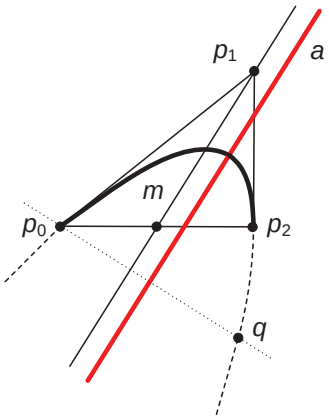


Figure 10: Technique for finding the axis of a parabolic curve from a parabolic segment's Beizer triangle. The Bezier triangle $p_0p_1p_2$ describes both a curve segment (solid black curve) and an infinite parabolic curve (parts of which are shown as dashed lines). The "midline" m passing through p_1 and the midpoint of p_0p_2 is known to be parallel to the axis of the curve [Hea97]. Any line perpendicular to m will intersect the curve at points which are symmetric with respect to the axis; thus, the dotted line through p_0 enables us to find a symmetric point q. Averaging p_0 and q gives us a point through which the axis (solid red line) must pass, so displacing m to pass through this point yields the axis line a.

case, we can obtain w by substituting p_3 into the implicit equation[21] for conic curves and solving for w, yielding $w = \sqrt{\frac{b_1^2}{4b_0b_2}}$, where the b_i are the barycentric coordinates of p_3 with respect to the triangle $p_0...p_2$.

If w is equal to 1, the curve segment is part of a parabolic curve. Such a curve segment has the property that the line l connecting the midpoint of the line p_0p_2 to the middle point p_1 is parallel to the axis a of the underlying parabola [Hea97]. Perpendicular to this line, we construct a chord that intersects the curve twice. Averaging the intersection points yields a point on the axis, and thus the axis line, the point at which it intersects the curve, and the corresponding parameter value.[22] Figure 10 gives a graphical representation of this process.

Otherwise, (assuming a nondegenerate conic curve), the segment is part of either a hyperbolic ($w > 1$) or elliptic ($w < 1$) curve. By statically expanding the definitions of the barycentric coordinates in Farin's barycentric version of the implicit equation

$$\tau_1{}^2 - 4w^2\tau_0\tau_2 = 0$$

where the τ_i are the barycentric coordinates of the query point (x, y) with respect to the control points p_i, and equating the result with Pavlidis' version of the implicit equation [Pav83, p. 5, equation (3.1)], we obtain representations for each of the the six coefficients of that equation

$$f(x,y) \quad = \quad ax^2 + 2hxy + by^2 + 2ex + 2gy + c \quad = \quad 0$$

in terms of the segment's control points and w parameter value:

[21] See Farin [Far89, p. 90] or Pratt [Pra85, p. 152].

[22] The parameter value is obtained by substituing the parameterized formula for the curve into the equation defining the line, which results in an expression that is quadratic in the parameter, and can be solved by appling the quadratic formula.

$$
\begin{aligned}
a &= {y_0}^2 + {y_2}^2 - (2y_2y_0) + 4w^2(y_2y_0 - y_2y_1 - y_1y_0 + {y_1}^2) \\
b &= {x_0}^2 + {x_2}^2 - (2x_2x_0) + 4w^2(x_2x_0 - x_2x_1 - x_1x_0 + {x_1}^2) \\
c &= {y_0}^2x_22 + {x_0}^2{y_2}^2 - 2y_0x_2x_0y_2 + 4w^2({x_2}^2y_2y_0 - y_1x_2x_1y_0 \ - x_1y_2y_1x_0 + {y_1}^2x_2x_0) \\
g &= y_0x_2x_0 + y_2x_0x_2 - y_0x_2x_2 - x_0x_0y_2 + \\
&\quad 2w^2(-x_1x_1y_2 - {x_1}^2y_0 +_y 1x_2x_1 + x_1y_1x_0 + x_1y_2x_0 + x_2x_1y_0 - 2y_1x_2x_0) \\
e &= x_0y_2y_0 + x_2y_0y_2 - x_0y_2y_2 - y_0y_0x_2 + \\
&\quad 2w^2(-y_1y_1x_2 - {y_1}^2x_0 +_x 1y_2y_1 + y_1x_1y_0 + y_1x_2y_0 + y_2y_1x_0 - 2x_1y_2y_0) \\
h &= (x0 * y2 + y0 * x2 - y0 * x0 - y2 * x2 + \\
&\quad 2w^2(-y_0x_2 - y_2x_0 + y_2x_1 + x_2y_1 + x_1y_0 + y_1x_0 - 2y_1x_1)
\end{aligned}
$$

Prior to performing queries on a curve, we compute the values of these coefficients and use them to evaluate various expressions from [Pav83, Section 3.1]. which yield the center point and axis lines of the curve. In particular, the center point (through which both axes pass) is

$$p_c = (\frac{-eb + gh}{ab - h^2}, \frac{-ag + eh}{ab - h^2})$$

and the axes are parallel to the eigenvectors of Q, where

$$Q = \left| \begin{array}{cc} a & h \\ h & b \end{array} \right|$$

If the segment is part of an ellipse, both axes are relevant; if it is hyperbolic, the relevant (major) axis is the one that intersects the curve segment. In either case, we compute the parameter values at which the relevant axes intersect the curve.

At this point, we have 1 to 4 parameter values at which the curve is intersected by an axis. At most two of these values will lie within the [0, 1] parameter interval defining the curve segment. We save these relevant parameter values, along with the corresponding curve normals, for later use in mapping query points to initial parameter intervals for the iterative solver (see Figure 1).

5.2 Distance Properties

In Section 3.2, we asserted the following properties of a symmetric conic curve C and query point q:

- The minimum distance vector v from q to C does not cross any axis of C, and
- Any other distance vector w from q to C must cross an axis of C.

Assertion (1) can be proven true by negation. If the query point q and the minimum-distance curve point p were to lie on opposide sides of a curve axis, we could reflect p across the axis, yielding a new point p' which would be closer to q. Thus, q and the curve point closest to q must lie in the same axis-partitioned region of the plane.

To prove Assertion (2), we make use of the property that outward-facing curve normals do not intersect one another or any curve axes.[23] Thus, for q outside of C, only one normal of C (coincident with v) intersects q, and does not cross an axis, so the statement is vacuously true.

For q inside of C, we first consider the portion of C lying in the same axis-bound region as q. Any inward-facing normal intersecting q (there is only one, coincident with v) will do so before reaching a curve axis. As for the portion of C lying in a different region, any inward normal (indeed, any vector whatsoever) must necessarily cross an axis to reach q.

[23] This is true only because we are explicitly ignoring the minor axis in the hyperbolic case; see the description of axis finding above in Section5.1.