DOI: 10.1111/j.1467-8659.2009.01635.x

EUROGRAPHICS 2010 / T. Akenine-Möller and M. Zwicker
(Guest Editors)

Volume 29 (2010), Number 2

Sketching Clothoid Splines Using Shortest Paths

Ilya Baran[1] Jaakko Lehtinen[1] Jovan Popović[1,2,3]

[1] MIT CSAIL [2] Adobe Systems Inc., Advanced Technology Labs [3] University of Washington

Abstract

Clothoid splines are gaining popularity as a curve representation due to their intrinsically pleasing curvature, which varies piecewise linearly over arc length. However, constructing them from hand-drawn strokes remains difficult. Building on recent results, we describe a novel algorithm for approximating a sketched stroke with a fair (i.e., visually pleasing) clothoid spline. Fairness depends on proper segmentation of the stroke into curve primitives — lines, arcs, and clothoids. Our main idea is to cast the segmentation as a shortest path problem on a carefully constructed weighted graph. The nodes in our graph correspond to a vastly overcomplete set of curve primitives that are fit to every subsegment of the sketch, and edges correspond to transitions of a specified degree of continuity between curve primitives. The shortest path in the graph corresponds to a desirable segmentation of the input curve. Once the segmentation is found, the primitives are fit to the curve using non-linear constrained optimization. We demonstrate that the curves produced by our method have good curvature profiles, while staying close to the user sketch.

1. Introduction

Constructing high-quality curves from hand-drawn input is important in both freehand illustration and sketch-based modeling applications (e.g., [IMT99,BBS08]). However, the curve literature has traditionally concentrated more on specification of curves through geometric constraints, such as fixed positions or tangents, rather than directly by sketching. In this work, we leverage the use of *clothoid splines* [MT91] as a first-class representation for sketched strokes. Clothoid splines have a piecewise linear curvature profile: they consist of a sequence of lines, circular arcs, and clothoid curves. The defining property of a clothoid (or Euler spiral) is that its curvature changes linearly with arclength. Levien [Lev09] provides an excellent discussion of the history and properties of clothoids. We describe an algorithm for fitting clothoid splines to complex, possibly closed sketches, using a combination of discrete and continuous optimization. We simultaneously minimize the number of curve segments, as required by fairness, and deviation from the stroke, while strictly enforcing a maximal order of continuity between segments. We demonstrate high-quality results on complex examples and provide comparisons to both state of the art techniques and commercial illustration software.

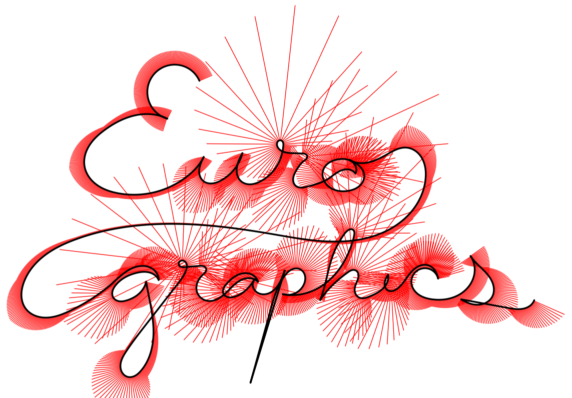
Figure 1: *A clothoid spline sketched using our algorithm. The red comb illustrates curvature along the path.*

Specifically, we address the key challenge of segmenting the input stroke into curve primitives — lines, arcs, and clothoids — by casting it as a shortest path problem on a weighted graph. After the segmentation is found, the individual primitives are matched up using nonlinear constrained optimization to guarantee continuity across the seg-

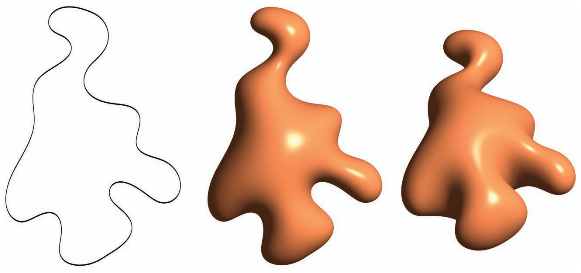

Figure 2: *A clothoid spline is inflated into a surface.*

ments. Our method allows both open and closed sketches of high complexity to be represented faithfully using few curve primitives. Figure 1 provides an example curve, and Figure 2 shows a 3D surface inflated from a curve sketched using our method. By controlling the edge costs using a few intuitive parameters, the user can generate a broad range of results – for example, the user may choose to produce only G^1 line-arc splines, favor fewer segments over approximation accuracy, disallow lines and arcs as primitives, or penalize inflections to varying degree (Figure 3). In addition, we describe an algorithm for editing curves by oversketching such that fairness is maintained, including in the part where the edit is connected to the rest of the stroke. Alternatively, existing algorithms for editing clothoid splines through control points [Lev09] could be used.

2. Related Work

Modeling and representing planar curves is one of the earliest applications of computer aided design and computer graphics. While we seek to represent hand-drawn strokes in a fair manner, early work on curve specification and editing is, in contrast, generally more concerned with interpolation of point data [Mor92]. However, the question of how to quantify the *fairness* of curves, and to design representations and algorithms that produce such curves, has received significant attention. Fairness is universally accepted to mean that the curvature of a curve behaves "nicely". According to Farin et al. [FRSW87], "[...] curvature (should) be almost piecewise linear, with a small number of segments." Note that this definition naturally includes straight lines, circular arcs, and clothoids (curves with a linear curvature profile).

Traditionally, sketched strokes are represented by fitting piecewise polynomial curves to the input, and additional fairing (smoothing) steps [FRSW87, SF90] are carried out to increase visual quality. Most often fairing techniques cannot, however, achieve pleasing curvature without deviating significantly from the original stroke. These algorithms typically use curve representations that are not designed to be intrinsically fair. A canonical example is the cubic spline: it is easy to fit to point data, but few guarantees can be given about its curvature profile.

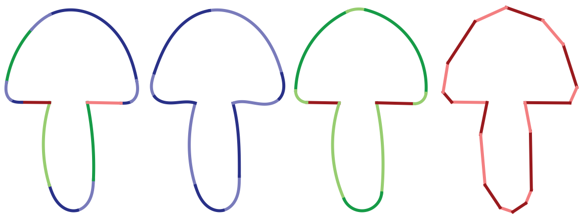

Figure 3: *Four curves generated from a single sketch by varying the parameters. Left to right: default G^2, only clothoids with no inflection penalty and a lower error cost, G^1 arcs and lines, and polyline. Lines are red, arcs are green, and clothoids are blue.*

This paper is inspired by the work of McCrae and Singh [MS08]. They presented a simple method for fitting a piecewise clothoid curve to a sketch using a combination of discrete and continuous optimization. The method relies on piecewise linear approximation of the discrete curvature profile computed from the sketch, and integrating the resulting curvature twice to yield the actual curve. While this produces good results for many inputs, the approximation error in the curvature space also gets integrated twice, causing the result to drift from the original stroke (Figure 10). In addition, the method cannot generate closed curves, although a modified method can [MS09]. Prior to this work, clothoid segments have been fit to geometric point and curvature constraints [NMK72, PN77, Sch78] and used for constructing splines [MW89, MT91, SK00], but their use as a sketching primitive has not been widely investigated.

Arc splines and biarc splines are G^1 sequences of circular arcs. Several authors have investigated their use for approximating point sequences and curves, e.g. [MW92, HE05]. The algorithm of Drysdale et al. [DRS08] computes a biarc spline that is guaranteed to have the minimum number of primitives while remaining within a set tolerance from the input. To obtain a provable guarantee, they only consider biarcs that start and end at input points and have prescribed tangents, which is a severe restriction. Similar in spirit to our work, they cast the segmentation into a shortest path problem on a graph constructed from overlapping biarc fits. However, they treat the input points as nodes and primitives as edges, while we treat the primitives as nodes and transitions between them as edges, enabling us to optimize over transitions, and not just the primitives. While we cannot give provable guarantees, incorporating the quality of transitions into the shortest path cost enables us to enforce G^2 continuity by avoiding unworkable transitions.

3. Method

The input to our algorithm is a sequence of 2D points from a user's mouse or stylus. Our method performs several pro-

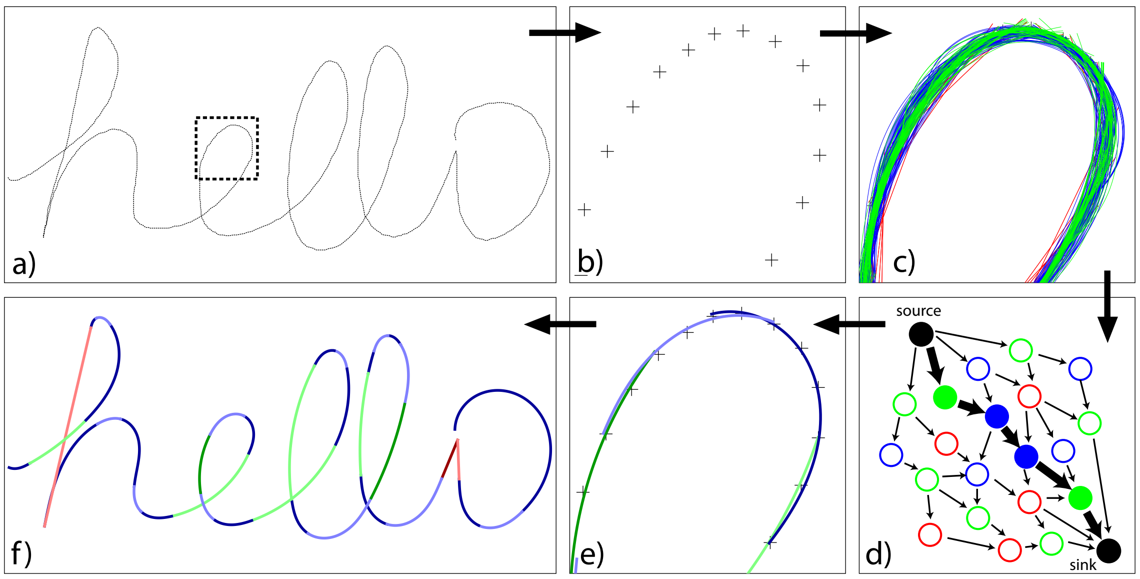

Figure 4: *The steps in our method.* ***a)*** *The input stroke.* ***b)*** *A zoomed-in view of the top part of the letter 'e', showing the resampled input.* ***c)*** *An overcomplete set of curve primitives is fit to the samples. Red denotes lines, green denotes arcs, and blue denotes clothoids.* ***d)*** *A graph is constructed from the curve primitives. Nodes denote primitives and edges denote transitions between primitives. The shortest path in the graph corresponds to good segmentation of the input into segments.* ***e)*** *The segmentation that corresponds to the shortest path. Note that the primitives do not quite match up.* ***f)*** *The final result, with primitives rendered in different colors, obtained by solving a non-linear program that enforces the desired order of continuity between primitives in the shortest path.*

cessing steps on these points to end up with a fair curve approximating the user input (Figure 4):

1. **Closed Curve Detection** — determine whether the sketched curve is almost closed, and, if so, make it precisely closed.
2. **Corner Detection** — determine which samples are intended to be sharp corners. This partitions the stroke into disjoint sub-strokes with G^0 joins.
3. **Resampling** — to reduce the problem size and make the sampling more regular, resample the sketched stroke in a curvature-sensitive way.
4. **Primitive Fitting** — for every contiguous subsequence of samples, fit a candidate line, arc, and clothoid. This results in an overcomplete set of overlapping primitives.
5. **Graph Construction** — construct a weighted graph with the primitives as nodes and transitions between primitives as edges, such that weights denote quality of the transition. To control the output of our algorithm, the user provides the costs for a line, an arc, and a clothoid, for G^0, G^1, and G^2 transitions, for inflections (points where the curvature changes sign), the approximation error cost and the penalty for short primitives.
6. **Shortest Path** — find an acceptable shortest path through the graph, validating transitions in the process. This step picks out a high-quality segmentation of the input stroke into curve primitives and transitions between them. For closed curves, we find an approximate shortest cycle.
7. **Merging** — enforce the continuity constraints on the chosen primitives by solving a nonlinear program.

In our work, all curves (including polylines) are parameterized by arclength parameter s. The tolerances and other distances are in pixels, and we assume that the monitor or tablet is roughly 100 DPI.

3.1. Closed Curve and Corner Detection

To determine whether the curve is closed, we threshold the distance between the first and last point, using 15 pixels as the cutoff. If the curve is closed, we find the two points on opposite ends of the curve that are closest to each other and make them the new start and end points, trimming off the ends. We then geometrically close the curve by moving all of the points as follows: let $\mathbf{v}$ be the vector from the first to the last point. Each point is moved by $(0.5 - s/l) \cdot \mathbf{v}$, where s is the arclength parameter of the point along the curve and l is the total curve length.

For corner detection, we developed a method that measures how likely a sample point is to be a corner by comparing how well its neighborhood can be approximated by a single arc against how well it can be approximated by two

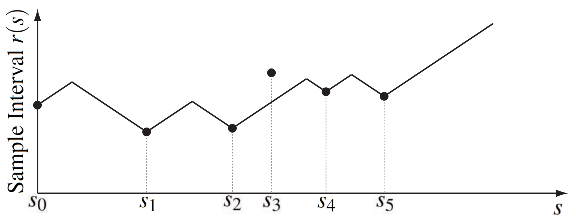

Figure 5: *In this figure, the target resampling interval $r(s)$ is computed from curvature estimates at six original samples s_i. The value of $r(s_i)$ at the original samples is $2\pi/\gamma c_i$ except at s_3, where that would result in too large a sample interval, given the neighbors. The slope of all of the lines is $\pm\beta$.*

arcs that meet at that point. However, the focus of this paper is not corner detection and we do not claim it as a contribution. Better results could likely be obtained by incorporating stroke speed, as done by Sezgin [Sez01].

3.2. Resampling

The initial mouse or tablet samples can be very numerous, redundant, and irregular. We carefully resample the polyline formed by the initial samples: too many samples will result in poor performance, while too few will lead to poor reproduction of the user input.

We use the corners to partition the input points into separate polylines and process each polyline individually for resampling. Let s be the arclength parameter of a polyline and s_i be the parameters at the polyline vertices. For resampling, we first construct a sampling rate function $r(s)$ that defines the desired local step size along the curve (Figure 5). Our function ensures that the sampling is denser in areas of high curvature, while enforcing that the rate changes smoothly. This is similar to a guidance field used for remeshing [SSFS06]. We estimate the curvature c_i at the polyline vertices and define

$$r(s) = \min_i (\beta|s - s_i| + 2\pi/\gamma c_i),$$

where β is the sample rate falloff (we use 0.2), and γ is the number of points we want in a circle (we use 15). We clamp $r(s)$ to be between two and 1000 pixels to avoid pathological cases. The curvature at a vertex is estimated by fitting an arc to the neighborhood of the vertex (15 pixels in each direction) using the method in Section 3.3 and taking the arc curvature as c_i.

We compute the resampling by starting from $s = 0$ and taking steps according to $r(s)$ such that the step between samples is as large as possible, without violating the sampling rate requirement. More precisely, we find parameters s'_j for the new points such that $s'_0 = 0$ and:

$$s'_j - s'_{j-1} \leq \min_{s\in[s'_{j-1},s'_j]} r(s).$$

We stop when we go past the end of the curve. At this point, because $|r(a) - r(b)| \leq \beta|a - b|$, the ratio between two adjacent sample intervals is at most $1+\beta$. However, we now have the last sample past the end of the polyline and we need to move it to the end. Simply moving it to the end would result in a non-uniform sampling, while scaling all parameters could move samples off high-curvature regions. We therefore only move the last four samples inward so that the last sample coincides with the end of the curve. We finally compute the new sample points $\mathbf{p}_j$ along the polyline using the parameters s'_j.

3.3. Primitive Fitting

To enable optimization over primitive curve shapes, we need each primitive curve to be defined in terms of a few variables. We use a starting point, a starting direction, and a length to define a line segment. An arc segment also has a starting curvature, and a clothoid also has an ending curvature. Using these definitions allows an arc to degenerate to a line segment or a clothoid to an arc.

Both fitting primitives to a sequence of samples and enforcing continuity constraints involves optimizing the variables that define the primitives, using distance to the samples as an objective function. For example, a point on a line segment L at distance s from the start is $L(x,y,\alpha,l,s)$, where x, y, α, and l are the four variables that define the line segment. In order to perform optimization, we need to be able to compute the Jacobian of the point position with respect to the variables, as well as the parameter s.

While this is easy for a line, we need to be careful even with arcs: when an arc has very small curvature (say, $\kappa = \varepsilon$), we cannot use sines and cosines to evaluate it because the center is very far away. Approximating the arc with a line when ε is below a threshold, i.e., setting $C(x,y,\alpha,l,\kappa,s) \approx L(x,y,\alpha,l,s)$ gives good results for the position, but $\partial C/\partial\kappa$ is wrong. Using $C \approx L(x,y,\alpha,l,s) + \kappa s^2(\sin\alpha, -\cos\alpha)/2$ yields the correct derivatives. Clothoids that degenerate to lines or arcs similarly need approximations that can be derived by finding a perturbation of the line or arc that has the same curvature profile as the clothoid to first order. Because we use automatic differentiation, this is easier than approximating the Jacobian directly near degenerate configurations.

From the resampling, we have n sample points $\mathbf{p}_i$ and we fit primitives to every subsequence of them that does not span a corner, resulting in a set of overlapping primitives. Primitives that are too far off from the sketch are discarded from further consideration. To make the fitting independent of the sampling rate, with each sample point, we associate a weight, $w_i = \|\mathbf{p}_{i-1} - \mathbf{p}_i\| + \|\mathbf{p}_{i+1} - \mathbf{p}_i\|$ (with indices clamped if the curve is open and taken modulo n if the curve is closed). For a sequence of points $\mathbf{p}_i, \ldots, \mathbf{p}_{i+k}$ (where the index is taken modulo n if the curve is closed) and a candidate curve C, we evaluate the quality of the fit with the

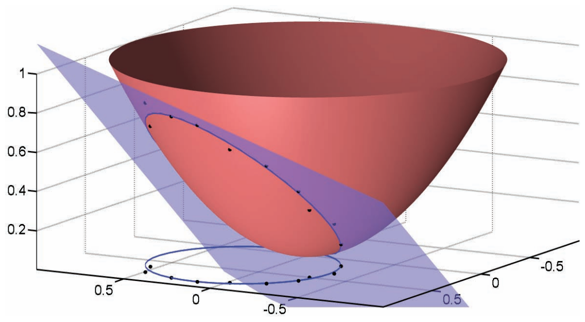

Figure 6: *The points on the plane are lifted onto the paraboloid $z = x^2 + y^2$ and a plane is fit to the lifted points. The intersection of this plane and the paraboloid, projected onto the xy plane, is a circle that is a good fit to the data points.*

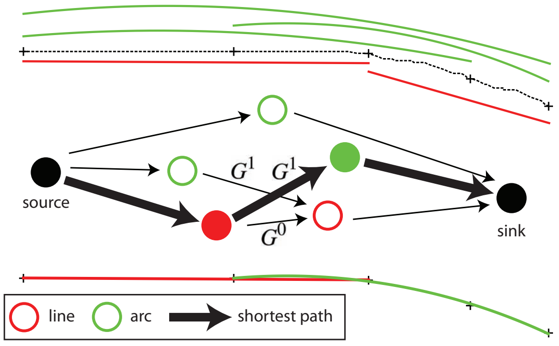

Figure 7: *A simplified example showing a subset of the primitives fitted to a user stroke (resampled to five samples). Top: exploded view of primitives. Middle: constructed graph on these primitives and shortest path. The graph is weighted, so the single arc is not necessarily the shortest path because it is a worse approximation. Bottom: the two primitives in the shortest path.*

following objective function:

$$2w_i\|\mathbf{p}_i - C(0)\|^2 + 2w_{i+k}\|\mathbf{p}_{i+k} - C(l)\|^2 + \sum_{j=i+1}^{i+k-1} w_j d_C(\mathbf{p}_j)^2, \tag{1}$$

where d_C is the distance to the curve and $C(0)$ and $C(l)$ its endpoints. We weigh endpoints more heavily to facilitate transitions.

For each starting point i, we fit line segments to $\mathbf{p}_i, \ldots, \mathbf{p}_{i+k}$ for increasing k until the fit error exceeds a specified tolerance. The method to fit a line is well known: the line that minimizes the weighted sum of squared distances to the points passes through the mean $\bar{\mathbf{p}} = \sum_j w_j \mathbf{p}_j / \sum_j w_j$ (where j indexes over $[i, i+k]$) and its direction is the eigenvector of $\sum_j w_j(\mathbf{p}_j - \bar{\mathbf{p}})(\mathbf{p}_j - \bar{\mathbf{p}})^T$ that corresponds to the larger eigenvalue. To get a line segment, we find the endpoints by projecting $\mathbf{p}_i$ and $\mathbf{p}_{i+k}$ onto the fit line.

For arcs, there is no exact solution in closed form, but very good approximations using algebraic distance are possible (e.g., [Pra87]). We use a fast and simple method: lifting the points $\mathbf{p}_j = (x_j, y_j)$ onto a paraboloid $(x_j, y_j) \rightarrow (x_j, y_j, x_j^2 + y_j^2) = \mathbf{p}'_j$ takes circles in 2D to ellipses in 3D (Figure 6). Conversely, the plane of the ellipse uniquely determines the original circle. Finding the weighted best fit plane to the 3D points is easy: it passes throught the mean $\bar{\mathbf{p}}' = \sum_j w_j \mathbf{p}'_j / \sum_j w_j$ and its normal is the eigenvector of $\sum_j w_j(\mathbf{p}'_j - \bar{\mathbf{p}}')(\mathbf{p}'_j - \bar{\mathbf{p}}')^T$ that corresponds to the smallest eigenvalue. The projection of the intersection of the plane and the paraboloid $z = x^2 + y^2$ onto the xy plane yields a circle that fits the 2D points well. This method is not invariant to the choice of coordinate system and we use $\mathbf{p}_i$ as the origin.

The line and arc fitting methods are very fast: because the means and the matrices can be computed incrementally, we can compute the $k-1$ best fit lines (or circles) to $\mathbf{p}_i, \ldots, \mathbf{p}_j$ for all j with $1 < j \leq k$ in $O(k)$ time. Because an arc has more degrees of freedom than a line segment, we can generally fit arcs to longer subsequences of points.

We fit clothoids numerically. If an arc fits the k points well enough not to be discarded, we use it as an initial guess, and if not, we use the clothoid fit to all but the last point, which we computed previously. Starting with the initial guess, we use the Gauss-Newton algorithm to optimize the objective function (1) over the clothoid parameters. An alternative initial guess would be to fit a line in curvature space [MS08], or a parabola in direction space.

The objective function is a sum of squares because it simplifies optimization, but for determining if the primitive is good enough to be considered further, we use maximum error instead:

$$\tau = \max\left(\|\mathbf{p}_i - C(0)\|,\ \|\mathbf{p}_{i+k} - C(l)\|,\ \max_{j=i+1}^{i+k-1} d_C(\mathbf{p}_j) \right) \tag{2}$$

We use a cutoff of five pixels.

3.4. Graph Construction

We now construct a weighted directed graph such that a path in the graph corresponds to an approximation of the curve with a sequence of primitives and transitions. The total cost of the nodes and edges along a path should reflect the quality of the curve approximation, with user-specific weights determining the behavior. We construct a graph node for every primitive that we fit earlier. Edges are constructed for every possible transition between the primitives (Figure 7): a primitive C is connected to all other primitives C' whose starting point is sufficiently close to the end point of C (see below). If the sketch is not closed, we construct a special start node, with edges to each primitive that starts on the first sample and a special end node, with edges from each primitive that ends on the last sample. Conceptually, both nodes and edges in the graph have weights, but instead of keeping the weights on the node, half of each node weight is added to all of the incoming edges of the node and half to all of the outgoing edges. The subsequent graph search therefore deals with an edge-weighted graph.

To compute the weight of a node corresponding to a primitive, we start with the user-specified cost of the curve type (line, arc, or clothoid). In the default scenario, lines have the lowest cost and clothoids have the highest, but the user may wish to adjust them or disallow primitives of a particular type altogether. We add an approximation penalty, computed from the error τ in Equation (2): $p(\tau) = w_E \cdot \max(\tau - 1, 0)^2$, where w_E is the user-set error weight. Using $p(\tau)$ ensures that curves with residual under one pixel do not get penalized at all and larger residuals get penalized progressively more severely. Finally, because short clothoid segments harm fairness, we add a penalty for curve primitives shorter than a threshold: $w_S \cdot \max(l_{\min} - l, 0)^2$, where w_S is the user-set penalty weight, $l_{\min}$ is the shortness threshold (we use 30 pixels), and l is the length of the primitive.

An edge encodes a G^0, G^1, or G^2 transition between two curves. A G^2 transition is only possible when at least one of the curves is a clothoid, and a G^1 transition is only possible when at least one of the curves is an arc or a clothoid. At corners only G^0 transitions are possible. The main consideration in constructing transitions is that they cannot be geometrically enforced until the entire path has been chosen: because primitives are fit independently, their endpoints are unlikely to match up at all, much less with any sort of higher order continuity. Therefore, when we set up a transition edge, it needs to be likely that that transition can actually be enforced and the weight of the transition edge should reflect the difficulty of enforcing this transition. Because of this, we only set up G^2 transitions between curves that overlap by two sample intervals (i.e., the first curve ends at sample i and the second starts at sample $i-2$) and G^1 transitions between curves that overlap by one sample interval. We construct G^0 transition edges between curves without overlap.

The weight of an edge is the sum of two components: the first is the base cost for the type of transition. In the default scenario, G^1 and G^0 transitions have very high base cost, while G^2 has zero base cost. The second component of the edge weight is the estimate of how much extra penalty for the deviation from the user samples would be introduced by enforcing the transition constraints. Recall that τ_1 and τ_2 are the original errors for the two curve primitives computed from Equation (2), and let τ'_1 and τ'_2 denote the estimated errors after the continuity constraints have been enforced (computed as described below). To penalize the additional error caused by enforcing continuity, we add $p(\tau'_1) - p(\tau_1) + p(\tau'_2) - p(\tau_2)$ to the edge cost.

Because of the large number of edges in the graph, we estimate τ'_1 and τ'_2 using very cheap heuristics and will refine the edge weight once a candidate shortest path is found (as described in Section 3.5). Our cheap estimate of the deviation is based on how far the curve primitives are from satisfying the transition continuity constraints. For each curve, we compute τ' as follows. Let d be the degree of continuity and let $\mathbf{p}_i$ for $i = 1, \ldots, d+1$ be the one, two, or three samples on which the two primitives overlap. Let s_i^1 and s_i^2 be the parameters of the closest point to $\mathbf{p}_i$ on the first and the second curve primitive, respectively. We estimate c_0, the deviation from satisfying G^0 continuity by assuming the curves can be joined at any of the overlapping samples and taking the minimum distance: $c_0 = \min_i \|C_1(s_i^1) - C_2(s_i^2)\|$, where C_1 and C_2 are the curve primitives. Similarly, the deviation from satisfying G^1 continuity is the minimum difference in the curves' direction over the samples $c_1 = \min_i \cos^{-1}\left(C'_1(s_i^1) \cdot C'_2(s_i^2)\right)$. However, if the differences are of different signs at two samples, then somewhere in between, the two curves must have parallel tangent vectors and we therefore set $c_1 = 0$. The deviation from satisfying G^2 continuity is measured similarly: $c_2 = \min_i \left|\kappa_1(s_i^1) - \kappa_2(s_i^2)\right|$ (again, if the curvature difference changes sign, we set $c_2 = 0$).

For a G^2 transition between two clothoids, the continuity differences are divided equally between them and the new transition error is:

$$\tau' = \tau + \frac{1}{2}\left(c_0 + c_1 l/2 + c_2 l^2/4\right),$$

where l is the primitive curve length. The weights are motivated by considering how much the primitive will move on average, given a change in position, angle, or curvature. For G^0 and G^1 transitions, we omit the relevant terms in computing τ'. When the G^2 transition is between a clothoid and a line, we add the full $c_2 l^2/4$ term to the clothoid's τ', rather than half to each. When the G^2 transition is between an arc and a clothoid, we assign twice the usual c_2 to the arc (because a change in curvature affects the entire arc) and, if the other end of the arc is not on a corner or endpoint, twice the usual c_2 to the clothoid (to penalize the reduction in degrees of freedom). To save time, we do not construct edges for which both curves' τ' is higher than ten pixels (twice the original τ cutoff).

3.5. Shortest Path

We now need to find the shortest path (or shortest cycle, if the curve is closed) in the graph, but there are two difficulties. The first is that the edge weights we computed are just estimates — we would like a more accurate verification that the transition is feasible before we commit to it. The second difficulty is that while the shortest path algorithm runs in $O(E)$ time (our graph is acyclic), the best shortest cycle algorithm known takes $\tilde{O}(VE)$ time [Dem09], which is prohibitively expensive.

To deal with the first difficulty, after we find a shortest path, we verify every edge of the path by enforcing the continuity constraints and optimizing the fit, effectively executing the final merge step (as described in Section 3.6) for just the two curves. For each curve, we compute the adjusted error (2). To estimate the effect on the other transitions of the two curves, we add how far the curve moves as a result of enforcing the continuity constraints to the adjusted error for that curve. If the new error is greater than predicted, we increase the cost of the edge by the difference. We then rerun the shortest path algorithm, until no adjustments need to be made. The shortest path may also produce a configuration that we need to avoid: a clothoid with G^2 transitions to lines on both sides, forcing the clothoid to also be a line. Although we could avoid such paths by constructing two graph nodes for each clothoid (one that allows G^2 transitions from lines and one that allows transitions to lines), it would hurt performance. Instead, if the found shortest path contains such a configuration, we simply delete the more costly of the two transitions from the graph.

We can obtain a significant speedup by taking advantage of the fact that the graph changes little and edge costs only grow as a result of verifying the candidate paths as described above. Initially, we don't just compute the shortest path from the source to the target, but rather the distance from every node in the graph to the target (this takes the same amount of time). When computing subsequent shortest paths, we use the computed (but underestimated, as a result of edge weight increases) distance to the target as an A^* heuristic [HNR68]. The exact path length to the target is the best possible A^* heuristic for a graph because it results in A^* taking the shortest path directly without exploring the rest of the graph. In our case, the increasing edge costs make the heuristic suboptimal, but it is nevertheless very good, admissible, and consistent. Every few runs of A^*, we run the full shortest-path to recompute the distances to the target node in order to improve the heuristic.

Looking for the optimal cycle for closed curves is prohibitively expensive, so we use a heuristic. We assume that the best sequence of primitive curves approximating one part of the closed curve is not too dependent on the sequence of primitive curves approximating a far-away part. We therefore start with an arbitrary node and find the shortest path from the node, back to itself, using the algorithm described above. We then take the node closest to the middle of the resulting path and find the shortest path from that node to itself. We repeat this once more, as subsequent iterations did not improve the result further in our experiments.

3.6. Merging

We now have a sequence of curve primitives and continuity constraints between them, but the primitives do not satisfy the constraints. To obtain the final curve, we solve for the parameters of all primitives simultaneously, by minimizing the sum of squared distances from all samples to their curve primitive(s) subject to the continuity constraints. We use the SNOPT [GMS02] nonlinear optimization package. To construct the initial guess, the separate primitive curves are a good starting point, but they overlap with each other at G^1 and G^2 transitions. We therefore trim them by projecting the middle user sample onto the curves for G^2 transitions, or the midpoint between two user samples for G^1 transitions. This brings the initial guess closer to satisfying the G^0 constraints.

The precise objective function is defined as follows: let $\mathbf{p}_1, \ldots, \mathbf{p}_n$ be the user samples. As before, let $w_i = \|\mathbf{p}_{i-1} - \mathbf{p}_i\| + \|\mathbf{p}_{i+1} - \mathbf{p}_i\|$. For each point $\mathbf{p}_i$ that belongs to the interior of a curve C, we add $w_i d_C(\mathbf{p}_i)^2$ to the objective function. For endpoints and G^0 transitions, we add w_i times the squared distance to the actual point. The final curve minimizes this objective and exactly satisfies continuity and curvature sign constraints.

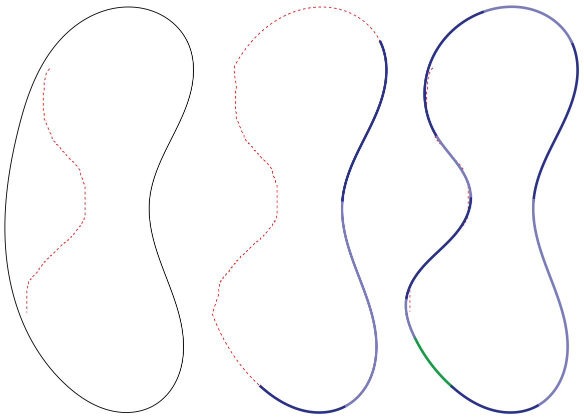
Figure 8: *Left to right: an oversketch stroke is drawn; the stroke is integrated into a portion of the curve; our method fits a clothoid spline to the integrated stroke, subject to the continuity constraints at the endpoints.*

3.7. Oversketching

While there are existing techniques for editing the resulting curve by dragging [Lev09], a common alternative method for editing freehand curves is oversketching. Supporting oversketching is relatively simple in our framework. We start by projecting the start and end points of the oversketch to the curve to identify the region of the curve that the user intends to replace (Figure 8). When this is ambiguous, we assume that the user is replacing the shorter of two possibilities. If only one endpoint is close to the curve and the original curve is not closed, we infer that the user wants to replace the original curve up to an endpoint.

To avoid discontinuities at the start and end of the oversketch, we move the endpoints of the new sketch onto the original curve and linearly attenuate this translation over a small region on the new sketch. To maintain continuity, we expand the region on the original curve that will eventually be replaced by at least 20 pixels on each side, and up to a primitive transition (or curve endpoint). We then densely sample the expanded regions on the original curve, combine these with the new sketch samples, and feed this to our algorithm as a new stroke (Fig. 8, middle).

The main algorithm is modified slightly for oversketching. The most important difference is that the start and/or end curves are the existing curve(s) adjacent to the region we are replacing. We use them as the sole source/sink nodes in the graph, and keep them fixed for transition validation and the merging step. We also adjust resampling to sample more densely in the transition region, to ensure that a smooth transition between the oversketch and the rest of the curve is possible. Finally, we assume that the user is more precise in oversketching than in the original sketch and increase the error cost accordingly.

3.8. Inflections

One criterion of curve fairness is that unnecessary inflection points should be avoided. To incorporate a user-set penalty for inflections, we keep track of inflections from the primitive fitting stage on, and account for inflections in the shortest path stage in a way that correctly predicts the inflections that the output curve will have. Simply computing the sign of curvature at the endpoints of the primitives is insufficient because the curvature may be zero. In particular, if the path contains a G^2 clothoid-line-clothoid sequence, it needs to be penalized for an inflection precisely if the clothoids have different signs of curvature.

We resolve this problem by disambiguating zero curvature. We store a "logical" sign of curvature, i.e., $+$ or $-$, at each endpoint. For arcs and clothoids, this is simply the sign of curvature of the primitive at the endpoint. For each line primitive, we generate two graph nodes, one with both ends marked with positive logical curvature, and one with both ends marked with negative logical curvature. When constructing the graph, we use the stored logical sign of curvature to determine costs. If the curvature signs on the endpoints of a clothoid are different, the inflection penalty is added to the cost of the clothoid. If the curvature signs are different between two curve endpoints joined by a G^2 transition, we add the inflection penalty to the corresponding edge. This correctly handles the case when the geometric curvature of the primitive is zero. In particular, a clothoid-line-clothoid sequence where the curvature remains nonnegative (resp. non-positive) is not counted as an inflection.

We further avoid inflections as follows. When a clothoid primitive has different curvature signs at the ends, sometimes a clothoid *without* an inflection can be almost as good a fit. Therefore, whenever a clothoid fit results in an inflection, we fit two additional clothoids to the same set of samples, one with both ends constrained to nonpositive curvature, and with both ends constrained to nonnegative curvature.

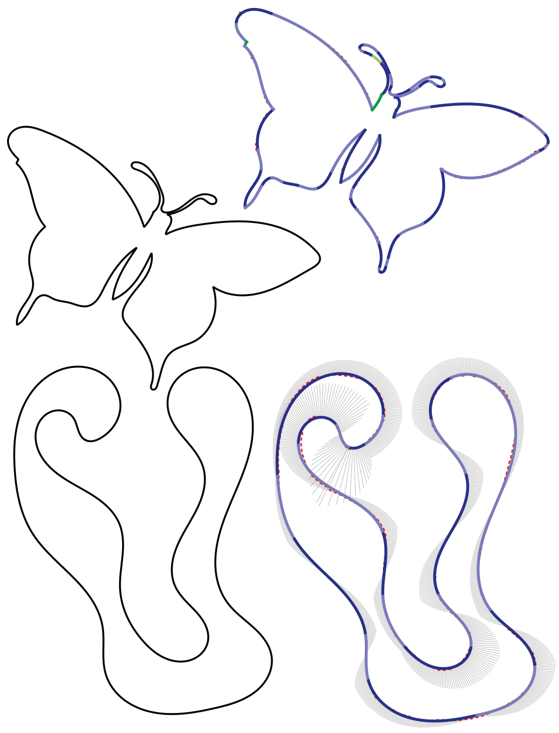
Figure 9: *Two curves produced by our algorithm. The right column shows the individual primitives and, by overlaying on the user sketch, shows the fidelity of the results.*

Because the final merge step can flip the sign of curvature, we must enforce that the output curve has no inflections except those that are accounted for by the shortest path. This can be done by constraining all curvatures to have their original signs, but that is too restrictive: for example, if the shortest path has allowed an inflection at the G^2 transition between two clothoids, there is no reason not to allow the inflection point to be in the interior of one of the clothoids instead. We lift the curvature sign constraint for arcs, for clothoids that already have an inflection, and for clothoid endpoints that are on a G^2 transition to a curve endpoint with the opposite logical curvature sign. Because the number of inflections in a G^2 chain of curve primitives is at most the number of clothoids in the chain, lifting the above restrictions cannot increase the number of inflections in the final result. We enforce the curvature sign restrictions both in the transition verification and the final merge.

Curve Name	Preprocess	Fit	Make Graph	Find Path	Merge	Total	Samples	Nodes	Edges
Squiggle	0.09	0.19	0.30	0.33	0.11	1.02	95	5,148	87,816
Squiggle G^1	0.08	0.02	0.11	0.14	0.06	0.41	95	3,026	36,192
Highheel	0.22	0.45	0.72	0.47	0.17	2.03	135	11,226	288,998
Closed Squiggle	0.14	0.33	0.34	0.42	0.36	1.59	128	6,624	105,544
Closed Squiggle G^1	0.14	0.02	0.09	0.28	0.13	0.66	128	3,574	36,374
Butterfly	0.33	0.58	0.81	1.56	1.53	4.81	268	16,186	304,753
Hello	0.09	0.42	0.53	0.52	0.76	2.32	189	10,674	181,021

Table 1: *Timing results (seconds), number of samples, and graph size for the curves in Figures 10, 9, and 4. The G^1 rows report the results for computing arc, rather than clothoid splines. Preprocessing comprises corner detection and resampling.*

4. Results

We evaluate the quality of the output spline by how smooth it appears (*fairness*) and how closely it approximates the user sketch (*fidelity*). These two goals are in conflict: a spline with nearly perfect fidelity will have all of the noise in the user input and will not be fair. The sketch tools with which we experimented, including our own, have controls for balancing fidelity against fairness. For each curve, we manually tuned the controls of the tools to try to match the point on the tradeoff curve. For our method, the parameter we tuned was the error cost.

We compared our method to McCrae and Singh's method [MS08] using their publicly available code. No other method we know of generates a clothoid spline from a freehand sketch. We also compared the results to the output of the pencil tool of Adobe Illustrator$^{\text{TM}}$ 14.0 and the freely available Inkscape software. Finally, we ran a nonlinear optimization of the sum of Moreton's MVC functional [Mor92] with the squared distance to the user sketch (the relative weights of the two terms providing the fidelity-fairness tradeoff). A few comparisons are given in Figure 10, and Figure 9 shows two more curves sketched in our system. The supplemental pages provide a more complete comparison for the various methods and also demonstrate the arc splines our method produces.

The examples demonstrate that our method is able to handle very complex curves. Compared to other methods, our method typically produces more accurate and fairer curves. The shortest path optimization results in an economical use of primitives — this is especially obvious in the arc-and-line splines, which do not have to enforce G^2 transitions.

The various algorithms tend to make several different kinds of errors. The main difficulty all of the algorithms have (including ours, to a lesser degree) is that the fairness-fidelity tradeoff is resolved differently in different parts of the curve. In other words, some part of the curve retains the noise of the user's stroke, while another part smooths out intentional sketch features as noise. The MVC fitting method is the worst in this regard: high-curvature regions (like the top of the shank and the toe on the high-heel shoe) get smoothed out, leading to poor fidelity and spurious inflections nearby, while low curvature regions don't get faired sufficiently. Also, corner detection is not completely robust in any of the algorithms, leading to both false positives and false negatives. In our method, inflection avoidance sometimes fails to prevent inflections and sometimes smooths out meaningful features. No other method explicitly avoids inflections.

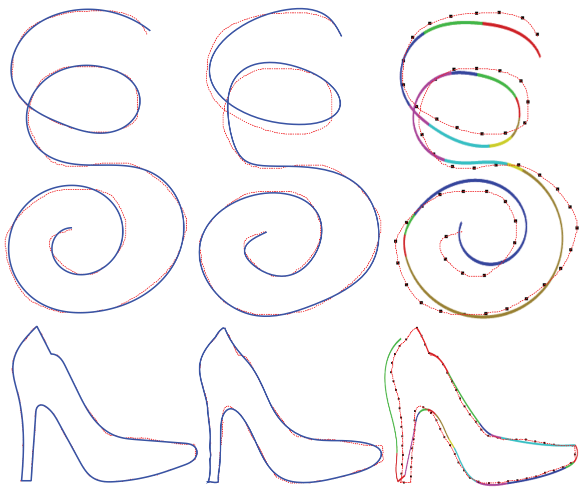
Figure 10: *Left to right: our result, Adobe Illustrator result, McCrae and Singh result [MS08]. For these examples, our method produces a curve that is both a better fit and more fair.*

Performance Constructing and searching a large graph and doing nonlinear optimization makes our algorithm significantly slower than most existing curve sketching methods. However, on a modern computer, it is still fast enough for a comfortable interactive experience. In addition, our implementation is single-threaded, as is SNOPT, which leaves many opportunities for optimization, since most of the time-consuming operations are parallelizable. In Table 1, we report the timings on a 2.66GHz Intel Core i7-920.

5. Conclusions

Limitations The proposed method produces higher quality curves from hand-drawn sketches than previously possible, but it does so at the expense of both complexity and speed. While our method is slower than existing ones, we do not believe that the performance is prohibitive, and there is room for additional optimizations. The complexity is significant: although the underlying idea of casting curve fitting as a shortest path problem is clean and simple, making it work requires substantial engineering (our prototype implementation is about 4,000 lines of C++, not including linear algebra and GUI libraries) and introduces a reliance on nonlinear optimization. Additionally, there are many "magic constants" that affect the algorithm's performance. However, because they usually have intuitive geometric meanings, we did not find it difficult to tune them. A proper model of the noise in user input may provide a principled way of obtaining some of these values. The nonlinear optimization works reliably the vast majority of the time because we start with a very good initial guess, but we have observed a few failures. Oversketching provides a simple way to work around them when they happen.

Conclusions Clothoid splines are gaining popularity as a stroke representation due to their inherent fairness. While they are mathematically and algorithmically more complex than polynomial splines, we strongly believe their inherent high quality outweighs the costs. This paper addresses key questions in fitting clothoid splines to user input, enabling faithful and fair representation of hand-drawn strokes. Specifically, we cast the segmentation of the input stroke into primitives as a graph problem, and fit clothoid segments to the input directly without resorting to curvature profile space and integration. Combined with oversketching, our method allows the specification and editing of fair but complex curves. Our results demonstrate clear advantages in comparison to both previous academic work and commercial tools, both in terms of fidelity and fairness.

6. Acknowledgments

We thank Tony DeRose and Mark Meyer for early discussions. Thanks to Saku Lehtinen and Daniel Vlasic for helpful feedback. Thanks to Emily Whiting for drawing some of our examples.

References

[BBS08] BAE S., BALAKRISHNAN R., SINGH K.: ILoveSketch: as-natural-as-possible sketching system for creating 3D curve models. In *Proc. UIST* (2008), pp. 151–160. 1

[Dem09] DEMAINE E. D.:. personal communication, 2009. 7

[DRS08] DRYSDALE R. S., ROTE G., STURM A.: Approximation of an open polygonal curve with a minimum number of circular arcs and biarcs. *Computational Geometry 41*, 1-2 (2008), 31 – 47. 2

[FRSW87] FARIN G., REIN G., SAPIDIS N., WORSEY A. J.: Fairing cubic b-spline curves. *Computer Aided Geometric Design 4*, 1-2 (1987), 91 – 103. Topics in CAGD. 2

[GMS02] GILL P., MURRAY W., SAUNDERS M.: SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Journal on Optimization 12*, 4 (2002), 979–1006. 7

[HE05] HELD M., EIBL J.: Biarc approximation of polygons within asymmetric tolerance bands. *Computer-Aided Design 37*, 4 (2005), 357 – 371. 2

[HNR68] HART P., NILSSON N., RAPHAEL B.: A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern. 4*, 2 (July 1968), 100–107. 7

[IMT99] IGARASHI T., MATSUOKA S., TANAKA H.: Teddy: a sketching interface for 3D freeform design. In *Proc. ACM SIGGRAPH 99* (1999), pp. 409–416. 1

[Lev09] LEVIEN R. L.: *From Spiral to Spline: Optimal Techniques in Interactive Curve Design*. PhD thesis, University of California, Berkeley, 2009. 1, 2, 7

[Mor92] MORETON H. P.: *Minimum curvature variation curves, networks, and surfaces for fair free-form shape design*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1992. 2, 9

[MS08] MCCRAE J., SINGH K.: Sketching piecewise clothoid curves. In *Sketch-Based Interfaces and Modeling* (2008). 2, 5, 9

[MS09] MCCRAE J., SINGH K.: Sketch-based path design. In *Proc. Graphics Interface* (2009), pp. 95–102. 2

[MT91] MEEK D. S., THOMAS R. S. D.: A guided clothoid spline. *Computer Aided Geometric Design 8*, 2 (1991), 163 – 174. 1, 2

[MW89] MEEK D. S., WALTON D. J.: The use of cornu spirals in drawing planar curves of controlled curvature. *J Comput. Appl. Math. 25*, 1 (1989), 69 – 78. 2

[MW92] MEEK D. S., WALTON D. J.: Approximation of discrete data by G1 arc splines. *Computer-Aided Design 24*, 6 (1992), 301 – 306. 2

[NMK72] NUTBOURNE A., MCLELLAN P., KENSIT R.: Curvature profiles for plane curves. *Computer-Aided Design 4*, 4 (1972), 176 – 184. 2

[PN77] PAL T., NUTBOURNE A.: Two-dimensional curve synthesis using linear curvature elements. *Computer-Aided Design 9*, 2 (1977), 121 – 134. 2

[Pra87] PRATT V.: Direct least-squares fitting of algebraic surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 87)* (July 1987), pp. 145–152. 5

[Sch78] SCHECHTER A.: Synthesis of 2d curves by blending piecewise linear curvature profiles. *Computer-Aided Design 10*, 1 (1978), 8 – 18. 2

[Sez01] SEZGIN T.: *Feature point detection and curve approximation for early processing of free-hand sketches*. Master's thesis, Massachusetts Institute of Technology, 2001. 4

[SF90] SAPIDIS N., FARIN G.: Automatic fairing algorithm for b-spline curves. *Computer-Aided Design 22*, 2 (1990), 121 – 129. 2

[SK00] SCHNEIDER R., KOBBELT L.: Discrete fairing of curves and surfaces based on linear curvature distribution. In *In Curve and Surface Design: Saint-Malo* (2000), University Press, pp. 371–380. 2

[SSFS06] SCHREINER J., SCHEIDEGGER C., FLEISHMAN S., SILVA C.: Direct (re) meshing for efficient surface processing. *Computer Graphics Forum 25*, 3 (2006), 527–536. 4