Advanced Drawing Beautification with ShipShape

Jakub Fišer[a], Paul Asente[b], Stephen Schiller[b], Daniel Sýkora[a]

[a] *Czech Technical University in Prague, FEE*
[b] *Adobe Research*

Abstract

Sketching is one of the simplest ways to visualize ideas. Its key advantage is its easy availability and accessibility, as it require the user to have neither deep knowledge of a particular drawing program nor any advanced drawing skills. In practice, however, all these skills become necessary to improve the visual fidelity of the resulting drawing. In this paper, we present ShipShape—a general beautification assistant that allows users to maintain the simplicity and speed of freehand sketching while still taking into account implicit geometric relations to automatically rectify the output image. In contrast to previous approaches ShipShape works with general Bézier curves, enables undo/redo operations, is scale independent, and is fully integrated into Adobe Illustrator. We show various results to demonstrate the capabilities of the proposed method (Figure 1).

Keywords: Drawing system, Input beautification, Vector graphics, Visual feedback

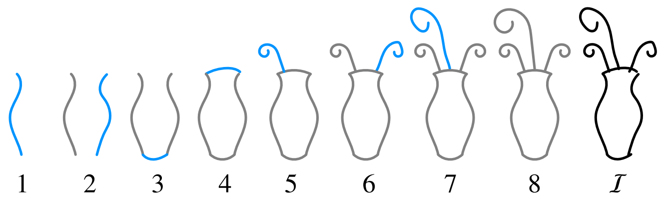

Figure 2: Incremental beautification workflow. Every newly drawn stroke (blue) is beautified using previously created data (gray). The first stroke is left unchanged. As the drawing continues, more suitable geometric constraints emerge and are applied, such as path identity (2,6,7), reflection (2,6) or arc fitting (3,4). For comparison with the final beautified output (8), $\mathcal{I}$ shows the original input strokes.

1. Introduction

Sketching with a mouse, tablet, or touch screen is an easy and understandable way to create digital content, as it closely mimics its real-world counterpart, pen and paper. Its low demands make it widely accessible to novices and inexperienced users. However, its imprecision means that it is usually only used as a preliminary draft or a concept sketch. Making a more polished drawing requires significantly more time and experience with the drawing application being used. Furthermore, when working with drawing or sketching software, users are often forced to switch between different drawing modes or tools or to memorize cumbersome shortcut combinations.

While we do not question the necessity or usefulness of complex tools to achieve non-trivial results, we argue that for certain scenarios, such as geometric diagram design or logo study creation, the *interactive beautification* [1] approach is more beneficial. Such workflows retain the intuitiveness of freehand input while benefiting from an underlying algorithm that automatically rectifies strokes based upon their geometric relations, giving them more formal appearance. With the quickly growing popularity of touch-enabled devices, the applicability of this approach expands greatly. However, whatever the potential of automatic beautification in a more general sketching context, most of the existing applications focus on highly structured drawings like technical sketches.

One of the biggest challenges in drawing beautification is resolving ambiguity of the user input, since the intention and its execution are often considerably dissimilar. Additionally, this issue becomes progressively more complex as the number of primitives present in the drawing increases.

In this paper, we present a system for beautifying freehand sketches that provides multiple suggestions in spirit of Igarashi et al. [1]. Strokes are processed incrementally (see Figure 2) to prevent the combinatorial explosion of possible outputs. Unlike previous work, our approach supports polycurves composed of general cubic Bézier curves in addition to simple line segments and arcs. The system is scale-independent, and can easily be extended by new operations and inferred geometric constraints that are quickly evaluated and applied. The algorithm was integrated into Adobe Illustrator, including undo/redo capability. We present various examples to demonstrate its practical usability.

2. Related Work

The need to create diagrams and technical drawings that satisfy various geometric constraints led to the development of complex design tools such as CAD systems. However, these systems' complexity often limits their intuitiveness. Pavlidis and Van Wyk [2] were one of the first to try to alleviate this conflict by proposing a method for basic rectification of simple rectangular diagrams and flowcharts. However, their process became ambiguous and prone to errors when more complex

Figure 1: Examples of drawings created using ShipShape. The final drawings (black) were created from the imprecise user input (gray) by beautifying one stroke at a time, using geometric properties such as symmetry and path identity. See Figure 17 for more results.

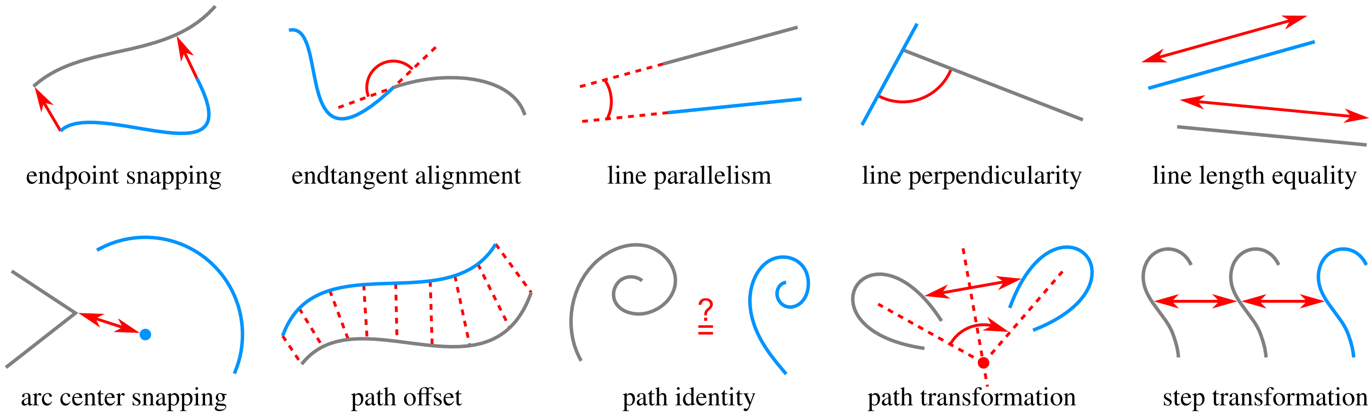

Figure 3: Supported geometric rules and transformations in our framework. The blue paths represent the data being beautified, while gray paths are data already processed. For more detailed description of the criteria used to evaluate these constraints, see Section 3.1.

drawings were considered, since the method needed to drop many constraints to keep the solution tractable.

To alleviate this limitation, Igarashi et al. [1] proposed an interactive beautification system in which the user added strokes one by one and the system improved the solution incrementally while keeping the previously processed drawing unchanged. This solution kept the problem tractable even for very complex drawings. Moreover, the system also presented several beautified suggestions and let the user pick the final one. This brought more user control to the whole beautification process. Following a similar principle, other researchers developed systems for more specific scenarios such as the interactive creation of 3D drawings [3], block diagrams [4, 5], forms [6], and mathematical equations [7].

However, a common limitation of the approaches mentioned above is that they treat the image as a set of line segments. To alleviate this drawback Paulson and Hammond [8] proposed a system called *PaleoSketch* that fit the user input to one of eight predefined geometric shapes, such as line, spiral or helix. In a similar vein, Murugappan et al. [9] and Cheema et al. [10] allowed line segments, circles and arcs.

Related to drawing beautification, there are also approaches to beautify curves independently, without considering more complex geometric relationships. Those approaches are orthogonal to our pipeline. They use either geometric curve fitting [11, 12] or some example-based strategy [13, 14]. Additionally, advanced methods for vectorizing and refining raster inputs have been proposed [15, 16], which enable users to convert bitmap images into high quality vector output. However these do not exploit inter-stroke relationships. In our case we assume that the built-in curve beautification mechanism of Adobe Illustrator preprocesses the user's rough input strokes into smooth, fair paths.

This paper extends our previous work [17]. In Section 3.1 we discuss improvements to the arc and circle center rules, and introduce a generalized transformation adjustment framework. Section 3.4 describes a new method for curve alignment, and Section 3.5 describes the transformation adjustment mechanism in detail. Finally, Section 4 describes a new framework for handling curves with corners.

3. Our Approach

A key motivation for our system is wanting to work with arbitrarily curved paths. This capability was not available in previous beautification systems. Although some can recognize a variety of curves including spirals and general 5th degree polynomials (*PaleoSketch* [8]), they recognize them only in isolation and do not allow to take other existing paths into consideration, which is important for interactive design.

Systems like that of Igarashi et al. [1] generate a set of potential constraints and then produce suggestions by satisfying subsets of these. A key challenge that prohibits simply generalizing these systems to support general curved paths is the number of degrees of freedom, which boosts the number of potential constraints that need to be evaluated. Moreover, unlike line or arc segments, many of a general path's properties, for example the exact coordinates of a point joining two smooth curves, do not have any meaning to the user. It would not be helpful to add constraints for this point. Finally, satisfying constraints on a subset of the defining properties might distort the path into something that barely resembles the original. Supporting generalized paths requires a different approach.

Our system is based on an extensible set of self-contained geometric rules, each built as a black box and independent of other rules. Every rule represents a single geometric property, such as having an endpoint snapped or being a reflected version of an existing path. The input to each rule is an input path consisting of an end-to-end connected series of Bézier curves, and the set of existing, resolved paths. The black box evaluates the likelihood that the path conforms to the geometric property, considering the resolved paths, and outputs zero or more modified versions of the path. Each modified version gets a score, representing the likelihood that the modification is correct.

For example, the same-line-length rule would, for input that is a line segment, create output versions that are the same lengths as existing line segments, along with scores that indicate how close the segment's initial length was to the modified length. Each rule also has some threshold that determines that the score for a modification is too low, and in that case it does not output the path.

The rules also mark properties of the path that have become fixed and therefore can no longer be modified by future rules. For example, the endpoint-snapping rule marks one or both endpoint coordinates of a path as fixed. The same-line-length and parallel-line rules do not attempt to modify a segment with two fixed endpoints.

Since the rules do not depend on each other, it is easy to add new rules to support additional geometric traits. Figure 3 shows an illustrated list of rules supported in our system.

Chaining the rules can lead to complex modifications of the input stroke and is at the core of our framework. We treat the rule application as branching in a directed rooted tree of paths, where the root node corresponds to the unmodified input path. Each branch of the tree corresponds to a unique application of one rule and the branch is given a weight corresponding to the rule's score.

To find suitable transformations for the user input, we traverse down to the leaf nodes (see Figure 4).

Formally, given a node n^i with Bézier path p^i, the set of resolved paths S, and the set of all rules $r_j \in R$, we compute an output set $P^i = \left\{r_j\left(p^i, S\right)\right\}$. We then create a child node n^i_j for each $p^i_j \in P^i$. If P^i is empty, n^i is a leaf node.

Since we need to compare scores among different rules, likelihoods are always normalized into the interval $[0, 1]$. If a rule generates any modified paths, it also generates a copy of the unmodified path, indicating the suggestion that the rule did not apply. The likelihood for the unmodified path is 1 minus the maximum likelihood of any modified path.

We can then use all scores from the nodes we visited while descending into a particular leaf node n to calculate the overall likelihood score for the chained transformation as

$$\overline{\mathcal{L}_i} = 1 - \prod_{k=1}^{d-1}\left(1 - \mathcal{L}\left(r_j\left(a^k, S\right)\right)\right) \qquad (1)$$

where d is the depth of n in the tree, a^k is the kth ancestor of n, and $\mathcal{L}\left(r_j\left(a^k, S\right)\right)$ denotes the likelihood score from applying rule r_j to node a^k.

We expand the search tree in a *best-first search* manner, where the order of visiting the child nodes is determined by the overall score $\overline{\mathcal{L}}$ of the node's path. While traversing the tree, we construct a suggestion set Q of leaf nodes, which is initially empty and gets filled as the leaf nodes are encountered in the traversal. Once not empty, Q helps prune the search. Before we expand a particular subtree, we compare the geometric properties of its root with properties of each path $q \in Q$. If all tested properties are found in some path q, the whole subtree can be omitted from further processing (see Figure 5).

Furthermore, to keep the user from having to go through too many suggestions, we limit the size of Q. Since we traverse the graph in a best-first manner, we stop the search after finding some number of unique leaf nodes (10 in our implementation).

3.1. Supported Rules and Operations

Geometric transformations in our framework are evaluated by testing various properties of the new path and the set of pre-

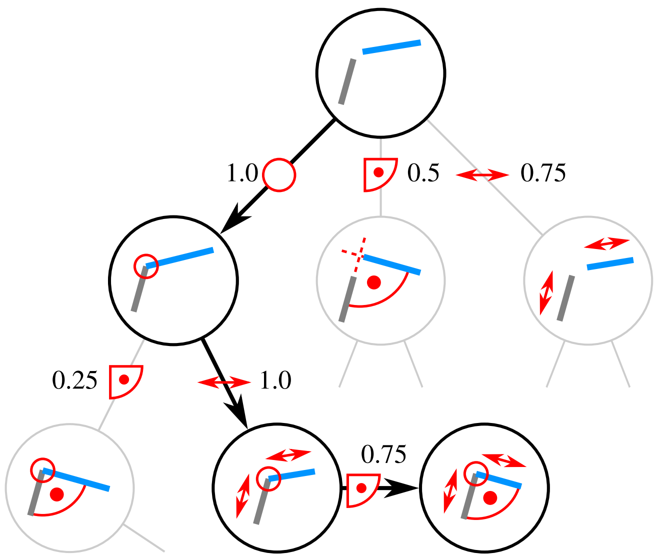

Figure 4: Successive rule evaluation and application. In this example, the evaluation engine consists of three geometric rules—endpoint snapping, perpendicularity, and length equality. The old data (gray path) is fixed in the canvas. When a new path (blue) is added, it becomes the root node of the evaluation graph and the expansion begins by testing all rules on it. A likelihood score is calculated for each rule application and the tree is expanded using a best-first search scheme, until leaf nodes are reached. Due to the significant redundancy in the search space, many leaf nodes will contain duplicate suggestions. Therefore, we prune the graph during the expansion step using the information from already reached leaf nodes (see Section 3 and Figure 5 for more information).

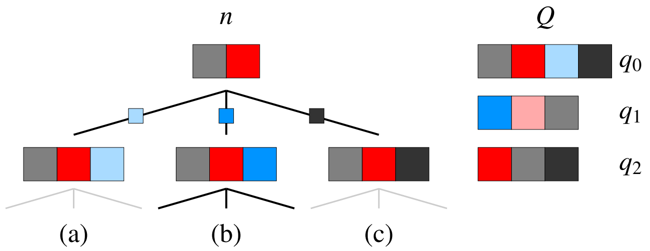

Figure 5: Search graph pruning. The rules are represented by colored boxes with hue being distinct rules and lightness their unique applications (e.g., if red color represents endpoint snapping, then different shades of red correspond to snapping to different positions). An inner node n has been expanded into three branches (a,b,c). Before further traversal, all subtrees stemming from the child nodes of n are tested against suggestions $q \in Q$. Here, branches (a) and (c) are fully contained in q_0 and q_2 respectively and thus only branch (b) is evaluated further.

viously drawn and processed paths. While tests of some properties are simple, others, such as path matching, require more complex processing. We first summarize rules supported by our system (illustrated in Figure 3), and then we present some additional implementation issues including a more detailed description for non-trivial rules.

Line Detection We estimate a path's deviation from straightness by measuring the ratio between its length and the distance between its endpoints, as in QuickDraw [10].

Arc Detection We sample the input path and perform a least-squares circle fit on the samples to obtain center and radius parameter values. To determine the angular span value, we project the samples onto the circle fit. The arc is then sampled again and we evaluate the discrete Fréchet distance [18] between the arc samples and the samples of the input path. When the span is close to 2π or the path is closed, we replace it with a full circle.

Endpoint Snapping We look at the distance between each of the path endpoints and resolved endpoints. Additionally, we also try snapping to inner parts of the resolved paths. Specialized tests based on the properties of line segments and circular arcs lower the computational complexity of this operation. Note that we do not join the two end-to-end-snapped paths. This can cause unpleasant artifacts where they meet, but the effect of a join can be mimicked by using round end caps on the strokes.

End Tangent Alignment If the path endpoint is snapped, we measure the angle between its tangent and the tangent of the point it is attached to.

Line Parallelism and Perpendicularity We compare the angle between two line segment paths with the angle needed to satisfy the parallelism or perpendicularity constraint. Additionally, we also take the distance between the line segments into account to slightly increase the priority of nearby paths. To evaluate these properties on the input non-rectified paths, we use their line segments approximations, i.e., line segments connecting their two endpoints.

Line Length Equality We evaluate the ratio of length of both tested line segments. As in previous case, we incorporate their mutual distance in the final likelihood computation.

Arc and Circle Center Snapping Similar to endpoint snapping, we evaluate the distance between the current arc center and potential ones, in this case endpoints of other paths, other centers, centers of rotations, and centers of regular polygons composed from series of line segments. However, as arcs with small angular span are noticeably harder to draw without a guide (see Figure 6a), the center of the initial arc fit might be located too far apart from the desired center point (Figure 6b) and therefore using fixed distance, when looking for potential center-snapping points, might not be sufficient. To address this issue, we adaptively change this distance to $\max(D, 2r(1-\theta/2\pi))$, where θ is the span of the tested arc, r is its radius and D is the standard search distance radius ($D = 30$ view-space pixels in our implementation).

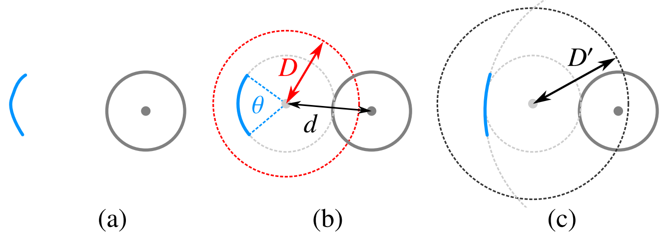

Figure 6: Adaptive arc/circle center-point-snap search distance refinement. Arc segments with small angular span are often drawn very imprecisely (a). When the engine fits an exact arc into such data, its center is often too far from the desired center point, as the distance d between them is bigger than the limit D under which the prospective center point positions are looked for (b). Adaptive expansion of the search radius D' increases the likelihood that even the imprecise input will give the user the expected (precise) output.

Path Identity To detect that two paths have similar shapes, we align them and compute their discrete Fréchet distance. More details are given in Section 3.4.

Transformation Adjustment For a tested path x and resolved reference path y of the "same shape" (determined by successful application of the path-identity rule) we perform a variety of modifications to the transformation to create symmetries, align paths, and equalize spacing. More details are given in Section 3.5.

Path Offset Offset paths generalize line parallelism. To detect them, we go along the tested path and measure its distance to the reference path. More details are given in Section 3.6.

3.2. View-Space Distances

Testing paths for different geometric properties ultimately requires measuring lengths and distances. While many path attributes can be compared using relative values, absolute values are still necessary, e.g., for snapping endpoints. Using absolute values, however, leads to unexpected behavior when the canvas is zoomed in and out. To eliminate this problem, we compute all distances in view-space pixels, making all distance tests magnification-independent.

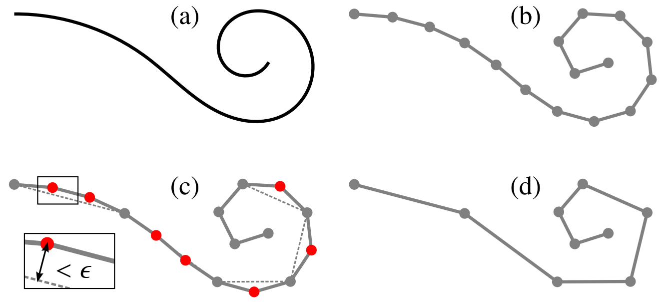

Figure 7: Path sample simplification. The original Bézier path (a) is equidistantly sampled, giving a polyline (b). The Ramer–Douglas–Peucker algorithm then recursively simplifies the polyline by omitting points closer than ϵ (c) to the current approximation, finally constructing simplified polyline (d).

3.3. Path Sampling

Working with cubic Bézier curves analytically is inconvenient and difficult. Many practical tasks, such as finding a path's length or the minimal distance between two paths, can only be solved using numerical approaches. Therefore, we perform all operations on sampled paths. Since the resolved paths do not change, we can precompute and store the samples for resolved paths, and sample only new paths. Furthermore, to reduce the memory requirement and computational complexity of different path comparisons, we simplify the sampling using the *Ramer–Douglas–Peucker* algorithm [19, 20]. For a polyline p, this finds a reduced version $p\prime$ with fewer points within given tolerance ϵ, i.e., all points of $p\prime$ lie within the distance ϵ of the original path (see Figure 7). Our implementation uses $\epsilon = 4$ view-space pixels at the time the path was drawn.

3.4. Path Matching

A key part of our contribution involves resolving higher-level geometric relations like path rotational and reflection symmetry. To identify these relations, we must first classify paths that are the "same shape"—paths that are different instances of the same "template".

To evaluate the similarity between two sampled paths p_a and p_b, we employ a discrete variant of *Fréchet distance* [18], a well-established similarity measure. Formally, it is defined as follows: Let (M, d) be a metric space and let the path be defined as a continuous mapping $f : [a, b] \rightarrow M$, where $a, b \in \mathbb{R}, a \leq b$. Given two paths $f : [a, b] \rightarrow M$ and $g : [a\prime, b\prime] \rightarrow M$, their Fréchet distance δ_F is defined as

$$\delta_F(f, g) = \inf_{\alpha,\beta} \max_{t\in[0,1]} d\left(f(\alpha(t)), g(\beta(t))\right), \quad (2)$$

where α (resp. β) is an arbitrary continuous non-decreasing function from $[0, 1]$ onto $[a, b]$ (resp. $[a\prime, b\prime]$). Intuitively, it is usually described using a leash metaphor: a man walks from the beginning to the end of one path while his dog on a leash walks from the beginning to the end of the other. They can vary their speeds but they cannot walk backwards. The Fréchet distance is the length of the shortest leash that can allow them to successfully traverse the paths.

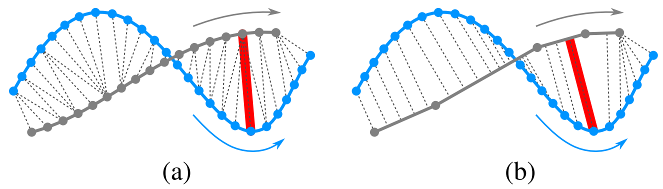

Figure 8: Discrete Fréchet distance. The minimum length of the line connecting ordered sets of point samples (a). Since we store the resolved paths in the simplified form, we compute the Fréchet distance between an ordered set of points and an ordered set of line segments (b) rather than between two point sets.

As outlined by Eiter and Mannila, this can be computed for two point sets using a dynamic programming approach. The extension to point and line-segment sets (Figure 8b) is then straightforward. However, the measure takes into account the absolute positions of the sample points, while we are interested in relative difference. Therefore, we have to adjust the alignment of the two tested paths. We then compute the discrete Fréchet distance between the aligned paths, divided by the length of the new path to obtain the relative similarity measure.

An affine similarity transform is a composition of a rotation, a uniform scale, and a translation. To align the paths, we find the affine similarity matrix that transforms the reference path to match the new path as closely as possible.

Assume the rotation angle is θ, the scale is s, and the translation is (tx, ty). Define $scos = s * cos\ \theta$ and $ssin = s * sin\ \theta$. The matrix is then

$$\begin{bmatrix} scos & -ssin & 0 \\ ssin & scos & 0 \\ tx & ty & 1 \end{bmatrix} \quad (3)$$

We compute the affine similarity transformation matrix M as follows. We first create two equal-length lists of points, each consisting of N equally-spaced samples from the reference and new paths. If $\{P_i\}$ are the points from the reference path and $\{Q_i\}$ the points from the new path, we find the M that minimizes the sum of the squared distances

$$E = \sum_{i=1}^{N} \|P_i * M - Q_i\|^2 \quad (4)$$

This is a quadratic function of $scos, ssin, tx$, and ty and can be solved as a least-squares problem over these four variables.

Before computing the Fréchet distance, we multiply the reference path samples by M. If the Fréchet distance indicates that the paths are sufficiently similar, we create a suggestion consisting of the reference path transformed by this same M.

A path that is a transformed copy of another path is permanently annotated as such, thereby allowing us to optimize path matching by only testing against a single instance of the path. For later processing, we also annotate the path with the transformation matrix.

If the drawing already contains multiple instances of a path, we consider it more likely that the user intended a new path to match. We therefore boost its score s by replacing it with $1 - (1 - s)^{ln\ i}$ where i is the number of existing instances.

Because the new path might be a reflected and/or reversed version of the reference path, we perform four tests between them to determine the correct match.

3.5. Transformation Adjustment

If the test path is a transformed version of a reference path, there are various tests we perform to adjust the transformation matrix to make the result more pleasing. We first begin by separating the matrix in Equation 3 into separate rotation, scale, and translation components as follows:

$$\begin{aligned} rotation &= atan2(ssin, scos) \\ scale &= \sqrt{scos^2 + ssin^2} \\ translation &= (tx, tx) \end{aligned} \tag{5}$$

The transformation can be adjusted in various ways, often generating multiple suggestions. Although we optimized path matching to only compare against one instance of a path that has multiple copies in the drawing, we test the transformation relative to each copy; see Figure 9a.

Rotation Snapping If the rotation component is close to an angle that is an integral divisor of 2π, it is snapped to being that angle (e.g., to 45 degrees; see Figure 10b4).

Scale Snapping If the scale component is close to an integer or to 0.5, it is snapped to being that exact scale.

Translation Snapping Translation snapping takes several forms:

- If the transformation contains a rotation component, we find the rotation center and compare it to existing points in the drawing. If it is sufficiently close we adjust the translation to place the center of rotation at that point.
- If the test path is a reflected version of the resolved path, we first compute the axis of reflection and reflect the resolved path across this axis. If the test path is sufficiently close to this reflected path, we adjust the translation to move it to that position.
- In other cases, we snap the x and y components of the translation to zero.

Step Transform Snapping Step transform snapping allows the user to create multiple, equally transformed copies of a path (see Figure 10b3). When we snap a path to an instance of a path, we store the relative transformation to that instance as the *step transform*. The step transform is the relative transform of the most highly-scoring suggestion. In Figure 9b, the existing drawing contains three resolved paths that are all the same shape. R was drawn first, and is the reference path. C is the first copy, and its step transform is the transformation from R to C. D is the second copy, and it was horizontally snapped to C. Because the transformation from C scored more highly (containing a snap) than the transformation from R, the step transform for D is the relative transform from C to D.

Step transform snapping compares the transformation from a path instance to the step transform for that instance. If the two transformations are similar, then a step-snapping suggestion is generated. In Figure 9c, the newly drawn path T is compared to all three existing instances R, C and D. The transformation M_{DT} from D to T is similar to the step transform of D. This generates a step-snapping suggestion to place T in the position that exactly matches the step transform; see Figure 9d.

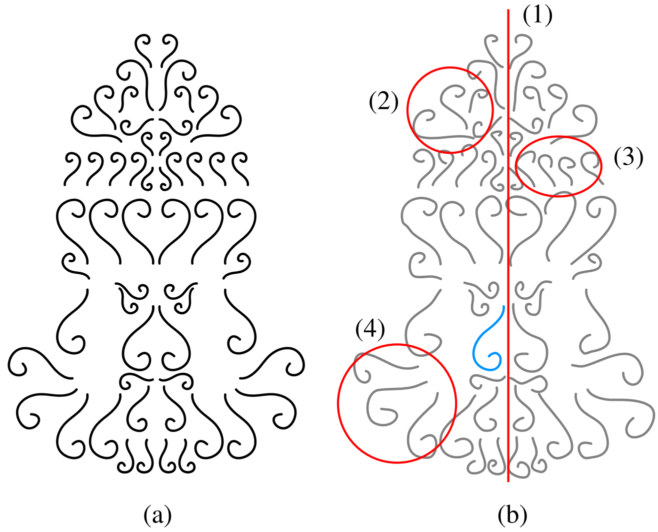

Figure 10: Practical application of transformation adjustment of the imprecise input (b) to obtain highly symmetrical output (a). We apply reflection axis (1), step transform (2,3) and rotation (2,4) snapping. Also note that the whole drawing is composed of strokes of the same shape.

Although this example only includes translation in the step transform, they are fully general, and can include rotation, scale, and reflection (see Figure 10b2).

Reflection Axis Snapping Users often want to reflect multiple paths against the same axis of reflection (for example, see the bear in Figure 1), or want to reflect a path across an existing line segment. To accommodate this, we collect all existing axes of reflection and line segments. If the new path is reflected, we compare its axis of reflection to these potential axes, and if it is close, we generate a suggestion to reflect across this axis (see Figure 10b1). Further, we strengthen the likelihood for an axis that has already been used multiple times by replacing the score s with $1 - (1 - s)^{ln\ i}$ where i is the number of times that axis has been used.

3.6. Offset Path Detection

Offset paths extend the concept of parallelism from line segments to paths. To detect them, we construct a normal line from each sample of the new path. If the line hits an existing reference path, we measure the distance between the sample point and the closest point on the reference. Note that we do not use

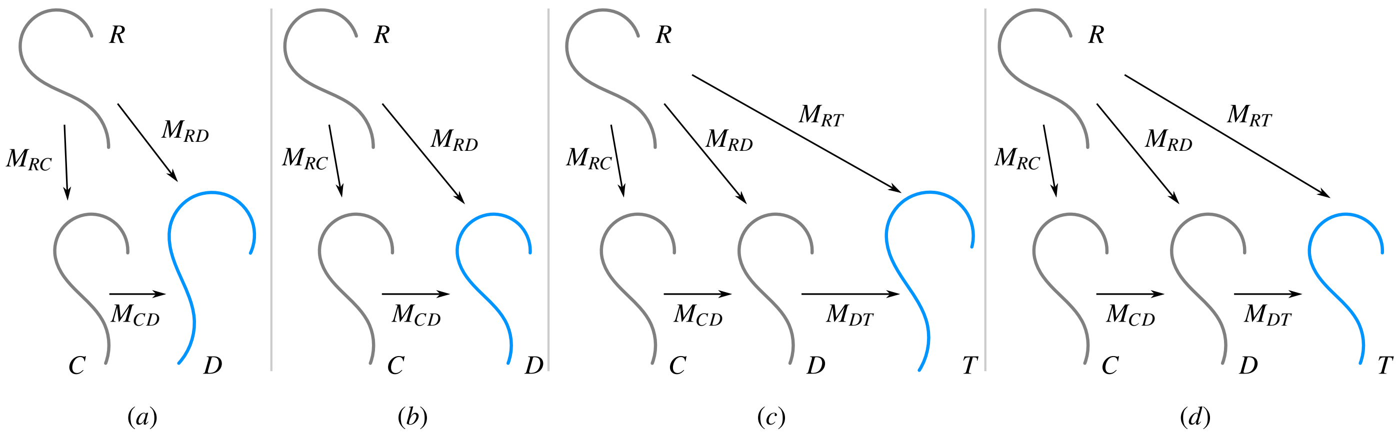

Figure 9: Transformation adjustment and transformation step snapping. The reference path R already has a copy C in the drawing, with M_{RC} being the transformation from R to C. D is the test path with M_{RD} being the transformation from R to D. Transformation adjustment considers both M_{RD} and the derived relative matrix M_{CD} that transforms C to D (a). The step transform for D is then M_{CD}, the relative transform from C (b). The relative transform for T relative to D is similar to the step transform for D (c). Applying M_{CD} to D generates a well-spaced suggestion (d).

the distance between the sample point and the line-path intersection, since this would require the user to draw the approximate offset path very precisely. We store the measured distance along with its sign, i.e., on which side of the new path the hit occurred. We then sort all the hit information according to the distance, creating a cumulative distribution function, and pick two values corresponding to $(50 \pm n)$-th percentiles (n being 25 in our implementation). By comparing the sign and distance values of these samples, we calculate the likelihood of the new path being an offset path of the reference path (see Figure 11). If the likelihood is high, we replace the new path with an offset version of the reference.

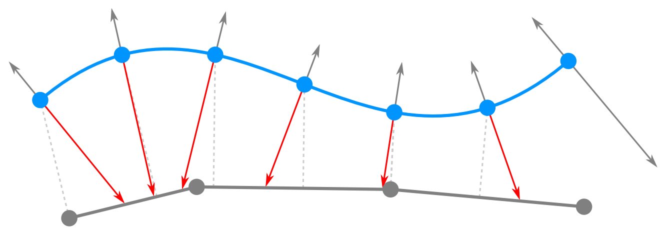

Figure 11: Offset path detection. A line is constructed from each point on the sampled path (blue circles) in the normal direction. If an existing reference path is hit (red rays), the minimal distance from the sample to the reference path is calculated (dashed lines) and used in offset-path-likelihood computation (see 3.6).

4. Multi-Segment Stroke Processing

The single stroke processing approach gives the user the opportunity to immediately see the results of the input being beautified. However, in certain cases, like drawing simple triangles or squares, this workflow can be tedious and decrease the overall fluency of the beautification pipeline. To this end, we introduce an additional step into our scheme that lets the evaluation engine process strokes with multiple segments. These segments are defined as parts of the unprocessed user input, split by corner features. Once divided, the evaluation engine can process the simple segments using the geometric rules introduced in Section 3.1.

4.1. Corner Detection

When the raw freehand input stroke is drawn by the user, it is converted to a sequence of cubic Bézier curves and passed to the beautification pipeline. The first step is to test it for the presence of corner points. Because the initial curve fitting is done by the host application (e.g., Adobe Illustrator), we cannot simply rely on the assumption that corners can only occur at the junction of two Bézier curves. For example, in Figure 12a, the apparent corner in the lower right is actually a small-radius curve. We initially sample the curves with a small step size (2 view-space pixels) and calculate the tangent vector at each sample point. Using a sliding window of three successive samples, we calculate the angular turn value at every sample position except the first and last. Local maxima in this turn sequence provide the places to break the original input sequence into segments. To handle outliers like the unwanted "hooks" at the ends, we discard segments whose length is small compared to the rest of the segments (less than 15% of the length-wise closest other segment).

4.2. Segment Processing

The segments of the complex user input can then be processed one at the time using the same approach used for the simple input described in Section 3. There are, however, important issues to address. Most notably, processing multi-segment input involves automatic selection of intermediate outputs, which would otherwise be done by the user. As the number of potential outputs rises exponentially, we cannot explore the whole search space. Therefore, we perform two reduction steps to make the evaluation of complex inputs computationally feasible within real-time-to-interactive response time. First, we limit the number of unique suggestions for each segment to 3 (whereas

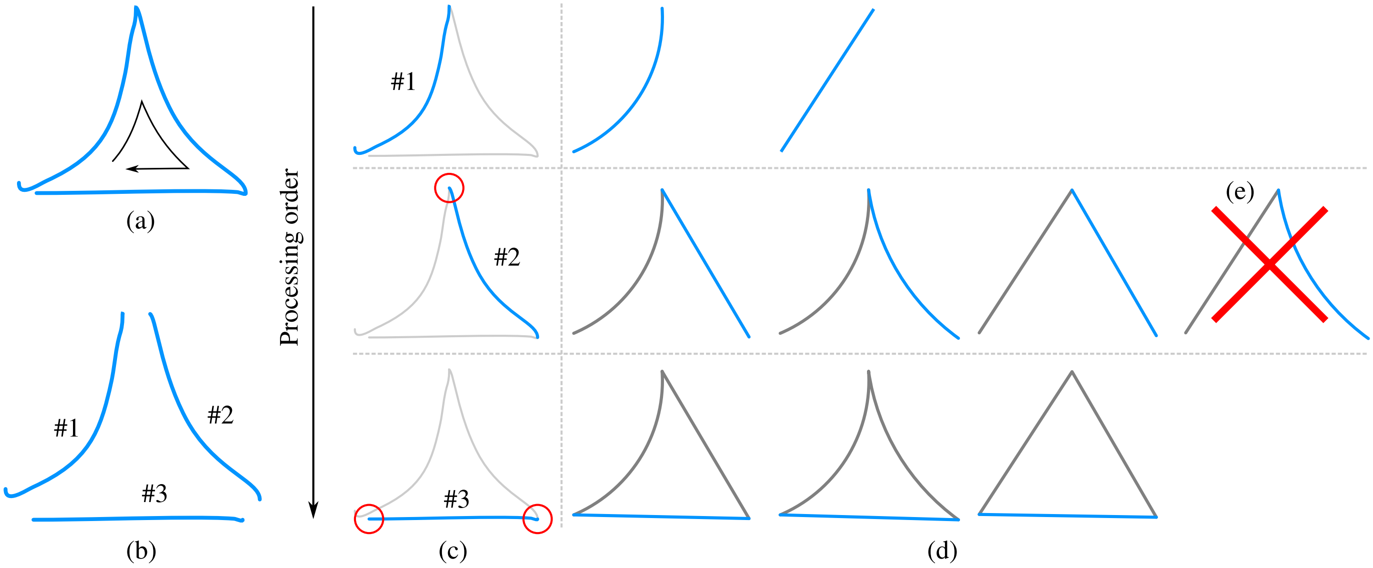

Figure 12: Multi-segment stroke processing pipeline. When a complex stroke is drawn (a), it is tested for the presence of corner points. If no corner points are found, the processing continues as described in Section 3. If one or more corner points are detected (see Section 4.1 for more details), the original stroke is split and broken into segments (b). The segments are then processed sequentially. After each individual segment is added (c, from top to bottom), suggestions are generated (d) using previous segments as well as old strokes. In particular, beginning with the second segment, the beginning endpoint is constrained to match the final endpoint of the previous segment (c, red circles, see Section 4.3). After generating suggestions for a segment (d, from top to bottom), an optional set reduction can be done (e) to keep the evaluation sufficiently fast (see Section 4.2).

the single-segment input can produce up to 10 suggestions). This might seem to be a very severe restriction, but the split segments are typically simple paths with very little ambiguity. Second, we process the individual segments in a breadth-first manner that lets us execute another reduction once all the parallel states reach the same depth (i.e., they all have the same number of processed segments; see individual rows in Figure 12d). For this step, we assign each intermediate state a value calculated as the arithmetic mean of the scores of the processed segments. Then, only N_{IS} intermediate states are kept and evaluated further while the rest are discarded (Figure 12e). The performance of multi-segment input processing is determined by the number of segments K and the intermediate stack size N_{IS}, with $N_{IS} = 1$ being performance-wise equal to sequential processing of individual segments. In our implementation, $N_{IS} = 10$ and strokes constituted of up to 10 segments can be processed without noticeable lagging.

4.3. Internal Segment Restrictions

As the individual segments are pieces of one original input curve, we must ensure that the beautified segments are consecutively joined. Thus, we constrain the position of the first endpoint of each segment after the first (rows 2,3 in Figure 12c). Additionally, if the input stroke is closed, we also constrain the last segment's final endpoint (row 3 in Figure 12c). As a side effect, this also helps to decrease the ambiguity.

4.4. Segment Joining And Further Behavior

Once all the segments have been processed, we create the final output stroke by joining them together. This way, the combined beautified input stroke can be used by rules such as curve identity. Internally, the beautification engine keeps also tracks the individual segments so that they behave as if they were drawn one after each other. This lets the geometric rules show the expected behavior, e.g., the corners of a complex stroke can be used as snapping points.

5. Implementation Details

While using an existing API requires us to conform to its design rules, it also eliminates the need to handle many tasks unrelated to the research project, such as tracking the input device, fitting paths to the samples, and managing the undo/redo stack. It also benefits the users, as they are not forced to learn yet another user interface, and can instead take advantage of built-in tools of the existing program. Therefore, we decided to integrate our system into Adobe Illustrator as a plugin using its C++ SDK.

As described previously, our method is based on evaluating different geometric rules on a new path using the previously drawn and resolved paths. Thus, we need to be able to detect when a new path is created or an old one is modified or deleted. To this end, we serialize all the path data and store a copy in the document. Illustrator activates our system whenever the user modifies the document. We deserialize the data and compare the paths to the actual paths in the document to detect changes. If we find a new path, we process the new path and update the serialized data. Similarly, when a path is modified, it is treated as new one and reprocessed. Deleting paths does not affect the remaining ones. To support undo and redo, we store the serialized data into a part of document that is managed by the undo/redo system.

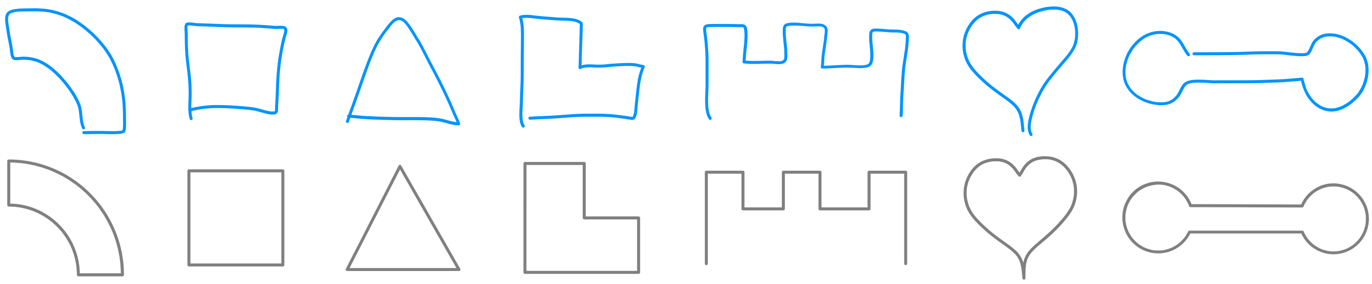

Figure 13: Examples of multi-segment stroke processing. The input strokes (blue) are broken into individual segment that are sequentially processed using the single-segment evaluation engine (Section 3) and merged after the processing is finished (see Section 4 for details).

The presentation of the suggestions is deliberately kept as simple as possible and only one suggestion is shown at the time. The user switches among the suggestions using an additional Illustrator tool panel. The last suggestion in the list is always the original input path and is thus easily accessible. Currently, the list of inferred constraints is shown in textual form in the order in which they were traversed in the search space tree (see Figure 18c). The user selects the current suggestion by drawing a new path or changing the selection. To provide additional assistance for the user, we also present a simple visualization of the applied rules together with rectified path. This visual annotation provides immediate feedback about the imposed constraints and relations of the user input (see Figure 15).

To further exploit the built-in tools, we support the "Transform Again" feature for rotational symmetry. If the resolved path is a rotated copy of an existing path, it is noted as such so that a new, properly-rotated copy will be created if the user invokes the "Transform Again" command. The user only needs to draw two rotated instances of a path and then can create additional properly-rotated paths without drawing them (see Figure 18d). Recall that the rotation angle is adjusted to the nearest integer quotient of 2π, so additional paths can form full n-fold rotational symmetry.

The constraints imposed by ShipShape can easily be avoided for certain paths by placing them in layers that are not being rectified. In our implementation, ShipShaperuns only on the default layer.

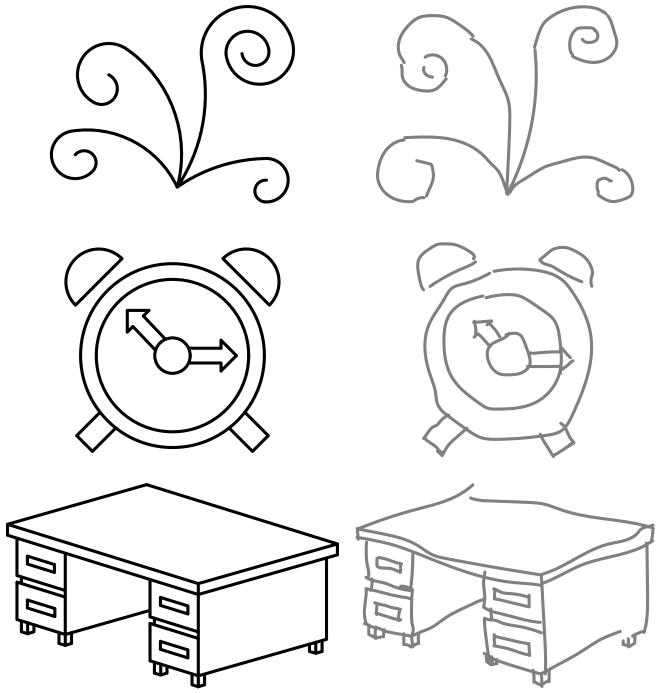

Figure 14: Evaluation study drawings. The users were asked to recreate these drawings using our ShipShape prototype: Task drawing (left, black), representative raw input (right, gray).

6. Results

To evaluate the effectiveness of our method, we conducted a preliminary study. We created a plugin for Adobe Illustrator that was installed on a PC with a 23in LCD monitor and a consumer computer mouse as the input device. Six people participated in this study. All of them worked with Illustrator on a daily to weekly basis, but in all cases, their primary work-related tool was a CAD program. First, the users were given a brief introduction and demonstration of our system's concept, capabilities and limitations, with a few practical examples. The participants could adjust Illustrator settings and the mouse sensitivity according to their needs, and then spent 1 to 3 minutes in free drawing, to get briefly accustomed to the system and the workflow.

The users were then shown three simple illustrations (see Figure 14) and presented with the task of drawing each of them anew, using both native Illustrator tools and our prototype, while we measured their drawing times. First, the participants were asked to recreate the figures using any suitable tools and approaches, i.e., they could use all the available tools and modes, such as copying or reflecting. Rather than creating the exact copies of the reference drawings, we directed them to focus on preserving the geometric relations. Interestingly, despite the users' relatively equal level of experience, they often took very dissimilar ways to recreate the task's drawing.

In the second part of the test, the participants were only allowed to use the pencil tool with the ShipShape prototype turned on. The only additional allowed operation was undo. Similarly to the first part of each drawing, the users took a different approaches to complete the goal, however, with a sin-

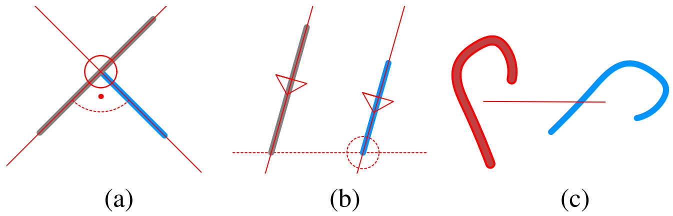

Figure 15: Visual annotation hints. Overlaid visual annotations show which rules have been applied, e.g., line perpendicularity and endpoint snapping (a), line parallelism and single coordinate snapping (b) or path identity (c).

gle exception, they were all able to finish all three designated drawings from Figure 14. The initial measurements (Figure 16) suggest that drawing beautification is more suited for simpler drawings and tasks. For example, copying a large part of the bottom-left drawing in Figure 14 was always faster than redrawing it.

The main interest of this study was to identify the weak points and bottlenecks of our approach and to test how successful our prototype was in generating correct suggestions. The overall feedback from the participants was positive. They found the tool useful and easy to use. Most of the participants, however, considered the limitation of using a single tool only too restrictive, and suggested incorporating parts of our approach (smart snapping, automatic tangent adjustments, etc.) into the relevant built-in tools. All the participants liked the idea of visual annotations (Figure 15) and found it helpful. Several users did not like the way the alternative suggestions were presented and explored (see the small gray box in canvas in Figure 18) and preferred to undo and redraw the particular strokes.

Additional results are shown in Figure 17. Note that an important part of the drawing workflow was relying on Illustrator's built-in support for curve smoothing when creating original paths—those that are not copies of other paths. These are shown in blue in Figure 17, and they function as "template" paths for the beautification. Other strokes drawn afterwards can be much more imprecise (see Figure 1 and Figure 17c–g).

7. Limitations and Future Work

A common problem of drawing beautification techniques is the quick growth of the number of possible suggestions as the drawing becomes more complex and many satisfiable geometric constraints emerge. Our approach addresses this by combining best-first search with a limited suggestion set size, but additional heuristic-based pruning of the search space, possibly based on empirical measurements, could improve the suggestion set.

Currently, when the user changes an already-resolved path, it is treated as a new one. In some cases, however, it would be beneficial to not only reprocess the modified path but also all other paths being in relationship with it, for example changing any reflected or rotated versions of the path.

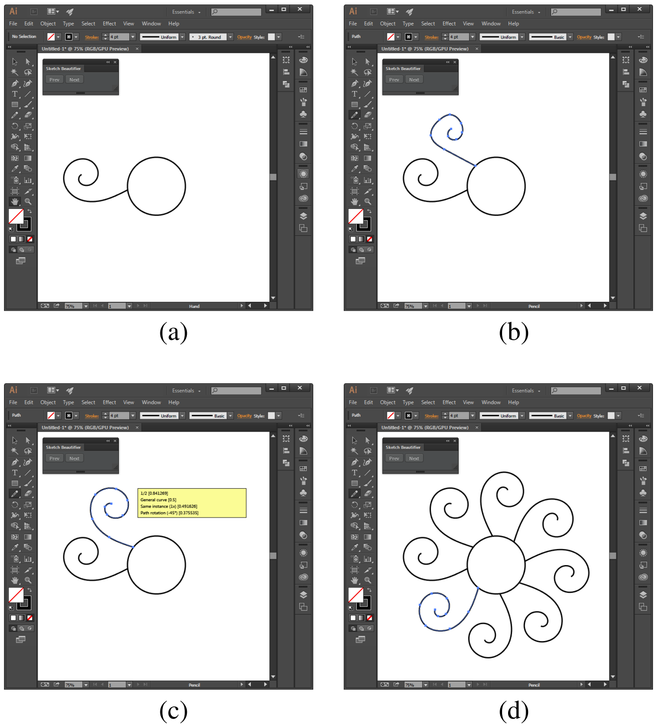

Figure 18: Exploiting the "Transform Again" feature. Illustrator allows the user to repeat the last transformation. When a new path is added (b) to the canvas (a), it is processed and output suggestions are generated. If the user chooses a suggestion that is a rotation (c) we enable the "Transform Again" feature. The user can then easily complete the 8-fold rotational symmetry drawing (d). See Section 5

8. Conclusion

In this paper, we presented an efficient method for beautification of freehand sketches. Since the user input is often imprecise and thus ambiguous, multiple output suggestions must be generated. To this end, we formulated this problem as search in a rooted tree graph where nodes contain transformed input stroke, edges represent applications of geometric rules and suitable suggestions correspond to different paths from root node to some leaf nodes. To avoid the computational complexity of traversal through the whole graph, we utilized a best-first search approach where the order of visiting tree nodes is directed by the likelihood of application of the particular geometric rules.

On top of this framework, we developed a system of self-contained rules representing different geometric transformations, which can be easily extended. We implemented various rules that can work not only with simple primitives like line segments and circular arcs, but also with general Bézier curves, for which we showed how to detect previously unsupported relations such as curve identity or rotational and reflection symmetry.

We demonstrated the usability and potential of our method on various complex drawings. Thanks to the ability to process general curves, our system extends the range of applicability of freehand sketching, which was limited previously to drawings in specialized, highly-structured applications like forms or technical diagrams. We believe that this advantage will become

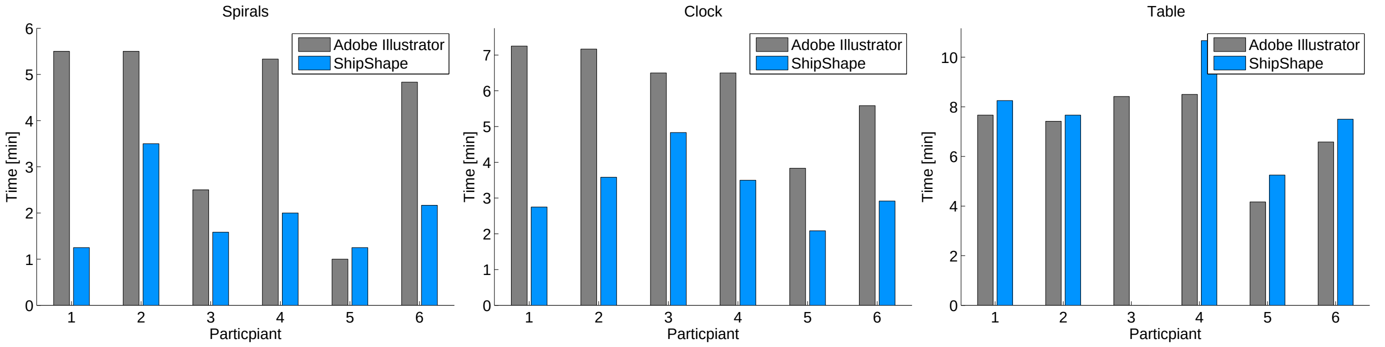

Figure 16: Comparison of drawing performance. The participants were asked to recreate the drawings from Figure 14 using either native tools of Adobe Illustrator (red) or ShipShape prototype (blue line). For simpler drawing, such as the spiral or the clock, ShipShape typically outperformed Illustrator. However, with more complex drawings (table), the utilization of different tools is faster.

even more apparent with the widespread adoption of touch-centric devices, which rely much less on classical beautification techniques that are based upon menu commands and multiple tools.

9. Acknowledgements

We would like to thank all the anonymous reviewers for their constructive comments. This research began as an internship by Jakub Fišer at Adobe Research, and was partly supported by the Technology Agency of the Czech Republic under the research program TE01020415 (V3C – Visual Computing Competence Center) and partially by the Grant Agency of the Czech Technical University in Prague, grant No. SGS13/214/-OHK3/3T/13 (Research of Progressive Computer Graphics Methods).

Appendix A. Rules Evaluation

The rules are evaluated using a piecewise-linear ramp functions, both continuous and discontinuous. These functions transform the input values, such as angular differences or view-space distances, to likelihood values from the interval [0, 1] used to direct the tree expansion and final suggestion sorting described in the paper. For each rule listed in section 3.1 in the main paper, we show the exact scoring function we used in our implementation.

Appendix A.1. Line Detection

As in QuickDraw [10], we calculate the deviation from straightness $D = |1 - |l_c|/|l_l||$, where $|l_c|$ is length of sampled Bézier curve and $|l_l|$ length of line segment between its endpoints. If D is lower than the threshold 0.05, we set the likelihood $\mathcal{L}_{LD}$ of the curve being a line segment to $1 - D$.

Appendix A.2. Arc Detection

The arc is described by three parameters – center, radius and angular span. We initially sample the input path and obtain the circle fit center location and radius value using least-squares approach. We then project the samples onto the optimal circle, using the circle center as the center of the projection, to determine the span value. Having these three values, we construct the arc suggestion and compute its similarity with the input using discrete Fréchet distance between the original samples and the suggested arc's samples. This distance is then used to calculate the final output likelihood $\mathcal{L}_{AD}$ (Figure A.19). If the detected span is higher than $2\pi - \pi/13$ or the input path is closed, the output span is set to 2π to suggest full circle output.

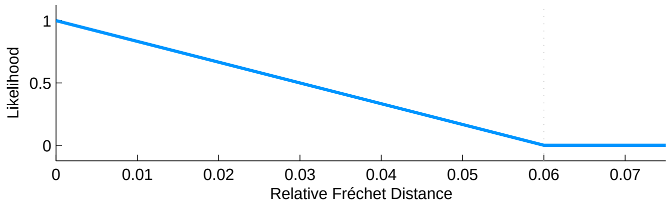

Figure A.19: Relative discrete Fréchet distance evaluation in *Arc Detection* rule.

Appendix A.3. Endpoint Snapping

We measure the distances between the endpoint and the points of interest (other endpoints, arc centers, etc.) in view-space pixels and transform them to final likelihoods $\mathcal{L}_{ES}$ (Figure A.20). As the users can end strokes relatively precisely even with devices such as mouse or touchpad, there is no tolerance zone in the scoring function.

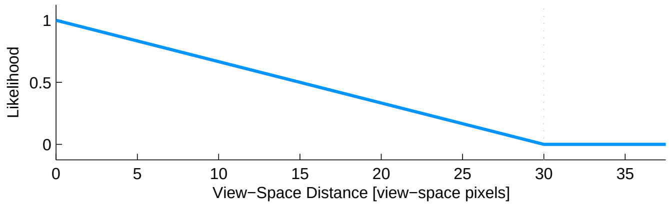

Figure A.20: View-space distance evaluation in *Endpoint Snapping* rule.

Appendix A.4. End Tangent Alignment

The angular difference between the curve endpoint and the endpoint it is connected to is directly transformed to final likelihood $\mathcal{L}_{ETA}$ (Figure A.21).

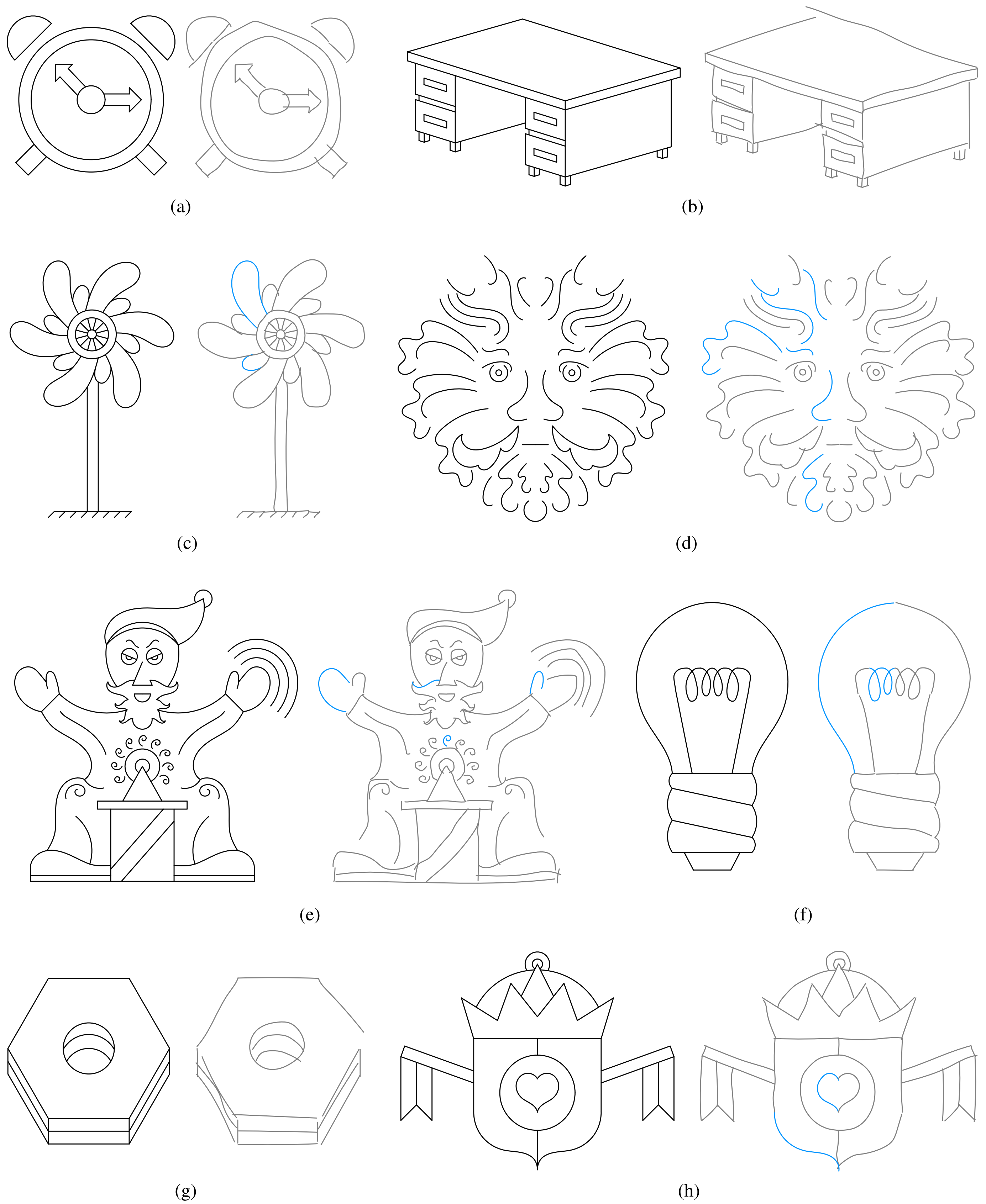

Figure 17: Various results obtained using our method. The side-by-side pairs show the beautified output (black) and the original input strokes (gray). Note that we do not perform any curve smoothing, beyond what is provided by Illustrator. Therefore, when dealing with general curves, the first "template" strokes (blue) have to be drawn more precisely or be smoothed using built-in Illustrator capabilities.

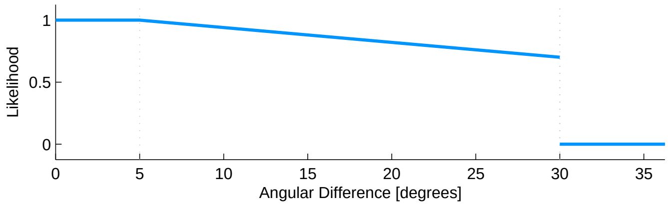

Figure A.21: Angular difference evaluation in *End Tangent Alignment* rule.

Appendix A.5. Line Parallelism and Perpendicularity

We measure the angular difference between the direction vectors of two line segments to obtain the likelihood $\mathcal{L}_{dff}$ (Figure A.22 top). To increase the final likelihood of nearby line segments, we also score the view-space distance between tested line segments – $\mathcal{L}_{dst}$ (Figure A.22 bottom). The output suggestion with likelihood $\mathcal{L}_{LP} = \mathcal{L}_{dff}\mathcal{L}_{dst}$ is produced, if $\mathcal{L}_{LP} > 0.7$.

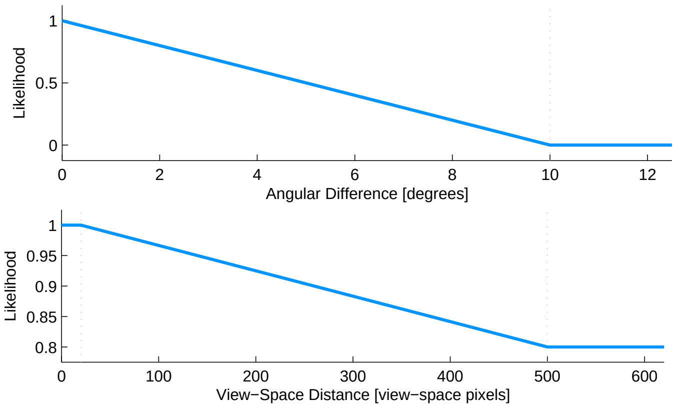

Figure A.22: Angular difference evaluation (top) and view-space distance evaluation (bottom) in *Line Parallelism* and *Line Perpendicularity* rules.

Appendix A.6. Line Length Equality

We measure the line length difference relative to a tested line segment to get the likelihood $\mathcal{L}_{dff}$ (Figure A.23 top) and also the likelihood $\mathcal{L}_{dst}$ (Figure A.23 bottom) based on relative distances of existing line segments to the tested one. Similarly to line parallelism rule, the final likelihood is computed as $\mathcal{L}_{LLE} = \mathcal{L}_{dff}\mathcal{L}_{dst}$ and an output suggestion is produced, if $\mathcal{L}_{LLE} > 0.7$.

Appendix A.7. Arc and Circle Center Snapping

Based on the arc's span θ and radius r, we first determine the search distance $D' = \max(D, 2r(1-\theta/2\pi))$, where the default distance $D = 30$ (view-space pixels) is equal to the one used in endpoint snapping and also the final likelihood $\mathcal{L}_{ACCS}$ is then computed using the same ramp function (Figure A.20).

Appendix A.8. Path Identity

We compute the discrete Fréchet distance between the tested path and the existing one, as described in Section 3.4. The absolute distance δ_F is then made relative to the length of the tested path and used to compute the likelihood $\mathcal{L}_{PI}$ (Figure A.24).

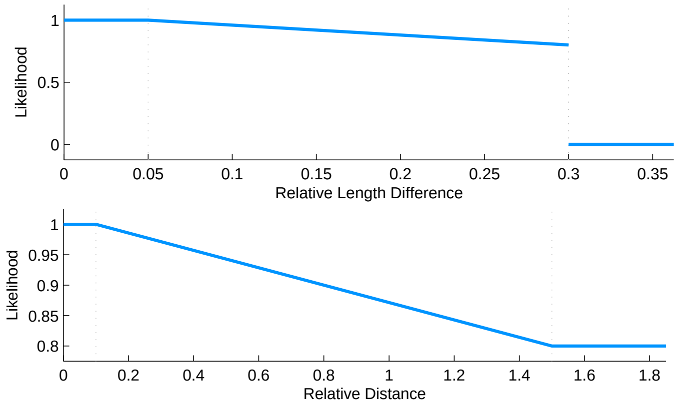

Figure A.23: Relative length difference evaluation (top) and relative distance evaluation (bottom) in *Line Length Equality* rule.

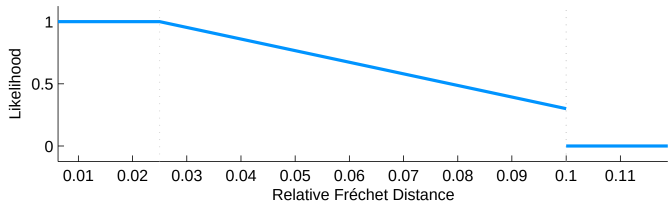

Figure A.24: Relative discrete Fréchet distance evaluation in *Path Identity* rule.

Appendix A.9. Path Transformation Adjustment

We compute four separate likelihoods for the angular difference $\mathcal{L}_a$, the scale difference $\mathcal{L}_s$, the x offset $\mathcal{L}_x$, and the y offset $\mathcal{L}_y$, and perform only those with non-zero values. The final likelihood is $\mathcal{L}_{TA} = 1 - (1-\mathcal{L}_a)(1-\mathcal{L}_s)(1-\mathcal{L}_x)(1-\mathcal{L}_y)$. Note that the maximum likelihood is relatively small compared to other rules; if they were larger, this would usually overwhelm the likelihoods of other suggestions.

Appendix A.10. Path Offset

The process to obtain samples along the tested path together with their signed distances to the existing path is described in Section 3.6. To compute the likelihood $\mathcal{L}_{PO}$ we evaluate the relative distance difference between 25th and 75th quantile from the sorted hit data (Figure A.26).

[1] Igarashi T, Matsuoka S, Kawachiya S, Tanaka H. Interactive beautification: A technique for rapid geometric design. In: Proceedings of ACM Symposium on User Interface Software and Technology. 1997, p. 105–14.
[2] Pavlidis T, Van Wyk CJ. An automatic beautifier for drawings and illustrations. ACM SIGGRAPH Computer Graphics 1985;19(3):225–34.
[3] Igarashi T, Hughes JF. A suggestive interface for 3d drawing. In: Proceedings of ACM Symposium on User Interface Software and Technology. 2001, p. 173–81.
[4] Plimmer B, Grundy J. Beautifying sketching-based design tool content: Issues and experiences. In: Proceedings of Australasian Conference on User Interface. 2005, p. 31–8.
[5] Wang B, Sun J, Plimmer B. Exploring sketch beautification techniques. In: Proceedings of ACM SIGCHI New Zealand Chapter's International Conference on Computer-human Interaction: Making CHI Natural. 2005, p. 15–6.
[6] Zeleznik R, Bragdon A, chi Liu C, Forsberg A. Lineogrammer: Creating diagrams by drawing. In: Proceedings of ACM Symposium on User Interface Software and Technology. 2008, p. 161–70.

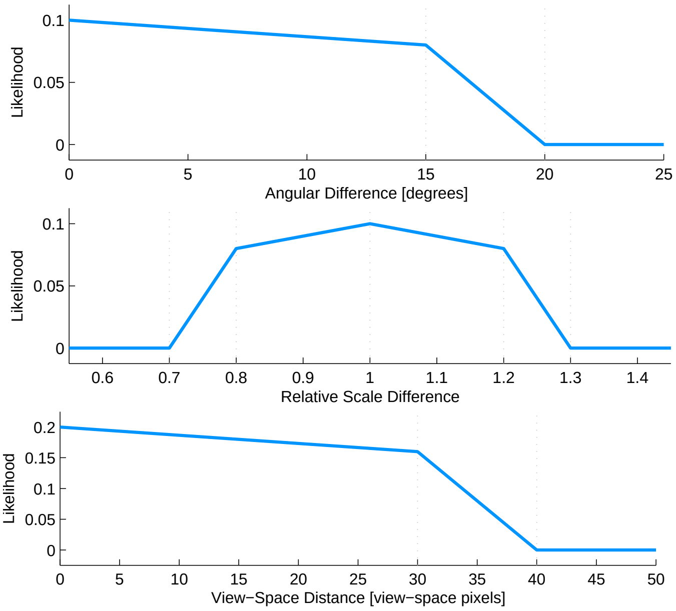

Figure A.25: Angular difference evaluation (top), relative scale evaluation (middle) and view-space distance evaluation (bottom) in *Transform Adjustment* rule.

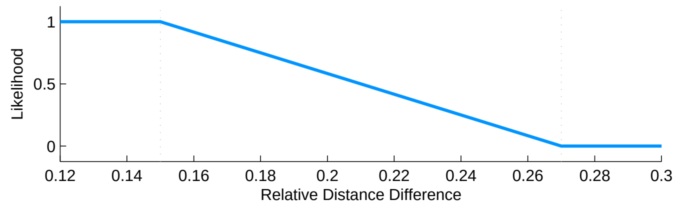

Figure A.26: Relative distance difference evaluation in *Path Offset* rule.

[7] LaViola Jr. JJ, Zeleznik RC. Mathpad2: A system for the creation and exploration of mathematical sketches. ACM Transactions on Graphics 2004;23(3):432–40.

[8] Paulson B, Hammond T. Paleosketch: Accurate primitive sketch recognition and beautification. In: Proceedings of International Conference on Intelligent User Interfaces. 2008, p. 1–10.

[9] Murugappan S, Sellamani S, Ramani K. Towards beautification of freehand sketches using suggestions. In: Proceedings of Eurographics Symposium on Sketch-Based Interfaces and Modeling. 2009, p. 69–76.

[10] Cheema S, Gulwani S, LaViola J. Quickdraw: Improving drawing experience for geometric diagrams. In: Proceedings of SIGCHI Conference on Human Factors in Computing Systems. 2012, p. 1037–64.

[11] Baran I, Lehtinen J, Popovic J. Sketching clothoid splines using shortest paths. Computer Graphics Forum 2010;29(2):655–64.

[12] Orbay G, Kara LB. Beautification of design sketches using trainable stroke clustering and curve fitting. IEEE Transactions on Visualization and Computer Graphics 2011;17(5):694–708.

[13] Lee YJ, Zitnick CL, Cohen MF. Shadowdraw: real-time user guidance for freehand drawing. ACM Transactions on Graphics 2011;30(4):27.

[14] Zitnick CL. Handwriting beautification using token means. ACM Transactions on Graphics 2013;32(4):8.

[15] Noris G, Hornung A, Sumner RW, Simmons M, Gross M. Topology-driven vectorization of clean line drawings. ACM Transactions on Graphics 2013;32(1):11.

[16] Su Q, Li WHA, Wang J, Fu H. Ez-sketching: Three-level optimization for error-tolerant image tracing. ACM Transactions on Graphics 2014;33(4):9.

[17] Fišer J, Asente P, Sýkora D. Shipshape: A drawing beautification assistant. In: Sketch-Based Interfaces and Modeling. 2015, p. 9.

[18] Eiter T, Mannila H. Computing discrete Fréchet distance. Tech. Rep.; Technische Universität Wien; 1994.

[19] Ramer U. An iterative procedure for the polygonal approximation of plane curves. Computer Graphics and Image Processing 1972;1(3):244–56.

[20] Douglas DD, Peucker KT. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. Cartographica: The International Journal for Geographic Information and Geovisualization 1973;10(2):112–22.