EUROGRAPHICS Symposium on Sketch-Based Interfaces and Modeling
Ergun Akleman (Editor)

EXPRESSIVE 2015

ShipShape: A Drawing Beautification Assistant

J. Fišer[†1], P. Asente[2] and D. Sýkora[1]

[1]CTU in Prague, FEE
[2]Adobe Research

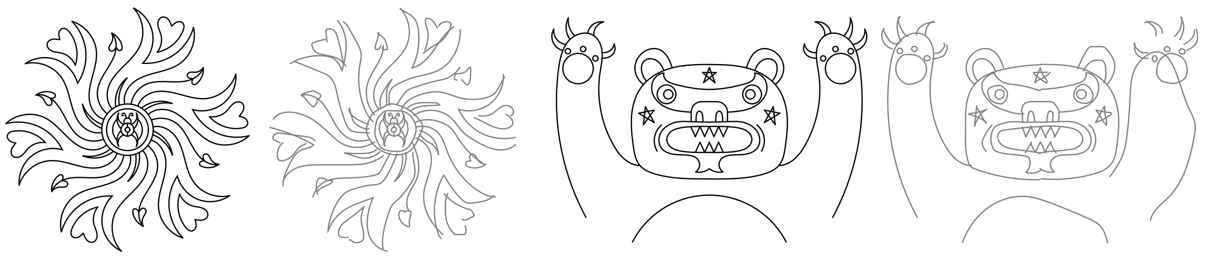


Figure 1: *Examples of drawings created using ShipShape. The final drawings (black) were created from the imprecise user input (gray) by beautifying one stroke at a time, using geometric properties such as symmetry and curve identity. See Figure 11 for more results.*

Abstract

Sketching is one of the simplest ways to visualize ideas. Its key advantage is requiring the user to have neither deep knowledge of a particular drawing software 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 demonstrate various results to demonstrate capabilities of the proposed method.

Categories and Subject Descriptors (according to ACM CCS): I.3.4 [Computer Graphics]: Graphics Utilities—Paint systems I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction techniques I.4.3 [Image Processing and Computer Vision]: Enhancement—Geometric correction H.5.2 [User Interfaces]: Input devices and strategies—

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

[†] e-mail:fiserja9@fel.cvut.cz

DOI: 10.2312/exp.20151178

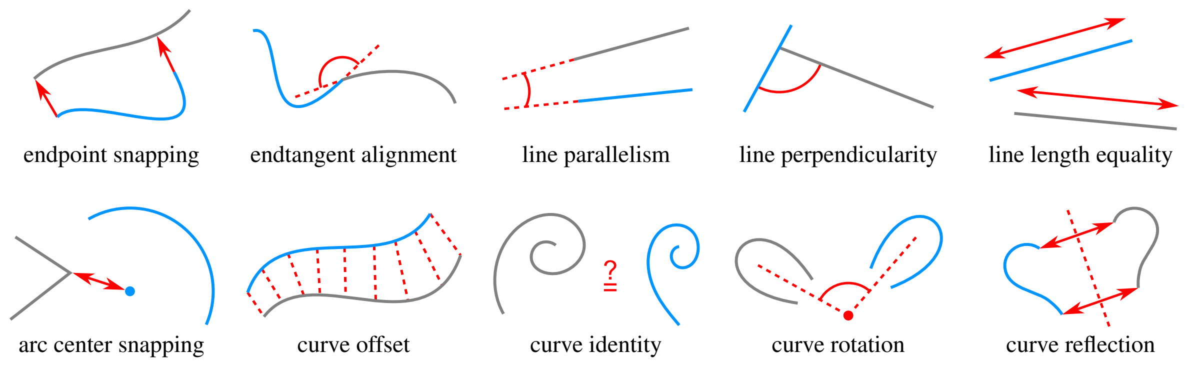


Figure 2: *Supported geometric rules and transformations in our framework. The blue paths represent the data being beautified, while gray paths are data already processed. For more detailed description of the criteria used to evaluate these constraints, see Section 3.1.*

logo study creation, the *interactive beautification* [IMKT97] 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. [IMKT97]. Strokes are processed incrementally (see Figure 3) to prevent the combinatorial explosion of

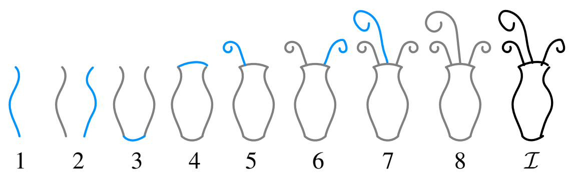


Figure 3: *Incremental beautification workflow. Every newly drawn stroke (blue) is beautified using previously created data (gray). The first stroke is left unchanged. As the drawing continues, more suitable geometric constraints emerge and are applied, such as curve identity (2,6,7), reflection (2,6)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

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 [PVW85] 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 drawings were considered, since the method needed to drop many constraints to keep the solution tractable.

To alleviate this limitation, Igarashi et al. [IMKT97] 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 a more specific scenarios such as the interactive creation of 3D drawings [IH01], block diagrams [PG05, WSP05], forms [ZBcLF08], and mathematical equations [LZ04].

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 [PH08] 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. [MSR09] and Cheema et al. [CGL12] 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 [BLP10, OK11] or some example-based strategy [LZC11,Zit13]. Additionally, advanced methods for vectorizing and refining raster inputs have been proposed [NHS*13, SLWF14], 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 pre-processes the user's rough input strokes into smooth, fair paths.

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* [PH08]), 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. [IMKT97] 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 2 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.

As previously described, each rule outputs a likelihood score for the transformed path. Since we need to compare scores among different rules, they are always normalized into the interval $[0,1]$. 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

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

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

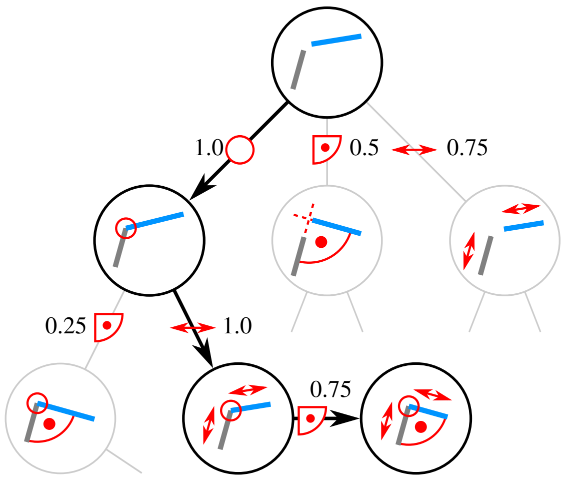


Figure 4: *Successive rule evaluation and application. In this example, the evaluation engine consists of three geometric rules—endpoint snapping, perpendicularity, and length equality. The old data (gray path) is fixed in the canvas. When a new path (blue) is added, it becomes the root node of the evaluation 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).*

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 previously 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 2), and then we

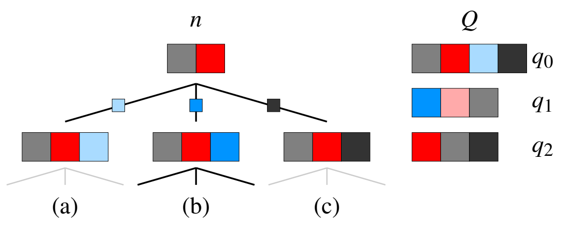


Figure 5: *Search graph pruning. The rules are represented by colored boxes with hue being distinct rules and lightness their unique applications (e.g., if red color represents endpoint snapping, then different shades of red correspond to snapping to different positions). An inner node n has been expanded 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.*

present some additional implementation issues including a more detailed description for non-trivial rules.

Line Detection We estimate path's deviation from straightness by measuring the ratio between its length and the distance between its endpoints, as in QuickDraw [CGL12].

Arc Detection We compute the approximate curvature by calculating angles between tangents of successive path samples. We check whether it is sufficiently uniform and whether the angular span is within an expected range, to prevent treating slightly bent lines or spirals as circular arcs. When the span is close to 2π or the path is closed, we replace it with 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 explicitly join adjacent segments, as this would change their semantic meaning. However, this effect can be easily mimicked by using round caps on the curves.

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.

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.

Path Identity To detect that two paths have similar shapes, we align them and compute their discrete Fréchet distance [EM94]. We also account for different scales. More details are given in Section 3.4.

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 3.5.

Path Rotational Symmetry For a tested path x and resolved reference path y of the "same shape" (determined by successful application of the path-identity rule), we find the optimal rotation matrix (i.e, center point c and angle of rotation α) that transforms y to y' and minimizes the distance between endpoints of x and y'. Similarly to arc center snapping, we try to adjust the position of the rotation center and snap it to some existing point. Since our system works in incremental fashion, we also adjust the angle of rotation to the nearest integer quotient of 2π, so that additional paths can be placed to form full n-fold rotational symmetry.

Path Reflection Symmetry Similar to rotational symmetry, we find an axis that minimizes the difference between the reflected version of existing path y and the tested path x. Nearby existing reflection axes are tested, and we also try to make the found axis parallel with a coordinate axis.

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.

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 [Ram72, DP73]. For a polyline p, this finds a reduced version p' with fewer points within given tolerance ε, i.e., all points of p' lie within the distance ε of the original path (see Figure 6). Our implementation uses $\varepsilon = 4$ view-space pixels at the time the path was drawn.

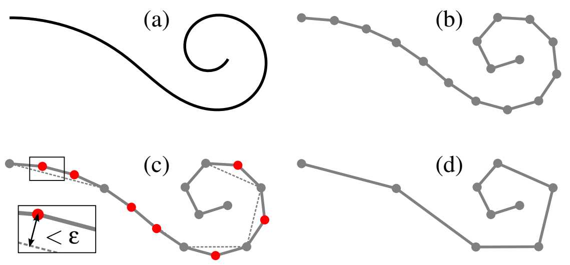


Figure 6: *Path sample simplification. The original Bézier path (a) is equidistantly sampled, giving a polyline (b). The Ramer–Douglas–Peucker algorithm then recursively simplifies the polyline by omitting points closer than ε (c) to the current approximation, finally constructing simplified polyline (d).*

3.4. Path Matching

A significant 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* [EM94], 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 $f : [a,b] \to M$, where $a,b \in \mathbb{R}$, $a \leq b$. Given two paths $f : [a,b] \to M$ and $g : [a',b'] \to M$, their Fréchet distance δ_F is defined as

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

where α (resp. β) is an arbitrary continuous non-decreasing function from $[0,1]$ onto $[a,b]$ (resp. $[a',b']$). 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.

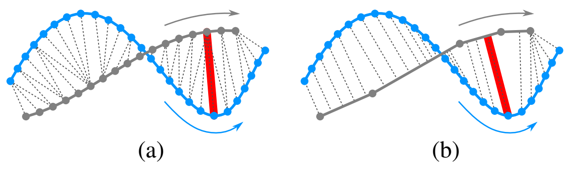


Figure 7: *Discrete Fréchet distance. The minimum length of the line connecting ordered sets of point samples (a). Since we store the resolved paths in the simplified form, we compute the Fréchet distance between an ordered set of points and an ordered set of line segments (b) rather than between two 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 7b) 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 (Figure 8). First, we translate, rotate, and uniformly scale the two paths so that their endpoints match. Second, we scale them (possibly non-uniformly) so that their bounding boxes match. We then compute the discrete Fréchet distance divided by the length of the tested path to obtain the relative similarity measure.

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

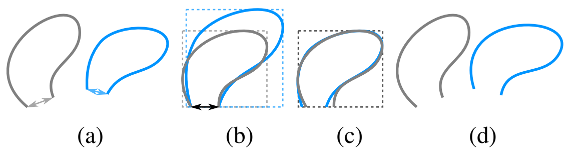


Figure 8: *Path alignment for the computation of the similarity measure. When testing a new path (blue) for similarity with an old one (gray) (a), we match the endpoints of the two paths (b). Then we scale them so their bounding boxes match (c). If the new path is then evaluated as being sufficiently similar, a new, properly-positioned instance of the old path replaces the tested path (d).*

3.5. 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 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 9). If the likelihood is high, we replace the new path with an offset version of the reference.

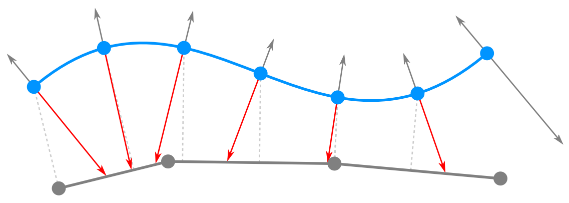


Figure 9: *Offset path detection. A line is constructed from each point on the sampled path (blue circles) in the normal direction. If an existing reference path is hit (red rays), the minimal distance from the sample to the reference path is calculated (dashed lines) and used in offset-path-likelihood computation (see 3.5).*

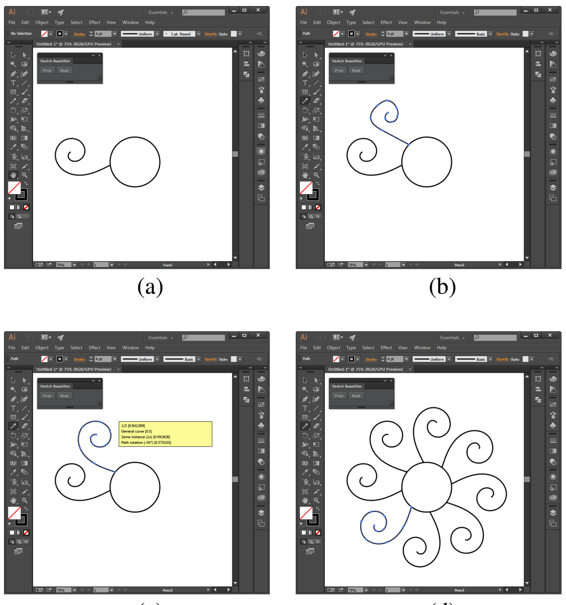
4. Implementation Details

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.

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 10c). The user selects the current suggestion by drawing a new path or changing the selection.

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 10d). Recall that the rotation angle is adjusted to the nearest integer quotient of 2π, so additional paths can form full n-fold rotational symmetry.


1/2 [0.841269]
General curve [0.5]
Same instance (1x) [0.491626]
Path rotation (-45°) [0.375535]

(a) (b)

(c) (d)

Figure 10: *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 4*

5. Results

We created a plugin for Illustrator that created the results shown in Figures 1 and 11. To evaluate our method, we conducted an informal study. We showed four people the ShipShape prototype and asked them to try it out. None of the users was professional artist, however, they were all generally proficient at common computer work. The overall feedback was positive. The users considered the workflow easy to learn and liked the simplicity of the interaction style, using the Pencil Tool exclusively. Generally, the users mentioned the need to draw more precisely as the drawings got more complex, which stems from the fact that with more paths, the probability of the input satisfying some unintended geometric constraints rises. Users also appreciated being able to use "Transform Again" feature (see Figure 10) as they often found it tedious to draw rotated elements manually.

The displayed results vary from relatively simple sketches to quite complex drawings. Despite the limited set of supported operations, the users often found ways to mimic additional constraints. For example, we do not currently support snapping of an endpoint or midpoint to a reflection axis. However, it can be achieved by drawing a line segment representing the axis, using it as the anchor, and later deleting it (done in Figure 11a,f and the supplementary video).

An important part of the 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 11, and they function as "template" paths for the beautification. Other strokes drawn afterwards can be much more imprecise (see Figure 1 and 11c–g).

6. 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.

While we deliberately kept the user interface minimalistic, we believe that a visualization of the inferred constraints could help the users to better understand which rules were applied. For example, we could visualize a reflection axis or highlight the existing paths that contributed to the current result.

7. 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 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.

8. 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).

References

[BLP10] BARAN I., LEHTINEN J., POPOVIC J.: Sketching clothoid splines using shortest paths. *Computer Graphics Forum 29*, 2 (2010), 655–664. 3

[CGL12] 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), pp. 1037–1064. 3, 4

[DP73] DOUGLAS D. D., PEUCKER K. T.: 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 10*, 2 (1973), 112–122. 5

[EM94] EITER T., MANNILA H.: *Computing discrete Fréchet distance*. Tech. rep., Technische Universität Wien, 1994. 5

[IH01] IGARASHI T., HUGHES J. F.: A suggestive interface for 3d drawing. In *Proceedings of ACM Symposium on User Interface Software and Technology* (2001), pp. 173–181. 2

[IMKT97] 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), pp. 105–114. 2, 3

[LZ04] LAVIOLA JR. J. J., ZELEZNIK R. C.: Mathpad2: A system for the creation and exploration of mathematical sketches. *ACM Transactions on Graphics 23*, 3 (2004), 432–440. 2

[LZC11] LEE Y. J., ZITNICK C. L., COHEN M. F.: Shadowdraw: real-time user guidance for freehand drawing. *ACM Transactions on Graphics 30*, 4 (2011), 27. 3

[MSR09] 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), pp. 69–76. 3

[NHS*13] NORIS G., HORNUNG A., SUMNER R. W., SIMMONS M., GROSS M.: Topology-driven vectorization of clean line drawings. *ACM Transactions on Graphics 32*, 1 (2013), 11. 3

[OK11] ORBAY G., KARA L. B.: Beautification of design sketches using trainable stroke clustering and curve fitting. *IEEE Transactions on Visualization and Computer Graphics 17*, 5 (2011), 694–708. 3

[PG05] PLIMMER B., GRUNDY J.: Beautifying sketching-based design tool content: Issues and experiences. In *Proceedings of Australasian Conference on User Interface* (2005), pp. 31–38. 2

[PH08] PAULSON B., HAMMOND T.: Paleosketch: Accurate primitive sketch recognition and beautification. In *Proceedings of International Conference on Intelligent User Interfaces* (2008), pp. 1–10. 3

[PVW85] PAVLIDIS T., VAN WYK C. J.: An automatic beautifier for drawings and illustrations. *ACM SIGGRAPH Computer Graphics 19*, 3 (1985), 225–234. 2

[Ram72] RAMER U.: An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing 1*, 3 (1972), 244–256. 5

[SLWF14] SU Q., LI W. H. A., WANG J., FU H.: Ez-sketching: Three-level optimization for error-tolerant image tracing. *ACM Transactions on Graphics 33*, 4 (2014), 9. 3

[WSP05] 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), pp. 15–16. 2

[ZBcLF08] 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), pp. 161–170. 2

[Zit13] ZITNICK C. L.: Handwriting beautification using token means. *ACM Transactions on Graphics 32*, 4 (2013), 8. 3

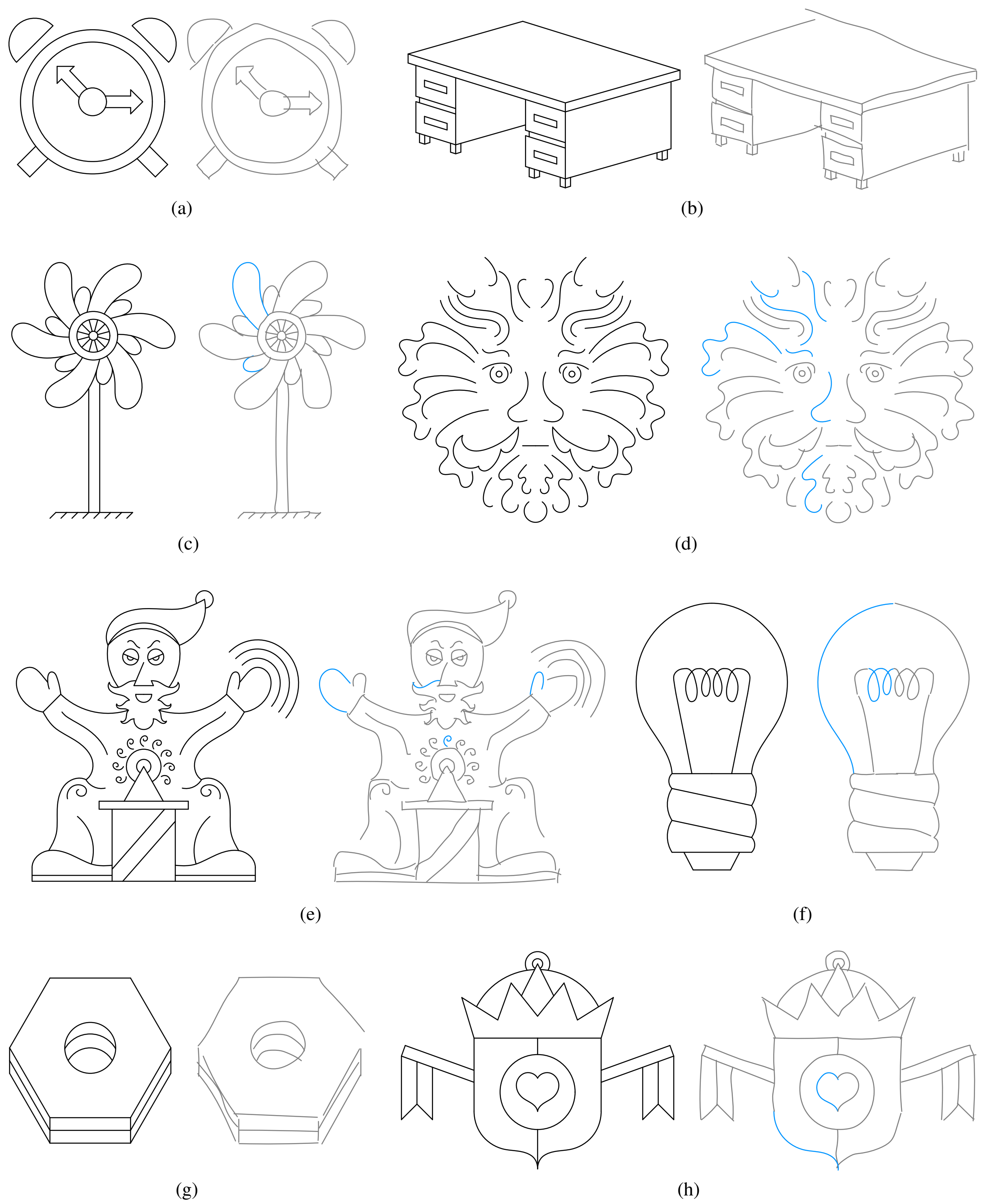


Figure 11: *Various results obtained using our method. The side-by-side pairs show the beautified output (black) and the original input strokes (gray). Note that we do not perform any curve smoothing, beyond what is provided by Illustrator. Therefore, when dealing with general curves, the first "template" strokes (blue) have to be drawn more precisely or be smoothed using built-in Illustrator capabilities.*