Sketch Beautifier

Aditi Mithal
2013122 IIIT-Delhi

Sambhav Satija
2013085 IIIT-Delhi

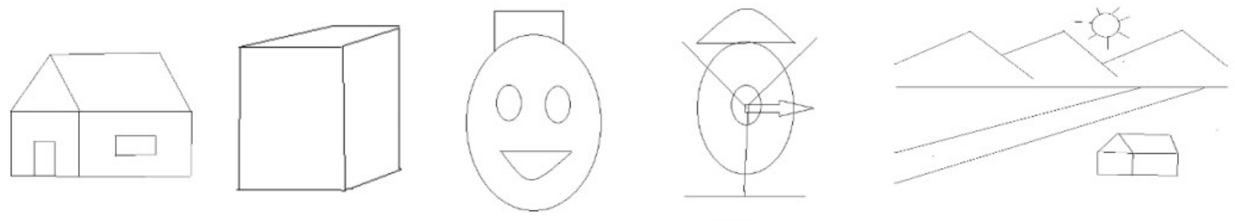

Figure 1:*Beautified Sketches*

Abstract

There are a lot of tools which try to clean up the hand drawn path into cleaner sketches. We will try to document our approach as we try to create a tool that too will work on beautifying the sketch. For this, the initial drawn sketch is identified as simple primitives. Then, constraints are applied for the corresponding primitive and the original image in order to try and clean up the image within certain rules.

Keywords: line, sketch, constraint, detection, points, snapping, beautify

1 Introduction

It's well known that sketching is one of the simplest ways to visualize ideas. The key is that it doesn't require the user to have deep knowledge of a particular drawing software nor any advanced drawing skills. It helps to put across creative ideas easily. However, in order for ideas to be easily imaginable, clean and precise sketches are necessary which are cumbersome to create via softwares and almost impossible to draw freehand. Here, we document our approach of creating a tool that tries to understand the sketches drawn by the user into cleaner figures.

Pipeline:

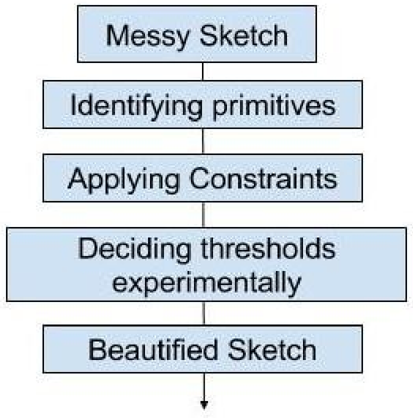

Figure *1: Pipeline of steps followed to beautify a sketch*

Milestones:

- Literature Review
- Detecting primitives - Line, Circle
- Applying Constraints - Parallel, Endpoint Snapping, Lines drawn w.r.t. other line objects
- Adding circle constraints. Nesting constraints in order to create better sketches.

Features:

- Enable user to make a freehand sketch and the tool beautifies it.
- Clear the screen
- Redo/Undo
- Save a file

2 Approach

Our aim is to provide users a tool to beautify the sketches/ drawings they make.

Beautification involves three steps:

- Recognize the sketch and categorize it into primitives
- Apply the constraints and suggest the most suitable change.
- Apply Beautification

We have a step by step procedure to beautify the sketch

2.1 Recognizing the Sketch/Shape

We have some Primitives(expandable) under which the components of a sketch can be classified:

- Line
- Circle(circular curve)

For recognizing the primitive types, we took to following the techniques developed in previous works, or where required, our own implementations.

- Line detection: We find the ratio between its length and the distance between its endpoints , thus finding the straightness of a line. ratio~1.0 for a line. This was similar to the method used by Fiser et al[1].

```
$let\ \alpha\ =\ 0.90\ (straight\ line\ threshold)$
$let\ p\ =\ all\ traced\ points$
$pathSum = \sum_{i=1}^{n} euclid(p[i], p[i-1])$
$lineSum = euclid(p[0],\ p[n])$
$if\ (lineSum/pathSum) > \alpha :$
    $return\ leastSquareError(p[0],\ p[n])$
```

To find the length of the path (drawn): we track the distance between every consecutive point and find their sum . This gives us the length of the path drawn by the user

- Circle Detection: For detecting circles, we tried various checks. In one of the earlier implementations, we tried to find the centre of the circle by clustering the intersections of the normals of all the points on the path, and finding the average of the points with the highest number of nearest numbers. This however, didn't work well practically. Surprisingly, a simple algorithm of approximating the centre as the mean of all the points on the path worked pretty well practically.

```
$let\ p\ be\ all\ points\ in\ drawn\ path$
$n = \{normal\ at\ (x) \mid x \in p\}$
$c = \{intersection(i,j) \mid i,j \in\ n\}$
$centre = mean(pointsWithMaxNN)$
```

2.2 Apply Constraints and suggest suitable changes

Once we categorize the sketched path into a primitive, we apply various constraints/rules, which return the image suggestions on applying that specific constraint. Every constraint carries a threshold which is taken experimentally.

Line	Parallel
	Horizontal
	Vertical
	End Point Snapped
Circle	Snapping of Line with centre and Circumference
	Concentric Circles

3 Calculation and Constraint Analysis

3.1 Sampling the points

Since the total complexity depends on n(number of points in the path), we have to reduce the number of points in the path passed to the classifier. For this, we did a linear sampling of the points.

$$\alpha\ =\ 1000$$
$$f_{sample}(X)\ =\ X\ \ if\ |X|\ <\ \alpha$$
$$f_{sample}(X)\ =\ \{X_{(i*|X|)/1000}\ \forall\ i\ \in\ \{1...\alpha\}\}$$

3.2 Endpoint Snapping:

The distance between each of the path endpoints and resolved endpoints is taken. If the distance is within a threshold, the lines would be snapped at the endpoints. This would cover endpoint pruning as well.

Previous works[1] were able to achieve snapping to inner parts of the resolved paths. This can be trivially achieved by marching along the resolved path with the marching steps as the new end points and re-using endpoint snapping.

$$let\ LP\ =\ \{x \mid x\ \in\ AllPrimitives \wedge x\ is\ st.\ line\}$$
$$let\ \delta\ =\ 0.005\ (min.\ snapping\ dist.)$$
$$k\ = new\ classified\ straight\ line.$$
$$\begin{aligned} let\ X\ =\ \{x.start,\ x.end \mid s.t.\ & |x.start - k.start|\ < \delta\ or \\ & |x.start - k.end|\ < \delta\ or \\ & |x.end - k.start|\ < \delta\ or \\ & |x.end - \ k.end|\ < \delta\ \} \end{aligned}$$
$$k.x\ =\ min(x\ \in\ X \mid s.t.\ x\ < \delta\ )$$
$$k.y\ =\ min(x\ \in\ X \mid s.t.\ x\ < \delta$$

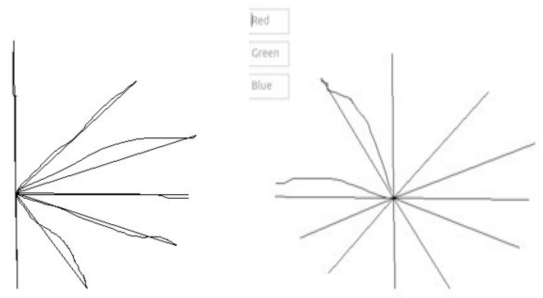

Figure 2.1, 2.2: ***End points snapped of line segments***

3.3 Axes Snapping

The slope of the drawn line and the principal axes are taken , if that angle falls within a threshold then the line is snapped to the axes.

```
let $\delta$ = slope((15 * $\pi$)/180) (min. angle)
let k = new classified straight line.
let X = $\{\pi, -\pi, \pi/2, -\pi/2\}$
x = min (x s.t.
        |atan(slope(x)) − atan(slope(k))|)
k = rotate(k, x − atan(slope(k)))
```

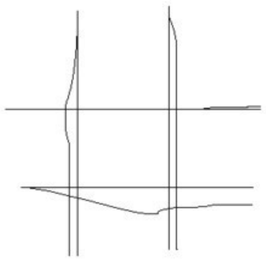

Figure 3: ***Line segments snapped to Axes***

3.4 Snapping Line with Circumference

If the distance between one of the endpoints of a line segment and the circumference of the circle is within a threshold, then that point which lies on the circumference and is nearest to the line segment is considered as one of the endpoints of the new line segment (which is to be drawn). This snaps the line segment to the circumference of the circle.

```
let $P_{circle}$ = circle in consideration
if euclidDist($P_{circle.circumference}$, point) < $\delta$ :
    point = $Line_{circle.centre, point}$.intersection($P_{circum}$)
```

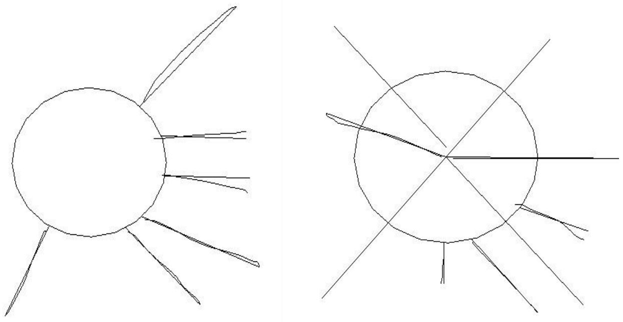

Figure 4.1, 4.2: ***Line Segments snapped to Circumference and Centre of the Circle***

3.5 Line Parallelism / Parallel Line Snapping

We compare the angle between two line segment paths with the angle needed to satisfy the parallelism constraint.

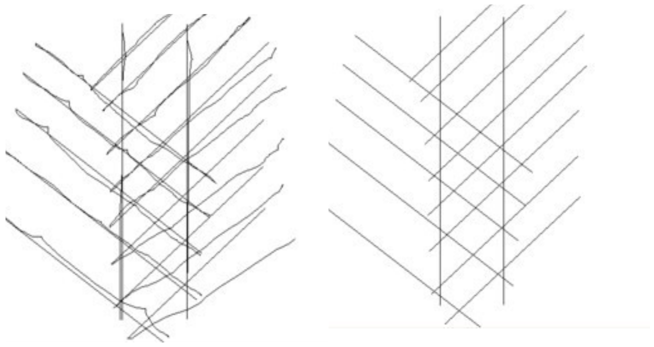

Figure 5.1, 5.2: ***Line segments - parallel to each other***

```
let LP = $\{x \mid x \in AllPrimitives \wedge x \text{ is st. line}\}$
let $\delta$ = slope((15 * $\pi$)/180) (min. angle)
let X = $\{slope(x) \ \forall x \in LP\}$
let k = new classified straight line.
x = min (x s.t. |slope(x) − slope(k)|)
k = rotate(x, slope(x) − slope(k))
```

3.6 Concentric Circles

The centre of the two circles are taken into account, if the distance between those two centres are within a threshold then the centres are snapped and the concentric circles are formed.

```
let O = AllPrimitives
let $O_{circle}$ = $\{x \mid x \in O \wedge x \text{ is a circle}\}$
selected = nil
least = inf
for c in $O_{circle}$ :
    dist = euclidDist(newC.centre, c.centre)
    if dist < least :
        least = dist
        selected = c
if euclidDist(selected.centre, newC.centre) < $\delta$ :
        newC.centre = selected.centre
```

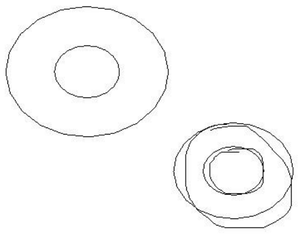

Figure 6: ***Centres of the circles snapped to make them concentric***

3.7 Nesting constraints

These constraint checks are further nested[3], in an elegantly recursive manner. These nested constraint checks lead to intuitively better sketches, for instance, a line segment might be needed to snap to the centre of the circle as well as to some other line's endpoint.

For this, every constraint result is associated with clamps, which prevent movement of the sketch for the case that the constraint is applied. The clamp structure used in the demonstration , for instance, remembers whether the endpoint(which) is fixed, the slope is fixed and if the circle's centre is fixed, among others.

Taking this approach involved changing all the previous constraint checking methods to work around the clamps applied by previous transformations. Each transformation returned a score ,which directly relates to how close the transformation into the cleaned object is to the originally drawn path.

This is implemented by a recursive call to the same constraint-checking family method, while passing in all the clamps before the current constraint until a depth(5, in this case).

Scores are returned with each of the result, and the best result is returned.

Pseudocode:

```
def apply(points, depth, clamps) :
    choices = []
    choices.append(parallelConstraint(points, clamps))
    choices.append(axesConstraint(points, clamps))
    choices.append(pointSnapping(points, clamps))
    if depth == 5 :
        return best choice according to score
    children = []
    for c in choices :
        new = modify points according to c
        children.append(apply(new, depth + 1, c))
    return best child from children acc. to score.
```

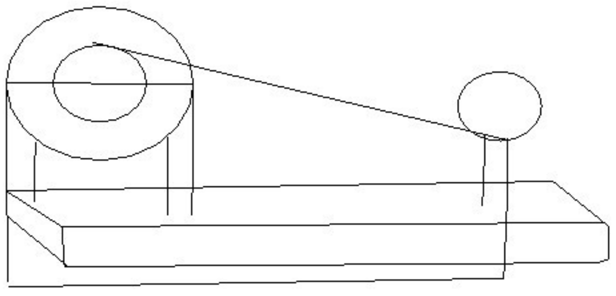

Figure 7: ***Complex sketches possible due to recursive constraints.***

4 Related Work

In the paper by Hse et al.[3] more primitives were added, which seem to be too complex for a simple sketch smoother. The primitives include parallelograms, cubes and regular hexagons. Their preprocessor does a good job of breaking the drawn sketch into the already understood primitives.

The publication by Fiser[1] had a special supplementary section devoted to the experimentally found best results for all the thresholds and the weight of the corresponding scores for each constraint checking. While this seems more tuned for simple sketching, a more comprehensive approach would be to be build a better model to adapt to the suggestions of the users and retrain on the fly.

5 Future Work

- The process explained until now should work when the user adds simple strokes one at a time. For instance, he'll need 4 strokes to create a rectangle. This isn't good enough. To tackle this, the hand drawn path will go through a preprocessor, that will chunk the path into simpler primitives, and give the necessary information for each path(like length of the path.). To do this, it would go through the path drawn and keep drawing tangents at each point. If the slope changes abruptly, we consider that as a break in the path. Now for all these chunks, we feel that all the rules should be applied combinatorially. The best result might be deduced from experimentation.

- Implement 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.

- Select a sketch

- Delete a sketch component

- Edit,reposition and resize a sketch

6 Conclusion

A tool that will help a user to beautify the messy hand-drawn sketches and provide functionalities like save file clear all etc.

7 References

[1]Fišer, Jakub, Paul Asente, and Daniel Sýkora. "ShipShape: a drawing beautification assistant." Proceedings of the workshop on Sketch-Based Interfaces and Modeling. Eurographics Association, 2015.

[2]Murugappan, Sundar, Subramani Sellamani, and Karthik Ramani. "Towards beautification of freehand sketches using suggestions." Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling. ACM, 2009.

[3]Hse, Heloise Hwawen, and A. Richard Newton. "Recognition and beautification of multi-stroke symbols in digital ink." Computers & Graphics 29.4 (2005): 533-546.