Interactive Vectorization

Jun Xie
University of Washington
junx@uw.edu

Holger Winnemöller
Adobe Systems, Inc.
hwinnemo@adobe.com

Wilmot Li
Adobe Systems, Inc.
wilmotli@adobe.com

Stephen Schiller
Adobe Systems, Inc.
schiller@adobe.com

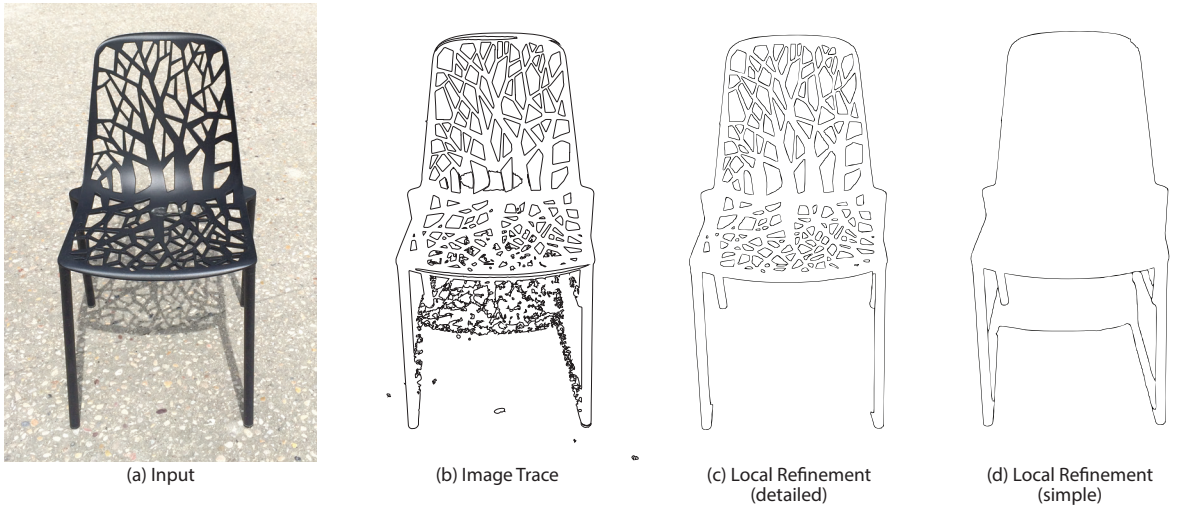

Figure 1. Automatic versus interactive vectorizaton. Given an input image (a), adjusting the global parameters of the Image Trace tool in Adobe Illustrator produces a vectorization that captures the detail in the chair back and seat at the expense of unwanted noise in the shadow region. Our split, merge and smooth tools allow users to locally refine the output to produce the desired result. With just a few minutes of interaction, we are able to create both detailed (c) and highly simplified (d) vectorizations of the chair.

ABSTRACT

Vectorization turns photographs into vector art. Manual vectorization, where the artist traces over the image by hand, requires skill and time. On the other hand, automatic approaches allow users to generate a result by setting a few global parameters. However, global settings often leave too much detail/complexity in some parts of the image while missing important details in others. We propose interactive vectorization tools that offer more local control than automatic systems, but are more powerful and high-level than simple curve editing. Our system enables novices to vectorize images significantly faster than even experts with state-of-the-art tools.

ACM Classification Keywords

I.3.6 Methodology and Techniques: Interaction techniques; I.4 Image Processing and Computer Vision; I.4.6 Segmentation

CHI 2017, May 06 - 11, 2017, Denver, CO, USA

ISBN 978-1-4503-4655-9/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3025453.3025872

Author Keywords

image vectorization; user interaction; image analysis

INTRODUCTION

Most digital images today are represented either as *raster* or *vector* graphics. Raster graphics map well onto imaging sensors, computer memory, and display devices. On the other hand, vector graphics can represent visual data in a more compact and resolution-independent manner that supports artifact-free scaling. Moreover, most vector representations can be directly and conveniently modified by manipulating control points and visual parameters. For these reasons, vector graphics are a popular choice for signage, clip art, logos, scalable UI widgets, and animated 2D graphics.

There are two main approaches for creating vector graphics. The first is to manually define curves and regions using a vector drawing tool, either working from scratch or tracing over a reference image. While this workflow offers complete control over the final result, it requires a lot of time and expertise to create high quality content. The other approach is to convert a raster image directly into vector format by automatically *vectorizing* the image. Automatic vectorization can produce high fidelity approximations of raster images and is much faster than manual drawing. However, especially for natural images,

the vector output is often overly complex and inconvenient for subsequent editing operations. For example, the automatic vectorization in Figure 1b contains a lot of small, noisy marks in the shadow below the chair. While modifying vectorization parameters (e.g., overall level-of-detail) can help, such global controls are often not sufficient; filtering out unwanted noise in one part of the image may also remove semantically important details elsewhere. As a result, most automatic vectorization results still require tedious manual clean up that is sometimes more costly than directly defining curves and regions.

Given these limitations of both manual and automatic vectorization workflows, producing vector content from input images can be a very time-consuming undertaking, even for highly experienced graphic designers. Moreover, the task is simply out of reach for most novice users, who may want to create simple scalable graphics for a web page, Facebook invite, or t-shirt design.

To address this problem, we present a system that allows users to *interactively* vectorize a raster image. Our key idea is to incorporate high-level user feedback to guide the vectorization, while still leveraging automatic techniques to generate vector elements whenever possible. Specifically, our system provides a set of interactive, local refinement tools that allow the user to add, remove, and refine vector elements by brushing over parts of the image. Unlike standard manual vector editing tools, our refinement operations take into account the underlying image data (color, gradients, contour maps) to interpret the user interactions. This allows users to quickly perform many modifications with very rough brush strokes.algorithms and enables more specific, controlled modifications.

To support this workflow, our system first analyzes a given input image to construct a hierarchical graph structure of closed contour regions based on all the detectable edges in the image. This data structure represents a set of candidate edges that the user can potentially add to the system. Brushing on the image locally adjusts an edge threshold that determines which edges to add or remove. In addition, the user can directly draw custom edges or modify the shape of existing edges. Our system ensures that all edges are part of closed regions, which allows us to reduce most adjustments to just two region-based operations (*Split* and *Merge*), thereby minimizing the number of user tools and simplifying the editing process.

We used our system to produce a variety of vector outputs across many common image categories. The majority of these results required ten minutes or less of interaction time, most of which involved rough brushing to add or remove detail from various parts of the image. Findings from an exploratory user study suggest that our interactive refinement tools are accessible to novice users and more efficient than existing automatic vectorization tools for creating clean vector outputs.

RELATED WORK

Automatic Vectorization Image Vectorization is a long standing problem in Computer Graphics. Most approaches focus on automatic vectorization and maximum visual fidelity; that is, being able to describe image shapes and gradients as accurately and with as few primitives as possible. Currently, derivatives of Gradient Meshes [17] and Diffusion Curves [13] are capable of producing highly faithful reproductions of input images [19, 7, 18, 9]. Even with modern systems, visual fidelity comes at a cost. Representing intricate image detail requires a large number of control points that are difficult to manipulate in the final artwork. In addition, extracted vector shapes commonly follow isocontours of colors, which tend not to coincide with semantic object contours. Finally, automatic systems do not provide local control over the level-of-detail, which often results in too many outlines in some regions and too few outlines in others.

Interactive Vectorization It is generally easier to allow the user to guide the vectorization process interactively than to fix up an automatic result without guidance. For example, Active Contours [8] and Intelligent Scissors (aka Live Wire or Magnetic Lasso) [11] snap user drawn curves to strong image contours. However, they require the user to completely draw along all lines to be vectorized. While such approaches are useful for indicating a few object boundaries in input images, fully indicating all desired exterior and interior contours of objects quickly becomes burdensome to the user. Several works aid a user in sketching a source photograph [10, 16, 20]. While these methods identify salient edges in an image for the user to trace, they are more concerned with producing a good looking line-drawing, rather than easily editable and meaningful vector regions.

Image Segmentation & Matting Interactive segmentation techniques like Graph-cut [3], and GrabCut [15] similarly address the boundary detection problem. But while these methods focus primarily on isolating a foreground object from its background, vectorization often requires selecting and outlining many individual shapes. In such scenarios, selecting positive and negative examples for graph-cut, or accurately selecting a Grabcut region for every vector shape to be extracted becomes very tedious. While automated segmentation algorithms have also been proposed, they suffer from the same aforementioned limitations as automatic vectorization techniques. Our own work can be seen as a carefully designed set of tools and interaction techniques to enable convenient, high-level control of image segmentation for the purpose of vectorization. Concretely, our work is based on the Ultra Metric Contour Model (UCM) by Arbelaez et al. [1].

Scope Richardt et al. [14] discuss how to extract the parameters for semi-transparent gradient layers of a vector image. Image segmentation and semantic annotation is completely manual. In our work, we are only concerned about the vector shapes, not the acquisition of color parameters (we simply assign the average color of a region). Similarly, we employ simple path-tracing and Bézier-fitting for the actual vectorization of our contour graph, as these aspects are not the focus of our interactive technique. Unlike special purpose approaches, such as vectorization of scanned line drawings [5, 12], we assume photographs of arbitrary scenes as input.

METHOD

System Overview

Given an input image, our system first performs a brief pre-processing computation to identify the complete set of potential edge candidates. Once this step is complete, the system displays a faded version of the source image overlaid with the current selected set of edges. We allow users to refine the edges via two types of adjustments: a *global* parameter that determines the threshold of currently visible edges, and a set of *local* tools that manipulate the visibility and structure of edges in a spatially-varying manner. We also provide a slider for the source image opacity and navigation controls for panning/zooming the canvas.

Contour Generation

To facilitate interactive refinement of edges for a given image, we first construct a hierarchical data structure that supports both global and local adjustments. The hierarchy is represented as an *edge map* — i.e., an image corresponding spatially to the original image, but where the value of each pixel represents the strength of an edge at that location. The edge map is structured to preserve two key properties:

Closed regions. For any given threshold value λ, i.e. a global detail level, the set of edges whose corresponding pixels in the edge map is greater than λ will produce a set of contours that form a planar map [2] (i.e., partition the plane into closed regions). We represent the planar map as an undirected *Contour Graph* $\mathscr{G} = \{\mathscr{N}, \mathscr{E}\}$, where the nodes $\mathscr{N}$ represent junctions of three or more contours (i.e., T-junctions) and the edges $\mathscr{E}$ represent either portions of a region boundary that connect adjacent T-junctions or entire closed paths without T-junctions.

Hierarchical Organization. Regions are organized hierarchically such that, as we decrease λ, the new contours that appear subdivide the regions generated by higher threshold values. At the top level of the hierarchy (largest λ), there is a single region defining the boundary of the entire image, which gets progressively subdivided into increasingly detailed regions (with lower and lower edge strengths) as we move down the hierarchy.

To generate this hierarchical edge map, we first compute a raw edge strength map using the *Structure Edges* (SE) algorithm proposed by Dollár et al. [4]. The values in this edge map do not preserve the properties defined above. For example, since the edge strengths vary continuously, a given value of λ may select partial edges that do not form closed regions. Thus, we process the SE result using the Ultrametric Contour Map (UCM) algorithm [1], which modifies the edge map to produce a hierarchical structure of closed regions. Note that UCM also ensures that all edges are one pixel wide, so that there is only one path connecting any two adjacent T-junctions in the map.

While the combination of SE and UCM generally yields a robust and meaningful image decomposition into contour regions [1], we observed that the SE/UCM algorithm sometimes produces superfluous regions around small image structures;

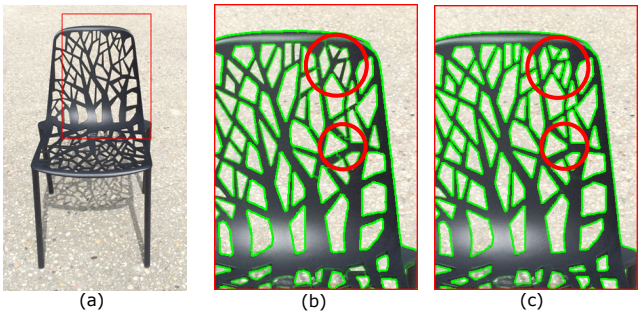

Figure 2. Ultrametric Contour Maps before and after local gradient re-weighting. (a) Original source image. (b) Overlay of original UCM. (c) Overlay of UCM after gradient re-weighting. Note the noisy/missing edges in (b) compared with the edges in (e)

the SE in those regions tend to be weak and noisy due to the ambiguity between structure and texture at small scales. In these situations, we re-weight the SEs with the local image gradient (with a 3×3 support), as the smaller kernel size is better at resolving small image elements. Fig. 2 shows the resultant UCM before and after re-weighting.

All the editing operations described below are carried out on the *Contour Graph*. These edits rely on three additional properties that we store on each edge E_{ij} of the graph: the binary variable $v_{ij} \in \{0,1\}$, is 1 when the E_{ij} is visible and 0 when E_{ij} is hidden; s_{ij} is the strength of E_{ij}; and $\mathbf{p}_{ij}$ are all the 2D pixel locations along the edge.

Global Editing

The Ultrametric Contour algorithm assigns each edge path in $\mathscr{G}$ a uniform edge strength, in which higher values indicates stronger edges. Thus, global editing is achieved by adjusting the global edge weight, λ, so that edges with strength larger than the threshold are visible. As a result, a lower threshold presents more detailed edges over the entire image. By changing the global threshold values, the user can adjust the level of detail over the entire image.

Local Editing

The intended result of a vectorization often contains edge-contours spanning several, possibly non-continuous, and locally varying strength levels, so that a single global threshold parameter does not suffice to extract these edges. Additionally, like any edge detection framework, our SE+UCM approach is liable to produce jagged edges in noisy areas. To address these issues, we supply three tools to locally modify the visibility and structure of the *Contour Graph*: The *Split* tool, the *Merge* tool, and the *Smooth* tool; all of them implemented as brushes with adjustable sizes.

Split

The Split brush allows the user to put a rough stroke around regions where they want to add an underlying edge by lowering the edge visibility threshold. Since we work with closed regions represented by edge contours, a user could create additional regions conceptually either by "creating sub-regions in this area with finer detail" (focus is on *regions*), or by "creating a cut along this edge to split this region in two" (focus is on

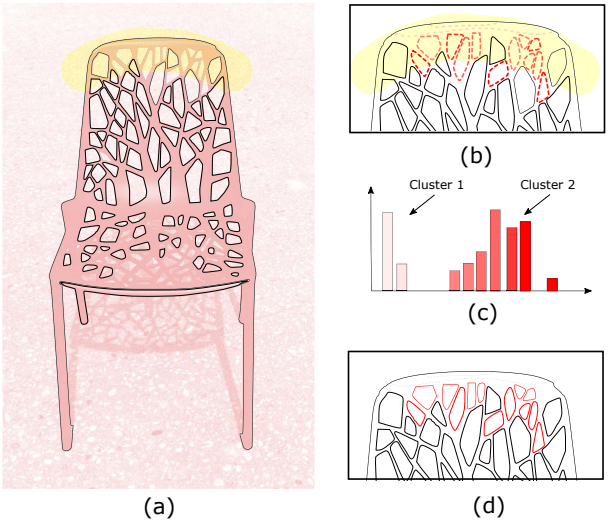

Figure 3. System's response to *Paint* action. (a) Overlay of visible edges and user's Split brush (yellow). (b) Currently hidden edges (red). (c) Histogram of edges in brush region. (d) Visible edges after lowering local threshold.

edges). In practice we found both mental models being used, and we could have made two different brush tools for each of them. To simplify the user interface and interaction, we provide only one Split brush, and the two types of operation are triggered by different interaction behaviors. The *region* approach is triggered by a *Paint* action where a user brushes broadly over an area, while the *edge* approach is triggered by a *Cut* action where the user follows fairly accurately along a desired edge. We first describe how the two actions work before explaining how we automatically infer which action to execute based on a given user interaction.

When the user applies a Split brush on a region, our system initially identifies all candidate edges with respect to either the *Paint* or *Cut* behavior.

For the *Paint* behavior, we collect all the hidden edges $\mathbf{E}^{iv} = \{E_{ij} | v_{ij} = 0\}$ with at least one pixel under the brush region in the graph. Our goal is to divide this set of edges into two clusters based on their edge strengths, assuming that a cluster of similar-strength edges might be semantically related. We use K-Means with $K = 2$ to effect this clustering. We then choose edges whose strengths fall into the cluster with higher values, corresponding to the hidden edges with higher saliency, as the *Paint* candidates, denoted as $\mathbf{E}^{p}$, and illustrated in Fig. 3.

For the *Cut* action, we start with the set of edges $\mathbf{E}^{p}$ defined above, and then leverage the graph's connectivity by selecting a set of connected edges that are aligned with the user's input stroke. Specifically, we describe the user's input stroke by a list of N polyline points in 2D: $\mathbf{c}_{1:N} = [c_1, ..., c_N]$. Let $\mathbf{M}$, the *stroke map*, be a Distance Transform [6] of a binary image of the current user stroke. In other words, the value of a pixel p in $\mathbf{M}$ is the distaance from p to the nearest pixel marked by a user stroke. We assign the weights to E_{ij} as

$$w_{ij} = -(s_{ij} + \sigma \sum_{t \in \mathbf{p}_{ij}} \mathbf{M}(t)) \qquad (1)$$

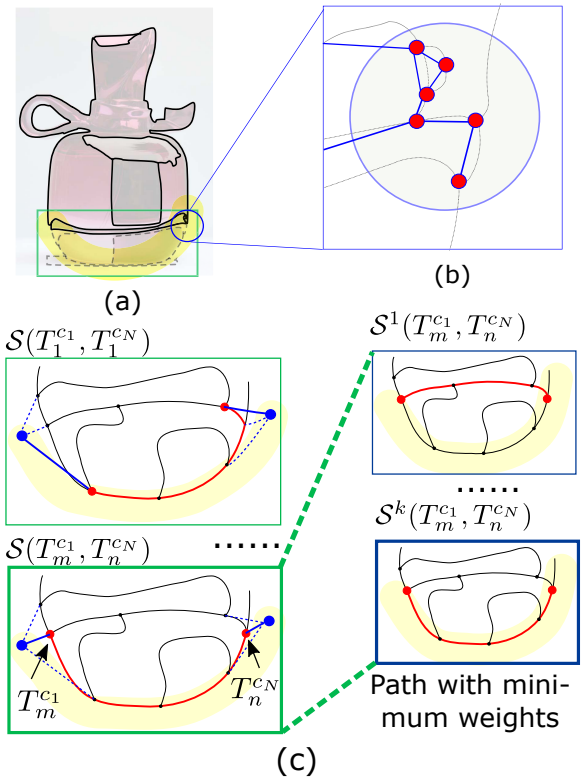

Figure 4. System's response to *Cut* action. (a) Overlay of visible edges and user's Split brush (yellow). (b) Closeup of $\mathcal{G}$ with nodes (red), visible edges (blue), hidden edges (grey). (c, left) Two potential start/end locations (red dots) given the start/end locations (blue dots) of user stroke. (c, right) Two possible paths for the same start/end positions.

Here, s_{ij} is the edge strength based on UCM contour intensity. The second term, associated with the *stroke map*, measures the shape similarity between the edge and user's input. The constant σ balances between edge fidelity and shape matching costs. We set $\sigma = 5.0$ in our implementation. The weights are only updated for the hidden edges. We set the weight of the rest of edges as *infinite*. (These weight changes can be considered as only persisting during the following computations. Final weights are assigned after the intended behavior is identified and the corresponding action performed.)

Given the user stroke $\mathbf{c}$, we first search for the nearest T-junctions that are close to the starting point (c_1) and ending point (c_N) of the user's input stroke within a distance r (we set $r = 40$ empirically) in the *Contour Graph* $\mathcal{G}$. The resulting T-junction sets are denoted $\mathbf{T}^{c_1}$ and $\mathbf{T}^{c_N}$. Based on the above edge weights, we compute a shortest path $\mathcal{S}(T_m^{c_1}, T_n^{c_N})$ between any of the two T-junctions $T_m^{c_1}$ and $T_n^{c_N}$, which denote the m^{th} and n^{th} node in set $\mathbf{T}^{c_1}$ and $\mathbf{T}^{c_N}$, respectively. We select the shortest path that minimizes the following matching cost as the edge candidates $\mathbf{E}^{t}$ for the *Cut* action:

$$d(m,n) = \gamma(\|c_1 - T_m^{c_1}\|^2 + \|c_N - T_n^{c_N}\|^2) + \sum_{\substack{(i,j) \in \\ \mathcal{S}(T_m^{c_1}, T_n^{c_N})}} w_{ij}. \qquad (2)$$

The first part of Eq. 2 penalizes the distance between the starting and ending points of user's input and the optimal path. The second part leverages the shape matching cost of

the shortest path, which is defined as the sum of weights along the path. γ adjusts the contribution of each component and we set $\gamma = 1.0$ in our implementation. We denote d_{MIN} as the minimum cost. Fig. 4 illustrate how we compute the candidate edges for the *Cut* action.

Now we have computed the candidate edges $\mathbf{E}^p$ and $\mathbf{E}^t$ for the *Paint* and *Cut* actions respectively. To infer the actual behavior intended by the user, based on their input stroke, we employ the following heuristics: (1) For *Paint*, as many of the intended edges as possible should be covered by the user's brush, and conversely, as much of the user's brush as possible should be occupied by the intended edges. (2) For *Cut*, the curve shape of intended edges should be similar and close to the user's input stroke. We define a score for each behavior as follows:

$$score(Paint) = exp(-\alpha \cdot (1 - \frac{2 \cdot O_{BE} \cdot O_{EB}}{O_{BE} + O_{EB}})), \quad (3)$$

$$score(Cut) = exp(-\beta \cdot d_{MIN}). \quad (4)$$

α and β are tuned parameters that are set empirically. In our implementation, $\alpha = 1$ and $\beta = 0.02$. $O_{BE} = |\{\mathbf{E}^t | \mathbf{E}^t \in \mathbf{c}\}|/|\mathbf{E}^t|$ stands for the brush coverage on the edges (percent of edge pixels inside the brush region) and $O_{EB} = |\{\mathbf{c} | \mathbf{c} \in \mathbf{E}^t\}|/|\mathbf{c}|$ denotes the edge coverage on the brush (percent of input stroke points inside the edge region, which is formulated as local disks around the edge pixels).We predict the behavior with larger score as the user's intended behavior. Candidate edges with respect to that behavior will be selected as the final edges to be shown to the user. We thus lower the local threshold of candidate edges with respect to that behavior and make those edges visible.

Freehand Behavior. We also allow the system to snap the user's brush stroke to underlying image features, which have higher edge probability when there is no explicit UCM edge that can be identified. The freehand mode will be enabled either automatically when we cannot detect reasonable edges for the *Paint* and *Cut* actions (i.e. scores are too small), or manually when the user presses the SHIFT key during brushing.

Merge

The *Merge* brush is designed as the dual operation of *Split*. It merges two regions by removing edges that separate these regions. In testing, we found that both the region-centric and edge-centric cognitive models used the same interaction gesture — a simple crossing of the edge to be removed. The region-centric interpretation is "connect these two regions", whereas the edge-centric interpretation is "cross-out (delete) this edge". For this reason, only one merge behavior is defined. To select the edges to merge, we use the same clustering strategy as above, but this time the candidate edges are the lower value cluster of visible edges (as opposed to the higher value cluster of hidden edges). We then increase the local threshold to hide the corresponding edges.

Post Split And Merge Operations

Since the *Split* operation adds edges, it may introduce edges which are not part of a closed region. Similarly, the *Merge* brush will delete edges, possibly leaving remnants that are not part of a closed region. Thus, after both operations we need to clean up the *Contour Graph* to ensure that all regions are closed. This is done by simply hiding all such orphan edges by adjusting the local edge weight.

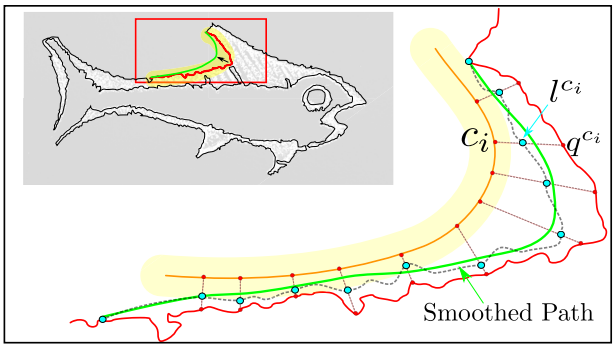

Figure 5. Smooth operation. Top left: Overlay of noisy edges (red), user's smoothing brush (yellow), and the resultant smoothed edges (green). Bottom right: User's brush constrains the closest correspondences to be pulled towards the input stroke (cyan dots), while the general shape of the edges is still preserved (gray, dash curve). We then subdivide the optimized corresponding points based on B-Spline constraint to obtain a smoothed curve (green).

We also mark any edges that are modified through *Split* or *Merge* operations, so that if the global threshold, λ, is altered, the visibility of those edges will not be overwritten.

Smoothing

We also allow users to locally smooth edges by interactively overdrawing them with the *Smoothing* brush.

For the smoothing operation, our system first searches for and highlights a linear sequence of edges in $\mathscr{G}$ that match the user's brush strokes. The polyline points representing the chosen edges are modified to both be close to user's input strokes, and to retain the high frequency shape variation of original edge. Last, we fit a B-spline to the polyline points to smooth them, sampling the B-spline to replace the original points. All of this is repeated in real-time, with display updates, as the user adds stroke points.

To find the graph edges to be smoothed in an online manner we use the following algorithm. We denote the user's sampled stroke points as $s_i, i = 1...,H...$, where H is the current number of stroke points, and increases each time a new point is added. For each new stroke point, s_H, we first update the stroke map $\mathbf{M}$, as well as the weight of visible edges according to Eq. 1. We also find the closest polyline point to s_H. To keep the ensuing notation compact we denote this closest polyline point as a function of the index of the H-th stroke point: $\eta(H)$.

When the user adds the H-th stroke point, we find the closest T-junction to $\eta(H)$ in the *Contour Graph* $\mathscr{G}$, denoted $q^{\eta(H)}$. With the closest T-junction $q^{\eta(H-1)}$ to the previous $q^{\eta(H-1)}$, computed as before, we compute a shortest graph path $\mathscr{S}(q^{\eta(H-1)}, q^{\eta(H)})$ between the points. If $\eta(H-1) = \eta(H)$, or if $q^{\eta(H)}$ is too far from $\eta(H)$, we just ignore $\eta(H)$ and proceed until the next stroke point meets our requirements. The shortest path $\mathscr{S}(q^{\eta(H-1)}, q^{\eta(H)})$ along with the previously computed paths $\mathscr{S}(q^{\eta(1)}, q^{\eta(2)})$, ... $\mathscr{S}(q^{\eta(H-1)}, q^{\eta(H)})$ form the optimal path, $\mathscr{S}(q^{\eta(1)}, q^{\eta(H)})$. As a result, the polyline from $\eta(1)$ to $\eta(H)$ will match the user's strokes reasonably well. Since some $\eta(H)$

are eliminated, as mentioned above, we may not end up with H points, but only N. Unused stroke points are dropped from the list s_i.

Once we have the above path through $\mathscr{G}$, we compute a subsampled list of points from the polylines of that path. The subsampling is done by choosing every K-th point, where K is set to $min(10, N/3)$. Our smoothing optimization uses this list of subsampled points, denoted $\mathbf{d}_i, i = 1, N$. Note that these will be in 1:1 correspondence with a set of sampled stroke points, $\mathbf{s}_i$. The smoothing operates by moving the $\mathbf{d}_i$ towards their corresponding $\mathbf{s}_i$, while retaining the variation of the originally traced edges. We use an energy minimization formulation:

$$E(\mathbf{l}^{1:N}) = E_{guide} + \theta \cdot E_{var}. \quad (5)$$

The parameter θ weights the influence of the two terms. The optimization produces a list of N points $\mathbf{l}^{1:N}$ as the modified location of the $\mathbf{d}^{1:N}$.

The first term E_{guide} encourages the alignment between the stroke points $\mathbf{s}_i$ and the correspondences polyline points $\mathbf{d}^{1:N}$:

$$E_{guide}(\mathbf{l}^{1:N}) = \sum_{i=1}^{N} \|\mathbf{l}^i - \mathbf{d}_i\|^2. \quad (6)$$

The second term preserves the variations along the original edge path:

$$E_{var}(\mathbf{l}^{1:N}) = \sum_{i=2}^{N} \|(\mathbf{l}^i - \mathbf{l}^{i-1}) - (\mathbf{d}^i - (\mathbf{d}^{i-1})\|^2. \quad (7)$$

Once the optimal positions $\mathbf{l}^{1:N}$ are obtained, we perform B-Spline sampling along the optimized correspondences, replacing the original edge path with the curve anchored by the subdivided points, and update $\mathscr{G}$ accordingly. The sampling rate is related to the smoothing brush radius, with larger radius resulting in a smoother curve (Fig. 5).

RESULTS

We used our system to vectorize a variety of input photos across several common image categories, including professional product shots, landscapes, interior scenes, hand-drawn artwork, and logos. We focused on creating clean vector outputs that separate semantically distinct parts into individual regions, rather than more photo-realistic results that capture detailed appearance variations due to illumination and texture. Such clean vectorizations are useful for many graphic design tasks (e.g., creating stylized illustrations of real-world objects and acquiring editable representations of logos and lettering in the wild) but are typically hard to generate using automatic tools. Figures 1, 6, 7, and our supplemental materials show a subset of our results, and Table 1 summarizes the pre-processing time, distribution of split, merge and smooth operations, and the total user interaction time.

While generating these vectorizations, we found that global adjustments alone were never sufficient to obtain the desired result. To illustrate this, Fig. 6 compares the result after global adjustment with the final result after local refinement for five input images. Fog in the mountain scene reduces the contrast between successive layers of terrain, which results in a "floating" peak in the middle of the globally adjusted result. In the shirt image, obtaining detailed edges in the logo results in unwanted noise. For the fruit and flower images, illumination effects combined with the relatively uniform reflectance properties of the subject(s) make it hard to automatically extract the edges between individual apples or petals.

Image	Pre (s)	Splt	Mrg	Sm	Man	Undo	Usr (s)
mountains	22	**18.5**	**13.5**	**8.5**	**4**	**17.5**	**443**
shirt	54	**15**	**6**	**12.5**	**2.5**	**7**	**263**
chair	54	45	41	0	2	20	363
fruit	19	0	38	34	8	4	372
flower	48	5	5	19	5	3	168
fish	15	13	65	68	4	29	1011
perfume	14	26	29	30	2	30	606
car	30	56	57	29	10	29	884
keyboard	22	75	78	42	10	39	840
piechart	15	30	60	60	2	49	609
coffee	25	41	21	80	35	21	898
logo	98	2	19	59	7	4	366

Table 1. Summary of statistics from results. We report the pre-processing time (Pre), the number of split (Splt), merge (Mrg), smooth (Sm), manual split (Man), and undo (Undo) operations, and the total user interaction time (Usr) for each image. The top two examples are from our user study, and the bold numbers are median values from the logged usage data. The next five images are shown throughout the paper. The final five images are in our supplemental materials.

Our interactive local refinement tools enabled us to quickly add or remove the relevant edges in different parts of the image to achieve the desired result. None of our results required more than 20 minutes of user interaction, and most took 10 minutes or less to create. Moreover, the process of interactive refinement facilitates the exploration of design variations. Fig. 1 shows two vectorizations of a chair, and Fig. 7 shows three versions of a perfume bottle.

USER STUDY

To investigate how our approach facilitates the vectorization process for novices, we conducted an exploratory user study. A key design challenge for our evaluation is identifying a valid baseline against which to compare our system. The most obvious choice are existing commercial vector graphics tools, many of which have specific vectorization features. However, these tools tend to be complex, hard to learn, and primarily targeted at expert users. Informal pilot studies indicated that it would be extremely difficult for novices to complete even simple vectorization tasks with such expert tools. Thus, rather than comparing our system against novices using commercial tools, we felt that a more interesting and ecologically valid baseline for comparison is experts using existing vector tools for vectorization tasks. The performance of such experts defines both an approximate lower bound for the amount of time required to vectorize a given image and a gold standard for the quality of the vectorization result.

To achieve a meaningful comparison between novices using our system and experts using existing vector tools, we ran a two-part study. We asked 12 vector graphics novices to produce a target output from two input images (*mountains* and *shirt* from Fig. 6). We also asked seven experts to vectorize the same two input images using their tool(s) of choice. Novices

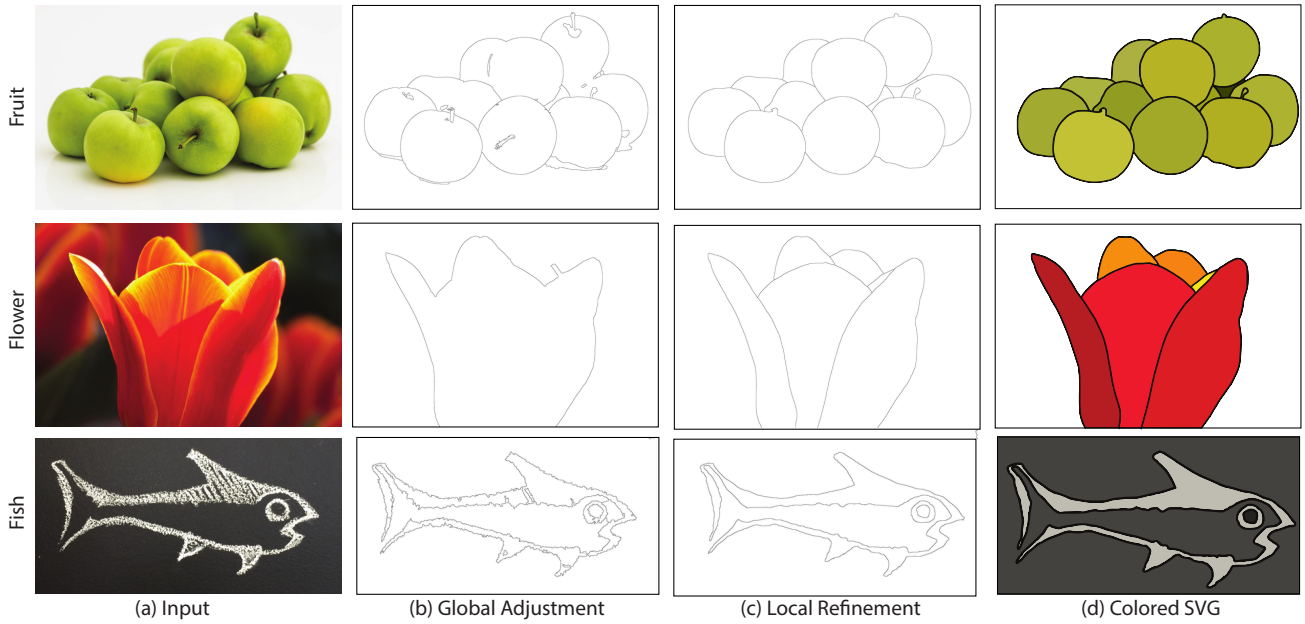

Figure 6. Summary of results. For each example, we show the input image (a), the intermediate result after global adjustment (b), the final result in our system (c), and exported in SVG format with colors averaged per region (d).

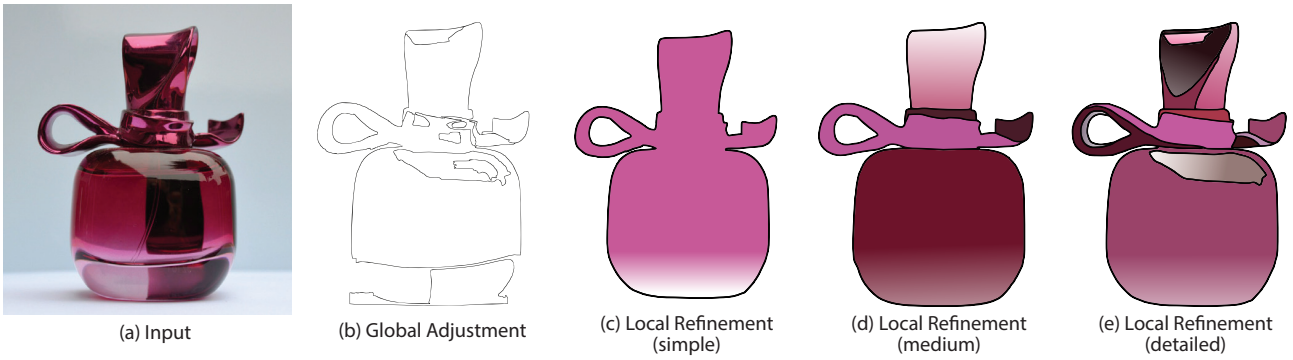

Figure 7. Variations. We used our system to generate three vectorizations of a perfume bottle with increasing amounts of detail. We manually applied simple gradients to color each closed region. [If this figure does not show gradients, please use Acrobat Reader for viewing.]

and experts were identified via a recruiting survey where people self-rated their vector graphics experience on a scale from 1 (no experience) to 5 (use vector graphics all the time). We defined those with ratings of 3 or lower as novices. Amongst the remaining respondents, we selected seven experts who have extensive graphic design experience in vectorizing images. Six of the seven experts have performed image vectorization in professional settings (e.g. converting line-art into vectors). To reduce variability in the experimental setup, we picked experts who are very familiar with Adobe Illustrator (5–20 years of usage), an industry-standard vector graphics tool with an automatic vectorization feature called *Image Trace*. All novices and three of the experts were recruited from a large software company. The other experts were recruited via the Internet. Two novices and two experts were female.

Images

We chose two very different images for our study. The *mountains* image is a landscape photograph with many smooth gradients, and the *shirt* image focuses on a crisp logo (see Figure 8). To define the target vectorized output for the tasks, we asked two professional designers to manually trace a set of outlines over each image with the goal of extracting a "*meaningful*" representation of the image "*that could then be used in a graphic design or similar composition.*" To make it easier for us to identify the drawn outlines, we gave the designers printed versions of the images and instructed them to trace with a dark pen or pencil. The results from the experts were qualitatively similar. To minimize the task time, we chose the traced result with the fewest number of individual outlines as the target vectorization for each image. Figure 8 shows the two selected targets (one from each designer), and the supplemental materials include all four traced images.

Methodology

For the novice sessions, we first guided participants through a warmup task on a simple input image to introduce our system. We then asked them to perform two vectorization tasks (in

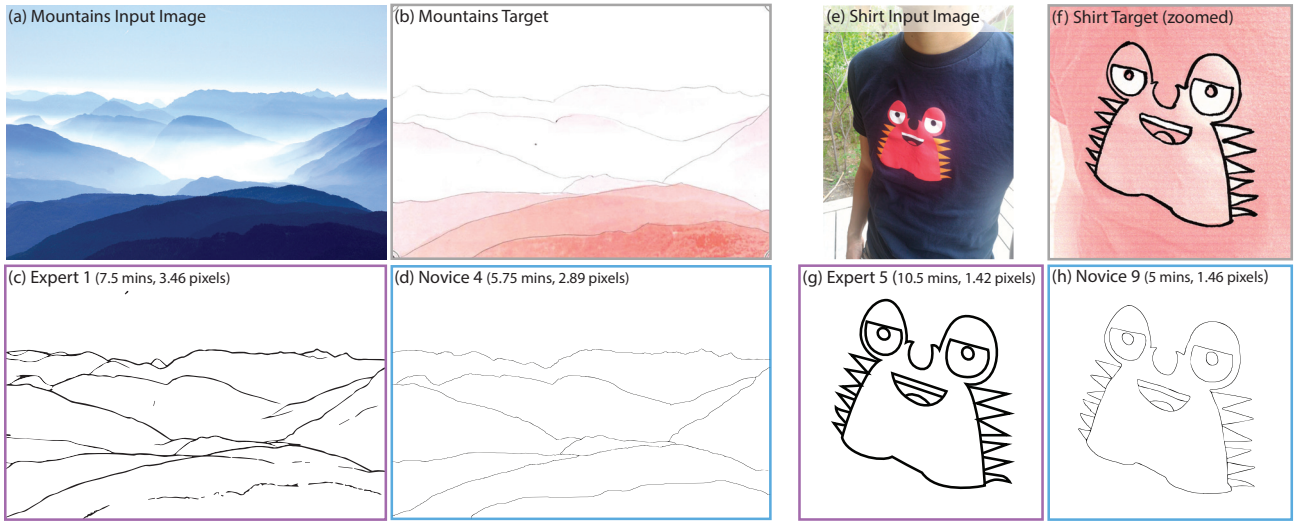

Figure 8. Sample user study results. We show the input (a, e), target (b, f), lowest error expert (c, g), and novice (d, h) results from our user study.

randomized order) as quickly as possible. For each task, we left the target vectorized output open in a separate window for reference. We also displayed a short *cheat sheet* listing the various elements of our user interface with the relevant keyboard shortcuts (see supplemental materials). We did not talk or answer any questions during the task, and the participant informed us when they were done. For the expert sessions, we asked participants to use their typical workflow to vectorize the images as quickly as possible. The remaining setup was identical to the novice session. Due to time constraints, each expert only worked with one of the two images.

For all sessions, we observed the workflows and measured task completion times. In addition, for the novices, we recorded tool usage logs and at the end of each session, asked participants to fill out an exit form (Figure 9) and discuss what they liked and disliked about our system. To measure the similarity between each user-generated result I and the corresponding target output J, we compute the distance (in pixels) from each outline pixel in I to the nearest outline pixel in J and vice versa. The mean distance d over all the outline pixels approximates the dissimilarity, or error, between I and J. Since the input images have similar resolutions (*mountains*:1024x683, *shirt*: 768x1024), the scale of the errors is comparable across both tasks. To limit the impact of outliers on d, we clamp the individual distance values to 20 pixels. In other words, we equally penalize each outline pixel that is at least 20 pixels away from any corresponding outline. Note that d can be considered a variant of the Hausdorff distance, which is a classic measure of inter-shape similarity.

Findings

Here, we summarize our key findings regarding task performance, common workflows, and qualitative feedback from participants.

Task Performance

Figure 10 summarizes the task performance of the novices and experts by plotting each user-generated result based on completion time t (x-axis) and similarity to the target d (y-axis). We found that novices generally took less time than the experts to complete the tasks. For the *shirt* task, 10 of the 12 novices completed the task more quickly than the fastest expert, and for the *mountains*, 9 novices were faster than the fastest expert. For both images, the median completion time for novices was less than the fastest expert, and no novice took more time than the slowest expert.

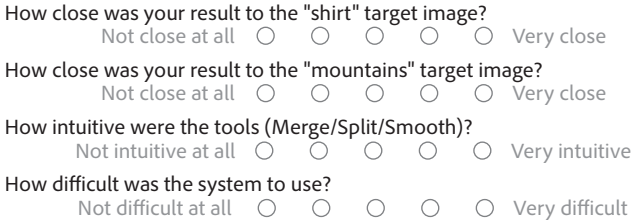

Figure 9. Exit form for novice session.

In addition to speed, we consider how well participants completed the tasks (i.e., the error of their results with respect to the targets). On this front, the most proficient experts achieved slightly better (lower) errors than the median novice for both images. However, it is worth noting that overall, the novice results were very similar to the targets. For the *shirt*, all novice results were within 0.85 pixels of the best expert result, and for the *mountains*, all but one of the novice results were also within 0.85 pixels of the best expert result. In the *shirt* results, we noticed that 9 of the novices accidentally left a few extraneous outlines in regions of the image away from the logo. Since most users zoomed into the logo while working, these stray edges were not visible within the viewport. Aside from these artifacts (which would be easy to clean up using the Merge brush) the outlines of the logo are generally very close to the target in all the novice results, as indicated by the relatively low error values. All novice and expert results for both images are included in the supplemental materials.

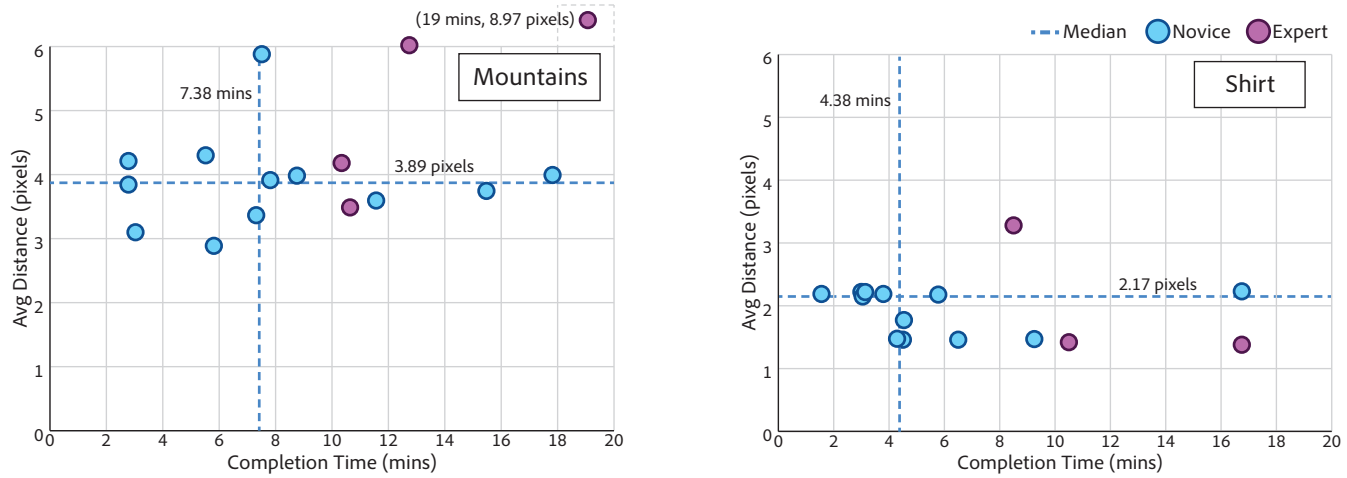

Figure 10. Analysis of vectorization results. The scatter plots show the completion time and error of novice and expert results for the two input images in our study. Novices are generally faster and achieve a comparable amount of error compared to the experts.

Workflow

Most novices and experts adopted a similar high-level strategy: adjusting global vectorization parameters first and then switching to low-level refinement to achieve the target output. However, there were notable differences in how participants executed these two phases of the workflow.

Global Adjustments. Compared to novices, most experts spent far more time performing trial-and-error adjustments to global settings — i.e., the Image Trace parameters — oscillating between settings that provide too much or too little detail in various parts of the image. In contrast, novices generally spent far less time adjusting the global threshold parameter before moving on to the refinement phase. We note that the two experts who produced the highest error results for each image relied mainly on Image Trace and did not do much additional refinement. In addition, one expert decided to manually draw the lines on the input image after first experimenting with Image Trace, while two others immediately decided to draw without using Image Trace at all.

Removing Edges. Most novices and experts adjusted global settings to produce a result with too many edges. To remove extraneous edges, experts used tedious manual refinement strategies, such as combining multiple paths into a single closed region and selecting, splitting, and deleting all the relevant (partial or complete) paths. Novices were able to remove unwanted edges much more efficiently by using our Merge brush with a coarse-to-fine approach. They started by roughly brushing over large regions to merge away most edges and then performed more careful brushing in specific regions where necessary.

Adding Edges. In general, experts did not add new edges to the Image Trace results. While the three experts who manually drew lines were very proficient, it still took on the order of minutes to trace a single path of moderate complexity. With our system, most novices adopted a similar coarse-to-fine strategy for adding edges as they did for removing edges. That is, they first brushed very loosely and quickly with the Split brush to see if the system automatically added the desired edges. Generally, such interactions produced the desired result. If not, brushing a bit more carefully often produced the right result. Finally, in a small number of scenarios, novices resorted to the manual Split brush to explicitly add an edge.

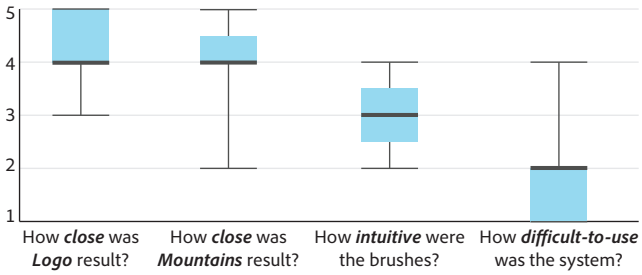

Figure 11. Exit questionnaire results.

Adjusting Edge Geometry. While Illustrator provides tools for automatically smoothing and refining path geometry, the experts did not use these features (perhaps in order to complete the tasks more quickly). In contrast, with our system, novices used the Smooth brush extensively to beautify edges, typically toward the end of an editing session. In addition, many novices used a combination of Split and Merge operations to refine the shape of edges in a more controlled manner, which is a technique that we demonstrated during the warmup task.

Tool Distribution. The median tool usage statistics (bold rows in Table 1) indicate that novices performed a much larger proportion of Merge operations with the *mountains* image compared to the *shirt*. Since the *shirt* had more high contrast edges, most users were able to quickly add the desired edges with the Split brush. In contrast, some Split operations on the *mountains* image would not produce the desired target outline due to the smooth gradients. Thus, participants ended up using more Merge or Undo operations to refine the result.

Qualitative Feedback

Figure 11 summarizes how novices responded to our exit questionnaire. Overall, they felt able to achieve vectorization results that were close to the targets, which aligns with our

quantitative error analysis (Figure 10). In addition, novices generally agreed that our system was not very difficult to use. While the ratings for the "intuitiveness" of the Merge, Split, and Smooth tools were more neutral, the open-ended responses during the exit interview suggest that this was in part due to some specific and easy-to-address usability issues. Several participants noted that the terms *Merge* and *Split* were unintuitive and suggested alternative tool names like *Remove/Add Edges*. Others pointed out that the shortcut keys for the tools were not intuitive.

In general, novices liked the simplicity of our system. They appreciated the small number of tools, which made the system "easy to use,""clean," and (despite the neutral ratings in the exit form) "intuitive" to many participants. Several novices commented specifically on the Merge and Split tools, saying that they found those operations to be "fun," "fast," and "appealing" for adding and removing edges in the image. One user explained that the reason he did not start each task by manipulating the global threshold parameter was that he felt that it would be just as efficient to quickly add and remove edges with the Merge and Split tools.

The most consistent negative feedback was around the Smooth brush. In particular, the fact that the system tries to select and smooth a single connected path of edges between T-junctions sometimes causes the brush to attach to an unexpected portion of the drawing. In some cases, when there are relatively few T-junctions near the brushed region, the system may not select any portion of the path at all. While users do receive visual feedback of the selected paths via real-time highlighting, they often did not understand why they were not able to smooth a specific portion of the drawing despite several attempts to brush over the region. Aside from smoothing, two users mentioned that the global threshold slider seemed a bit slow. In our prototype implementation, it sometimes takes about half a second for the edges to update after the slider moves.

Discussion

Both the observed behavior and qualitative feedback from novices suggest that the split-and-merge paradigm at the heart of our approach is accessible and learnable, even for users who do not have significant vector graphics experience. It seemed that many users simply equated splitting and merging operations with adding and removing edges, respectively. That said, there were a few isolated cases where users expressed surprise at the result of a Split or Merge operation. However, even if their mental model of the underlying representation was not entirely accurate, the fact that they were generally able to work around this suggests that their level of understanding was sufficient for practical purposes.

One somewhat surprising result from our study is how well the novices performed both in terms of the speed and quality of their results in comparison with the experts. That said, it is important to note that our experimental design does *not* perform a direct, controlled comparison between our tool and existing automatic vectorization techniques. Thus, our completion time data certainly does not represent conclusive evidence that our interactive vectorization system is more efficient than Adobe Illustrator. Rather, the goal of our expert sessions was to provide a reasonable, ecologically valid baseline for the given vectorization tasks, and we believe the findings from our study provide a strong indication that our interactive refinement tools make it significantly easier to produce clean, vectorization results from input images.

CONCLUSION

As discussed above, the performance of our interactive vectorization system in the user study, and the users' responses to our system were highly encouraging. But as always, there is room for improvement.

The system is still susceptible to noise artifacts in the vicinity of small structures. Additionally, we know of some images where strong edges are not detected by our system. We traced the latter back to the specific training data used to train the SE classifier (Segmentation Dataset and Benchmark, BSDS 500), which also contained examples of similar false negatives. As such, the the system could potentially be enhanced by improving the training data. An interesting and obvious approach would be to re-train the SE classifier with user-guided vectorization results from our system. The original training set contained mostly natural images. It would be feasible to train an edge detector specifically for other types of input images: design graphics, line art, sketches, etc.

Currently, smoothing modifies the edge geometry in our hierarchical Contour Graph, which may have unintuitive consequences for subsequent edits. Copying modified edges to preserve the original edges is a simple fix for this issue.

Our decision to only support closed contour regions is a double-edge sword. On the one hand, it imposes a much cleaner vector structure than the *vector soup* of other approaches. It also allows for small, gesture-like user actions to operate intuitively on large edges or multiple edges at once — for the most part. However, in some cases, especially for small or very complex regions, the result of an edit operation can have a larger effect than anticipated. Luckily this did not prove to be a major burden for our users, but we would like to ensure a level of locality to edits, where appropriate. Another consequence of the closed-contour requirement is that we cannot have isolated edges, such as might occur at surface marks or cusps. Currently, these are handled by our system as very thin closed regions. In the future, these could be represented as special leaf-nodes of our graph representation. Our system currently only assigns a single color per region. In the future, more complex appearances could be captured by converting regions into Diffusion Curves [13] or Gradient Meshes [9].

We believe that interactive vectorization is successful because it leverages the source image's data while the user is manipulating the vector data. This concept could and should be extended. In the future, we will investigate what type of data should be attached to the final vector graphic objects to enable continued high-level editability while retaining backward compatibility with existing vector specifications.

REFERENCES

1. Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. 2011. Contour Detection and Hierarchical Image Segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 5 (May 2011), 898–916. DOI: http://dx.doi.org/10.1109/TPAMI.2010.161
2. P. Baudelaire and M. Gangnet. 1989. Planar Maps: An Interaction Paradigm for Graphic Design. In *Proceedings of SIGCHI 1989*. ACM, 313–318. DOI: http://dx.doi.org/10.1145/67449.67511
3. Yuri Boykov, Olga Veksler, and Ramin Zabih. 2001. Fast Approximate Energy Minimization via Graph Cuts. *IEEE Trans. Pattern Anal. Mach. Intell.* 23, 11 (Nov. 2001), 1222–1239. DOI:http://dx.doi.org/10.1109/34.969114
4. Piotr Dollár and C. Lawrence Zitnick. 2013. Structured Forests for Fast Edge Detection. In *ICCV*. International Conference on Computer Vision. DOI: http://dx.doi.org/10.1109/ICCV.2013.231
5. Jean-Dominique Favreau, Florent Lafarge, and Adrien Bousseau. 2016. Fidelity vs. Simplicity: A Global Approach to Line Drawing Vectorization. *ACM Trans. Graph.* 35, 4, Article 120 (July 2016), 10 pages. DOI: http://dx.doi.org/10.1145/2897824.2925946
6. Pedro F. Felzenszwalb and Daniel P. Huttenlocher. 2004. *Distance Transforms of Sampled Functions*. Technical Report. Cornell Computing and Information Science.
7. Peter Ilbery, Luke Kendall, Cyril Concolato, and Michael McCosker. 2013. Biharmonic Diffusion Curve Images from Boundary Elements. *ACM Trans. Graph.* 32, 6, Article 219 (Nov. 2013), 12 pages. DOI: http://dx.doi.org/10.1145/2508363.2508426
8. Michael Kass, Andrew Witkin, and Demetri Terzopoulos. 1988. Snakes: Active Contour Models. *International Journal of Computer Vision* 1, 4 (1988), 321–331. DOI: http://dx.doi.org/10.1007/BF00133570
9. Yu-Kun Lai, Shi-Min Hu, and Ralph R. Martin. 2009. Automatic and Topology-preserving Gradient Mesh Generation for Image Vectorization. *ACM Trans. Graph.* 28, 3, Article 85 (July 2009), 8 pages. DOI: http://dx.doi.org/10.1145/1531326.1531391
10. Alex Limpaecher, Nicolas Feltman, Adrien Treuille, and Michael Cohen. 2013. Real-time Drawing Assistance Through Crowdsourcing. *ACM Trans. Graph.* 32, 4, Article 54 (July 2013), 8 pages. DOI: http://dx.doi.org/10.1145/2461912.2462016
11. Eric N Mortensen and William A Barrett. 1995. Intelligent Scissors for Image Composition. In *Proceedings of the 22nd annual conference on Computer Graphics and Interactive Techniques*. ACM, 191–198. DOI:http://dx.doi.org/10.1145/218380.218442
12. Gioacchino Noris, Alexander Hornung, Robert W. Sumner, Maryann Simmons, and Markus Gross. 2013. Topology-driven Vectorization of Clean Line Drawings. *ACM Trans. Graph.* 32, 1, Article 4 (Feb. 2013), 11 pages. DOI:http://dx.doi.org/10.1145/2421636.2421640
13. Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, and David Salesin. 2008. Diffusion Curves: A Vector Representation for Smooth-shaded Images. *ACM Trans. Graph.* 27, 3, Article 92 (Aug. 2008), 8 pages. DOI: http://dx.doi.org/10.1145/1360612.1360691
14. C. Richardt, J. Lopez-Moreno, A. Bousseau, M. Agrawala, and G. Drettakis. 2014. Vectorising Bitmaps into Semi-transparent Gradient Layers. In *Proceedings of the 25th Eurographics Symposium on Rendering (EGSR '14)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 11–19. DOI: http://dx.doi.org/10.1111/cgf.12408
15. Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. 2004. "GrabCut": Interactive Foreground Extraction Using Iterated Graph Cuts. *ACM Trans. Graph.* 23, 3 (Aug. 2004), 309–314. DOI: http://dx.doi.org/10.1145/1015706.1015720
16. Qingkun Su, Wing Ho Andy Li, Jue Wang, and Hongbo Fu. 2014. EZ-sketching: Three-level Optimization for Error-tolerant Image Tracing. *ACM Trans. Graph.* 33, 4, Article 54 (July 2014), 9 pages. DOI: http://dx.doi.org/10.1145/2601097.2601202
17. Jian Sun, Lin Liang, Fang Wen, and Heung-Yeung Shum. 2007. Image Vectorization Using Optimized Gradient Meshes. *ACM Trans. Graph.* 26, 3, Article 11 (July 2007). DOI:http://dx.doi.org/10.1145/1276377.1276391
18. Tian Xia, Binbin Liao, and Yizhou Yu. 2009. Patch-based Image Vectorization with Automatic Curvilinear Feature Alignment. *ACM Trans. Graph.* 28, 5, Article 115 (Dec. 2009), 10 pages. DOI: http://dx.doi.org/10.1145/1618452.1618461
19. Guofu Xie, Xin Sun, Xin Tong, and Derek Nowrouzezahrai. 2014b. Hierarchical Diffusion Curves for Accurate Automatic Image Vectorization. *ACM Trans. Graph.* 33, 6, Article 230 (Nov. 2014), 11 pages. DOI: http://dx.doi.org/10.1145/2661229.2661275
20. Jun Xie, Aaron Hertzmann, Wilmot Li, and Holger Winnemöller. 2014a. PortraitSketch: Face Sketching Assistance for Novices. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 407–417. DOI: http://dx.doi.org/10.1145/2642918.2647399