





Supplement: Image Vectorization via Gradient Reconstruction

Souymodip Chakraborty¹ , Vineet Batra¹, Ankit Phogat¹, Vishwas Jain¹, Jaswant Singh Ranawat¹
Michal Lukáč¹ , Matthew Fisher¹  and Kevin Wampler¹ 

¹Adobe Systems

1. Color-Difference-Based Segmentation

We segment the pixels in $\bar{\mathcal{D}}$ based on color differences, computed in the CIELAB color space to effectively capture perceptual variations. This process yields a segmentation S along with a corresponding set of constant fill functions $F \subset \mathcal{K}_0$. The set F is represented as a mapping $S \rightarrow \mathcal{K}_0$. For any segment s , the fill function $F(s)$ denotes the constant color of the segment.

The algorithm starts with an initial segmentation S , where each pixel is treated as an individual segment. The corresponding fill functions F for each segment are initialized as constant fill functions, with values set to the color of the respective pixel, i.e.,

$$F(p) = \text{lab}(I(p)) \quad \text{for a pixel } p.$$

The segmentation is refined iteratively by merging neighboring segments if the color difference between their respective fill functions is below a predefined threshold τ_s . This threshold τ_s serves as the criterion for determining color equivalence between segments.

ALGORITHM 1: Color-Difference-Based Segmentation

Input: $\bar{\mathcal{D}}$: set of pixels, τ_s : merging threshold.
 $S \leftarrow \bigcup_{p \in \bar{\mathcal{D}}} \{p\}$; // Initialize each pixel as a segment
 $F \leftarrow \bigcup_{p \in \bar{\mathcal{D}}} \{\text{lab}(I(p))\}$; // Initialize fill functions

while true do
 $S' \leftarrow S$; // Create a copy of current segments
 for $s \in S$ **do**
 $s^* \leftarrow \underset{s' \in \text{nbh}(s)}{\text{argmin}} \|F(s) - F(s')\|_2$; // Find closest neighbor
 if $\|F(s) - F(s^*)\|_2 \leq \tau_s$ **then**
 merge(S', s, s^*); // Merge segments
 $F(s \cup s^*) \leftarrow \text{mean}(F(s), F(s^*))$; // Update fill
 end
 end
 if $S = S'$ **then**
 break; // Stop if no changes
 end
end
Output: S ; // Final segmentation

The output segmentation S consists of segments such that the

color difference between any two neighboring segments is greater than the threshold τ_s .

2. Discontinuity Aware Segmentation

The multicut constraint optimization problem defined in Equation (1) is addressed by iteratively processing the discontinuity pairs.

$$C^* \triangleq \underset{C \subseteq E}{\text{argmin}} \sum_{e \in C} w(e) \quad (1)$$

s.t. $\forall_{(u,v) \in \mathcal{A}}$, u and v is disconnected in $\mathcal{G}(V, E \setminus C, W)$

The procedure begins with an ordered list of discontinuity pairs and applies the min-cut algorithm to each pair, provided a valid path exists that connects the corresponding segments in set A .

For each pair, the algorithm evaluates the weighted sum of differences in size and color between the segments, which serves as the metric to guide the segmentation process. This iterative refinement ensures that segments are separated at points of significant discontinuity, optimizing the multicut objective. Finally, the connected components of the resulting graph are extracted to form the final segments. The detailed steps are described in Algorithm 2.

ALGORITHM 2: Discontinuity-Aware Segmentation

Input: $\mathcal{G} = (V, E, W)$; // Input graph with vertices V , edges E , and weights W
 $\mathcal{A} = (a_1, \dots, a_n)$; // Ordered list of discontinuity pairs
 $\mathcal{G}' \leftarrow \mathcal{G}$; // Initialize the working graph

for $(u, v) \in \mathcal{A}$ **do**
 while $\text{BFS}(\mathcal{G}', u, v)$ exists **do**
 $\mathcal{G}' \leftarrow \text{mincut}(\mathcal{G}', u, v)$; // Apply min-cut to separate u and v
 end
end
Output: $S \leftarrow \text{connected-components}(\mathcal{G}')$; // Extract the final segments

This algorithm ensures that segments are iteratively refined by separating pairs with significant discontinuities while maintaining the overall consistency of the graph. The connected components of the modified graph \mathcal{G}' represent the final segmentation.

3. Curve Fitting

Curve fitting simplifies region boundaries by converting pixel-based edges into structured paths. The process involves constructing an edge network, extending paths through junctions, and simplifying paths using line segments and Bézier curves.

3.1. Edge network construction

Boundaries between pixels of different regions are detected and connected in a counter-clockwise manner to form a poly-line tracing of each region. At each boundary, we have two (directed) edges, one for each adjacent region (see figure 1 left). Corresponding directed edges are merged and organized into a planar network of undirected paths between junction nodes, where a junction node is defined as any vertex with valence not equal to 2. See figure 1 (middle). To our advantage, this representation is gap-free by construction.

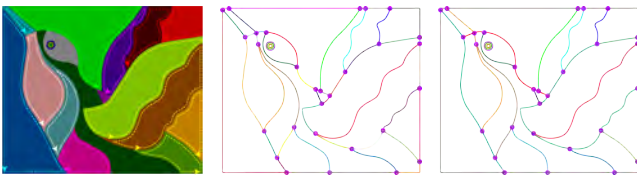


Figure 1: Poly-line tracing of few regions (left). Edge network (middle), extended path edge network (right). Extended path across junction have same colour.

3.2. Extend paths through junctions

We stitch together paths through each junction vertex by considering the tangent of incoming paths to each junction and greedily matching off pairs of paths whose tangent angle difference is closest to π , merging them into a single path extending through the junction vertex (similar to the approach taken in [KL11]). See figure 1 (right). This provides the opportunity (but not the requirement) that the curves be smooth across T- and other multi-way junctions. Whether the curves are actually G^1 or C^0 continuous at the junction will depend on the following path simplification step.

3.3. Path simplification

Each path in the vectorized image is up until now a polygon or poly-line of pixel-sized segments. This is approximated using a more user-friendly sequence of lines and Bézier curves using a parameter ϵ limiting how far the simplified path is allowed to differ from the input poly-line. In all examples in this paper we use $\epsilon = 1.5$ pixels. The path simplification algorithm consists of the following steps:

3.3.1. Select key vertices

To speed up later steps, a small subset of the vertices in the input are selected to serve as *key vertices*, where each line or Bézier in the simplified path must start and end at a key vertex. The key vertices are chosen by running the Douglas-Peucker algorithm with an error threshold of ϵ on the input and picking any vertex retained as a key vertex.

3.3.2. Calculate tangents

To simplify smoothly joining curves together in the final path, a tangent vector is computed at every key vertex. To compute a tangent vector at a key vertex with index i , we fit lines to the points in

the input path within the subrange from $i - k$ to $i + k$ for progressively larger values of k until the root-mean-square error between the best-fit line and the points in the subrange exceeds $\frac{\epsilon}{2}$. Any pair of curves with G^1 continuity will match this tangent at the key vertex where they join.

3.3.3. Soft corner detection

To get a high-quality result, it's important to get sharp corners in the right places. This involves determining if each key vertex should have C^0 or G^1 continuity. We have gotten the best results by performing a heuristic soft corner detection which gives each key vertex j a score $c(j)$ where $c(j) = -1$ enforces G^1 continuity at j , $c(j) = 1$ enforces C^0 continuity, and values in between represent ambiguous cases.

We determine $c(j)$ at each key vertex by fitting three shapes to the region of the path near vertex j : a line, a circular arc, and a "corner" shape consisting of a pair of lines meeting at a vertex. Each of these shapes is fit to the points in the input path from $j - k$ to $j + k$ for progressively larger values of k until the maximum distance between any point in the subrange and the best-fit shape exceeds ϵ . Let L , S , and C be the number of points used to fit the line, circle, and corner shapes respectively.

Intuitively, if C is larger than either L or S , then the corner-shape matches a larger portion of the curve near the junction vertex than either a circle or a line, and thus the vertex is more likely to be a corner. We express this with the following heuristic:

- If $L \geq S$: $c(j) = \frac{-L}{\max(L,C)}$
- If $L < S$:
 - If $S \geq C$: $c(j) = -1 \cdot f\left(\frac{S+1}{C+1}\right)$
 - If $S < C$: $c(j) = f\left(\frac{C+1}{S+1}\right)$

Where the function f ensures that the result is always in the range from -1 to 1 . We define f as:

$$f(x) = 1 - \frac{1}{1 + 5(x - 1)}$$

3.3.4. Path fitting with dynamic programming

We fit a path of connected lines and Bézier curves to the input poly-line using a dynamic programming algorithm inspired by [BLP10]. We begin by fitting an over-complete set of curves to the path consisting of both lines and cubic Bézier curves. This includes a line segment between each adjacent pair of key vertices, and four cubic Bézier curves for each pair of key vertices (both adjacent and not) for each of the four combinations of C^0/G^1 continuity at each endpoint. These Béziers are fit using the algorithm described by [Sch90], and any Bézier for which the maximum distance to one of the original poly-line curves exceeds ϵ is discarded. The resulting set of lines and Béziers is organized into a directed acyclic graph of curves connected at key vertices, all of which are guaranteed to lie within ϵ of the input path.

To pick a single path of connected curves, we assign a cost to each curve and to each connection between two curves. A standard dynamic programming algorithm is then used to solve for a simplified path with minimal sum of curve- and connection-costs. Our costs are heuristically defined as follows:

- At each connection j where two curves meet a cost of $J_{C^0}(j)$ or $J_{G^1}(j)$ is assigned based on if the curves meet with C^0 or G^1

Name	Size	Time(ms)	L1(CIELAB)	SSIM
2 (a)	1024x1120	2085	0.01041	0.9577
2 (b)	1024x655	1357	0.05546	0.7147
2 (c)	1024x683	1400	0.03859	0.7977
2 (d)	1024x683	1355	0.07173	0.7926
2 (e)	1024x1024	1833	0.02036	0.8978
2 (f)	1024x1024	1883	0.02521	0.8757
2 (g)	1024x1024	2189	0.04887	0.6750
2 (h)	1024x1024	2234	0.05495	0.6756
2 (i)	1024x1024	2167	0.04483	0.7587
2 (j)	1024x1024	2026	0.04359	0.7520
2 (k)	1024x1120	2076	0.01751	0.9206
2 (l)	1024x1120	2124	0.01578	0.9379
2 (m)	1024x1583	3403	0.04045	0.7005
2 (n)	1024x1444	2441	0.00680	0.9772
2 (o)	2048x2048	9379	0.007421	0.9602
2 (p)	2048x2048	9970	0.01631	0.9387
2 (q)	2048x2048	9780	0.02949	0.8818

Table 1: Evaluation of our method across different image sizes, assessed by execution time (milliseconds), color accuracy by average L1 loss in CIELAB, and structural similarity (SSIM).

continuity respectively. This is calculated based on the corner-ness score $c(j)$:

- if $c(j) > \frac{1}{4}$ the connection is assumed to be a likely-corner, so $J_{C^0}(j) = \frac{10}{1+g(c(j), \frac{1}{4})}$ and $J_{G^1}(j) = 10g(c(j), \frac{1}{4})$.
- if $c(j) < 0$ the connection is assumed to be likely-smooth, so $J_{C^0}(j) = 10 + 10g(-c(j), 0)$ and $J_{G^1}(j) = 0$.
- otherwise the connection is ambiguous, so bias toward G^1 continuity since it tends to look better in ambiguous cases: $J_{C^0}(j) = 10$ and $J_{G^1}(j) = 0$.

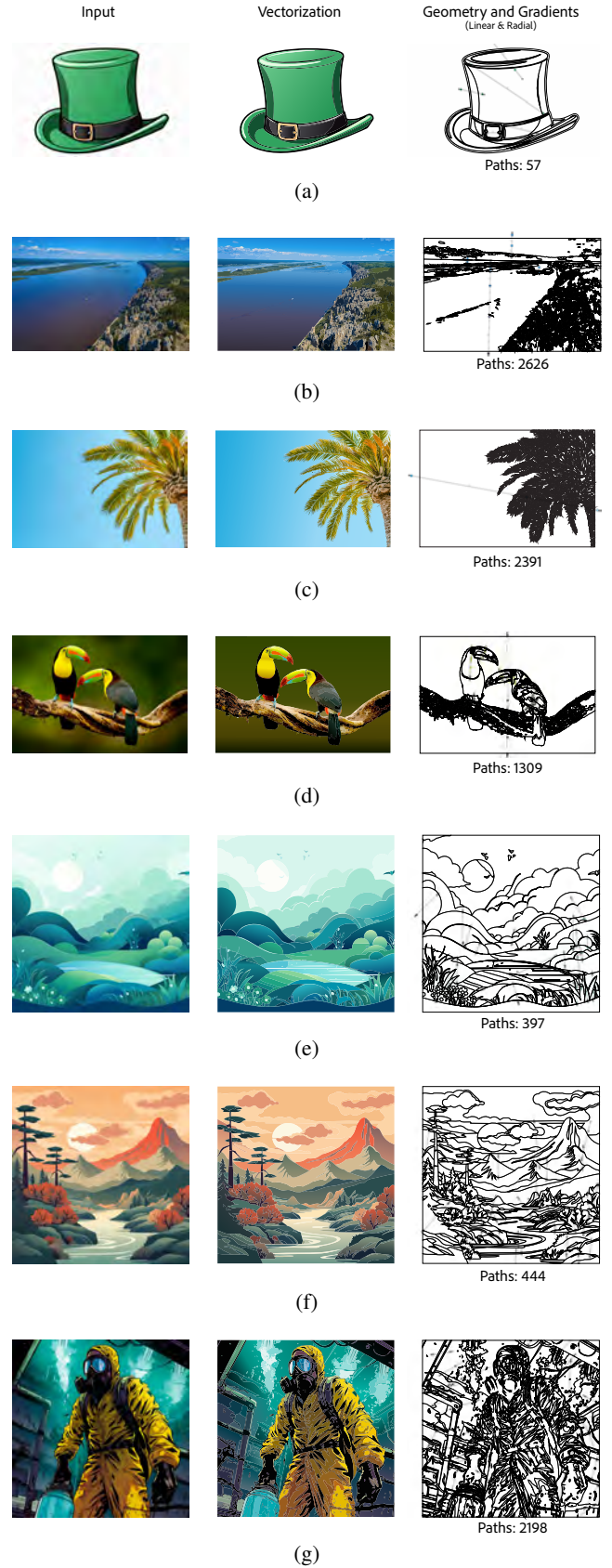
The function g above maps a corner-ness score (always between 0 and 1 since we use $g(-c(j))$ when $c(j) < 0$) into a cost compatible with the line and Bezier costs described in the next two subsections. Many functions are possible here, but the one we use is: $g(c, \alpha) = \frac{1}{1 - \min(\frac{c-\alpha}{1-\alpha}, 0.99)^2} - 1$.

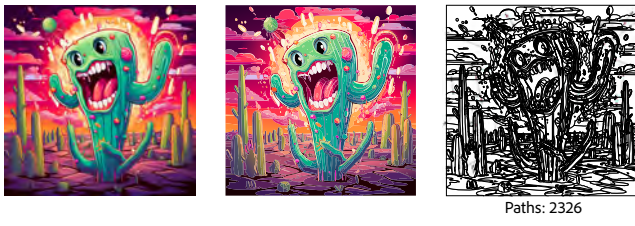
- Each line segment incurs a cost of $3.9 + \delta \cdot E$. Where E is the squared-error of the line's distance to the points to which it is fit and $\delta = 10^{-6}\epsilon$. The $\delta \cdot E$ term is included to disambiguate between paths with otherwise equal costs.
- Each Bézier incurs a cost of $4 + \delta \cdot E + \sum_j J_{G^1}(j)$ where the $\sum_j J_{G^1}(j)$ term accounts for the fact that a Bezier may pass over one or more intermediate key vertices, and should incur a cost of $J_{G^1}(j)$ for each.

The end result of all this is a path of connected curves which matches the input poly-line to within a distance of ϵ , uses a small number of curves, and intelligently chooses where to place C^0 corners based on global context of the input path.

4. Results

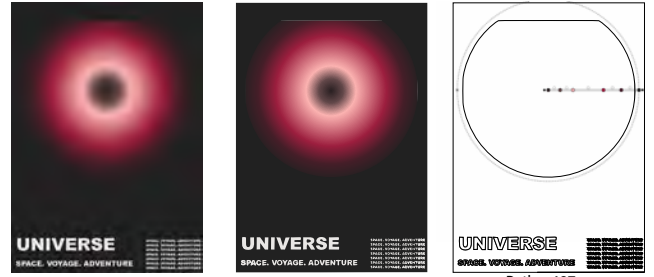
This section presents further evaluations of the proposed method, including quantitative metrics such as execution time, color accuracy (L1 in CIELAB), and structural similarity (SSIM) in Table 1. Additionally, visual results are provided, comparing input raster images, their vector reconstructions, and the underlying geometry.





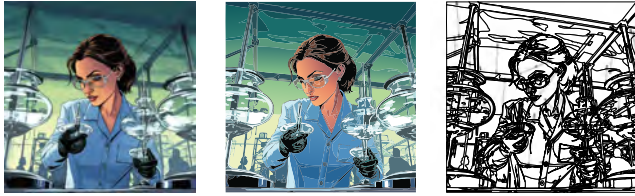
(h)

Paths: 2326



(n)

Paths: 487



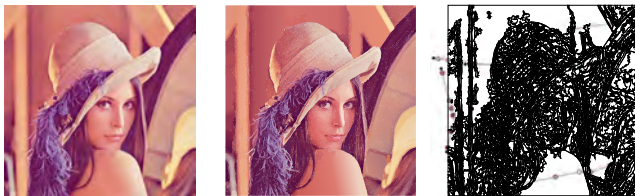
(i)

Paths: 1640



(o)

Paths: 117



(j)

Paths: 3536



(p)

Paths: 361



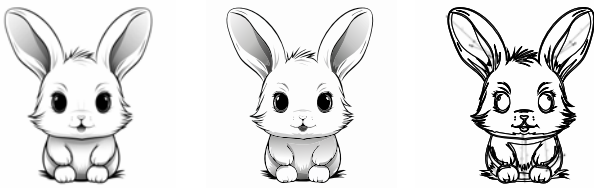
(k)

Paths: 221



(q)

Paths: 1674



(l)

Paths: 149



(m)

Paths: 3554

Figure 2: Results demonstrating our vectorization method on various input images. For each row: (left) original raster image, (middle) our vector reconstruction, and (right) the paths used in reconstruction.

References

- [BLP10] BARAN, ILYA, LEHTINEN, JAAKKO, and POPOVIC, JOVAN. “Sketching Clothoid Splines Using Shortest Paths”. *Computer Graphics Forum* (2010). ISSN: 1467-8659. DOI: [10.1111/j.1467-8659.2009.01635.x](https://doi.org/10.1111/j.1467-8659.2009.01635.x) 2.
- [KL11] KOPF, JOHANNES and LISCHINSKI, DANI. “Depixelizing Pixel Art”. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)* 30.4 (2011), 99:1–99:8 2.
- [Sch90] SCHNEIDER, PHILIP J. “An algorithm for automatically fitting digitized curves”. *Graphics Gems*. USA: Academic Press Professional, Inc., 1990, 612–626. ISBN: 0122861695 2.