CS 838: Advanced Graphics
Project 1: Stylized Editable Vectorizations

Jake Rosin
March 13th 2009

Abstract

The goal of this project was the creation of simple, recognizable and easily editable vectorized representations of real-world image inputs. Towards this goal I implemented a novel vectorization system using the combination of two preexisting, but relatively new, algorithms - one for image abstraction and simplification, one for segmentation and vectorization of abstract or cartoon images. I also experimented with methods of detecting occlusion and filling holes in the vectorized result, with a qualitative goal of simplifying the steps necessary to modify or delete regions within the image and produce a visually consistent result.

1 Image Abstraction

The first step in my method stylizes and abstracts the input image to a form better suited for vectorization. For this I use the shape-simplifying abstraction technique of Kang and Lee [2]. Their method involves the iterative application of the evolution equation

$$I_t = s \cdot \kappa || \bigtriangledown I|| \tag{1}$$

where

$$\kappa = \frac{I_x^2 I_{yy} - 2 I_x I_y I_{xy} + I_x^2 I_{xx}}{(I_x^2 I_y^2)^{3/2}} \tag{2}$$

$$s = (1 - r) + r \cdot |t(p) \cdot \bigtriangledown I(p)^{\perp}|. \tag{3}$$

The value κ is local isophote curvature, and comes from viewing the image as depth map. The term s allows parameter r to control the degree to which abstraction follows existing image edges. $t(p)$ is the tangent vector at pixel p in a smoothed vector field as described in *Coherent line drawing* [3].

By itself, this evolution function produces blurred and unimpressive images, but when combined with applications of a shock filter the results are much more striking. The shock filter is implemented as a 3×3 min, or max, filter. The min filter is applied when $(\triangle G_\sigma * I) > 0$; otherwise, the max filter is used.

The parameters acknowledged in [2] are the number of iterations of the evolution function, k (the number of iterations between applications of the shock filter), r and σ. Although implementation of this algorithm in Matlab took about a day, and another two to port the code to C++ for efficiency, additional time had to be spent determining values for the parameters that went unmentioned in the algorithm's description. The evolution function, for example, defines the derivative of the image with respect to time. As the image evolves its derivative is recomputed; the obvious question is how far to advance time between recomputations, making Δt a tunable parameter. I eventually settled on $\Delta t = 1$; a smaller time-step produces results which are only barely visually distinguishable (Figure 1) when k and the number of iterations are scaled to compensate, but the results shown in [2] are described only in terms of number of iterations; there is no mention of the time-step between them.

Another unexpected hurdle involved finding the right parameter values for the computation of $t(p)$. Two values needed to be tuned: the radius of the smoothing window, and the number of iterations before returning the smoothed result. After some experimentation I now use 4 iterations of the algorithm in [3], with a window radius that is usually around 3 or 4 pixels (the former is a constant in my code; using fewer

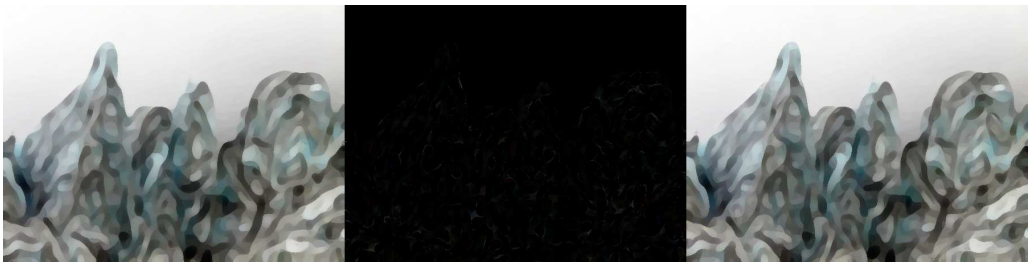

Figure 1: Left: the abstraction of an image of ice columns, with iters$= 40, r = 0, \sigma = 1, k = 10, \Delta t = 1$. Right: an abstraction of the same image, with iters$= 400, r = 0, \sigma = 1, k = 100, \Delta t = 0.1$. Center: the difference between the two results. Be sure to zoom in.

iterations results in noticable bias towards image noise over image edges, while using more significantly slows the computation).

The implementation of this stage of the project was finished at the week 3 milestone, including testing and parameter-setting.

2 Segmentation

I used a novel segmentation algorithm given in *Vectorizing Cartoon Animations* [7]. The approach described by Zhang et al. is specifically tailored towards the simple shapes and contiguous regions of modelable (if not constant) color found in cartoon animations and stills. Although they predict poor results if the algorithm is run on photographic images, my expectation was that the simplification and abstraction process would produce images "cartoony enough" for this approach to work well. Here is their segmentation method, in short:

1. Extract decorative lines [5] and mask those pixels, so they are not included in the segmentation.
2. Find image edges which were not accounted for in the first step; mask them as well.
3. For $r = r_{\max}$ to $r = 1$:
   (a) Simulate the movement of a "trapped ball" of radius r, constrained by the masked image pixels, to find a region of sufficient size. This is approximated using a fill operation, then erosion and dilation of size $r - 1$. Any contiguous blob of remaining pixels is considered to be a new region. Multiple regions are identified in this step, and the following steps are applied to all of them.
   (b) Fit an HSV color model to each region. Quadratic models are used by Zhang et al..
   (c) Expand each region by iteratively including the unlabeled neighbor pixel with the smallest error under the appropriate color model, until no pixels with error under some threshold can be found.
   (d) Include the pixels from all new regions in the set of masked pixels.
4. Control the number of resulting regions by removing any region with size under some threshold and adding its pixels to the neighboring region with the lowest-error color model. The final output is a list of decorative edge pixels (represented by color, width and location), a set of regions (each comprised of a contiguous blob of pixels) and a color model for each.

This approach differs from most other segmentation methods by treating detected edges as a hard constraint. The authors assert that this would result in inaccurate segmentations strongly influenced by noise

when applied to photographs, but that in cartoon images such edges only occur at legitimate region boundaries. The shock filter applied as part of the shape-simplification algorithm moves images towards this state as well; with repeated applications the image will move towards a piecewise constant result.

I hit a few snags implementing this algorithm. First, the algorithm does not have well-defined behavior for edge pixels. Although pixels detected as decorative lines are eventually represented by vectorized paths, other edge pixels which represent the division between two regions of different color will never be represented. In my implementation I added a step before regions are combined which assigns every previously masked edge pixel (not decorative line pixels) to the neighboring region with the lowest-error color model (ignoring threshold used in the similar step for region growing).

Another issue appeared due to the use of HSV color models. Regions with hue ≈ 0 would produce obviously incorrect reconstructions; eventually this problem was identified as the result of RGB to HSV conversion, which mapped visually indistinguishable values of red to dramatically different hues (hue being cycic with $0 \equiv 1$). Switching to explicit modeling of the RGB channels resolved the problem.

I also noticed that the quadratic model would occassionally overfit regions with image noise, resulting in poor reconstruction over most of the region. I eventually modified the implementation to explicitly chooses a constant or linear model, if such a model gives a low enough reconstruction error.

Since I didn't expect the abstracted images to have decorative lines of the type common in cartoon images, my implementation skips the first step. It thus gives poor results on genuine cartoon images, but works fine for the abstracted images I used as input.

3 Vectorization

Vectorizing the regions produced by the segmentation step proved more of a challenge, mostly because the vectorization method used is only glossed over by Zhang et al. (see section 5*C* of [7]). Here is the method I used:

1. Place sub-pixel "contour points" around the region boundaries of the segmented image. Contour points occur on the edges or corners of a pixel, in these circumstances:

   - The edge between two adjacent pixels from different regions.
   - A corner whose four surrounding pixels comprise at least three different regions (mark this as a curve endpoint).
   - A corner which is part of a horizontal or vertical edge (i.e., the four surrounding pixels have two total labels, and are divided into 1×2 horizontal or vertical rectangles.
   - A square corner: three of the four surrounding pixels have the same region label, and the one remaining is the tip of a 90 degree corner extending at least two pixels.

   Also place contour points around the boundary of the image itself.

2. Trace the boundary of each region to determine which contour points form its outline.

3. Find maximum curvature points; mark them as curve endpoints.

4. Fit a cubic bezier to each set of contour points such that its endpoints are points previously marked as such. If the approximation error is too high, add another endpoint (so that the set of points is represented by two cubic beziers). Repeat this until approximation error falls below an acceptable threshold.

5. Each region in the vectorization is represented by a set of curve sequences, where each set begins and ends at the same point. If the region has more than one set, then one is its outer boundary and the rest represent holes. Any curve aside from those forming the outer edge of the image will appear in the boundaries of exactly two regions. Such a curve will be duplicated exactly in its two occurrences (otherwise slight gaps might appear between regions).

Bezier curves are fit to the contour point sequences following the method described in *Capturing outlines of planar images using Bezier cubics* [4], with maximum curvature points selected using the algorithm described by Chetverikov in [1]. The method of tracing the contour points of region boundaries is based on the approach taken in [4] in the similar task of tracing boundary pixels, with modification for the slightly different problem setting (sub-pixel contours).

I used SVG as the output format, and spent some time learning it using an online resource [6]. SVG was chosen for ease of construction and viewing (the former using human-readable ascii files, the latter using any modern web browser), but unfortunately the format imposed some restrictions on image output. Specifically, I could find no method available for filling a region with quadratic (or even by-channel linear) color. The closest approximation available seemed to be linear gradients. Eventually I decided to use constant-color regions only, and modified my segmentation code to exclusively produce constant color models. Additionally, an assumption I was making about path rendering turned out to be violated at some stage of the process (either something intrinsic to the SVG format or to Firefox's rendering). Despite neighboring regions using exactly the same control points to plot their boundaries, slight cosmetic gaps appear between them - see Figure 2. These gaps are eliminated by drawing the filled paths with a nonzero stroke width (and stroke color matching the interior of the region), but this slightly expands the apparent boundaries of regions when rendered.

4 Dealing with Occlusions

Initially my plan was to implement as many improvements to the vectorized output as I could. The first I focused on was identifying an ordering of regions - i.e., which regions are partially occluded, and by which - and filling in the holes that would be left if the top regions were moved. As it turns out, that's the only one I did.

4.1 Top Layer Identification

My method for identifying the top layer uses a heuristic based entirely on the region and its immediate neighbors; as such, it does not necessarily produce a consistent ordering between nonadjacent regions if their neighbors change. Instead, it's meant to find some region that appears to sit on top of its neighbors, without considering whether it's truly the top region in a global sense.

The top region is assumed to be free of holes (in fact, I make the universal assumption that any hole is actually an occlusion). Among the regions that qualify, I choose the region with the highest weighted sum of these measures:

1. Convexity - subtract from 2π the sum of all left turns taken in a clockwise circumnavigation of bezier control points.

2. Inwardness - for each intersection point along the boundary (where three or more regions meet) find α_i, the maximum angle between the intersection edge and the region boundary (one side or the other). Sort all αs in ascending order, and take the sum $(\frac{\pi}{2}-\alpha_1)+(\frac{\pi}{2}-\alpha_2)+...\min(0,\frac{\pi}{2}-\alpha_3)+\ldots+\min(0,\frac{\pi}{2}-\alpha_n)$. The first two points can provide a positive contribution; the rest can only hurt a region's score (preventing a bias towards a layer with many intersection edges).

3. Smoothness - the negative sum of the angular difference in slope at any G1 discontinuous points along the boundary (differences $< \frac{\pi}{8}$ are ignored).

4. Simplicity - 3 minus the number of incoming intersection edges.

This method seems to work well enough, but I haven't spent much time testing and tuning it. In fact, my code still uses uniform weights. I've observed that inwardness is usually the most contributing factor in determining the top layer.

Figure 2: (a): An abstracted and vectorized image of Abraham Lincoln, rendered with stroke width of 0. (b): the same vectorization, with stroke width of 1.

4.2 Hole-filling

Hole-filling is the combination of region expansion / reshaping, and region reordering, such that background regions are explicitly drawn to fill the space covered by higher regions, followed by those top regions which then occlude them.

My initial work on hole filling involved attempting to extrapolate the regions surrounding the top layer in order to fill the hole in a convincing and reasonable way. The goal was to pair up the incoming edges of all partially occluded region using G1 continuous curves. One feature of the vectorizations produced by the earlier steps was expected to make this simpler: although there may be some neighboring regions which border the top layer in more than one contiguous boundary line, there will always be at least one neighbor with exactly one such boundary line. One boundary line means two incoming edges, which could be connected. This would hopefully reduce some other neighbor to only two unconnected incoming edges (the other two being shared by the previous region), allowing them to be matched and connected. Obviously there are cases which cannot be solved using this method, so while I worked on an implementation I focused on handling the simple cases as correctly as possible, while allowing for robustness in cases which were not easily solvable (by using nice extrapolation for all the applicable neighbors, then simply assigning the rest of the space to one of the remaining regions). Unfortunately, even given the specific qualities of the vectorization result, I couldn't get a working solution to this problem - too many boundary cases and bad assumptions in my initial work.

The approach I eventually settled on was to pick just one neighboring region and expand it to fill the space. I looked at growing either the next highest or the lowest neighbor. Expanding the highest neighbor introduced another rendering artifact, since it would result in the boundary for a region near the top being drawn multiple times (with sub-pixel detail), and occassionally color from a covered region would appear near the edge of the occluding region. This can be seen in the example images on my project report page.

One advantage of producing vectorization with filled holes is that if regions are skipped when the image is rendered, the result will still be aesthetically consistent (no gaps), and for the most part the change will not affect severely affect the image content. Applications include removing regions to reduce filesize, or sending regions using progressive transmission (a truncated transmission will produce a result similar to a full transmission, without obvious missing pieces). The SVG images available from my project page can be experimented on - just open one in a text editor and remove a set of regions (each defined path is a single region). Unfortunately, there is no distinction made between background and foreground regions, so one will occasionally fill the other.

5 Future Work

One obvious direction for expansion, given another week or two, would be to improve the system of selecting a region to fill another. At present there is no emphasis on color consistency or foreground/background separation; to truly possess the advantages mentioned in subsection 4.2, hole-filling would need to maintain some sort of visual consistency, whether in the form of preferring low-error color models, maintaining the edges between objects, or some other method. Although I haven't done any work in this direction yet, all of the pieces I would need (at least for the minimum-error color method) are already implemented.

Another distinct but related direction would be to focus on reducing the file size of the vector representation without significantly affecting the visual result. Partially occluded regions can be represented more simply, allowing the regions over them to form their visual boundaries. An alternate approach modifies the vectorization method described in section 3 to smooth out areas with high-detail boundaries (i.e., sections represented by many curves). These could be safely combined.

6 What I Learned and Read

A bunch. I hadn't done any work in vectorization or abstraction before, so pretty much everything in this project was new to me. I also learned that papers that seem straightforward and detail-oriented end up

lacking a lot of information necessary for implementation.

The resources already referenced were put to use in my implementation; before settling on a project design, I also read most of the papers mentioned on my readings page, although I haven't been faithfully updated the status of each paper as I read them.

References

[1] D. Chetverikov. A simple and efficient algorithm for detection of high curvature points in planar curves. In *CAIP*, pages 746–753, 2003.

[2] H. Kang and S. Lee. Shape-simplifying image abstraction. *Comput. Graph. Forum*, 27(7):1773–1780, 2008.

[3] H. Kang, S. Lee, and C. K. Chui. Coherent line drawing. In *NPAR '07: Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*, pages 43–50, New York, NY, USA, 2007. ACM.

[4] M. Sarfraz and A. Masood. Technical section: Capturing outlines of planar images using bézier cubics. *Comput. Graph.*, 31(5):719–729, 2007.

[5] C. Steger. An unbiased detector of curvilinear structures. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(2):113–125, 1998.

[6] w3school. Svg tutorial. http://www.w3schools.com/svg/svg_intro.asp.

[7] S.-H. Zhang, T. Chen, Y.-F. Zhang, S.-M. Hu, and R. R. Martin. Vectorizing cartoon animations. *IEEE Transactions on Visualization and Computer Graphics*, 99(2), 5555.