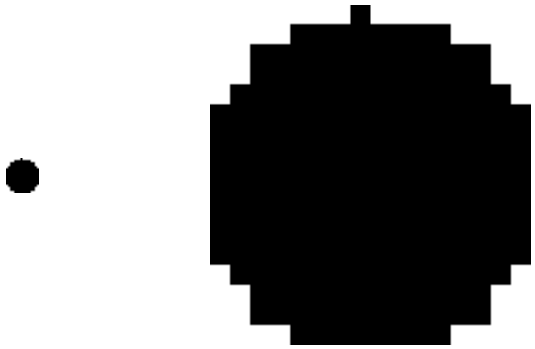
Jeffrey Madsen
27.11.06

Vectorization of Raster Images Using Polygon Tracing

1. Vector Graphics

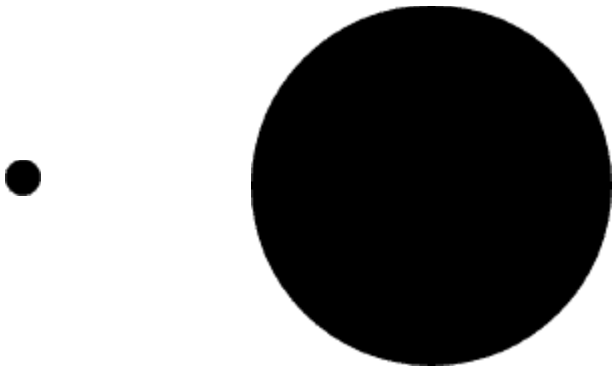
There are two main ways to represent a two dimensional image, raster or vector graphics. Most image formats, such as JPG, GIF, BMP, and PNG, are raster graphics, also called bitmap graphics, meaning they store the image as a grid of pixels. BMP is the simplest of these formats, storing the 24-bit color at each pixel. Other formats like JPG and GIF have optimizations to reduce the size of images, but they still represent the image as an array of pixels. This is very useful for most images, in particular for photographs. However, raster graphics have some weak points. Most prominently, if a raster image is scaled, it becomes blurry and pixelated. For example, here is a picture of a circle drawn in MS Paint, and the same circle magnified 10x:



While the small image looks like a circle, the blown up image has jagged edges that make it look bad. In this simple example, we could fix the problem by just drawing a larger circle in Paint. However, if there are a lot of shapes, or the shapes are not as simple as circles, it could take a lot of work to reproduce the image at a higher resolution.

Vector graphics are the solution to this problem. Instead of representing images by pixels, vector graphics formats, such as SVG, DXF, SWF, and PS, use parameters to represent shapes. A circle, for example, has a center and a radius. Other common vector primitives include lines, specified by their endpoints, and Bezier curves, specified by their control points. Most vector formats can also store other information, such as stroke width, stroke color, and fill color. An obvious advantage of vector graphics is that the file size is small. To store a line, for example, the file does not have to store information about all the pixels in the line, but rather just two endpoints. The vector format also fixes

the problem of scaling. The following is a small circle drawn in Inkscape, a vector graphics editor, and the same circle magnified 10x:



The circle still looks good no matter what size it is because the renderer which converts the vector image to a bitmap can produce a good-looking circle of any given radius. Other transformations, such as rotations, can also be performed on vector graphics without affecting the quality.

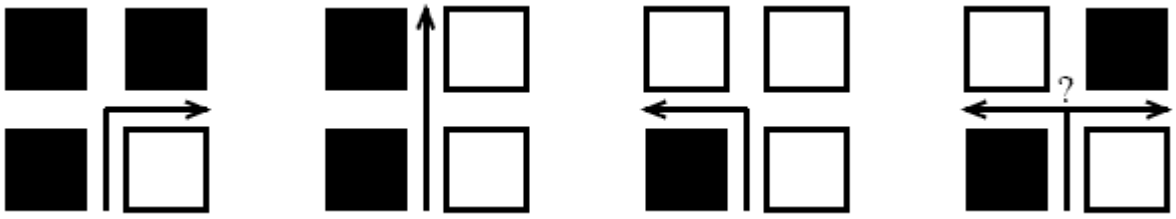
Because vector graphics are easy to manipulate, they are often better than raster images. However, vector graphics are only practical for images which can be easily represented as combinations of simple shapes. In particular, vector graphics are not good for representing photographs or realistic drawings. A common use of vector graphics is for diagrams.

Rendering, the process of converting from vector to raster graphics so they can be displayed, is simple. The reverse process, called tracing, which takes a raster image and converts it to vectors, is not so simple. There are a variety of algorithms to perform this, each producing different results, since the vectorized image is just an approximation of the original image. After all, if the vectorized image looked just like the original, it would be no better than a raster image. The algorithm described here was created by Peter Selinger and is called Potrace, short for polygon tracing. Potrace can be found at http://potrace.sourceforge.net. Potrace has been integrated with the popular open source vector graphics editor Inkscape. Potrace has three main steps: edge detection, polygon optimization, and smoothing.

2. Edge Detection

Potrace works on an input of a black and white bitmap image. The images produced by Potrace consist of shapes made from Bezier curves. These curves describe the boundary of black regions. In order to convert the raster image to such a vector image, then, the first step is to figure out which pixels from the original image constitute the borders of black regions.

First, an edge is found. An edge is defined to be a border between a white pixel and a black pixel. The edge is assigned a direction so that when moving from the first endpoint to the second, the black pixel is on the left. Then, a path extension algorithm is used to find the next edge in the path.



This picture shows the possible arrangements of pixels and the choice for the next edge in each case. In the first three scenarios, it's easy to decide which edge to pick up next to form the boundary of the black region. In the last case, it's not so obvious. The direction that is chosen in this case can be set by changing a parameter in the input to Potrace.

This process is repeated until we reach the starting point, at which point we have found a closed path which encloses a black region. Next, we remove said region from the image, because we are done with it. If there is still black left in the image, we continue this process. Finally, once all the borders of the black regions have been detected, any sufficiently small regions (determined by an input parameter) are removed, for they are most likely artifacts and not part of the actual image.

3. Polygon Optimization

Once the border has been found, the next step is to approximate the border with a polygon. First, we figure out which border pixels it is possible to connect with a straight line such that the line passes through all the border pixels between its endpoints. This information is used to compute the set of possible polygons whose vertices have integer coordinates and which approximate the border. We only consider those polygons whose edges pass through every pixel of the border. The polygon with the fewest vertices which meets this criterion is considered optimal. To decide between polygons of the same number of vertices, a mathematical formula is used to compute a value called the penalty for each edge. Roughly speaking, the penalty measures the average distance from the edge to the pixels it approximates. The polygon which minimizes the total penalty is the optimal one.

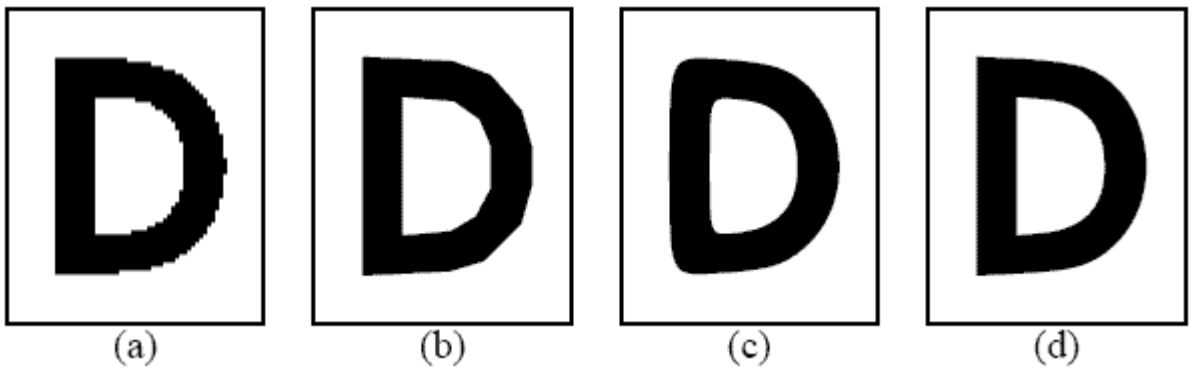
Once the optimal polygon has been found, its vertices are adjusted. Until now, we have only considered polygons with integer coordinates. Thus, we have found the optimal placement of the vertices to within a distance of ½. The vertices are now moved so as to minimize the edge penalty, without straying more than ½ from the original location.

4. Smoothing and Corner Detection

So far, the algorithm has produced the optimal polygon to approximate each shape. That's not what we're looking for, however. The goal of this process is to produce an approximation with Bezier curves. The final step, smoothing, converts a polygon into a set of Bezier curves.

Recall that a Bezier curve is a cubic curve defined by four control points. The first and fourth control points give the locations of the two endpoints of the curve, while the second and third indicate the direction and magnitude of the derivative of the curve at each endpoint. The smoothing algorithm chooses the midpoints of the edges of the polygon to be the endpoints of the Bezier curves. The second and third control points are chosen on the polygon edges through the endpoints, so that each Bezier curve is tangent to the polygon at its endpoints. It remains to determine where on the edge these control points should go. The distance along the edge toward the vertex of the polygon where the two edges meet is given by the parameter α.

A formula is used to compute α. If $\alpha < 1$, the control point is placed with ratio α along the edge. If $\alpha > 1$, this means that the edges come together at a tight corner, and a Bezier curve would not look good at this vertex. Thus, when this happens the vertex is not approximated by a Bezier curve, and instead the original polygon edges are used. This is called corner detection. It is important to have the correct threshold for corner detection, as too few or too many corners in an image cause it to look bad. An example is this letter D:



(a) is the original. (b) shows what the result would look like with too low of a threshold. The program detects corners where none should exist. (c) shows what it would look like with no threshold. The program fails to detect the corners, and thus uses Bezier curves instead. (d) shows the optimal result, with the right number of corners.

This is the basic algorithm of Potrace. There are several optimizations to it. For example, at the end of the algorithm, several Bezier curves may be combined into one curve if it does not affect the result greatly. There are also algorithms for precomputing edge penalties. More information can be found at the Potrace website.

Sources

http://potrace.sourceforge.net

http://en.wikipedia.org/wiki/Vector_graphics

http://www.inkscape.org

Images from http://potrace.sourceforge.net/potrace.pdf