CSC320— Visual Computing, Winter 2005

Assignment 3: Painterly Rendering

Due: 9:10am, Wed., Mar. 23 (at the start of the lecture)
This assignment is worth 10 percent for your grade in this course.

Background. In this assignment you will implement a basic system to produce a rendering of an input image in the style of a painting. Download the paper by Peter Litwinowicz on painterly rendering, available from the table of assignments on the course homepage. The paper goes beyond what we will do in this assignment. In particular, we will not be concerned with rendering image sequences (i.e., ignore section 3.C on frame to frame coherence). You can also ignore brush textures and anti-aliased stroke rendering, and thin-plate spline interpolation, as discussed in Litwinowicz's paper. I don't expect you to know what anti-aliasing is, or what thin-plate splines are. But I believe you will find the remaining portions of the paper both readable and motivating.

Handout Code. In `painterlyHandout.zip` I have included a test image and some simple starter Matlab code. Running the handout Matlab script file `painterlyHandout.m` produces the result shown below.

Figure 1: Original `orchid.jpg` image and the image output by the handout code.

By browsing the handout code, you will see that `painterlyHandout.m` inputs an original image, then randomly samples image pixels and renders fat brush strokes on a new canvas. The brush strokes have the RGB values of the sampled pixel from the original image. The radius, orientation and length of each brush stroke are fixed constants (although some strokes are cropped to fit within the original image frame). Your job is to build on this handout code in the manner described below.

Some of the Matlab implementations of these operations may be quite slow. It may be useful to crop an illustrative region out of the test image `orchid.jpg` in order develop your program.

Covering the Canvas [5pts]. Your first task is to ensure that the canvas is covered by painted strokes. The handout code simply selects a fixed number of stroke centers at random and, as a result, may leave various pixels unpainted. The unpainted pixels have RGB values of -1. Replace this blind random sampling by a more systematic sampling of unpainted pixels. In particular, initialize the canvas to -1, and then loop until all pixels are painted. On each iteration of the loop use Matlab's find operation to locate all pixels which remain unpainted. Randomly select one of these pixels as the center of the next paint stroke, and then paint the corresponding stroke on the canvas (as done in the handout-code). As described in Litwinowicz's paper, it is important to randomly select the next center pixel to use, rather than processing the pixels in scan-line order, in order to avoid the appearance of a systematic ordering of the strokes.

Computing Canny Edgels [5pts]. Your next task is to compute a Canny edgel image. This can

be done by extracting code from the Matlab script file `cannyTutorial.m` and the M-file `cannyEdgelDemo.m`. The edge code should run on a single monochrome image. Litwinowicz suggests using the intensity image

$$I(\vec{x}) \;=\; 0.30\,R(\vec{x}) + 0.59\,G(\vec{x}) + 0.11\,B(\vec{x}), \tag{1}$$

where $R(\vec{x})$, $G(\vec{x})$, and $B(\vec{x})$ are the three colour channels of the original image.

For the edge detection, use $\sigma = 2.0$ for the standard deviation of the Gaussian derivative filters (used by the Canny edge detector). Adjust the threshold on gradient amplitudes so that most of the edgels separating distinct regions of the image are detected but, at the same time, not too many edgels appear in other image regions. There is no perfect threshold. You may wish to adjust this threshold after the clipping code described in the next section is implemented.

Clipping Paint Strokes at Canny Edgels [5pts]. Given the binary image marking the locations of Canny edgels, use this image to clip painted strokes. One approach for doing this is described in Appendix A of Litwinowicz's paper. We will make a few simplifications to this process.

Suppose $\vec{c}$ is the integer valued pixel location for the center of the stroke to be painted. Let $\vec{t} = (t_1, t_2)^T$ be the tangent direction (non-integer valued) for the stroke.

If there is an edgel at pixel $\vec{c}$ then paint the stroke with a length of 0 using `paintStroke.m`. This will paint a disk of the stroke radius centered on $\vec{c}$. Otherwise, we wish to walk along the line in direction $\vec{t}$ from pixel $\vec{c}$ until we either find an edgel in the Canny edge map or we have gone more than the distance `halfLen` from the center pixel. One endpoint of the paint stroke is determined by the first edgel found along this line or, if no edgel is found, then the last pixel within the maximum distance `halfLen` of the start pixel $\vec{c}$ is used. The other endpoint of the segment is determined in the similar way, walking in the opposite direction $-\vec{t}$ from $\vec{c}$. (You can use loops to find these endpoint positions.)

One remaining detail is specifying which discrete pixels are visited when we walk along the line segment starting at $\vec{c}$ in the direction $\vec{t}$. Suppose $|t_1| \geq |t_2|$, so the desired stroke is within 45 degrees of horizontal. Then let $\vec{s} = \vec{t}/|t_1|$, so the first component of $\vec{s}$ is ± 1 and the second component satisfies $|s_2| \leq 1$. Then for $k = 1$ up to K we visit pixels

$$\vec{x}_k = \vec{c} + \text{round}(k\vec{s}). \tag{2}$$

Notice $\vec{x}_k$ is integer valued and, since $s_1 = \pm 1$, we always step exactly one pixel to the left or right each time we increment k. The maximum value K is determined by the largest k such that $|\vec{x}_k - \vec{c}| \leq \text{halfLen}$. For directions which are closer to being vertical, so $|t_2| > |t_1|$, use $\vec{s} = \vec{t}/|t_2|$ instead. In this case, $s_2 = \pm 1$ and $\vec{x}_k$ increments one pixel up or down with each step in k.

Note we are simply using the Canny edgel positions to crop the stroke, rather than looking for a decrease in the gradient amplitude, as described in Litwinowicz's paper. Also, notice that we only check for edgels along the center line of the painted stroke, as determined by the pixels $\vec{x}_k$ in equation (2), ignoring any edgels that may intersect the stroke off of this line (recall the stroke has a nonzero radius and may be significantly wider than this center line).

Orienting the Paint Strokes [5pts]. So far the strokes all have the same orientation θ. We wish to set the orientation of each stroke to be normal to the gradient of image intensity at the initial pixel $\vec{c}$ of the stroke. This will cause the strokes to be roughly aligned with the edge orientation, and with contours of constant intensity.

In order to pre-compute a smooth image of gradient directions, first filter the intensity image with derivatives of Gaussians, as in the Canny edge detector. However, use a larger standard deviation σ for these Gaussians than was used to compute the Canny edgels (eg. use $\sigma = 4$ instead of $\sigma = 2$). Also, in order to cover most of the image with estimated gradient directions, use a much smaller threshold on the minimum size of the gradient to be considered. The combined effect of using a larger value of σ and a lower threshold on the gradient size, should provide a gradient orientation estimate at most pixels.

Matlab code for computing such a gradient direction image can be extracted from the M-file `cannyEdgelDemo.m`. In particular, see the computation of the array of gradient directions `dirIm`. The gradient angle $\theta(\vec{x})$ at a pixel $\vec{x}$ is just given by π times the value of `dirIm` evaluated at that pixel. A second array, `enoughGradAmp`, provides an indicator of whether or not the length of the gradient is larger than the given threshold.

A paint stroke centered at pixel $\vec{c}$ should then be drawn with orientation $\theta(\vec{c}) + \pi/2$, where $\theta(\vec{c})$ is the gradient orientation as described in the previous paragraphs. That is, the tangent to the stroke is defined to be $\vec{t} = (\cos(\theta(\vec{c}) + \pi/2), \sin(\theta(\vec{c}) + \pi/2))$. The addition of $\pi/2$ makes the stroke orthogonal to the gradient, and therefore roughly parallel to curves of constant intensity. The stroke should then be clipped by any intervening edgels, as described in the previous section.

One remaining detail is to decide what to do at pixels which have gradients whose length is below the minimum threshold (i.e., `enoughGradAmp` at that pixel is false). The paper by Litwinowicz suggests interpolating neighbouring values of the gradient using something called a thin-plate spline. This is beyond the scope of this assignment. Instead, here we will use a constant default direction θ_0 for any stroke centered at these pixels.

Random Perturbations [5pts]. Finally, in the subsection titled Random Perturbations in Section 3.A of Litwinowicz's paper, the addition of random variations to both the colour and the stroke orientation is described. Implement these random variations. Note that here our RGB values are in the range $[0, 1]$ while in the paper they are assumed to be $[0, 255]$. Thus instead of perturbing the colour coefficients by random amounts in the range $[-15, 15]$, use $[-15/255, 15/255]$ instead. Also perturb the intensity and the stroke orientation, as described in this subsection of the Litwinowicz's paper.

What to Hand In. Include a short write up describing your program. Don't repeat the details described above, but rather describe any changes you might have made. In particular, provide the specific values of the parameters you used (for example, the Gaussian standard deviations for edge detection and for gradient orientation estimation, the thresholds on the minimum size of the gradient to be used in these two stages, and the stroke radius and maximum half-length parameters). Include a printed listing of your program. In addition, include printed output images for each of the following:

- the Canny edgel image used for clipping the strokes,
- the gradient orientation image used for orienting the strokes (similar to either the colour or monochrome gradient orientation output from `cannyEdgelDemo.m`),
- a painterly image constructed without adding the random perturbations,
- a painterly image with the random perturbations (as described in the previous section).

All of these should be done for the original image `orchid.jpg` or, if your code is too slow to be able to conveniently paint the entire canvas, crop `orchid.jpg` to a small illustrative region. Finally electronically submit your Matlab code in one zip file titled `painterly.zip`.