Parker Greene

PSET #10 Write-up: NPR

For this pset, I chose to implement the painterly rendering pset from a previous year's assignment. I did all parts of the pset, including two-scale oriented painterly rendering in standard C++, and then re-implemented the two-scale oriented painterly rendering algorithm in Halide. I chose this assignment because I have done a lot of similar NPR projects in computer graphics and thought it would be interesting to apply these methods to digital photos. I then implemented the painterly rendering algorithm described in the handout in Halide because the standard C++ was so slow.

Background

Painterly rendering is the concept of taking a digital photo and restyling it to appear as if it was painted with a brush.

The algorithm I implemented is very similar to the technique described in Hertzmann [1998] in which large oriented strokes are applied first, then smaller ones are added on top of that. See: (https://mrl.nyu.edu/publications/painterly98/)

The earlier work in this field was done for animation rendering and used information about the 3D geometry in Meier [1996] (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.1805&rep=rep1&type=pdf)

Another approach is to have a "brush dictionary" and then learning to use certain brushes for specific parts of the "painting" as in Zeng et. al. [2005]. My implementation only uses one brush texture that is resized for finer details. (http://www.stat.ucla.edu/~mtzhao/research/parse2paint/)

Algorithm

The two-scale oriented paint algorithm takes as input the image to make painterly and an image of a brush texture. The brush texture is a grayscale image, where the the brush texture is applied to the photo at N random (x, y) locations and the color of the brush is the pixel at (x, y). The brush size, as well as the number of strokes, can be changed. Additionally, the brush strokes are oriented to match the geometry of the photograph.

The first step of the algorithm is to compute an angle for every pixel of the image. This is done by computing the image's structure tensor (from Harris corner detection), which is a 2x2 matrix. The angle of the eigenvector corresponding to the smallest eigenvalue at tensor(x, y) is the angle for pixel (x, y).

The drawing of the brushes is then done in two passes:

The first pass of the algorithm rescales the brush texture to the desired size (a parameter of the function). Then, N random pixels are sampled from the image, the brush is rotated to match the previously computed angle, and the rotated brush is drawn with the color at input(x, y) centered at (x, y).

The second pass of the algorithm draws a brush with ¼ the size of the brush from the first pass at areas of finer detail. To figure out where there are areas of finer detail, we must compute a sharpness map of the input image. To do this, we compute a high-pass image of the input luminance (luminance image - blurred luminance image), square it, then blur it again. We then normalize the map so that all values are in a 0-1 range and use this sharpness image as an importance map for our random sampling. That is, when we pick a random point (x, y) to draw the brush, we check the importance map at (x, y) and discard that random point with a probability of importance_map(x, y). If we do not reject the sampled point, then we rotate the scaled brush texture and draw it at (x, y), as we did in the first pass.

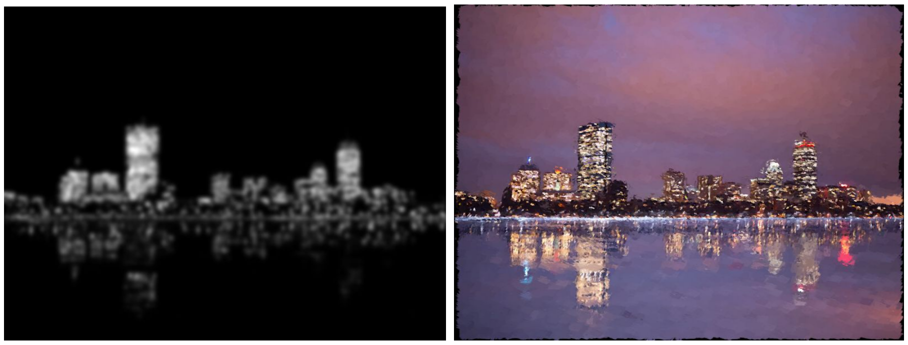
A minor detail is that in the second pass, we actually draw N*k brush strokes to account for all of the rejected samples, where k = 1/(average pixel value of importance map). Also, in both passes we multiply the color of the brush 1-noise/2 + noise*random(0, 1) for every stroke, where noise is an input parameter to the algorithm.

Implementation

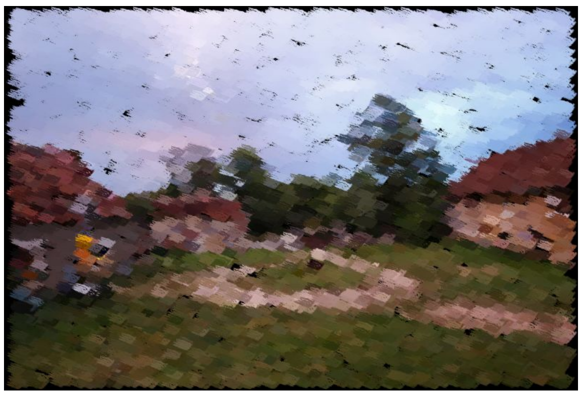
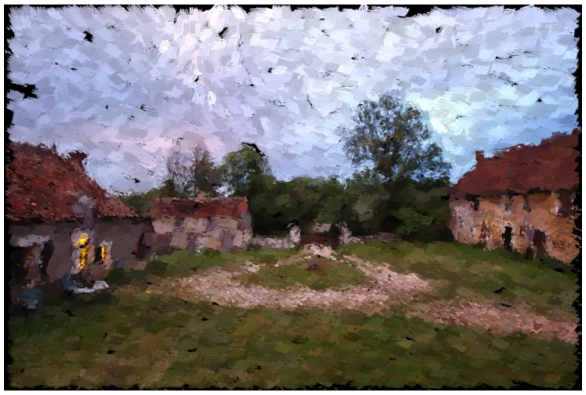
My C++ implementation follows the above steps and matches the steps in the handout. I converted the necessary Python helper functions from the starter code to C++. My Halide implementation only does the two-scale oriented paint algorithm, rather than having a separate function for the single-scale algorithm, which I felt wasn't necessary, since the two-scale oriented paint was much more interesting anyway.

I found the C++ implementation pretty straightforward, but the Halide implementation was very difficult as expected, both in implementing it and optimizing it as much as possible. The speedups weren't as much as I hoped, but still pretty good, especially on large inputs (see results below for comparisons). I think this has a lot to do with the fact that locality is very difficult to achieve, since we are computing random (x, y) locations for every brush stroke.

Sharpness Mask and Final output:

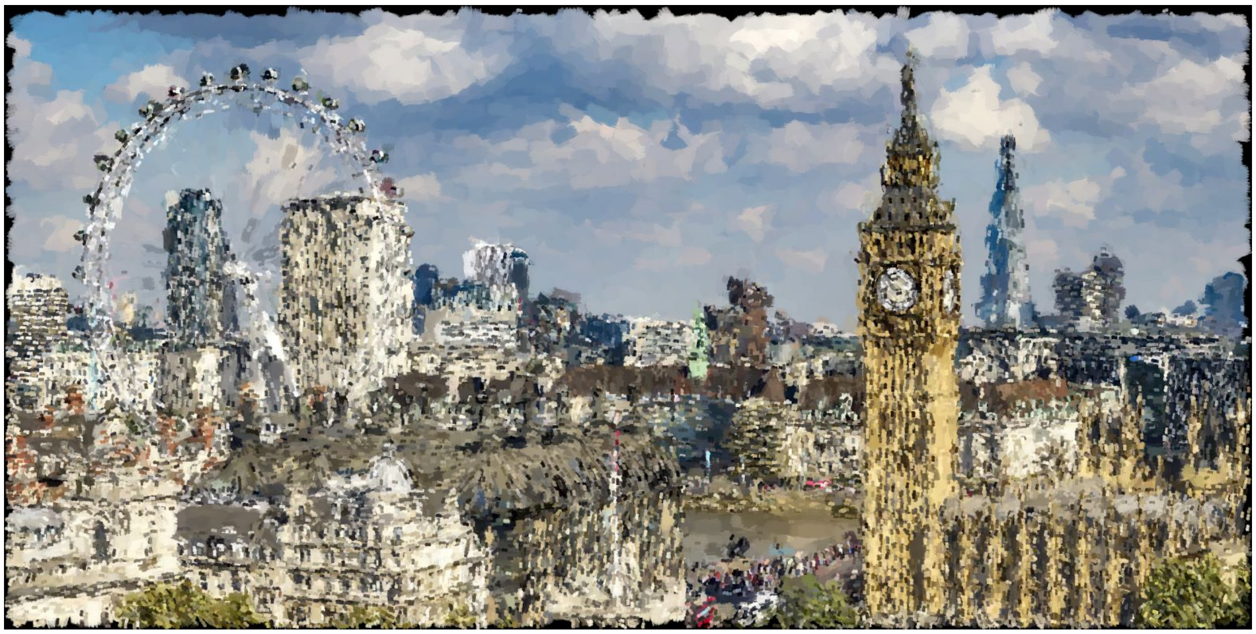


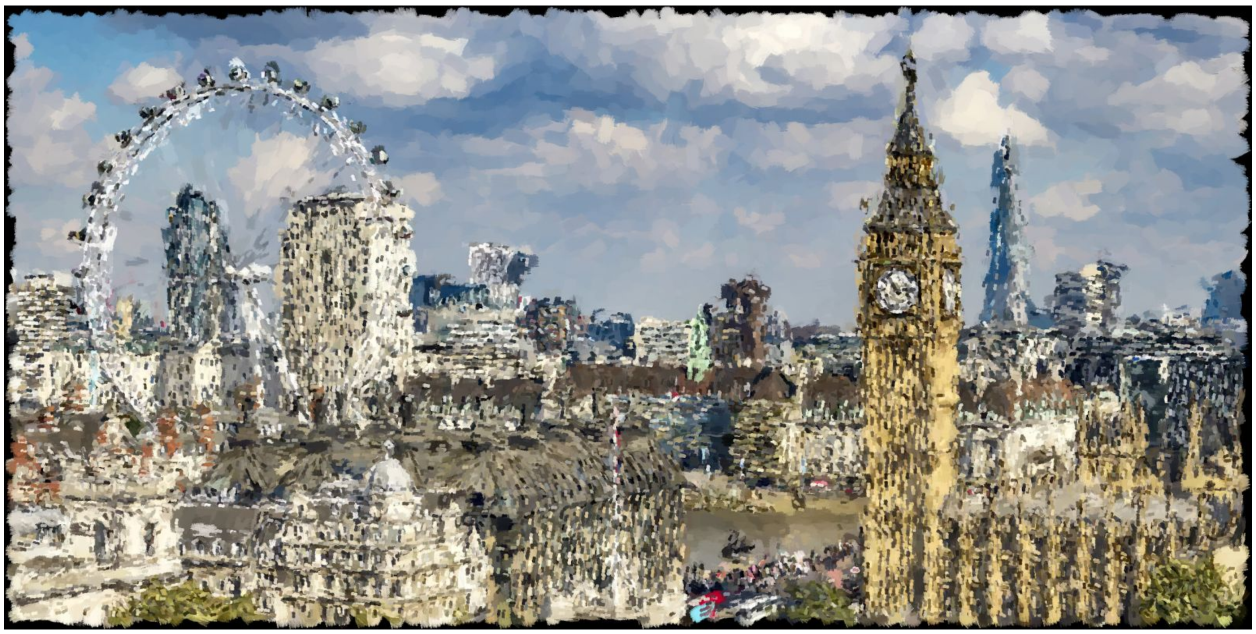
Unoriented single scale (left) vs. Two-scale oriented (right):

Results
10,000 strokes on an 800x532 image for C++ (top) and Halide (bottom) implementations.
The average runtime for the C++ implementation was 7041.8 ms (0.06 megapixels/sec).
The average runtime for the Halide implementation was 1733.4 ms (0.245 megapixels/sec).

30,000 strokes on an 2560x1280 image for C++ (top) and Halide (bottom) implementations.
The average runtime for the C++ implementation was 73967.2 ms (0.044 megapixels/sec).
The average runtime for the Halide implementation was 2877 ms (1.14 megapixels/sec).





The following is averages for running both implementations on an 800x600 image 5 times each while varying N, the number of brush strokes. I cut off the timing at 60 seconds.

N	5000	10000	15000	20000	30000	40000	50000
Halide time (s)	2.128	2.202	2.942	3.598	6.219	6.055	5.468
C++ time (s)	27.67	50.433	60+	60+	60+	60+	60+

The quantitative results for the above pictures and chart show that while the Halide implementation doesn't perform the computation in real-time, it is much faster and impressively scalable both in terms of N and the number of pixels of the image. Qualitatively, there is no difference in the output of the implementations (other than slight variations due to the random nature of this algorithm, of course).