MIT EECS 6.815/6.865: Assignment NPR:

Painterly Rendering

Due Wednesday Nov 13 at 9pm

1 Summary

- Render a paintbrush onto an image
- Single-scale painterly rendering
- Importance sampling using rejection
- Two-scale painting
- Orientation computation using the structure tensor (6.865 only)
- Two-scale oriented painting (6.865 only)
- Choose your topic for Pset 13

In this assignment, we will take photographs as input and turn them into a painterly rendition. We will stochastically splat brush strokes on the image, varying the size, angle, and density of strokes. Below is one example of the type of rendition we will generate.

In a nutshell, we generate random locations y, x, read the color in a reference photograph, and splat a small image of a brush stroke at this location in the output image. We modulate the size and orientation of the brush strokes to make things more interesting.

2 Paintbrush splatting

Write a function `brush(out, y, x, color, texture)` that takes as input a mutable image `out` and draws ("splats") a single brush stroke centered at `y, x`. The appearance of the brush is specified by an opacity `texture` and a 3-array `color`.

In order to make life easier and avoid boundary problems, ignore cases where `y` and `x` are closer than half of the texture size away from the image boundary (in otherwords, your function should do nothing in this case). Plus this will give you the nice rough edges shown in the above image.

`texture` is a regular 3-channel image that specifies the opacity at each pixel. It will be greyscale (all three channels are the same) in practice. An opacity of zero means that the pixel is unaffected, and 1 means that the pixel is updated with `color`. For values between 0 and 1, the output is a linear combination of the old value and `color`. We advise that you generate an image of the same size as `texture` that has a constant color equal to `color`. Compositing the brush into `out` is then a simple numpy array operation with the help of the slicing operator ":" to select the appropriate sub-region of `out`.

3 Painterly rendering

3.1 Single scale

Write a function
`singleScalePaint(im, out, importance, texture, size=10, N=1000, noise=0.3)`
that creates a simplistic rendering by splatting a paintbrush at random locations. For now, we'll ignore the `importance` parameter.

First, scale the `texture` image so that it has maximum size `size`. Use the provided `scaleImage` function or your own method.

For each of `N` random locations y, x splat a brush in `out` using the above function. Generate random locations using `rnd.randrange(start, stop)`, assuming you imported the random module using `import random as rnd`; or using `numpy.random.randint(low, high)`.

The color of the brush should be read from `im` at y, x. To make things more interesting, modulate the color by multiplying it by a random amount proportional to `noise`: (1-noise/2+noise*`numpy.random.rand(3)`)

Make sure you debug your function.

3.2 Importance sampling with rejection

We will vary the density of strokes according to an importance map `importance`, which is encoded as a regular image where all 3 channels are the same. We want the density of strokes to be proportional to this map. The simplest way to do this is to use *rejection sampling*. For this, you should reject samples with a probability proportional to `importance[y, x]`. Draw a random number using `rnd.random()`, assuming you imported the random module using `import random as rnd`; or using `numpy.random.rand()` in order to avoid conflicts with numpy's `random` and reject the sample if this random draw is greater than `importance[y, x]`. Note how samples are always accepted when importance is 1, always rejected when it's 0, and in general, accepted with a probability equal to the importance value.

Since we rejected a number of samples, we do not splat the required `N strokes`. In order to fix this, multiply `N` by a normalization factor based on the average probability of accepting samples. You can also use a while loop, but it involves less probabilistic calculation and is therefore less fun.

Modify your function `singleScalePaint` to take the importance parameter into account and debug it using some importance map of your choice.

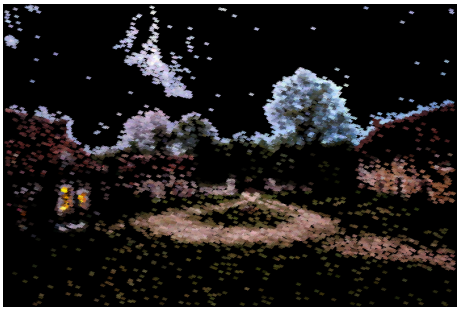
3.3 Two-scale painterly rendering

Use the above function to implement a method `painterly(im, texture, N=10000, size=50, noise=0.3)` that returns a painterly rendering of `im` where a first layer of coarse strokes is used, followed by a refinement with smaller strokes in regions of high detail.

In the first pass, use brushes of size `size`, and in the second pass, brushes of size `size/4`. The first pass should use a constant importance map and the second finer pass will only add strokes where the image has strong high frequencies, as indicated by a sharpness map similar to that of pset 11. You can use the provided `sharpnessMap` function.

My result is below

And I also removed either the first or second pass to show their individual contributions



4 Oriented painterly rendering (6.865 only)

We now seek to orient the stroke direction according to the content of the input image. For this, we will rely on the structure tensor used for Harris corner detection. The structure tensor is better than the gradient for this purpose.

4.1 Orientation extraction

We know that the structure tensor characterizes how much an image changes in all directions. The eigenvectors of the structure tensor indicate the directions of maximum and minimum variation. We want to align our strokes along edges and oriented structures, which means that they should follow the *smallest*

eigenvector. We will first experiment with orientation extraction before using it for our painterly rendering.

Write the function `computeAngles(im)` that takes an image and returns a new image of the same size where each pixel has been replaced by the angle between its local edge orientation and the horizontal line. To do this, use `tensor=computeTensor(im, sigmaG=3, factor=5)` to extract a structure tensor at a slightly large scale. Then use `np.linalg.eigh` to extract the eigenvectors and compute the angle of the smallest one with respect to the horizontal direction (use `np.arctan2` for this).

4.2 Single-scale

Write a function `singleScaleOrientedPaint(im, out, thetas, importance, texture, size, N, noise, nAngles=36)` that behaves similarly to the non-oriented version, excepts that brush strokes get oriented according to the provided `thetas` map (which should be equal computeAngles(im) in most cases).

For this, we encourage you to use the provided function `rotateBrushes(texture, n)` which takes as input a single texture and returns a list of `n` textures rotated by $i2\pi/n$.

4.3 Two scale

Write a function `orientedPaint(im, texture, N=7000, size=50, noise=0.3)` that returns a painterly image where strokes are oriented along the directions of maximal structure. Use a two-scale approach as above. You can use the same angle map (thetas) for both approaches. My results are at the beginning of the assignment, and below.

5 Your image

Apply one of your functions to your own input and include it as "MyPainting.png".

6 Submission

Turn in your image, python files, and make sure all your functions can be called from a module `npr.py`. Put everything into a zip file and upload to Stellar.

Include a `README.txt` containing the answer to the following questions:

- How long did the assignment take?
- Potential issues with your solution and explanation of partial completion (for partial credit)
- Anything extra you may have implemented
- Collaboration acknowledgement (but again, you must write your own code)
- What was most unclear/difficult?
- What was most exciting?