6.815 ps10

Jenna Himawan

December 2019

1 Introduction

For ps10, I decided to do the non-photorealistic rendering pset from a past year. I thought the results were simply beautiful, and I was eager to implement code that I could use to create painterly renderings of whatever images I wished.

2 Background

Some interesting papers that I found which deal with this subject are:

- Non-photorealistic rendering using watercolor inspired textures and illumination: shape and texture information can be more effectively communicated (compared to standard rendering models) with a technique that applies multiple layers of semitransparent "paint". Textures are calculated by taking a LIC (Line Integral Convolution) of Perlin Noise along one of the principal curvature directions. The thickness of each layer is determined using a modified Phong shading model.
- Painterly Rendering for Video and Interaction: a painterly video or animation can be created by "painting over" a previous frame, only applying paint where the source video is changing. In order to ensure that video noise does not cause a static region to get repainted, this technique uses difference masking to determine where there is significant motion.
- Non-Photorealistic Rendering Techniques for Motion in Computer Games: motion can be represented in an abstract fashion using the 3 techniques of squash-and-stretch, multiple images, and motion lines. Squash-and-stretch refers to the deformation of an object as it travels; this can be based on velocity and/or acceleration. Multiple images refers to creating "shadows" of an object as it travels; these "shadows" are often made to be more and more transparent as they get farther away from the current position of the object, and may also be stylized by drawing them only partly (in streaks). Motion lines refers to creating lines which trail behind an object; the opacity of these can change as a function of the distance from the object's current position.

3 Algorithm Overview

In order to create a "painterly" image, we apply two layers of brush strokes. The first layer consists of big brush strokes and acts as a "base layer". The second layer consists of small brush strokes and acts as a "details layer".

In order to generate the first layer, we randomly select N points in the input image, and put paint "splats" in the corresponding locations in the output image. These paint splats have approximately the same color as the pixel from the input image (we can add a bit of noise for a nice effect). Their shape is determined by a texture parameter (which is an image that contains the shape

of each paint splat) and an angle (which is equal to the angle of the eigenvector that corresponds to the smallest eigenvalue for the tensor at that location).

In order to generate the second layer, we find the high frequencies of the image, and use it to assign an "importance" score to every pixel; "importance" is a number between 0 and 1. When we select points of our input image for splatting, we reject a pixel with probability (1.0 - importance). When we reject a pixel, we move on to the next point, but we have to make sure that approximately N splats are going to be created; we can do this by calculating an average probability of selection over the image and increasing "N" by an appropriate factor, or we can count the number of splats until we get to N. Once we have the importance scores, we put down splats much like we did in the first layer. The colors and orientations are determined in the same ways, but we scale the size of our brush down from the original template.

4 Implementation

For this pset, I implemented the functions:

- `void brush(Image & out, const int x, const int y, const float color[], const Image & texture)`: creates a paint splat at coordinate `(x, y)` on the output image `out`, with RGB color `color` and texture `texture`. If the `(x, y)` coordinate is close enough to the boundary that some of the splat would go outside the image bounds, don't put any splat; otherwise, sets `out(xCoord, yCoord, c) = color[c]*opacity + (1.0f - opacity)*out(xCoord, yCoord, c)`, where `opacity` is equal to the opacity of the texture image at a relevant location.

- `void singleScalePaint(const Image & im, Image & out, const Image & importance, const Image & texture, const int size=10, const int N=1000, const float noise=0.3)`: creates a layer of paint on the output `out` using the input image `im`. Importance scores are given by `importance`, brush texture by `texture`, maximum brush size by `size`, number of splats by `N`, and color noise by `noise`. Generates random `(x, y)` coordinates in `im` and, with probability determined by `importance(x, y)`, creates a paint splat whose color is equal to the R, G, and B values of `im(x, y)`, each multiplied by `(1.0f-noise/2.0f+noise*`(random float between 0 and 1)`)`.

- `Image painterly(const Image & im, const Image & texture, const int size=50, const int N=10000, const float noise=0.3)`: paints a base layer and then a details layer. The base layer has a uniform importance of 1.0. The details layer has an importance image which is the high frequency component of `im`, and it uses a brush of max size `size/4`.

- `Image computeAngles(const Image & im)`: compute, for each pixel, the angle (in radians counterclockwise from the horizontal) of the eigenvector

that corresponds to the smallest eigenvalue of the image tensor. The relevant calculations are:

```
Eigen::SelfAdjointEigenSolver<Eigen::Matrix2f> solver;
// Compute before fetching eigenvalues / eigenvectors
solver.compute(H);
// Eigenvalues and eigenvectors are in ascending sorted order;
1st <-> smallest
Vec2f eigenvector = solver.eigenvectors().col(0);
// Store the angles in an Image
angles(x, y) = atan2(eigenvector.y(), eigenvector.x());,
```

where `H` is a 2x2 matrix which contains `gradientX`^{2} in its upper left, `gradientY`^{2} in its bottom right, and `gradientY*gradientY` in its other positions.

- `void singleScaleOrientedPaint(const Image & im, Image & out, const Image & importance, const Image & texture, const int size=10, const int N=1000, const float noise=0.3, const int nAngles=36)`: the same as singleScalePaint, except it allows for a certain number of brush angles; the rotated versions of the brush splats are generated as the first step of the function, and they are used in subsequent steps. Each angle, as generated by the `computeAngles` function, is rounded to the nearest available angle. I decided to implement this by making a vector of Images, one for each rotation of the brush, and then indexing into it with `int rotatedBrushIndex = ((int)round(normalizedAngle * ((float)nAngles)/(2*M_PI))) % nAngles`.

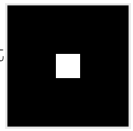
- `Image orientedPaint(const Image & im, const Image & texture, const int size=50, const int N=7000, const float noise=0.3)`: this code is exactly the same as `painterly`, except it calls `singleScaleOrientedPaint`.

The trickiest parts were figuring out how to deal with texture opacity in between 0 and 1, and how to get the right orientation for the paint splat.

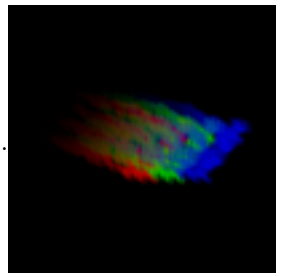
5 Test Cases

More test cases are available in my code, but I'm going to briefly discuss some of the important ones.

`testPointBrush()`: uses a "texture" which is a single point to paint a white color on a black image.

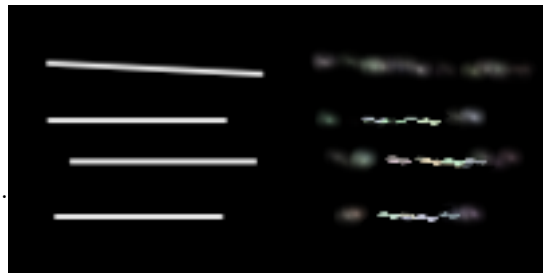


`testColoredBrushStrokes()`: paints multiple brush strokes of different colors.



`testSingleScaleImportance()`: tests having an importance of 1.0 in the top left, 0.0 everywhere else.

`testSingleScaleOrientedLines()`:
tests the orientation of brushstrokes on an image made of horizontal lines.
On the left is the input; on the right is the output.



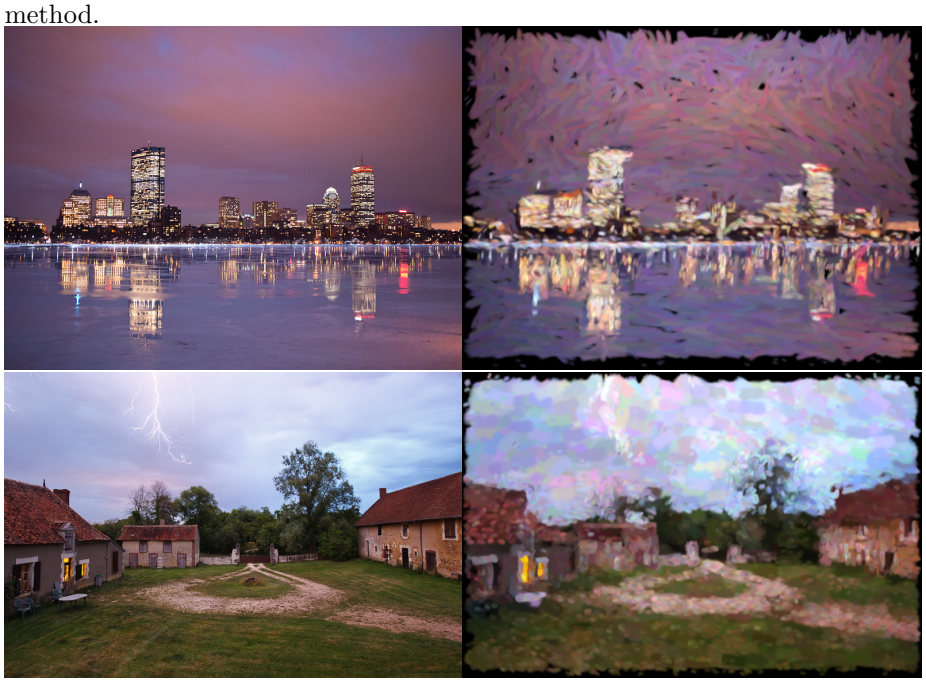
6 Statistics

The runtime of painterly seems to be pretty good. The amount of time it takes to render is dependent on the size of the texture and the number of splats. With size=50, N=7000, it takes about 5 seconds on my computer (a 2013 MacBook Air), and this is sufficient to handle an 800x600 image. As you move to larger images, the size and / or number of splats will have to increase, but the runtime is still reasonable for the 3368x3024 image that I experimented with.

7 Inputs and Results

Inputs are on the left; results are on the right.
The Boston and Ville Perdue images were made using the default `orientedPaint`

method.



I really wanted to make a beautiful painterly image of Simmons, so I made some adjustments to the painterly method. The image below is the result of 3 layers of paint; a first with an "opaque" brush that is designed to cover as much of the image as possible (size 50, 70000 strokes); a second with a "faded" brush that is designed to create nice mottled effects (size 50, 50000 strokes); and then a third with a "semi-faded" brush that is designed to fill in the details a bit (size 30, 50000 strokes). The input is here (small) and the output is on the next page (larger, to make texture visible).