Implementation of an Impressionistic Oil Painting Effect Algorithm

Aidan Johnson
johnsj96@uw.edu

12 December 2017

EE 440 Final Project Report
Introduction to Digital Image Processing
Department of Electrical Engineering
University of Washington

1 | Objective

This report details an implementation of an artistic effect algorithm described in a 1998 paper by Hertzmann [1]. The algorithm renders an input image as a painting using curved brush strokes. The 'brush strokes' paint and follow the edges of the image that are detected front the gradient of the image. The oil painting-esque output is constructed from superimposing painted layers that have increasing detail. The detail is controlled by the fineness of the paint brush. The curving strokes creates a very impressionistic representation of the image, making this one of the most aesthetic results of the algorithm. Having visited Musée de l'Orangerie in Paris to see the *Water Lilies* (*Nymphéas*) series by the Impressionist artist Claude Monet, I was particularly interested in attempting to render images in an Impressionist style.

After consulting literature on non-photorealistic rendering (NPR), I found the often-cited 1998 paper by Hertzmann. This paper proposed an algorithm that robustly rendered images with spline brush strokes [1]. It provided psuedocode and recommended style parameters. I decided to focus on implementing [1]'s algorithm for rendering Impressionist paintings with predefined and customised parameters. Originally, the algorithm in [1] had more style parameters that I only saw as computationally more expensive. Lastly, my goal was also to design an easy-to-use, yet customisable graphical user interface (GUI). In the end, I was able to accomplish this, and implement the Hertzmann algorithm with satisfactory results.

2 | Algorithm

The algorithm is dependent on the input parameters for the selected style. The most important parameter is the set of circular brush sizes described their radii. Per the recommendation of Hertzmann, a user of the GUI can only input no more than three brush sizes. Any more makes the rendering unbearably slow. The smallest brush radius R paints the finer details while the largest brush radius paints more coarsely. The input image is first blurred with a Gaussian kernel to detect the edges of the image. The blurring is performed by convolving the kernel with the image. The Gaussian kernel is defined by its standard deviation which is the product of the current brush radius and a scaling factor fs. The image is filtered with a decreasing standard deviation so that the base layer contains the lower frequency features while the topmost layers contain the higher frequency details. In effect, using smaller and smaller brushes captures the detail of the image more completely.

For each brush radius (and thus blurred image), the image is defined into a grid defined by a step parameter fg and brush radius R. The algorithm decides whether to paint a stroke starting in a grid area by computing whether the difference in colour of the grid block in the image and the constant coloured canvas is greater than the user-defined

threshold. The colour difference is calculated is pixel-by-pixel and then summed for each grid block. If the difference exceeds the threshold, the origin of the brush stroke is selected to be the pixel where the colour difference is the greatest for the block.

Then, at these Cartesian coordinates, a brush stroke begins to be painted. To prevent the painting rendering from looking non-human created (and therefore less artistic and creative), the order or rank of the brush strokes is randomly assigned by randomly ordering the grid blocks. The colour of the brush stroke is set by the origin pixel. To determine the path of the brush stroke, the gradient of the luminance at the origin pixel is calculated. Luminance is calculated from the RGB colour values. The next coordinate-pair of the point placement of the stroke is computed from scaling the vertical and horizontal gradient by the brush radius and adding it to the previous coordinate-pair such that the brush stroke moves along normal to the image gradients. Since there are two normal directions, the direction is less than 90 degrees from the previous direction is chosen to limit curvature.

Additionally, a brush stroke is limited by the maximum and minimum length (*maxLen* and *minLen*) parameters as well as the checking for a zero gradient and when the colour difference between the pixel of the blurred image and the canvas is less than that of the pixel and the stroke colour. The stroke path can be distorted with the curvature filter parameter *fc* to either make the stroke more or less curved. With a set of points describing the brush stroke spline, the stroke path is then painted. The path is painted by placing a circular colour element at each point of the stroke (for all three RGB colour channels). The paint stroke is applied to the layer. The stroke is only painted on the layer where no other paint exists.

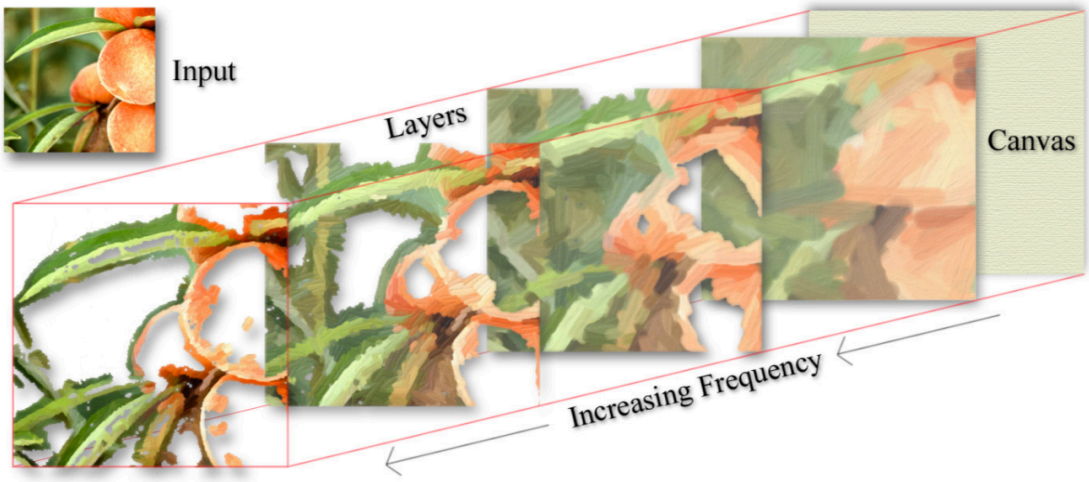

Figure 1: Illustration of the independent brush stroke layers [2].

Once the layer is painted, it is then superimposed on the canvas, which has the same dimensions of the original image. The canvas also is assigned a constant RGB colour C

(much like how one is all white in reality). The layer is painted 'over' (replaces) any already existing painted layer on the canvas and paints any blank canvas pixels. The layers build up to create a more refined and detailed rendering. The process described in the previous paragraphs is repeated for each brush size. The resulting rendering is then returned to be displayed or saved.

3 | Results & Discussion

To check whether code matched the original algorithm implementation, I rendered the same image that was included in [1]. My implementation matches the paper's provided output, considering the by-design random stroke order. The only flaw I noticed with my implantation was the speed of rendering. While [1] did not state the time it took to render the images included in the paper, to render the image (Figure 2, below) took less than 1 minute (the fastest at 41 seconds). Performance when rendering images much larger than the 600x400 example test image (e.g., when the dimensions are 4000x3000) is fairly poor—and dare I say painfully slow. For the same brush radii of 2, 4, and 8 pixels, the rendering time jumps to minute range. I am disappointed with this result, however I understand why it is so slow. It is computationally expensive to create a brush stroke and check for each grid block. Nonetheless, the output painted image (Figure 3) does capture the Impressionist aesthetic desired.

Figure 2: Original rendering test image [1].

Figure 3: The rendered image in an Impressionistic style.

To further demonstrate the algorithm, I rendered my own image below in Figures 4 and 5. The total elapsed time for rendering this image was 25 minutes. However, the result is highly attractive, personally, well worth the wait (perhaps drink some tea while it renders like I did).

Figure 4: My own photo taken at Brooklyn Botanic Garden.

Figure 5: Impressionist rendering of the photo.

4 | Instructions

For ease-of-use, I packaged my GUI and MATLAB problem as an App. The App package can be installed to MATLAB and the user can then load an image of their choice and set the rendering parameters. This was done to ensure the GUI opened up correctly (the individual code files are also provided). The user could also set the parameters to the default Impressionist style (where the threshold is 50, grid size factor is 1, blur factor is 0.5, curvature filter factor is 1, maximum stroke length is 16 points, minimum stroke length is 4 points, and the set of brush size radii of 2, 4, and 8 pixels). For those that want to inspect the code, the `.m` files for the renderer and the GUI are provided. The render can be run from the command window console by calling `renderPaint.m` file with (where type is in brackets):

```
renderPaint(imgName [char], threshold [double], grid size [double], blur factor
            [double], curvature [double], max length [double], min length [double],
            brush radii [R1,R2,R3], canvas colour [R,G,B])
```

This returns a matrix representation of the output image, which can be written to a file with `imwrite()`. The GUI (see Figure 6 for a screenshot) can also be run by running the `renderPaintGUI.m` file.

Notwithstanding, to try rendering your own image, I recommend using the MATLAB App package. Once the App is installed, it can be run from APPS tab in MATLAB. First, load an image by pressing the 'Load' button located in the 'Action' box. Then select your image from the window. The image will then be displayed in the box on the left. After pressing 'OK' select the style you want ('Impressionist' or 'Custom') from the options in the 'Style' box. Selecting the 'Impressionist' style will grey out the text entry boxes for the parameters.

If 'Custom' is selected, enter your desired parameters in the 'Parameter' box. The threshold is recommended to be set at 50-100 (a higher threshold means a rougher painting). The grid size should be a non-zero integer (it can be less than 1 as long as the product of it and each radius is a positive non-zero integer to ensure the grid has integer dimensions). The blur factor and curvature filter factor can be a positive real number. A smaller blur factor makes the painting more impressionist due to increased noise. The canvas, which acts like the base of the painted image, has a constant colour set by RGB values (numbers between 0 and 255). For the brush sizes, at least one brush radius must be inputted. These too should be positive integers.

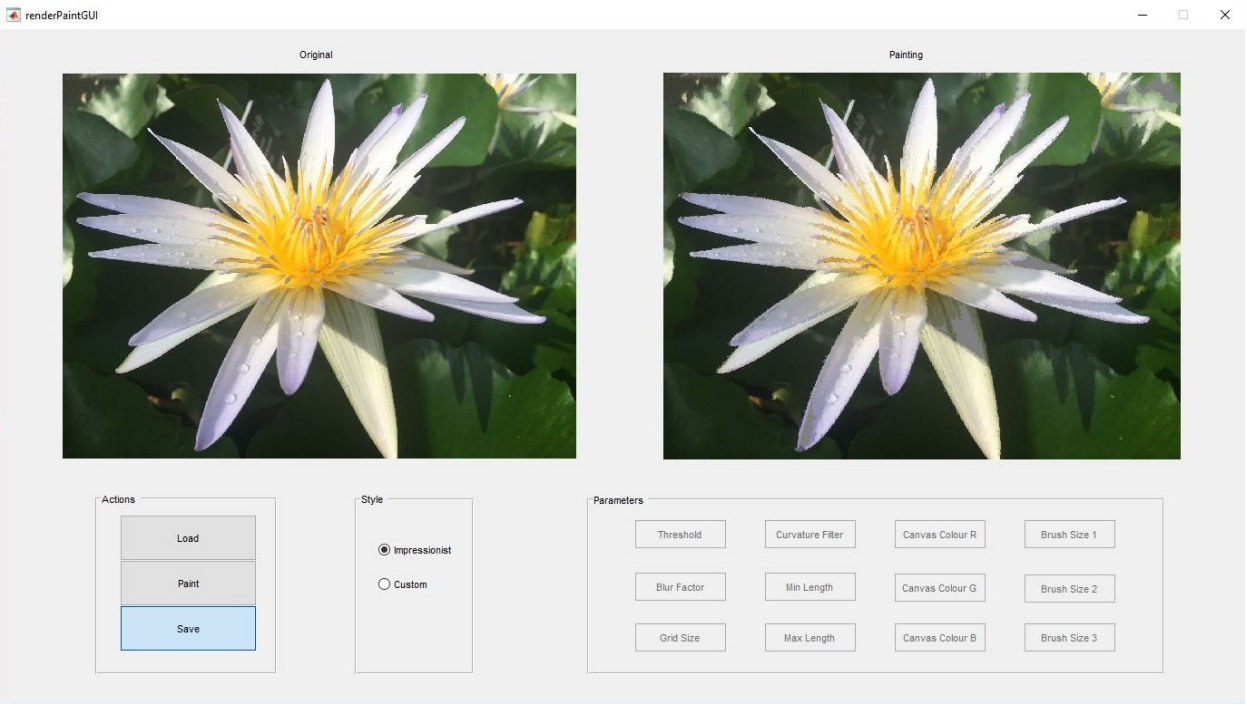

Figure 6: Screenshot of the GUI.

When you are happy with the parameters you entered, press the 'Paint' button for the program to render your image. Make sure to on press the 'Paint' button only once, otherwise the painting process may begin again, forcing you to wait until the rendering is complete. Once the rendering is complete, the rendered image will be displayed in the

box on the right. If you are satisfied with the rendering, you can save it by pressing the 'Save' button. It will be saved in the same directory where the original image is located and will be named '`painted_`' followed by the original image file name.

5 | Comments

While the my program produces impressive results and results that match the original implementation, there is room for improvement. Namely, the algorithm could have higher performance. It could also have other preset styles for a user to select. For example, [1] provided other preset styles like a Pointillist, Expressionist, and Colourist style. The original algorithm also introduced other parameters for opacity of the brush strokes and colour jitter that added random noise to hue, saturation, value, and the RGB colour channels [1]. On a more technical side, the quality of the rendering could be improved by implemented the variation proposed by Hays and Essa in 2004 [2]. These authors were inspired by [1] but implement an algorithm that orients the brush strokes to the globally strongest interpolated gradients. This method, however, is more resource and time intensive, as reported by Kagaya et al., who propose a similar algorithm for video frames [3].

This project was a refreshing fusion of art, image processing, and computer science. I learned how artistic approaches could be applied in digital image processing. It helped me demonstrate what I learned about digital image processing this quarter (not to mention something to show to family this winter holiday). Now I have a greater understanding, and appreciation, of the algorithms that are behind Adobe Photoshop's artistic effects features.

6 | References

[1] Aaron Hertzmann. 1998. Painterly rendering with curved brush strokes of multiple sizes. In Proceedings of the 25th annual conference on Computer graphics and interactive techniques (SIGGRAPH '98). ACM, New York, NY, USA, 453-460. DOI: http://dx.doi.org/10.1145/280814.280951

[2] James Hays and Irfan Essa. 2004. Image and video based painterly animation. In Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering (NPAR '04), Stephen N. Spencer (Ed.). ACM, New York, NY, USA, 113-120. DOI: http://dx.doi.org/10.1145/987657.987676

[3] Mizuki Kagaya, William Brendel, Qingqing Deng, Todd Kesterson, Sinisa Todorovic, Patrick J. Neill, and Eugene Zhang. 2011. Video Painting with Space-Time-Varying Style Parameters. IEEE Transactions on Visualization and Computer Graphics 17, 1 (January 2011), 74-87. DOI: http://dx.doi.org/10.1109/TVCG.2010.25