Nicholas Jiang COS 426: Computer Graphics

From Pictures to Paintings: A Tool for Artistic Styling and Visualization

Introduction

From cavemen to Monet to Picasso, painting and painters have been a part of human culture for thousands of years. In this project, I sought to continue this trend by transitioning the brush stroke style to the digital screen. Though I am not a painter myself, I enjoy viewing paintings, particularly those of the Impressionist or Post-Impressionist style employed by artists such as Claude Monet and Vincent Van Gogh respectively. Indeed, my inspiration for my custom filter from the first assignment was Vincent Van Gogh's Starry night, in which the brushstrokes follow the curves of not only the objects in the scene but the wind, the clouds, and the light from the moon and stars (Figure 1.).

For my final project, I sought to emulate this technique, creating digital paintings from input images. Coming into the project, I had little to no idea as to how this would actually be done. Since the paintings I appreciated seemed to flicker and almost flow along the brushstrokes, my initial idea was to create a 2D fluid simulation, where each pixel in an input image became a fluid particle and the particles were acted upon by the laws of fluid dynamics and also by forces pushing them along the curves and edges in the initial image. Right away in my attempts to implement this, I struggled. I tried working in an iPython Notebook since I found that there were numerous Python modules that could be used to help me through the process, such as ones that would calculate the gradient of the input image which was necessary in identifying the edges in the picture. The only issue with this was that I have little experience in Python. As such, I quickly found myself to be in way over my head and decided that a change of direction was necessary.

In my research looking for other potential ideas, I stumbled upon a paper written by Professor Aaron Hertzmann from NYU in 1998 (he may no longer be a professor there) title Painterly Rendering with Curved Brush Strokes of Multiple Styles. Indeed, this paper was the one referenced in the first assignment to create a Paint filter. I had implemented a small portion of it during the first assignment for that filter, namely drawing circles of different colors and sizes which yielded a somewhat artificially painted result. For this assignment, I sought to implement the entirety of Professor Hertzmann's rendering technique and to build upon it in some fashion, though I was not entirely sure how the latter portion of my goal would be realized.

As you will see, I believe that I achieved my goal of implementing a tool that could be used to render normal images in a painted fashion. More so, working from Professor Hertzmann's paper the tool I coded is able to simulate a variety of artistic styles ranging from Impressionism to Pointillism to Watercolor. Finally, I feel that I was also able to take the project one step further by adding the ability to visualize the "painting" process through the creation of GIFs that show the actual stages the digital canvas goes through before yielding its final result.

Figure 1. *Starry Night* by Vincent Van Gogh

Methodology

Before diving into the details of my implementation, I will give an overview of the tool's use and high-level properties. The basic idea is that using a base image as a reference we will be able to create a stylized rendering that gives a painted appearance with distinguishable brushstrokes, without distorting the initial image to be unrecognizable (unless that is the goal in which case that is achievable too). In my research I found that a variety of other implementations existed but the others all had features that were not ideal. Some sought to simulate the physical properties of the brush upon a paper/canvas-textured surface and while this approach yielded impressive results it had the issue of being very computationally expensive as can be expected by performing an actual simulation of the physical action of painting a picture. Other simpler techniques often had the issue of using uniform brushstrokes, either all of the same size, direction, or other property which yielded somewhat artificial results that seemed far too rigid to be appreciated as true artwork. The way the technique I employed differs from these other approaches is precisely in the details highlighted above that the other methods lacked in.

Hertzmann's method is a relatively inexpensive way of rendering a picture that captures a painted look due to its variation in brush size and the fact that the brush strokes are curves as opposed to straight lines. Similar to how many artists actually create their physical paintings, this technique works in multiple passes, painting a few layers of brush strokes on top of each other as opposed to creating the entire resulting picture in a single pass through the canvas. Furthermore, it draws from traditional painting technique in that the earlier layers are rendered with larger brushes while the later layers are rendered with finer brushes, allowing for general objects to be captured early on then refined in later passes. A key feature of the implementation that helps to avoid the appearance of a more artificial result is the fact that the level of detail in terms of brush sizes is not uniform: the finer brushes in later layers are only used to refine areas where the difference between the pixels already on the canvas and the corresponding pixels from the source image is over a certain threshold, essentially limiting the finer brush strokes to areas where more detail is necessary.

While Hertzmann's paper gives a general overview of the process and even some pseudocode indicating how one would go about implementing it, the details of implementation were left up to the coder to decide what approach was best. My implementation is written in JavaScript, building off the foundation provided by the first assignment. I integrated it as another filter but it is more complex than

the other filters spanning about 600 lines and taking in 13 parameters. One of these parameters is the set of brush sizes, in terms of pixel radius, to be used in the rendering. This parameter is read in as a string in the form "x8x4x2," which would denote three brushes of radii 8, 4, and 2 pixels respectively. The filter first sorts the brush sizes in an array to ensure that the layers are drawn in order of decreasing brush size. Next a new image is created to act as the "canvas" in the rendering, after which we start the process of "painting" each layer, with one layer per brush size. For each layer, a new reference image is generated by applying a Gaussian blur with sigma equal to $\mathbf{f_s}$*$\mathbf{r}$, where $\mathbf{f_s}$ is some constant and **r** is the brush radius. The purpose of using a blurred image as the reference is that with wider brushes we only want to worry about capturing larger, more general colors as opposed to falsely representing a whole section of the image as a certain color just because the single pixel we sampled in that area was a different color from the rest.

Before we start painting the layer, we first calculate the difference grid, a 2-D array which stores the RGB distance between the canvas and the reference image (blurred image for that layer) at each pixel. The goal of this is to be able to tell which pixels differ from the reference image to the point that they must be painted over in this layer to capture the finer details. One key point to note is that in calculating the difference between the canvas and the reference image, the difference method is coded so that it will automatically return Number.MAX_SAFE_INTEGER when comparing the blank canvas to any other pixel, ensuring that all potential strokes will be painted in the first layer, preventing large "holes" in the result. After calculating the difference grid, we iterate through the grid points spaced $\mathbf{f_g}$*$\mathbf{r}$ away from each other in the image (where $\mathbf{f_g}$ is a constant) and check whether the average RGB difference of pixels within the box of radius $\mathbf{f_g}$*$\mathbf{r}$ is greater than a predefined threshold **T**. We space out the checks since for a brush of radius **r** it is unnecessary to draw a stroke starting at every pixel. If the difference is greater than **T**, this means that more detail is needed and so we "make" a stroke starting at that x and y location. A key point is the fact that in "making" a stroke we do not actually paint it on the canvas, all strokes in a layer are calculated before any of them are actually rendered.

The defining feature of strokes in Hertzmann's method as opposed to other similar attempts is their property of curvature and variable length that allows for more natural appearing brush strokes which follow the contours of the original image and extend from natural starting points to natural ending points instead of having roughly regularly-placed, short, straight strokes. To create a stroke, first we calculate its color, as each stroke is of a single color, by sampling the reference image at that location and then randomly adjusting the hue, saturation, lightness, and r, g, b values by $\mathbf{j_h}$, $\mathbf{j_s}$, $\mathbf{j_v}$, $\mathbf{j_r}$, $\mathbf{j_g}$, and $\mathbf{j_b}$. In Hertzmann's paper the value is adjusted but I ended up altering the lightness instead since adjusting the value occasionally yielded odd-looking results. After picking the color, we define control points that are used to specify how and where the stroke will be rendered on the canvas. The way these control points are chosen is perhaps the most important factor in making the image appear like an artistic rendering of the initial image. To create each control point we take the previous point (starting with the initial x and y positions) and calculate the gradient of the image at that location. This gives us the direction of greatest difference from the current pixel, indicating that an "edge" runs normal to this direction. As such we choose the direction perpendicular to the gradient as the direction in which the next point will be placed. Since there are two normal directions, we choose the one that minimizes the curvature of the stroke (i.e. so that the stroke continues to curve in the same general direction). The next control point is placed at distance **r** away from the previous control point in the calculated direction. This process is continued until either: the point is off-screen, the number of control points for this stroke is above **minL** and the color at this location differs from the stroke color by more than it differs from the canvas, or the number of control points for this stroke is above **maxL**. We then store the stroke in an array where each stroke is defined by its radius, color, and set of control points.

After calculating all the strokes to be placed we then iterate over them and actually render them on the canvas. Since the strokes are calculated from top left to bottom right, rendering them one over

another in the order they were calculated would yield regularities that would appear artificial in the final image. To address this there are two solutions: use the "Painter's Algorithm" and randomly shuffle the array and then paint them all back to front or use a z-buffer and randomized z-values to only ever render a pixel at a location if it is the "front-most" pixel so far at that location. Ironically enough, even though we are seeking to make a painting we neglect the "Painter's Algorithm" and use the far more efficient z-buffer method. In rendering each stroke, I used different methods depending on how many control points the stroke had. This part was somewhat up to me since Hertzmann's paper merely suggests to use a spline approach. If the number of control points was one, I simply drew a circle of radius **r** at that location. If the number of control points was two or three, I drew lines connecting the points together in order by drawing **r** circles evenly spaced between the pairs of points. My logic in drawing **r** circles between each two points was that the points were placed **r** distance apart so my approach should yield approximately one circle per pixel distance between the points, leading to smooth looking lines. Finally, for strokes with at least four control points, I used a cubic b-spline as described in lecture to generate a smooth curve approximately following the control points, drawing **r** circles along the curve for every set of four points. The paint circle function I wrote also supports a constant **A** that defines how transparent each stroke is and used similar to how it was used in composite from assignment 1 to combine pixels from different images.

Once all strokes are rendered, that layer is complete and the program starts calculating and rendering the next layer until the final image is completed and returned. This process, while somewhat complicated in the details, is overall straightforward and logically makes sense in terms of achieving our goal by only adding detail where it is needed. While this algorithm allows for the painting of images in a traditional Impressionist style, its abilities extend far beyond that simple example. Since it is a function of 13 different parameters, an enormous variety of artistic styles can be rendered from a source image by simply changing the input parameters. To showcase this flexibility, my main Paint filter has all 13 parameters available to change, bounded by what seem to me to be logical min and max values. In addition, I implemented a few other filters showcasing specific artistic styles that have a fixed set of 13 parameter values defined and call the Paint function with those values and the input image. Most amazingly, since all of the parameters have physical meaning and are largely independent from one another, to get a mix of styles, such as a style between Impressionism and Expressionism, one merely needs to average their values for each parameter and run the Paint function with those averaged parameters. This particular feature allows for the exploration of styles beyond those seen in traditional painting history by potentially combining styles that existed centuries apart.

In making this tool I added lots of small optimizations as, though it is less expensive than simulating a physical brush on paper, the Paint operation can take anywhere between 10 seconds to a minute in practice (with my optimizations). My optimizations include: calculating the b-spline weights once per layer since the weighting and spacing of points will be the same for each layer, calculating the difference grid and image gradient once per layer, and only HSL converting if the HSL values were actually going to be changed.

Up until this point, everything I did had already been implemented before by Hertzmann. Though I did need to figure out how exactly to work out the details, which was actually very difficult at times, his paper guided my general process and, as such, I wanted to add something of my own design or integrate some other technique/feature from someone else. One of the things I was considering was adding more variation to each individual paint stroke, to somehow capture the texture/light effects present in real paintings. Upon researching this area I found the paper Fast Paint Texture which looked promising as it gave a way to add height mapping and using Phong shading to make the strokes more realistic… but it turns out that that was actually also written by Professor Hertzmann. In fact, looking at his web page he seemed to be the guru of stroke-based, non-photorealistic rendering, having published

a 173-page dissertation on the topic (and other NPR/imaging techniques). As such I was left somewhat lost as to what I could add.

One thing that I started to think about arose after trying to explain the idea behind my final project and Hertzmann's method to multiple different people, including those such as my family members who have little to no computer science knowledge. The process of painting in layers was fairly understandable, I believed, but people often got overwhelmed by the other points I tried to mention. As such, I decided to try and create a visualization of how the process worked that would highlight (not literally) the layering of strokes so people could understand the core concept. My idea was to create a GIF where each frame would be a snapshot of the canvas at a certain point and together they would illustrate the creative process. Initially I tried to create a GIF at 20fps with 1 frame per stroke… until I estimated that this would end up being an approximately 2-hour long GIF, not exactly feasible. To remedy this, I instead devised a method so that the canvas would be captured every fifth of the way through each layer and these images together would be assembled into a GIF. Actually implementing this required a lot of finagling and figuring out how the animated GIFs worked from Assignment 1. My implementation is somewhat of a hack, but then again isn't that what Computer Graphics is all about? In implementing this feature, I changed a couple things from the simple description I provided above. Firstly, since I had the special property that the distance from the canvas to any pixel was Number.MAX_SAFE_INTEGER and that the pixel values automatically clamped themselves I had set the original canvas pixels to be (0,0,0,0) in terms of RGBA. While this fixed the distance problem, I had to address this in my GIF creation by converting all pixels with an alpha value of 0 to white in the copy of the canvas I was adding to the gifEncoder. Another design decision I made was that I added 6 frames of the initial blank canvas, 6 more frames at the end of each layer, and an additional 12 frames at the end of the final picture so that it would pause at these points allowing the viewer to see the process at these checkpoints along the way.

Results and Discussion

Overall I am very happy with the way this project turned out. I feel that I definitely achieved my goal of creating digital paintings from images, indeed going far past my own expectations thanks to the help of Professor Hertzmann's work. Furthermore, I greatly enjoyed this project and being able to see the product of my work on the screen, especially in the form of the animated GIFs which I believe do a very good job of illustrating the multi-layered technique employed in this approach. Going further, in continuing this project I would look into a variety of different directions in which to expand. As previously mentioned, I believe adding more variety to the brush strokes themselves instead of simple circle-based curves would greatly enhance the visual effects. Additionally, I would like to be able to simulate other artistic styles not achievable by this multi-layered, brush stroke technique such as those employed in traditional Asian art that uses sparse calligraphic strokes to make impressive paintings or more shape/form focused imagery such as in Picasso's pieces. In following up on my visualization technique, ideally I'd want to make some way to visualize the actual painting of each stroke, perhaps by capturing pictures after each circle is painted and compiling them into a movie of sorts. Of course this would require an enormous number of frames so I'm not exactly sure how it would be achieved. Overall, I found this undertaking to be one of my favorite projects so far at Princeton and am very happy with what I was able to create. I have added some results from my project below for viewing. To try it out for yourself, go to http://njiang747.github.io/Painting426.

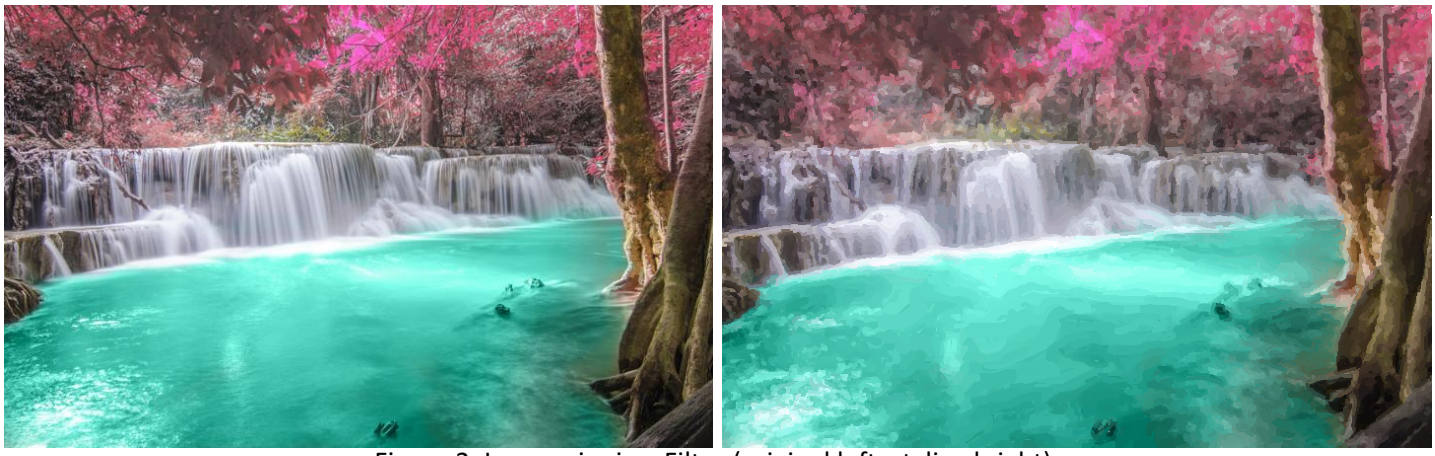

Figure 2. Impressionism Filter (original left, stylized right)

Figure 3. Expressionism Filter (original left, stylized right)

Figure 4. Color Wash Filter (original left, stylized right)

Figure 5. Pointillism Filter (original left, stylized right)

Figure 6. Water Color Filter (original left, stylized right)

Figure 7. Averaging the parameters of Impressionism (top-left) with Expressionism (top-right) yields a style that combines elements from both (bottom)