Comp Photography
Final Project

Brian Ellingson

Spring 2018

bellingson3@gatech.edu

Who Needs Real Paint?

An automated pipeline to generate painterly renderings by simulating brush strokes.

The Goal of Your Project

Original project scope: The original scope was to produce painterly, and pen and ink style renderings based on real photographs. Both reference papers, one by Hertzmann (painterly renderings), and one by Winkenbach and Salesin (pen and ink) present techniques that focus on rendering through simulating strokes. The idea was to develop a pipeline that could be used for either with slight modifications.

What motivated you to do this project? I took several art classes through Junior High and High School and enjoyed them. This seemed like a fun project and it helped that it is a CP topic applied to something I am familiar with.

Scope Changes

- Did you run into issues that required you to change project scope from your proposal?

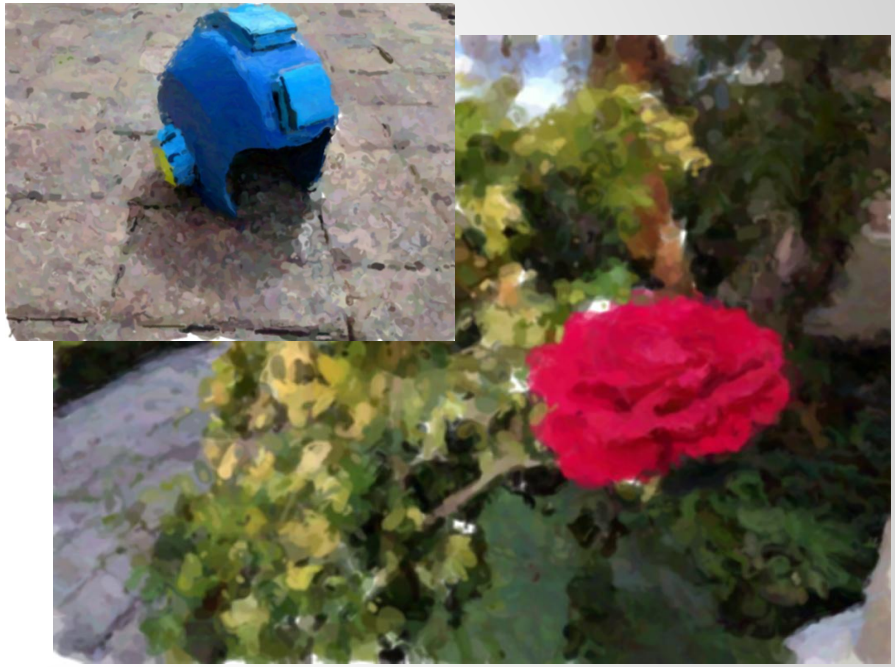
After developing the pipeline for painterly renderings I quickly realized that while the pen and ink paper used similar broad concepts, the intricacies of their methods were significantly different. It would have been much more than “slight” modifications to make a pen and ink rendering pipeline from my current implementation and stay true to the approach taken by Winkenbach and Salesin.

- Give a detailed explanation of what changed:

Rather than reproducing results from both papers, I instead focused my efforts on reproducing Hertzmann’s pipeline.

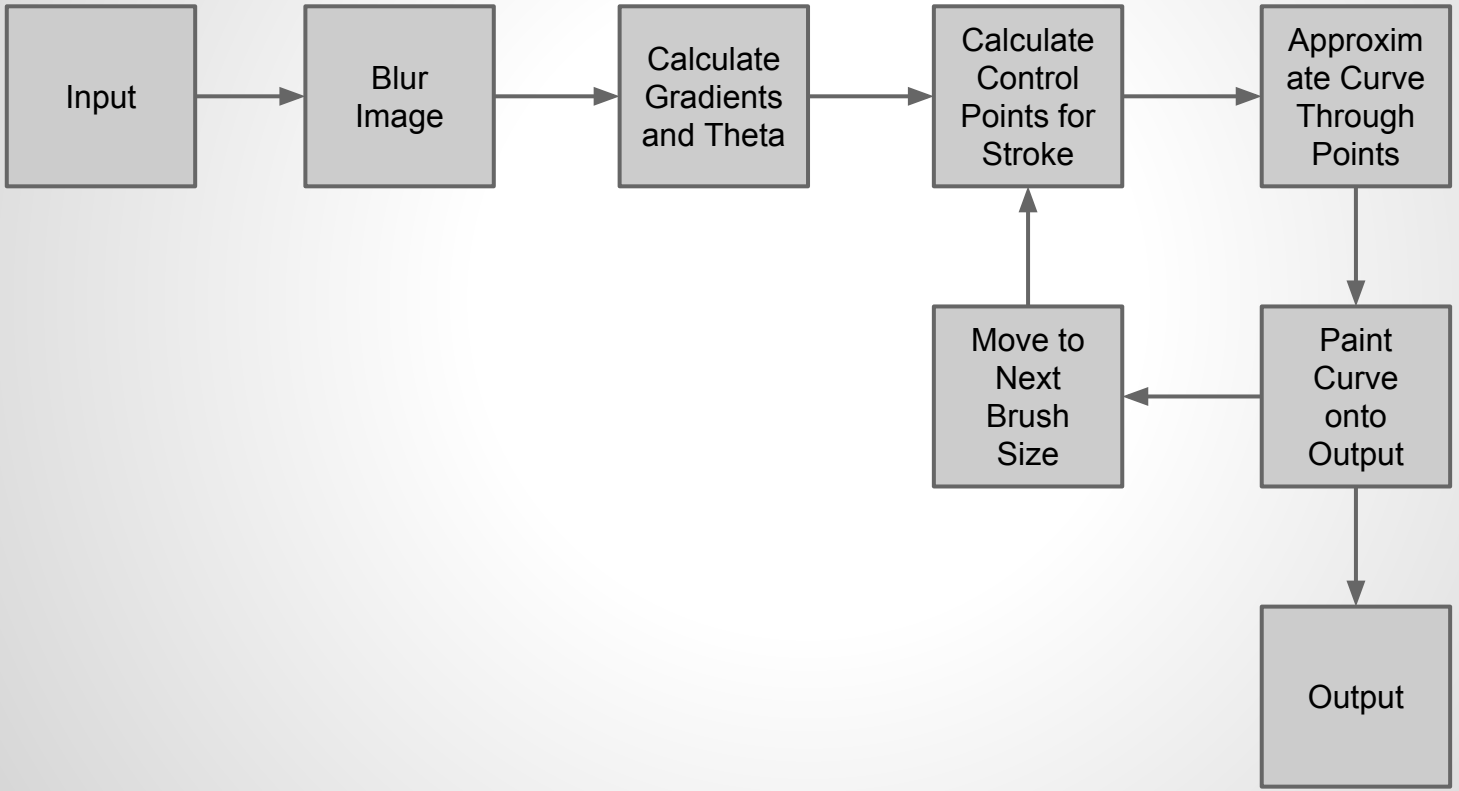
Showcase

Input



Output

Project Pipeline



Project Pipeline

Parameters Provided by User

- Brush Sizes: provided as a list of brush radii in pixels starting with the largest
- Max Brush Stroke Length: The maximum length of each brush stroke in pixels
- Control Point Distance Modifier: A float value which is multiplied by the brush radius to determine the distance between control points when calculating a brush stroke. A small value like 1.0 tends to produce strokes that better follow the contours, while 2.0 produces longer more curved strokes that tend to extend over color boundaries.
- Paint Opacity: A float value between 0 and 1.0 that affects the opacity of each brush stroke as it is painted on the canvas. A lower value will have more of the previous strokes show through the top layer strokes.
- Color Jitter: A value to add an element of randomness to the color of the brush stroke. A random number (-j, j) inclusive is added to each color channel of the brush stroke.
- Feathering: A float value used as the sigma for The Gaussian Blur on each stroke. With feathering, even when the max value is used for opacity, the outer edges of the brush stroke will be semi-transparent.
- Color Threshold: This is the maximum allowed color difference between the brush stroke starting point and the next possible control point. A higher value trends toward longer strokes that extend past color boundaries.
- Window Size: This is the size of the windows the image is separated into. It also dictates how many strokes of each brush size are made as there is one stroke per brush size per window.

Project Pipeline

The input image is first blurred to blend some of the contrast areas together and reduce fine details that may contribute to premature termination of brush strokes. The Sobel gradients are then calculated for the blurred image. The gradients are then used to create a “theta” matrix that represents the angle of the normal of the gradient at each given point. An empty image is also created with the same size as the original to serve as an empty canvas. The render() function is then called. The render() function iterates through the list of brush sizes determined by the user. The image is separated into equal sized windows. There will be one brush stroke starting point per window per brush size. On the first and largest brush size, the starting point of the stroke is a random point within its window. On subsequent brush sizes, the starting point is the point within the window with the largest difference between the original image and the painterly image being rendered. Each brush stroke starting point is then passed to the calculate_control_points() function. The next control point is a certain distance in the direction given by the theta matrix where distance = brush_radius * distance_modifier. Because there are two directions normal to the gradient (theta and theta + pi), the angle with the smaller difference from the previous angle is chosen to help prevent sharp angles in the brush strokes.

Project Pipeline

A brush stroke is terminated if it reaches the designated maximum length, or if the color of the next control point exceeds the given difference threshold. Once a set of control points is calculated for each brush stroke, they are iterated through in a random order to prevent an apparent uniformity. Each set of control points is then passed to the get_curve() function. The get_curve() function approximates a spline curve using scipy.interpolate and returns a series of points along a curve that are then passed to the paint_stroke() function. The paint_stroke() function utilizes cv2.polyline to draw the stroke onto a temporary black image with a float value of 1.0 . A Gaussian Blur is applied to the temporary image to induce feathering on the outer edges of the stroke. The temporary is then multiplied by the opacity modifier. This temporary image is then used as a weight for applying the stroke to the painterly rendering. This is then repeated for each brush size to produce the final output image.

Demonstration: Result Sets

This is the original image used as an input. When we moved into the house there was a small rose bush that we thought was dead. We trimmed it down to nothing but a stump, and this little trooper sprouted from there, and bloomed. The photo was taken using Pixel XL smartphone using automatic settings. The next few slides I will be showcasing some of the different renderings made using this photo.

Demonstration: Result Sets

The parameters used for this rending are as follows:

Brushe sizes: [16, 8, 4, 2]
Max length: 300 (in pixels)
Control Point Distance: 2.0
Paint Opacity: 0.6
Color Jitter: 10
Feathering: 5.0
Difference Threshold: 60
Window Size: 16

With these parameters, the output mostly resembles an acrylic painting.

Demonstration: Result Sets

The parameters used for this rending are as follows:

Brushe sizes: [16, 2]
Max length: 300 (in pixels)
Control Point Distance: 3.0
Paint Opacity: 0.2
Color Jitter: 10
Feathering: 9.0
Difference Threshold: 60
Window Size: 16

The intent here was to produce a watercolor rendering by using low opacity, high feathering and a smaller number of total brush strokes. The overall style is right, but a proper watercolor would still have a bit more detail.

Demonstration: Result Sets

The parameters used for this rending are as follows:

Brushe sizes: [8, 4, 4, 2]
Max length: 300 (in pixels)
Control Point Distance: 3.0
Paint Opacity: 0.9
Color Jitter: 5
Feathering: 1.0
Difference Threshold: 35
Window Size: 8

This is an attempt to produce an oil painting look. The high opacity, lower difference threshold, and less feathering help maintain more of the detail from the original images and give the look of a thicker more concealing paint.

Demonstration: Result Sets

This is the original image used as an input. I made this Megaman helmet for Halloween several years ago. It is sitting on the back patio and the shadow is caused by an overcast sun. This photo was also taken with my Pixel XL smartphone using all of the automatic settings.

Demonstration: Result Sets

The parameters used for this rending are as follows:

Brushe sizes: [16, 8, 4, 2]
Max length: 300 (in pixels)
Control Point Distance: 1.0
Paint Opacity: 0.8
Color Jitter: 5
Feathering: 5.0
Difference Threshold: 60
Window Size: 16

While the helmet is the obvious focus of this photo, I feel that the pattern of the brick pavers was an ideal addition for this type of rendering. The borders produced an excellent brush stroke effect.

Demonstration: Result Sets

This is the original image used as an input. This was a photo at the beach shortly before a shore dive in Fort Lauderdale just North of Commercial Pier. If you look very closely you can just barely make out part of the pier in the distance.

Demonstration: Result Sets

The parameters used for this rending are as follows:

Brushe sizes: [16, 8, 4, 2]
Max length: 300 (in pixels)
Control Point Distance: 1.0
Paint Opacity: 0.8
Color Jitter: 5
Feathering: 5.0
Difference Threshold: 60
Window Size: 16

I like the sky in this result, but I wasn't very please with the sand on the beach. The numerous shadows from foot prints cause the dark splotches. Also I hoped for the horizon to have more straight lines.

Demonstration: Result Sets

This is the original image used as an input. This is a photo of Emma. Emma is a dog. Emma does dog things on the rare occasion that she gets out of bed. She can be seen here in her standard position.

Demonstration: Result Sets

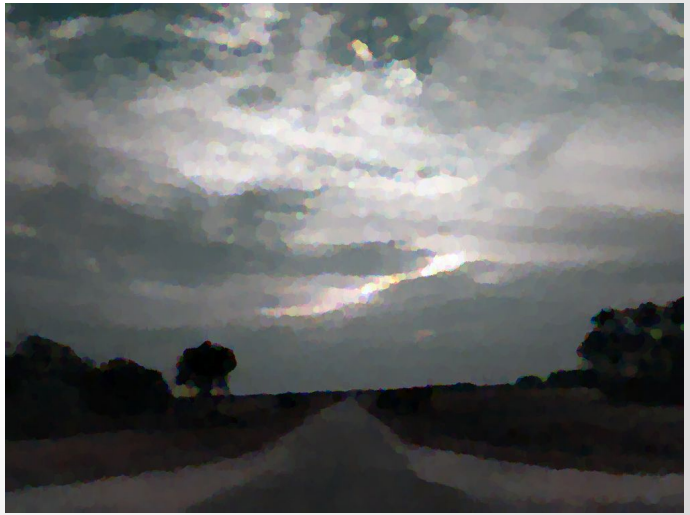
The parameters used for this rending are as follows:

Brushe sizes: [16, 8, 4, 2]
Max length: 200 (in pixels)
Control Point Distance: 1.0
Paint Opacity: 0.8
Color Jitter: 5
Feathering: 4.0
Difference Threshold: 60
Window Size: 8

I’m not too excited about this result. It has less of a painted appearance than the other results and looks more like it was a simple filter rather than algorithm to paint by simulated brush strokes.

Project Development

Before even diving into the technical paper by Hertzmann, I took a naive approach at creating a painterly rendering through randomization. This was mostly just for fun but also to get the creative juices flowing and knock out some of the simpler parts of the code such as reading and writing images. I separated the color channels. I specified a target similarity metric. I repeatedly drew a circle of random radius 2 - 5 in a random location with a random luminance. If the circle made the sketch more similar to the original image it would stay, otherwise it would thrown out. Once an image for each color channel was produced that met the target similarity they were recombined to make the final output. The results were surprisingly better than I had expected and it was a big motivator to get me started on the actual project. The scene is from a nearby semi-wooded area known as “The Compound”. It land was owned by a development company that went bankrupt shortly after building all of the roads for a planned community. Now it is a popular place for off-road vehicles, and ultralight aircrafts.



Project Development

Overall, development went relatively smoothly, but there were a couple hurdles and minor hiccups that caused me problems in the process. Calculating the theta values involves taking the arc tangent of the x gradient divided by the y gradient. In certain spots the y gradient has a value of zero causing an error. To alleviate this problem I simply add a negligibly small value to the y gradient. After that, it seemed to be working smoothly. All of the vertical and horizontal contours of the image were being followed by the brush strokes, but the diagonals were wrong. The addition of a negative sign solved this problem. I didn't have a chance to thoroughly examine the root cause of this problem, but I suspect it is related to the differences in domain and range between the gradients in the image space vs the np.arctan() function. The next problem was converting the control points to an approximated curve. My first attempt was using numpy.polyfit() and numpy.polyval(). It worked some of the time, but it would always cause warnings and errors when the control points exhibited a complicated path. It would also cause odd behavior for near horizontal lines causing them to extend much too far. I finally realized that the cause of the problem is I needed to make a function of time with x,y output rather than a function of x with y output. This lead me to scipy.interpolate() which does just that. The next big problem was finding the proper parameters to produce the desired results. In many cases the rendering can take a large amount of time because it is not the most optimized code, so there was a lot of waiting involved. Another I faced was determining where a brush stroke should start.

Project Development

The paper wasn't completely clear on how they determine where a brush stroke should start. There was a small level of assumption involved when I opted to place one stroke per window on the jitter grid.

One of the things from the Hertzmann paper I was unable to implement was a minimum stroke length. The need for a minimum stroke length can easily be seen in my results of my dog Emma as many of the finer strokes appear to be just circles. They were unclear as to how the minimum stroke was enforced. Enforcing a maximum stroke length is simple and straightforward as you simply terminate the stroke when the limit is reached. My guess for one way to implement the minimum length would be to prepend to the start of the stroke in the opposite direction if the stroke is terminated before the minimum was met. I struggled to think of a graceful way to implement this that wouldn't further increase the time complexity overhead of the pipeline.

If I had more time, I would dedicate it to optimizing the code to run faster. If I had even more time than that I would start exploring ways of simulating not only the brush stroke but the physics of different kinds of paint and how it spreads when it is applied in different situations. For example, watercolor paints will spread into wet areas readily.

Computation: Code Functional Description

Painterly Class:

This is the object that encompasses all of the functionality of the pipeline.

__init__():

This is the constructor method for the Painterly class, it receives an image object (np array), and the tunable parameters stated earlier. Upon instantiation it creates the empty canvas, and the theta matrix from gradients. The theta matrix is calculated as the arctan of -x gradient / y gradient.

calc_control_points():

This method receives a start point and a brush radius. From the start point it moves in the direction given by the theta matrix at that point for a distance of r * D where r is the brush radius and D is the control point distance modifier. If the new point is within the color threshold of the starting point, the current brush color differs from the original image less than the current painting differs from the original image, and the maximum stroke length has not been met, the new point is added to the list of control points. The root sum squared difference is used to calculate the color similarity. If the angle given by the theta matrix plus pi is closer to the previous angle, then theta plus pi is used for the next angle. This helps alleviate sharp turns in the brush strokes.

Computation: Code Functional Description

```
30      def calc_control_points(self, (x, y), r):
31          points = [(x, y)]
32          moving = True
33          start_color = self.image[y, x]
34          angle = self.theta[y, x]
35          while moving:
36              try:
37                  prev_angle = self.theta[y, x]
38                  dx = np.cos(angle)
39                  dy = np.sin(angle)
40                  x += int(dx * r * self.D)
41                  y += int(dy * r * self.D)
42                  new_color = self.blurred[y, x]
43                  skt_color = self.sketch[y, x]
44                  angle = self.theta[y, x]
45                  if np.abs(angle - prev_angle) > np.abs(np.pi + angle - prev_angle):
46                      angle += np.pi
47                  if len(points) >= (self.L / r * self.D):
48                      #print "Max stroke length met"
49                      moving = False
50                  elif self.rssd(start_color, new_color) > self.CT:
51                      #print "Color difference threshold exceeded"
52                      moving = False
53                  elif self.rssd(start_color, new_color) > self.rssd(start_color, skt_color):
54                      #print "Sketch color is closer match"
55                      moving = False
56                  else:
57                      points.append((x, y))
58              except:
59                  #print "Target index out of bounds"
60                  moving = False
61          points = np.asarray(points)
62          return points
```

calc_control_points() is easily the central focus of the project as this is where the control points that dictate the shape of the stroke are determined.

Computation: Code Functional Description

```
65      def get_curve(self, control_points):
66          deg = 3
67          if len(control_points) <= 3:
68              deg = len(control_points) - 1
69          if len(control_points) <= 1:
70              return control_points
71          tck, u = interpolate.splprep([control_points[:, 0], control_points[:, 1]], s = 0, k = deg)
72          unew = np.arange(0, 1.01, 0.01)
73          out = interpolate.splev(unew, tck)
74          stroke = np.zeros((len(out[0]), 2), dtype=np.int32)
75          stroke[:, 0] = out[0]
76          stroke[:, 1] = out[1]
77          return stroke
```

get_curve():
This method works in conjunction with calc_control_points() to generate a set of points that represent a curve. The degree used for approximating the curve is set to 3 unless the number of control points is 3 or less. If there is only a single control point, the approximation is skipped. This mostly a wrapper method for scipy.interpolate to approximate a spline curve. The points returned by get_curve() are used to actually draw the brush stroke. The bulk of the scipy.interpolate implementation is taken straight from the scipy documentation.

Computation: Code Functional Description

rssd():

This is a simple convenience function to give the root sum squared difference for two numpy arrays. Data type conversion is handles inside the function to avoid accidental data loss from overflow in unsigned types. This is used for calculating the similarity metric for colors.

max_diff():

This method receives two images and returns the point given as an offset from the center where the two images different the most. This is used for determining stroke start points within the grid windows. When all the mean difference is equal to the max difference (the two images match) a random offset is generated and returned.

get_strokes():

This method iterates through all of the grid windows for a given brush size and returns a list of all of the calculated strokes.

Computation: Code Functional Description

paint_stroke():
This method takes the list of points along a curve and paints them onto the canvas. It uses the cv2 method polylines() to draw the curve with color 1.0 onto a temporary image of 0 values. The Gaussian Blur is applied to the temporary image with the feathering parameter as the sigma value. The temporary image is then multiplied element-wise by the opacity parameter. The temporary image is then used as a weight to apply the brush stroke to the working sketch. A 0 value on the temporary would mean no change to the working sketch while a 1.0 value would mean the color of the brush stroke would fully cover the spot.

```
113     def paint_stroke(self, stroke, r):
114         x, y = stroke[0, 0], stroke[0, 1]
115         jitter = np.random.randint(-self.J, self.J + 1, 3)
116         base_color = self.image[y, x].astype(np.int32) + jitter
117         base_color[base_color < 0] = 0
118         base_color[base_color > 255] = 255
119         base_color = base_color.astype(np.uint8)
120         temp = np.zeros(self.image.shape, dtype=np.float64)
121         cv2.polylines(temp, [stroke], False, (1., 1., 1.), thickness= r * 2)
122         temp = cv2.GaussianBlur(temp, (5, 5), self.F)
123         temp = temp * self.O
124         self.sketch = temp * base_color + (1 - temp) * self.sketch
```

Computation: Code Functional Description

get_windows():

This is another convenience method. You pass in the center point of a window and a given offset (window size divided by 2), and it returns both the window cutout from the original image and the cutout from the current working sketch. These are used with the max_diff() function to determine the brush stroke starting point.

render():

This is essentially the controller method. Once the Painterly object is instantiated properly this is the only method that needs to be called in order to generate the painterly rendering. It can be called with or without parameter. The show_img parameter when set to True will display the progress of the image being rendering showing as each brush stroke is placed. It can be fun to watch but because it renders the image at each brush stroke it can add to the amount of time the process takes.

Any additional details?

- Last chance reminder, we are looking forward to seeing your best work.
- Discuss things that do not seem to belong anywhere else.

Resources

https://mrl.nyu.edu/publications/painterly98/hertzmann-siggraph98.pdf

http://web.cs.ucdavis.edu/~ma/SIGGRAPH02/course23/notes/papers/Winkenbach.pdf

https://docs.scipy.org/

Appendix: Your Code

Code Language: Python

Modules: cv2 (opencv), numpy, scipy

List of code files:

- *main.py*
- *painterly.py*

Image resources in ./images/input/:

- *emma.jpg*
- *beach.jpg*
- *megaman.jpg*
- *rose.jpg*
- *flamingo.jpg*

Credits or Thanks

I would like to thank my wife for putting up with me constantly saying, “No I can’t leave the house today, I have to do computer stuff.” and “Help me take this picture, your camera is confusing.”