Comp Photography
Final Project

Ronak Patel

Spring 2019

rpatel88@gatech.edu

Painterly Rendering

The goal of this project is to imitate different artistic styles using an input image and brushing strokes onto a digital canvas. The brush strokes are computed and performed programmatically. Artistic style being explored are Impressionism, Expressionism, Color Washing, and Pointillism

Example input

Example output

The Goal of Your Project

Original project scope: The original scope of this project is to imitate different artistic styles using programmatically calculated curved brush strokes on a canvas. Styles are created by using a series of layer and brush sizes to capture broader and fine details. The goal is to emulate impressionist, expressionist, color wash, and, pointillist styles from the paper “Painterly Rendering with Curved Brush Strokes of Multiple Sizes” by Aaron Hertzmann.

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

What motivated you to do this project?

My motivation for the project was my interest in each of the artistic styles being imitated. I enjoy looking at these artistic styles in museums and art galleries. Furthermore, the project involved using filtering principles taught in this class and the ability to “paint” programmatically which was new to me.

Project inputs and outputs can be found here:
https://drive.google.com/open?id=1_qnjOjAdUl0PaKCO4RpRAYco4P9VmMoS

Scope Changes

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

  Originally the scope of the project was reduced to the implementation of just the impressionist style due to difficulties faced with producing the correct brush strokes and with painting the entire bounds of the canvas.

- Give a detailed explanation of what changed

  The impressionist style painting required the simplest parameters. However, once this was at an acceptable state, other styles were explored by these adjusting parameters. While these styles did not implement color jitter or alpha blended brush strokes correctly, an output for each style was still produced with the parameters specified in the paper. The outputs did not truly reflect the artform.

Showcase

INPUTS

OUTPUTS

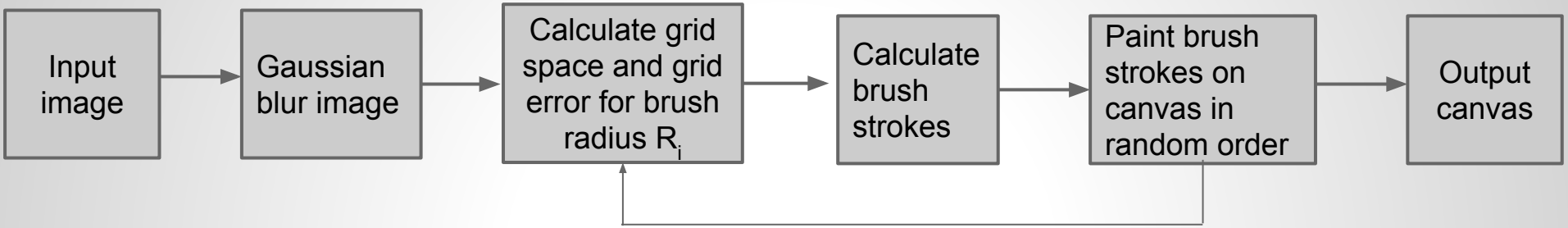
Colorist Wash

Expressionist

Impressionist

Pointillist

Project Pipeline



- An image is input into the program with initialized style parameters
- A reference image is created by using a gaussian blur with a kernel size calculated from the standard deviation of the brush stroke radius.
- A grid space is calculated as a function of the grid space parameter and the radius. Using this, a start point for the strokes is calculated by finding where the max error is within a grid space, and passed into the makeSplineStroke() method to calculate splined brush strokes
- Brush strokes are calculated using gradient directions of the reference image
- Brush strokes are painted on the canvas for the radius specified.
- Process is repeated for all other brush stroke radii and the final output canvas is outputted

Demonstration: Result Sets

Here are outputs for the four painting styles being implemented within the project. The input is a regular image and the outputs are stylized versions of the input image. We can see that some of the styles look the very similar.This is thought to be a result of the initial style parameters passed and the “painting” library (cv2.polylines) used to create the strokes. Furthermore, there are areas of the output that are white when they should not be. This is thought to be due to improper bounding of the grid in relation to the reference image size.

INPUT

Colorist Wash Style

Expressionist Style

Impressionist Style

Pointillist Style

Demonstration: Result Sets

INPUT

Colorist Wash Style

Expressionist Style

Impressionist Style

Pointillist

Demonstration: Result Sets

INPUT

Colorist Wash Style

Expressionist Style

Impressionist Style

Pointillist Style

Demonstration: Result Sets

SAMPLE PAPER INPUT

Colorist Wash Style

Expressionist Style

Impressionist Style

Pointillist Style

Project Development

My project began by reading the paper “Painterly Rendering with Curved Brush Strokes of Multiple Sizes”. After an understanding of the paper was grasped, I began stepping through and implementing the pseudo-code shown in the research paper. The first snippet of the pseudo-code is shown below:

```
function paint(sourceImage, R_1 ... R_n)
{
   canvas := a new constant color image

   // paint the canvas
   for each brush radius R_i,
         from largest to smallest do
   {
      // apply Gaussian blur
      referenceImage = sourceImage * G_(f...Ri)
      // paint a layer
      paintLayer(canvas, referenceImage, R_i)
   }

   return canvas
}
```

Essentially here, I was initializing a painting canvas and then looping through each brush radius and passing a gaussian blurred image into the paintLayer method. This is done because a brush strokes of the largest size are painted first and then smaller sizes are painted to highlight details in the image. Although the implementation on this section was straightforward, I found that I was misinterpreting the referenceImage equation by defining the Gaussian kernel by the standard deviation of the standard deviation of all the radii multiplied by the fs (SigmaX = f_s*std(radii)) instead of just the standard deviation of the radius being used multiplied by f_s (SigmaX = f_s*radius).

Project Development

Next, the paintLayer method was created following the procedure outlined in the snippet shown on the right. There were difficulties implementing this portion of the program because it was not clear how to correctly calculate the difference image defined in the paper as “D”. During my first implementation, I incorrectly defined D since I did not account for the initial value that filled my canvas with. It was recommended to fill the canvas with a large positive number but it was not clear that this had to be taken out when finding the difference image. Furthermore, finding the correct way to implement the grid area, “M”, was more complicated than expected since the stepsizes affect the grid area along the edges of the image. As a result, “M” variable had to be slightly modified to account for the painting of the canvas near the edges in future implementations. Similarly, the areaError variable had to be recalculated near the edges where the area of the grid size was not exactly grid^2.

When moving onto calculating variables x_1 and y_1 given by the argmax operation, the variable bounds had to be defined so a coordinate system was relative to the grid area being evaluated. This was not explained well in the paper, but was necessary for a more accurate starting point to be passed in the makeStroke function.

```
procedure paintLayer(canvas,referenceImage, R)
{
   S := a new set of strokes, initially empty

   // create a pointwise difference image
   D := difference(canvas,referenceImage)

   grid := f_g R

   for x=0 to imageWidth stepsize grid do
      for y=0 to imageHeight stepsize grid do
      {
        // sum the error near (x,y)
        M := the region (x-grid/2..x+grid/2,
                         y-grid/2..y+grid/2)
        areaError := Σ_{i,j∈M} D_{i,j} / grid^2
        if (areaError > T) then
        {
           // find the largest error point
           (x1,y1) := arg max_{i,j∈M} D_{i,j}
           s :=makeStroke(R,x1,y1,referenceImage)
           add s to S
        }
      }

   paint all strokes in S on the canvas,
      in random order
}
```

Project Development (cont'd)

Next, the makeSplineStroke function was implemented to calculate the curved brush strokes to be painted. Research had to be done in order to correctly calculate gradient magnitude and gradient directions, but this was done by graying the reference image and passing Sobel filters in the x and y directions.

After experiencing errors in this section of the code, an additional check had to be implemented to avoid painting in areas that were out of the bounds of the canvas. Although this was not shown in the research paper, it was necessary to avoid errors returned from the program.

After my initial pass at implementing these three functions, my output looked like the image below for the impressionist style:

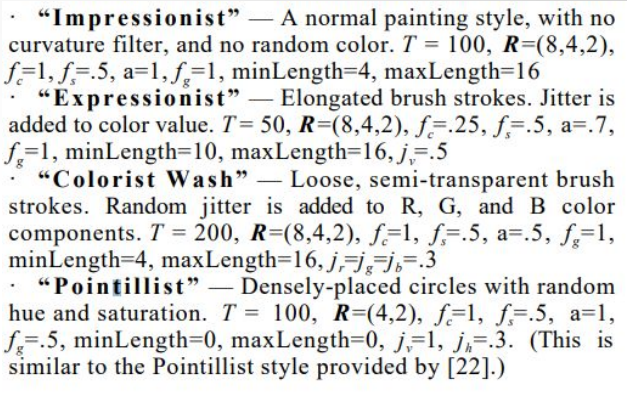
```
function makeSplineStroke(x0,y0,R,refImage)
{
   strokeColor = refImage.color(x0,y0)
   K = a new stroke with radius R
         and color strokeColor
   add point (x0,y0) to K
   (x,y) := (x0,y0)
   (lastDx,lastDy) := (0,0)

   for i=1 to maxStrokeLength do
   {
      if (i > minStrokeLength and
        |refImage.color(x,y)-canvas.color(x,y)|<
        |refImage.color(x,y)-strokeColor|) then
           return K

      // detect vanishing gradient
      if (refImage.gradientMag(x,y) == 0) then
        return K

      // get unit vector of gradient
      (gx,gy) := refImage.gradientDirection(x,y)
      // compute a normal direction
      (dx,dy) := (-gy, gx)

      // if necessary, reverse direction
      if (lastDx * dx + lastDy * dy < 0) then
        (dx,dy) := (-dx, -dy)

      // filter the stroke direction
      (dx,dy) :=fc*(dx,dy)+(1-fc)*(lastDx,lastDy)
      (dx,dy) := (dx,dy)/(dx^2 + dy^2)^1/2
      (x,y) := (x+R*dx, y+R*dy)
      (lastDx,lastDy) := (dx,dy)

      add the point (x,y) to K
   }
   return K
}
```

Project Development (cont’d)

I then made corrections to the Gaussian kernel, the difference image (D), the grid portion (M), and the areaError, my output looked like the image below:

Project Development (cont'd)

Finally, other the other styles had to be implemented by adjust the initial parameters in the program. Here, I ran into more difficulties when defining the bounds when producing the pointallist image. Since the f_g parameter was equal 0.5, the grid size for a radius of 2 (from the initial radii set [4,2]) would be floored to 0 by equation int(fg*radius//2). In order to overcome this issue, I put an additional check if the grid size was zero and adjust it to a value of 1. At this point, I was able to output an image for every style shown in the research paper, and I did so by matching the initial input parameters outlined in the paper (shown on the right) with exception to the jitter parameter j, which was not implemented because it was unclear where and how to implement this parameter.



- **"Impressionist"** — A normal painting style, with no curvature filter, and no random color. T = 100, $\boldsymbol{R}$=(8,4,2), f_c=1, f_s=.5, a=1, f_g=1, minLength=4, maxLength=16
- **"Expressionist"** — Elongated brush strokes. Jitter is added to color value. T= 50, $\boldsymbol{R}$=(8,4,2), f_c=.25, f_s=.5, a=.7, f_g=1, minLength=10, maxLength=16, j_v=.5
- **"Colorist Wash"** — Loose, semi-transparent brush strokes. Random jitter is added to R, G, and B color components. T = 200, $\boldsymbol{R}$=(8,4,2), f_c=1, f_s=.5, a=.5, f_g=1, minLength=4, maxLength=16, j_r=j_g=j_b=.3
- **"Pointillist"** — Densely-placed circles with random hue and saturation. T = 100, $\boldsymbol{R}$=(4,2), f_c=1, f_s=.5, a=1, f_g=.5, minLength=0, maxLength=0, j_v=1, j_h=.3. (This is similar to the Pointillist style provided by [22].)

What would I do differently?

Although the output for the impressionist style was close to the sample output from the research paper, the other styles were difficult to replicate and looked very similar to each other despite using the parameters and code outlined in the paper. If I had more time to better my output, I would experiment with using a different painting library and test more parameters for each painting style. I believe a stronger painting library will result in a more artistic finish, rather than just using curved polylines in OpenCV's library. I would also further experiment with the bounds of the grid section used to calculate area and define control points passed into the makeSplineStroke function. It is believed that improper bounds has led to white areas in some of the outputs shown.

Computation: Code Functional Description

```
import sys
import os
import cv2
import math
import numpy as np
import pandas as pd
from random import shuffle

def load_images_from_folder(folder):
    images = []
    for filename in os.listdir(folder):
        img = cv2.imread(os.path.join(folder,filename))
        if img is not None:
            images.append(img)
    return images
```

To start the program, libraries were imported and images were read using the load_images_from_folder method.

Computation: Code Functional Description

```
if __name__ == "__main__":
    imgs = load_images_from_folder("images")
    i=1
    for img in imgs:

        name = ("%d_impressionist_img.jpg" %(i))
        threshold = 100
        maxStrokeLength = 17
        minStrokeLength = 5
        fc = 1
        fs = 0.5
        a = 1
        fg = 1
        radii = [8,4,2]
        out = paint(img)
        cv2.imwrite(name, out)

        name = ("%d_expressionist_img.jpg" %(i))
        threshold = 50
        maxStrokeLength = 17
        minStrokeLength = 11
        fc = 0.25
        fs = 0.5
        a = 0.7
        fg = 1
        radii = [8,4,2]
        out = paint(img)
        cv2.imwrite(name, out)

        name = ("%d_colorist_img.jpg" %(i))
        threshold = 200
        maxStrokeLength = 17
        minStrokeLength = 5
        fc = 1
        fs = 0.5
        a = 0.5
        fg = 1
        radii = [8,4,2]
        out = paint(img)
        cv2.imwrite(name, out)
        name = ("%d_pointillist_img.jpg" %(i))
        threshold = 100
        maxStrokeLength = 1
        minStrokeLength = 1
        fc = 1
        fs = 0.5
        a = 1
        fg = 0.5
        radii = [4,2]
        out = paint(img)
        cv2.imwrite(name, out)

        i+=1
```

Next, style parameters were initialized for each style and the paint() method was called for each of the loaded images

Computation: Code Functional Description

```
def paint(s_img):
    canvas = np.ones(s_img.shape)*550
    print canvas.shape
    for radius in radii:
        print "painting radius ", radius
        ref_img = cv2.GaussianBlur(s_img,(0,0),fs*radius,fs*radius)
        out_img = paintLayer(canvas,ref_img, radius)
    return out_img
```

The paint() method was defined by taking a source image as an input. From here a canvas initialized with a large positive value and a reference image was defined for each radius (or brush stroke thickness). These parameters were then passed into the paintLayer() method.

Computation: Code Functional Description

```
def paintLayer(canvas, ref_img, radius):
    S = []
    step = int(fg*radius)
    if step//2<=0:
        h_step = 1
    else:
        h_step=step//2
    diff_img = img_diff(canvas,ref_img)
    for x in range(0,canvas.shape[1],step):
        for y in range(0,canvas.shape[0],step):
            x_min = max(0, x-h_step)
            y_min = max(0, y-h_step)
            x_max = min(canvas.shape[1]-1, x+h_step)
            y_max = min(canvas.shape[0]-1, y+h_step)
            M = diff_img[y_min:y_max+1, x_min:x_max+1]
            area_error = M.sum()/((x_max-x_min)*(y_max-y_min))
            if area_error>threshold:
                x1,y1 = np.unravel_index(M.argmax(),M.shape)
                x0 = x1%(h_step)+x_max-h_step
                y0 = y1%(h_step)+y_max-h_step
                s = makeSplineStroke(x0, y0, radius, ref_img, canvas)
                S.append(s)

    shuffle(S)
    for stroke in S:
        stroke = np.array(stroke)
        color = ref_img[stroke[0][1],stroke[0][0],:].astype(int).tolist()
        color.append(a)
        cv2.polylines(canvas,[stroke],False,color, radius)
    return canvas

def img_diff(canvas, ref_img):
    diff = canvas - 550 - ref_img
    return np.sqrt(np.sum(diff**2, axis=-1))
```

In the paint layer method a stroke array was initialized and a full grid step and a half grid step was calculated. Next, a difference image was calculated using the helper function shown at the bottom. Next, for each step in the x and y directions of the canvas, a grid was defined by the x_min, y_min, x_max, and y_max variables. These variables accounted for the different grid sizes near the end of the canvas using min and max operations. Area error as then calculated as a function of the grid area and the error was checked against a threshold value to see if the area should be painted. The painting stroke begins where at the point with the largest error so an argmax operation was performed and x and y coordinates relative to the entire canvas position were passed into the makeSplineStroke() method. The output of this method returns the strokes to be painted. These were shuffled and an alpha parameter was passed into the stroke color. Finally, these were painted using cv2.polylines.

Computation: Code Functional Description

```
def makeSplineStroke(x0,y0,R,refImage, canvas):
    strokeColor = refImage[y0,x0,:].astype(int).tolist()
    K = [[x0, y0]]
    x,y = x0, y0
    lastdx, lastdy = 0,0
    gray = cv2.cvtColor(refImage, cv2.COLOR_BGR2GRAY)
    # ksize = int(R*0.7)
    # if ksize<3:
    #     ksize = 3
    sobelX = cv2.Sobel(gray, cv2.CV_64F, 1, 0,ksize=5)
    sobelY = cv2.Sobel(gray, cv2.CV_64F, 0, 1,ksize=5)

    for i in range(1, maxStrokeLength+1):
        if i>minStrokeLength and abs(pixel_diff(refImage[y,x,:],canvas[y,x,:])) < abs(pixel_diff(refImage[y,x,:],strokeColor)):
            return K

        gx = sobelX[y,x]
        gy = sobelY[y,x]
        mag = math.sqrt(gx**2 + gy**2)
        if mag == 0:
            return K

        dx, dy = -gy, gx
        if lastdx*dx + lastdy*dy < 0:
            dx, dy = -dx, -dy
        dx = (fc*dx + (1-fc)*lastdx)/math.sqrt(dx**2 + dy**2)
        dy = (fc*dy + (1-fc)*lastdy)/math.sqrt(dx**2 + dy**2)
        x,y = int(x+R*dx), int(y+R*dy)

        if x>canvas.shape[1]-1 or y>canvas.shape[0]-1 or x<0 or y<0:
            break

        lastdx, lastdy = dx, dy
        K.append([x,y])
    return K
```

The make splineSplineStroke() method begins with defining a stroke color based on the reference image. Next, a vector was initialized to store stroke coordinate. x, y, lastdx, and lastdy variables were initialized to the splined stroke calculation. Similarly, sobel filters in the x and y directions were implemented to find gradient values and to do a magnitude check. Next a for loop was implemented to limit the stroke length of the stroke to be calculated. A check was put in place to make sure the color of the stroke does not deviate too much from the stroke control point. An image gradient magnitude check was put in place (mag==0), if this criterion is satisfied, further points for the spline would not be calculated and only the initial stroke point would be returned. Next, a check was done to see if it would be necessary to reverse the direction of the stroke by setting dx, dy = -dx, -dy. The dx, dy variables were then updated and used to filter the stroke direction and curvature using the curvature variable fc and the magnitude of dy+dx. New stroke points were then calculated based on previous x,y values, new dx, dy values, and the radius used for painting. Finally, the lastdx, lastdy variables were updated and the calculated stroke points (x,y) were appended to the stroke vector K.

Resources

- https://www.mrl.nyu.edu/publications/painterly98/hertzmann-siggraph98.pdf
- https://stackoverflow.com/questions/9482550/argmax-of-numpy-array-returning-non-flat-indices
- https://stackoverflow.com/questions/20271479/what-does-it-mean-to-get-the-mse-mean-error-squared-for-2-images

Appendix: Your Code

Code Language: Python 2 (can use Python 3 as well)

List of code files:

- painterly.py

Credits or Thanks

- Thank you professor and TA’s for all the help throughout the semester! Without you, I would not be able to implement a project like this!