Oil painting style for image and video

指導教授：莊永裕
資工三 B04902074 黃紹曄
資工三 B04902078 范延愷

Problem

我們希望可以把照片以及影片轉換成帶有油畫效果的感覺，並且採用兩種方法。

Approach1

總共需要兩個參數
Radius: 對每一個 pixel 所需要計算的圓範圍的半徑大小
Level of Intensity: 對於亮度的分類需要分多細（越大代表越細）

1. Algorithm
   首先，必須對每個 pixel 做下列動作，相當於將圓圈內所有 pixel 分進 Level of Intensity 這麼多個區間，並且將所有同區間的 rgb 值的總合記下來:

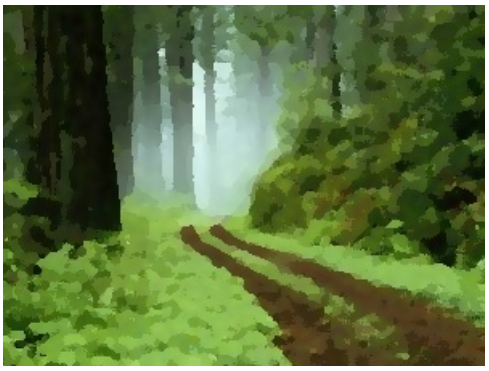
```
for (each pixel)
{
        for (each pixel, within radius r of pixel)
        {
                int curIntensity = (int)((double)((r+g+b)/3)*intensityLevels)/255.0f;
                intensityCount[curIntensity]++;
                averageR[curIntensity] += r;
                averageG[curIntensity] += g;
                averageB[curIntensity] += b;
        }
}
```

   接著找出最多元素每一個區間，並且將其對應的 rgb 值做平均，設定為這點的色彩。

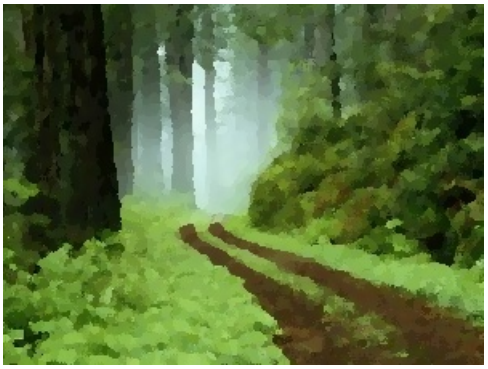
```
for (each level of intensity)
{
        if (intensityCount[i] > curMax)
        {
                curMax = intensityCount[i];
                maxIndex = i;
        }
}

// Step 3, calculate the final value
finalR = averageR[maxIndex] / curMax;
finalG = averageG[maxIndex] / curMax;
finalB = averageB[maxIndex] / curMax;
```

2. Result

(a)原始圖片

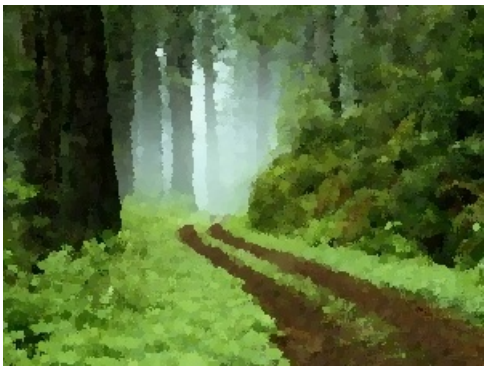


(b) (Radius, Level of Intensity) = (3, 10)



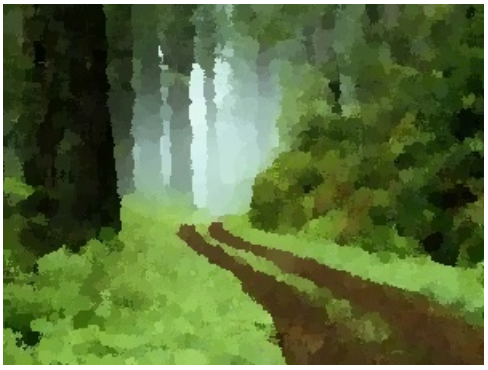
(c)(Radius, Level of Intensity) = (3, 20)

(d) (Radius, Level of Intensity) = (3, 40)



(e)(Radius, Level of Intensity) = (3, 100)

(f) (Radius, Level of Intensity) = (3, 200)



(g)(Radius, Level of Intensity) = (5, 100)

(h) (Radius, Level of Intensity) = (5, 200)

3. Conclusion

   從照片裡面可以看得出來幾件事，第一，當 Level of Intensity 增加之後邊緣的部分會因為分層分太細而不夠銳利，第二，radius 可以想象成筆刷的半徑，若是過大會讓高頻的部分資訊流失過度而像是一坨東西在上面。

   這個演算法的優點在於不複雜，並且在適當的參數下邊緣表現的很好;而缺點是看起來很像是用點上去的，沒有一般油畫人用筆’’畫下去”的感覺。

4. Reference

   http://supercomputingblog.com/graphics/oil-painting-algorithm/

Approach2

基於上面的結論，我們實作了第二個演算法，致力於展現”畫下去”的感覺。需要的參數只有筆刷半徑（R1-Rn）的陣列， 其他的參數直接寫在程式裡了。

1. Algorithm

   將筆刷的半徑由大到小排列，以 sigma = Ri 的 Gaussian Blur 的照片作為參考，依序在畫布（canvas）上塗上顏色（printLayer）。

```
function paint(sourceImage, R1 ... Rn)
{
   canvas := a new constant color image

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

   return canvas
}
```

   printLayer 的實作如下:

   以每一個 pixel 為圓心取半徑為 R 的範圍作為 grid，如果這個 grid 跟參考（referenceImage）同個範圍的 pixel 的差距大於 T，則以此 pixel 作為起點存下這一條線（makeStroke）。

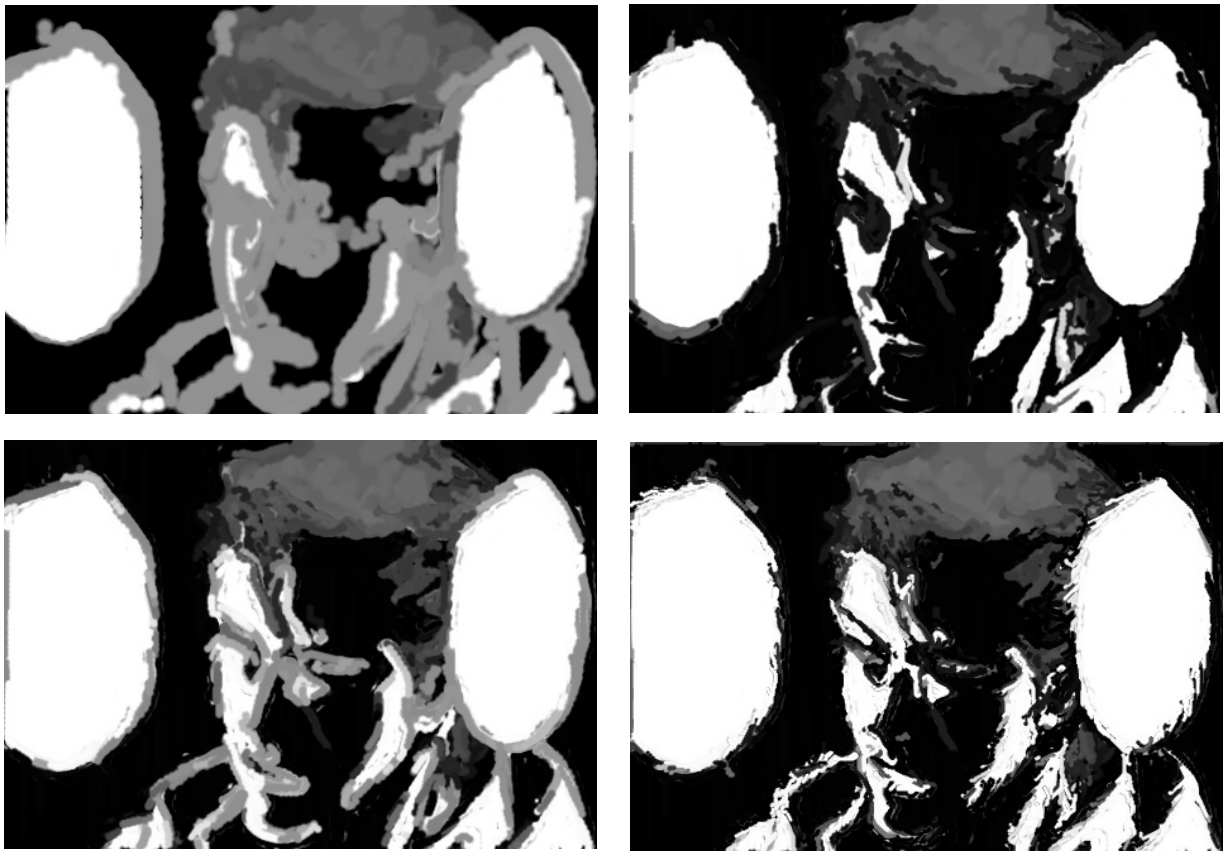
```
procedure paintLayer(canvas,referenceImage, R)
{
   S := a new set of strokes, initially empty

   // 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} / grid²
        if (areaError > T) then
        {
           // find the largest error point
           (x_1,y_1) := arg max_{i,j∈M} D_{i,j}
           s :=makeStroke(R,x_1,y_1,referenceImage)
           add s to S
        }
      }

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

畫線的部分是採用起點的 pixel 的顏色，沿著 gradient 的垂直方向畫，若是參考圖的顏色已經比起畫布上本來的顏色，和選用的顏色差距過大則結束這筆，並且在有必要的時候轉向。詳細的三個參數 fc、minStrokeLength、maxStrokeLength，分別代表線的圓滑程度、最短、最長的線長，都是可以自定的。

```
function makeSplineStroke(x_0,y_0,R,refImage)
{
   strokeColor = refImage.color(x_0,y_0)
   K = a new stroke with radius R
         and color strokeColor
   add point (x_0,y_0) to K
   (x,y) := (x_0,y_0)
   (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) :=f_c*(dx,dy)+(1-f_c)*(lastDx,lastDy)
      (dx,dy) := (dx,dy)/(dx² + dy²)^{1/2}
      (x,y) := (x+R*dx, y+R*dy)
      (lastDx,lastDy) := (dx,dy)

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

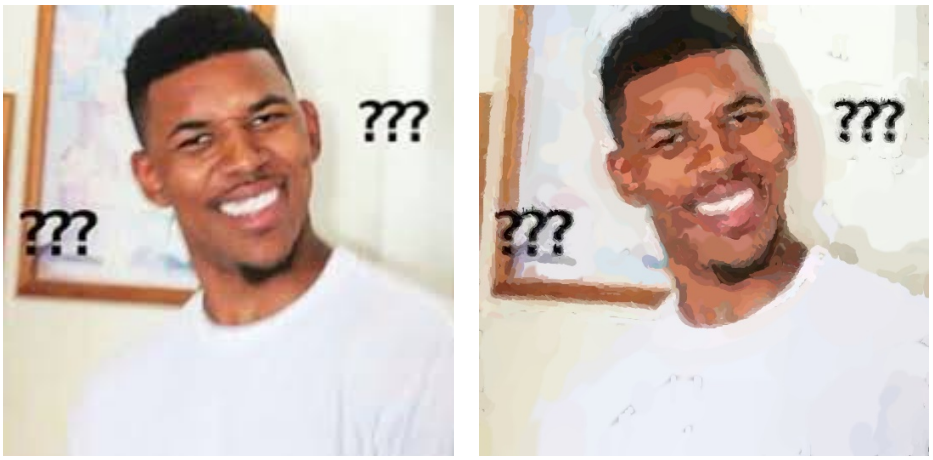
2. Result

(a)這張圖大小是 1170*810，參數的半徑是 8 4 4 2，特別用來體現圖片大小跟筆刷大小對於邊緣的影響。從左上到右下分別是每塗上一層的結果。



(b)這張圖跟(a)唯一不同的只有大小是 469*324。

(c) 參數 64 32 32 8 4 4 2 左邊是原圖，右邊是油畫效果的圖



(d) 左邊是原圖，與右邊是油畫效果（參數是 64 32 32 8 4 4 2）

(e) 參數 164 32 32 8 4 4 2 左邊是原圖，右邊是油畫效果的圖

(f) 參數 64 32 32 8 4 4 2 左邊是原圖，右邊是油畫效果的圖

3. Conclusion

首先是參數的調整，論文中有提供幾個參數，但是在我們的程式中，這些參數不知道為什麼不太管用，經過幾番測試，我們選出了幾個合適的參數 T=40，maxStrokeLength=16，minStrokeLength=1，fc=0.9。除此之外，筆刷半徑的參數經過多次的測試，我們選擇先把圖片壓成大約長邊為 1200 的大小，然後筆刷半徑使用 64 32 32 8 4 4 2，然後再調回原來的大小，這樣做出來的效果比較理想。

過程中發現幾個問題，首先是筆刷不能夠太小，半徑為 2 就是極限了，因為在移動筆刷的過程中，移動的距離是依照半徑大小來決定的（越大移動越遠），若是半徑太小會因為取整的關係容易變成垂直或是水平的線條，而導致邊緣有嚴重的鋸齒，而半徑為 2 的時候也會有這樣的情況發生，只是還勉強可以接受。隨之而來的就是圖片本身也不能夠太小，600*400 以上會是比較合適的選擇。

第二個問題是畫出來的線像是毛毛蟲一樣，不太能夠做出尖角，一開始的方法只有一直加更小筆刷，但是又會遇到前一個問題，於是我有個有趣的發想:在畫畫的時候，即使只有一定寬度的筆刷，也能夠透過刷多次不同的顏色做出尖角或是明顯的交界（第一次是物件的顏色，第二次背景的顏色），為什麼不能用同樣的半徑塗兩次或是更多呢?於是就有了像是 8 4 4 2 這樣的參數，只塗兩次的原因在於第三次之後的效果不明顯，不如用更小的筆刷，而我們也不需要在一開始就做出尖角的細節，所以選擇放在中間的寬度。

然而到最後也有一些無法解決的問題，像是做出來的圖雖然整體看起來是可以接受的，但是放大仔細看之後，仍然會有一些扭曲的線。

4. Reference

Proj_09_Painterly_Rendering_hertzmann-siggraph98

Video result

1. Approach

先將影片根據 fps 壓縮成 frame 後，套用上述的 approach1，得到所有 frame 所跑出來的結果，再根據原本的 fps 壓回 video。

2. conclusion

影片的結果我們放在同一個資料夾當中，其中 Algorithm1 的影片效果比較好，Algorithm 2 的影片會有閃爍的現象，我們認為是因為 Algorithm 2 使用不同大小的筆刷去產生圖片，因此每一個 frame 之間都會有些微的差異，這樣就會造成前後 frame 不一致，結果跑出來的影片就會有閃爍的現象，且這些閃爍的部分都分布於較細微的地方。而 Algorithm 1 的方法是一個點一個點去畫，且有半徑大小，因此前後

frame 就不會有太大的差異，這樣也比較不會造成閃爍。

3. Reference
   http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15463-s10/www/final_proj/www/pgcallah/PGCpaper.pdf