Painterly Rendering

18011688 차태관

[사용된 함수들]

double getDiff(CvScalar src, CvScalar dst)
// RGB값 차이를 계산할 함수
double getGradient(CvScalar src, CvScalar dst)
//색 차이를 반환하는 함수 (gradient 의 방향을 알기위해 제곱하지 않음.)
void makeSplineStroke(IplImage* canvas, IplImage* ref, CvPoint** S, CvScalar** C, int R[], int Ri, int GX, int GY)
//Spline을 만드는 함수
void paintStroke(IplImage* src, IplImage* dst, CvPoint** S, CvScalar** C, int R[],int Ri,int GX, int GY)
//Circle을 형성하는 함수
void paintLayer(IplImage* canvas, IplImage* ref,IplImage* src, int R[], int I)
//레이어를 칠하는 함수
void paint(IplImage* src,IplImage* dst, int R[])
//paint 함수

[2.1 코드 구현]

설명하기 전에, 모든 함수와 코드의 구현은 논문에 나와있는 의사 코드를 기반으로 동일하게 이루어지도록 했다.

먼저 원을 이용해 그림을 표현하는 2.1부터 구현하였다. 먼저 붓 개수를 5개, 붓 크기를 각각 35, 15, 7, 5. 3 으로 설정 한 후, 레퍼런스 이미지를 붓 크기로 가우시안 블러링을 한 후, paintLayer함수를 호출했다.

이미지를 처리할 때는 그리드를 활용했다. 그리드는 이미지를 붓의 크기로 나누어 주어 영역을 나누는 것으로, MP19 강의에서 교수님이 제시해주셨던 방법을 그대로 사용했다.

-2.1 PaintLayer 함수 호출 후

먼저 2차원 배열 CvPoint** S, CvScalar** C를 동적할당했다. 각각 그리드 개수 만큼을 동적할당 했으며 2차원으로 초기화되어 각 그리드의 행렬과 대응되게 했다.

이때 그리드의 크기는 붓의 크기이다. 배열 S 는 캔버스와 가장 오차가 큰 점의 좌표를 저장할 배열이고, 배열 C는 그 대응되는 좌표의 레퍼런스 이미지 컬러 값이다.

2중 for문을 만들어 전체 grid에 대하여 실행하는데, 각 그리드 내의 범위를 2중 for문으로 다시 훑었다. 이는 그리드 내에서 캔버스와 가장 오차가 큰 점을 찾기 위함인데, 가장 오차가 큰 점을 찾아서 S 배열에 저장하고, 오차가 큰 점을 저장함과 동시에 해당하는 그리드의 면적의 오차를 전부 더한 후 면적으로 나누어 주어 평균 오차를 계산했다. 만약에 평균 오차가 1200 이하라면 S배열에 저장할 좌표를 (-100, -100) 으로 설정해주어 실제로는 캔버스에 나타나지 않도록 했다. 이렇게 하는 이유는 면적의 평균 오차가 일정 비율보다 낮다면 캔버스에 원을 그린다면 오히려 그림의 품질이 떨어질 것이기 때문이다.

4중 for문이 종료되고 나면 프로그램을 실행 할 때 사용자로부터 입력받은 그리기 모드 (0 = Circle, 1 = Stroke) 에 따라 paintStroke() 혹은 makeSplineStroke()를 호출했다. 호출이 끝난 뒤에는 동적할당한 배열들의 메모리를 해제했다.

2.1, 원을 그리는 것이기 때문에 paintStorke() 가 호출된다.

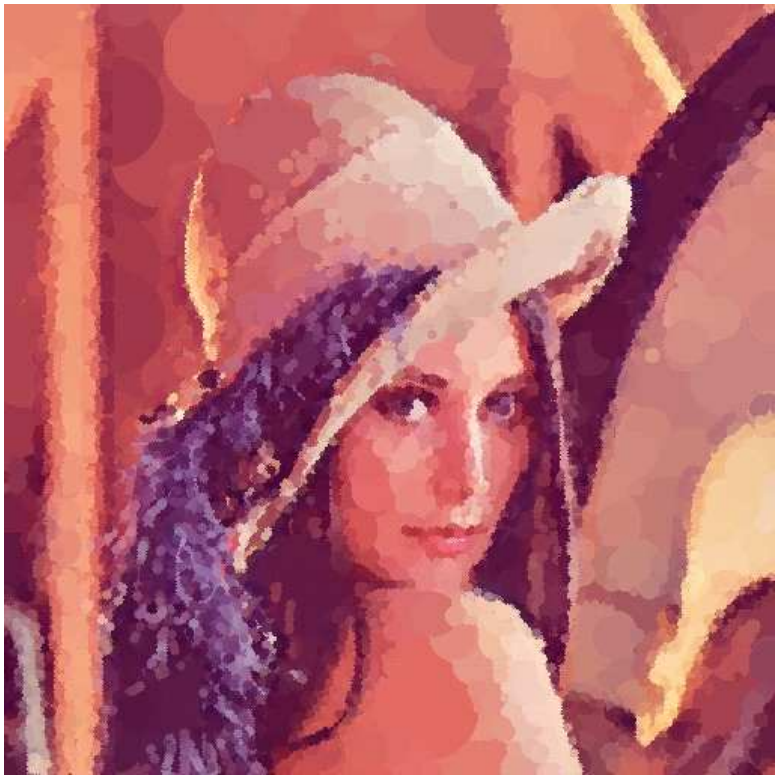
-2.1 PaintStroke 함수 호출 후

논문의 의사코드에 따르면, “Paint all strokes in S on the Canvas, in random order” 이기 때문에 paintLayer 함수에서 저장한 각 그리드에서의 최대 오차점들의 좌표에 원을 무작위 순서로 그려주어야 한다. 왜냐하면 0,0부터 순차적으로 원을 그려주게 되면, 항상 일정한 그림이 나오는 것은 물론 그림처럼 느껴지기 힘들 것이다.(화가들이 왼쪽 위부터 색을 채우며 그림을 그리진 않는다. 이하 생략)

따라서 무작위 순서로 원을 그려주기 위해 V 라는 2차원 정수배열을 동적할당 했다. 배열의 크기는 그리드의 크기와 같으며 배열은 0으로 초기화된다. 이 배열에서 0은 방문하지 않은 그리드를, 1은 방문한 그리드를 나타낸다.

랜덤 함수를 이용해 그리드 중 하나를 선택하고, V배열의 값이 1이면 다른 그리드를 계속해서 선택해 1이 아닐 때까지 그리드를 선택한다. 이후 for문을 실행하는데, for문의 범위는 0~gridX*girdY 까지 로 실행한다(그리드의 개수만큼 시행). 이렇게 무작위의 순서로 원을 그리면, 그림이 완성된다.

아래는 완성된 이미지의 예시이다.



원의 크기가 다양하게 나타나며, 색깔이 일정한 위치는 큰 원으로 나타나고, 좀 더 디테일한 부분들은 작은 원으로 나타나는 것을 볼 수 있다. 이것은 이전에 평균 오차가 일정 수준 아랫니면 캔버스에 원을 그릴 필요가 없다는 설명에 기반한 결과이다.

[2.2 코드 구현]

PaintLayer 함수 호출 까지는 2.1 과 동일하다. CvPoint** S, CvScalar** C 배열을 선언하고 배열에 정보들을 저장한 후, paintStroke 함수 대신 makeSplineStroke 함수를 호출한다.

2.2 에서는 원 대신에 스플라인을 사용한다. 스플라인이란 점들을 직선으로 연결한 선으로 화가의 붓을 본딴 것이 될 것이다. 여기서는 gradient를 이용해 그림이 밝아지는 방향 벡터의 법선 벡터를 따라 좌표가 이동할 것이다.

-2.2 makeSplineStroke 함수 호출 후

2.1 과 동일하게 V 정수 2차원 배열을 동적할당 했다. 여기서는 2.1 과 다르게 CvPoint * SP 배열을 사용했는데, 스플라인을 만들 좌표들을 순서대로 저장할 CvPoint 배열이다.

for문은 마찬가지로 그리드의 크기만큼 (gridX * gridY) 실행해준다. 이번에는 여러 개수의 스플라인들을 그릴 것인데, 무작위로 정한 처음 좌표는 그리드별로 계산했던 최대 오차를 가지고 있는 점으로부터 시작한다. 그렇게 시작한 처음 좌표는 무조건 SP 배열에 들어가게 되고, 다음 점의 좌표부터는 gradient를 계산하여 결정한다.

본격적인 gradient 벡터를 구하기 전에, 각 스플라인의 좌표를 구할 for문을 실행할 때 y,x 좌표가 캔버스의 범위를 벗어나면 for문을 종료하고, 내가 설정해 놓은 최소 스트로크 횟수를 초과하고, 캔버스와 레퍼런스 이미지의 색 차이가 붓 색깔과 레퍼런스 이미지의 차이보다 작다면 for문을 종료했다. (코드 참조 요망)

gradient의 벡터는 손쉽게 구할 수 있다. 기준 좌표의 오른쪽 좌표로부터 기준 좌표를 빼주면 그것이 gx(gradient x 축 변화량), 기준 좌표의 아래 좌표로부터 기준 좌표를 빼 주면 gy(gradient y 축 변화량) 가 된다.

논문에 서술된 대로 그림의 밝기가 밝아지는 방향과는 별개로 gradient를 그릴때는 gradient 방향의 수직 벡터 방향으로 색이 그려진다. 따라서 gx gy의 법선 벡터를 구하는데, 이때 법선 벡터는 매우 쉽게 구할 수 있다. dx = -gy, dy = gx 로 대입하면 법선 벡터가 된다.

법선 벡터(dx,dy)를 구한 후에는 이전의 좌표벡터와 현재의 법선벡터를 내적해 0 이하일 때는 벡터의 방향을 반대로 바꾸어 주었다. 이는 스플라인이 급한 커브를 하는 것을 막기 위한 것으로 실제 화가가 그리는 것처럼 스플라인을 형성하기 위함이다 (실제 화가는 윤곽을 따라 그림을 그리지 붓을 급커브 시키진 않을 것이다).

처음 좌표를 정한 후 붓의 반지름(R)만큼 좌표를 이동시킬 때는 삼각함수를 이용했다. MP18 강의에서 익힌 각도를 통한 좌표 구하기(atan2 이용) 을 사용했고 R 값 또한 알고 있었기 때문에 손쉽게 구하기가 가능했다. 스플라인의 첫 좌표일 때는 첫 좌표로부터 이동했고, 스플라인의 두 번째 이상 좌표일 때는 이전의 좌표를 저장해 놓은 다음에 다음 control point 로 이동했다.

하나의 스플라인을 정하는 과정이 끝나면 cvLine 함수를 이용해 SP 배열에 저장해놓았던 좌표들을 전부 이어 주었다. 이때 라인의 두께는 붓의 크기가 된다. 이를 실행하면 아래와 같다.

그림이 유화처럼 성공적으로 다시 그려진 것을 볼 수 있다.

[시행착오]

과제를 수행하면서 여러 시행착오에 봉착했었다. 처음에는 2차원 배열을 사용하지 않고 데이터를 저장하여 오히려 연산이 복잡해지는 경우를 만나거나, 배열의 인덱스를 초과하여 데이터를 조회하는 초보적인 실수 등을 했다. 이는 디버깅을 하며 예외처리를 철저하게 해 주어 해결했다. 알고리즘을 이해하고 프로그래밍을 했는데 실행이 정상적으로 되지 않아 좌절했는데, 간단한 예외 처리에서 실수를 해서 많은 시간을 허비한 것이 허탈했다.

[총평]

논문의 내용과 의사 코드를 이용해 논문을 실제 코드로 변환하는 연습을 해 볼 수 있었다. 4-1 논문 요약 과제를 좀 더 잘 완수하지 못한 것이 아쉬움이 되었다. 직접 구현이 어려울 것 같았던 첫인상과는 달리 의사 코드도 명확하고, 원리도 간단해 생각했던 시간보다는 적은 시간 안에 과제를 해결할 수 있었다.

항상 예외처리를 철저하게 해주어야 한다는 교훈 또한 얻을 수 있었고, 논문을 통해 실제 결과로 나타내어 성취감도 얻었으며, 체계적인 프로그래밍에 대한 경험 또한 한 단계 성장했다고 느꼈다.