Technical_report

2376029 김문경

introduction

이 프로젝트에서는 이미지 필터링을 C++와 OpenCV를 이용하여 구현했다. RGB 컬러 이미지와 grayscale 이미지에 대하여 Mean, gaussian, sobel, laplacian, 그리고 unsharp mask filtering 기법을 사용했다. 또한 tickmeter를 사용해 separable 방법으로 필터링을 적용했을 때와, 일반적인 방법으로 필터링을 적용했을 때의 소요 시간을 비교했다.

코드 구현

▼ 필터별 알고리즘 요약

필터	설명
Mean	- 이미지 intensity의 평균값 사용 - 경계 처리 포함
Gaussian	- 2D 및 Separable 방식으로 구현 - 경계 처리 포함
Sobel	- 1차 미분 기반 엣지 검출 - 주어진 X, Y 방향 커널 사용
Laplacian	- 2차 미분 기반 엣지 검출
Unsharp Masking	- (원본 - k*blur 이미지) 방식으로 선명도 향상

▼ 경계 처리 옵션

- zero-paddle
  - 이미지 밖은 0으로 간주

```
if (!strcmp(opt, "zero-paddle")) {
            float sum1 = 0.0;
            for (int a = -n; a <= n; a++) {
               for (int b = -n; b <= n; b++) {

                  if ((i + a <= row - 1) && (i + a >= 0) && (j + b <= col - 1) && (j + b >= 0)) {
                     sum1 += kernelvalue*(float)(input.at<G>(i + a, j + b));
                  }
               }
            }
            output.at<G>(i, j) = (G)sum1;
         }
```

  이미지 row, col 밖의 픽셀 (i, j)에 대해서는 아예 계산하지 않는다

- mirroring
  - boundary를 이미지 내부로 반사

```
else if (!strcmp(opt, "mirroring")) {
            float sum1 = 0.0;
            for (int a = -n; a <= n; a++) {
               for (int b = -n; b <= n; b++) {

                  if (i + a > row - 1) {
                     tempa = i - a;
                  }
                  else if (i + a < 0) {
                     tempa = -(i + a);
                  }
```

```
                else {
                    tempa = i + a;
                }
                if (j + b > col - 1) {
                    tempb = j - b;
                }
                else if (j + b < 0) {
                    tempb = -(j + b);
                }
                else {
                    tempb = j + b;
                }
                sum1 += kernelvalue*(float)(input.at<G>(tempa, tempb));
            }
        }
        output.at<G>(i, j) = (G)sum1;
    }
```

if 문 설명

<X 좌표 / 세로 방향 처리>

상황	조건	적용 방식
커널이 아래 경계를 벗어남	i + a > row	tempa = i -a
커널이 위쪽 경계를 벗어남	i + a < 0	tempa = -(i + a)
이미지 안쪽일 때	0 ≤ i + a ≤ row	그대로

<Y 좌표 / 가로 방향 처리>

상황	조건	적용 방식
커널이 오른쪽 경계를 벗어남	j + b > col	tempb = j - b
커널이 왼쪽 경계를 벗어남	j + b < 0	tempb = -(j + b)
이미지 안쪽일 때	0 ≤ j + b ≤ col	그대로

즉, 이미지 밖으로 나가려 하면 경계를 기준으로 안쪽으로 대칭되도록 인덱스를 조절

- adjustkernel
  - 유효 커널 영역만 정규화

```
else if (!strcmp(opt, "adjustkernel")) {
        float sum1 = 0.0;
        float sum2 = 0.0;
        for (int a = -n; a <= n; a++) { // for each kernel window
            for (int b = -n; b <= n; b++) {
                if ((i + a <= row - 1) && (i + a >= 0) && (j + b <= col - 1) && (j + b >= 0)) {
                    sum1 += kernelvalue*(float)(input.at<G>(i + a, j + b));
                    sum2 += kernelvalue;
                }
            }
        }
        output.at<G>(i, j) = (G)(sum1/sum2);
    }
```

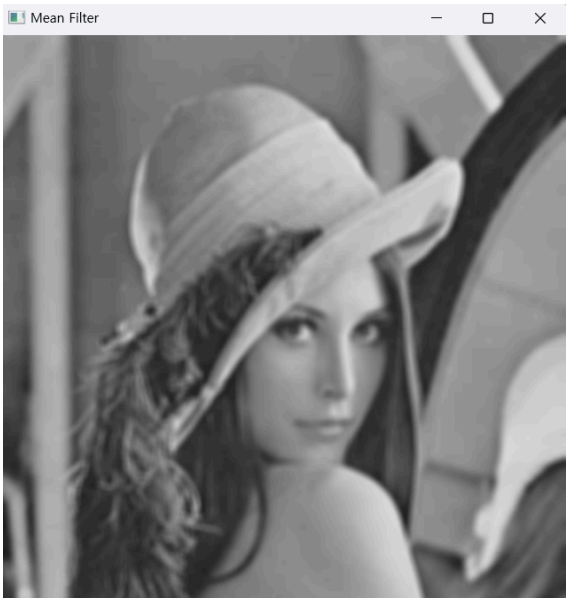
  - 이미지 row, col 밖의 픽셀 (i, j)에 대해서는 아예 계산하지 않는다
  - sum1 ⇒ 이미지 안의 픽셀만 convolution한 것들의 합
  - sum2 ⇒ 이미지 안의 kernel value들의 합

상황	동작
이미지 안 쪽	전체 커널 사용
경계에 걸릴 때	바깥쪽 커널 무시

커널 값을 다시 정규화 함으로써 경계에서도 자연스러운 필터링 값을 얻을 수 있다

Result

▼ 필터 전 후 이미지 결과

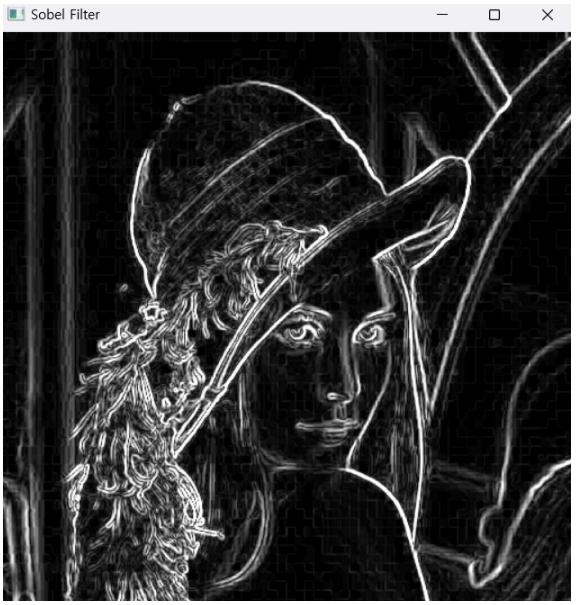


▲ low-pass filter라 blurry함

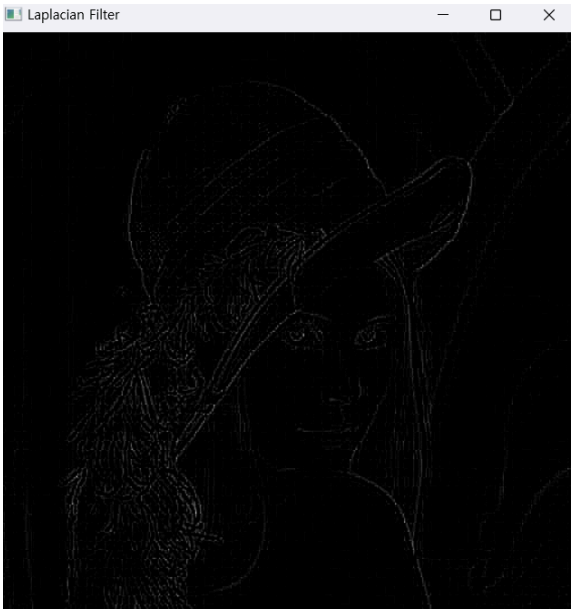
▲ low-pass filter라 blurry함

하지만 mean filter보다 더 자연스러움

왜냐면 커널 안에서 중심 픽셀에 가까운 애들한테 더 큰 가중치를 주고 평균을 내는 방식이기 때문이다.



▲ high-pass filter by 1차 미분 기반 필터 이기 때문에 엣지가 잘 검출됐음을 볼 수 있다



▲ high-pass filter by 2차 미분 기반 필터 이기 때문에 엣지가 잘 검출됐음을 볼 수 있다
근데 양수/음수 값이 0 주변에 몰려 있어서 출력 값이 좀 어둡다

▲ k = 0.5일 때

$$Output = (I - kL)/(1 - k)$$

I = input image, k = parameter, L = 가우시안 필터 적용한 이미지

▲ k = 0.7일 때

$$Output = (I - kL)/(1 - k)$$

I = input image, k = parameter, L = 가우시안 필터 적용한 이미지

k가 0.5일 때와 비교해 봤을 때, k 값이 더 클수록 블러된 이미지(L)의 영향을 더 많이 빼기 때문에 고주파 성분이 더욱 강조되었음을 알 수 있음

분석

▼ **성능 비교 (TickMeter 사용)**

필터 종류	이미지 크기	실행 시간 (ms)
Gaussian Gray	512 * 512	103.54
Gaussian RGB	512 * 512	152.011
Gaussian Gray Separable	512 * 512	80.1561
Gaussian RGB Separable	512 * 512	110.334
Unsharp masking Gray	512 * 512	145.895
Unsharp masking RGB	512 * 512	149.948

1. **Gaussian vs Separable Gaussian**
   - 동일한 필터여도, separable 방식이 평균적으로 약 20 ~ 25% 정도 빠른 성능을 보여줬다
   - 이는 2D convolution 대신, 1D 커널을 두 번 적용함으로써 연산량이 O(n^2) → O(2n)으로 줄어들었기 때문이다

2. **Gray vs RGB**
   - RGB 이미지의 경우, 세 채널에 대한 연산이 필요하기 때문에 Grayscale보다 약 1.4~ 1.5배 가량 더 많은 시간이 소요 된다

3. **Unsharp Masking**
   - Gaussian filtering을 선행한 후, 추가적인 연산이 필요하기 때문에 단독으로 Gaussian 필터를 적용했을 때보다 더 오랜 시간이 걸린다
   - 특히, 고주파 강조를 위한 연산이 포함되므로 총 실행 시간은 약 40%정도 더 오래 걸린다.

결론

1. Separable Gaussian 필터는 동일한 결과를 내면서도 연산 효율성이 높아, 속도가 중요한 상황에서 더 적합한 방법임을 확인하였다.
2. RGB 및 Unsharp Masking은 연산량 증가에 따라 시간이 더 소요되나, 시각적으로 더 선명하고 개선된 결과를 제공하는 장점이 있다.

본 프로젝트를 통해 필터를 직접 구현하며 이미지 처리 알고리의 원리를 자세히 알 수 있게 되었다. 특히, 경계 처리 및 성능 개선을 위한 방법도 알게 되었다.