OPEN CV 없이 구현한 C언어 영상처리

[Intel] 엣지 AI SW 아카데미
절차지향 프로그래밍

허찬욱

프로젝트 개요

프로젝트 목표

이미지 파일을 C언어로 구현한
알고리즘으로 보정하기

프로그램명

GrayScale Image Processing
(Version 1.0)

프로젝트 기간

2024.03.05 ~ 2024.03.19

주요 기능

화소점 처리, 기하학 처리, 화소영역 처리,
히스토그램 처리

개발환경

Windows 10 Pro 64bit
Visual Studio 2022

전체 소스코드 링크

https://blog.naver.com/hew0916/223388456736

특이사항

콘솔창 및 텍스트 입력 기반 실행창
Raw 형식의 파일만 저장 가능
정방형 & GrayScale Image 사용

구조도

D:/Mypic

mypet.RAW
(512X512)

loadImage()

메모리(RAM)

inW

inH

inImage 배열

영상처리 알고리즘

outH

outImage 배열

outW

saveImage()

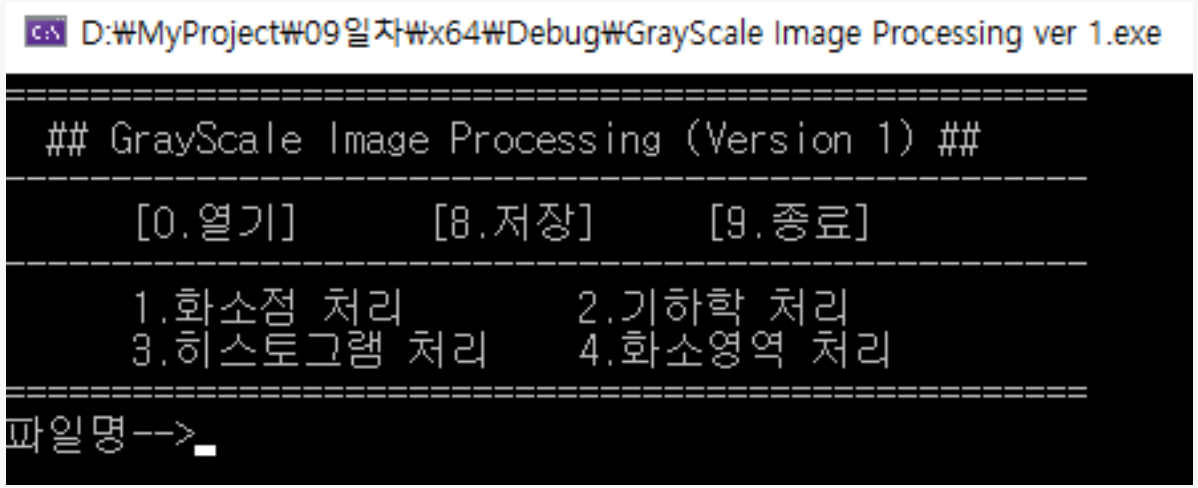
out1.raw
(256x256)

out2.raw
(512x512)

printImage()

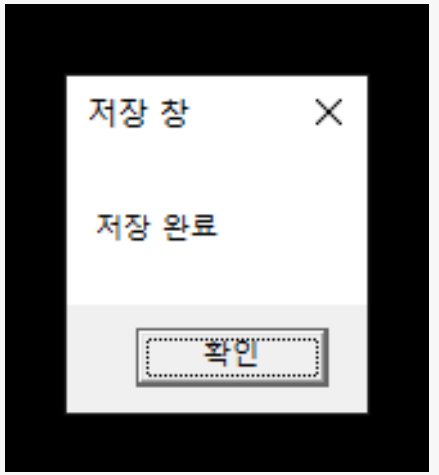
Display

메인화면 구성



[0.열기]를 누른 후 파일명을 입력하면
해당 파일이 Print 됨

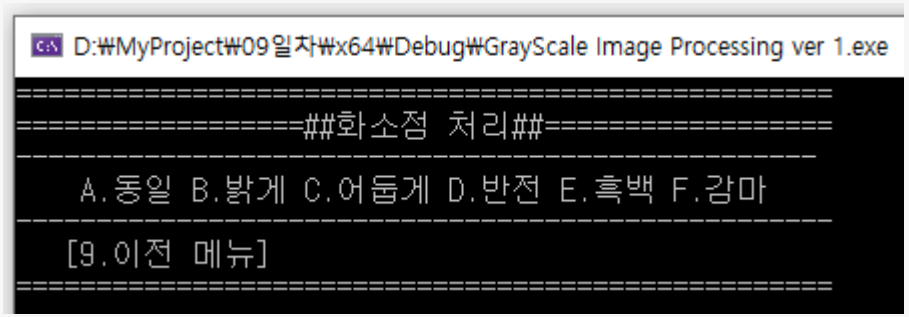
저장완료 알림 창



영상처리 알고리즘을 활용해 이미지 편집 후
메인화면으로 돌아와 [8.저장]를 눌러
raw 파일로 저장 가능

1. 화소점 처리

- 원 화소의 값이나 위치를 바탕으로 단일 화소값을 변경하는 기술
- 종류 : A. 동일 B. 밝게 C.어둡게 D. 반전 E. 흑백 F. 감마



원본 이미지

A. 동일

equalImage()

출력 이미지

```
outImage[i][k] = inImage[i][k];
```

1. 화소점 처리

원본 이미지

B. 밝게

addImage()

outImage[i][k] = inImage[i][k] + val;

입력값에 따라 밝아지고 어두워지게 구현
(Val=80 입력)

C. 어둡게

subImage()

outImage[i][k] = inImage[i][k] - val;

D. 반전

reverseImage()

outImage[i][k] = 255 - inImage[i][k];

각 화소의 값이 영상 내에 대칭이 되는 값으로 변환

1. 화소점 처리

원본 이미지

E. 흑백

bwImage()

```
if (inImage[i][k] > 127)
                outImage[i][k] = 255;
else
                outImage[i][k] = 0;;
```

F. 감마

gammaImage()

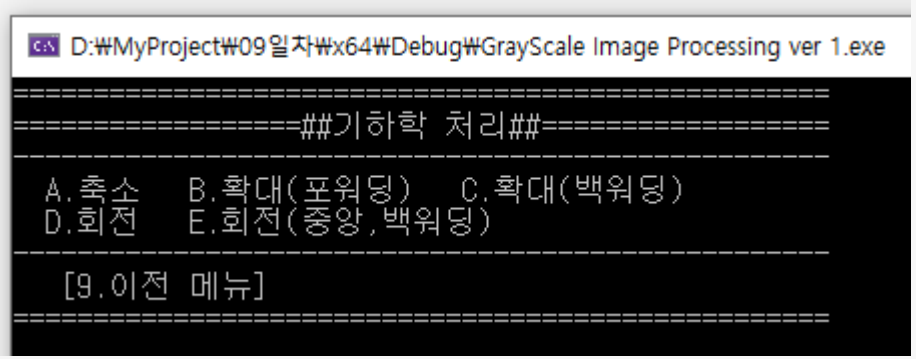
gamma = 0.2 대입

gamma = 1.8 대입

```
outImage[i][k] = 255.0 * pow(inImage[i][k] / 255.0, gamma);
```

2. 기하학 처리

- 영상을 구성하는 화소의 공간적 위치나 배열을 재배치 하는 기술
- 종류 :
  A. 축소 B. 확대(포워딩) C. 확대(백워딩)
  D. 회전 E. 회전 (중앙, 백워딩)



원본 이미지(512x512)

A. 축소

zoomOut()

Scale = 2 입력

출력 이미지(256x256)

```
outImage[(int)(i / scale)][(int)(k / scale)] = inImage[i][k];
```

2. 기하학 처리

원본 이미지(512x512)

Scale=2 입력

B. 확대(포워딩)

zoomIn()

```
outImage[(int)(i * scale)][(int)(k * scale)] = inImage[i][k];
```

홀 문제가 발생한
포워딩 출력 이미지(1024x1024)

C. 확대(백워딩)

zoomIn2()

보간 처리한
백워딩 출력 이미지(1024x1024)

```
outImage[i][k] = inImage[(int)(i / scale)][(int)(k / scale)];
```

2. 기하학 처리

D. 회전
rotate()

degree=45 입력

```
int degree = getIntValue();
double radian = degree * 3.141592 / 180.0;

int xd = (int)(cos(radian) * xs - sin(radian) * ys);
int yd = (int)(sin(radian) * xs + cos(radian) * ys);

if ((0 <= xd && xd < outH) && (0 <= yd && yd < outW))
outImage[xd][yd] = inImage[xs][ys];
```

E. 회전(중앙, 백워딩)
rotate2()

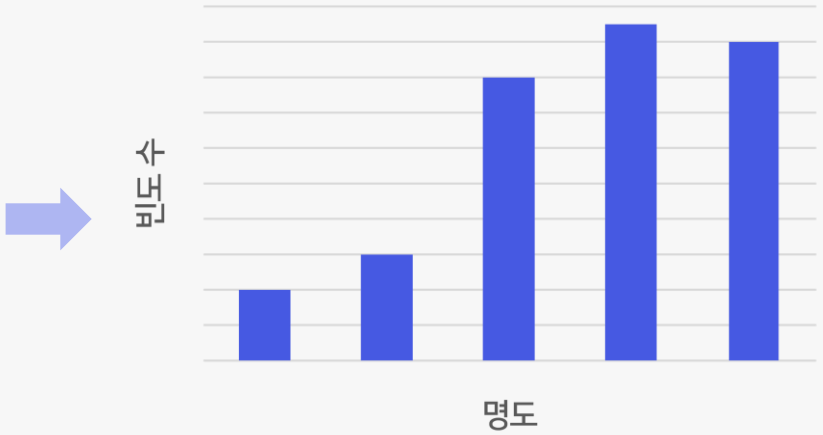
degree=30 입력

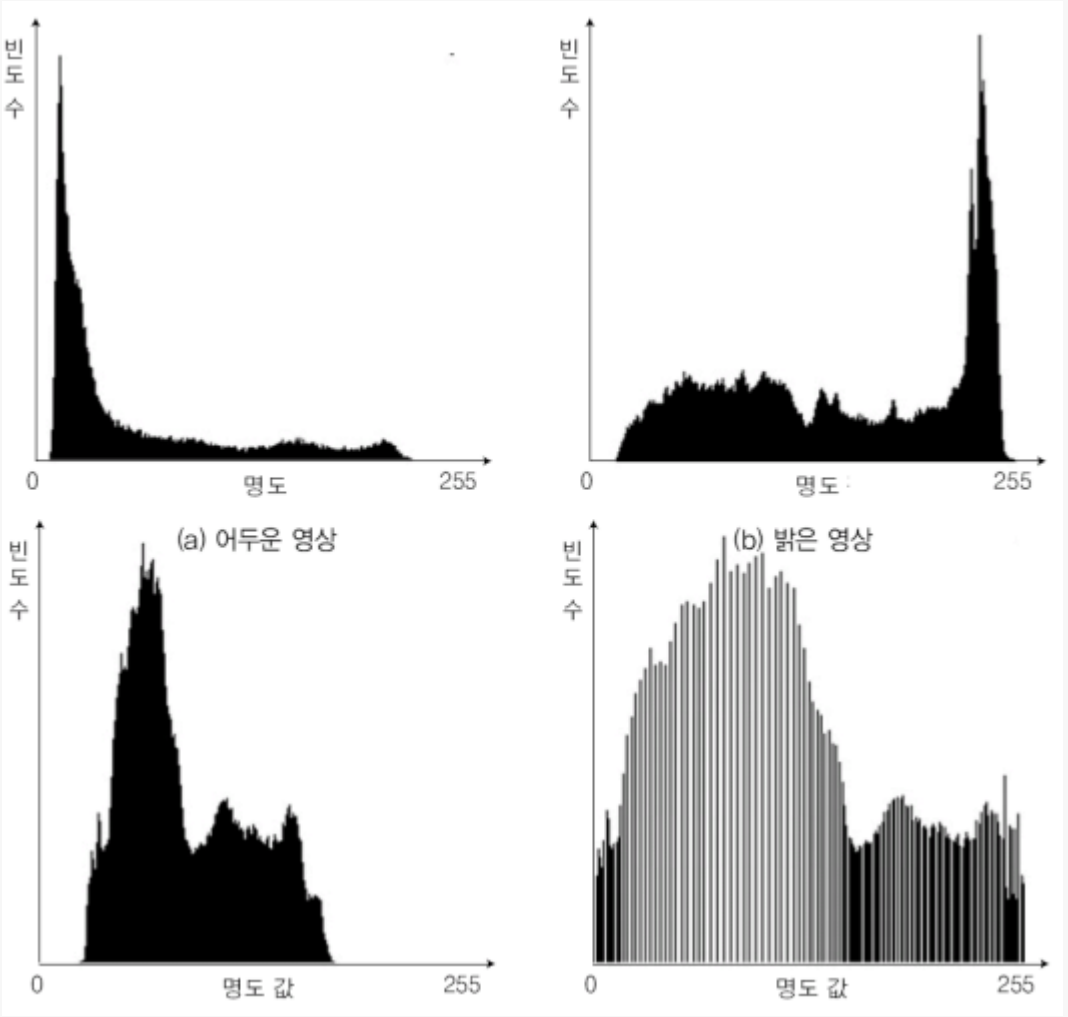
```
int degree = getIntValue(); double radian = -degree * 3.141592 / 180.0;

int cx = inH / 2; int cy = inW / 2;

int xs = (int)(cos(radian) * (xd - cx) + sin(radian) * (yd - cy));
int ys = (int)(-sin(radian) * (xd - cx) + cos(radian) * (yd - cy));
xs += cx; ys += cy;

if ((0 <= xs && xs < outH) && (0 <= ys && ys < outW))
outImage[xd][yd] = inImage[xs][ys];
```

3. 히스토그램이란?

- 표로 되어 있는 분포데이터를 막대 그래프로 나타낸 것
- 종류 :
  A. 스트래칭 B. 엔드-인 C. 평활화

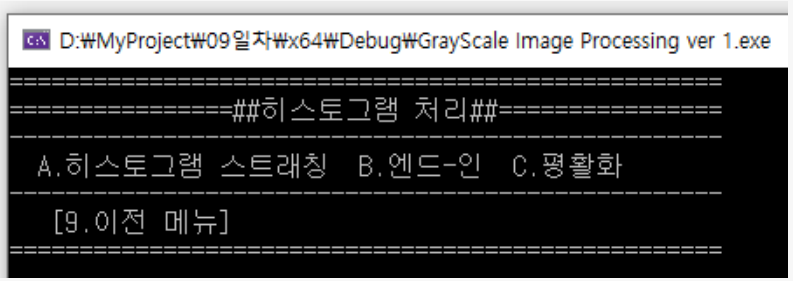
6	6	6	7
4	5	5	3
2	1	1	3
0	0	1	3



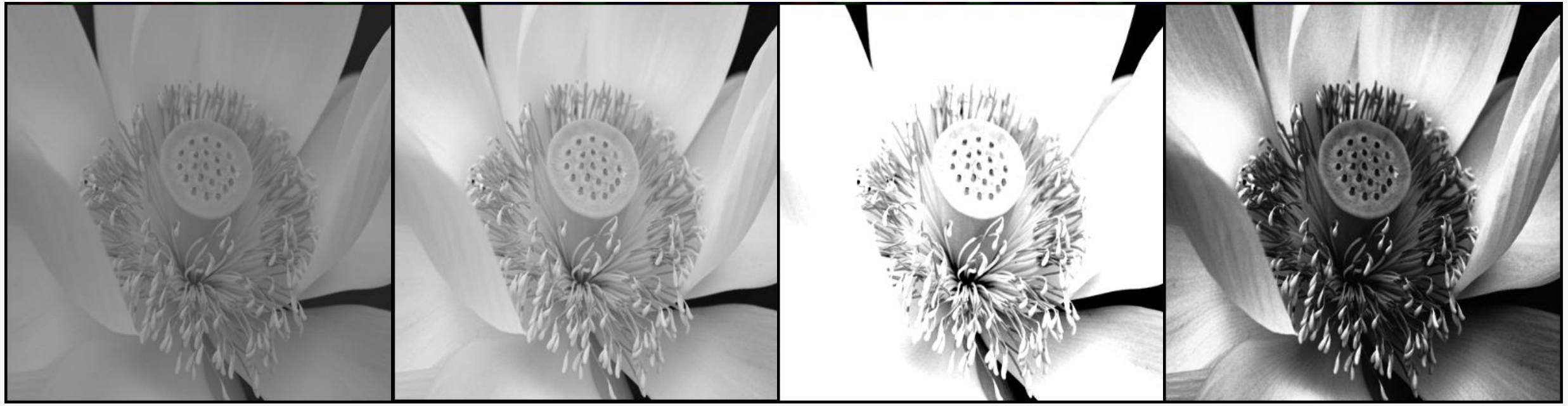


(C)명암 대비가 낮은 영상 (d)명암 대비가 높은 영상

3. 히스토그램 처리



원본 A. 스트래칭 B. 엔드-인 C. 평활화

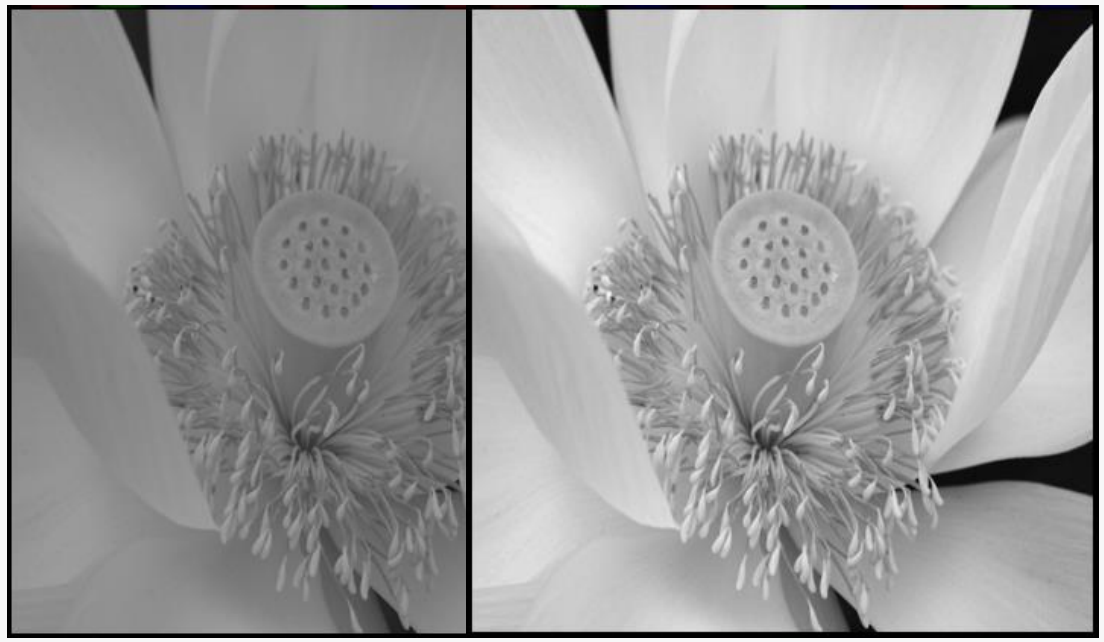


3. 히스토그램 처리

A. 명암 대비 스트래칭
- 히스토그램을 모든 영역으로 확장시켜 영상의 모든 범위의 화소값을 포함

$$outImage = \frac{inImage - low}{high - low} \times 255$$

원본 A. 스트래칭

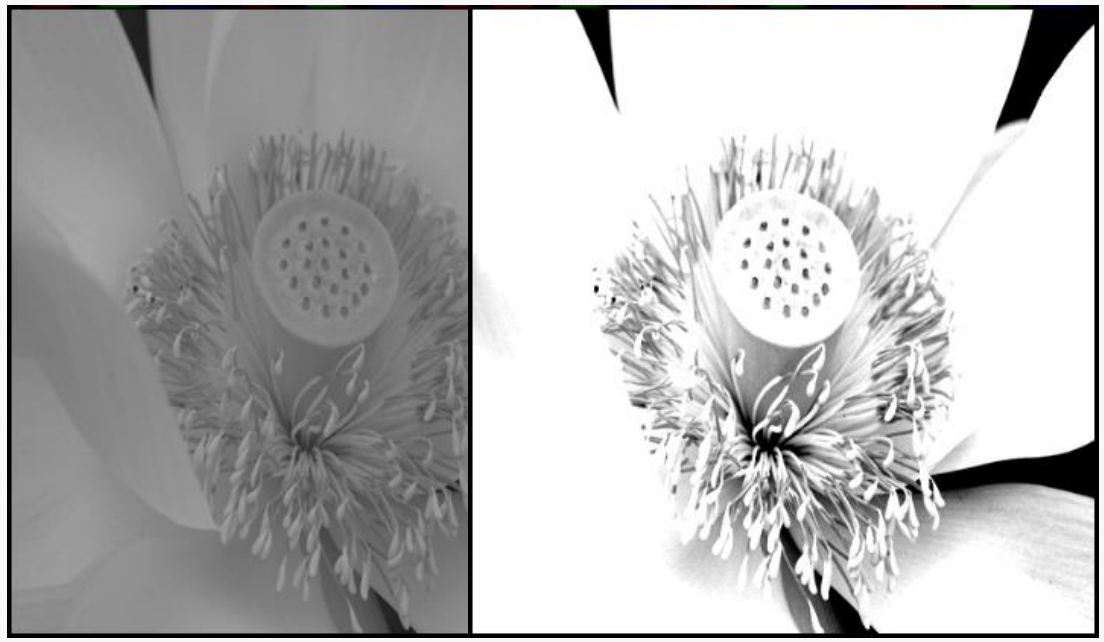


```
int high = inImage[0][0], low = inImage[0][0];
for (int i = 0; i < inH; i++) {
        for (int k = 0; k < inW; k++) {
                if (inImage[i][k] < low)
                  low = inImage[i][k];
                if (inImage[i][k] > high)
                  high = inImage[i][k];
        }
}
int old, new;
for (int i = 0; i < inH; i++) {
        for (int k = 0; k < inW; k++) {
                old = inImage[i][k];
                new = (int)((double)(old - low) / (double)(high - low) * 255.0);
                if (new > 255)
                  new = 255;
                if (new < 0)
                  new = 0;
                outImage[i][k] = new;
        }
}
```

3. 히스토그램 처리

B. 엔드-인 탐색
- 명암 대비 스트레칭에서 일정한 양의 화소를 흑백으로 지정하여 히스토그램의 분포를 더 균일하게 만드는 방법

원본 B. 엔드-인



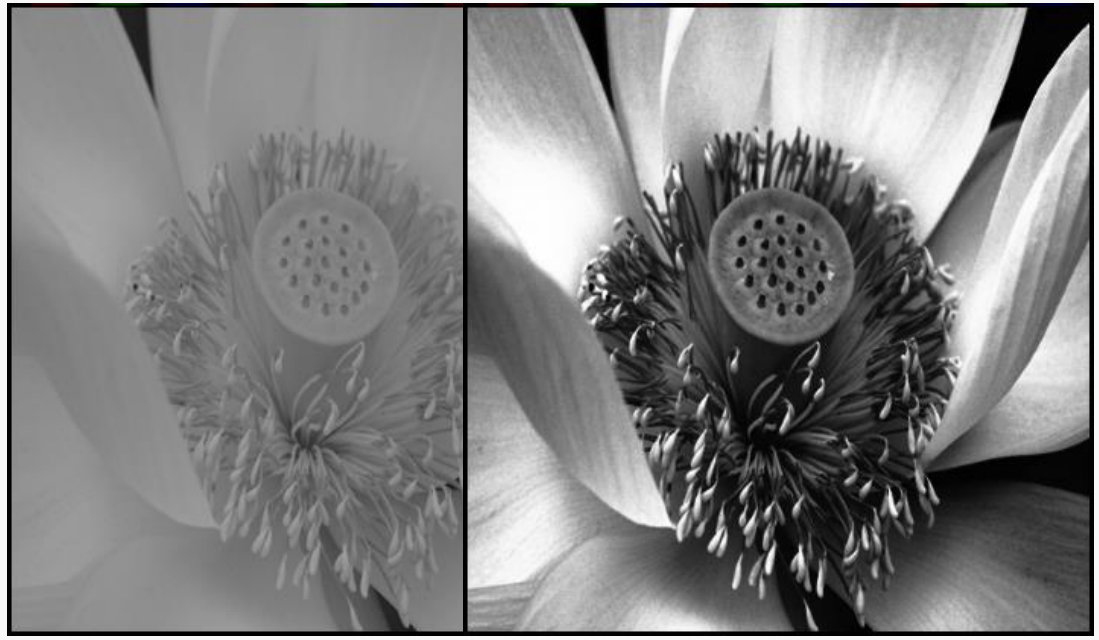
$$outImage = \begin{cases} 0 & inImage \le low \\ \dfrac{inImage - low}{high - low} \times 255 & low \le inImage \le high \\ 255 & high \le inImage \end{cases}$$

```
int high = inImage[0][0], low = inImage[0][0];
for (int i = 0; i < inH; i++) {
        for (int k = 0; k < inW; k++) {
            if (inImage[i][k] < low)
              low = inImage[i][k];
            if (inImage[i][k] > high)
              high = inImage[i][k];
        }
}

high -= 50;  low += 50;                    //스트래칭에서 이부분만 추가

int old, new;
for (int i = 0; i < inH; i++) {
        for (int k = 0; k < inW; k++) {
            old = inImage[i][k];
            new = (int)((double)(old - low) / (double)(high - low) * 255.0);
            if (new > 255)
              new = 255;
            if (new < 0)
              new = 0;
            outImage[i][k] = new;
        }
}
```

3. 히스토그램 처리

C. 평활화
- 어둡게 촬영된 영상의 히스토그램을 조절하여 명암 분포가 빈약한 영상을 균일하게 만드는 방법

원본 C. 평활화



$$sum[i] = \sum_{j=0}^{i} hist[j]$$

$$n[i] = sum[i] \times \frac{i}{N(\text{총화소수})} \times I_{\max}(\text{최대 명도값})$$

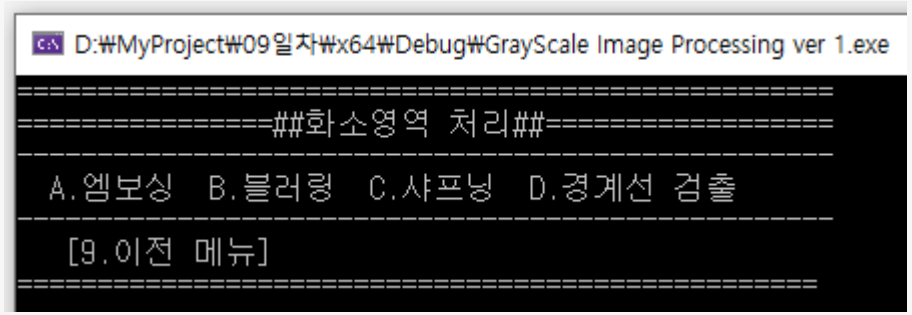
```
// 1단계 : 빈도수 세기(=히스토그램) histo[256]
int histo[256] = { 0, };
                histo[inImage[i][k]]++;

// 2단계 : 누적히스토그램 생성
int sumHisto[256] = { 0, };
sumHisto[0] = histo[0];

for (int i = 1; i < 256; i++)
                sumHisto[i] = sumHisto[i - 1] + histo[i];

// 3단계 : 정규화된 히스토그램 생성  normalHisto = sumHisto * (1.0 / (inH*inW) ) * 255.0;
double normalHisto[256] = { 1.0, };
for (int i = 0; i < 256; i++)
                normalHisto[i] = sumHisto[i] * (1.0 / (inH * inW)) * 255.0;


// 4단계 : inImage를 정규화된 값으로 치환
outImage[i][k] = (unsigned char)normalHisto[inImage[i][k]];
```

4. 화소영역 처리란?

- 화소점 처리(입력 화소) + 주변 화소 값도 고려하는 공간 영역 연산
- 회선 처리 또는 컨벌루션 처리(Convolution Processing)라고 함

$$Output\_pixel[x,y] = \sum_{m=(x-k)}^{x+k} \sum_{n=(y-k)}^{y+k} (I[m,n] \times M[m,n])$$

- **Output_pixel[x, y]: 회선 처리로 출력한 화소**
- **I[m, n]: 입력 영상의 화소**
- **M[m, n]: 입력 영상의 화소에 대응하는 가중치**



- 원시 화소와 이웃한 각 화소에 가중치를 곱한 합을 출력 화소로 생성
- 종류:
  A. 엠보싱 B. 블러링 C. 샤프닝 D. 경계선 검출

4. 화소영역 처리

A. 엠보싱 (Embossing) 효과

원본 A. 엠보싱

```
double mask[3][3] = { // 엠보싱 마스크
{-1.0, 0.0, 0.0},
{ 0.0, 0.0, 0.0},
{ 0.0, 0.0, 1.0} };

// 임시 메모리 할당(실수형)
double** tmpInImage = mallocDoubleMemory(inH + 2, inW + 2);
double** tmpOutImage = mallocDoubleMemory(outH, outW);

tmpInImage[i][k] = 127; // 임시 입력 메모리를 초기화(127) : 필요시 평균값

tmpInImage[i + 1][k + 1] = inImage[i][k]; // 입력 이미지 --> 임시 입력 이미지

// *** 회선 연산 ***
double S += tmpInImage[i + m][k + n] * mask[m][n];
tmpOutImage[i][k] = S;
tmpOutImage[i][k] += 127.0; //후처리

// 임시 출력 영상--> 출력 영상.
if (tmpOutImage[i][k] < 0.0)
outImage[i][k] = 0;
else if (tmpOutImage[i][k] > 255.0)
outImage[i][k] = 255;
else
outImage[i][k] = (unsigned char)tmpOutImage[i][k];
```

4. 화소영역 처리

B. 블러링 (blurring) 효과

원본 B. 블러링


```
double mask[3][3] = {      // 블러링 마스크
{1. / 9, 1. / 9, 1. / 9},
{1. / 9, 1. / 9, 1. / 9},
{1. / 9, 1. / 9, 1. / 9} };      //마스크만 변경

// 임시 메모리 할당(실수형)
double** tmpInImage = mallocDoubleMemory(inH + 2, inW + 2);
double** tmpOutImage = mallocDoubleMemory(outH, outW);

tmpInImage[i][k] = 127; // 임시 입력 메모리를 초기화(127) : 필요시 평균값

tmpInImage[i + 1][k + 1] = inImage[i][k]; // 입력 이미지 --> 임시 입력 이미지

// *** 회선 연산 ***
double S += tmpInImage[i + m][k + n] * mask[m][n];
tmpOutImage[i][k] = S;
~~tmpOutImage[i][k] += 127.0; //후처리~~

// 임시 출력 영상--> 출력 영상.
if (tmpOutImage[i][k] < 0.0)
outImage[i][k] = 0;
else if (tmpOutImage[i][k] > 255.0)
outImage[i][k] = 255;
else
outImage[i][k] = (unsigned char)tmpOutImage[i][k];
```

4. 화소영역 처리

C. 샤프닝 (Sharpening) 효과

원본 C. 샤프닝

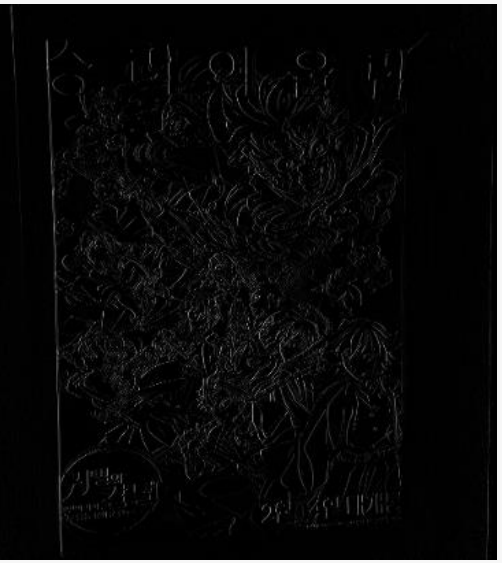

```
double mask[3][3] = {     // 샤프닝 마스크
{-1.0 , -1.0 , -1.0 },
{-1.0 ,  9.0 , -1.0 },
{-1.0 , -1.0 , -1.0 }   };     //마스크만 변경

// 임시 메모리 할당(실수형)
double** tmpInImage = mallocDoubleMemory(inH + 2, inW + 2);
double** tmpOutImage = mallocDoubleMemory(outH, outW);

tmpInImage[i][k] = 127; // 임시 입력 메모리를 초기화(127) : 필요시 평균값

tmpInImage[i + 1][k + 1] = inImage[i][k]; // 입력 이미지 --> 임시 입력 이미지

// *** 회선 연산 ***
double S += tmpInImage[i + m][k + n] * mask[m][n];
tmpOutImage[i][k] = S;
tmpOutImage[i][k] += 127.0; //후처리

// 임시 출력 영상--> 출력 영상.
if (tmpOutImage[i][k] < 0.0)
outImage[i][k] = 0;
else if (tmpOutImage[i][k] > 255.0)
outImage[i][k] = 255;
else
outImage[i][k] = (unsigned char)tmpOutImage[i][k];
```

4. 화소영역 처리

D. 경계선 검출

원본

D. 경계선 검출



```
double mask[3][3] = {    // 경계선 검출 마스크
{0.0, 0.0, 0.0},
{-1.0, 1.0, 0.0},
{0.0, 0.0,0.0 }  };      //마스크만 변경

// 임시 메모리 할당(실수형)
double** tmpInImage = mallocDoubleMemory(inH + 2, inW + 2);
double** tmpOutImage = mallocDoubleMemory(outH, outW);

tmpInImage[i][k] = 127; // 임시 입력 메모리를 초기화(127) : 필요시 평균값

tmpInImage[i + 1][k + 1] = inImage[i][k]; // 입력 이미지 --> 임시 입력 이미지

// *** 회선 연산 ***
double S += tmpInImage[i + m][k + n] * mask[m][n];
tmpOutImage[i][k] = S;
~~tmpOutImage[i][k] += 127.0; //후처리~~

// 임시 출력 영상--> 출력 영상.
if (tmpOutImage[i][k] < 0.0)
outImage[i][k] = 0;
else if (tmpOutImage[i][k] > 255.0)
outImage[i][k] = 255;
else
outImage[i][k] = (unsigned char)tmpOutImage[i][k];
```

향후 발전 방향

- 더욱 다양한 영상처리 알고리즘 생성
- 입력값 오류 처리
- 효과 누적 및 취소 기능 추가
- 정방향이 아닌 다양한 크기의 영상 입력이 가능하게 조정