C언어 기반의 영상처리 프로그래밍

([Intel] 엣지 AI SW 아카데미)

과목 : 절차지향 프로그래밍

성명 : 유지승

목차

프로젝트 개요

Opencv를 사용하지 않고 c언어를 사용하여 영상처리 프로그램 구현

진행 기간	03월 7일 ~ 03 월 18일
개발 환경	OS : Windows 10 (64bit) Tool : Visual Studio 2022 언어 : C언어
동작 구성	inImage() ➔ processing(코딩한 영상 처리 함수 사용) ➔ outImage() ➔ printImage() (화면에 노출)

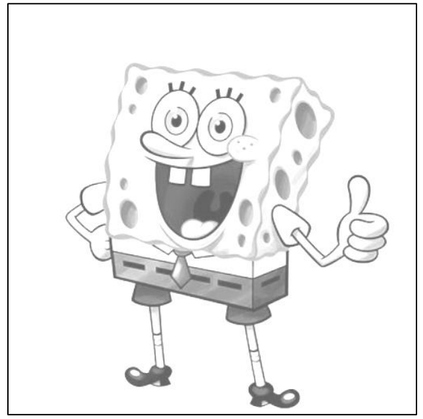
산술연산 밝기 처리

화소 점 처리

- 밝게(덧셈)

inImage()값에 int값을 입력 받은 만큼 더해 영상이 밝아짐.

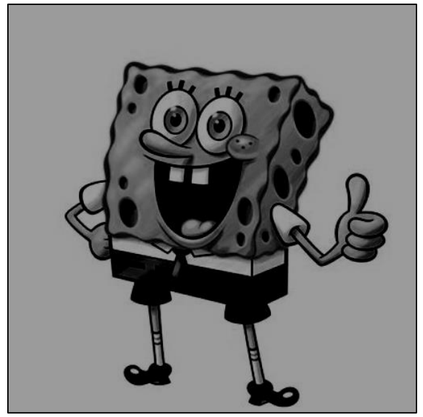
```
if (inImage[i][k] + val < 255)
    outImage[i][k] = inImage[i][k] + val;
else
    outImage[i][k] = 255;
```

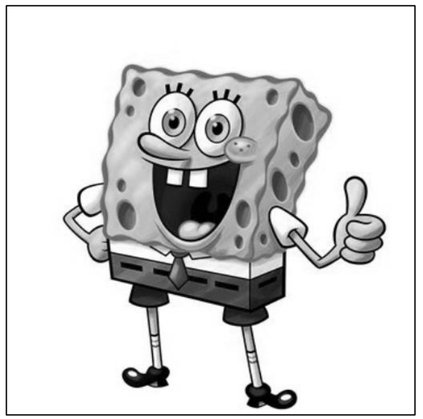


- 어둡게(뺄셈)

inImage()값에 int값을 입력 받은 만큼 빼서 영상이 어두워짐.

```
if (inImage[i][k] - val > 0) {
    outImage[i][k] = inImage[i][k] - val;
}
else outImage[i][k] = 0;
```





원본

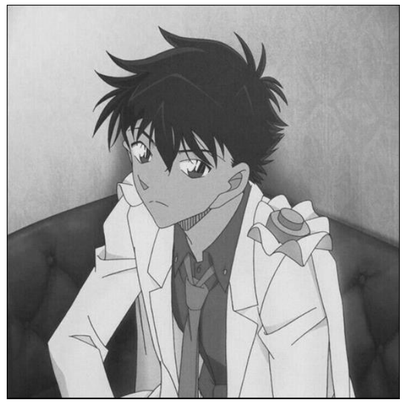
반전, 평균 흑백

화소 점 처리

- 반전

Max값인 255에서 inImage값을 빼서 노출시킴.

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



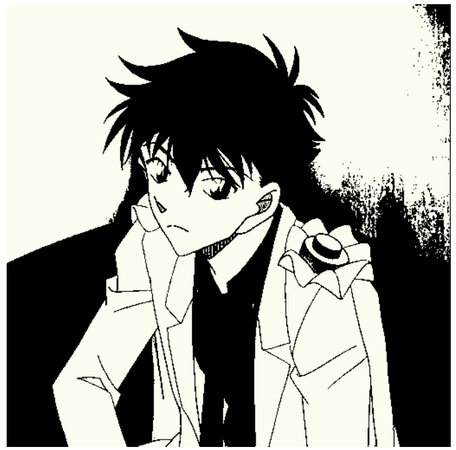
원본

- 흑백

화소의 평균보다 높으면 Max(255) 낮으면 Min(0) 으로 변경하여 노출시킴

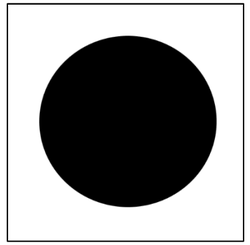
```
hap += inImage[i][k];
avg = hap / (inH * inW);
```

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

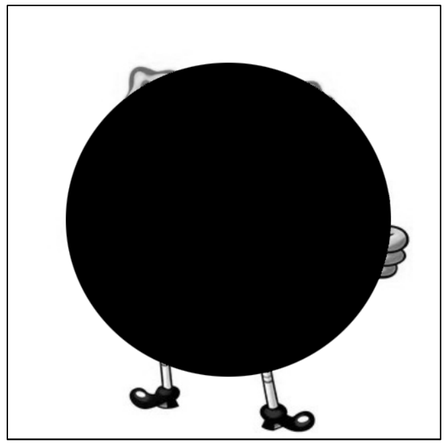


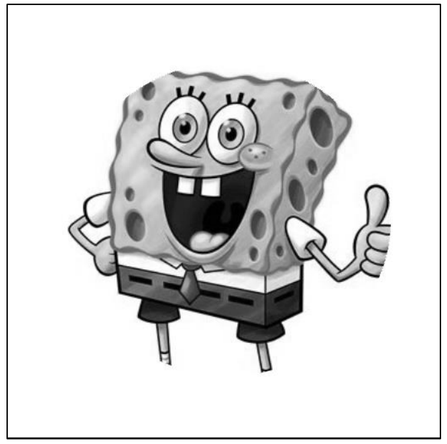
AND, OR, XOR

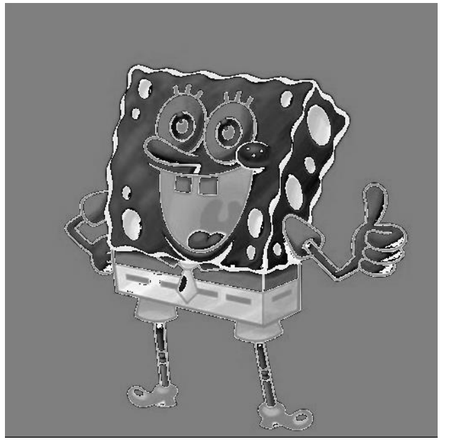
화소 점 처리



원형 mask







- AND

inImage값과 mask값을 비교하여 둘다 1인 부분 노출.

```
outImage[i][k] = inImage[i][k] & mask1[i][k];
```

- OR

inImage값에 mask값이 추가됨.

```
outImage[i][k] = inImage[i][k] | mask1[i][k];
```

- XOR

inImage값과 compare값 비교하여 다를때만 1출력

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

축소, 확대

기하학 처리

원본

▪ 축소

inImage값을 사용자의 입력값으로 나눠서 outImage로 출력.

```
outH = (int)(inH / scale);
outW = (int)(inW / scale);
```

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

▪ 확대

inImage값에 사용자가 입력한 값을 곱하여 outImage 출력

이미지에 홀(빈 픽셀)이 생겨 품질이 좋지 않음

```
outH = (int)(inH * scale);
outW = (int)(inW * scale);
```

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

확대 (보완 : 백워딩)

기하학 처리

▪ 확대 (포워딩)

포워딩으로 확대 시 홀(빈 픽셀)로 인한 품질 저하

```
outH = (int)(inH * scale);
outW = (int)(inW * scale);
```

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

▪ 확대(백워딩)

커진 outImage에 inImage파일을 가져와 빈 픽셀을 없앰.

```
outH = (int)(inH * scale);
outW = (int)(inW * scale);
```

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

양선형 보간법 확대처리

기하학 처리

원본

▪ 양선형 보간법

- 화소당 선형 보간을 세 번 수행하며, 새롭게 생성된 화소는 가장 가까운 화 소 네 개에 가중치를 곱한 값을 합해서 얻음.

```
outH = (inH*scale);
outW = (inW*scale);
mallocOutputMemory();

for (int i = 0; i < inH; i++) {
    for (int k = 0; k < inW; k++) {
        outImage[i * scale][k * scale] = inImage[i][k];
    }
}
```

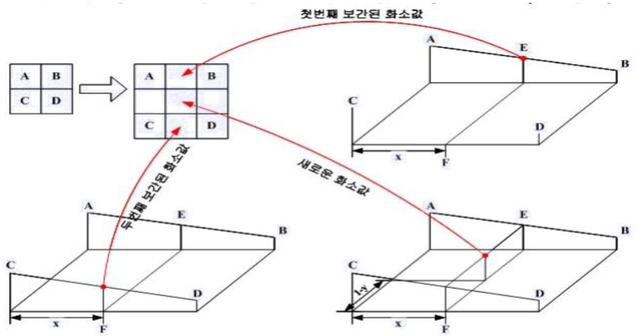
- 홀(빈 픽셀) 이외의 공간들을 inImage로 채움

```
r_H = i / scale;                        // 현재 위치를 scale 크기의 블록 중 몇 번째 쯤인지 연산
r_W = k / scale;

i_h = (int)floor(r_H);                  // 블록의 가로세로 칸을 저장
i_w = (int)floor(r_W);

s_H = r_H - i_h;                        // 현재 블록 내에서의 위치를 저장
s_W = r_W - i_w;
```

- 각 변수 선언

```
if (i_h < 0 | i_h >= (inH - 1) | i_w < 0 | i_w >= (inW - 1)) {
    outImage[i][k] = 255;               // 현재 칸이 보간을 진행할 수 없다면 흰색으로 지정
    // 주로 왼쪽과 아래쪽이 그런 경향이 있음
}
else {
    C1 = (double)inImage[i_h][i_w];             // 좌 상단점 지정
    C2 = (double)inImage[i_h][i_w + 1];         // 우 상단점 지정
    C3 = (double)inImage[i_h + 1][i_w + 1];     // 우 하단점 지정
    C4 = (double)inImage[i_h + 1][i_w];         // 좌 하단점 지정
    outImage[i][k] = (unsigned char)(C1 * (1 - s_H) * (1 - s_W)
        + C2 * s_W * (1 - s_H) + C3 * s_W * s_H + C4 * (1 - s_W) * s_H);
    // 각 점들과의 위치 비율을 반영해서 점의 색 지정
}
```

- 공식에 대입



(z : 찾고 싶은 값)

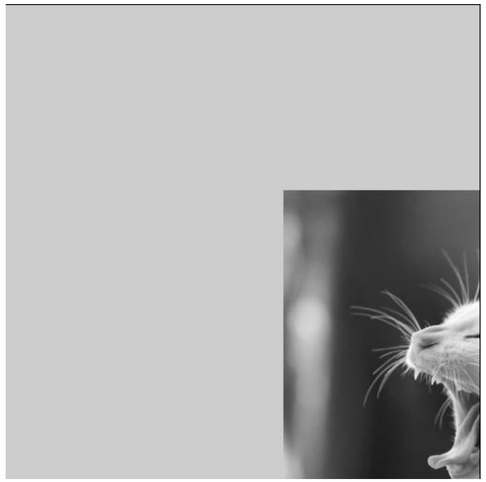
$$z = (1-p)(1-q)a + p(1-q)b + (1-p)qc + pqd$$

- 양선형 보간법 공식

- 결과값
  보다 부드러운
  스케일링 가능

이동, 회전 처리

기하학 처리



▪ 이동

inImage값에 사용자가 입력한 x와 y값을 덧셈,뺄셈을 하여 outImage를 이동시킴.

```
int xmove, ymove;
printf("x증가");
xmove = getIntValue();
printf("y증가");
ymove = getIntValue();
```

```
if ((0 <= i - xmove && i - xmove < outH) && (0 <= k - ymove && k - ymove < outW))
    outImage[i][k] = inImage[i - xmove][k - ymove];
```

▪ 회전

영상을 θ만큼의 각도로 회전 시킴

$$\begin{bmatrix} x_{dest} \\ y_{dest} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_{source} \\ y_{source} \end{bmatrix}$$

X, Y값 회전 공식

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

컴퓨터는 radian 값만 받을 수 있음

```
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];
```

회전 처리(보완)

기하학 처리

원본

- 회전

- **문제점**
  일반 회전 시 화면 밖으로 넘어가게 되고 홀(빈 픽셀)도 노출됨

- 회전(중앙,백워딩)

- 중앙값 좌표를 기준으로 회전 시키고 outImage에 inImage 매칭 시켜 홀(빈 픽셀)을 없앰.

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

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

- 중앙을 기준으로 회전시키고 백워딩을 이용하여 홀(빈 픽셀)을 없앰.
- 문제점 : 회전시켰을 시 그림이 화면에서 벗어남

- 회전(출력 이미지 크기 변경)

- 회전 시 변경된 사진의 크기를 다시 받은 뒤 outImage의 정중앙에 배치함.

```
double radian = -degree * 3.141592 / 180.0;
double radian90 = 90 * 3.141592 / 180.0;
outH =(int)(inW*cos(radian90 + radian)+ (inH *(cos(radian)))); //outImage 배열의 크기를 돌린 각도에 맞게 증가
outW = (int)(inH*cos(radian90 + radian) +(inW *(cos(radian))));
```

```
int xs = (int)(cos(radian) * (xd - outH / 2) + sin(radian) * (yd - outW / 2));
int ys = (int)(-sin(radian) * (xd - outH / 2) + cos(radian) * (yd - outW / 2));
xs += inH/2;
ys += inW/2;
if ((0 <= xs && xs < inH) && (0 <= ys && ys < inW))
    outImage[xd][yd] = inImage[xs][ys];
```

좌우 ,상하 반전

기하학 처리

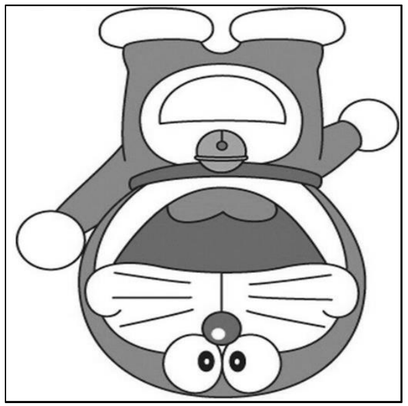
원본

▪ 좌우 반전

- inImage의 이중 배열의 행 끝 부분을 outImage의 첫 부분으로 출력.

```
outImage[i][k] = inImage[i][(inW - 1) - k];
```

▪ 상하 반전

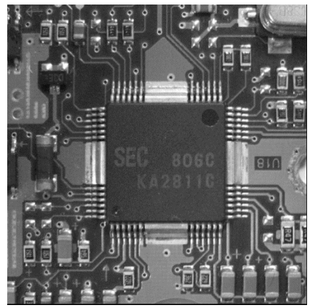


- inImage의 이중 배열의 열 끝 부분을 outImage의 첫 부분으로 출력.

```
outImage[i][k] = inImage[(inH - 1) - i][k];
```

엠보싱

화소 영역 처리



원본

- **엠보싱**
  - 입력 영상을 양각 형태로 보이게 하는 기술
  - 3x3마스크를 사용하여 영역 처리함.

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

- 3x3의 엠보싱 마스크 생성

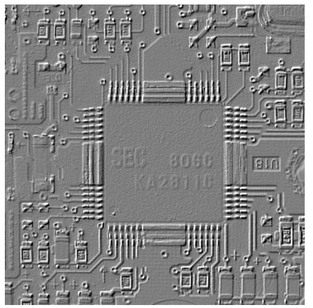
```
double** tmpInImage = mallocDoubleMemory(inH + 2, inW + 2);
double** tmpOutImage = mallocDoubleMemory(outH, outW);

// 임시 입력 메모리를 초기화(127) : 필요시 평균값
for (int i = 0; i < inH + 2; i++)
    for (int k = 0; k < inW + 2; k++)
        tmpInImage[i][k] = 127;
```

- 임시 입력 메모리와 출력 메모리 할당
- 임시 입력 메모리를 엠보싱 마스크 크기 만큼 크기 증가

```
// *** 회선 연산 ***
double S;
for (int i = 0; i < inH; i++) {
    for (int k = 0; k < inW; k++) {
        // 마스크(3x3) 와 한점을 중심으로한 3x3을 곱하기
        S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값.

        for (int m = 0; m < 3; m++)
            for (int n = 0; n < 3; n++)
                S += tmpInImage[i + m][k + n] * mask[m][n];

        tmpOutImage[i][k] = S;
    }
}
```

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

- 화소영역 처리공식 사용하여 대입 후 outImage출력



결과 이미지

블러링

화소 영역 처리

원본

- **블러링**
  - 영상의 세밀한 부분을 제거하여 영상을 흐리게 하거나 부드럽게 하는 기술.
  - 블러링 마스크의 크기에 따라 흐림의 강도가 높아짐

```
double mask[3][3] = { {1.0/9, 1.0 / 9, 1.0 / 9}, // 블러링 마스크
                      { 1.0 / 9, 1.0 / 9, 1.0 / 9},
                      { 1.0 / 9, 1.0 / 9, 1.0/9} };
// 임시 메모리 할당(실수형)
```

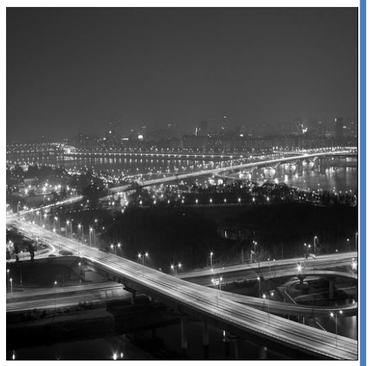
- 3x3의 블러링 마스크 생성

```
double mask[9][9] = { {1.0 / 81, 1.0 / 81, 1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81}, // 블러링 마스크
                      { 1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81},
                      { 1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81},
                        {1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81},
                        {1.0 / 81, 1.0 / 81, 1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81},
                        { 1.0 / 81, 1.0 / 81, 1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81 },
                        { 1.0 / 81, 1.0 / 81, 1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81 },
                        { 1.0 / 81, 1.0 / 81, 1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81 },
                        { 1.0 / 81, 1.0 / 81, 1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81,1.0 / 81 } };
```

- 9x9의 블러링 마스크 생성

샤프닝

화소 영역 처리



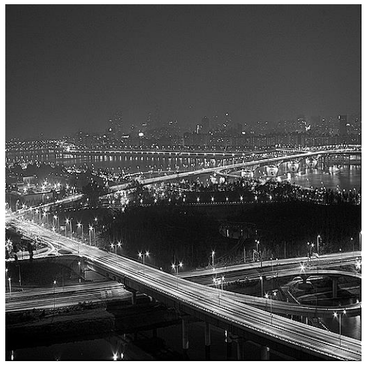
원본

▪ 샤프닝

- 블러링과는 반대되는 효과를 보이는 기법을 샤프닝(sharpening) 또는 영상 강화라고 함.
- 흐린 영상을 개선하여 선명한 영상을 생성하는 데 주로 사용됨.

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

- 3x3의 샤프닝 마스크 생성

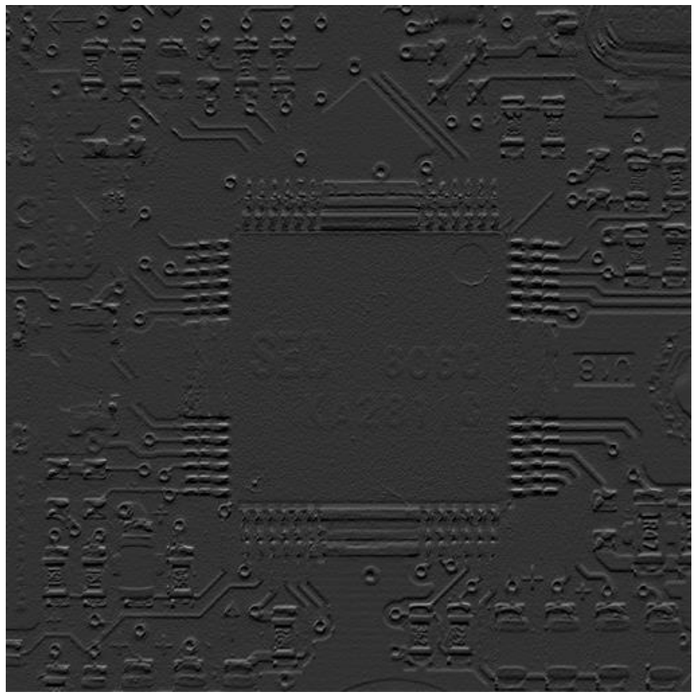


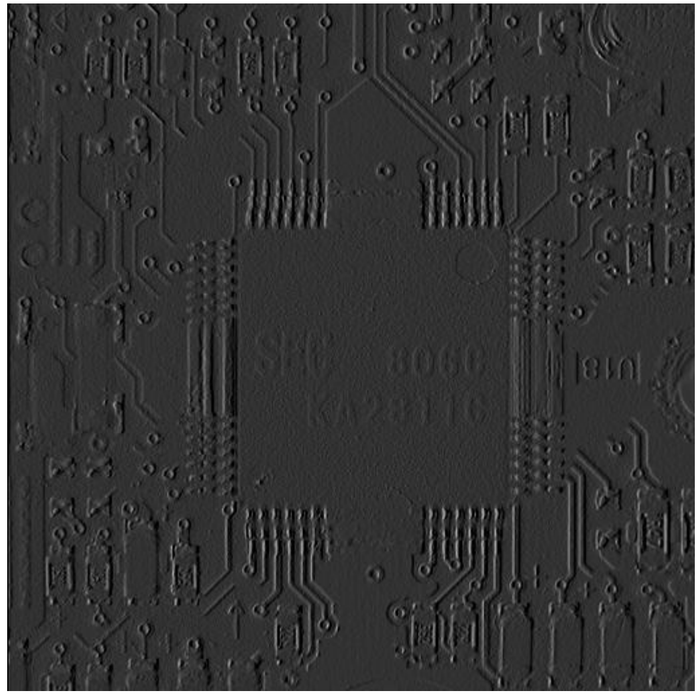
```
double mask[3][3] = { {0, -1.0, 0}, // 샤프닝 마스크2
                      {-1.0, 5.0 ,-1.0},
                      {0, -1.0 , 0} };
```

- 3x3 샤프닝 마스크 2 생성

수평, 수직 엣지검출

경계선 검출





▪ 수평 엣지검출

- 수평으로된 엣지를 감지함.

▪ 수직 엣지검출

- 수직으로된 엣지를 감지함.

스트레칭

히스토그램 처리

- **스트레칭**
  - 명암대비를 향상시키는 연산으로, 낮은 명암 대비를 보이는 영상의 화질을 향상 시킴
  - 이미지의 가장 밝은 화소 값과 가장 어두운 화소값을 기준으로 늘린다

$$new\ pixel = \frac{old\ \ pixel - low}{high - low} \times 255$$

- 기존 명암대비 스트레칭 공식

원본

```
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];
    }
}
```

- 최대 최소 화소 구하기

```
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;
    }
}
```

- 공식 대입

결과값

평활화

히스토그램 처리

원본

▪ 평활화 단계

- 어둡게 촬영된 영상의 히스토그램을 조절하여 명암분포를 균일하게 만들어줌.

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

```
//2단계 : 누적 히스토그램 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 이미지로 치환

for (int i = 0; i < inH; i++) {
    for (int k = 0; k < inW; k++) {
        outImage[i][k] = (unsigned char)normalHisto[inImage[i][k]];
    }
}
```

결과값

마무리

좋았던 점	• C언어에 대해서 다시 한번 공부할 수 있었음. • Opencv를 사용하지 않고 c언어만을 이용해 영상 처리하는 법을 배우게 됨. • 기능을 추가 하면서 추가하지 못한 기능들에 대해 생각해 보게 됨.
아쉬웠던 점	• 회전부분에서 약간의 혼돈이 왔는데 이부분으로 인해 다른 기능에 대한 시간이 부족했음. • 히스토그램과 화소 영역처리 부분의 다른 기능들을 구현할 시간 부족.
추후 계획	• 구현했던 영상처리 알고리즘의 완벽한 이해와 구현하지 못한 기능들의 구현. • c++ MFC를 활용하여 영상처리