[Intel] 엣지 AI SW 아카데미 _ 절차 지향 프로그래밍

C++ 언어로 구현한 Image Processing (Feat. MFC)

임지원

프로젝트 개요

목표

OpenCV 없이 C++ 언어를 기반으로 영상 처리 프로그램을 구현

개발 환경

OS : Windows 11 Pro
Tool : Microsoft Visual Studio 2022

코드

https://github.com/Jday4612/Image-Processing

프로젝트 구조도

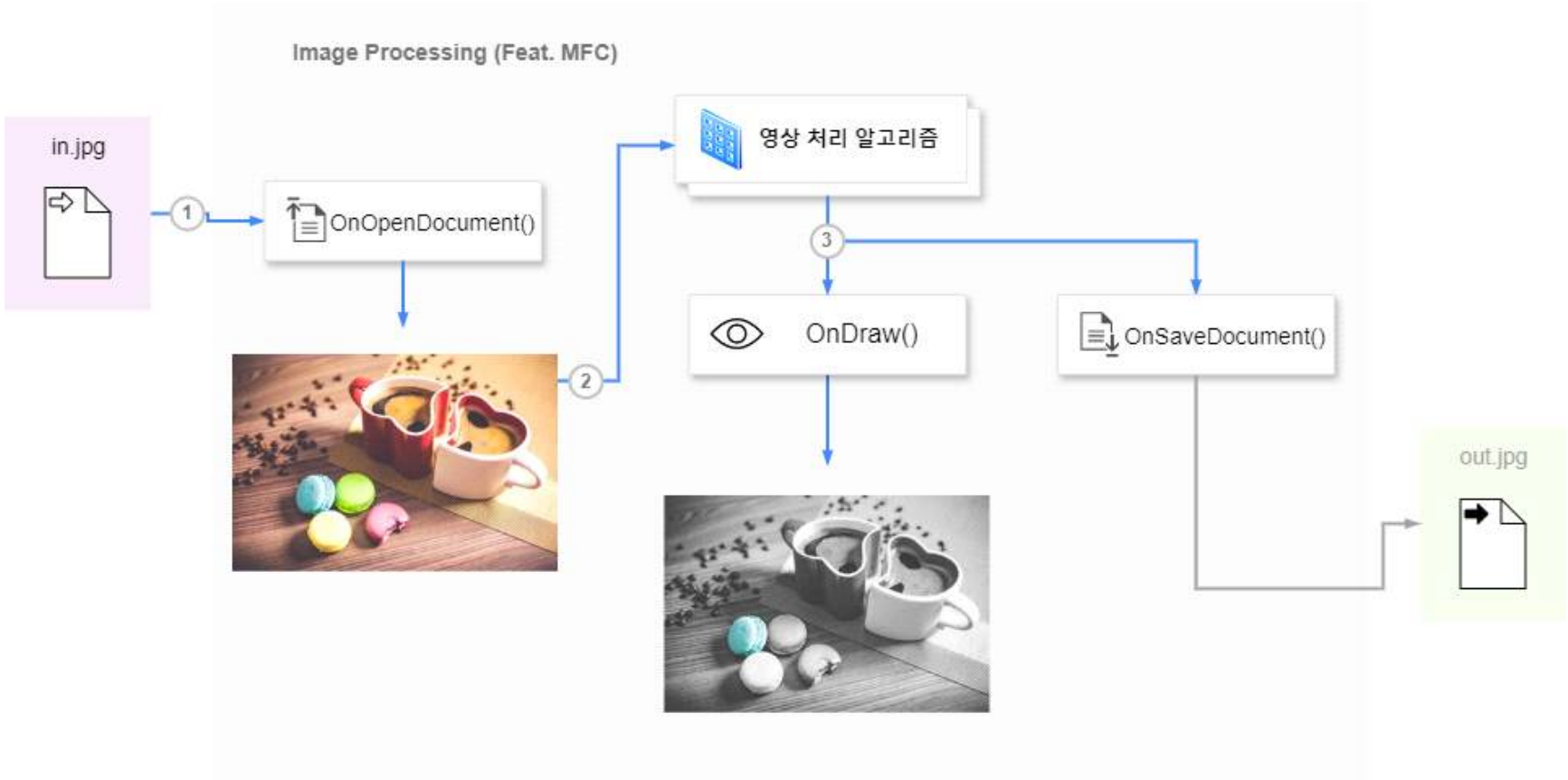
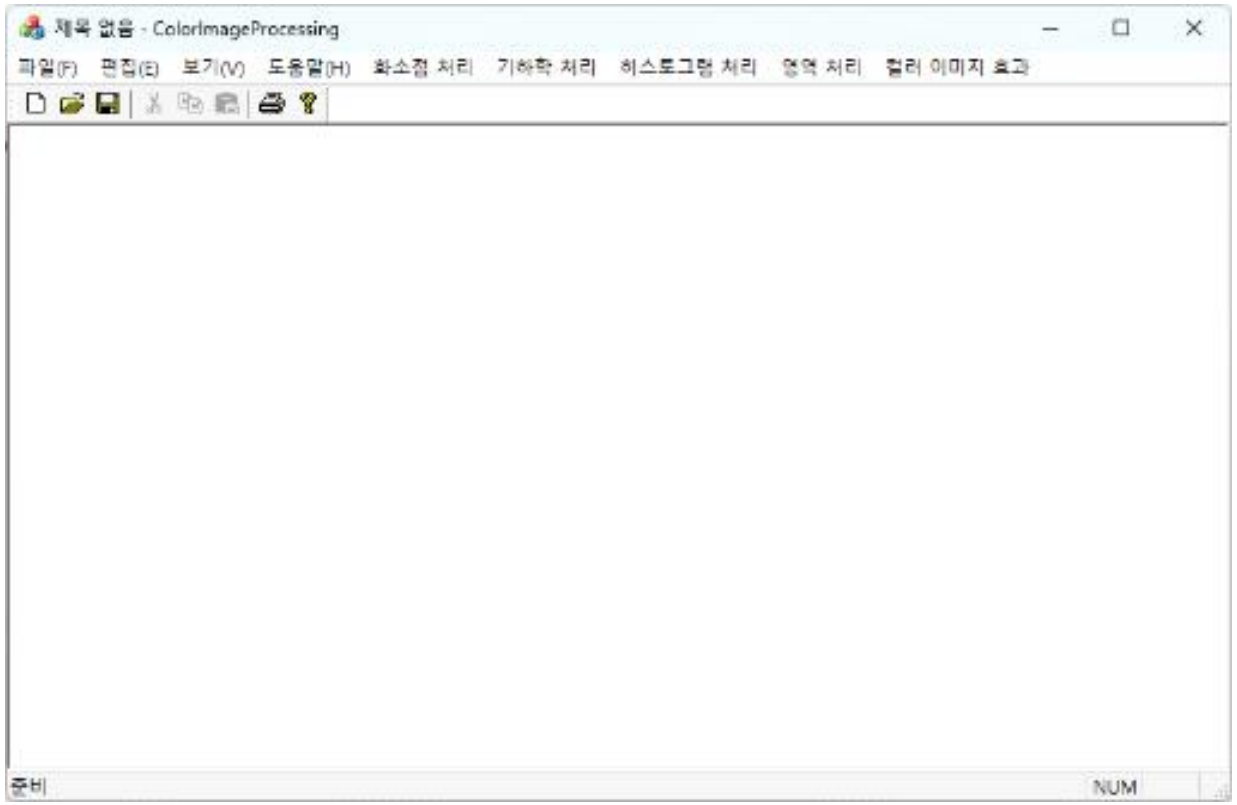


Image Processing (Feat. MFC)
in.jpg
1
OnOpenDocument()
영상 처리 알고리즘
3
OnDraw()
OnSaveDocument()
2
out.jpg

메인 화면



화소점 처리

- 동일 이미지
- 그레이스케일
- 밝기 변경 >
- AND
- OR
- XOR
- NOT
- 감마
- 파라볼라 >
- 포스터라이징
- 강조
- 흑백 >

기하학 처리

- 축소 >
- 확대 >
- 회전 >
- 미러링 >
- 이동
- 모핑

영역 처리

- 엠보싱 >
- 블러링 >
- 샤프닝 >
- 가우시안
- 경계선 >

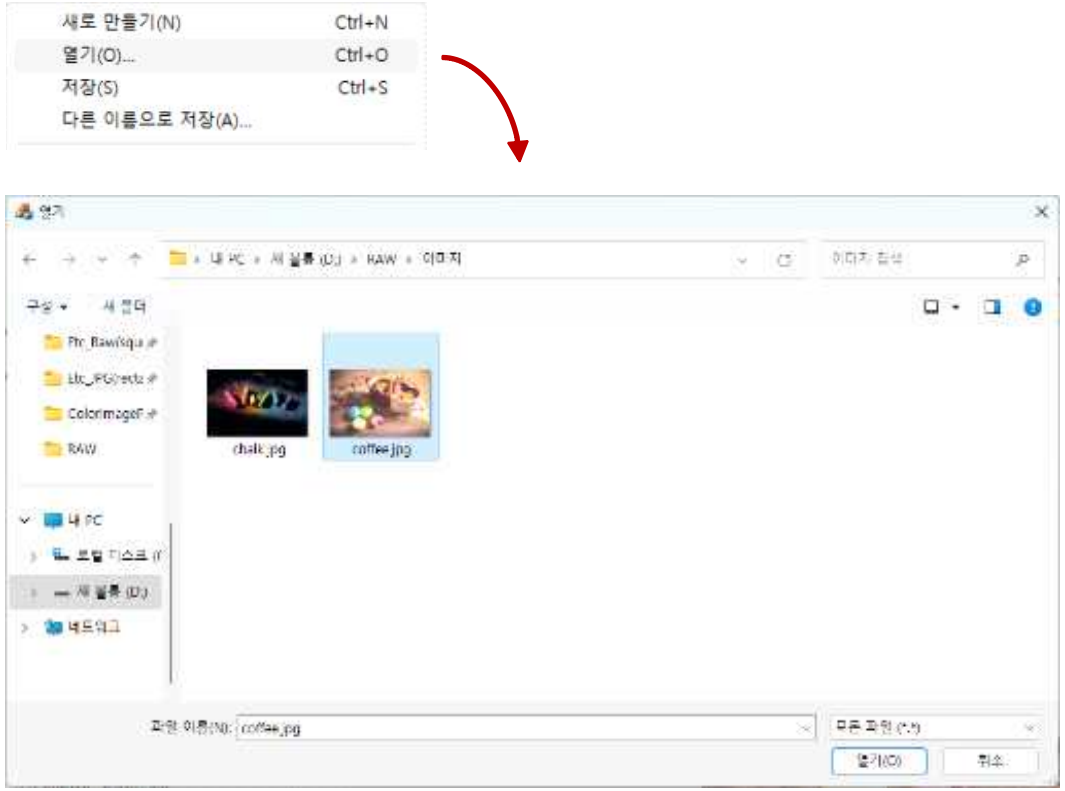
히스토그램 처리

- 스트레칭
- endIn
- 평활화

컬러 이미지 효과

- 채도 변경
- 색 추출

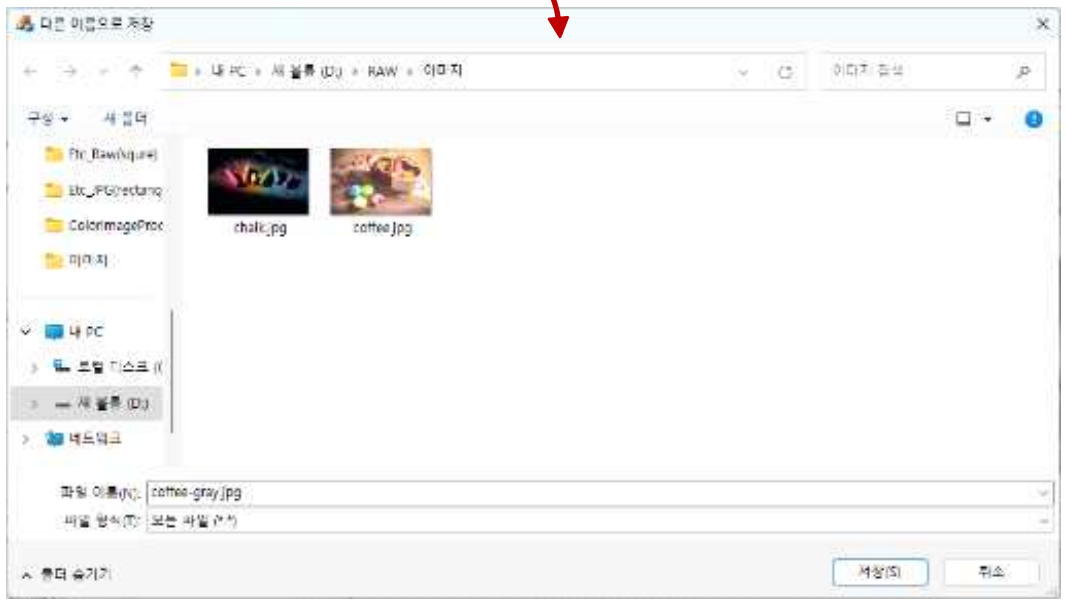
파일 - 열기



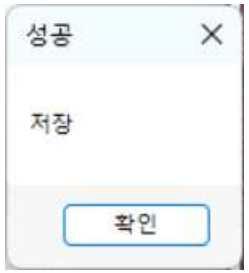
선택된 이미지 파일 출력

파일 - 저장

원하는 이름으로 파일 저장



성공적으로 저장 시,
[저장 성공] 창과 함께
폴더에 저장된 모습 확인 가능



화소점 처리 - 동일

: 원 화소의 값이나 위치를 바탕으로 단일 화소 값을 변경하는 기술

Input

Output

[화소점 처리 – 동일 이미지]

입력 값을 그대로 출력 값으로 변환

```
m_outImageR[i][j] = m_inImageR[i][j];
m_outImageG[i][j] = m_inImageG[i][j];
m_outImageB[i][j] = m_inImageB[i][j];
```

화소 점 처리 - Grayscale

Input

Output

[화소점 처리 – 그레이스케일]

RGB 이미지를 회색조 이미지로 변환

```
avg = (m_inImageR[i][j] + m_inImageG[i][j] + m_inImageB[i][j]) / 3.0;
m_outImageR[i][j] = m_outImageG[i][j] = m_outImageB[i][j] = (unsigned char)avg;
```

화소 점 처리 - 밝게/어둡게

Input1

Output1 (+50)

Input2

Output2 (-50)

[밝기 변경 > 밝게/어둡게]

정수(-255 ~ 255) 입력 시,
해당 값을 더해 밝기 변경

```
// R
px = m_inImageR[i][j] + value;

if (px < 0)
    m_outImageR[i][j] = 0;
else if (255 < px)
    m_outImageR[i][j] = 255;
else
    m_outImageR[i][j] = px;

// G
px = m_inImageG[i][j] + value;

if (px < 0)
    m_outImageG[i][j] = 0;
else if (255 < px)
    m_outImageG[i][j] = 255;
else
    m_outImageG[i][j] = px;

// B
px = m_inImageB[i][j] + value;

if (px < 0)
    m_outImageB[i][j] = 0;
else if (255 < px)
    m_outImageB[i][j] = 255;
else
    m_outImageB[i][j] = px;
```

화소 점 처리 - 더 밝게/어둡게

Input1

Output1 (2.0)

Input2

Output2 (0.5)

[밝기 변경 > 더 밝게/어둡게]

소수(0.0 ~) 입력 시,
해당 값을 곱해 밝기 변경

```
// R
px = m_inImageR[i][j] + value;

if (px < 0.0)
    m_outImageR[i][j] = 0;
else if (255.0 < px)
    m_outImageR[i][j] = 255;
else
    m_outImageR[i][j] = (unsigned char)px;

// G
px = m_inImageG[i][j] + value;

if (px < 0.0)
    m_outImageG[i][j] = 0;
else if (255.0 < px)
    m_outImageG[i][j] = 255;
else
    m_outImageG[i][j] = (unsigned char)px;

// B
px = m_inImageB[i][j] + value;

if (px < 0.0)
    m_outImageB[i][j] = 0;
else if (255.0 < px)
    m_outImageB[i][j] = 255;
else
    m_outImageB[i][j] = (unsigned char)px;
```

화소 점 처리 - AND

Input

Output

[화소점 처리 – AND]

특정 데이터와 AND 연산 수행

```
// R
px = (m_inImageR[i][j] & m_dataImage[i][j]);

if (px < 0)
    m_outImageR[i][j] = 0;
else if (255 < px)
    m_outImageR[i][j] = 255;
else
    m_outImageR[i][j] = px;

// G
px = (m_inImageG[i][j] & m_dataImage[i][j]);

if (px < 0)
    m_outImageG[i][j] = 0;
else if (255 < px)
    m_outImageG[i][j] = 255;
else
    m_outImageG[i][j] = px;

// B
px = (m_inImageB[i][j] & m_dataImage[i][j]);

if (px < 0)
    m_outImageB[i][j] = 0;
else if (255 < px)
    m_outImageB[i][j] = 255;
else
    m_outImageB[i][j] = px;
```

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

화소 점 처리 - OR

Input

Output

[화소점 처리 – OR]

특정 데이터와 OR 연산 수행

```
// R
px = (m_inImageR[i][j] | m_dataImage[i][j]);

if (px < 0)
    m_outImageR[i][j] = 0;
else if (255 < px)
    m_outImageR[i][j] = 255;
else
    m_outImageR[i][j] = px;

// G
px = (m_inImageG[i][j] | m_dataImage[i][j]);

if (px < 0)
    m_outImageG[i][j] = 0;
else if (255 < px)
    m_outImageG[i][j] = 255;
else
    m_outImageG[i][j] = px;

// B
px = (m_inImageB[i][j] | m_dataImage[i][j]);

if (px < 0)
    m_outImageB[i][j] = 0;
else if (255 < px)
    m_outImageB[i][j] = 255;
else
    m_outImageB[i][j] = px;
```

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

화소 점 처리 - XOR

Input

Output

[화소점 처리 – XOR]

특정 데이터와 XOR 연산 수행

```
// R
px = (m_inImageR[i][j] ^ m_dataImage[i][j]);

if (px < 0)
    m_outImageR[i][j] = 0;
else if (255 < px)
    m_outImageR[i][j] = 255;
else
    m_outImageR[i][j] = px;

// G
px = (m_inImageG[i][j] ^ m_dataImage[i][j]);

if (px < 0)
    m_outImageG[i][j] = 0;
else if (255 < px)
    m_outImageG[i][j] = 255;
else
    m_outImageG[i][j] = px;

// B
px = (m_inImageB[i][j] ^ m_dataImage[i][j]);

if (px < 0)
    m_outImageB[i][j] = 0;
else if (255 < px)
    m_outImageB[i][j] = 255;
else
    m_outImageB[i][j] = px;
```

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

화소 점 처리 - NOT

Input

Output

[화소점 처리 – NOT]

화소 비트가 모두 반전됨

```
m_outImageR[i][j] = 255 - m_inImageR[i][j];
m_outImageG[i][j] = 255 - m_inImageG[i][j];
m_outImageB[i][j] = 255 - m_inImageB[i][j];
```

화소 점 처리 - 감마

Input

Output(0.5)

[화소점 처리 – 감마]

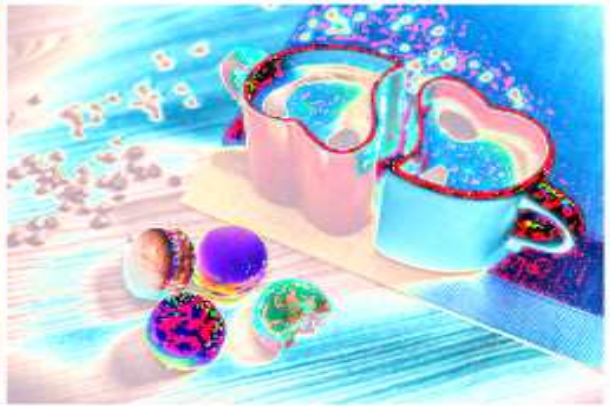
소수(0.0 ~) 입력 시,
해당 값에 따라 밝기 조절

```
// R
px = 255 * pow(m_inImageR[i][j] / 255.0, gamma);

if (px < 0.0)
    m_outImageR[i][j] = 0;
else if (255.0 < px)
    m_outImageR[i][j] = 255;
else
    m_outImageR[i][j] = (unsigned char)px;

// G
px = 255 * pow(m_inImageG[i][j] / 255.0, gamma);

if (px < 0.0)
    m_outImageG[i][j] = 0;
else if (255.0 < px)
    m_outImageG[i][j] = 255;
else
    m_outImageG[i][j] = (unsigned char)px;

// B
px = 255 * pow(m_inImageB[i][j] / 255.0, gamma);

if (px < 0.0)
    m_outImageB[i][j] = 0;
else if (255.0 < px)
    m_outImageB[i][j] = 255;
else
    m_outImageB[i][j] = (unsigned char)px;
```

화소 점 처리 – 파라볼라(CAP)

Input

Output



[파라볼라 > CAP]

파라볼라 CAP 수식을 통해 변환
-> 밝은 곳이 입체적으로 보임

```
m_outImageR[i][j] = (int)(-255 * pow(m_inImageR[i][j] / 127.0 - 1.0, 2) + 255.0);
m_outImageG[i][j] = (int)(-255 * pow(m_inImageG[i][j] / 127.0 - 1.0, 2) + 255.0);
m_outImageB[i][j] = (int)(-255 * pow(m_inImageB[i][j] / 127.0 - 1.0, 2) + 255.0);
```

화소 점 처리 – 파라볼라(CUP)

Input

Output

[파라볼라 > CUP]

파라볼라 CUP 수식을 통해 변환
-> 어두운 곳이 입체적으로 보임

```
m_outImageR[i][j] = (int)(255.0 * pow(m_inImageR[i][j] / 127.0 - 1.0, 2));
m_outImageG[i][j] = (int)(255.0 * pow(m_inImageG[i][j] / 127.0 - 1.0, 2));
m_outImageB[i][j] = (int)(255.0 * pow(m_inImageB[i][j] / 127.0 - 1.0, 2));
```

화소 점 처리 - 포스터라이징

Input

Output(8)

[화소점 처리 – 포스터라이징]

정수(3 ~) 입력 시,
해당 값으로 명암 값 수를 변경

```
// R
for (int k = 1; k < bit; k++) {
    if (k == (bit - 1))
        m_outImageR[i][j] = 255;
    else if (m_inImageR[i][j] <= 255 / (bit - 1) * k) {
        if (k == 1)
            m_outImageR[i][j] = 0;
        else
            m_outImageR[i][j] = 255 / (bit - 2) * (k - 1);

        break;
    }
}
```

화소 점 처리 - 강조

Input

Output(50, 100)

[화소점 처리 – 강조]

시작 지점(0 ~ 255)과
끝 지점(시작 ~ 255) 입력 시,
해당 범위 내의 화소만 강조

```
// R
if (start <= m_inImageR[i][j] && m_inImageR[i][j] <= end)
    m_outImageR[i][j] = 255;
else
    m_outImageR[i][j] = m_inImageR[i][j];

// G
if (start <= m_inImageG[i][j] && m_inImageG[i][j] <= end)
    m_outImageG[i][j] = 255;
else
    m_outImageG[i][j] = m_inImageG[i][j];

// B
if (start <= m_inImageB[i][j] && m_inImageB[i][j] <= end)
    m_outImageB[i][j] = 255;
else
    m_outImageB[i][j] = m_inImageB[i][j];
```

화소 점 처리 - 흑백

Input

Output

[흑백 > 기본]

각 화소를 128을 기준으로 이진화 적용

```
// R
if (m_inImageR[i][j] < 128)
    m_outImageR[i][j] = 0;
else
    m_outImageR[i][j] = 255;

// G
if (m_inImageG[i][j] < 128)
    m_outImageG[i][j] = 0;
else
    m_outImageG[i][j] = 255;

// B
if (m_inImageB[i][j] < 128)
    m_outImageB[i][j] = 0;
else
    m_outImageB[i][j] = 255;
```

화소 점 처리 - 흑백(평균값)

Input

Output

[흑백 > 평균값]

각 화소를 평균값을 기준으로 이진화 적용

```
for (int i = 0; i < m_inH; i++)
    for (int j = 0; j < m_inW; j++) {
        avgR += m_inImageR[i][j];
        avgG += m_inImageG[i][j];
        avgB += m_inImageB[i][j];
    }

avgR /= m_inH * m_inW;
avgG /= m_inH * m_inW;
avgB /= m_inH * m_inW;
```

```
for (int i = 0; i < m_inH; i++) {
    for (int j = 0; j < m_inW; j++) {
        // R
        if (m_inImageR[i][j] < avgR)
            m_outImageR[i][j] = 0;
        else
            m_outImageR[i][j] = 255;

        // G
        if (m_inImageG[i][j] < avgG)
            m_outImageG[i][j] = 0;
        else
            m_outImageG[i][j] = 255;

        // B
        if (m_inImageB[i][j] < avgB)
            m_outImageB[i][j] = 0;
        else
            m_outImageB[i][j] = 255;
    }
}
```

화소 점 처리 - 흑백(중앙값)

Input

Output

[흑백 > 중앙값]

각 화소를 중앙값을 기준으로 이진화 적용

```
for (int i = 0; i < m_inH; i++) {
    for (int j = 0; j < m_inW; j++) {
        arrR[cnt] = m_inImageR[i][j];
        arrG[cnt] = m_inImageG[i][j];
        arrB[cnt] = m_inImageB[i][j];
        cnt++;
    }
}

std::sort(arrR, arrR + cnt);
std::sort(arrG, arrG + cnt);
std::sort(arrB, arrB + cnt);

medR = arrR[(int)(m_inH * m_inW / 2)];
medG = arrG[(int)(m_inH * m_inW / 2)];
medB = arrB[(int)(m_inH * m_inW / 2)];
```

```
for (int i = 0; i < m_inH; i++) {
    for (int j = 0; j < m_inW; j++) {
        // R
        if (m_inImageR[i][j] < medR)
            m_outImageR[i][j] = 0;
        else
            m_outImageR[i][j] = 255;

        // G
        if (m_inImageG[i][j] < medG)
            m_outImageG[i][j] = 0;
        else
            m_outImageG[i][j] = 255;

        // B
        if (m_inImageB[i][j] < medB)
            m_outImageB[i][j] = 0;
        else
            m_outImageB[i][j] = 255;
    }
}
```

기하학 처리 - 축소

: 영상을 구성하는 화소의 공간적 위치를 재배치하는 기술

Input

Output(2)

[축소 > 기본]

정수(1 ~) 입력 시,
포워딩으로 (해당 값)배 축소

```
m_outImageR[i / scale][j / scale] = m_inImageR[i][j];
m_outImageG[i / scale][j / scale] = m_inImageG[i][j];
m_outImageB[i / scale][j / scale] = m_inImageB[i][j];
```

기하학 처리 - 축소(평균값)

Input

Output(2)

[축소 > 평균값]

정수(1 ~) 입력 시,
평균값을 기준으로 (해당 값)배 축소

```
int avgR = 0, avgG = 0, avgB = 0, cnt = 0;

for (int n = 0; n < scale; n++) {
    if (m_inH <= i + n)
        break;

    for (int m = 0; m < scale; m++) {
        if (m_inW <= j + m)
            break;

        avgR += m_inImageR[i + n][j + m];
        avgG += m_inImageG[i + n][j + m];
        avgB += m_inImageB[i + n][j + m];
        cnt++;
    }
}

m_outImageR[i / scale][j / scale] = avgR / cnt;
m_outImageG[i / scale][j / scale] = avgG / cnt;
m_outImageB[i / scale][j / scale] = avgB / cnt;
```

기하학 처리 - 축소(중앙값)

Input

Output(2)

[축소 > 중앙값]

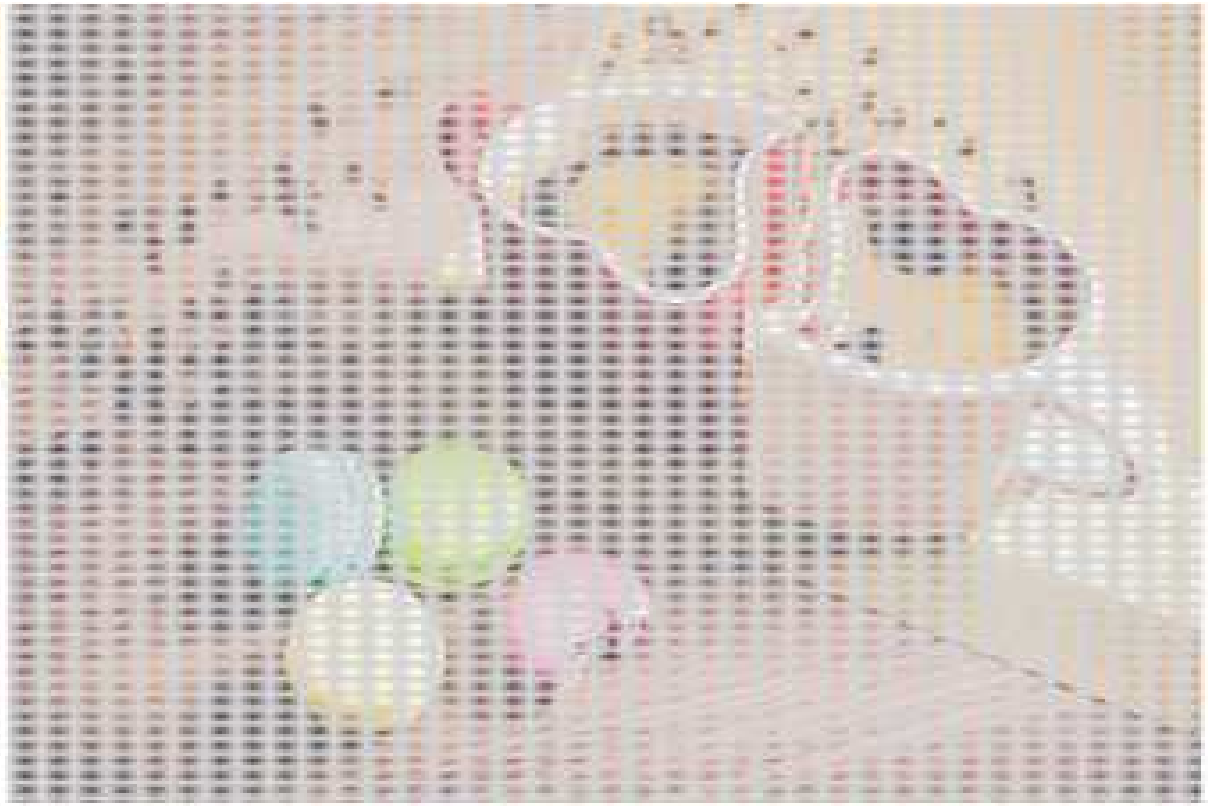
정수(1 ~) 입력 시,
중앙값을 기준으로 (해당 값)배 축소

```
int medR, medG, medB, cnt = 0;

for (int n = 0; n < scale; n++) {
    if (m_inH <= i + n)
        break;

    for (int m = 0; m < scale; m++) {
        if (m_inW <= j + m)
            break;

        arrR[cnt] = m_inImageR[i + n][j + m];
        arrG[cnt] = m_inImageG[i + n][j + m];
        arrB[cnt] = m_inImageB[i + n][j + m];
        cnt++;
    }
}

std::sort(arrR, arrR + cnt);
std::sort(arrG, arrG + cnt);
std::sort(arrB, arrB + cnt);

medR = arrR[cnt / 2];
medG = arrG[cnt / 2];
medB = arrB[cnt / 2];

m_outImageR[i / scale][j / scale] = medR;
m_outImageG[i / scale][j / scale] = medG;
m_outImageB[i / scale][j / scale] = medB;
```

기하학 처리 - 확대(포워딩)

Input

Output(2)



[확대 > 포워딩]

정수(1 ~) 입력 시,
포워딩으로 (해당 값)배 확대

```
m_outImageR[i / scale][j / scale] = m_inImageR[i][j];
m_outImageG[i / scale][j / scale] = m_inImageG[i][j];
m_outImageB[i / scale][j / scale] = m_inImageB[i][j];
```

기하학 처리 - 확대(백워딩)

Input

Output(2)

[확대 > 백워딩]

정수(1 ~) 입력 시,
백워딩으로 (해당 값)배 확대

```
m_outimageR[i][j] = m_inimageR[i / scale][j / scale];
m_outimageG[i][j] = m_inimageG[i / scale][j / scale];
m_outimageB[i][j] = m_inimageB[i / scale][j / scale];
```

기하학 처리 - 확대(양선형 보간)

Input

Output(2)

[확대 > 양선형 보간]

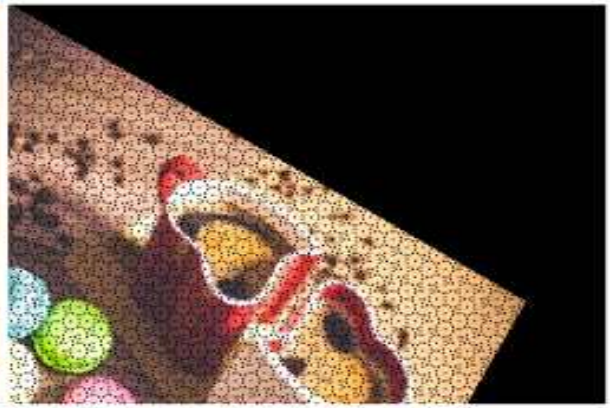
정수(1 ~) 입력 시,
양선형 보간으로 (해당 값)배 확대

```
double rH = (double)i / scale;
double rW = (double)j / scale;

int iH = (int)floor(rH);
int iW = (int)floor(rW);

double sH = rH - iH;
double sW = rW - iW;

if (iH < 0 || m_inH - 1 <= iH || iW < 0 || m_inW - 1 <= iW)
    m_outImageR[i][j] = m_outImageG[i][j] = m_outImageB[i][j] = 255;
else {
    // R
    double C1 = (double)m_inImageR[iH][iW];
    double C2 = (double)m_inImageR[iH][iW + 1];
    double C3 = (double)m_inImageR[iH + 1][iW + 1];
    double C4 = (double)m_inImageR[iH + 1][iW];

    m_outImageR[i][j] = (unsigned char)(C1 * (1 - sH) * (1 - sW)
        + C2 * sW * (1 - sH) + C3 * sW * sH + C4 * (1 - sW) * sH);
```

기하학 처리 - 회전

Input

Output(30)



[회전 > 기본]

정수(0 ~ 360) 입력 시,
해당 값만큼 회전

```
int xs = i;
int ys = j;

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

if ((0 <= xd && xd < m_outH) && (0 <= yd && yd < m_outW)) {
    m_outImageR[xd][yd] = m_inImageR[xs][ys];
    m_outImageG[xd][yd] = m_inImageG[xs][ys];
    m_outImageB[xd][yd] = m_inImageB[xs][ys];
}
```

기하학 처리 - 회전(중앙&백워딩)

Input

Output(30)

[회전 > 중앙&백워딩]

정수(0 ~ 360) 입력 시,
중앙을 기준 삼아 백워딩으로 해당 값만큼 회전

```
int xd = i;
int yd = j;

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 < m_outH) && (0 <= ys && ys < m_outW)) {
    m_outImageR[xd][yd] = m_inImageR[xs][ys];
    m_outImageG[xd][yd] = m_inImageG[xs][ys];
    m_outImageB[xd][yd] = m_inImageB[xs][ys];
}
```

기하학 처리 - 회전(확대)

Input

Output(2, 30)

[회전 > 확대]

확대 정도(1 ~)와
회전 각도 (0 ~ 360) 입력 시,
해당 값만큼 확대 및 회전

기하학 처리 - 회전(확대&양선형)

Input

Output(2, 30)

[회전 > 확대&양선형]

확대 정도(1 ~)와
회전 각도 (0 ~ 360) 입력 시,
해당 값만큼 양선형 확대 및 회전

기하학 처리 - 미러링(상하)

Input

Output

[미러링 > 상하]

수직을 기준으로 이미지 뒤집기

```
m_outImageR[i][j] = m_inImageR[m_inH - 1 - i][j];
m_outImageG[i][j] = m_inImageG[m_inH - 1 - i][j];
m_outImageB[i][j] = m_inImageB[m_inH - 1 - i][j];
```

기하학 처리 - 미러링(좌우)

Input

Output

[미러링 > 좌우]

수평을 기준으로 이미지 뒤집기

```
m_outImageR[i][j] = m_inImageR[i][m_inW - 1 - j];
m_outImageG[i][j] = m_inImageG[i][m_inW - 1 - j];
m_outImageB[i][j] = m_inImageB[i][m_inW - 1 - j];
```

기하학 처리 - 이동

Input

Output(20, 30)

[기하학 처리 - 이동]

이동할 Height와 Width 입력 시,
기존 위치에서 해당 값만큼 이동

```
int nx = i + posH;
int ny = j + posW;

if ((0 <= nx && nx < m_outH) && (0 <= ny && ny < m_outW)) {
    m_outImageR[nx][ny] = m_inImageR[i][j];
    m_outImageG[nx][ny] = m_inImageG[i][j];
    m_outImageB[nx][ny] = m_inImageB[i][j];
}
```

기하학 처리 - 모핑

Input

Output

[기하학 처리 - 모핑]

모핑할 다른 파일을 선택 시,
모핑 변환 (두 파일을 혼합)

```
u = ((i * m_inH) + (j + 1.0)) / (m_inH * m_inW);
// R
tmp = (int)((1.0 - u) * m_inImageR[i][j] + u * tmpImageR[i][j]);

if (tmp < 0)
    m_outImageR[i][j] = 0;
else if (255 < tmp)
    m_outImageR[i][j] = 255;
else
    m_outImageR[i][j] = (unsigned char)tmp;
```

히스토그램 처리 - 스트레칭

히스토그램 : 데이터를 막대 그래프 모양으로 나타낸 것

Input

Output

[히스토그램 처리 - 스트레칭]

특정 부분에 집중된 히스토그램을
모든 영역으로 확장

```
// R
if (m_inImageR[i][j] < lowR)
    lowR = m_inImageR[i][j];

if (highR < m_inImageR[i][j])
    highR = m_inImageR[i][j];
```

```
// R
px = (int)((double)(m_inImageR[i][j] - lowR) / (highR - lowR) * 255.0);

if (px < 0)
    m_outImageR[i][j] = 0;
else if (255 < px)
    m_outImageR[i][j] = 255;
else
    m_outImageR[i][j] = px;
```

히스토그램 처리 - endIn

Input

Output

[히스토그램 처리 - endIn]

일정한 양의 화소를 0이나 255로 지정

```
// R
if (m_inImageR[i][j] < lowR)
    lowR = m_inImageR[i][j];

if (highR < m_inImageR[i][j])
    highR = m_inImageR[i][j];
```

```
// R
highR -= 50;
lowR += 50;
```

```
// R
px = (int)((double)(m_inImageR[i][j] - lowR) / (double)(highR - lowR) * 255.0);

if (px < 0)
    m_outImageR[i][j] = 0;
else if (255 < px)
    m_outImageR[i][j] = 255;
else
    m_outImageR[i][j] = px;
```

히스토그램 처리 - 평활화

Input

Output

[히스토그램 처리 - 평활화]

영상의 밝기 분포를 재분배하여
명암 대비를 최대화

```
// 1단계 : 빈도 수 세기 (-히스토그램) histo[256]
int histoR[256] = { 0 }, histoG[256] = { 0 }, histoB[256] = { 0 };

for (int i = 0; i < m_inH; i++)
    for (int j = 0; j < m_inW; j++) {
        histoR[m_inImageR[i][j]]++;
```

```
// 2단계 : 누적 히스토그램 생성
int sumHistoR[256] = { 0 }, sumHistoG[256] = { 0 }, sumHistoB[256] = { 0 };
sumHistoR[0] = histoR[0];
sumHistoG[0] = histoG[0];
sumHistoB[0] = histoB[0];

for (int i = 1; i < 256; i++) {
    sumHistoR[i] = sumHistoR[i - 1] + histoR[i];
```

```
// 3단계 : 정규화된 히스토그램 생성 normalHisto = sumHisto * (1.0 / (inH * inW)) * 255.0
double normalHistoR[256] = { 0 }, normalHistoG[256] = { 0 }, normalHistoB[256] = { 0 };

for (int i = 0; i < 256; i++) {
    normalHistoR[i] = sumHistoR[i] * (1.0 / (m_inH * m_inW)) * 255.0;
```

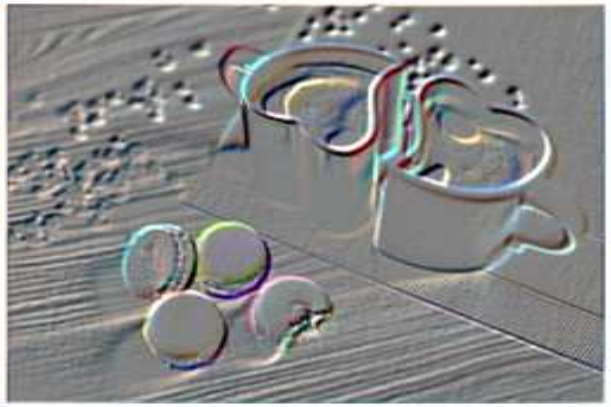
```
m_outImageR[i][j] = (unsigned char)normalHistoR[m_inImageR[i][j]];
```

영역 처리 - 엠보싱(RGB)

: 해당 입력 화소뿐만 아니라 그 주변의 화소도 함께 고려하는 공간 영역 연산

Input

Output



[엠보싱 > RGB]

RGB 컬러 모델을 양각 형태로 변환

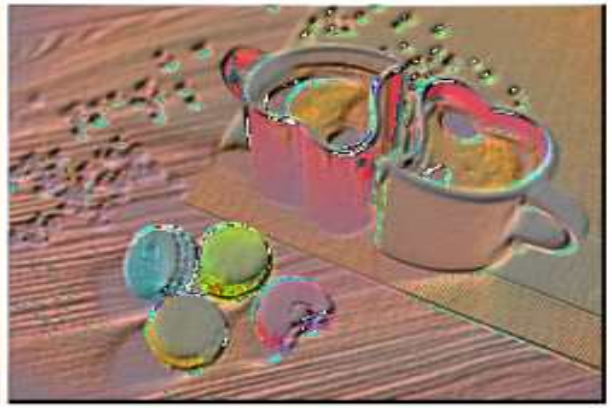
```
double mask[MSIZE][MSIZE] = {
    { -1.0, 0.0, 0.0 },
    {  0.0, 0.0, 0.0 },
    {  0.0, 0.0, 1.0 } }; // 엠보싱 마스크
```

```
// 마스크(3x3)와 한 점을 중심으로 한 3x3 곱하기
// R
S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값

for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImageR[i + m][j + n] * mask[m][n];

tmpOutImageR[i][j] = S;
```

영역 처리 - 엠보싱(HSI)

Input

Output



[엠보싱 > HSI]

RGB 모델을 HSI 모델로 변환 후,
양각 형태로 변환

```
//////// RGB 모델 --> HSI 모델 ////////////
for (int i = 0; i < m_inH; i++) {
    for (int j = 0; j < m_inW; j++) {
        double* hsi;
        unsigned char R, G, B;

        R = tmpInImageR[i][j]; G = tmpInImageG[i][j]; B = tmpInImageB[i][j];
        hsi = RGB2HSI(R, G, B);

        double H, S, I;

        H = hsi[0]; S = hsi[1]; I = hsi[2];
        tmpInImageH[i][j] = H; tmpInImageS[i][j] = S; tmpInImageI[i][j] = I;
    }
}
```

```
double S_VALUE = 0.0;

for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S_VALUE += tmpInImageI[i + m][j + n] * mask[m][n];

tmpInImageI[i][j] = S_VALUE;
```

```
////// HSI --> RGB ////////
for (int i = 0; i < m_inH; i++)
    for (int j = 0; j < m_inW; j++) {
        unsigned char* rgb;
        double H, S, I;

        H = tmpInImageH[i][j]; S = tmpInImageS[i][j]; I = tmpInImageI[i][j];

        rgb = HSI2RGB(H, S, I);
        tmpOutImageR[i][j] = rgb[0]; tmpOutImageG[i][j] = rgb[1]; tmpOutImageB[i][j] = rgb[2];
    }
```

영역 처리 - 블러링(3x3)

Input

Output

[블러링 > 3x3]

3x3 마스크로 영상의 세밀한 부분을 제거

```
double mask[MSIZE][MSIZE] = {
    { 1.0 / 9, 1.0 / 9, 1.0 / 9 },
    { 1.0 / 9, 1.0 / 9, 1.0 / 9 },
    { 1.0 / 9, 1.0 / 9, 1.0 / 9 } }; // 블러링(3x3) 마스크
```

```
// 마스크(3x3)와 한 점을 중심으로 한 3x3 곱하기
// R
S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값

for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImageR[i + m][j + n] * mask[m][n];

tmpOutImageR[i][j] = S;
```

영역 처리 - 블러링(9x9)

Input

Output

[블러링 > 9x9]

9x9 마스크로 영상의 세밀한 부분을 제거

```
double mask[MSIZE][MSIZE]; // 블러링(9x9) 마스크

for (int i = 0; i < MSIZE; i++)
    for (int j = 0; j < MSIZE; j++)
        mask[i][j] = 1.0 / 81;
```

```
// 마스크(9x9)와 한 점을 중심으로 한 9x9 곱하기
// R
S = 0.0; // 마스크 81개와 입력값 81개를 각각 곱해서 합한 값

for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImageR[i + m][j + n] * mask[m][n];

tmpOutImageR[i][j] = S;
```

영역 처리 - 샤프닝

Input

Output

[샤프닝 > 기본]

고주파에 해당하는 상세한 부분을
더욱 강조 -> 대비 효과를 증가

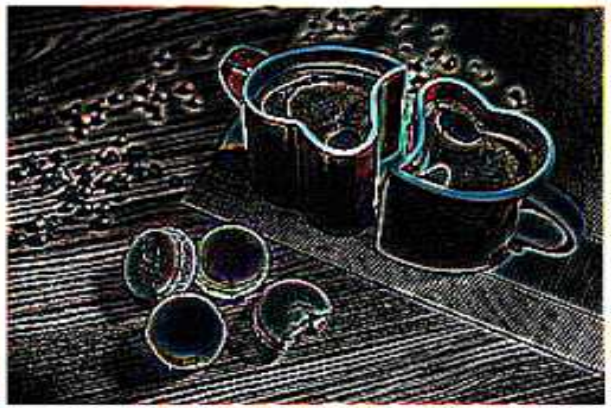
```
double mask[MSIZE][MSIZE] = {
    { 0.0, -1.0, 0.0 },
    { -1.0, 5.0, -1.0 },
    { 0.0, -1.0, 0.0 } }; // 샤프닝 마스크
```

```
// 마스크(3x3)와 한 점을 중심으로 한 3x3 곱하기
// R
S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값

for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImageR[i + m][j + n] * mask[m][n];

tmpOutImageR[i][j] = S;
```

영역 처리 - 샤프닝(고주파)

Input

Output



[샤프닝 > 고주파]

고주파 통과 필터를 적용
-> 고주파 성분 강조

```
double mask[MSIZE][MSIZE] = {
    { -1.0 / 9, -1.0 / 9, -1.0 / 9 },
    { -1.0 / 9, 8.0 / 9, -1.0 / 9 },
    { -1.0 / 9, -1.0 / 9, -1.0 / 9 } }; // 샤프닝(고주파) 마스크
```

```
// 마스크(3x3)와 한 점을 중심으로 한 3x3 곱하기
// R
S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값

for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImageR[i + m][j + n] * mask[m][n] * 20;

tmpOutImageR[i][j] = S;
```

영역 처리 - 샤프닝(저주파)

Input

Output(2.0)

[샤프닝 > 저주파]

소수(0.0 ~)를 입력 시,
(원 영상) – (저주파 통과 필터링 결과 영상)

```
double mask[MSIZE][MSIZE] = {
    { 1.0 / 9, 1.0 / 9, 1.0 / 9 },
    { 1.0 / 9, 1.0 / 9, 1.0 / 9 },
    { 1.0 / 9, 1.0 / 9, 1.0 / 9 } }; // 샤프닝(저주파) 마스크
```

```
// 마스크(3x3)와 한 점을 중심으로 한 3x3 곱하기
// R
S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값

for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImageR[i + m][j + n] * mask[m][n];

tmpOutImageR[i][j] = S;
```

```
// R
unsharp = alpha * m_inImageR[i][j] - tmpOutImageR[i][j];
```

영역 처리 - 가우시안

Input

Output

[영역 처리 - 가우시안]

고주파 성분을 제거해 영상을 부드럽게 만듦

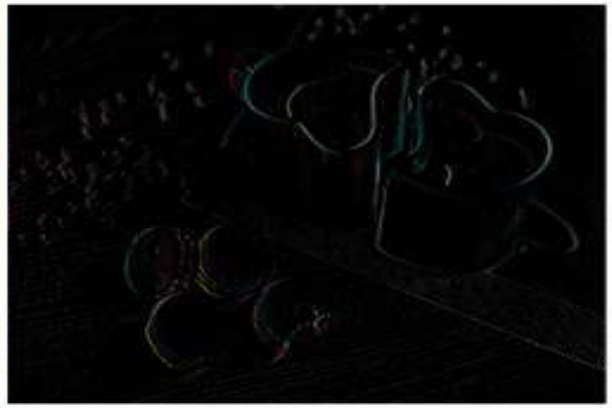
```
double mask[MSIZE][MSIZE] = {
    { 1.0 / 16, 1.0 / 8, 1.0 / 16 },
    { 1.0 / 8, 1.0 / 4, 1.0 / 8 },
    { 1.0 / 16, 1.0 / 8, 1.0 / 16 } }; // 가우시안 마스크
```

```
// 마스크(3x3)와 한 점을 중심으로 한 3x3 곱하기
// R
S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값

for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImageR[i + m][j + n] * mask[m][n];

tmpOutImageR[i][j] = S;
```

영역 처리 - 경계선(수직)

Input

Output



[경계선 > 이동과 차분 > 수직]

수직 경계선 검출

(*edge(경계선) : 영상에서 밝기가 급격하게 변하는 부분, 영상을 구성하는 객체 간의 경계)

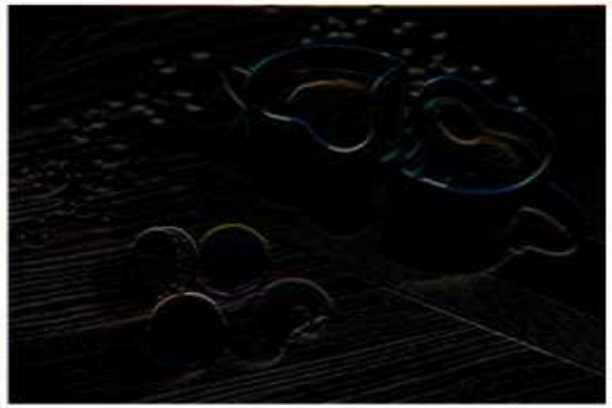
```
double mask[MSIZE][MSIZE] = {
    { 0.0, 0.0, 0.0 },
    { -1.0, 1.0, 0.0 },
    { 0.0, 0.0, 0.0 } }; // 수직 에지 검출 마스크
```

```
// 마스크(3x3)와 한 점을 중심으로 한 3x3 곱하기
// R
S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값

for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImageR[i + m][j + n] * mask[m][n];

tmpOutImageR[i][j] = S;
```

영역 처리 - 경계선(수평)

Input

Output



[경계선 > 이동과 차분 > 수평]

수평 경계선 검출

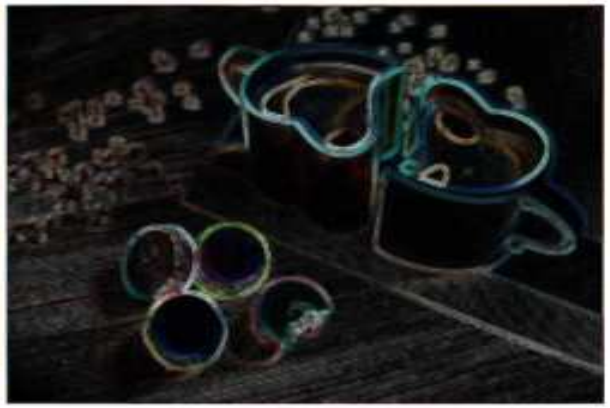
```
double mask[MSIZE][MSIZE] = {
    { 0.0, -1.0, 0.0 },
    { 0.0, 1.0, 0.0 },
    { 0.0, 0.0, 0.0 } }; // 수평 에지 검출 마스크
```

```
// 마스크(3x3)와 한 점을 중심으로 한 3x3 곱하기
// R
S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값

for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImageR[i + m][j + n] * mask[m][n];

tmpOutImageR[i][j] = S;
```

영역 처리 - 경계선(유사 연산자)

Input

Output



[경계선 > 연산자 > 유사 연산자]

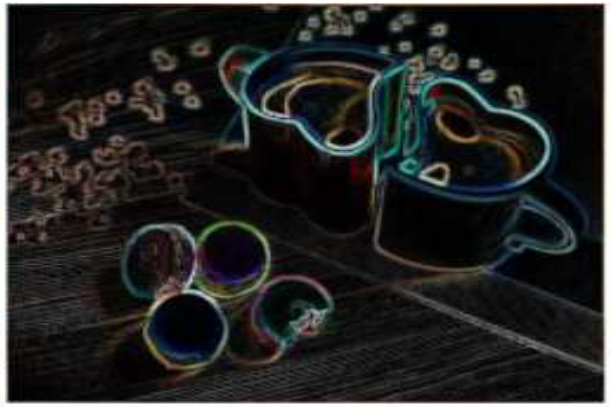
화소를 감산한 값에서
최대값을 결정하여 경계선 검출

```
// R
S = 0.0;

for (int m = 0; m < MSIZE; m++) {
    for (int n = 0; n < MSIZE; n++) {
        if (S < fabs(tmpInImageR[i + 1][j + 1] - tmpInImageR[i + m][j + n]))
            S = fabs(tmpInImageR[i + 1][j + 1] - tmpInImageR[i + m][j + n]);
    }
}

tmpOutImageR[i][j] = S;
```

영역 처리 - 경계선(차 연산자)

Input

Output



[경계선 > 연산자 > 차 연산자]

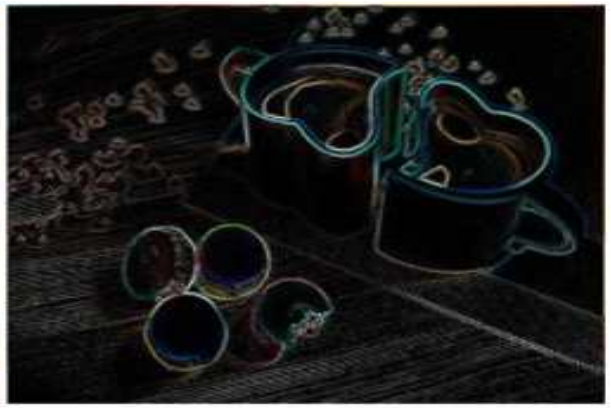
중앙을 기준으로 마주보는 값끼리 감산한 값에서 최대값을 결정하여 경계선 검출
(*뺄셈 연산이 유사 연산자와는 달리 화소당 네 번만 사용되어 빠르게 경계선 검출)

```
// R
S = 0.0;

for (int n = 0; n < MSIZE; n++) {
    if (S < fabs(tmpInImageR[i][j + n] - tmpInImageR[i + 2][j - n + 2]))
        S = fabs(tmpInImageR[i][j + n] - tmpInImageR[i + 2][j - n + 2]);
}

if (S < fabs(tmpInImageR[i + 1][j + 2] - tmpInImageR[i + 1][j]))
    S = fabs(tmpInImageR[i + 1][j + 2] - tmpInImageR[i + 1][j]);

tmpOutImageR[i][j] = S;
```

영역 처리 - 경계선(로버츠)

Input

Output



[경계선 > 1차 미분 > 로버츠]

로버츠 마스크를 적용해 경계선 검출
(*크기가 작아 매우 빠른 속도로 동작하나,
돌출된 값을 잘 평균할 수 없으며, 잡음에 민감함)

```
double maskRow[MSIZE][MSIZE] = {
    { -1.0, 0.0, 0.0 },
    { 0.0, 1.0, 0.0 },
    { 0.0, 0.0, 0.0 } }; // 로버츠 행 에지 검출 마스크

double maskCol[MSIZE][MSIZE] = {
    { 0.0, 0.0, -1.0 },
    { 0.0, 1.0, 0.0 },
    { 0.0, 0.0, 0.0 } }; // 로버츠 열 에지 검출 마스크
```

```
// R
S1 = tmpRowOutImageR[i][j];
S2 = tmpColOutImageR[i][j];

if (sqrt((double)(S1 * S1 + S2 * S2)) < 0.0)
    m_outImageR[i][j] = 0;
else if (255.0 < sqrt((double)(S1 * S1 + S2 * S2)))
    m_outImageR[i][j] = 255;
else
    m_outImageR[i][j] = (unsigned char)sqrt((double)(S1 * S1 + S2 * S2));
```

영역 처리 - 경계선(프리윗)

Input

Output

[경계선 > 1차 미분 > 프리윗]

프리윗 마스크를 적용해 경계선 검출
(*돌출된 값을 비교적 잘 평균화하나, 대각선보다 수평과 수직에 놓인 에지에 더 민감하게 반응함)

```
double maskRow[MSIZE][MSIZE] = {
    { -1.0, -1.0, -1.0 },
    { 0.0, 0.0, 0.0 },
    { 1.0, 1.0, 1.0 } }; // 프리윗 행 에지 검출 마스크

double maskCol[MSIZE][MSIZE] = {
    { 1.0, 0.0, -1.0 },
    { 1.0, 0.0, -1.0 },
    { 1.0, 0.0, -1.0 } }; // 프리윗 열 에지 검출 마스크
```

```
// R
S1 = tmpRowOutImageR[i][j];
S2 = tmpColOutImageR[i][j];

if (sqrt((double)(S1 * S1 + S2 * S2)) < 0.0)
    m_outImageR[i][j] = 0;
else if (255.0 < sqrt((double)(S1 * S1 + S2 * S2)))
    m_outImageR[i][j] = 255;
else
    m_outImageR[i][j] = (unsigned char)sqrt((double)(S1 * S1 + S2 * S2));
```

영역 처리 - 경계선(소벨)

Input

Output

[경계선 > 1차 미분 > 소벨]

소벨 마스크를 적용해 경계선 검출
(*돌출된 값을 비교적 잘 평균화하나,
대각선 방향에 놓인 에지에 더 민감하게 반응함)

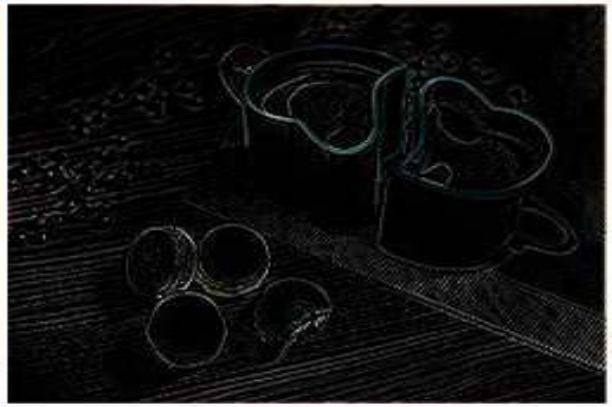
```
double maskRow[MSIZE][MSIZE] = {
    { -1.0, -2.0, -1.0 },
    { 0.0, 0.0, 0.0 },
    { 1.0, 2.0, 1.0 } }; // 소벨 행 에지 검출 마스크

double maskCol[MSIZE][MSIZE] = {
    { 1.0, 0.0, -1.0 },
    { 2.0, 0.0, -2.0 },
    { 1.0, 0.0, -1.0 } }; // 소벨 열 에지 검출 마스크
```

```
// R
S1 = tmpRowOutImageR[i][j];
S2 = tmpColOutImageR[i][j];

if (sqrt((double)(S1 * S1 + S2 * S2)) < 0.0)
    m_outImageR[i][j] = 0;
else if (255.0 < sqrt((double)(S1 * S1 + S2 * S2)))
    m_outImageR[i][j] = 255;
else
    m_outImageR[i][j] = (unsigned char)sqrt((double)(S1 * S1 + S2 * S2));
```

영역 처리 - 경계선(라플라시안)

Input

Output



[경계선 > 2차 미분 > 라플라시안]

라플라시안 마스크를 적용해 경계선 검출
(*모든 방향의 에지를 강조하나, 잡음 성분에
매우 민감하여 실제보다 더 많은 에지를 검출함)

```
double mask[MSIZE][MSIZE] = {
    { 0.0, -1.0, 0.0 },
    { -1.0, 4.0, -1.0 },
    { 0.0, -1.0, 0.0 } }; // 라플라시안 에지 검출 마스크
```

```
// 마스크(3x3)와 한 점을 중심으로 한 3x3 곱하기
// R
S = 0.0; // 마스크 9개와 입력값 9개를 각각 곱해서 합한 값

for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImageR[i + m][j + n] * mask[m][n];

tmpOutImageR[i][j] = S;
```

영역 처리 - 경계선(LoG)

Input

Output

[경계선 > 2차 미분 > LoG]

LoG 마스크를 적용해 경계선 검출
(*가우시안 스무딩으로 잡음 제거한 뒤,
에지를 강조하기 위해 라플라시안을 이용)

```
double mask[MSIZE][MSIZE] = {
    { 0.0, 0.0, -1.0, 0.0, 0.0 },
    { 0.0, -1.0, -2.0, -1.0, 0.0 },
    { -1.0, -2.0, 16.0, -2.0, -1.0 },
    { 0.0, -1.0, -2.0, -1.0, 0.0 },
    { 0.0, 0.0, -1.0, 0.0, 0.0 } }; // LoG 에지 검출 마스크
```

```
// 마스크(6x6)와 한 점을 중심으로 한 6x6 곱하기
// R
S = 0.0; // 마스크 25개와 입력값 25개를 각각 곱해서 합한 값

for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImageR[i + m][j + n] * mask[m][n];

tmpOutImageR[i][j] = S;
```

영역 처리 - 경계선(DoG)

Input

Output

[경계선 > 2차 미분 > DoG]

DoG 마스크를 적용해 경계선 검출
(*각 가우시안 연산에 분산 값을 서로 다르게 줘
이 차를 이용해 에지 맵을 구함
-> 계산 시간이 느린 LoG 연산자의 단점을 보안)

```
double mask[MSIZE][MSIZE] = {
    { 0.0, 0.0, -1.0, -1.0, -1.0, 0.0, 0.0 },
    { 0.0, -2.0, -3.0, -3.0, -3.0, -2.0, 0.0 },
    { -1.0, -3.0, 5.0, 5.0, 5.0, -3.0, -1.0 },
    { -1.0, -3.0, 5.0, 16.0, 5.0, -3.0, -1.0 },
    { -1.0, -3.0, 5.0, 5.0, 5.0, -3.0, -1.0 },
    { 0.0, -2.0, -3.0, -3.0, -3.0, -2.0, 0.0 },
    { 0.0, 0.0, -1.0, -1.0, -1.0, 0.0, 0.0 } }; // DoG 에지 검출 마스크
```

```
// 마스크(7x7)와 한 점을 중심으로 한 7x7 곱하기
// R
S = 0.0; // 마스크 49개와 입력값 49개를 각각 곱해서 합한 값

for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImageR[i + m][j + n] * mask[m][n];

tmpOutImageR[i][j] = S;
```

컬러 이미지 효과 - 채도 변경

Input

Output(-2.0)

[컬러 이미지 효과 – 채도 변경]

RGB 모델을 HSI 모델로 변환 후,
S(0.0 ~ 1.0) 값을 바꿔 채도 변경

```
// RGB --> HSI
double H, S, I;
unsigned char R, G, B;

R = m_inImageR[i][j];
G = m_inImageG[i][j];
B = m_inImageB[i][j];

double* hsi = RGB2HSI(R, G, B);
H = hsi[0]; S = hsi[1]; I = hsi[2];

// 채도(S) 조리개
S -= 0.2;
if (S < 0)
    S = 0.0;

// HSI --> RGB
unsigned char* rgb = HSI2RGB(H, S, I);
R = rgb[0], G = rgb[1], B = rgb[2];

m_outImageR[i][j] = R;
m_outImageG[i][j] = G;
m_outImageB[i][j] = B;
```

컬러 이미지 효과 - 색 추출

Input

Output

[컬러 이미지 효과 – 색 추출]

RGB 모델을 HSI 모델로 변환 후,
지정한 H(0 ~ 360) 값과 같은 화소만 추출

```
if (checkRed) {
    if ((0 <= H && H <= colorStart) || colorEnd <= H && H <= 360) {
        m_outImageR[i][j] = m_inImageR[i][j];
        m_outImageG[i][j] = m_inImageG[i][j];
        m_outImageB[i][j] = m_inImageB[i][j];
    }
    else {
        double avg = (m_inImageR[i][j] + m_inImageG[i][j] + m_inImageB[i][j]) / 3.0;
        m_outImageR[i][j] = m_outImageG[i][j] = m_outImageB[i][j] = (unsigned char)avg;
    }
}
else {
    // 오렌지 추출 (H : 8 - 20)
    if (colorStart <= H && H <= colorEnd) {
        m_outImageR[i][j] = m_inImageR[i][j];
        m_outImageG[i][j] = m_inImageG[i][j];
        m_outImageB[i][j] = m_inImageB[i][j];
    }
    else {
        double avg = (m_inImageR[i][j] + m_inImageG[i][j] + m_inImageB[i][j]) / 3.0;
        m_outImageR[i][j] = m_outImageG[i][j] = m_outImageB[i][j] = (unsigned char)avg;
    }
}
```

정리

느낀점

직접 영상 처리 프로그램을 C 언어, 파이썬, C++로 각각 구현해 봄으로써 한 가지 언어로만 프로젝트를 진행했을 때보다 구현한 기능들에 대해 더 잘 알고, 이해할 수 있었던 것 같다.

향후 발전 방향

- 더 다양한 영상 처리 기능 추가
- 마스크 크기 입력 받아 적용
- 버그 개선