[Intel] 엣지 AI SW 아카데미

객체지향 프로그래밍

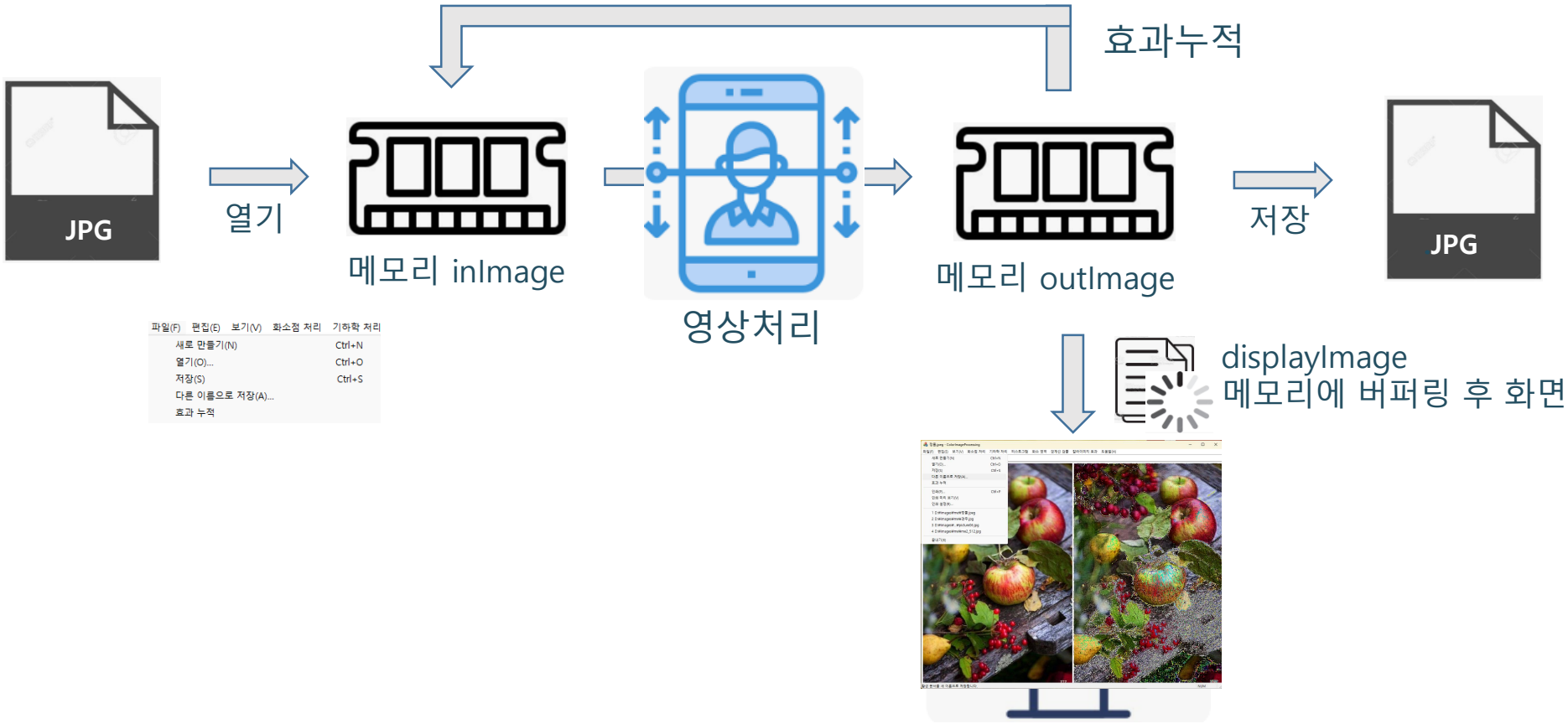
OpenCV 없이 MFC로 구현한 C++ Image Processing

이지원

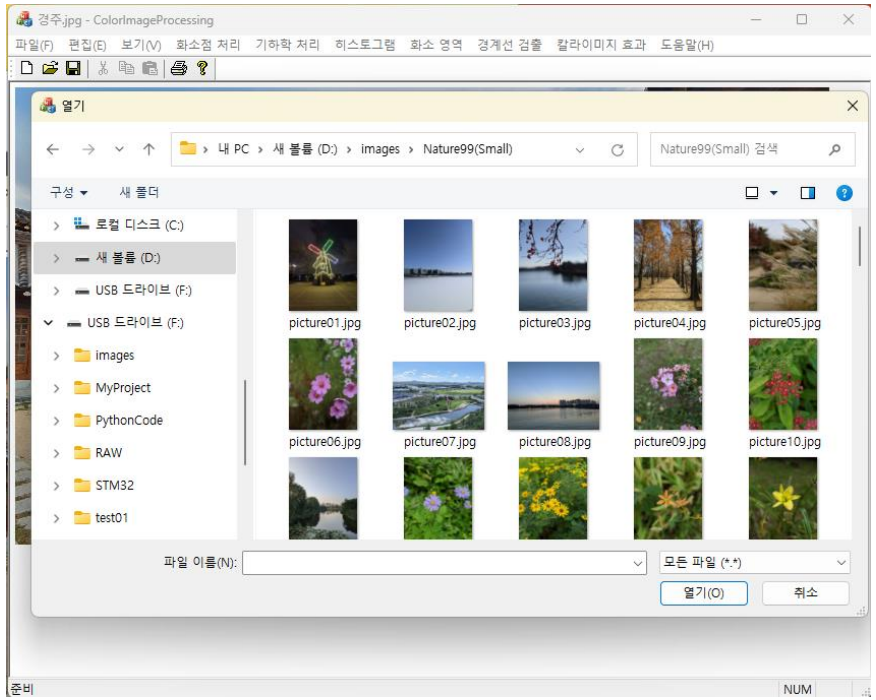
프로젝트 개요

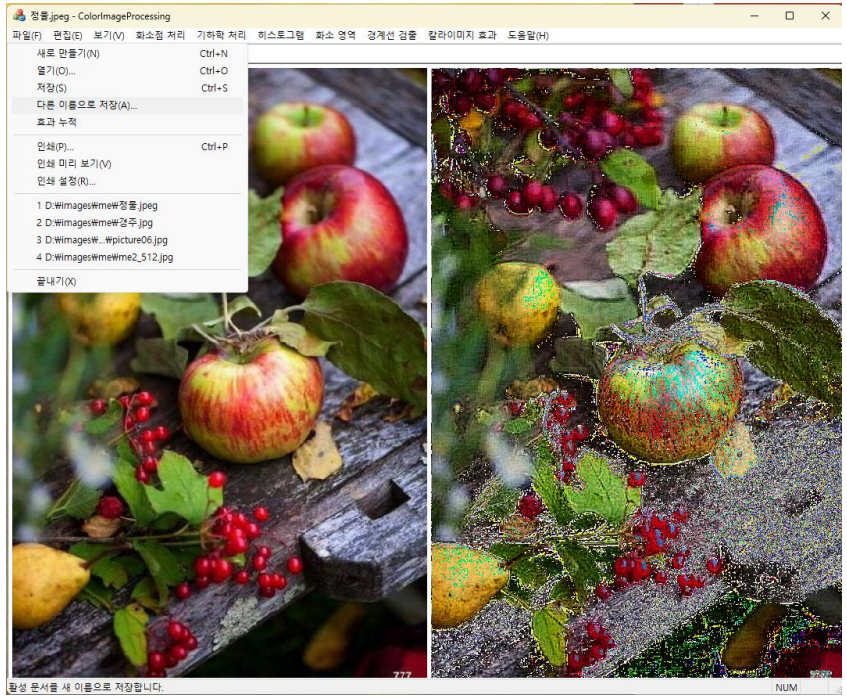
- 구현 목적
  - ✓ OpenCV 없이 MFC로 구현한 C++ Image Processing
  - ✓ 개발 기간: 2024.3.27~4.2 (C, Python 버전 포팅 및 기능 추가)
- 개발 환경
  - ✓ OS : Windows 11
  - ✓ Tool : Visual Studio Community 2002
  - ✓ Language : C++ (MFC)
- 주요 기능
  - ✓ 화소점 처리, 기하학 처리, 히스토그램 처리
  - ✓ 칼라 이미지 효과
  - ✓ 화소 영역 처리, 경계선 검출

프로그램 구조


효과누적
JPG
열기
메모리 inImage
영상처리
메모리 outImage
저장
JPG
displayImage
메모리에 버퍼링 후 화면에 draw

파일 ▶ 열기, 저장, 효과 누적


경주.jpg - ColorImageProcessing
파일(F) 편집(E) 보기(V) 화소점 처리 기하학 처리 히스토그램 화소 영역 경계선 검출 칼라이미지 효과 도움말(H)
열기
내 PC › 새 볼륨 (D:) › images › Nature99(Small)
Nature99(Small) 검색
구성
새 폴더
로컬 디스크 (C:)
새 볼륨 (D:)
USB 드라이브 (F:)
USB 드라이브 (F:)
images
MyProject
PythonCode
RAW
STM32
test01
picture01.jpg
picture02.jpg
picture03.jpg
picture04.jpg
picture05.jpg
picture06.jpg
picture07.jpg
picture08.jpg
picture09.jpg
picture10.jpg
파일 이름(N):
모든 파일 (*.*)
열기(O)
취소
준비
NUM


정물.jpeg - ColorImageProcessing
파일(F) 편집(E) 보기(V) 화소점 처리 기하학 처리 히스토그램 화소 영역 경계선 검출 칼라이미지 효과 도움말(H)
새로 만들기(N) Ctrl+N
열기(O)... Ctrl+O
저장(S) Ctrl+S
다른 이름으로 저장(A)...
효과 누적
인쇄(P)... Ctrl+P
인쇄 미리 보기(V)
인쇄 설정(R)...
1 D:₩images₩me₩정물.jpeg
2 D:₩images₩me₩경주.jpg
3 D:₩images₩...₩picture06.jpg
4 D:₩images₩me₩me2_512.jpg
끝내기(X)
환성 문서를 새 이름으로 저장합니다.
NUM

화소점 처리 ▶ 그레이스케일

원본 영상 결과 영상

✓화소점 처리란?

화소 점의 원래 값을 기준으로 화소 값을 변경하는 기술

0. 그레이스케일

→ RGB 값을 평균 내고 동일하게 함

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

화소점 처리 ▶ 산술연산

원본 영상

+60

밝게

−60

어둡게

✓화소점 처리란?

화소 점의 원래 값을 기준으로 화소 값을 변경하는 기술

1. 덧셈 연산

→ 이미지를 밝게

2. 뺄셈 연산

→ 이미지를 어둡게

```
if (m_inImageR[i][k] + value > 255)
    m_outImageR[i][k] = 255;
else if (m_inImageR[i][k] + value < 0)
    m_outImageR[i][k] = 0;
else
    m_outImageR[i][k] = m_inImageR[i][k] + value;
```

화소점 처리 ▶ 산술연산

원본 영상

X2

곱셈 연산

/2

나눗셈 연산

3. 곱셈 연산

→ 이미지를 밝게

```
px = int(inImage[i][k] * val)
```

4. 나눗셈 연산

→ 이미지를 어둡게

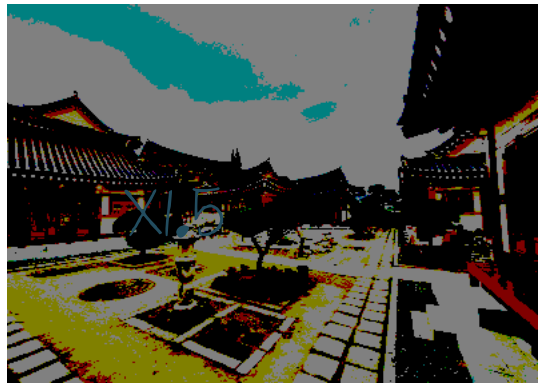
```
px = int(inImage[i][k] / val)
```

```
if (px > 255) :
    outImage[i][k] = 255
elif(px < 0) :
    outImage[i][k] = 0
else :
    outImage[i][k] = px
```

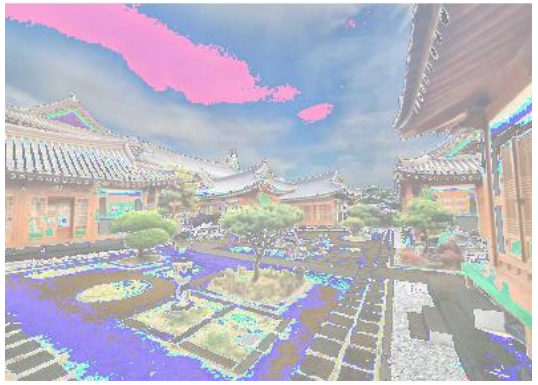
예외처리

화소점 처리 ▶ 논리연산

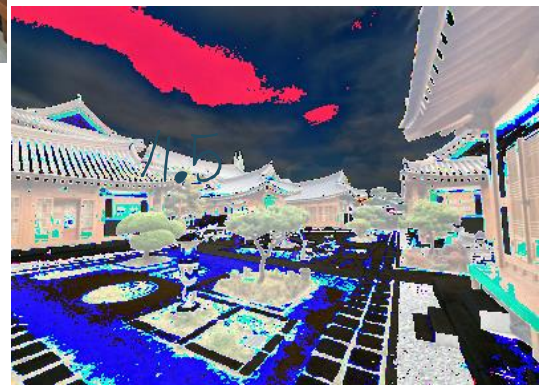
원본 영상



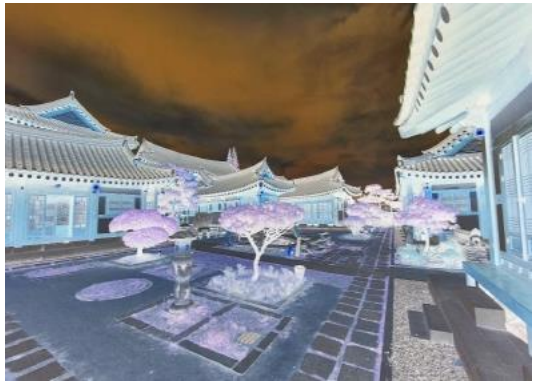
AND 128



OR 128



XOR 128



NOT (반전)

5. AND 연산

```
m_outImageR[i][k] = m_inImageR[i][k] & value;
```

6. OR 연산

```
m_outImageR[i][k] = m_inImageR[i][k] | value;
```

7. XOR 연산

→ 입력이 다를 때만 1로

```
m_outImageR[i][k] = m_inImageR[i][k] ^ value;
```

8. NOT 연산 (반전)

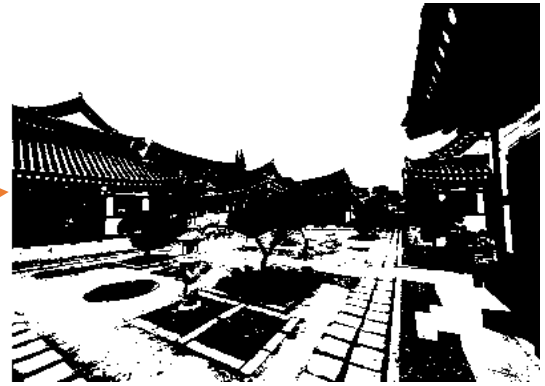
→ 검정색은 흰색으로, 흰색은 검정색으로

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

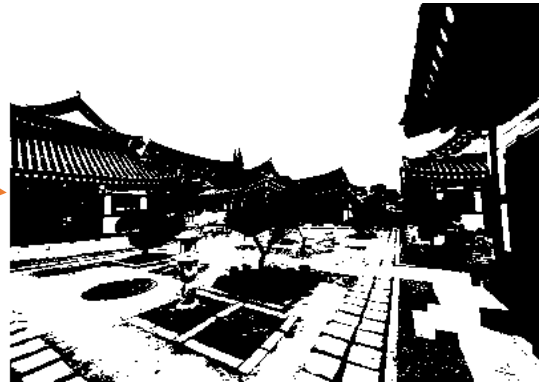
화소점 처리 ▶ 흑백 (이진화)

원본 영상

흑백



흑백



흑백(평균값, 126)

✓경계 값을 이용해 값이 두 개만 있는 영상으로 변환

$$\text{Output(q)} = \begin{cases} 255 & \text{Input(p)} \geq T \\ 0 & \text{Input(p)} < T \end{cases}$$

9. 흑백 → T = 128

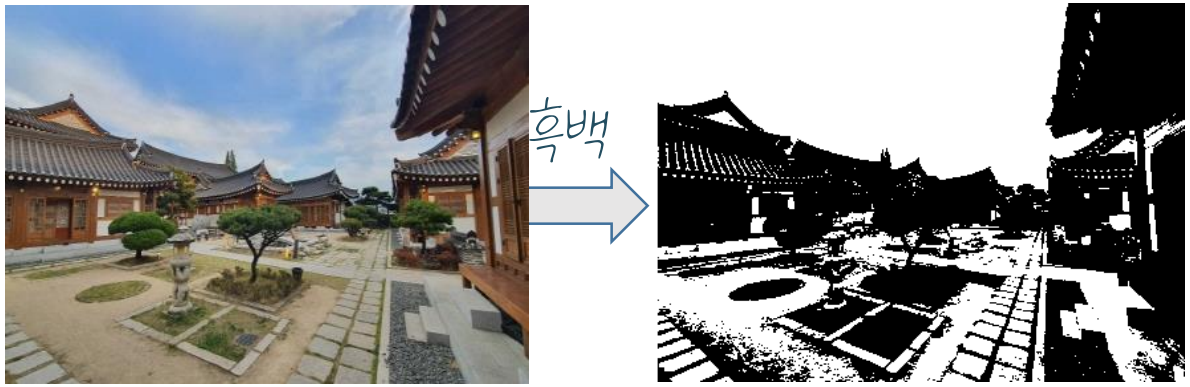
```
avg = (m_inImageR[i][k] + m_inImageG[i][k] + m_inImageB[i][k]) / 3.0;
if (avg < 128)
    m_outImageR[i][k] = m_outImageG[i][k] = m_outImageB[i][k] = 0;
else
    m_outImageR[i][k] = m_outImageG[i][k] = m_outImageB[i][k] = 255;
```

10. 흑백(평균값) → T= 평균값

```
avg = sum / (m_outW * m_outH * 3);

avgColor = (m_inImageR[i][k] + m_inImageG[i][k] + m_inImageB[i][k]) / 3.0;
if (avgColor < avg)
    m_outImageR[i][k] = m_outImageG[i][k] = m_outImageB[i][k] = 0;
else
    m_outImageR[i][k] = m_outImageG[i][k] = m_outImageB[i][k] = 255;
```

화소점 처리 ▶ 흑백 (이진화)



원본 영상 흑백(중앙값, 113)

11. 흑백(중앙값)

→ T= 중앙값

```
for (int i = 0; i < m_inH; i++) {
    for (int k = 0; k < m_inW; k++) {
        avg = (m_inImageR[i][k] + m_inImageG[i][k] + m_inImageB[i][k]) / 3.0;
        grayImage[i][k] = (unsigned char)avg;
        array[number] = (unsigned char)avg;
        number++;
    }
}

std::sort(array, array + number);
median = array[m_inH * m_inW / 2];

if (grayImage[i][k] < median)
    m_outImageR[i][k] = m_outImageG[i][k] = m_outImageB[i][k] = 0;
else
    m_outImageR[i][k] = m_outImageG[i][k] = m_outImageB[i][k] = 255;
```

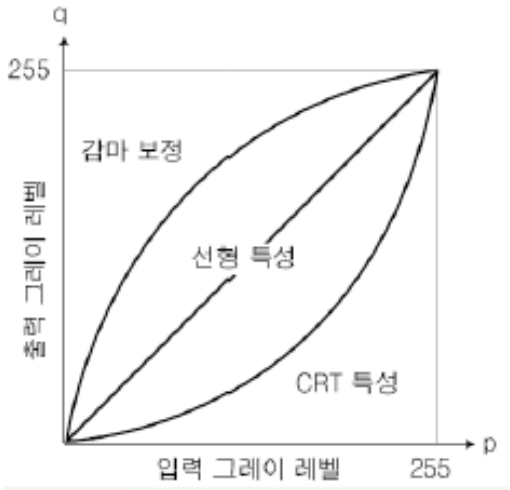
화소점 처리 ▶ 감마보정

원본 영상

감마 값 0.7로 보정

감마 값 1.3로 보정

(d) 감마 보정 변환 함수 그래프



12. 감마보정

- ✓ 인간의 시각이 비선형적이기 때문에 비선형 전달 함수를 거쳐 빛의 강도에 변화를 주는 과정
- ✓ 함수의 감마 값에 따라 밝기 조정
- ✓ 감마 값이 1보다 크면 영상이 어두워지고,

  1보다 작으면 영상이 밝아짐

```
temp = m_inImageR[i][k];
m_outImageR[i][k] = (unsigned char)(255.0 * pow(temp / 255.0, value));
```

감마값

화소점 처리 ▶ 파라볼라

원본 영상

파라볼라 CAP 보정

파라볼라 CUP 보정

✓ 영상에 입체감을 줌

13. 파라볼라 CAP

✓ 밝은 곳 입체형

```
temp = m_inImageR[i][k];
val = 255.0 - 255.0 * pow((temp / 128.0 - 1.0), 2);
```

14. 파라볼라 CUP

✓ 어두운 곳 입체형

```
temp = m_inImageR[i][k];
val = 255.0 * pow((temp / 128.0 - 1.0), 2);
```

화소점 처리 ▶ 포스터라이징

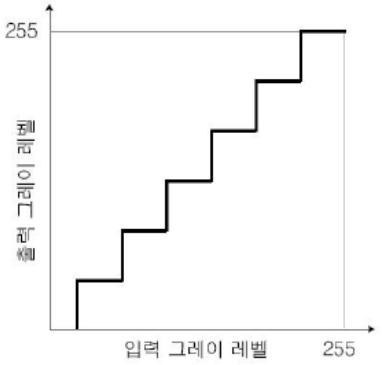
원본 영상 6단계로 변환

15. 포스터라이징

✓ 명암 값의 범위를 경계 값으로 축소

```
for (int i = 1; i < val - 1; i++) { // 영역 입력값에 비례하여 할당
    for (int j = 0; j < m_inH; j++) {
        for (int k = 0; k < m_inW; k++) {
            if (m_inImageR[j][k] > interval * i && m_inImageR[j][k] < interval * (i + 1))
                m_outImageR[j][k] = interval * (i + 1);
```

(c) 포스터라이징 변환 함수 그래프



화소점 처리 ▶ 범위 강조 변환

원본 영상

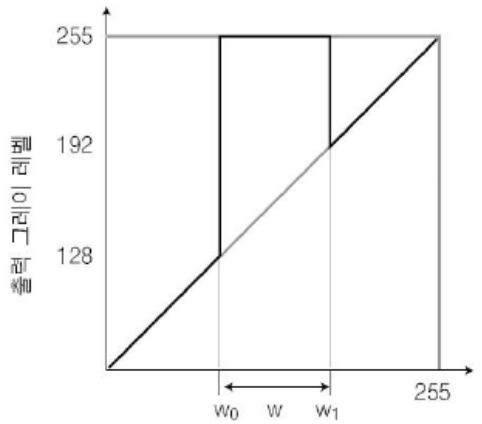
강조 변환
W0=100, W1=150

16. 범위 강조 변환

✓ 일정 범위의 화소만 강조하는 변환

```
if ((m_inImageR[i][k] >= start && m_inImageR[i][k] <= end) &&
    (m_inImageG[i][k] >= start && m_inImageG[i][k] <= end) &&
    (m_inImageB[i][k] >= start && m_inImageB[i][k] <= end)) {
    m_outImageR[i][k] = 255;
    m_outImageG[i][k] = 255;
    m_outImageB[i][k] = 255;
}
else {
    m_outImageR[i][k] = m_inImageR[i][k];
```

(c) 범위 강조 함수 그래프



기하학 처리

원본 영상

이동

좌우대칭

상하대칭

1. 이동

```
x = k - moveX;
y = i - moveY;
if (x >= 0 && x < m_inW && y >= 0 && y < m_inH)
    m_outImageR[i][k] = m_inImageR[y][x];
```

2. 좌우대칭

✓ 세로축을 중심으로 뒤집음

```
m_outImageR[i][k] = m_inImageR[i][m_outW - 1 - k];
```

3. 상하대칭

✓ 가로축을 중심으로 뒤집음

```
m_outImageR[i][k] = m_inImageR[m_inH - 1 - i][k];
```

기하학 처리 ▶ 확대

원본 영상

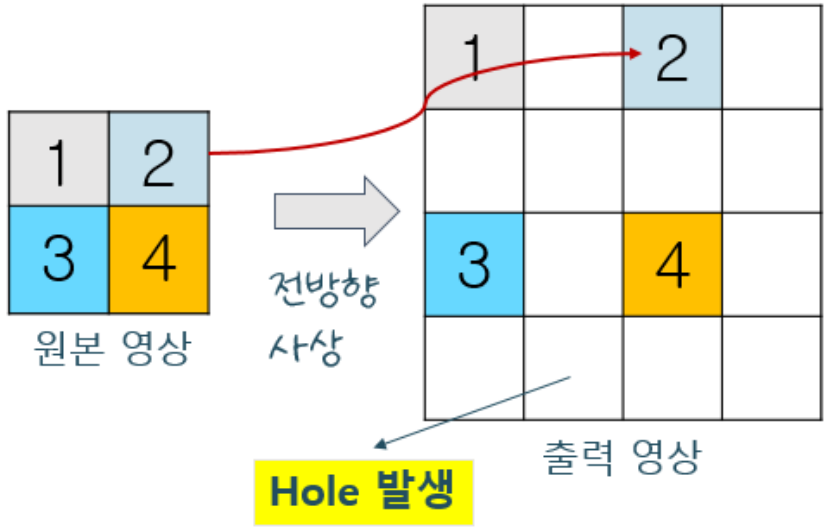
2배 확대, Hole 발생

```
m_outImageR[(int)(i * scale)][(int)(k * scale)] = m_inImageR[i][k];
m_outImageR[(int)(i * scale + 1)][(int)(k * scale + 1)] = m_inImageR[i][k];
```

4. 확대(포워딩)

✓ 전방향 사상

✓ Hole 문제: 임의의 화소가 목적 영상의 화소에 사상되지 않음



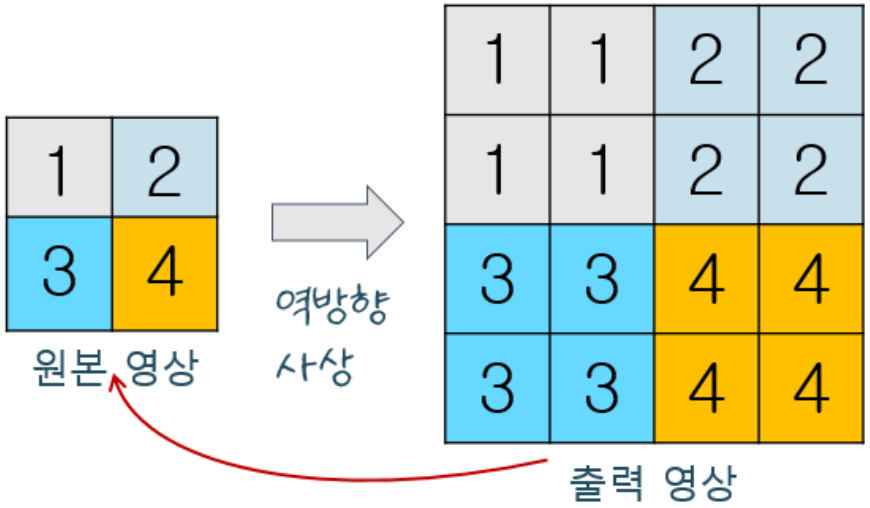
기하학 처리 ▶ 확대

원본 영상

2배 확대, Hole 발생 X

5. 확대(백워딩)

✓ 역방향 사상: 홀 문제가 일어나지 않음




```
for (int i = 0; i < m_outH; i++) {
    for (int k = 0; k < m_outW; k++) {
        m_outImageR[i][k] = m_inImageR[(int)(i / scale)][(int)k / scale];
```

기하학 처리 ▶ 확대

원본 영상

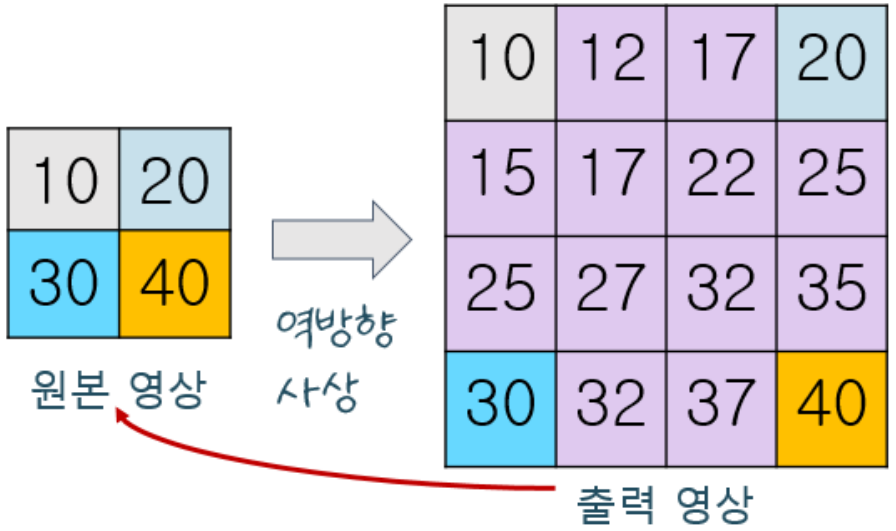
2배 확대, 양선형 보간

```
if (i_H < 0 || i_H >= (m_inH - 1) || i_W < 0 || i_W >= (m_inW - 1)) {
    m_outImageR[i][k] = 255;
else { // 소수점 값 보간하기
    C1 = (double)tempImageR[i_H][i_W];
    C2 = (double)tempImageR[i_H][i_W + 1];
    C3 = (double)tempImageR[i_H + 1][i_W + 1];
    C4 = (double)tempImageR[i_H + 1][i_W];

    newValue = (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);
    m_outImageR[i][k] = newValue;
```

6. 확대(양선형 보간법)

✓ 양선형 보간법: 화소당 선형 보간을 세 번하여, 가장 가까운 화소 네 개에 가중치를 곱한 값을 합해서 얻음



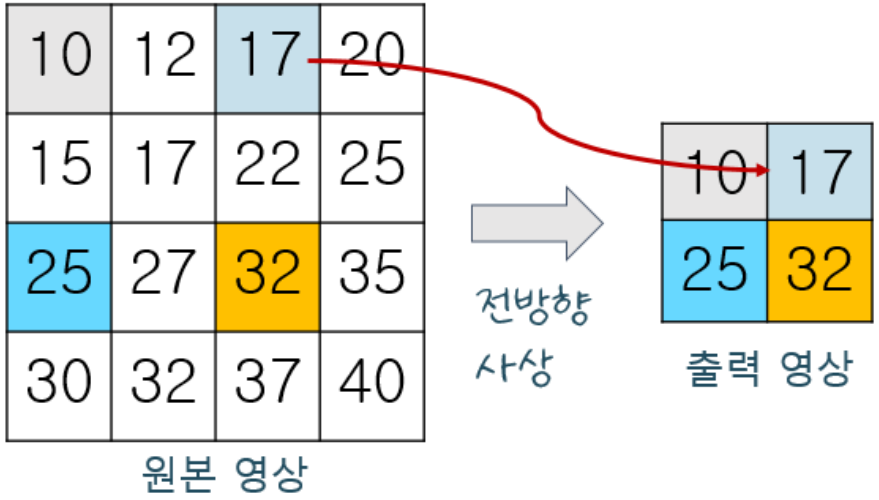
기하학 처리 ▶ 축소

원본 영상

2배 축소

7. 축소

✓ 에일리어싱: 영상의 크기를 많이 축소하면, 세부 내용을 상실하게 되는 현상




```
for (int i = 0; i < m_inH; i++) {
    for (int k = 0; k < m_inW; k++) {
        m_outImageR[(int)(i / scale)][(int)(k / scale)] = m_inImageR[i][k];
```

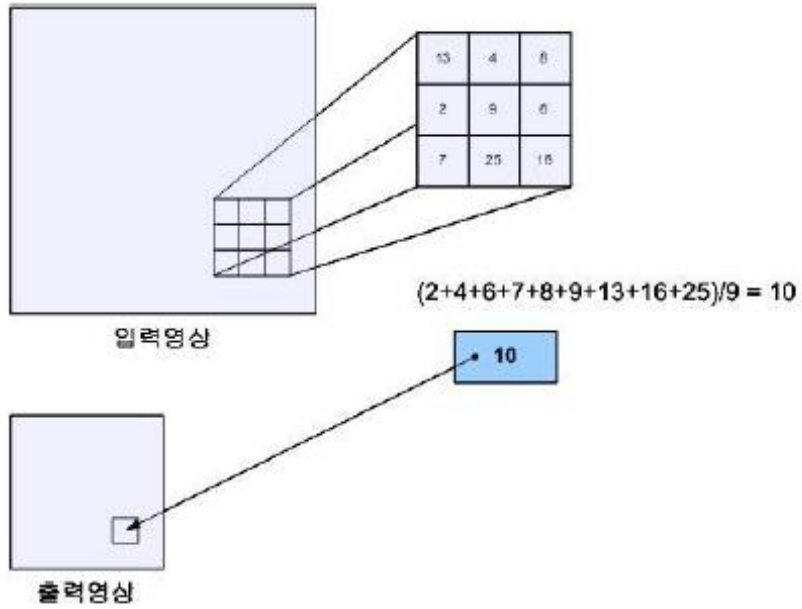
기하학 처리 ▶ 축소

원본 영상

2배 축소

```
for (int x = 0; x < scale; x++) {
    for (int y = 0; y < scale; y++) {
        sumR += m_inImageR[i + x][k + y];
avgR = (int)(sumR / (scale * scale));
m_outImageR[(int)(i / scale)][(int)(k / scale)] = avgR;
```

8. 축소 (평균값)



[그림 8-29] 평균 표현을 이용한 서브 샘플링의 동작

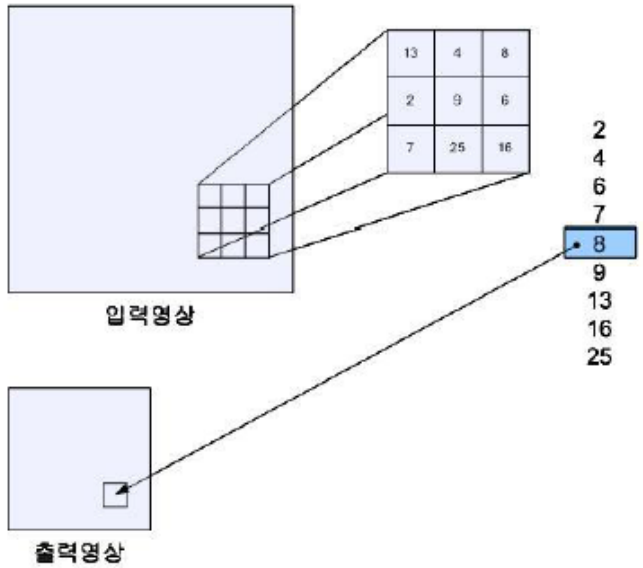
기하학 처리 ▶ 축소

원본 영상

2배 축소

```
number = 0;
for (int x = 0; x < scale; x++)
    for (int y = 0; y < scale; y++) {
        arrayR[number] = m_inImageR[i + x][k + y];
        number++;
std::sort(arrayR, arrayR + number);
medR = arrayR[(int)(scale * scale / 2)];
m_outImageR[(int)(i / scale)][(int)(k / scale)] = medR;
```

9. 축소 (중앙값)



[그림 8-28] 미디언 표현을 이용한 서브 샘플링 동작 과정

기하학 처리 ▶ 회전

원본 영상

45도 회전

전방향사상 시 홀 문제 발생
원점 기준 회전

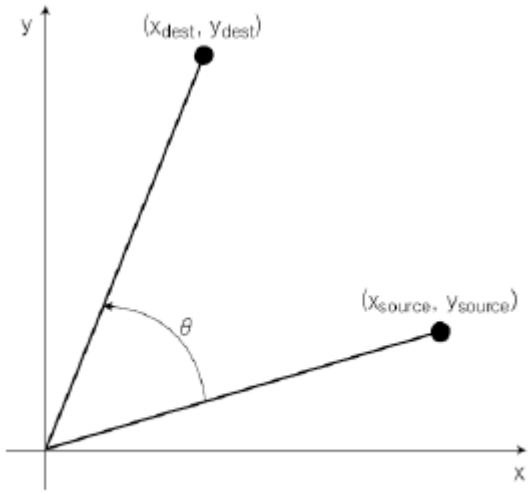
```
int xs = i;
int ys = k;
int xd = (int)(cos(radian) * xs - sin(radian) * ys);
int yd = (int)(sin(radian) * xs + cos(radian) * ys);

if ((0 <= xd && xd < m_outH) && (0 <= yd && yd < m_outW))
    m_outImageR[xd][yd] = m_inImageR[xs][ys];
```

10. 회전

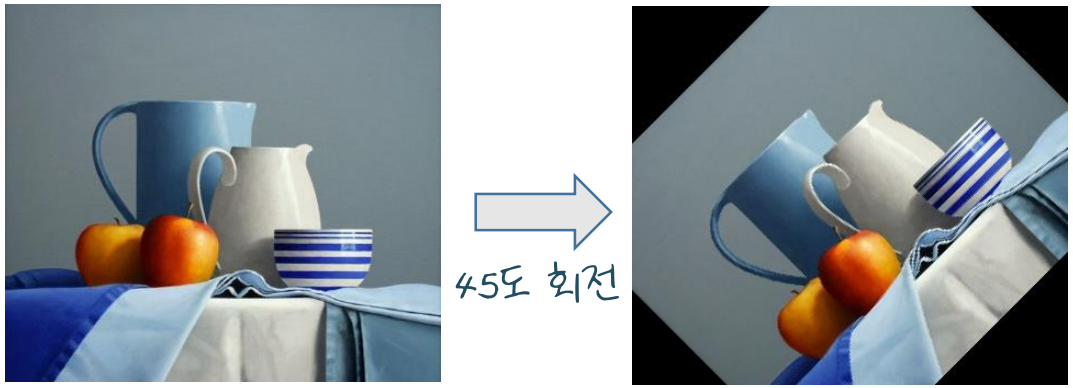
✓ 회전: 영상을 특정한 각도만큼 회전시키는 것

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



[그림 9-6] 회전 변환하여 좌표 변화

기하학 처리 ▶ 확대



원본 영상 중심점을 기준 회전

```
int cx = m_inH / 2;
int cy = m_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 < m_outH) && (0 <= ys && ys < m_outW)) {
    m_outImageR[xd][yd] = m_inImageR[xs][ys];
```

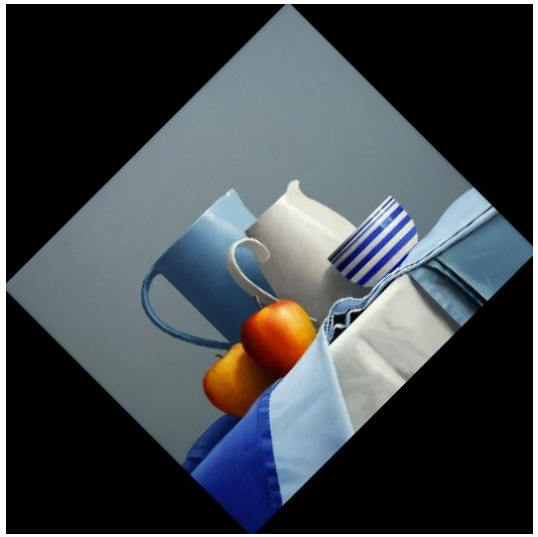
11. 회전 (중앙, 백워딩)

- ✓ 영상의 중심점을 기준으로 회전한 결과 보이는 부분이 줄어드는 것을 방지
- ✓ 역방향 사상: 홀 문제가 일어나지 않음
- ✓ 영상의 중심점이 (Cx, Cy)이고, 역방향 사상을 고려한 공식

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

기하학 처리 ▶ 확대

45도 회전



원본 영상

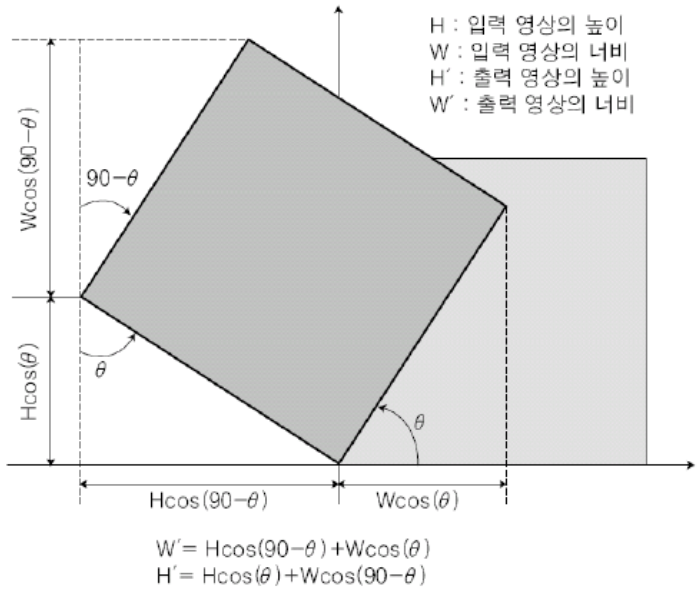
확대 회전

12. 회전 (확대)

✓ 영상의 크기를 고려한 회전 변환

✓ 잘려 나가는 부분이 없게 출력 영상 크기 계산

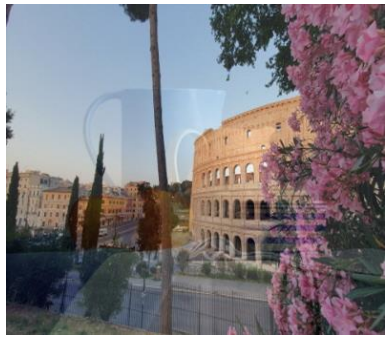
```
m_outW = (int)(m_inH * cos(radian2) + m_inW * cos(radian));
m_outH = (int)(m_inH * cos(radian) + m_inW * cos(radian2));
int centerX = m_outH / 2;
int centerY = m_outW / 2;
int xs = (int)(cos(radian) * (xd - centerX) + sin(radian) * (yd - centerY));
int ys = (int)(-sin(radian) * (xd - centerX) + cos(radian) * (yd - centerY));
if ((0 <= xs && xs < m_inH) && (0 <= ys && ys < m_inW)) {
    m_outImageR[xd][yd] = m_inImageR[xs][ys];
```



[그림 9-14] 회전 기하학적 변환의 출력 영상 크기

기하학 처리 ▶ 모핑



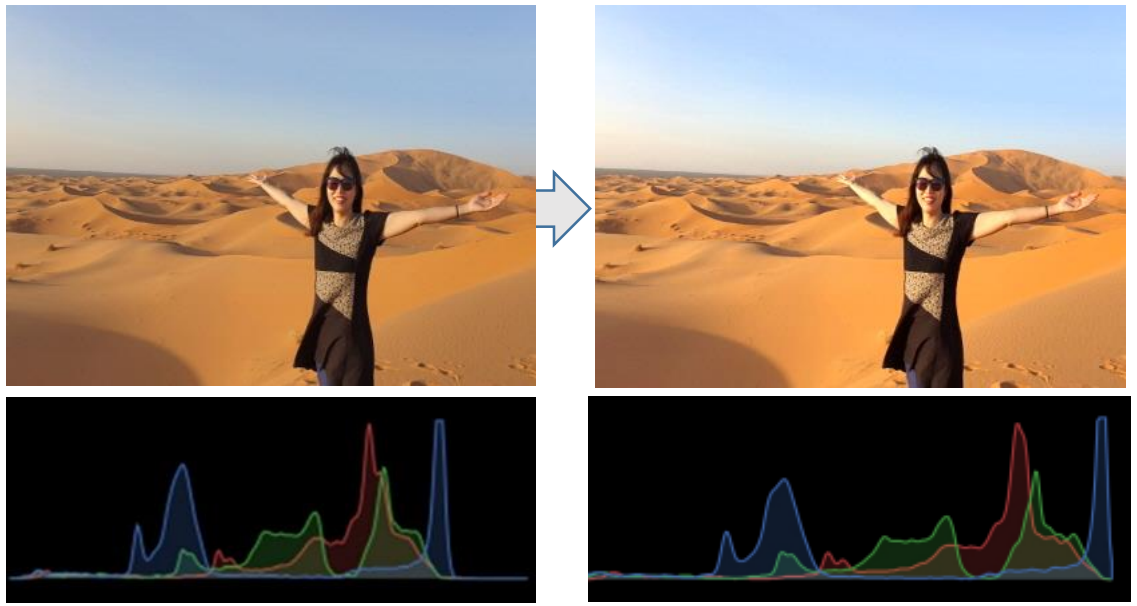

영상1 영상2

✓ 영상을 다른 영상으로 변환하는 기술

```
void CColorImageProcessingView::OnTimer(UINT_PTR nIDEvent)
    pDoc->OnMorph(m_nMorph);
    CRect r = { pDoc->m_inW + 10, 5, pDoc->m_inW + 10 + pDoc->m_outW, 5 + pDoc->m_outH };
    InvalidateRect(&r, TRUE); // OnDraw( ) 호출 효과

void CColorImageProcessingDoc::OnMorph(int morRate)
    m_outImageR[i][k] = (unsigned char)((m_inImageR[i][k] * (totalNum - morRate) / totalNum)
        + (m_morImageR[i][k] * (morRate) / totalNum));
```

히스토그램 처리 ▶ 스트레칭



원본 영상 변환 영상

✓ 히스토그램? 관측한 데이터가 분포된 특징을 한 눈에 볼 수 있도록 기둥 모양으로 나타낸 것

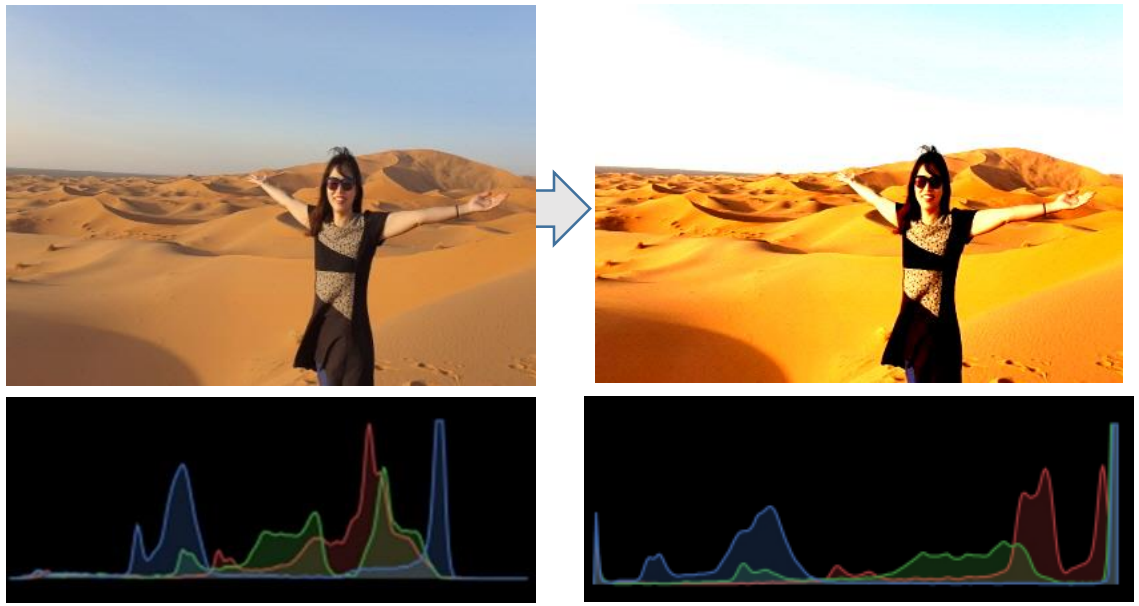
1. 히스토그램 스트레칭

✓ 명암 대비를 향상시키는 연산

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

- old pixel은 원 영상 화소의 명도 값
- new pixel은 결과 영상 화소의 명도 값
- low는 히스토그램의 최저 명도 값
- high는 히스토그램의 최고 명도 값

히스토그램 처리 ▶ 앤드-인



원본 영상 변환 영상

2. 히스토그램 앤드-인

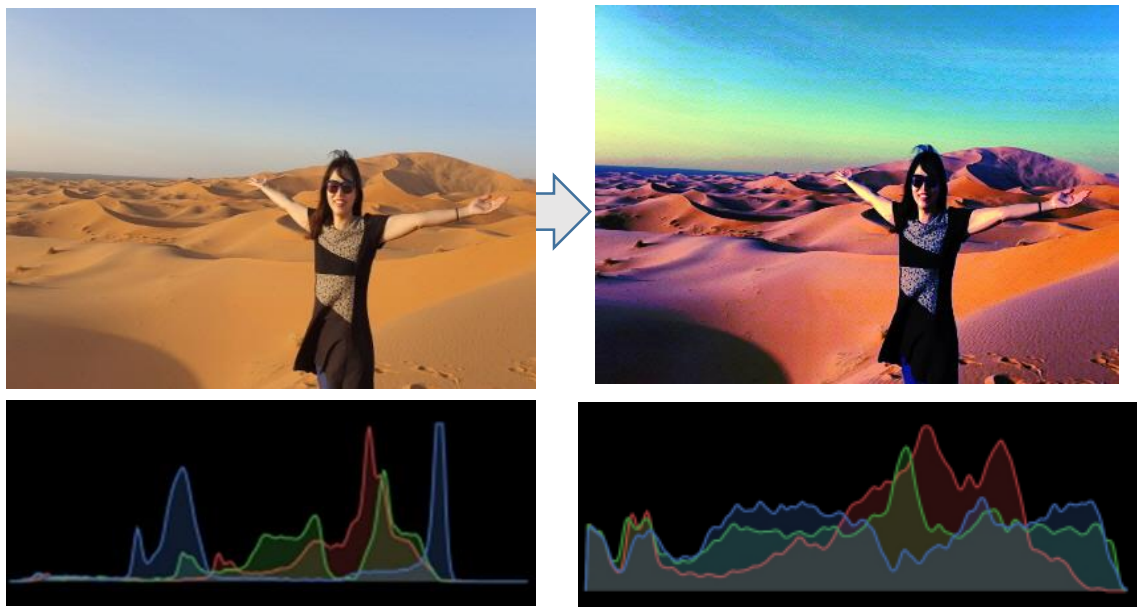
✓ 두 개의 임계 값을 사용하여 히스토스램의 분포를 좀더 균일하게 만듬

$$\text{new pixel} = \begin{cases} 0 & old\ pixel \le low \\ \dfrac{old\ pixel - low}{high - low} \times 255, & low \le old\ pixel \le high \\ 255 & \text{high} \le \text{old pixel} \end{cases}$$

```
high -= 50;
low += 50;
```

```
newVal = (int)((old - low) / (high - low) * 255.0);
if (newVal > 255)
    newVal = 255;
else if (newVal < 0)
    newVal = 0;
m_outImageR[i][k] = newVal;
```

히스토그램 처리 ▶ 히스토그램 평활화



원본 영상 변환 영상

3. 히스토그램 평활화

✓ 분포가 빈약한 영상을 균일하게 만듬

✓ 영상의 밝기 분포를 재분배하여 명함 대비 최대

✓ 1단계 : 히스토그램 빈도수 세기

```
histoR[m_inImageR[i][k]]++;
```

✓ 2단계 : 누적히스토그램 생성

```
sumHistoR[i] = sumHistoR[i - 1] + histoR[i];
```

✓ 3단계 : 정규화된 히스토그램 생성

```
normalHistoR[i] = sumHistoR[i] * (1.0 / (m_inH * m_inW)) * 255.0;
```

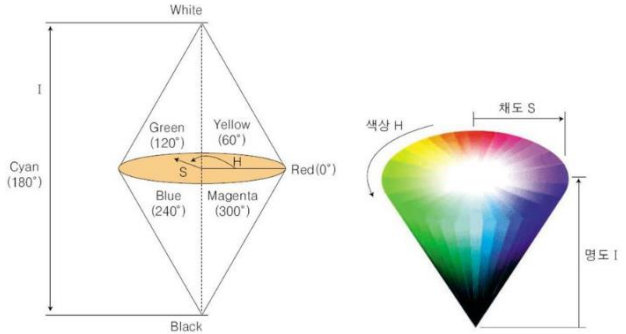
✓ 4단계 : 입력 이미지를 정규화된 값으로 치환

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

칼라 이미지 효과 ▶ 채도 변경

원본 영상

채도 -0.5



[그림 2-10] HSI 컬러 모델

✓HSI 컬러 모델?

H(색상), S(채도), I(명도) 3요소로 색을 분류하는 컬러 모델

1. 명도 변경

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

/// 채도(S) 변경
S = S + value;
if (S < 0)
    S = 0.0;
else if (S > 1.0)
    S = 1.0;
// HSI --> RGB
unsigned char* rgb = HSI2RGB(H, S, I);
R = rgb[0]; G = rgb[1]; B = rgb[2];
```

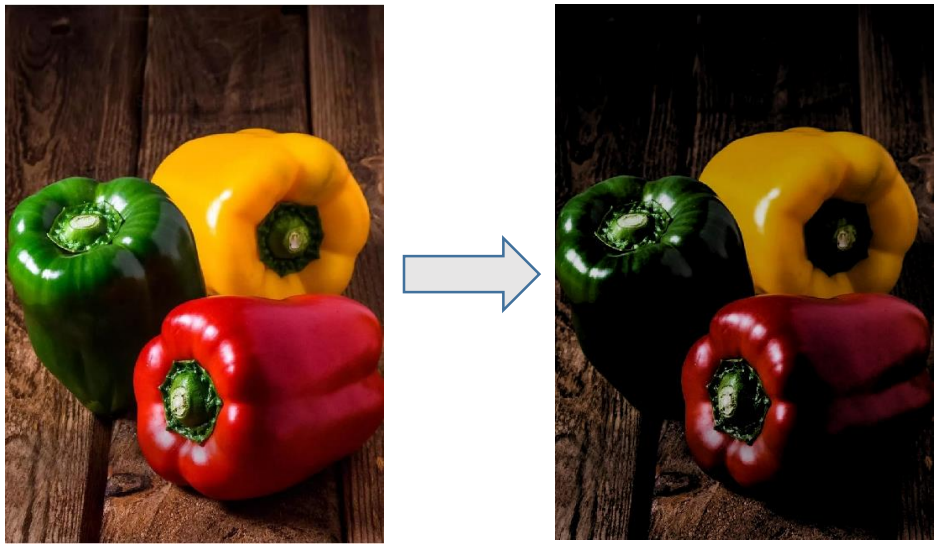
$$H = \cos^{-1}\left[\frac{0.5\cdot\{(R-G)+(R-B)\}}{\sqrt{(R-G)^2+(R-B)(G-B)}}\right]$$

$$S = 1 - \frac{3}{(R+G+B)}\cdot\min(R,G,B)$$

$$I = \frac{1}{3}(R+G+B)$$

```
// HIS 모델 값
// H(색상) : 0~360
// S(채도) : 0.0 ~ 1.0
// I(명도) : 0 ~ 255
```

칼라 이미지 효과 ▶ 명도 변경



원본 영상 명도 -60

2. 명도 변경

```
/// 채도(S) 변경
S = S + value;
if (S < 0)
    S = 0.0;
else if (S > 1.0)
    S = 1.0;
// HSI --> RGB
unsigned char* rgb = HSI2RGB(H, S, I);
R = rgb[0]; G = rgb[1]; B = rgb[2];
```

화소 영역 처리

✓원래 화소와 이웃한 각 화소의 가중치를 곱한 합을 출력 화소로 생성

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

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

I_1	I_2	I_3
I_4	I_5	I_6
I_7	I_8	I_9

(a) 입력 영상

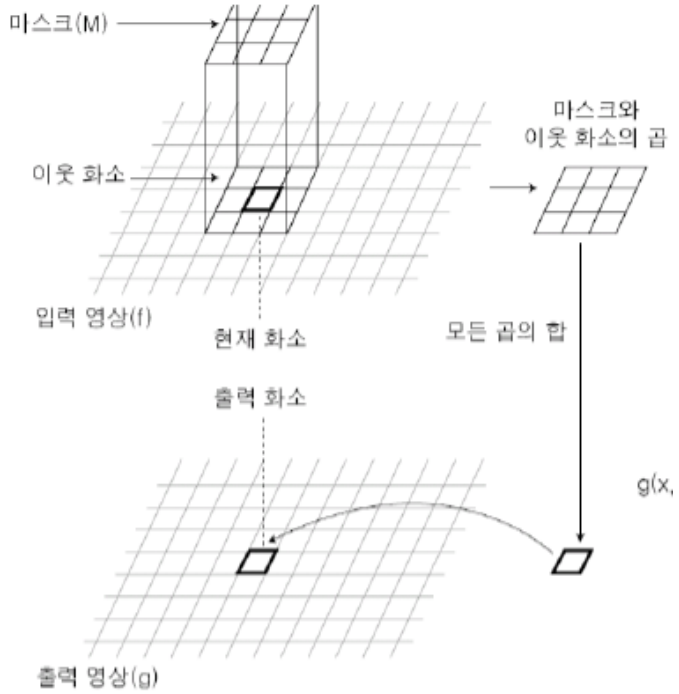
M_1	M_2	M_3
M_4	M_5	M_6
M_7	M_8	M_9

(b) 회선 마스크

출력 픽셀 값 :

$$I_1 \times M_1 + I_2 \times M_2 + I_3 \times M_3 + I_4 \times M_4 + I_5 \times M_5 + I_6 \times M_6 + I_7 \times M_7 + I_8 \times M_8 + I_9 \times M_9$$

✓가중치를 포함한 회선 마스크가 이동하면서 수행

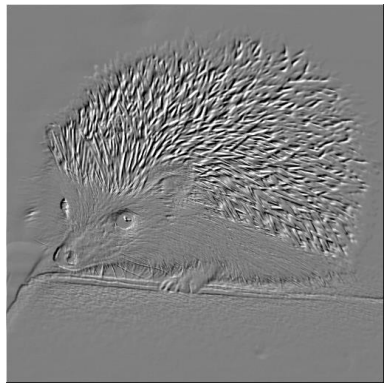


[그림 6-7] 디지털 영상에서 회선을 처리하는 과정

화소 영역 처리 ▶ 엠보싱

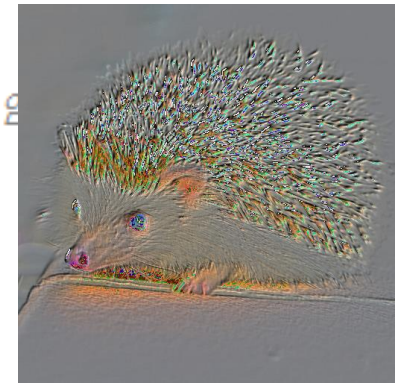
원본 영상

엠보싱



결과 영상

```
mask = [[-1.0, 0.0, 0.0], # 엠
        [0.0, 0.0, 0.0],
        [0.0, 0.0, 1.0]]
```



엠보싱(HIS)

1. 엠보싱, 엠보싱(HSI)

✓ 적절하게 구분된 경계선으로 영상이 양각된 것 처럼 느껴짐

```
# ** 회선 연산 **
for i in range(inH) :
    for k in range(inW) :
        # 마스크(3x3)와 한점을 중심으로 한 3x3을 곱하기
        S = 0.0 # 마스크 9개와 입력값 9개를 각각 곱해서 합한 값.
        for m in range(3):
            for n in range(3):
                S += tmpInImage[i + m][k + n] * mask[m][n]
        tmpOutImage[i][k] = S
```

화소 영역 처리 ▶ 블러링

원본 영상

블러링

3x3 마스크 적용

9x9 마스크 적용

2. 블러링(3x3), (9x9)

- ✓ 영상의 세밀한 부분을 제거하여 영상을 흐리거나 부드럽게 하는 기술
- ✓ 영상의 세밀한 부분은 고주파 성분인데, 고주파 성분을 제거해 줌.
- ✓ 블러링 회선 마스크는 모든 계수가 양수로 전체 합은 1

```
mask = [[1./9, 1./9, 1./9],
        [1./9, 1./9, 1./9],
        [1./9, 1./9, 1./9]]
```

3x3 마스크

```
[[1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81]]
```

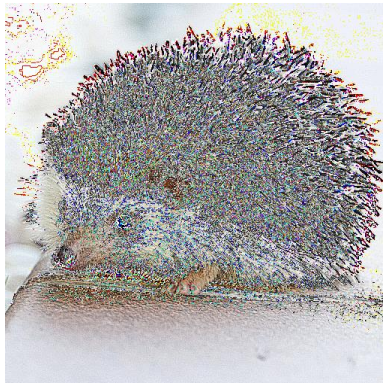
9x9 마스크

화소 영역 처리 ▸ 샤프닝

3. 샤프닝

- ✓ 영상의 상세한 부분을 더욱 강조하여 표현
- ✓ 상세한 부분은 고주파 성분이므로, 저주파 성분을 제거하면 샤프닝 효과

원본 영상

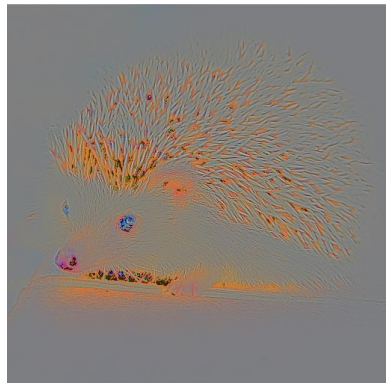


샤프닝

```
mask = [[-1., -1., -1.],
        [-1., 9., -1.],
        [-1., -1., -1.]]
```

4. 고주파 샤프닝

- ✓ 고주파 통과 필터링: 낮은 주파수 성분이 제거되어 경계선이 확연하게 보임
- ✓ 고주파 강조: 중요 성분이 남은 채 경계선 부분이 강조됨

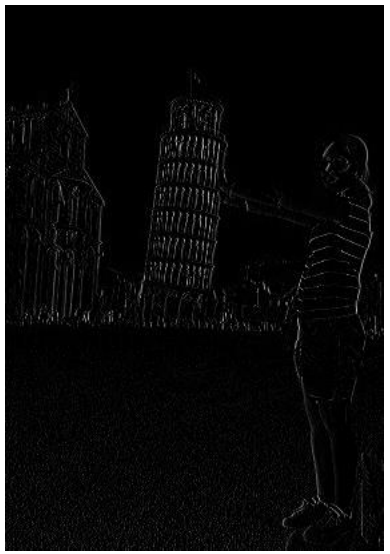


고주파 통과

```
mask = [[-1./9., -1./9., -1./9.],
        [-1./9., 8./9., -1./9.],
        [-1./9., -1./9., -1./9.]]
```

경계선 검출 ▶ 에지 검출기

원본 영상



수직 에지 검출

```
mask = [[0.0, 0.0, 0.0],
        [-1.0, 1.0, 0.0],
        [0.0,  0.0, 0.0]]
```

✓에지(edge)

- 디지털 영상의 밝기가 높은 값에서 낮은 값으로 변하는 지점
- 영상을 구성하는 객체 간의 경계(=경계선)

1. 에지 검출기 : 수직 에지 검출

✓ 화소 간의 차이를 이용하는 것

경계선 검출 ▶ 유사 연산자

원본 영상

유사연산자 에지 검출

```
max = 0.0; // 블록이 이동할 때마다 최대값 초기화
for (n = 0; n < 3; n++) {
    for (m = 0; m < 3; m++) {
        if (fabs(tmpInImage[i + 1][k + 1] - tmpInImage[i + n][k + m]) >= max)
            max = fabs(tmpInImage[i + 1][k + 1] - tmpInImage[i + n][k + m]);
    }
}
tmpOutImage[i][k] = max;
```

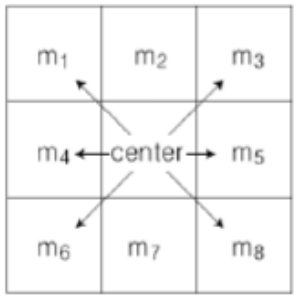
✓간단한 에지 추출 기법

- 연산 자체가 간단하고 빠름
- 유사 연산자(Homogeneity Operator)와 차 연산자(Difference Operator)가 있음

2. 유사 연산자

✓가장 단순한 에지 검출 방법

✓뺄셈연산이 여러 번 수행되어 계산 시간이 많이 소요



New Pixel = max(|center-m1|...|center-m8|)
총 8번 수행

ColorImageProcessing.exe

경계선 검출 ▶ 차 연산자

원본 영상

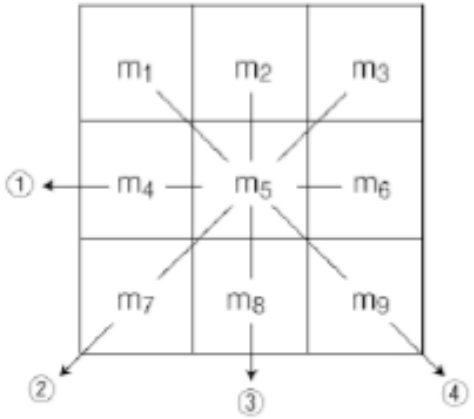
차연산자로 에지 검출

```
max = 0.0; // 블록이 이동할 때마다 최대값 초기화
temp = fabs(tmpInImage[i][k] - tmpInImage[i + 2][k + 2]);
if (temp >= max) max = temp;
temp = fabs(tmpInImage[i][k + 1] - tmpInImage[i + 2][k + 1]);
if (temp >= max) max = temp;
temp = fabs(tmpInImage[i][k + 2] - tmpInImage[i + 2][k]);
if (temp >= max) max = temp;
temp = fabs(tmpInImage[i + 1][k] - tmpInImage[i + 1][k + 1]);
if (temp >= max) max = temp;

tmpOutImage[i][k] = max;
```

3. 차 연산자

- ✓ 유사 연산자의 계산 시간이 오래 걸리는 단점을 보안
- ✓ 뺄셈 연산이 화소당 네 번으로 연산 시간이 빠름



경계선 검출 ▶ 로버츠(Roberts)

원본 영상

로버츠 행 검출 마스크로 에지 검출

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

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;
```

✓미분을 이용한 에지 검출 방법

- 영상의 에지는 화소의 밝기 값이 급격히 변하는 부분
- 함수의 변화분을 찾는 미분 연산이 이용됨

4. 로버츠 마스크

✓1차 미분을 이용한 회선 마스크

✓장점: 크기가 작아 빠른 속도로 동작

✓단점 : 돌출된 값을 잘 평균할 수 없으며, 잡음에 민감함

경계선 검출 ▶ 소벨(Sobel)

원본 영상

소벨 행 검출 마스크로 에지 검출

5. 소벨 마스크

✓1차 미분을 이용한 회선 마스크

✓장점: 돌출된 값을 비교적 잘 평균화 함.

✓단점: 대각선 방향에 놓인 에지에 더 민감하게 반응함

```
double mask[3][3] = { {-1.0, -2.0, -1.0},
                      {0.0,  0.0, 0.0},
                      {1.0,  2.0, 1.0} };

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;
```

경계선 검출 ▶ 라플라시안

원본 영상

라플라시안 회선 마스크로 에지 검출

```
double mask[3][3] = { {-1.0, -1.0, -1.0},
                      {-1.0,  8.0, -1.0},
                      {-1.0, -1.0, -1.0} };
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;
```

✓2차 미분을 이용한 에지 검출

- 장점: 미분을 한번 더 수행하므로, 1차 미분의 단점(에지가 있는 영역을 지날 때 민감하게 반응)을 완화시킴
- 단점: 고립된 잡음에 민감하고, 윤곽의 강도만 검출하지 방향은 구하지 못함.

6. 라플라시안 연산자

✓ 대표적인 2차 미분 연산자로, 에지 검출 성능이 우수함

✓ 에지의 방향은 검출하지 못하고, 실제보다 많은 에지를 검출

경계선 검출 ▶ LoG

원본 영상

LoG로 검출된 에지

```
double mask[5][5] = { {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} };
S = 0.0; // 마스크 25개와 입력값을 각각 곱해서 합한 값
for (int m = 0; m < 5; m++)
    for (int n = 0; n < 5; n++)
        S += tmpInImage[i + m][k + n] * mask[m][n];
tmpOutImage[i][k] = S;
```

7. LoG(Laplacian of Gaussian) 연산자

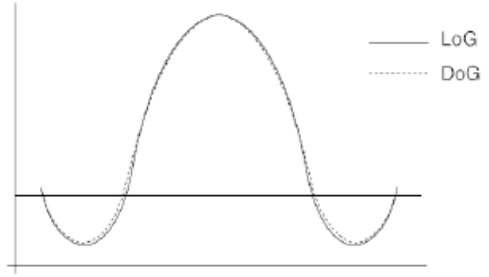
- ✓ 2차 미분 이용한 회선 마스크
- ✓ 잡음에 민감한 라플라시안의 문제를 해결하기 위해 만듦.
- ✓ 계산 시간이 많이 소요됨
- ✓ LoG 연산자 공식

$$\mathrm{LoG}(x,\ y) = \frac{1}{\pi\sigma^4}\left[1 - \frac{(x^2+y^2)}{2\sigma^2}\right] - e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$

경계선 검출 ▶ DoG

원본 영상

DoG로 검출된 에지



LoG와 DoG 그래프 비교

8. DoG(Difference of Gaussians) 연산자

✓2차 미분 이용한 회선 마스크

✓계산 시간이 많이 걸리는 LoG의 단점 보완 위해 등장

```
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} };

S = 0.0; // 마스크 49개와 입력값을 각각 곱해서 합한 값
for (int m = 0; m < MSIZE; m++)
    for (int n = 0; n < MSIZE; n++)
        S += tmpInImage[i + m][k + n] * mask[m][n];
tmpOutImage[i][k] = S;
```

마무리

- 느낀 점
  - 영상 처리의 종류와 구현 방법에 대한 알게 됨
  - 라이브러리가 제공하는 기능을 low level로 구현함으로써, data type conversion으로 인한 debugging에 자신감을 얻음
  - Python 버전을 C++로 포팅하여, MFC 기반 구현에 대해 익숙해짐.
- 부족한 점
  - 다양한 경우의 테스트 및 Error handling 필요
  - Mask를 입력 받음
- 발전 방향 구현 내용
  - Ctrl+Z (취소하기) 기능 구현

감사합니다