[Intel] 엣지 AI S/W 아카데미

절차 지향 프로그래밍

조수환

OpenCV 없이 구현한 Python 영상처리

프로젝트 개요

목적

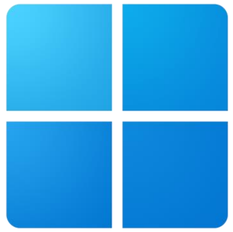
- Python 언어 프로그래밍 역량 강화
- 영상처리 알고리즘에 대한 이해

접근 방식

- Python을 통한 영상처리 파이프라인 직접 구현
- 이미지 파일 입출력 및 GUI 구현
- 영상처리 알고리즘 구현

개발 환경

- OS : Windows 11
- Language : C
- Tool : Visual Studio 2022



구현 기능

주요 기능

1. 화소 점 변환

- 밝기 조절
  - 어둡게/밝게
- 반전
- 포스터라이징
- 범위 강조
- 감마 보정
- 이진화
  - 직접 입력/평균값/중앙값
- 파라볼라 변환
  - CAP/CUP
- 히스토그램
  - 스트레칭
  - 평활화

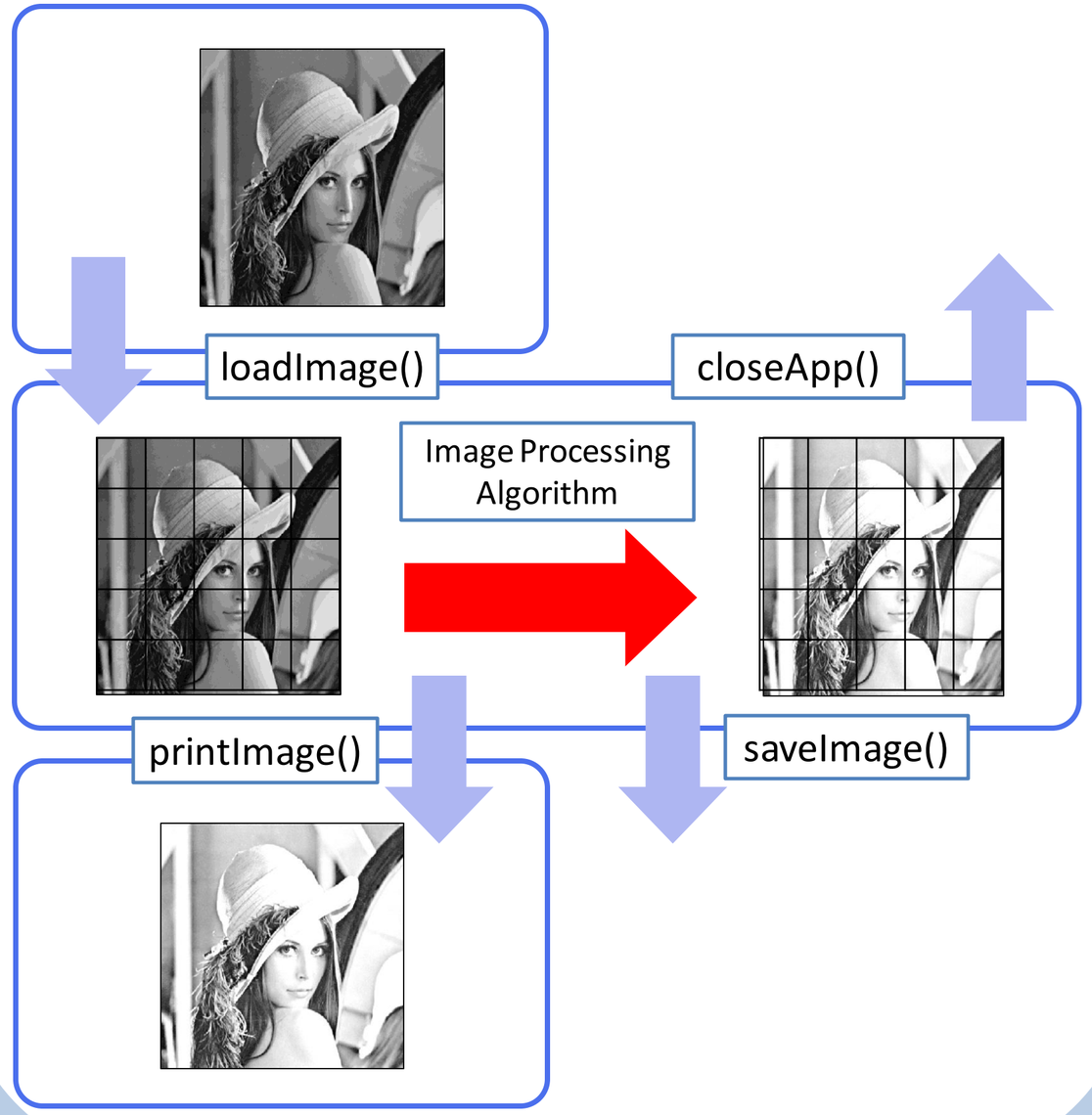
2. 영역 변환

- 엠보싱
- 블러링
- 스무딩
- 샤프닝
- 에지 검출
  - 소벨/프레윗/라플라시안
  - LOG/DOG
  - 케니

3. 기하학 변환

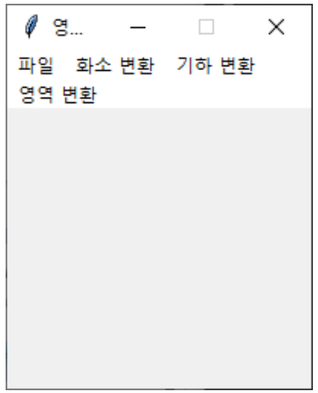
- 회전
- 크기 조절
- 이동
- 미러링(대칭)
  - 상하/좌우

구조도

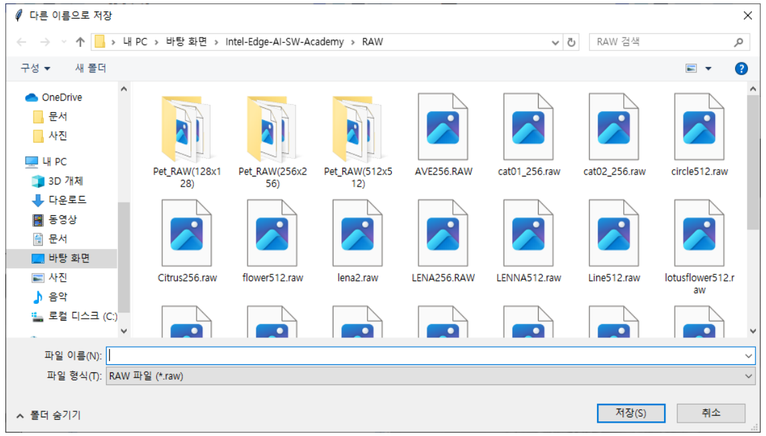


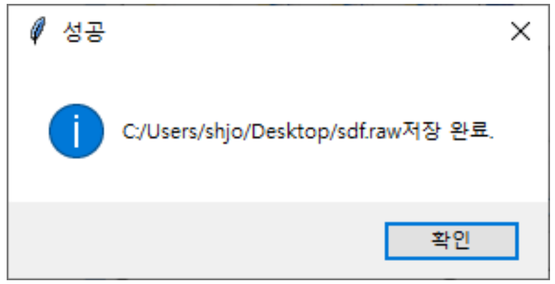
구현 기능

메인 화면 / 열기



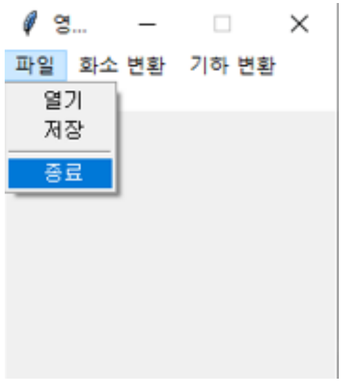
저장





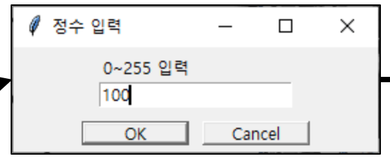
종료

```
def closeApp():  # 종료
    window.destroy()
```



화소 점 처리

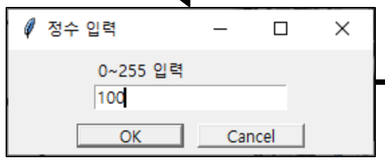
밝기 조절



밝기 증가

각 픽셀에 일정한 값을 더하여 영상의 전체 밝기를 증가

```
px = inImage[i][k] + val
if px > 255:
    px = 255
if px < 0:
    px = 0
outImage[i][k] = px
```



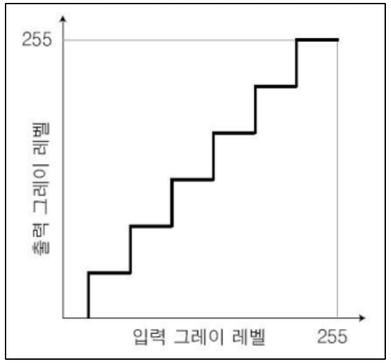
밝기 감소

각 픽셀에 일정한 값을 빼 영상의 전체 밝기를 감소

```
px = inImage[i][k] - val
if px > 255:
    px = 255
if px < 0:
    px = 0
outImage[i][k] = px
```

화소 점 처리

반전 / 포스터라이징

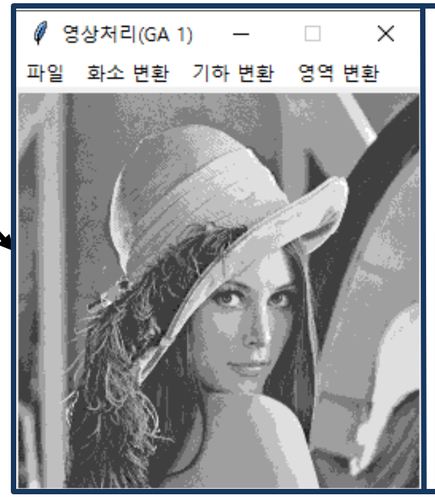


Output(q) = 255 - Input(p)

영상 반전

각 화소의 값이 영상 내에
대칭이 되는 값으로 변환

```
arrayImage = np.array(inImage)  # 스칼라 연산을 위해 변환
    outImage = 255 - arrayImage
```



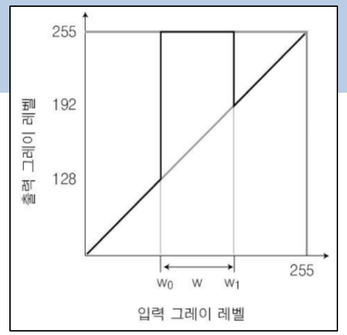
포스터라이징

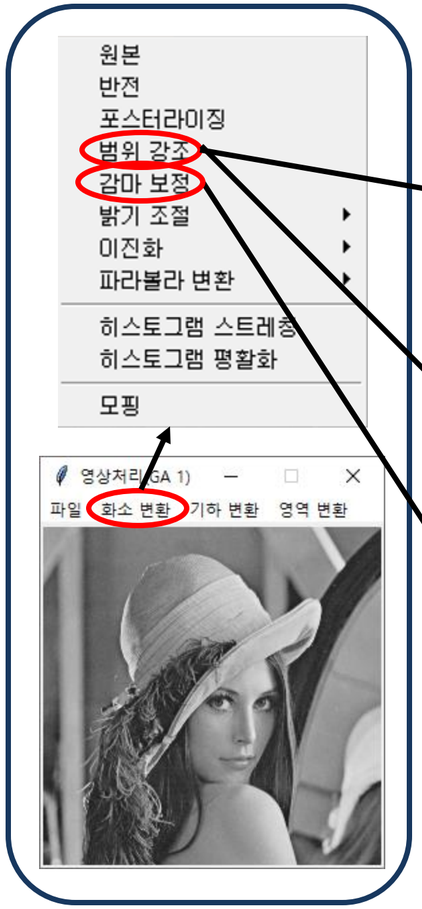
영상에서 화소에 있는 명암 값의
범위를 경계 값으로 축소

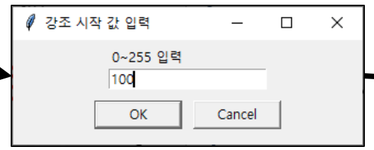
```
    for i in range(inH):
        for k in range(inW):
            if 0 <= inImage[i][k] <= 31:
                outImage[i][k] = 31
            elif 32 <= inImage[i][k] <= 63:
                outImage[i][k] = 63
            elif 64 <= inImage[i][k] <= 95:
                outImage[i][k] = 95
            elif 96 <= inImage[i][k] <= 127:
                outImage[i][k] = 127
            elif 128 <= inImage[i][k] <= 159:
                outImage[i][k] = 159
            elif 160 <= inImage[i][k] <= 191:
                outImage[i][k] = 191
            elif 192 <= inImage[i][k] <= 223:
                outImage[i][k] = 223
            elif 224 <= inImage[i][k] <= 255:
                outImage[i][k] = 255
```

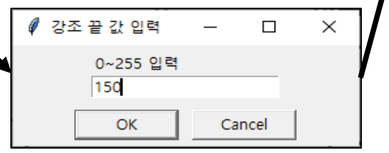
화소 점 처리

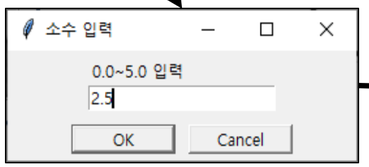
범위 강조 / 감마 보정











범위 강조

특정 범위의 화소를 강조

```
if (startVal < inImage[i][k] < endVal) or (endVal < inImage[i][k] < startVal):
    outImage[i][k] = 255
else:
    outImage[i][k] = inImage[i][k]
```

$$Output(q) = [Input(p)](1/\gamma)$$

감마 보정

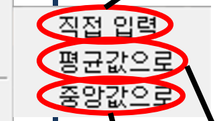
함수의 감마 값에 따라 영상을

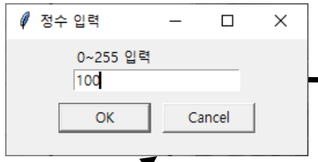
밝게 하거나 흐리게 조절

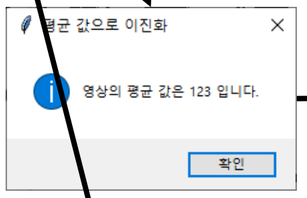
```
outImage = pow((arrayImage / 255.0), gam) * 255.0
outImage = outImage.astype(np.uint32)  # 자료형 변환
```

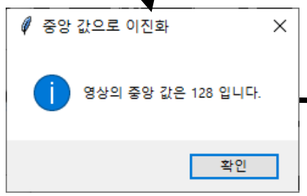
화소 점 처리

이진화

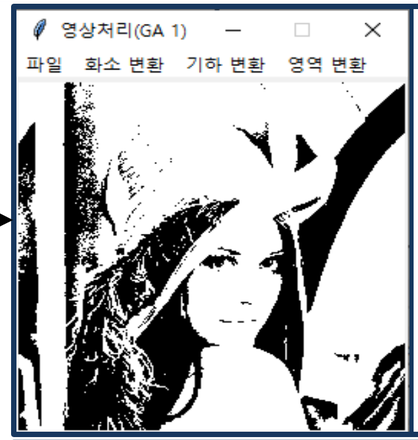








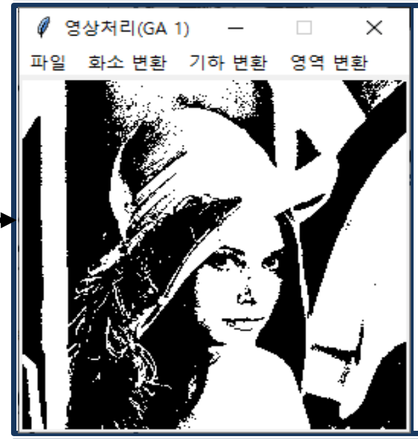
$$Output(q) = \begin{cases} 255 & Input(p) \geq T \\ 0 & Input(p) < T \end{cases}$$



직접 입력

입력한 값을 기준으로 영상을 이진화(0 or 255)

```
val = askinteger("정수 입력", '0~255 입력', maxvalue=255, minvalue=-255)

if inImage[i][k] > val:
    outImage[i][k] = 255
else:
    outImage[i][k] = 0
```



평균값으로

영상의 평균 화소 값을 기준으로 이진화

```
val = int(np.mean(inImage))

if inImage[i][k] > val:
    outImage[i][k] = 255
else:
    outImage[i][k] = 0
```

중앙값으로

영상의 중앙 화소 값을 기준으로 이진화

```
val = np.median(inImage)

if inImage[i][k] > val:
    outImage[i][k] = 255
else:
    outImage[i][k] = 0
```

화소 점 처리

파라볼라 변환

$$= 255(\frac{x}{127} - 1)^2$$

CAP(Corner Adaptive Point)

모서리와 변화가 큰 지점을 감지하여
특징점을 추출

```
arrayImage = np.array(inImage, dtype=float)
outImage = 255.0 * pow((arrayImage / 127.0 - 1.0), 2.0)
outImage = outImage.astype(np.uint32)
```

$$-255(\frac{x}{127} - 1)^2 + 255$$

CUP(Corner-based Unique Point)

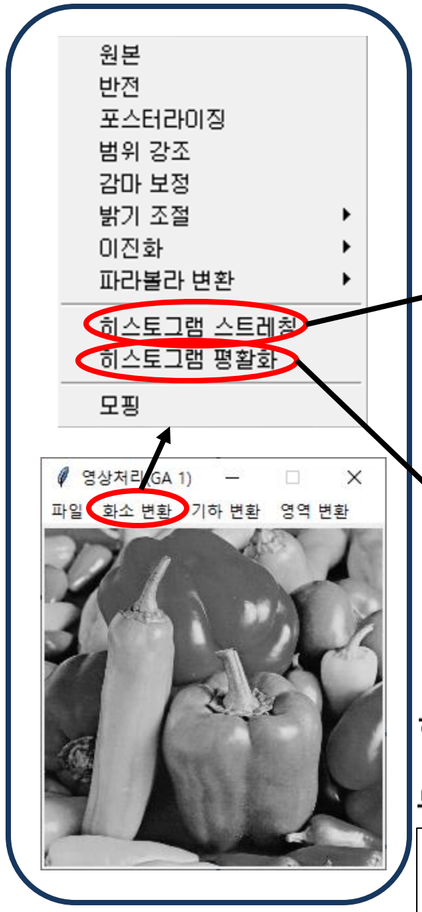
특징점이 유일하고 반복 가능하도록 보장하여
정확한 매칭을 위해 설계

```
arrayImage = np.array(inImage, dtype=float)
outImage = 255.0 - 255.0 * pow((arrayImage / 127.0 - 1.0), 2.0)
outImage = outImage.astype(np.uint32)
```

화소 점 처리

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

히스토그램 스트레칭 / 평활화



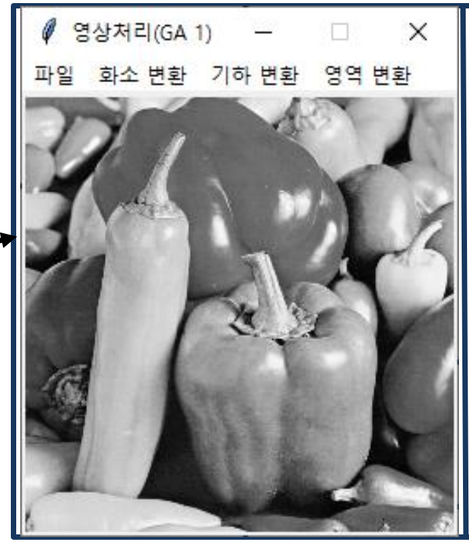
히스토그램 생성

누적 빈도 수(누적합)를 계산

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

누적 빈도 수를 정규화(정규화 누적합)

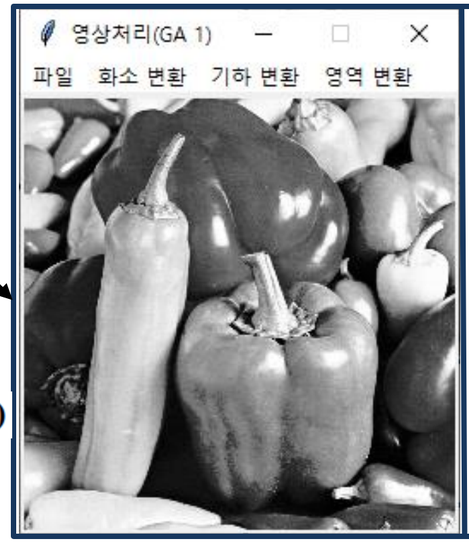
$$n[i] = sum[i] \times \frac{1}{N} \times I_{max}$$



히스토그램 스트레칭

낮은 명암 대비를 보이는 영상의 화질을 향상

```
high = np.max(inImage)
low = np.min(inImage)
# End-In 탐색 : 최대 최소값 사이를 좁혀 스트 레칭 효과를 극대화
# new = (old - low) / (high - low) * 255
old = inImage[i][k]
new = int(((old - low) / (high - low)) * 255.0)
if new > 255:
    new = 255
if new < 0:
    new = 0
outImage[i][k] = new
```



히스토그램 평활화

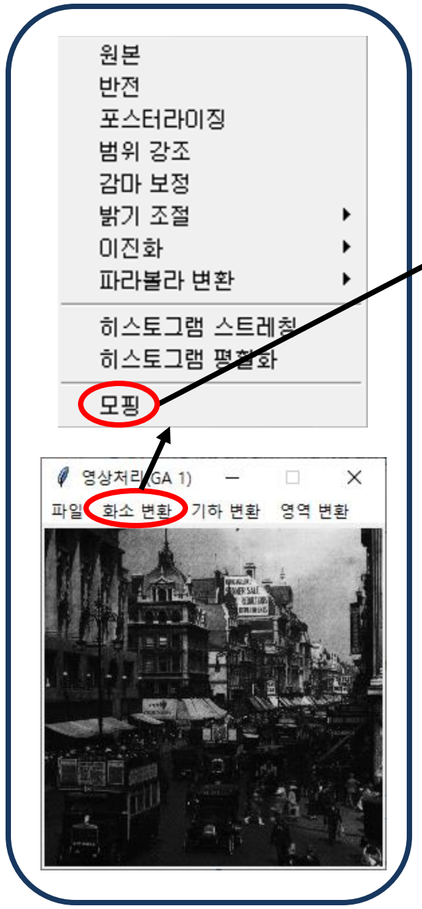
영상의 밝기 분포를 재분배하여

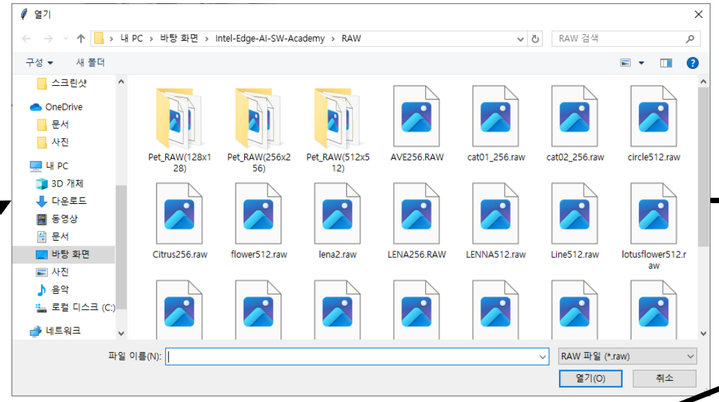
명암 대비를 최대화

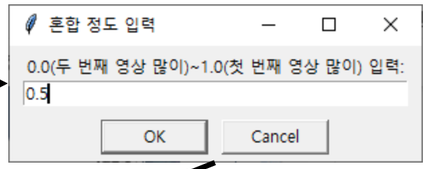
```
histogram, _ = np.histogram(inImage, bins=256, range=(0, 256))
cumHisto = np.cumsum(histogram)
normCumHisto = cumHisto * (1.0 / (inH * inW)) * 255.0
for i in range(inH):
    for k in range(inW):
        outImage[i][k] = normCumHisto[inImage[i][k]]
```

화소 점 처리

모핑







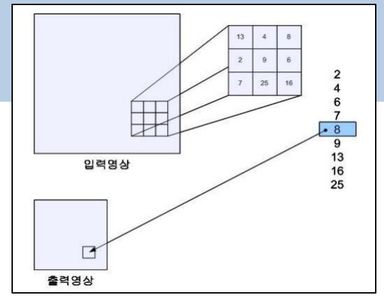
$$O(x,y) = (1-u) \times I_1(x,y) + u \times I_2(x,y)$$

모핑

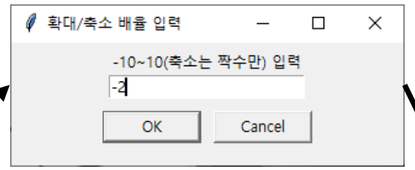
두 개 이상의 이미지나 동영상 사이의 부드러운 변환

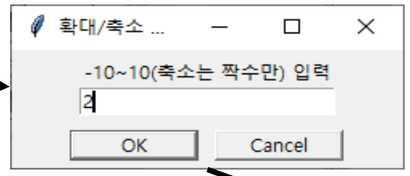
```
# 두 번째 영상 열기
fullName = askopenfilename(parent=window, filetypes=(('RAW 파일', '*.raw'), ('모든 파일', '*.*')))
fsize = os.path.getsize(fullName)  # 파일 크기
inH = inW = int(math.sqrt(fsize))
# 입력 메모리 할당
inImage2 = malloc2D(inH, inW)
# 영상 블렌딩
outImage = blend_factor * inImage + (1 - blend_factor) * inImage2
outImage = outImage.astype(np.uint8)
```

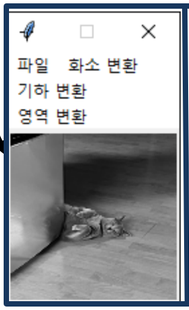
기하학 처리



크기 조절







크기 축소

영상의 모양은 변화시키지 않은 채 크기만을 축소(중간 값을 통한 축소)

```
subMat = inImage[i * scale:i * scale + scale, j * scale:j * scale + scale]  # 축소 영역
histo, _ = np.histogram(subMat, bins=256, range=(0, 256))  # 히스토그램 계산
cumsum = np.cumsum(histo)  # 누적 히스토그램 계산
median = np.argmax(cumsum >= (scale * scale) // 2)  # 중간값 계산
outImage[i][j] = median
```

크기 확대

영상의 크기를 확대(양선형 보간법을 통한 확대)

```
# 원본 이미지에서 해당 위치의 실수 좌표 계산  ex) 확대(3,4) -> 원본(1.5,2)
y_orig = y / scale
x_orig = x / scale
# 원본 이미지에서 가장 가까운 4개 픽셀의 정수 좌표 계산
y1 = int(y_orig)  # 좌측 상단
x1 = int(x_orig)
y2 = min(y1 + 1, inH - 1)  # 우측 하단
x2 = min(x1 + 1, inW - 1)
# 원본 이미지에서 해당 위치와 4개 픽셀 사이의 거리 계산
y_diff = y_orig - y1
x_diff = x_orig - x1
# 가장 가까운 4개 픽셀의 값 가져오기
val1 = inImage[y1][x1]
val2 = inImage[y1][x2]
val3 = inImage[y2][x1]
val4 = inImage[y2][x2]
# z = (1-p)(1-q)a + p(1-q)b + (1-p)qc + pqd
outImage[y][x] = int(val1 * (1 - x_diff) * (1 - y_diff) +
                     val2 * x_diff * (1 - y_diff) +
                     val3 * y_diff * (1 - x_diff) +
                     val4 * x_diff * y_diff)
```

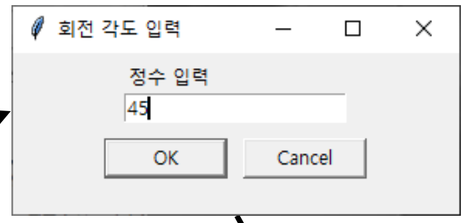
$$E = (1-x)A + xB = A + x(B-A)$$

$$F = (1-x)C + xD = C + x(D-C)$$

$$N = (1-y)E + xF = E + y(F-E)$$

기하학 처리

회전



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

$$W' = H\cos(90-\theta) + \cos\theta$$
$$H' = H\cos\theta + W\cos(90-\theta)$$

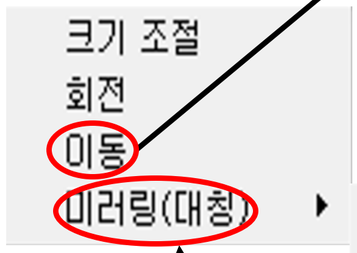
회전

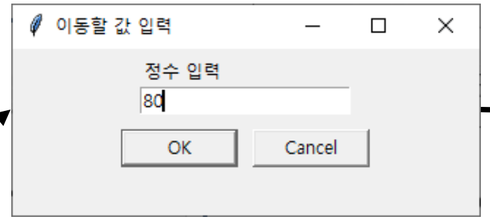
영상을 임의의 방향으로 특정한 각도만큼 회전(잘리지 않게 확대)

```
tmp_angle = angle % 90 * 3.141592 / 180.0
tmp_angle90 = (90 - angle % 90) * 3.141592 / 180.0
# 회전 각도에 따라 변화하는 출력 영상 크기 계산
outH = int(inH * np.cos(tmp_angle90) + inW * np.cos(tmp_angle))
outW = int(inW * np.cos(tmp_angle) + inH * np.cos(tmp_angle90))
outImage = malloc2D(outH, outW)
degree = angle * 3.141592 / 180.0

dx = (outW - inW) // 2
dy = (outH - inH) // 2
cx = outH // 2
cy = outH // 2
# 회전의 시작은 입력 영상 위치에서 해야 함
tmp_image = np.zeros((outH, outW), dtype=np.uint32)
tmp_image[dx:dx+inH, dy:dy+inW] = inImage

for i in range(outH):
    for k in range(outW):
        xd = i
        yd = k
        xs = int(np.cos(degree) * (xd - cx) + np.sin(degree) * (yd - cy)) + cx
        ys = int(-np.sin(degree) * (xd - cx) + np.cos(degree) * (yd - cy)) + cy
        if (0 <= xs < outH) and (0 <= ys < outW):
            outImage[xd][yd] = tmp_image[xs][ys]
```

기하학 처리

이동 / 미러링(대칭)





이동

영상의 모양은 변화시키지 않은 채 위치만을 이동

```
if move >= 0:  # 우측 하단으로 이동
    if (0 <= i < outH) and (0 <= k < outW):
        outImage[i + move][k + move] = inImage[i][k]
else:  # 좌측 상단으로 이동
    if (0 - move <= i < outH) and (0 - move <= k <
outW):
        outImage[i + move][k + move] = inImage[i][k]
```

상하 반전

좌우 반전

상하 반전

영상을 가로축으로 뒤집음

```
if option == 0:
    outImage[::-1, :] = inImage
```

좌우 반전

영상을 세로축으로 뒤집음

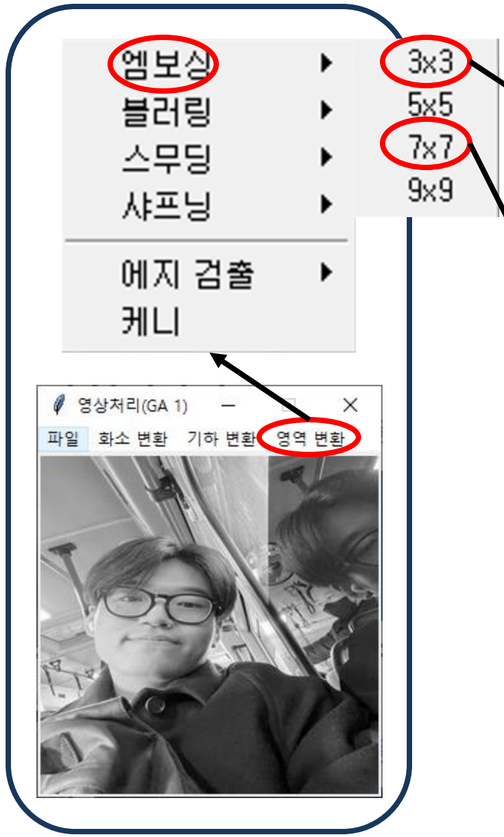
```
elif option == 1:
    outImage[:, ::-1] = inImage
```

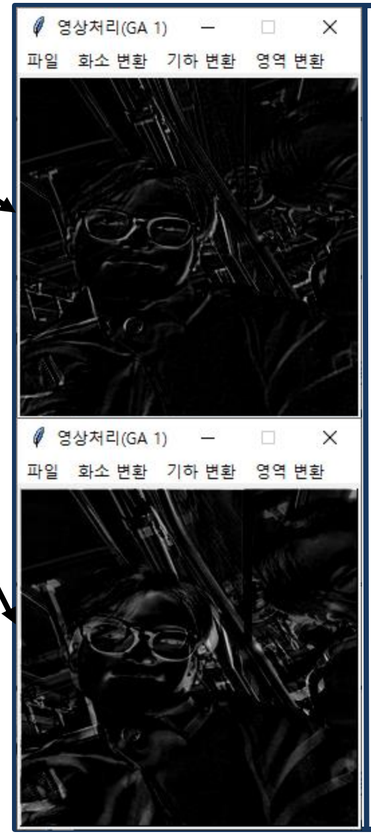
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -(x - x_0) \\ y \end{bmatrix} + \begin{bmatrix} x_0 \\ 0 \end{bmatrix}$$

영역 처리

-1	0	0
0	0	0
0	0	1

엠보싱





엠보싱

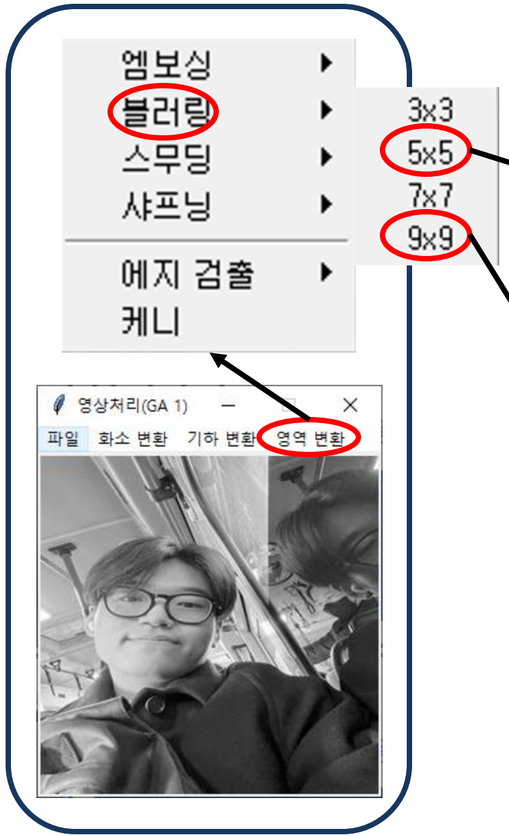
외곽선을 부각시켜 입체적인 느낌을 줄 수 있도록 변환

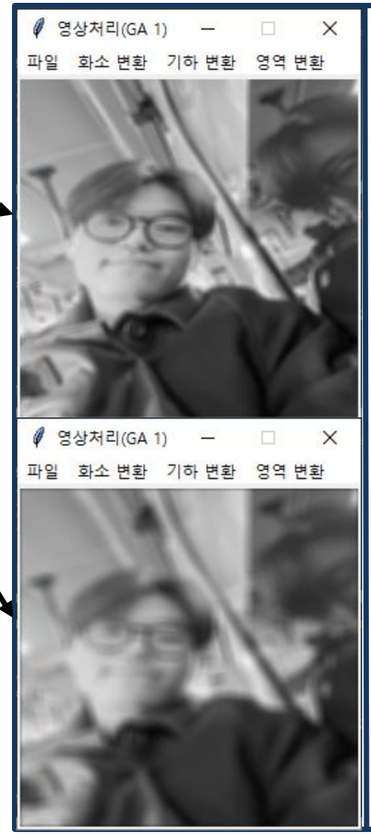
```
mask = np.zeros((masksize, masksize), dtype=float)
center = masksize // 2
mask[0, 0] = -1.0  # 처음 -1
mask[masksize - 1, masksize - 1] = 1.0  # 끝 1

# 입력 이미지 -> 임시 이미지에 넣기(패딩 완료)
tmpInput = np.pad(inImage, masksize // 2, mode='constant')
tmpOutput = np.zeros_like(outImage, dtype=float)
# 컨볼루션 연산
for i in range(outH):
    for k in range(outW):
        S = np.sum(tmpInput[i:i + masksize, k:k + masksize] * mask)
        tmpOutput[i, k] = S
# 클리핑
outImage = np.clip(tmpOutput, 0, 255).astype(np.uint8)
```

영역 처리

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

블러링





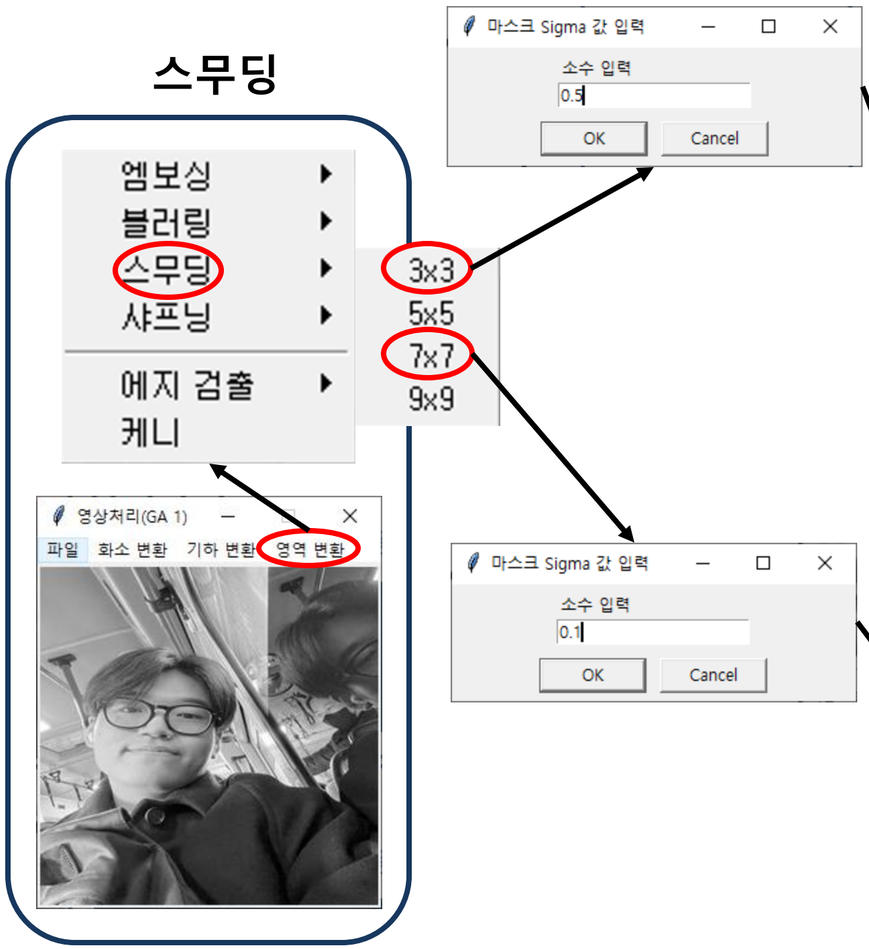
블러링(평균)

영상의 세밀한 부분을 제거하여 영상을 흐리게 하거나 부드럽게 변환

```
mask = np.zeros((masksize, masksize), dtype=float)
center = masksize // 2
mask = (1.0 / (masksize * masksize))  # 평균 마스크

# 입력 이미지 -> 임시 이미지에 넣기(패딩 완료)
tmpInput = np.pad(inImage, masksize // 2, mode='constant')
tmpOutput = np.zeros_like(outImage, dtype=float)
# 컨볼루션 연산
for i in range(outH):
    for k in range(outW):
        S = np.sum(tmpInput[i:i + masksize, k:k + masksize] * mask)
        tmpOutput[i, k] = S
# 클리핑
outImage = np.clip(tmpOutput, 0, 255).astype(np.uint8)
```

영역 처리

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16



스무딩(가우시안)

주변 화소 값들을 결합하여 영상의 선명도를 감소시키고 부드럽게 만드는 과정

```
mask = np.zeros((masksize, masksize), dtype=float)
center = masksize // 2
sigma = askfloat("마스크 Sigma 값 입력", '소수 입력')
for i in range(masksize):
    for j in range(masksize):
        # 가우시안 마스크 생성
        x = np.sqrt((np.power((i - center), 2) + np.power((j - center), 2)))
        gaussian = np.exp(-(x * x) / (2.0 * sigma * sigma)) / (sigma * np.sqrt(2.0 * np.pi))
        mask[i][j] = gaussian

# 입력 이미지 -> 임시 이미지에 넣기(패딩 완료)
tmpInput = np.pad(inImage, masksize // 2, mode='constant')
tmpOutput = np.zeros_like(outImage, dtype=float)
# 컨볼루션 연산
for i in range(outH):
    for k in range(outW):
        S = np.sum(tmpInput[i:i + masksize, k:k + masksize] * mask)
        tmpOutput[i, k] = S
# 클리핑
outImage = np.clip(tmpOutput, 0, 255).astype(np.uint8)
```

영역 처리

mask2		
0	-1	0
-1	5	-1
0	-1	0

mask3		
-1	-1	-1
-1	9	-1
-1	-1	-1

샤프닝

샤프닝

영상에서 선명도를 향상시키기 위해 주변 화소와의 대비를 강화하는 과정

```
mask = np.zeros((masksize, masksize), dtype=float)
center = masksize // 2
sigma = askfloat("마스크 Sigma 값 입력", '소수 입력')
for i in range(masksize):
  for j in range(masksize):
    if i == center and j == center:
        mask[i, j] = float(masksize * masksize)  # 중간 마스크 크기
    else:
        mask[i, j] = -1.0  # 나머지 -1

# 입력 이미지 -> 임시 이미지에 넣기(패딩 완료)
tmpInput = np.pad(inImage, masksize // 2, mode='constant')
tmpOutput = np.zeros_like(outImage, dtype=float)
# 컨볼루션 연산
for i in range(outH):
    for k in range(outW):
        S = np.sum(tmpInput[i:i + masksize, k:k + masksize] * mask)
        tmpOutput[i, k] = S
# 클리핑
outImage = np.clip(tmpOutput, 0, 255).astype(np.uint8)
```

영역 처리

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

에지 검출

프레윗(Prewitt) 마스크 사용

x축과 y축의 각 방향으로 차분을 세 번 계산하여

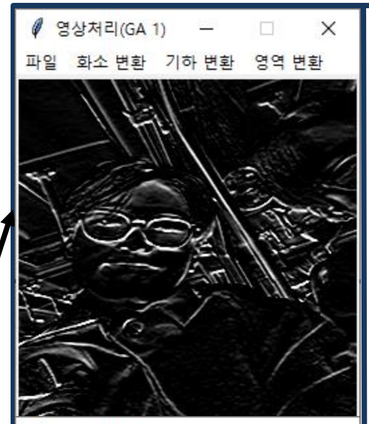
경계를 검출, 상하/좌우에 비해 대각선 검출에 취약

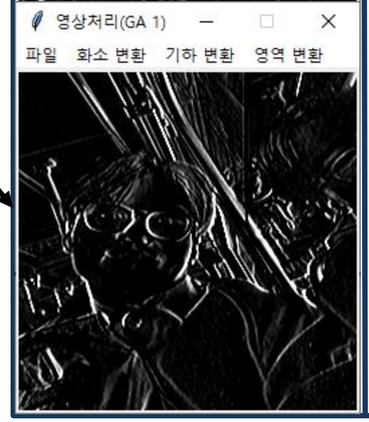
```python
if axis == 0:
    mask = np.array([[-1, -1, -1],
                     [0,  0,  0],
                     [1,  1,  1]])
elif axis == 1:
    mask = np.array([[-1,  0,  1],
                     [-1,  0,  1],
                     [-1,  0,  1]])
masksize = mask.shape[0]
# 입력 이미지 -> 임시 이미지에 넣기(패딩 완료)
tmpInput = np.pad(inImage, masksize // 2, mode='constant')
tmpOutput = np.zeros_like(outImage, dtype=float)
# 컨볼루션 연산
for i in range(outH):
    for k in range(outW):
        S = np.sum(tmpInput[i:i + masksize, k:k + masksize] * mask)
        tmpOutput[i, k] = S
# 클리핑
outImage = np.clip(tmpOutput, 0, 255).astype(np.uint8)
```

영역 처리

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

에지 검출

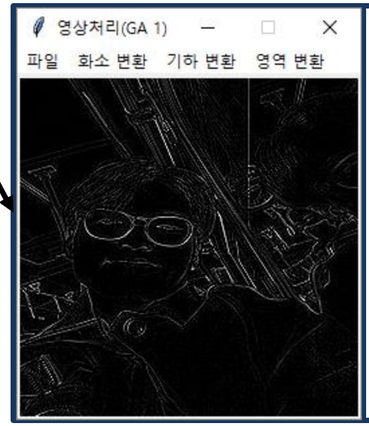




소벨(Sobel) 마스크 사용

중심 픽셀의 차분 비중을 두 배로 적용, 대각선 방향의 경계 검출까지 강력

```python
if axis == 0:
    mask = np.array([[-1, -2, -1],
                     [0,  0,  0],
                     [1,  2,  1]])
elif axis == 1:
    mask = np.array([[-1,  0,  1],
                     [-2,  0,  2],
                     [-1,  0,  1]])
masksize = mask.shape[0]
# 입력 이미지 -> 임시 이미지에 넣기(패딩 완료)
tmpInput = np.pad(inImage, masksize // 2, mode='constant')
tmpOutput = np.zeros_like(outImage, dtype=float)
# 컨볼루션 연산
for i in range(outH):
    for k in range(outW):
        S = np.sum(tmpInput[i:i + masksize, k:k + masksize] * mask)
        tmpOutput[i, k] = S
# 클리핑
outImage = np.clip(tmpOutput, 0, 255).astype(np.uint8)
```



라플라시안(Laplacian) 마스크 사용

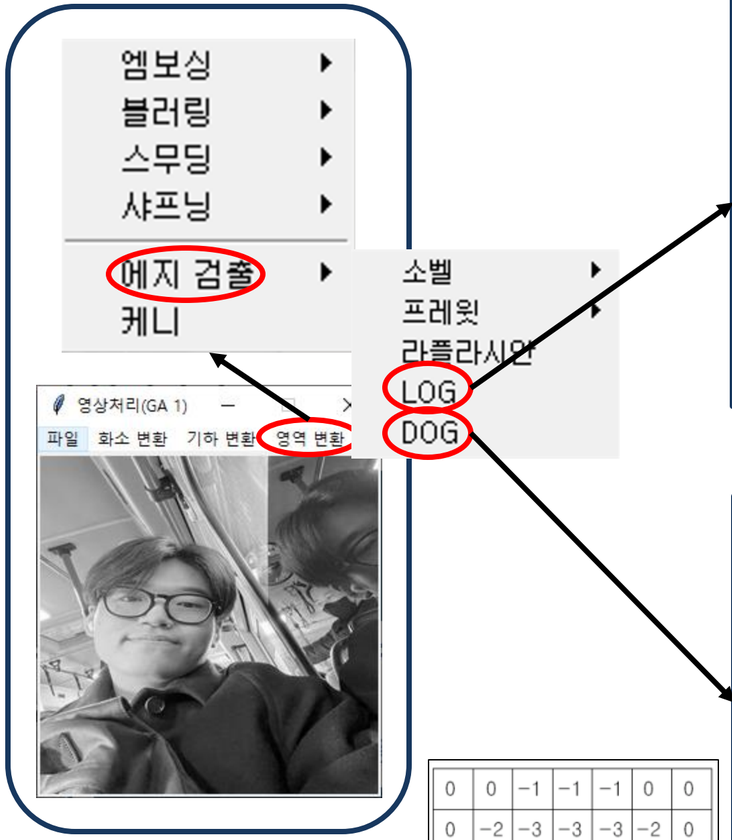
2차 미분을 적용, 경계를 더 제대로 검출 가능

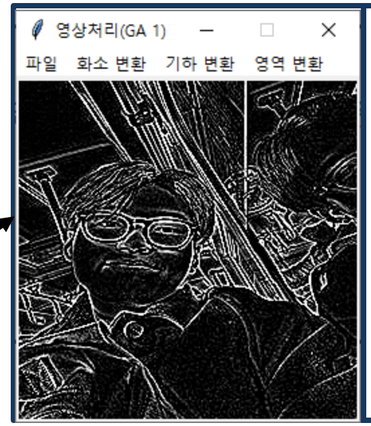
```python
mask = np.array([[0,  1,  0],
                 [1, -4,  1],
                 [0,  1,  0]])
masksize = mask.shape[0]
# 입력 이미지 -> 임시 이미지에 넣기(패딩 완료)
tmpInput = np.pad(inImage, masksize // 2, mode='constant')
tmpOutput = np.zeros_like(outImage, dtype=float)
# 컨볼루션 연산
for i in range(outH):
    for k in range(outW):
        S = np.sum(tmpInput[i:i + masksize, k:k + masksize] * mask)
        tmpOutput[i, k] = S
# 클리핑
outImage = np.clip(tmpOutput, 0, 255).astype(np.uint8)
```

영역 처리

0	0	−1	0	0
0	−1	−2	−1	0
−1	−2	16	−2	−1
0	−1	−2	−1	0
0	0	−1	0	0

에지 검출

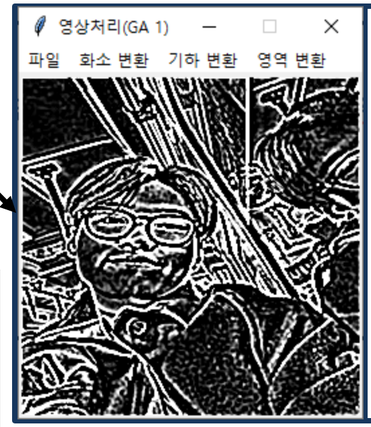




LOG(Laplacian of Gaussian) 마스크 사용

잡음에 민감한 라플라시안 마스크의 문제점 해결을 위해, 가우시안 스무딩 후 라플라시안 마스크 사용

```
mask = np.array([[0, 0, -1, 0, 0],
                 [0, -1, -2, -1, 0],
                 [-1, -2, 16, -2, -1],
                 [0, -1, -2, -1, 0],
                 [0, 0, -1, 0, 0]])
masksize = mask.shape[0]
# 패딩
# 컨볼루션
# 클리핑
```



DOG(Difference of Gaussians) 마스크 사용

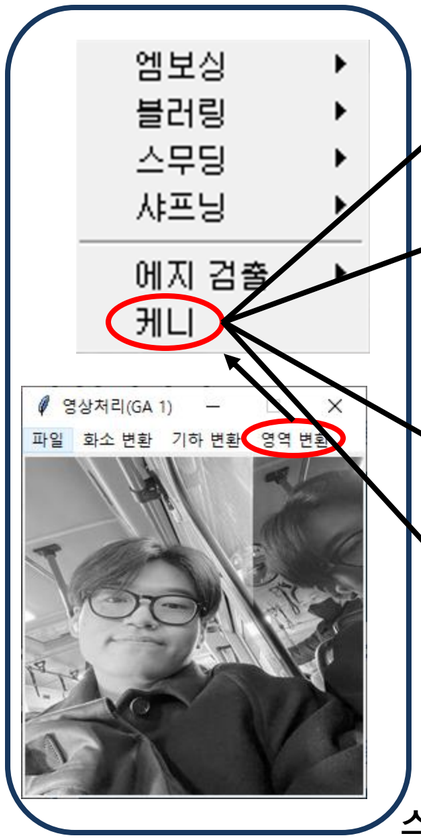
계산량이 많은 LOG의 단점 극복을 위해 서로 다른 가우시안 마스크의 차를 이용

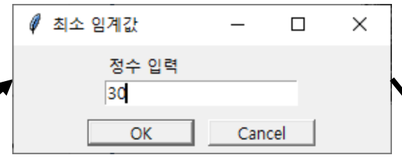
```
mask = np.array([[0, 0, -1, -1, -1, 0, 0],
                 [0, -2, -3, -3, -3, -2, 0],
                 [-1, -3,  5,  5,  5, -3, -1],
                 [-1, -3,  5, 16,  5, -3, -1],
                 [-1, -3,  5,  5,  5, -3, -1],
                 [ 0, -2, -3, -3, -3, -2, 0],
                 [ 0, 0, -1, -1, -1, 0, 0]
masksize = mask.shape[0]
# 패딩
# 컨볼루션
# 클리핑
```

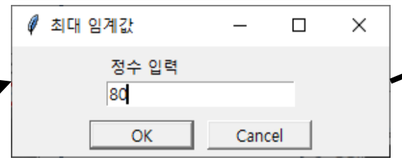
0	0	−1	−1	−1	0	0
0	−2	−3	−3	−3	−2	0
−1	−3	5	5	5	−3	−1
−1	−3	5	16	5	−3	−1
−1	−3	5	5	5	−3	−1
0	−2	−3	−3	−3	−2	0
0	0	−1	−1	−1	0	0

영역 처리

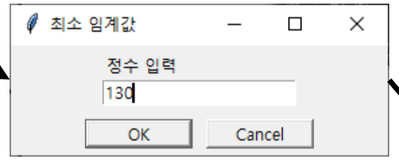
케니

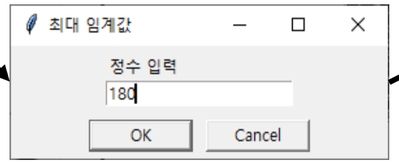


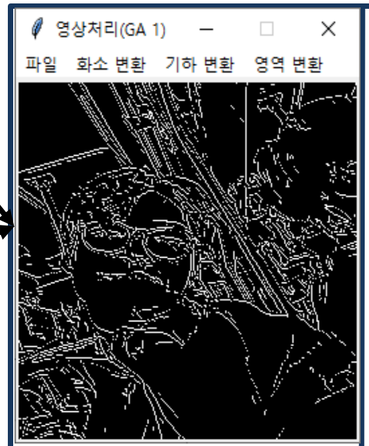


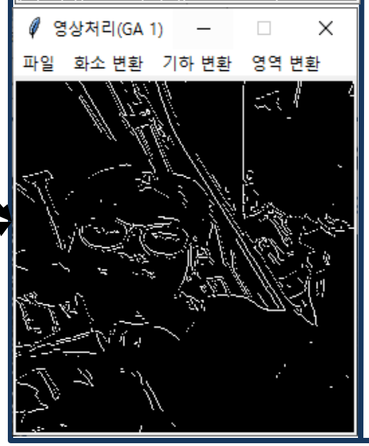


임계값 중요!









케니(Canny) 에지

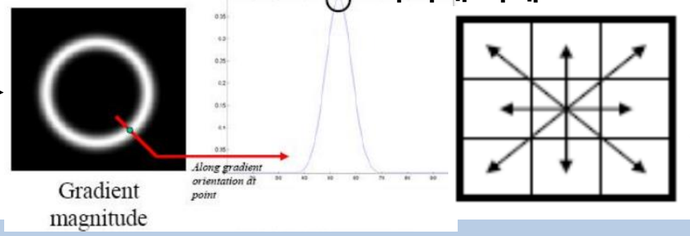
고급 경계 검출 알고리즘, 가우시안 필터를 적용해 노이즈 제거 후, 경계 검출 마스크를 통한 그래디언트 및 방향 계산 후 비최대 억제 + 이중 임계값 처리를 통해 정확한 경계 검출

```python
def nonmax_suppression(sobel, direction):  # 비최대 억제 처리 함수
    rows, cols = sobel.shape
    suppressed = np.zeros_like(sobel)
    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            angle = direction[i, j] * 180.0 / np.pi
            if (0 <= angle < 22.5) or (157.5 <= angle <= 180):
                q, r = sobel[i, j + 1], sobel[i, j - 1]
            elif (22.5 <= angle < 67.5):
                q, r = sobel[i - 1, j + 1], sobel[i + 1, j - 1]
            elif (67.5 <= angle < 112.5):
                q, r = sobel[i - 1, j], sobel[i + 1, j]
            else:
                q, r = sobel[i + 1, j + 1], sobel[i - 1, j - 1]
            if sobel[i, j] >= q and sobel[i, j] >= r:
                suppressed[i, j] = sobel[i, j]
    return suppressed

# 이중 임계값 처리 함수
def hysteresis_thresholding(suppressed, low_threshold, high_threshold):
    rows, cols = suppressed.shape
    edges = np.zeros_like(suppressed)
    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            if suppressed[i, j] >= high_threshold:
                edges[i, j] = 255
            elif low_threshold <= suppressed[i, j] < high_threshold:
                if (suppressed[i - 1:i + 2, j - 1:j + 2] >=
high_threshold).any():
                    edges[i, j] = 255
    return edges
```

스무딩

에지 검출

비최대 억제



$$E(i,j)=\begin{cases}255\,(strong\ edge), & \text{if 에지강도} > T_1\\ 125\,(weak\ edge), & \text{if 에지강도} < T_1\\ 0\,(no\ edge), & \text{if 에지강도} < T_2\end{cases}$$

이중 임계값

느낀점

C에서 Python으로 변경함으로써 더 효율적으로 변경할 수 있었고, C를 사용할 때 구현할 수 없었던 기능들을 구현할 수 있었습니다.

한계점

기능에 중점을 두었기 때문에 많은 오류 처리를 구현하지 못한 제한이 있습니다.

향후 발전 방향

알고리즘을 더 추가하고, 오류 처리를 강화하며, 최적화를 수행합니다.

감사합니다

Github : https://github.com/suhwanjo