컴퓨터그래픽스 2주차 과제 보고서
201300995 이상건 물리학과

구현 내용
가우스 필터 적용하는거임. 이게 뭐냐면

$SUM($

100	200	100	119	120
100	200	100	119	120
99	199	198	200	201
89	180	80	79	78
90	200	90	80	80

X $\frac{1}{9}$

1	1	1
1	1	1
1	1	1

$)$

=

100	200	100	119	120
100	200	100	119	120
99	199	150	200	201
89	180	80	79	78
90	200	90	80	80

오른쪽에 모두 1로 채워진 9x9박스가 있는데 이게 평범한 박스 필터링이다.
이 오른쪽에 원래 이미지에 각각의 원소를 곱해서 더해주는 박스를 커널이라고 정의하는데, 이 커널을 가우스 함수를 이용해서 구현하는 것이다.

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

$\sigma = 1$

$\frac{1}{sum}$

0.0029	0.0133	0.0219	0.0133	0.0029
0.0133	0.0596	0.0983	0.0596	0.0133
0.0219	0.0983	0.1621	0.0983	0.0219
0.0133	0.0596	0.0983	0.0596	0.0133
0.0029	0.0133	0.0219	0.0133	0.0029

이렇게. 이 박스가 가우스 필터 용 커널 박스가 된다.

가우스 함수로 커널의 크기를 크게 키우거나 줄일 수 있으며, 커널의 크기가 충분히 클 때 행렬을 나눠서 곱셈해서 계산하는 것과 통째로 곱셈해서 계산하는 것 중 어느것이 더 빠른 가를 실습하는 것이다. 이론상으로는 나눠서 연산하면 2n개고 통째로 연산하면 n^2이라 n이 충분히 크다면 나눠서 하는게 더 빨라야 한다.

이유(구현 방법)

```python
def my_getGKernel(shape, sigma):
    '''
    :param shape: 생성하고자 하는 gaussian kernel의 shape입니다. (5,5) (1,5) 형태로 입력받습니다.
    :param sigma: Gaussian 분포에 사용될 표준편차입니다. shape가 커지면 sigma도 커지는게 좋습니다.
    :return: shape 형태의 Gaussian kernel
    '''
    #Gaussian kernel 생성 코드를 작성해주세요.
    # gaus는 numpy 행렬인듯.
    gaus = np.ones((shape[0],shape[1]))
    middleX = shape[0]//2
    middleY = shape[1]//2
    # print(middleX, middleY)
    for i in range(shape[0]):
        for j in range(shape[1]):
            # gaus[i][j] = (i-middleX)+(j-middleY)
            # gaus[i][j] = np.exp(-1 * (np.square((i))+ np.square(j)) / (2 * np.square(sigma)))
            gaus[i][j] = np.exp(-1 * (np.square((i-middleX)) + np.square(j-middleY)) / (2 * np.square(sigma)))
            # gaus[i][j] = np.exp(-1*((i-middleX)^2+(j-middleY)^2)/(2*sigma^2))
    # print (gaus)
    return gaus
```

가우스 필터용 커널을 어떻게 했나 보자. 그냥 가우스 함수의 x,y 변수 부분이 index로 교체되었다고 보면 된다. 문제는 커널 같은 경우 정 중앙의 x,y값이 0,0 이 되어야 하지만 그냥 index 자체를 그대로 x,y로 대입하면 맨 왼쪽 위가 x,y가 0,0이 될 것이다. 이를 방지하기 위하여 커널 크기의 중간값을 구해 정 중앙의 index 의 x,y 가 0,0이 오도록 만들었다.

```python
def gaus_filtering(img, kernel, boundary = 0):
    '''
    :param img: Gaussian filtering을 적용 할 이미지
    :param kernel: 이미지에 적용 할 Gaussian Kernel
    :param boundary: 경계 처리에 대한 parameter (0 : zero-padding, default, 1: repetition, 2:mirroring)
    :return: 입력된 Kernel로 gaussian filtering이 된 이미지
    '''
    row, col = len(img), len(img[0])
    ksizeY, ksizeX = kernel.shape[0], kernel.shape[1]
    pad_image = my_padding(img, (ksizeY, ksizeX), boundary = boundary) # 경계가 padding된 이미지 생성.
    filtered_img = np.zeros((row,col), dtype = np.float32)
    # print(pad_image)
    # print("pad_image.shape[0] " + str(pad_image.shape[0]))
    # print("pad_image.shape[1] " + str(pad_image.shape[1]))
    # print(ksizeY, ksizeX)
    # # print(filtered_img)
    # print(row, col)
    # print(filtered_img.shape[0],filtered_img.shape[1])
    # print(filtered_img[2999][3999])
    # print(kernel.sum())
    # print(part.shape[0],part.shape[1])
    # print(pad_image[500-ksizeY//2:500+ksizeY//2][700-ksizeX//2:700+ksizeX//2])
    print(kernel.shape[0],kernel.shape[1])
    for i in range(row):
        # print(i)
        for j in range(col):
            #filtering 부분을 작성해주세요.
            # filtered_img[i][j] = pad_image[i-ksizeY//2:i+ksizeY//2,j-ksizeX//2:j+ksizeX//2] * kernel
            # filtered_img[i][j] = pad_image[i-ksizeY//2:i+ksizeY//2,j-ksizeX//2:j+ksizeX//2] * kernel
            # temp = pad_image[i:i + ksizeY, j:j + ksizeX]
            # temp2 = temp * (kernel / kernel.sum())
            # filtered_img[i][j] = np.round(temp2.sum())
            filtered_img[i][j] = np.round(((pad_image[i:i + ksizeY, j:j + ksizeX]) * (kernel / kernel.sum())).sum())
            # filtered_img[i][j] = pad_image[i:i+ksizeY,j:j+ksizeX] * kernel
```

가우스 함수 필터 정의 부분을 보면 일단 원본 이미지에 padding을 넣어 준 pad_image를 가우스 함수를 이용해 만든 커널 행렬과 연산해준다. 왜 padding을 해야 하냐면 만약 원본 이미지를 그대로 가져와서 index가 (0,0) 인 부분을 필터한다고 하자. 근데 만약 커널 크기가 51이면 index가 (-25,-25)~(25,25) 만큼이 필요하고 원본에선 - index가 없기 때문에 오류가 날 게 뻔하다. 그러므로 여유분의 공간을 만든 것이다.
여유분의 pad_image와 kernel과의 연산이 끝나면 단 하나의 숫자만 남을 것이다. 이 값을 내보내려는 이미지의 index값에 넣는다. 이 것을 for문을 통해 원래 이미지의 크기만큼 반복한다.

그래서 나온 결과물을 보면

왼쪽이 1d로 한거고 오른쪽이 2d로 한 거다. 결과물은 똑같다.

```
C:₩pythonenv₩Scripts₩python.exe E:/PycharmProjects/ComputerGraphics/my_filtering.py
1 101
101 1
57.6491271
101 101
89.3441306
```

커널 크기를 (101,101) 로 했을 때 걸린 시간은 1d로 나눠서 한 것이 2d로 한 거보다 더 적다. 이는 작은 이미지를 하든 큰 이미지를 하든 같았다.
하지만 커널 크기가 충분히 크지 않았을 때는 오히려 1d로 나눠서 한 것이 더 느렸다. 밑에가 그 결과 사진이다.

```
my_filtering ×
C:₩pythonenv₩Scripts₩python.exe E:/PycharmProjects/Compute
136.3912597
153.00639500000003

Process finished with exit code 0
```

커널 크기를 (51,51)로 하고 조금 더 뚜렷한 비교를 위해 위에서 했던 이미지보다 좀 더 큰걸로 계산한 결과다. 136.391... 이 2d로 계산했을 때고 153.006... 이 1d로 두 번 나눠서 계산했을 때의 결과다.
그래서 커널 크기가 충분히 클 경우 1d로 나눠서 두 번 계산하는 것이 더 빠르다. 이는 2n과 n^2 과의 차이에서도 나오는 현상이다. n이 작을 경우 n^2이 더 작지만 n이 커질 경우 2n이 더 작아지는 것이 그 이유일 것이다.

느낀 점
어려어려어려어려워라.

과제 난이도
어렵다.