1. Image Rotation

제공 코드명: rotate_skeleton_v2.cpp

코드 목적:

Nearest-neighbor interpolation 과 Bilinear interpolation 을 통해 Image Rotation 을 c++로 구현한다. 주어진 이미지("lena.jpg")를 반시계 방향으로 45 도 rotate 한 결과를 새 창에 띄워 출력한다.

코드 흐름:

1. 이미지 파일을 읽는다.
2. input 이미지파일, rotate 각도 (45), rotation option을 매개변수로 하는 함수 myrotate 호출
3. 결과 이미지 새 창에 출력

함수 설명:

myrotate:

매개변수: const Mat input, float angle, const char* opt

input: 입력 이미지 행렬

angle: rotate 할 각도

opt: interpolation 옵션. nearest/bilinear 사용가능.

함수 목적: input 을 angle 만큼, opt 방식으로 rotate 한 행렬 output 를 리턴.

코드 설명:

1)

```
int row = input.rows; int col = input.cols;
float sq_row = ceil(row * sin(radian) + col * cos(radian));
float sq_col = ceil(col * sin(radian) + row * cos(radian));
```

rotation 을 하면 이미지의 크기가 커진다. row 와 col 을 가로와 세로로 하는 사각형을 θ 만큼 회전하였을 때의 테두리 직사각형의 가로세로 길이를 계산해 결과물의 col row 를 구해준다. 행렬의 인덱스는 정수이기 때문에 ceiling 으로 실수를 올림해준다.

2)

```
Mat output = Mat::zeros(sq_row, sq_col, input.type());
```

sq_row 을 행, sq_col 을 열, input.type()을 자료형으로 하는 행렬을 0 으로 초기화한다

3) rotate 된 이미지의 모든 비트에 대해,

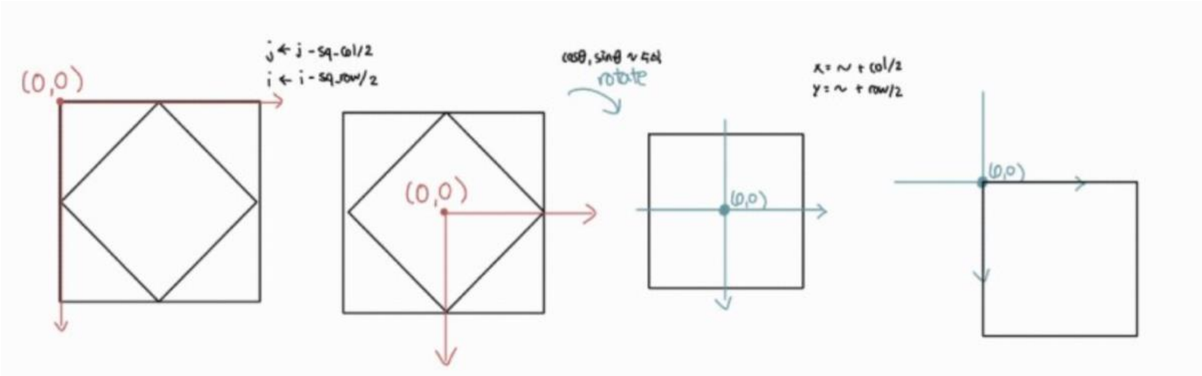
```
float x = (j - sq_col / 2) * cos(radian) - (i - sq_row / 2) * sin(radian) + col / 2;
float y = (j - sq_col / 2) * sin(radian) + (i - sq_row / 2) * cos(radian) + row / 2;
```

rotate 하기 전 좌표를 (x′, y′), rotate 후의 좌표를 (x_R, y_R)라 하자. (x_R, y_R)는 (x′,y′)를 rotate 한 점으로, 둘의 색상 정보는 같다. 즉, f(x_R, y_R)를 구하기 위해서는 (x_R, y_R)에 대응하는 점인 (x′,y′)을 찾은 뒤 f(x′,y′)값을 f(x_R, y_R)에 넣어주면 된다. target 의 원하는 위치의 색에 관한 데이터를 찾기 위해 이에 대응하는 source 의 (x,y)를 찾는것이 Inverse warping 이다.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x_R \\ y_R \end{bmatrix}$$

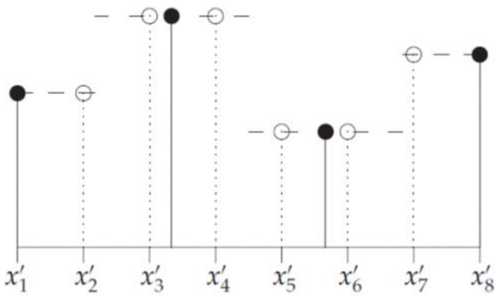
이 행렬에 따라, $x' = \cos\theta\, x_R - \sin\theta\, y_R$, $y' = \sin\theta\, x_R + \cos\theta\, y_R$ 이므로, I_R의 i 행 j 열 (x 좌표 j, y 좌표 i) 을 x_R, y_R에 대입해 x', y'를 구할 수 있다.

이미지에서 왼쪽 위를 원점으로 하기 위해 수식이 추가된다. j 에 (sq_col / 2)를, i 에 (sq_row / 2)를 빼 원점을 I_R의 한가운데로 만든 뒤, 코사인, 사인을 통해 45 도 돌려준다. 이때까지 원점은 rotate 전의 이미지의 가운데이다. 마지막으로 x 축으로 (col / 2), y 축으로 (row / 2)만큼 이동해 원점을 왼쪽 위 꼭짓점으로 만든다.



3) 데이터 복사- "nearest"

Nearest-neighbor interpolation 은 가까운 점의 값을 복사 붙여넣기한다.



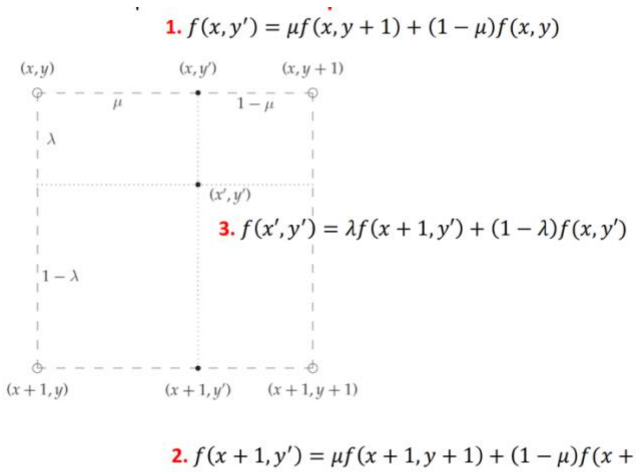

```
output.at<Vec3b>(i,j)=input.at<Vec3b>(round(x),round(y))
```

x, y(I_R의 점에 대응하는 I의 좌표) 를 반올림하여 가장 가까운 정수를 찾는다. 행렬 I의 정수 인덱스에만 값이 저장되기 때문이다.

행렬의 요소에는 at 함수로 접근한다. "행렬이름.at<자료형>(행,열)." 좌표 (x,y)로 접근하고 싶다면 at<자료형>(y,x)로 접근해야 한다.

4) 데이터 복사- "bilinear"

Bilinear interpolation 은 양쪽의 정수인덱스를 가진 비트의 f 값을 떨어진 거리의 비율에 따라 적용한다.



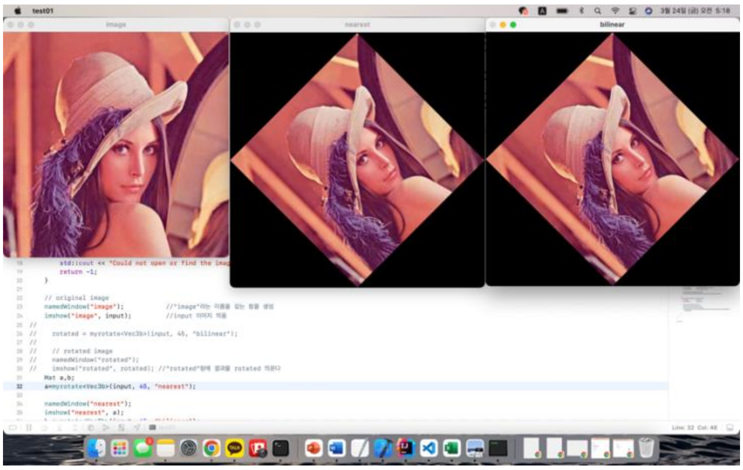

```
float mu=x-(int)x;                    //mu는 x의 소수부분
float lambda=y-(int)y;                //lambda는 y의 소수부분
output.at<Vec3b>(i,j)=(1-lambda)*(mu*input.at<Vec3b>(y,x+1) + (1-mu) * input.at<Vec3b>(y,x))
        + lambda*(mu*input.at<Vec3b>(y+1,x+1)+(1-mu)*input.at<Vec3b>(y+1,x));
```

실행 결과:

두 interpolation 옵션 모두 확인하기 위해 출력형태를 바꾸었다.

```
Mat n,b;
n=myrotate<Vec3b>(input, 45, "nearest");
namedWindow("nearest");
imshow("nearest", n);

b=myrotate<Vec3b>(input, 45, "bilinear");
namedWindow("bilinear");
imshow("bilinear", b);
```



2. Image Stitching

제공 코드명: stitching.cpp

코드 목적: 두 이미지 파일에 corresponding pixels 이 존재하고, 그 정보가 모두 제공되었다. I_2를 affine transform 하여 두 이미지가 이어지도록 합친다.

코드 흐름:

1. 두 이미지 파일(I_1, I_2)을 읽는다.
2. cal_affine 함수를 통해 A_{12}, A_{21} Matrix 를 구한다.
3. A_{21} 을 이용해 I_2'의 꼭짓점 p1-p4 를 계산한다.
4. boundary 를 구한다. (I_1, p1-p4 고려)
5. 행렬 I_f에 I_2'를 그린다.
6. blend_stitchong 함수를 통해 I_f에 I_1를 그린다. (I_2'도 고려)
7. 결과 이미지 새 창에 출력 후 파일로 저장

함수 설명:

cal_affine:

매개변수: int ptl_x[], int ptl_y[], int ptr_x[], int ptr_y[], int number_of_points

ptl_x[]: corresponding pixels 의 왼쪽 이미지에서의 x 좌표

ptl_y[]: corresponding pixels 의 왼쪽 이미지에서의 y 좌표

ptr_x[]: corresponding pixels 의 오른쪽 이미지에서의 x 좌표

ptr_y[]: corresponding pixels 의 오른쪽 이미지에서의 y 좌표

number_of_points: corresponding pixels 의 개수

함수 목적: ptl_x, ptl_y 와 계산해 ptr_x, ptr_y 를 구할 수 있는 Matrix(A_{12}, A_{21}) 반환

$$\begin{bmatrix} \mathrm{ptr\_x} \\ \mathrm{ptr\_}y \end{bmatrix} = A \begin{bmatrix} \mathrm{ptl\_x} \\ \mathrm{ptl\_y} \\ 1 \end{bmatrix}$$

blend_stitchong:

매개변수: const Mat I1, const Mat I2, Mat &I_f, int bound_l, int bound_u, float alpha

I_1: I_1이미지 데이터 담은 행렬. 이 함수를 통해 I_f에 I_1데이터 복사할 것이다.

I_2: I_2이미지 데이터 담은 행렬

I_f : 결과 이미지 담을 행렬. 현재 I_2'만 그려져 있고, 나머지 부분은 검정색

bound_l: I_f의 왼쪽 경계선

bound_u: I_f의 위 경계선

alpha: I_1와 I_2 blend 할 비율. 0.5

함수 목적: I_2'만 그려져 있는 I_f에 I_1데이터를 추가한다. I_1이미지의 범위에 I_2'가 이미 있다면 α를 고려해 blend 하고, 아무것도 없다면 I_1를 그대로 그려준다.

코드 설명:

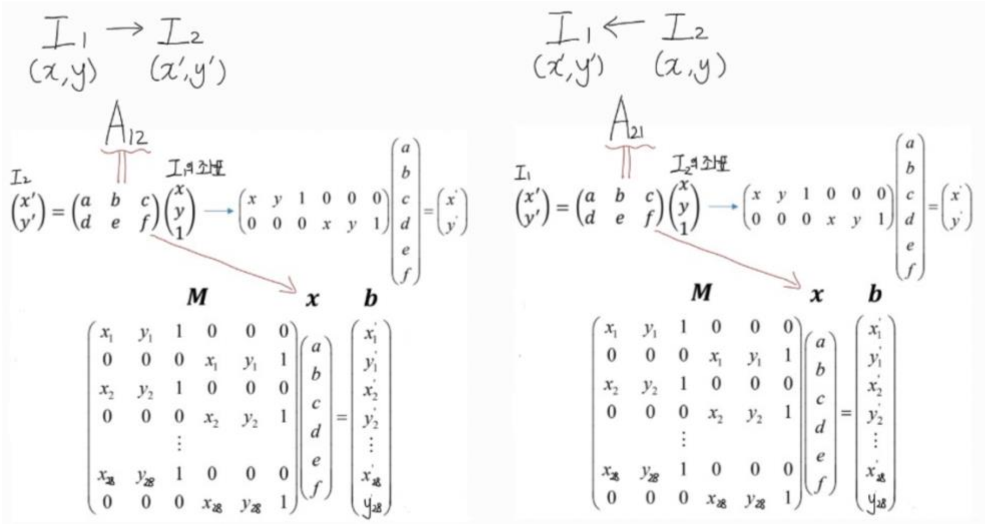
1)

```
I1.convertTo(I1, CV_32FC3, 1.0 / 255);
```

값을 CV_32FC3(32 비트 float)로 전환, 데이터(색 정보)를 255 로 나누어 (0.0 ~ 1.0) 사이의 값으로 변경한다.

2)

```
Mat A12 = cal_affine<float>(ptl_x, ptl_y, ptr_x, ptr_y, 28);
Mat A21 = cal_affine<float>(ptr_x, ptr_y, ptl_x, ptl_y, 28);
```



$$\boldsymbol{Mx} = \boldsymbol{b} \rightarrow \boldsymbol{x} = (\boldsymbol{M}^{\mathrm{T}}\boldsymbol{M})^{-1}\boldsymbol{M}^{\mathrm{T}}\boldsymbol{b}$$

I_1에 A_{12}곱하면 대응하는 I_2위의 좌표를 구할 수 있다.

A_{12}, A_{21} 구하는 과정은 위의 사진과 같다. M 과 b 를 만들어 x 를 구한다.

```
Mat cal_affine(int ptl_x[], int ptl_y[], int ptr_x[], int ptr_y[], int number_of_points) {
        Mat M(2 * number_of_points, 6, CV_32F, Scalar(0));
        Mat b(2 * number_of_points, 1, CV_32F);

        Mat M_trans, temp, affineM;
for (int i = 0; i < number_of_points; i++) {
  M.at<T>(2 * i, 0) = ptl_x[i];     M.at<T>(2 * i, 1) = ptl_y[i];        M.at<T>(2 * i, 2) = 1;
  M.at<T>(2*i+1,3) = ptl_x[i];     M.at<T>(2 * i + 1, 4) = ptl_y[i];     M.at<T>(2 * i + 1, 5) = 1;
  b.at<T>(2 * i) = ptr_x[i];          b.at<T>(2 * i + 1) = ptr_y[i];
} //위 그림대로 값 삽입
transpose(M, M_trans);
invert(M_trans * M, temp);
affineM = temp * M_trans * b;

return affineM;
```

cal_affine 함수는 위의 그림대로 행렬 M 과 b 에 값을 집어 넣은 뒤, 행렬의 transpose, invert, * 연산을 통해 원하는 x, A_{12} 또는 A_{21}를 반환한다.

3) A_{21} 을 이용해 I_2'의 꼭짓점 p1~p4 계산: forwarding warping

위의 2)를 통해 A_{21}를 구했다. A_{21}를$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$ 라고 하고,

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$ 이를 계산해보면,

$x' = ax + b\,y + c$, $y' = dx + e\,y + f$ 이다. (이 때, (x,y)는 I_2위의 점, (x',y')는 I_1위의 점)

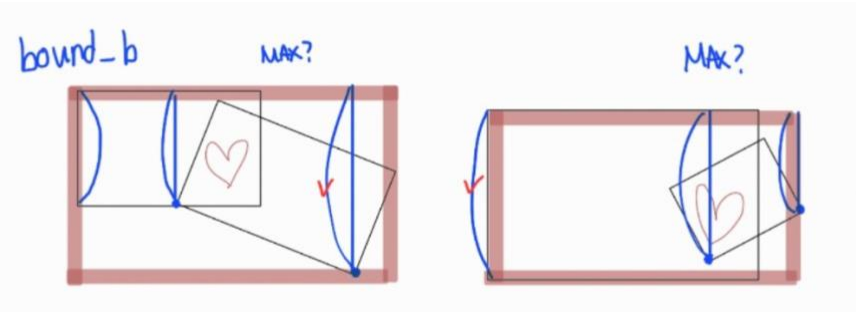
a= A21.at<float>(0), b= A21.at<float>(1) ... f=A21.at<float>(5)으로 접근 가능하다. 주어진 I_2의 점 p1: (0,0) , p2: (row, 0) , p3: (row, col) , p4: (0, col)을 위의 식의 x와 y에 대입해 I_1에서 어느 점과 대응하는지 그 좌표를 구할 수 있다. 주어진 좌표를 직접 계산해 이동할 위치의 좌표를 알아내는 것이 forwarding warping이다.

즉, I_2의 꼭짓점(p_{nx}, p_{ny})를 I_1의 corresponding pixels에 놓으면 그 좌표는 $p_n(\,a\,p_{ny} + b\,p_{nx} + c\,, d\,p_{ny} + e\,p_{nx} + f\,)$ 이다.

여기서 주어진 p1~p4의 값은 (x좌표 값, y좌표 값)이 아닌 (행, 열)이기에 주의해야 한다.

```
Point2f p1(A21.at<float>(0) * 0 + A21.at<float>(1) * 0 + A21.at<float>(2),
                    A21.at<float>(3) * 0 + A21.at<float>(4) * 0 + A21.at<float>(5));
Point2f p2(A21.at<float>(0) * 0 + A21.at<float>(1) * I2_row + A21.at<float>(2),
                    A21.at<float>(3) * 0 + A21.at<float>(4) * I2_row + A21.at<float>(5));
Point2f p3(A21.at<float>(0) * I2_col + A21.at<float>(1) * I2_row + A21.at<float>(2),
                    A21.at<float>(3) * I2_col + A21.at<float>(4) * I2_row + A21.at<float>(5));
Point2f p4(A21.at<float>(0) * I2_col + A21.at<float>(1) * 0 + A21.at<float>(2),
                    A21.at<float>(3) * I2_col + A21.at<float>(4) * 0 + A21.at<float>(5));
```

4) I_f의 경계(크기) 정하기



위의 예시만 봐도 알 수 있듯이 두 사진의 크기와 위치에 따라 결과물의 경계(크기)가 다르다. 하나하나 잘 고려해 정해야 한다.

위: 원점(0)과 I2'의 꼭짓점 중 가장 위(y 값 작은것)의 값

아래: I1 의 row(=y 값)과 I2'의 꼭짓점 중 가장 아래(y 값 큰것)의 값

왼쪽: I1 의 원점(0)과 I2'의 꼭짓점 중 가장 왼쪽(x 값 작은것)의 값

오른쪽: I1 의 col(=x 값)과 I2'의 꼭짓점 중 가장 오른쪽(x 값 큰것)의 값

```
int bound_u = (int)round(min(0.0f, min(p1.y, p4.y)));
int bound_b = (int)round(max(I1_row-1, max(p2.y, p3.y)));
int bound_l = (int)round(min(0.0f, min(p1.x, p2.x)));
int bound_r = (int)round(max(I1_col-1, max(p3.x, p4.x)));
```

5) A_{12}이용해 I_f에 $I_2{}'$ 그리기: Inverse warping

```
for (int i = bound_u; i <= bound_b; i++) {
   for (int j = bound_l; j <= bound_r; j++) {
      float x = A12.at<float>(0) * j + A12.at<float>(1) * i + A12.at<float>(2) - bound_l;
      float y = A12.at<float>(3) * j + A12.at<float>(4) * i + A12.at<float>(5) - bound_u;

      float y1 = floor(y);      float x1 = floor(x);
      float y2 = ceil(y);        float x2 = ceil(x);

      float mu = y - y1;      float lambda = x - x1;

      if (x1 >= 0 && x2 < I2_col && y1 >= 0 && y2 < I2_row)
         I_f.at<Vec3f>(i - bound_u, j - bound_l) =
          lambda * (mu * I2.at<Vec3f>(y2, x2) + (1 - mu) * I2.at<Vec3f>(y1, x2)).
            + (1 - lambda) *(mu * I2.at<Vec3f>(y2, x1) + (1 - mu) * I2.at<Vec3f>(y1, x1));
   }
}
```

I_f에 그리고 싶으므로, for문으로 I_f의 모든 픽셀에 접근한다.
A_{12}를 이용하면, $I_1(I_f)$의 점(j,i)와 대응하는 I_2의 점의 좌표를 알 수 있다.
$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$ $x' = aj + b\,i + c\ ,\ y' = dj + e\,i + f$ 이다. (이 때, (j,i)는 I_1위의 점, (x',y')는 I_2위의 점)
a= A12.at<float>(0), b= A12.at<float>(1) … f=A12.at<float>(5)으로 접근 가능하므로 대입하여 I_2의 좌표를 계산한다.
계산한 x와 y는 float 자료형이다. 이미지는 행렬에 저장되어, 정수 인덱스로 접근 가능하므로, 가까운 4개의 점과의 거리 비율을 계산하여 값을 구하는 bilinear interpolation를 진행한다.
I_1에 그림을 그리기 위해 대응하는 I_2의 좌표를 찾고, 데이터를 복사해오는 Inverse warping이다.

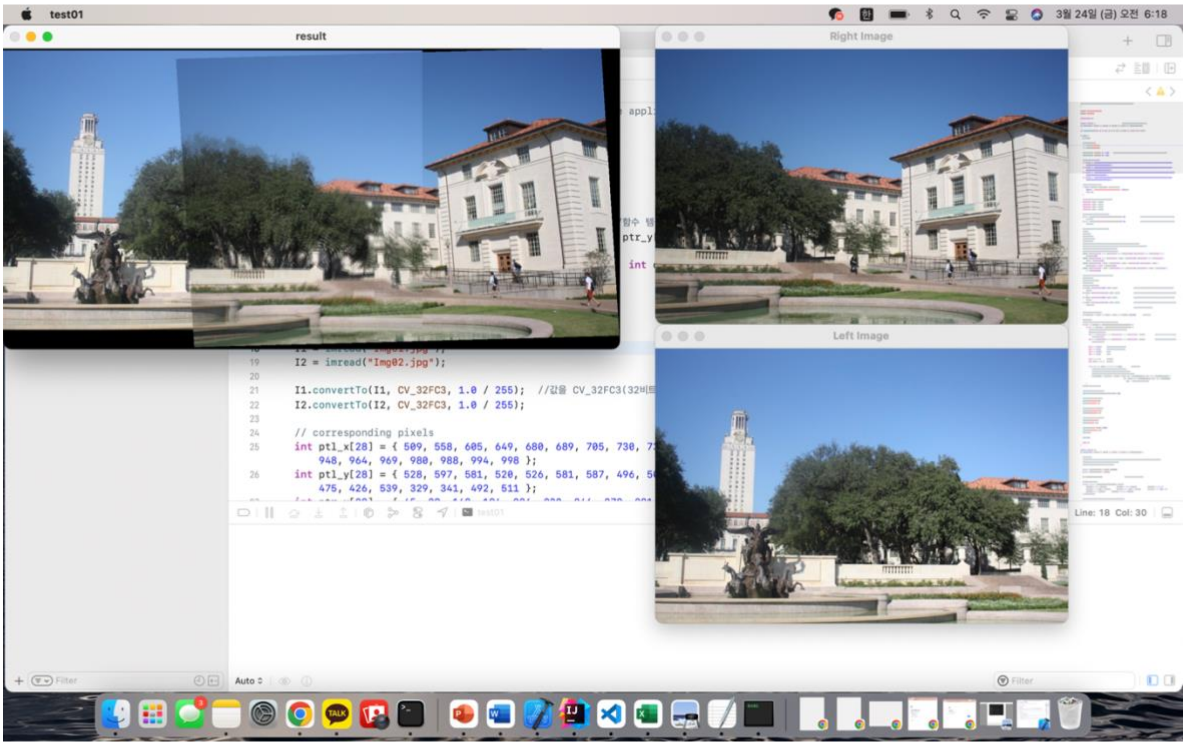
6) blend_stitchong 함수를 통해 I_f에 I_1 그리기

```
blend_stitching(I1, I2, I_f, bound_l, bound_u, 0.5);
```

I_2는 이미 그려져 있으니 I_1만 고려한다. for문을 통해 I_1의 범위에만 접근한다. 해당 픽셀의 값이 (0, 0, 0), 즉 검정색이라면 I_2와 겹치지 않는 부분이므로 I_1의 값만을 써넣는다. I_1의 범위중에 검정색이 아니라면, I_2가 이미 그려져있다는 의미이므로, α=0.5를 고려해 I_1과 I_2를 모두 그려준다.

```
for (int i = 0; i < I1.rows; i++) {
   for (int j = 0; j < I1.cols; j++) {
      bool cond_I2 = I_f.at<Vec3f>(i - bound_u, j - bound_l) != Vec3f(0, 0, 0) ? true : false;
      if (cond_I2)
         I_f.at<Vec3f>(i - bound_u, j - bound_l)
            = alpha * I1.at<Vec3f>(i, j) + (1 - alpha) * I_f.at<Vec3f>(i - bound_u, j - bound_l);
      else
         I_f.at<Vec3f>(i - bound_u, j - bound_l) = I1.at<Vec3f>(i, j);
   }
}
```

실행 결과:



참고자료:

오픈 SW 프로젝트 Lec02 수업자료