1. stitching.cpp

Result:

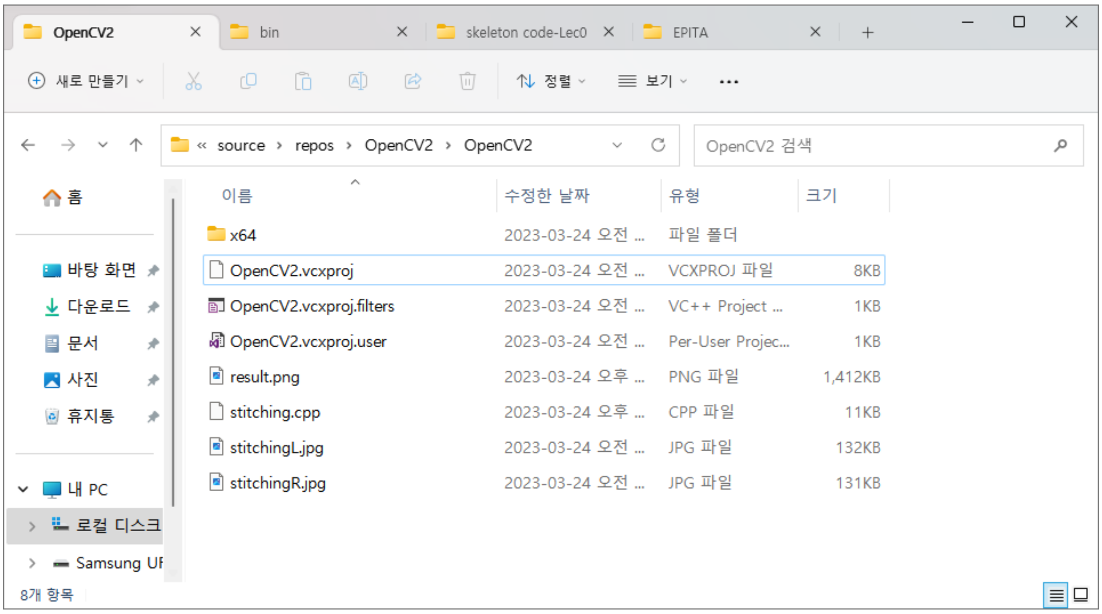

Image stitching using Affine Transformation

1. Estimate the affine transform

   For 28 pairs of corresponding pixels, affine transform for I1 → I2 can be computed as follows.

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

```
template <typename T>
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;

    // initialize matrix
    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];
    }

    // (M^T * M)^(-1) * M^T * b ( * : Matrix multiplication)
    transpose(M, M_trans);
    invert(M_trans * M, temp);
    affineM = temp * M_trans * b;

    return affineM;
}
```

   Similarly, affine transform for I2 → I1 can be obtained.

2. Merge two images

   A. The size of a final merged image I_f can be estimated by computing p1, p2, p3, p4 using A21.

```
// compute corners (p1, p2, p3, p4)
// p1: (0,0)
// p2: (row, 0)
// p3: (row, col)
// p4: (0, col)
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));

// compute boundary for merged image(I_f)
// bound_u <= 0
// bound_b >= I1_row-1
// bound_l <= 0
// bound_b >= I1_col-1
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)));

// initialize merged image
Mat I_f(bound_b - bound_u + 1, bound_r - bound_l + 1, CV_32FC3, Scalar(0));
```

   B. Perform the inverse warping using A12 within the region consisting of corners [p1, p2, p3, p4]

```
// inverse warping with bilinear interplolation
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 y2 = ceil(y);
        float x1 = floor(x);
        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));
    }
}
```

C. Blend two images I1 and I2

```
void blend_stitching(const Mat I1, const Mat I2, Mat &I_f, int bound_l, int bound_u, float alpha) {

    int col = I_f.cols;
    int row = I_f.rows;

    // I2 is already in I_f by inverse warping
    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);

        }
    }
}
```

2. rotate_skeleton_v2.cpp

Result:

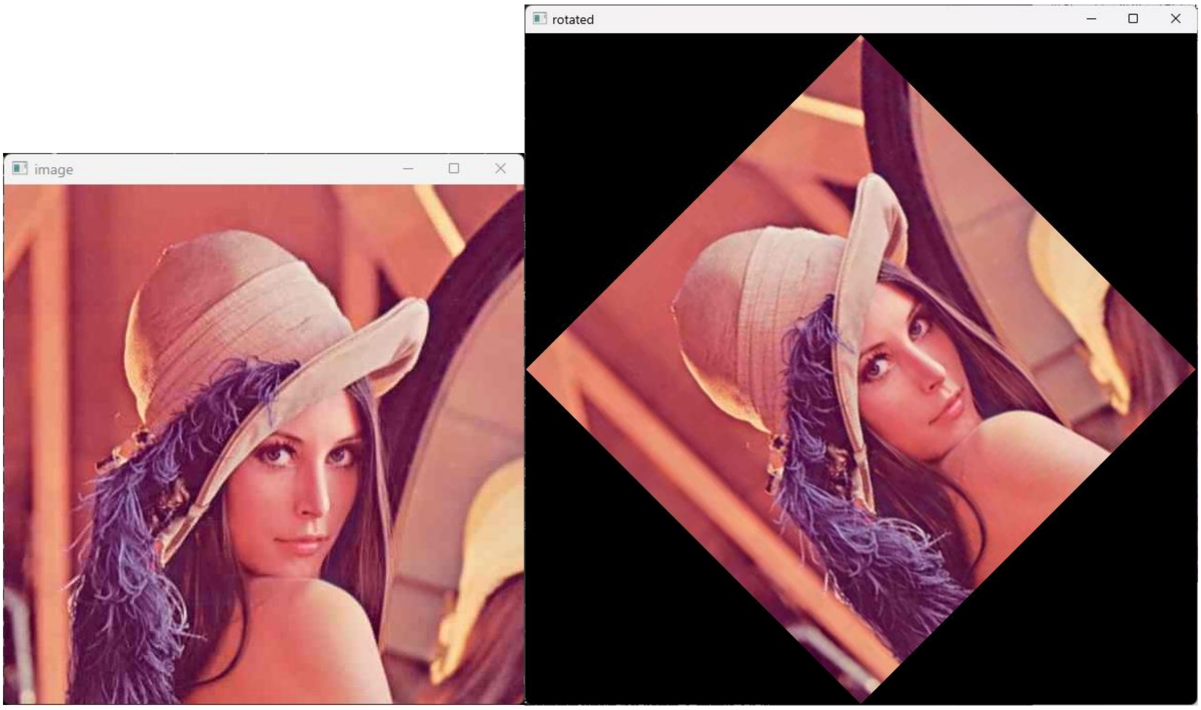

Image Rotation

1. Define an enlarged, rotated image 'rotated' class whose size is larger than that of an original image.

```
int row = input.rows;
int col = input.cols;

float radian = angle * CV_PI / 180;

float sq_row = ceil(row * sin(radian) + col * cos(radian));
float sq_col = ceil(col * sin(radian) + row * cos(radian));

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

2. Compute the inverse warping

```
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;
```

3. Compute f(x, y) using the interpolation technique.(nearest neighbor, bilinear interpolation), then f(x_rotated, y_rotated) = f(x_original, y_original)

```
for (int i = 0; i < sq_row; i++) {              //inverse warping
    for (int j = 0; j < sq_col; j++) {          //rectangle 범위 내에서
        float x = (j - sq_col / 2) * cos(radian) - (i - sq_row / 2) * sin(radian) + col / 2;   //x는 inverse warping해서 구해진 원래 image의 위치값(실수)
        float y = (j - sq_col / 2) * sin(radian) + (i - sq_row / 2) * cos(radian) + row / 2;   //y는 inverse warping해서 구해진 원래 image의 위치값(실수)

        if ((y >= 0) && (y <= (row - 1)) && (x >= 0) && (x <= (col - 1))) {   //x, y가 input image안의 값이면
            if (!strcmp(opt, "nearest")) {    //interpolation option이 bilinear이면
                int x0 = (int)round(x);       //위에서 구한 실수값을 반올림해서 가장 가까운 x좌표 찾기
                int y0 = (int)round(y);       //위에서 구한 실수값을 반올림해서 가장 가까운 y좌표 찾기

                output.at<Vec3b>(i, j) = input.at<Vec3b>(y0, x0);   //nearest neighbor 방식으로 (i, j)좌표에 있는 픽셀의 B, G, R 데이터 반환

            }
            else if (!strcmp(opt, "bilinear")) {    //interpolation option이 bilinear이면
                float y1 = floor(y);      //위에서 구한 실수값에서 내림한 y좌표
                float y2 = ceil(y);       //위에서 구한 실수값에서 올림한 y좌표
                float x1 = floor(x);      //위에서 구한 실수값에서 내림한 x좌표
                float x2 = ceil(x);       //위에서 구한 실수값에서 올림한 x좌표

                float mu = y - y1;        //실수 y의 소수부
                float lambda = x - x1;    //실수 x의 소수부

                output.at<Vec3b>(i, j) = lambda * (mu * input.at<Vec3b>(y2, x2) + (1 - mu) * input.at<Vec3b>(y1, x2)) +
                    (1 - lambda) * (mu * input.at<Vec3b>(y2, x1) + (1 - mu) * input.at<Vec3b>(y1, x1));
                //bilinear interpolation 방식으로 (i, j)좌표에 있는 픽셀의 B,G,R데이터 반환

            }
        }
        else            //x, y가 input image 안의 좌표가 아니면
            continue;   //다음좌표로 넘어감
    }
}
```