Computer Vision Assignment Report

Assignment 4 – Image morphing

2019742071 – 이민규

1. 과제 개요

1. : **Mesh warping**

- 이미지의 특징점을 이용하여 삼각형의 메쉬구조를 완성
- 이미지의 특징점의 평균값으로 이미지를 affine transform

2. : **Cross - dissolve**

- (a, b, r) 투표방법
- 그래디언트 기반 방법

2. 과제 수행 방법

1. : Triangulation

```cpp
// 삼각형
struct Triangle
{
    Point2f p1, p2, p3;
    // 포인트가 삼각형 내부에 포함되어 있는지 확인
    bool contains(Point2f pt) const;
    // 포인트가 삼각형의 외접원에 포함되어 있는지 확인
    bool circum_circle_contains(Point2f pt) const;
    // 삼각형이 동일한지 비교
    bool operator==(const Triangle& other) const;
};

bool Triangle::contains(Point2f pt) const
{   // 포인트가 삼각형 내부에 포함되어 있는지 확인

    float d1, d2, d3; // 입력된 점을 삼각형의 각 변을 기준으로 방향값을 계산해 저장한다
    bool has_neg, has_pos;

    d1 = (pt.x - p2.x) * (p1.y - p2.y) - (p1.x - p2.x) * (pt.y - p2.y);
    d2 = (pt.x - p3.x) * (p2.y - p3.y) - (p2.x - p3.x) * (pt.y - p3.y);
    d3 = (pt.x - p1.x) * (p3.y - p1.y) - (p3.x - p1.x) * (pt.y - p1.y);

    has_neg = (d1 < 0) || (d2 < 0) || (d3 < 0);
    has_pos = (d1 > 0) || (d2 > 0) || (d3 > 0);

    return !(has_neg && has_pos);
    // 만약 d1, d2, d3 중에 음수와 양수가 모두 있다면, 점 pt는 삼각형 내부에 있지 않다
    // 삼각형 내부에 있는 점은 삼각형의 세 변에 대해 모두 같은 방향에 있어야 한다
    // 따라서, !(has_neg && has_pos)가 참이면 점 pt는 삼각형 내부에 있다
}

bool Triangle::circum_circle_contains(Point2f pt) const
{
    // 포인트가 삼각형의 외접원에 포함되어 있는지 확인
    float ab = p1.x * p1.x + p1.y * p1.y;
    float cd = p2.x * p2.x + p2.y * p2.y;
    float ef = p3.x * p3.x + p3.y * p3.y;

    float circum_x = (ab * (p3.y - p2.y) + cd * (p1.y - p3.y) + ef * (p2.y - p1.y)) /
        (p1.x * (p3.y - p2.y) + p2.x * (p1.y - p3.y) + p3.x * (p2.y - p1.y)) / 2.f;
    float circum_y = (ab * (p3.x - p2.x) + cd * (p1.x - p3.x) + ef * (p2.x - p1.x)) /
        (p1.y * (p3.x - p2.x) + p2.y * (p1.x - p3.x) + p3.y * (p2.x - p1.x)) / 2.f;
    float circum_radius = sqrt((p1.x - circum_x) * (p1.x - circum_x) + (p1.y - circum_y) * (p1.y - circum_y)
    // 외접원의 중심점 좌표, 반지름 계산
    float dist = sqrt((pt.x - circum_x) * (pt.x - circum_x) + (pt.y - circum_y) * (pt.y - circum_y));
    // 주어진 점과 외접원의 중심과의 거리
    return dist <= circum_radius;
}

bool Triangle::operator==(const Triangle& other) const
{
    // 세 꼭짓점을 비교하여 같은 삼각형인지 판단
    return (p1 == other.p1 && p2 == other.p2 && p3 == other.p3) ||
        (p1 == other.p2 && p2 == other.p3 && p3 == other.p1) ||
        (p1 == other.p3 && p2 == other.p1 && p3 == other.p2);
}
```

< 그림1. 삼각형 메쉬 구조를 만들 기초구조체 선언 >

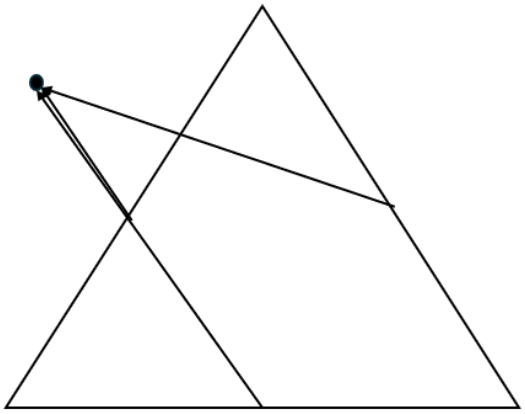
삼각형 메쉬 구조를 만들 기초적인 struct를 선언하였다. 구조체는 기본적으로 포인트가 삼각형 내부에 포함되어 있는지 확인하고, 포인트가 삼각형의 외접원에 포함이 되어있는지 확인하고 결론적으로 삼각형이 동일한지를 확인한다. 포인트가 삼각형 내부에 있는지 확인하는 방법은 다음과 같다.

1. 각 점이 삼각형의 각 변에 대해서 어떤 방향에 있는지 그 방향값을 계산해 저장한다.

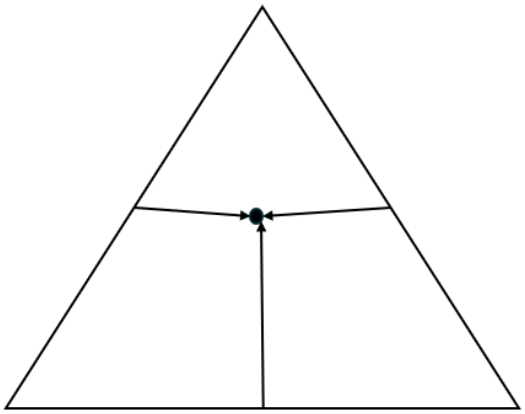
2. 만일 그 방향이 일관적이지 않다면 삼각형 내부에 없는 것이다.

```
d1 = (pt.x - p2.x) * (p1.y - p2.y) - (p1.x - p2.x) * (pt.y - p2.y);
d2 = (pt.x - p3.x) * (p2.y - p3.y) - (p2.x - p3.x) * (pt.y - p3.y);
d3 = (pt.x - p1.x) * (p3.y - p1.y) - (p3.x - p1.x) * (pt.y - p1.y);
```

위 코드는 벡터의 외적을 구현한 코드이다. 이를 이용하면 어떤 점이 삼각형의 특정방향에 있는지 확인 할 수 있다.



[Case 1]



[Case 2]

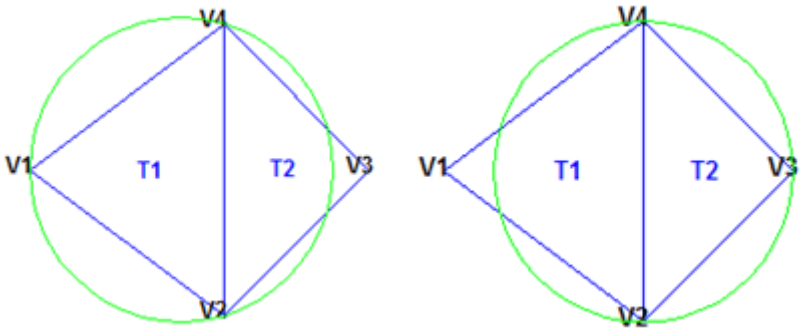
방향 d1, d2, d3를 구했을 때 그 값이 양수 음수인지를 판단한다. 삼각형 내부에 점이 있다면 case 2의 경우에는 d1, d2, d3모두 일관적인 부호를 가지고 있을 것이다. d1, d2, d3중 하나라도 부호가 다른 것이 있다면 삼각형 밖에 있는 것으로 판단한다.

이후, 삼각형의 세점을 이용해 외접원의 중심좌표와 반지름을 계산해낸다. 중심점과 점의 거리를 알면 이 점이 외접원의 내부에 있는지 외부에 있는지를 판단 할 수 있다.

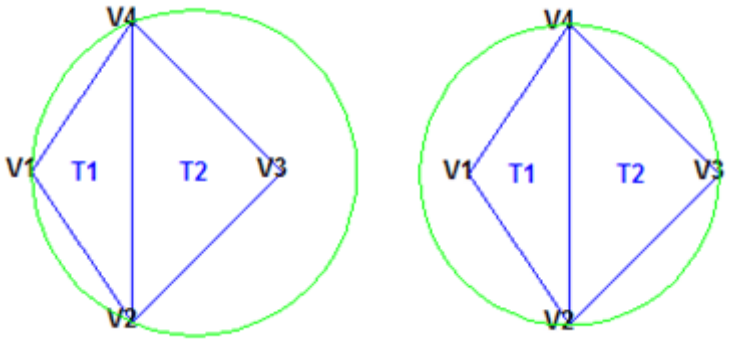
```cpp
vector<vector<int>> compute_delaunay_triangulation(vector<Point2f>& points, Rect rect) {

    // 초기 삼각형은 매우 크게설정
    float margin = 2000;
    Point2f p1(rect.x - margin, rect.y - margin);
    Point2f p2(rect.x + rect.width + margin, rect.y - margin);
    Point2f p3(rect.x + rect.width / 2.0f, rect.y + rect.height + margin);
    vector<Triangle> triangles = { { p1, p2, p3 } };

    for (const auto& pt : points)
    {
        vector<Triangle> badTriangles; // 외접원에 점 pt가 포함된 삼각형 저장
        vector<pair<Point2f, Point2f>> polygon;  // 제거된 삼각형의 변 저장 이 변을 이용해 새로운 삼각형을 생성
        // 폴리곤을 생성해서
        // 외접원에 포함여부 검사.
        for (const auto& tri : triangles)
        {
            if (tri.circum_circle_contains(pt))
            {
                badTriangles.push_back(tri);
                polygon.push_back({ tri.p1, tri.p2 });
                polygon.push_back({ tri.p2, tri.p3 });
                polygon.push_back({ tri.p3, tri.p1 });
            }
        }
        //해당 삼각형 제거
        triangles.erase(remove_if(triangles.begin(), triangles.end(), [&](const Triangle& t)
            {   // 각 삼각형 t가 제외대상인지 확인
                return find(badTriangles.begin(), badTriangles.end(), t) != badTriangles.end();
            }   // badTriangles 벡터에서 t를 찾으면, 해당 삼각형은 제거 대상
        ), triangles.end());
        // remove_if 함수가 반환한 새로운 끝부터 벡터의 실제 끝까지의 요소들을 제거

        // 삼각형 재생성.
        vector<pair<Point2f, Point2f>> uniqueEdges;
        for (auto& e1 : polygon)
        {
            bool isEdgeUnique = true;
            for (auto& e2 : polygon)
            {
                if ((e1.first == e2.second) && (e1.second == e2.first))
                {
                    isEdgeUnique = false;
                    break;
                }
            }
            if (isEdgeUnique)
                uniqueEdges.push_back(e1);
            // 다른 변과 중복되지 않는 변을 찾는다.
        }

        for (const auto& edge : uniqueEdges)
            triangles.push_back({ edge.first, edge.second, pt });
        // 새로 추가된 특징점을 삼각형의 변과 연결하여 새로운 삼각형을 만든다
    }
```

```
    // 연결된 선 제거.
    triangles.erase(remove_if(triangles.begin(), triangles.end(),
        [&](const Triangle& t)
        {
            return (t.p1 == p1) || (t.p1 == p2) || (t.p1 == p3) ||
                (t.p2 == p1) || (t.p2 == p2) || (t.p2 == p3) ||
                (t.p3 == p1) || (t.p3 == p2) || (t.p3 == p3);
        }), triangles.end());

    vector<vector<int>> delaunayTri;
    for (const auto& tri : triangles)
    {
        vector<int> indices;
        for (size_t i = 0; i < points.size(); ++i)
        {
            if (points[i] == tri.p1 || points[i] == tri.p2 || points[i] == tri.p3)
                indices.push_back(i);
        }
        if (indices.size() == 3)
            delaunayTri.push_back(indices);
    }

    return delaunayTri; // 삼각형들의 점 인덱스를 반환
}
```

< 그림2. compute_delaunay_triangulation() >



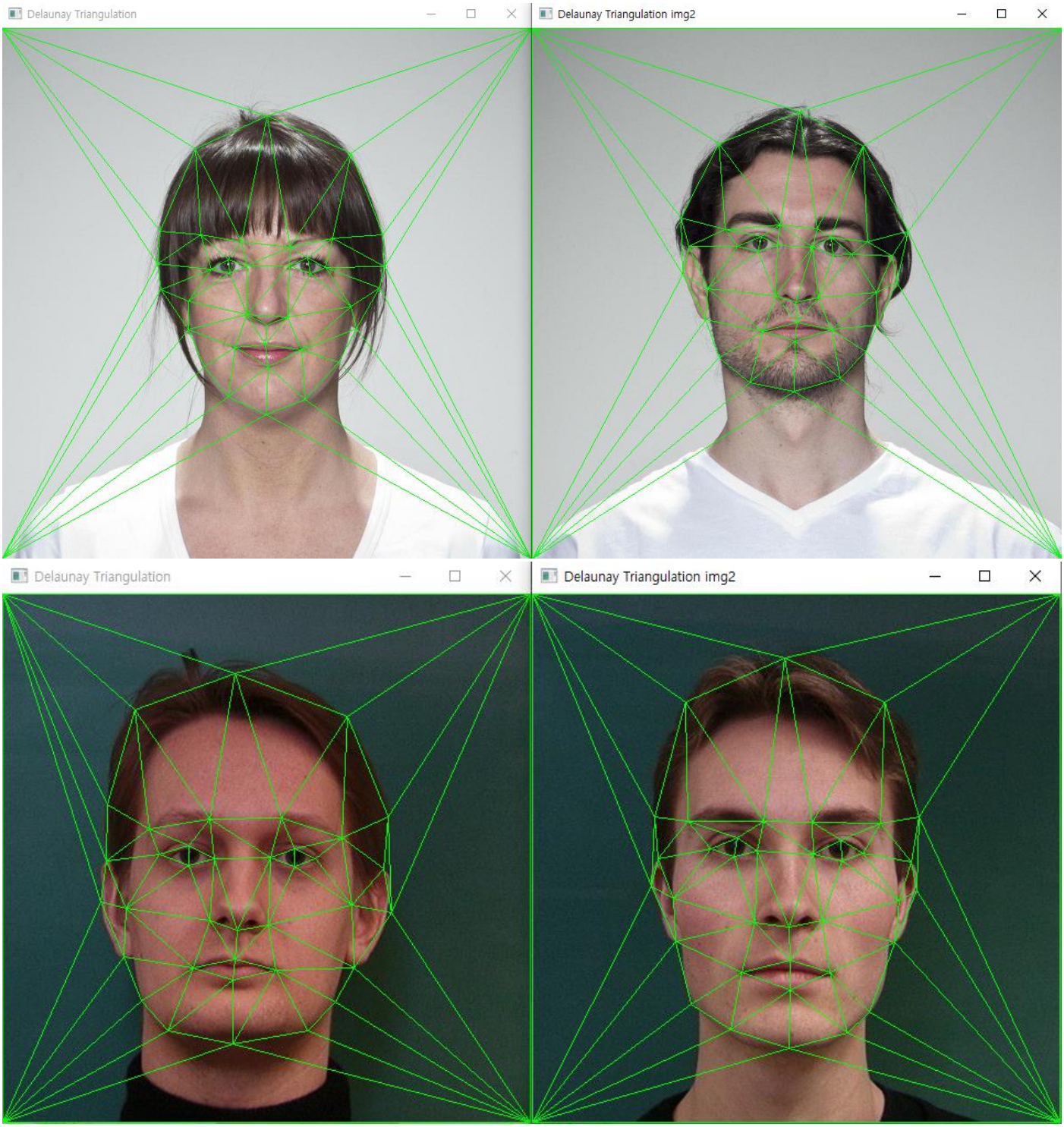
들로네 삼각분할은 위와 같이 t1, t2 삼각형에 대한 외접원을 만들었을 때, 그 외접원의 내부에 다른 점이 없어야 한다.

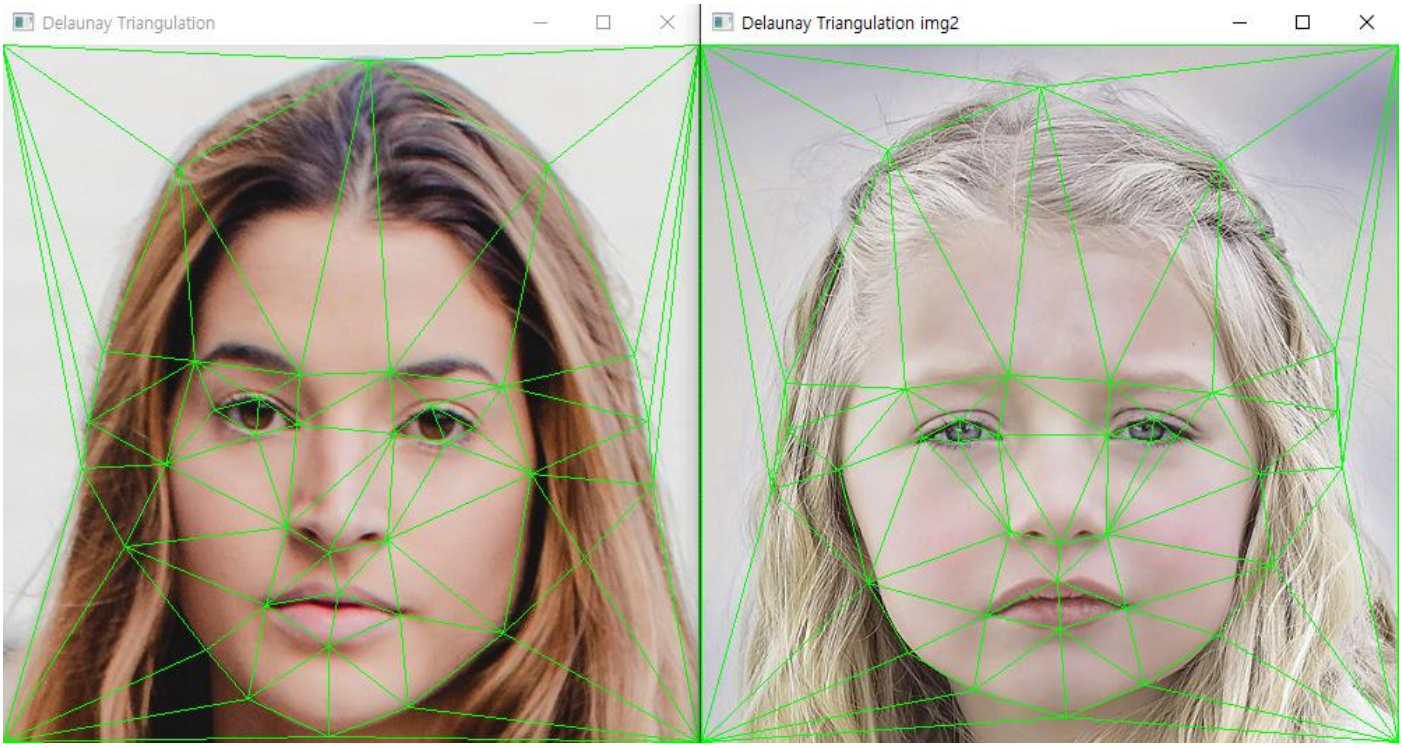


위와 같은 경우는 외접원 안에 다른 점이 포함이 되므로 들로네 삼각분할이 아니다.

처음 분할을 시각할때는 이미지 전체를 다 포함하도록 크게 마진을 준 삼각형을 정의한다. 이 삼각형은 추후 제거한다.

이 과정은 주어진 41개의 특징점에 대해 모두 적용이 된다. 외접원에 특징점이 포함이 되면 그 삼각형은 제거의 대상이 된다. 제거된 삼각형의 변은 polygon 벡터에 저장이 되며 이 벡터를 사용해 새로운 특징점과 연결하여 새로운 삼각형을 만든다. 새로 만들어진 삼각형들의 좌표는 저장이된다.

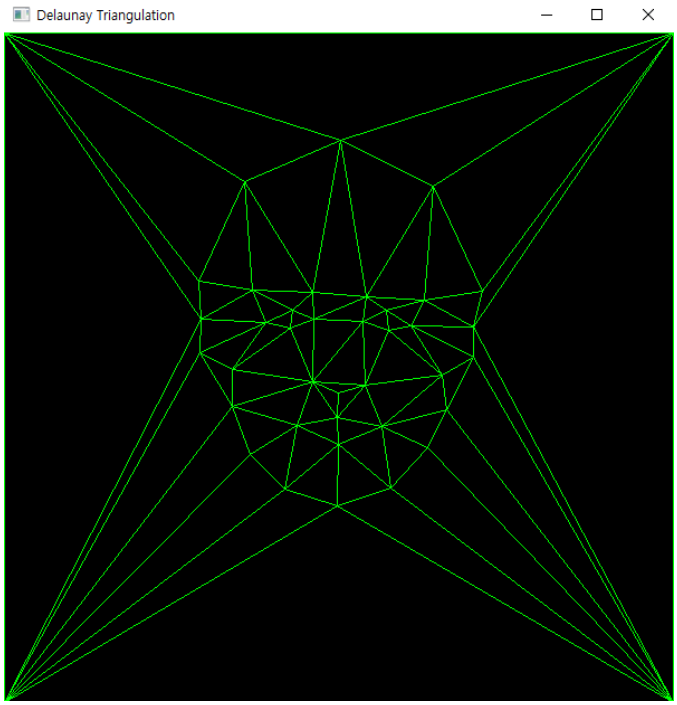




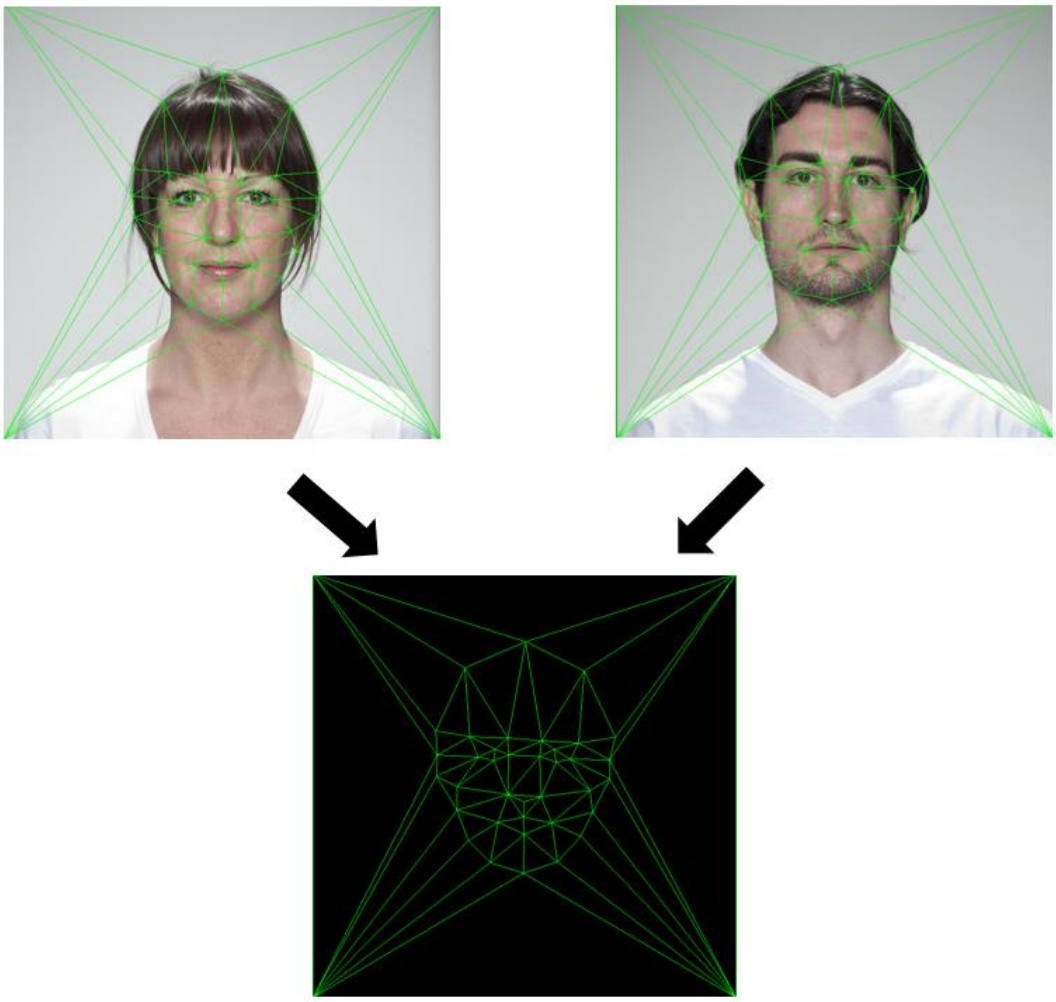
< 그림3. 각 이미지의 들로네 삼각분할 적용 결과>

위와 같이 들로네 삼각분할의 결과가 잘 적용되었음을 확인 가능하다.

구현 상 두 들로네 삼각분할 적용결과를 모두 사용하지는 못하였다. 따라서 차선책으로 사전에 두 특징점을 일정 비율로 중간점을 만들어두고 그 중간점으로 직접 이미지를 아핀 변환하도록 구현한다.



< 그림4. 두 특징점의 중간값으로 들로네 삼각분할 적용결과>



각 이미지를 중간 특징점으로 아핀 변환

```
void calculateAffineTransform(vector<Point2f>& src, vector<Point2f>& dst, Mat& M)
{   // 어핀 변환의 매트릭스
    Mat A(6, 6, CV_32F), B(6, 1, CV_32F);
    for (int i = 0; i < 3; i++)
    {
        A.at<float>(i * 2, 0) = src[i].x;
        A.at<float>(i * 2, 1) = src[i].y;
        A.at<float>(i * 2, 2) = 1;
        A.at<float>(i * 2, 3) = 0;
        A.at<float>(i * 2, 4) = 0;
        A.at<float>(i * 2, 5) = 0;

        A.at<float>(i * 2 + 1, 0) = 0;
        A.at<float>(i * 2 + 1, 1) = 0;
        A.at<float>(i * 2 + 1, 2) = 0;
        A.at<float>(i * 2 + 1, 3) = src[i].x;
        A.at<float>(i * 2 + 1, 4) = src[i].y;
        A.at<float>(i * 2 + 1, 5) = 1;

        B.at<float>(i * 2, 0) = dst[i].x;
        B.at<float>(i * 2 + 1, 0) = dst[i].y;
    }

    Mat X = A.inv(DECOMP_SVD) * B;
    M = (Mat_<float>(2, 3) << X.at<float>(0, 0), X.at<float>(1, 0), X.at<float>(2, 0),
        X.at<float>(3, 0), X.at<float>(4, 0), X.at<float>(5, 0));
}
```

```cpp
void warpAffineCustom(Mat& src, Mat& dst, Mat& M, Size size)
{   // 변환 매트릭스를 사용하여 원본 이미지를 변환된 이미지로 변환
    dst = Mat::zeros(size, src.type());
    for (int y = 0; y < src.rows; y++)
    {
        for (int x = 0; x < src.cols; x++)
        {   // 계산된 위치 pt가 변환된 이미지의 범위 내에 있는지 확인한 후,
            // 원본 이미지의 해당 픽셀 값을 변환된 이미지의 계산된 위치에 복사
            Point2f pt = Point2f(M.at<float>(0, 0) * x + M.at<float>(0, 1) * y + M.at<float>(0, 2),
                M.at<float>(1, 0) * x + M.at<float>(1, 1) * y + M.at<float>(1, 2));
            if (pt.x >= 0 && pt.x < size.width && pt.y >= 0 && pt.y < size.height)
                dst.at<Vec3b>(pt) = src.at<Vec3b>(y, x);
        }
    }
}
```

< 그림5. 어핀변환구현 함수정의 (구현시 오류있음) >

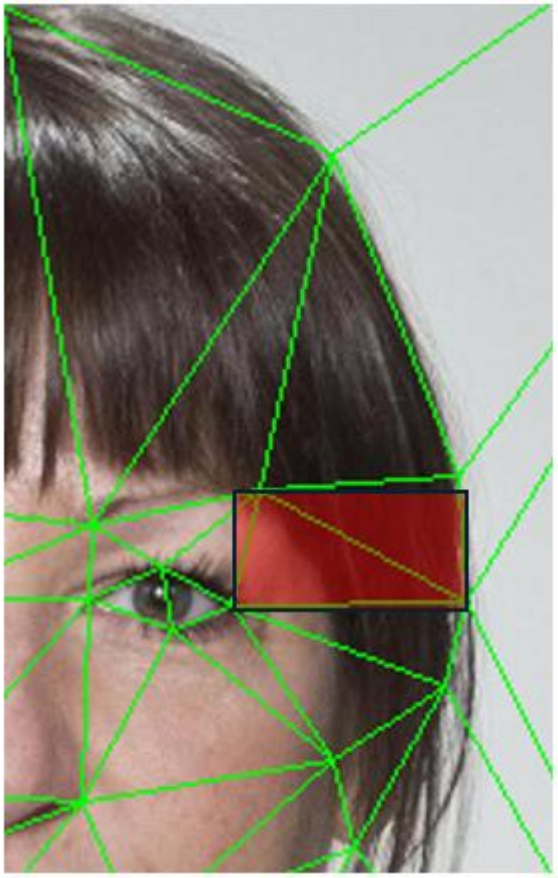
본 주어진 수식에 따라 어핀 변환을 구현하는 함수를 작성하였다. 변환 매트릭스를 선언하여 주어진 아핀 변환을 수행한다. 그러나 본 함수는 제대로 작동하지 않는 문제점이 있어 opencv내장 함수를 대신 사용하였다.

```cpp
// 만든 포인트로 와핑.
void morphTriangle(Mat& img1, Mat& img2, Mat& dst, vector<Point2f>& t1, vector<Point2f>& t2, vector<Point2f>& t, double alpha)
{
    Rect r = boundingRect(t); // 삼각형을 포함하는 가장 작은 사각형을 계산
    Rect r1 = boundingRect(t1);
    Rect r2 = boundingRect(t2);

    vector<Point2f> t1Rect, t2Rect, tRect;
    vector<Point> tRectInt;

    for (int i = 0; i < 3; i++) // 삼각형의 점들을 r, r1, r2 사각형의 좌표로 변환한다.
    {
        tRect.push_back(Point2f(t[i].x - r.x, t[i].y - r.y));
        tRectInt.push_back(Point(t[i].x - r.x, t[i].y - r.y));
        t1Rect.push_back(Point2f(t1[i].x - r1.x, t1[i].y - r1.y));
        t2Rect.push_back(Point2f(t2[i].x - r2.x, t2[i].y - r2.y));
    }

    Mat img1Rect, img2Rect;
    img1(r1).copyTo(img1Rect);
    img2(r2).copyTo(img2Rect);

    Mat warpImage1 = Mat::zeros(r.height, r.width, img1Rect.type());
    Mat warpImage2 = Mat::zeros(r.height, r.width, img2Rect.type());

    applyAffineTransform(warpImage1, img1Rect, t1Rect, tRect); // 두번의 어파인 변환 수행
    applyAffineTransform(warpImage2, warpImage1, t2Rect, tRect);
    /*
    imshow("Morphed Image bef1", warpImage1);
    waitKey(0);
    imshow("Morphed Image bef2", warpImage2);
    waitKey(0);
    */
```

```
    Mat imgRect = (1.0 - alpha) * warpImage1 + alpha * warpImage2; // alpha 값에 따라 이미지를 적절하게 혼합

    for (int i = 0; i < r.height; i++)
    {
        for (int j = 0; j < r.width; j++)
        {
            if (pointPolygonTest(tRectInt, Point2f(j, i), false) >= 0) // (j, i) 좌표가 tRectInt 삼각형 내부에 있는지 확인
            {   // 삼각형 내부에 있는 경우 imgRect의 픽셀 값을 dst의 대응 위치에 복사
                dst.at<Vec3b>(i + r.y, j + r.x) = imgRect.at<Vec3b>(i, j);
            }
        }
    }
}
```

< 그림6. morphTriangle() 함수구현>

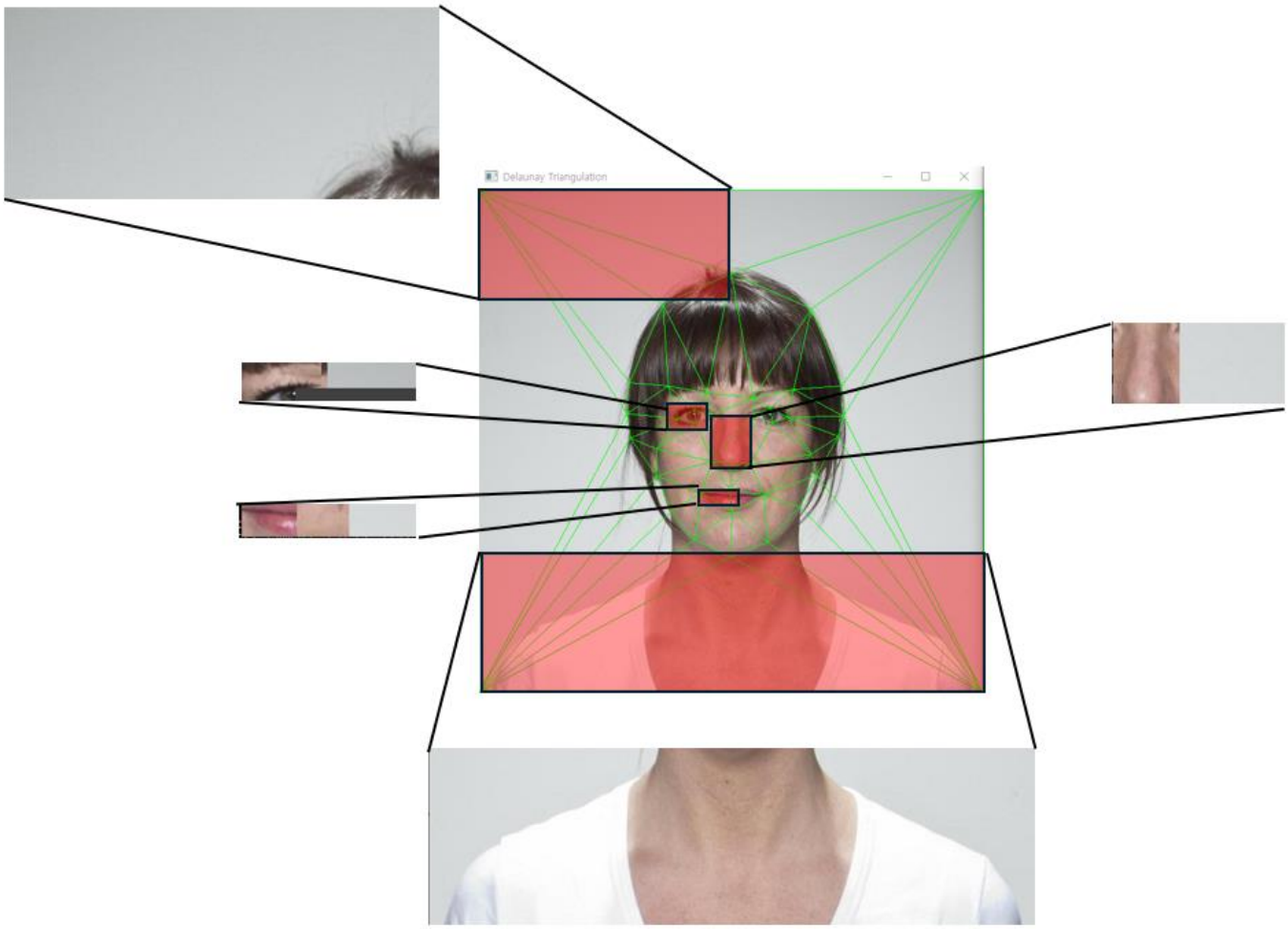
만든 포인트로 와핑을 수행한다. 와핑을 수행할때는 삼각형을 포함하는 가장 작은 사각형을 만들고 그 사각형 별로 와핑을 수행한다. 이는 각 삼각형 부분별로 별도의 와핑을 수행하기 위해서이다.



< 그림7. 삼각형별 사각형 정의 >

삼각형을 사각형 좌표계로 변환하는 이유는 각 삼각형별로 별로의 변환을 수행하기 위해서이다.

이 사각형 영역별로 어파인변환을 수행하면 해당 영역만을 복사하여 다시 붙이는 과정이 단순해진다.



< 그림8. 사각형 분할 결과 >

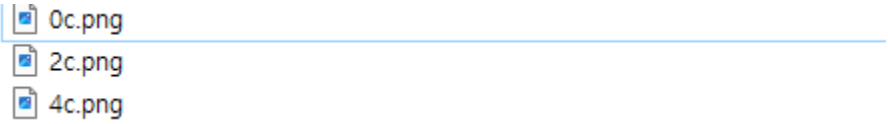
위 그림은 첫번째 입력 이미지의 사각형 분할 결과의 일부를 나타낸 것이다. 이 사각형의 영역별로 아핀 변환을 수행한 이후에 그 결과를 다시 융합한다. 다만 위의 이미지를 확인하면 사각형 분할시에 다른 이미지가 함께 섞여있는 것을 확인 할 수 있다. 이러한 이유 때문에 추후 결과가 노이즈가 있는 것으로 추정된다.

```
int main()
{
    for (int i = 0; i < 5; i++)
    {
        Mat img1 = imread(img_name[i]);
        Mat img2 = imread(img_name[i + 1]);

        if (img1.empty() || img2.empty())
        {
            cout << "Could not open or find the images!" << endl;
            return -1;
        }

        Point key_point1[41];
        Point key_point2[41];

        readKeypoints(txt_name[i], key_point1, 41);
        readKeypoints(txt_name[i + 1], key_point2, 41);

        vector<Point2f> points1, points2, points;
        for (int i = 0; i < 41; ++i)
        {
            points1.push_back(Point2f(key_point1[i].x, key_point1[i].y));
            points2.push_back(Point2f(key_point2[i].x, key_point2[i].y));
            points.push_back(Point2f((key_point1[i].x + key_point2[i].x) * 0.5, (key_point1[i].y + key_point2[i].y) * 0.5));
        }
        // key_point1과 key_point2 배열의 각 키포인트를 Point2f 타입의 벡터로 변환하여 points1과 points2에 저장
        // points 벡터는 points1과 points2의 중간점으로, 두 이미지 사이의 변형을 계산하는 데 사용

        Rect rect(0, 0, img1.cols, img1.rows); // 이미지 전체영역을 나타내는 사각형
        vector<vector<int>> delaunayTri1 = compute_delaunay_triangulation(points1, rect);
        vector<vector<int>> delaunayTri2 = compute_delaunay_triangulation(points2, rect);
        vector<vector<int>> delaunayTri = compute_delaunay_triangulation(points, rect);
```

```cpp
double alpha = 0.5; // Change this value for different morphing stages
Mat morphedImage = Mat::zeros(img1.size(), img1.type());

for (size_t i = 0; i < delaunayTri.size(); i++)
{
    vector<Point2f> t1, t2, t;
    for (size_t j = 0; j < 3; j++)
    {
        t1.push_back(points1[delaunayTri[i][j]]);
        t2.push_back(points2[delaunayTri[i][j]]);
        t.push_back(points[delaunayTri[i][j]]);
    }

    morphTriangle(img1, img2, morphedImage, t1, t2, t, alpha);
}

imshow("Morphed Image", morphedImage);
imwrite(to_string(i) + "c.png", morphedImage);
waitKey(0);
```

< 그림9. Main() >

메인 함수에서는 작업 디렉토리의 input 폴더의 images 폴더내의 주어진 입력이미지와 input 폴더의 labels 폴더내의 특징점이 포함된 텍스트파일을 읽어낸다. 이때, 텍스트 파일의 첫번째 줄은 무시해야한다.



최종 결과는 다음과 같이 0c, 2c, 4c의 이름으로 작업 디렉토리에 저장이 된다.

3. 결과분석

구현결과는 위와 같다. 첫번째 이미지는 어느정도 좋은 수준의 변환이 수행되었지만 두번째, 세번째 이미지에서는 변환이 깨지는 것을 확인할 수 있다. 이미지 전체적으로 삼각형형의 모양이 보이고 있다. 이와 같은 원인으로는 그림 8의 사각형 분할 단계에서 분할한 사각형이 주변의 노이즈도 함께 가져갔기 때문인 것으로 추정된다.

또한, 어파인 변환한 사각형을 융합하는 과정에 있어서도 매끄럽지 않은 구현이 이루어 진 것으로 생각 된다.

4. 고찰

본 구현에서는 두 얼굴 이미지를 델로네 삼각분할을 이용해 변환하고 중간이미지를 생성하는 것이었다. 이를 위해 이미지의 특징점을 추출하고 삼각분할을 수행하고 각 삼각형을 어파인 변환하여 중간형태의 이미지를 생성했다. 최종적으로 두 이미지를 혼합하여 하나의 이미지를 만들었다.

결과가 아쉬운 것은 상술했 듯 morphTriangle()의 구현상 문제점이 아니었을지 생각된다. 들로네 삼각변환까지는 구현이 잘 되었음을 확인했는데 이후의 과정을 보완하면 더욱 좋은 변환결과를 얻을 수 있을 것으로 기대한다.

이러한 과정을 수행하는데 추후에는 인공지능 모델을 사용해보는 것도 유효할 것으로 생각된다. 본 코드는 특징점이 어느정도 비슷한 경우에 좋은 성능을 내지만 실제 현실에서 촬영된 이미지는 얼굴이 기울어져 있거나 표정, 자세등이 완전히 다른 경우가 많다. 이런 경우는 invariant한 특징을 뽑아서 특징점으로 사용해야 할 것으로 생각되는데 이러한 과정은 인공지능이 잘 해낼 수 있을 것으로 기대한다.

이미지 생성이 폭발적으로 발전하고 있는 지금 본 코드의 구현시도는 결과와 관계없이 좋은 경험이 될 것 으로 생각한다.