Ex4: Image Warping&& Image Morphing

学号：17343124
姓名：伍斌
完成时间：2020.11.07

项目要求：

任务一：

- 输入图像：普通名片图像，上面可能有手写笔记或者打印内容，但是拍照时可能角度不正。
- 输出：已经矫正好的标准普通名片纸（长宽比 为 90mm : 54mm ），并裁掉无用的其他内容，只保留完整名片图像(不缩减原始图像中的名片大小,以利于后面作业的数字识别等).
- 参考方法：

结合前面的作业, 对输入图像做边缘检测和 Hough 变换, 然后求 名片 纸的四个顶点, 再进行 Warping （或者形变）为标准名片纸，最后裁剪。（具体方法同学可以根据具体情况自己设定）

- 作业要求:

1. 收集的名片用手机拍摄，尽量背景干净，名片图像清晰，提交名片数据集(不低于 40 张图像）。
2. 输出名片的 边缘检测 和顶点检测结果(在图上显示）。
3. 完成测试报告(不低于 10 张图像）。

- **思考：**

如何在保证精度的结果情况下加快运行速度。

任务二：

输入图像:

输出：根据 Image Morphing 的方法完成中间 11 帧的差值，得到一个 Image Morphing 的动画视频。

实现过程：

项目一：

对图像进行霍夫变换，根据 Ex3 的实验已经得出了边缘直线检测和交点计算，这次只需要根据上次画出的直线和顶点将直线围成的部分之外的其他部分裁去。

霍夫变换的内容在 Ex3 中已经有了详细描述，这里只进行部分代码展示：

```
class Hough {
private:
    CImg<float> grayImage; // 灰度图
    CImg<float> blurred_img; // 高斯滤波平滑得到的图
    CImg<float> houghspace; // 霍夫空间图
    CImg<float> hough_result; // 霍夫检测图
    CImg<float> card; // card 纸结果图
    vector<Point> peaks; // 霍夫空间直线经过最多的点
    vector<Line> lines; // 直线
    vector<Point> intersections; // 直线交点

    double sigma;
    double gradient_threshold;
    double vote_threshold;
    double peak_dis;
    int x_min, x_max, y_min, y_max;
public:
```

```
    Hough(CImg<float> srcImg, double sigma, double gradient_threshold,
double vote_threshold, double peak_dis);
    CImg<float> imageWarping(CImg<float> srcImg);
    CImg<float> RGBtoGray(const CImg<float>& srcImg); // 转灰度图
    CImg<float> initHoughSpace(); // 初始化霍夫空间
    void findPeaks(); // 投票算法
    void drawLines(); // 寻找并画出直线
    void drawIntersections(); // 寻找并画出直线交点
    vector<CImg<float> > computeTransformMatrix(CImg<float> card); // 计
算变换矩阵
    CImg<float> warping(CImg<float> srcImg); // image warping
};
```

相较于 Ex3 的代码，本次的代码只增加了如下两个函数：

```
    CImg<float> imageWarping(CImg<float> srcImg);
    CImg<float> warping(CImg<float> srcImg); // image warping
```

其中：

1. **warping（）**是将图像进行裁剪。

```
CImg<float> Hough::warping(CImg<float> srcImg) {
    // 名片的像素长宽分别是1134*661 或者531*319
    CImg<float> card(661, 1134, 1, 3, 0);
    //CImg<float> card(319, 531, 1, 3, 0);
    vector<CImg<float> > transform;
    transform = computeTransformMatrix(card); // 计算变换矩阵

    CImg<float> y(1, 2, 1, 1, 0);
    CImg<float> c(1, 2, 1, 1, 0);
    CImg<float> A(2, 2, 1, 1, 1);
    A(0, 0) = 0;
    A(0, 1) = card._width - 1;
    y(0, 0) = card._height - 1;
    y(0, 1) = 0;
    c = y.solve(A);
    CImg<float> temp1(1, 3, 1, 1, 1), temp2(1, 3, 1, 1, 1);
    cimg_forXY(card, i, j) {
        temp1(0, 0) = i;
        temp1(0, 1) = j;

        double inner_procuct = i * c(0, 0) - j + c(0, 1);
        temp2 = inner_procuct >= 0 ? transform[0] * temp1 : transform[1
] * temp1;
        temp2(0, 0) = temp2(0, 0) < 0 ? 0 : (temp2(0, 0) > x_max ? x_ma
x : temp2(0, 0));
```

```
        temp2(0, 1) = temp2(0, 1) < 0 ? 0 : (temp2(0, 1) > y_max ? y_ma
x : temp2(0, 1));
        card(i, j, 0) = srcImg(temp2(0, 0), temp2(0, 1), 0);
        card(i, j, 1) = srcImg(temp2(0, 0), temp2(0, 1), 1);
        card(i, j, 2) = srcImg(temp2(0, 0), temp2(0, 1), 2);
    }
    return card;
}
```

2. **imageWarping（）**是对整体变换算法进行封装。

```
CImg<float> Hough::imageWarping(CImg<float> srcImg) {
    this->grayImage = RGBtoGray(srcImg); // 转灰度图
    this->blurred_img = grayImage.get_blur(sigma); // 高斯滤波平滑
    this->houghspace = initHoughSpace(); // 初始化霍夫空间
    findPeaks(); // 找出霍夫空间中直线经过最多的点
    drawLines(); // 寻找并画出直线
    drawIntersections(); // 寻找并画出直线交点
    //hough_result.display();
    this->card = warping(srcImg); // image warping
    return card;
}
```

测试效果：

类似于 Ex3，根据 Hough 类的构造函数：

```
Hough(CImg<float> srcImg, double sigma, double gradient_threshold, doub
le vote_threshold, double peak_dis);
```

根据 10 张名片的测试，主要影响其实验效果的是参数 sigma 和 vote_threshold 。（由于没有找到足够数据，所以只完成了 7 张图片的测试，每个测例都已调参至相对满意状态）

1. 当输入的 sigma=6.5f，vote_threshold=500 时：

输入图片：

输出图片：

输出的直线与交点：

```
Line 0: y = -28.6363x + 2664.79
Line 1: y = 0.0874887x + 96.3667
Line 2: y = 57.29x + -51740.7
Line 3: y = -0.0349208x + 1611.98
Intersection 0: x = 89.4183, y = 104.19
Intersection 1: x = 36.8099, y = 1610.7
Intersection 2: x = 906.204, y = 175.649
Intersection 3: x = 930.707, y = 1579.48
CImg<float> (661x1134x1x3): this = 00000000007ffaa0, size = (661,1134,1,3) [8 Mio], data = (float*)000000000816e040..000
0000008a02087 (non-shared) = [ 13 16 20 24 33 37 35 31 ... 165 164 160 164 163 166 157 154 ], min = 0, max = 255, mean =
 148.942, std = 28.5924, coords_min = (1,962,0,0), coords_max = (126,364,0,0).
```

分析：

检测出 4 条直线 4 个顶点，并完美裁切输出名片。在此参数下 Hough 算法对本图片拟合效果较好。

2. 当输入的 sigma=4.5f，vote_threshold=400 时：

输入图片：

输出图片:

输出的直线与交点:

```
Line 0: y = -0.0524078x + 127.174
Line 1: y = 14.3007x + -2695.09
Line 2: y = 19.0811x + -14349.6
Line 3: y = -6.12303e-017x + 1071
Intersection 0: x = 196.631, y = 116.869
Intersection 1: x = 756.617, y = 87.5216
Intersection 2: x = 263.351, y = 1071
Intersection 3: x = 808.159, y = 1071
CImg<float> (661x1134x1x3): this = 00000000007ffaa0, size = (661,1134,1,3) [8 Mio], data = (float*)00000000070e4040..000
0000007978087 (non-shared) = [ 129 125 115 118 118 118 121 124 ... 151 152 152 152 151 149 148 148 ], min = 21, max = 23
8, mean = 196.333, std = 22.1864, coords_min = (268,302,0,0), coords_max = (433,636,0,2).
```

分析:

检测出 4 条直线 4 个顶点，并裁切输出名片。但输出的名片并没有完全裁剪掉边缘，按照图片直观来看，应该是光照导致边缘不明显，导致 sobel 算子未能成功识别出正确的边缘。但是根据反复调参后的结果，这已是本图片最佳拟合状态。故在此参数下 Hough 算法对本图片拟合效果一般。

3. 当输入的 sigma=5.5f，vote_threshold=500 时：

输入图片:

输出图片:

输出的直线与交点：

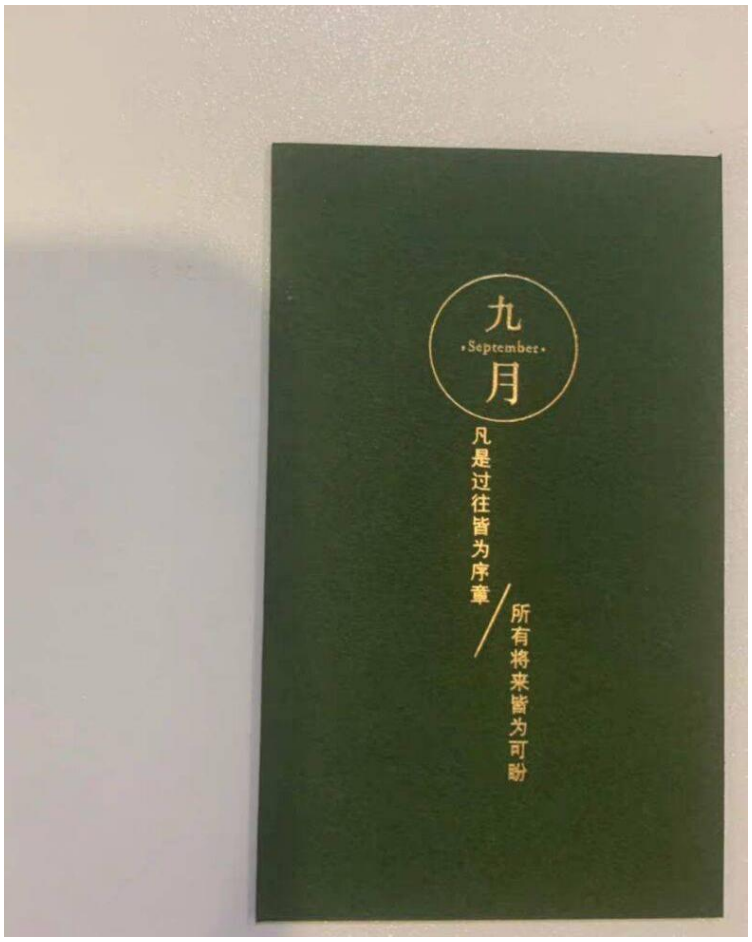
```
Line 0: y = -0.0174551x + 113.017
Line 1: y = 28.6363x + -4928.44
Line 2: y = 57.29x + -54204.6
Line 3: y = -infx + inf
Line 4: y = -0.0349208x + 1381.84
Intersection 0: x = 175.944, y = 109.946
Intersection 1: x = 947.828, y = 96.4728
Intersection 2: x = 220.091, y = 1374.16
Intersection 3: x = 969.673, y = 1347.98
CImg<float> (661x1134x1x3): this = 00000000007ffaa0, size = (661,1134,1,3) [8 Mio], data = (float*)0000000007200040..000
0000007a94087 (non-shared) = [ 135 132 134 140 138 131 129 129 ... 1 0 2 8 7 7 0 2 ], min = 0, max = 204, mean = 142.373
, std = 39.4424, coords_min = (307,725,0,0), coords_max = (642,20,0,2).
```

分析：

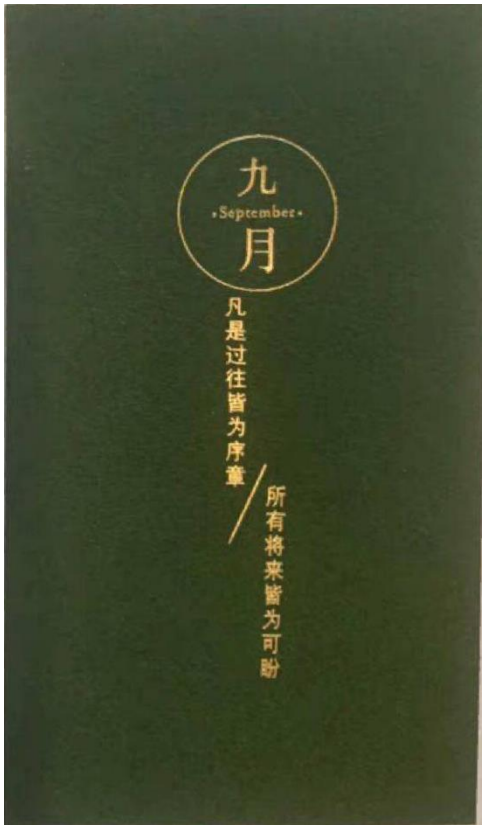
检测出 4 条直线 4 个顶点，并完美裁切输出名片。在此参数下 Hough 算法对本图片拟合效果较好。

4. 当输入的 sigma=4.5f，vote_threshold=400 时：

输入图片：


九
·September·
月
凡是过往皆为序章
所有将来皆为可盼

输出图片:


九
·September·
月
凡是过往皆为序章
所有将来皆为可盼

输出的直线与交点：

```
Line 0: y = 0.0349208x + 135.082
Line 1: y = -57.29x + 16272.8
Line 2: y = 57.29x + -42802.1
Line 3: y = -infx + inf
Line 4: y = -6.12303e-017x + 960
Intersection 0: x = 281.514, y = 144.913
Intersection 1: x = 749.929, y = 161.27
Intersection 2: x = 267.286, y = 960
Intersection 3: x = 763.871, y = 960
CImg<float> (661x1134x1x3): this = 00000000007ffaa0, size = (661,1134,1,3) [8 Mio], data = (float*)0000000004ee6040..000
000000577a087 (non-shared) = [ 198 185 185 172 111 112 112 117 ... 145 166 170 170 171 168 171 171 ], min = 0, max = 255
, mean = 58.6927, std = 20.2538, coords_min = (376,395,0,2), coords_max = (368,360,0,0).
```

分析：

检测出 4 条直线 4 个顶点，并完美裁切输出名片。在此参数下 Hough 算法对本图片拟合效果较好。

5. 当输入的 sigma=4.0f，vote_threshold=300 时：

输入图片：

输出图片：

输出的直线与顶点：

```
Line 0: y = -28.6363x + 1346.72
Line 1: y = -0.0174551x + 143.022
Line 2: y = -0.0174551x + 203.031
Line 3: y = 0.0349208x + 417.254
Line 4: y = -57.29x + 31858.1
Intersection 0: x = 42.0599, y = 142.288
Intersection 1: x = 39.963, y = 202.333
Intersection 2: x = 32.4183, y = 418.386
Intersection 3: x = 553.757, y = 133.356
Intersection 4: x = 552.709, y = 193.383
Intersection 5: x = 548.467, y = 436.407
CImg<float> (661x1134x1x3): this = 00000000007ffaa0, size = (661,1134,1,3) [8 Mio], data = (float*)0000000003c06040..000
000000449a087 (non-shared) = [ 126 126 126 126 126 126 126 126 ... 78 82 82 82 84 94 156 150 ], min = 0, max = 247, mean
 = 152.776, std = 33.6839, coords_min = (631,241,0,0), coords_max = (521,843,0,2).
```

分析：

输出了四条直线和六个交点，裁剪出的图片也非常不符合要求。但根据反复论证，这已经是本算法能输出的最佳图像。在此参数下 Hough 算法对本图片拟合效果非常差。效果差的主要原因：拍摄的图像背景和名片之间的色差太小，导致转化为灰度图像之后，用 sobel 算子计算边缘的难度大增，因为给出的直线与顶点均不对，所以不能裁剪出正确的名片。

6. 当输入的 sigma=6.0f，vote_threshold=500 时：

输入图片：

输出图片:

输出的直线与顶点:

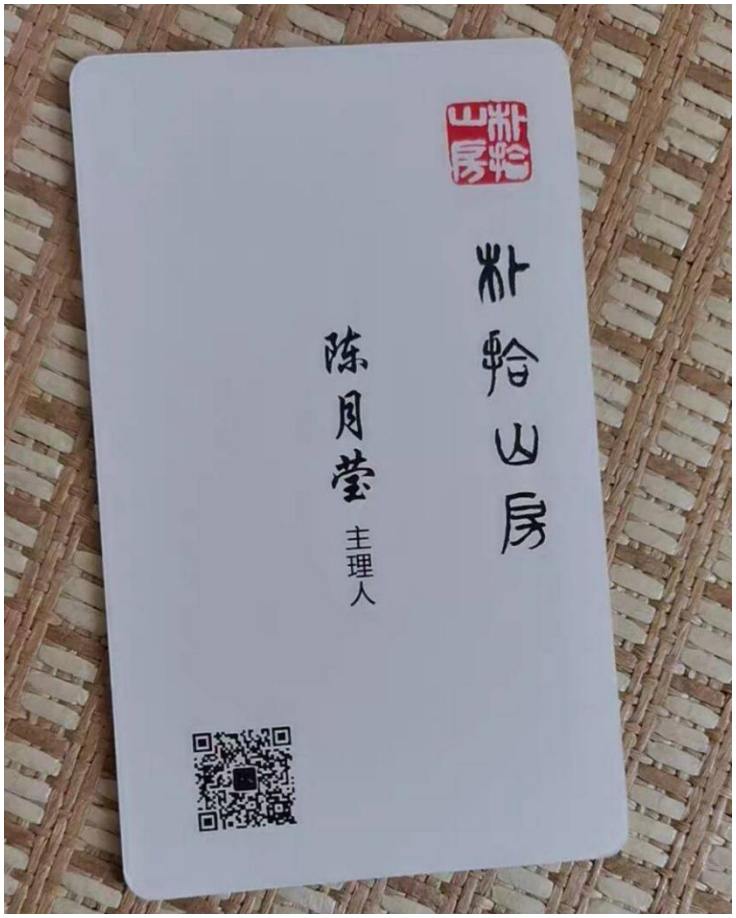
```
Line 0: y = 57.29x + -13178.7
Line 1: y = -infx + inf
Line 2: y = -0.0174551x + 358.055
Line 3: y = 28.6363x + -22292.6
Line 4: y = -0.0174551x + 1283.2
Intersection 0: x = 236.213, y = 353.931
Intersection 1: x = 252.356, y = 1278.79
Intersection 2: x = 790.496, y = 344.256
Intersection 3: x = 822.783, y = 1268.83
CImg<float> (661x1134x1x3): this = 00000000007ffaa0, size = (661,1134,1,3) [8 Mio], data = (float*)0000000007193040..000
0000007a27087 (non-shared) = [ 16 16 16 22 23 28 23 26 ... 51 47 39 42 39 31 31 25 ], min = 0, max = 219, mean = 86.1576
, std = 46.0702, coords_min = (64,0,0,0), coords_max = (413,584,0,1).
```

分析:

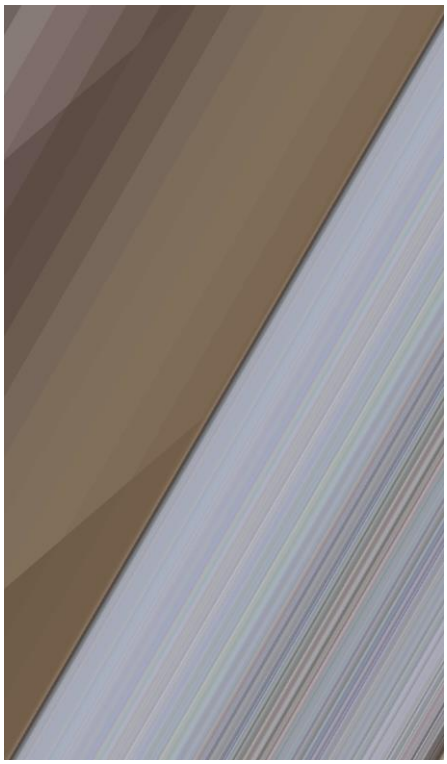
检测出 4 条直线 4 个顶点，并完美裁切输出名片。在此参数下 Hough 算法对本图片拟合效果较好。

7. 当输入的 sigma=6.0f，vote_threshold=500 时：

输入图片:



输出图片:



输出的直线与交点：

```
Line 0: y = 7.11537x + 0
Line 1: y = 14.3007x + -1060.83
Line 2: y = -0.0349208x + 62.0378
Line 3: y = 14.3007x + -9561.84
Line 4: y = -0.0874887x + 1076.09
Intersection 0: x = 147.639, y = 1050.51
Intersection 1: x = 8.67626, y = 61.7348
Intersection 2: x = 149.398, y = 1063.02
Intersection 3: x = 78.3275, y = 59.3025
Intersection 4: x = 148.52, y = 1063.1
Intersection 5: x = 671.328, y = 38.5945
Intersection 6: x = 739.353, y = 1011.41
CImg<float> (661x1134x1x3): this = 00000000007ffaa0, size = (661,1134,1,3) [8 Mio], data = (float*)000000000572c040..000
0000005fc0087 (non-shared) = [ 111 111 111 111 111 111 111 111 ... 88 88 107 134 121 131 150 162 ], min = 61, max = 196,
 mean = 131.865, std = 35.755, coords_min = (660,15,0,2), coords_max = (660,218,0,2).
```

分析：

输出了四条直线和七个交点，裁剪出的图片也非常不符合要求（根本没有检测到名片）。但根据反复论证，这已经是本算法能输出的最佳图像（调高两个值将不会检测到直线输出图片，调低则会输出类似的乱码）。故在此参数下 Hough 算法对本图片拟合效果非常差。效果差的主要原因：拍摄的图像背景和名片之间的色差太小，导致转化为灰度图像之后，用 sobel 算子计算边缘的难度大增，因为算法默认裁剪的名片是直线围出的最小区域，而的给出的直线与顶点均不对，所以不能裁剪出正确的名片。

思考：如何在保证精度的结果情况下加快运行速度。

（与 Ex3 一样，这部分有参考 CSDN 博客：https://blog.csdn.net/Jeanet_1/article/details/49387967）

Hough 变换检测直线的原理是首先得到边缘点，对每一边缘点得到过边缘点的所有直线，直线参数为极坐标下的(row,theta)，而每一个(row,theta)都对应极坐标中一个点，对每一条直线对应的点记为投票一次，最终记录投票数，投票数大于某一域值的即可看作一条直线。

由此我们可以看出，算法中计算量大，主要是因为：第一，对边缘点做一个 360 度的直线组，即经过该边缘点上的所有直线，对其对应的(row,theta)投票。第二，对每一边缘点都要重复此过程。在边缘点众多的情况下，这样的计算量是非常可怕的。

因此，对算法的改进主要针对此两个方面。

1、用两次 sobel 算子相减得到单相素宽度的边缘。sobel 算子提取边缘是不错的算法，速度相对较快。但一次 sobel 算子提取得到的边缘一般边缘点较多。用两次 sobel 算子相减可以得到单像素的边缘。这样可以大大减少边缘点的个数

2、计算每一边缘点的梯度方向，统计每个方向上的点的个数，找到边缘点个数最多的方向，做为粗略的直线方向。此处，为了容错，也可以考虑对梯度方向值进行拟合后，找几个波峰做为备选直线方向。视具体情况。

3、得到大致方向后，将与此方向偏差较多的边缘点剔除。做完这一步后，你会发现，边缘点已经减少很多了。而你需要的信息不会丢失。

4、在直线方向的正负 n 度内（这个 n 主要看你的容错标准），做 Hough 变换。找到直线。

项目二：（附加题）

实验原理

Cross-Dissolve 交叉融合，对两张图片每个像素点按一定的比例进行混合。

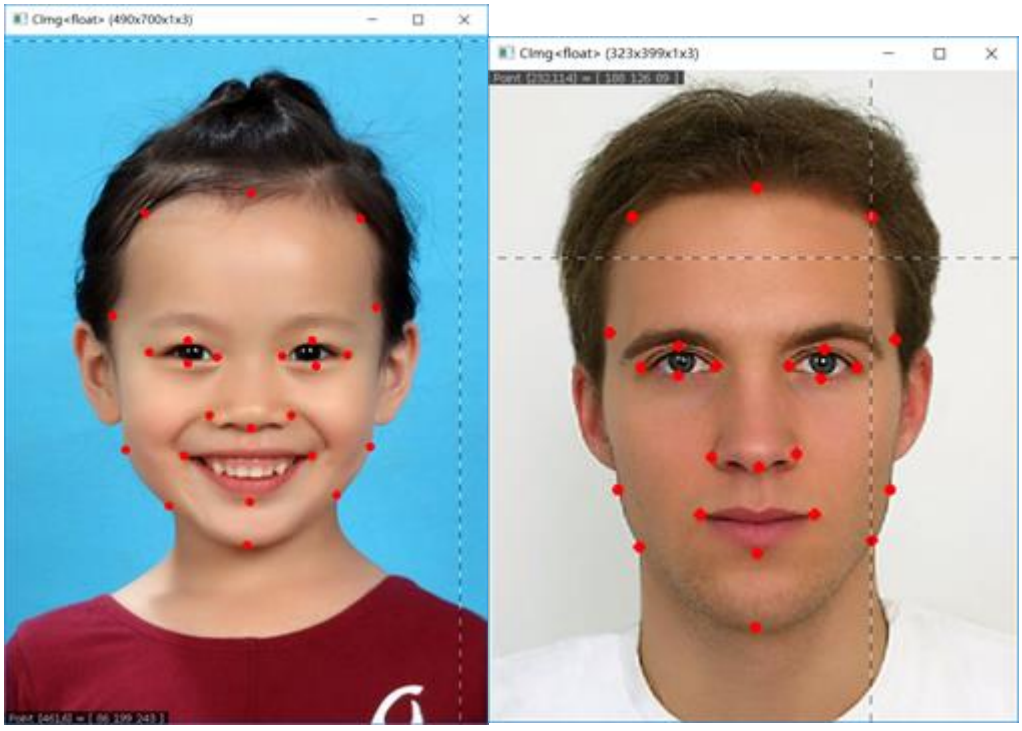
公式：Imagehalfway = (1-t) * Image1 + t * image2

但这种方法只适合图像对齐的情况，对于没有对齐的情况，可以采用局部变形的思想，先根据特征点划分出局部图像，然后对局部图像求出平均图像，在平均图像上使用交叉融合，需要的像素值可以使用原图像的双线性插值，最后把局部图像拼接成整体图像。

具体步骤

1. 标记特征点

首先，对图片进行特征点标记，可以使用 dlib 库对人脸实现自动标记，但是需要用到 opencv 库，这里进行手动标记，并已经把标记的点存入文件中。除了手动检测的这几个点，还给每张图片加入了 8 个点，分别是 4 个角点和每条片的中点。



2. 进行三角剖分

```
// src 图的三角形
void ImageMorphing::setSrcTriangleList() {
    for (int i = 0; i < index.size(); i++) {
        Triangle src_triangle(src_points[index[i][0]], src_points[index
[i][1]], src_points[index[i][2]]);
        src_triangle_list.push_back(src_triangle);
    }
}
// dst 图的三角形
void ImageMorphing::setDstTriangleList() {
    for (int i = 0; i < index.size(); i++) {
        Triangle dst_triangle(dst_points[index[i][0]], dst_points[index
[i][1]], dst_points[index[i][2]]);
        dst_triangle_list.push_back(dst_triangle);
    }
}
// morphing 过程中每一帧的三角形
void ImageMorphing::setMidTriangleList() {
    for (int i = 0; i < frame_cnt; i++) {
        vector<Point> temp_mid_points;
        for (int j = 0; j < src_points.size(); j++) {
            float mid_x = float(src_points[j].x) + float(i+1)/(frame_cn
t+1) * (dst_points[j].x - src_points[j].x);
            float mid_y = float(src_points[j].y) + float(i+1)/(frame_cn
t+1) * (dst_points[j].y - src_points[j].y);
            Point mid_point(mid_x, mid_y);
            temp_mid_points.push_back(mid_point);
        }
        mid_points.push_back(temp_mid_points);
    }

    for (int i = 0; i < frame_cnt; i++) {
        vector<Triangle> temp_mid_triangles;
        for (int j = 0; j < index.size(); j++) {
            Triangle mid_triangle(mid_points[i][index[j][0]], mid_point
s[i][index[j][1]], mid_points[i][index[j][2]]);
            temp_mid_triangles.push_back(mid_triangle);
        }
        mid_triangle_list.push_back(temp_mid_triangles);
    }
}
```

3. 重心法判断一个点是否在三角形内

```
bool ImageMorphing::isInTriangle(Point P, Triangle tri) {
    float x0 = tri.p3.x - tri.p1.x, y0 = tri.p3.y - tri.p1.y;
    float x1 = tri.p2.x - tri.p1.x, y1 = tri.p2.y - tri.p1.y;
    float x2 = P.x - tri.p1.x, y2 = P.y - tri.p1.y;

    float temp_00 = x0 * x0 + y0 * y0;
    float temp_01 = x0 * x1 + y0 * y1;
    float temp_02 = x0 * x2 + y0 * y2;
    float temp_11 = x1 * x1 + y1 * y1;
    float temp_12 = x1 * x2 + y1 * y2;

    float u = float(temp_11 * temp_02 - temp_01 * temp_12) / (float)(te
mp_00 * temp_11 - temp_01 * temp_01);
    float v = float(temp_00 * temp_12 - temp_01 * temp_02) / (float)(te
mp_00 * temp_11 - temp_01 * temp_01);
    if (u + v <= 1 && u >= 0 && v >= 0) return true;
    return false;
}
```

4. 计算变换矩阵

```
CImg<float> ImageMorphing::computeTransformMatrix(Triangle before, Tria
ngle after) {
    CImg<float> A(3, 3, 1, 1, 1);
    CImg<float> y1(1, 3, 1, 1, 0), y2(1, 3, 1, 1, 0);
    CImg<float> c1(1, 3, 1, 1, 0), c2(1, 3, 1, 1, 0);

    A(0, 0) = before.p1.x; A(1, 0) = before.p1.y;
    A(0, 1) = before.p2.x; A(1, 1) = before.p2.y;
    A(0, 2) = before.p3.x; A(1, 2) = before.p3.y;
    y1(0, 0) = after.p1.x; y2(0, 0) = after.p1.y;
    y1(0, 1) = after.p2.x; y2(0, 1) = after.p2.y;
    y1(0, 2) = after.p3.x; y2(0, 2) = after.p3.y;
    c1 = y1.solve(A);
    c2 = y2.solve(A);
    CImg<float> transform(3, 3, 1, 1, 0);
    for (int i = 0; i < 3; i++) {
        transform(i, 0) = c1(0, i);
        transform(i, 1) = c2(0, i);
    }
    transform(2, 2) = 1;
    return transform;
}
```

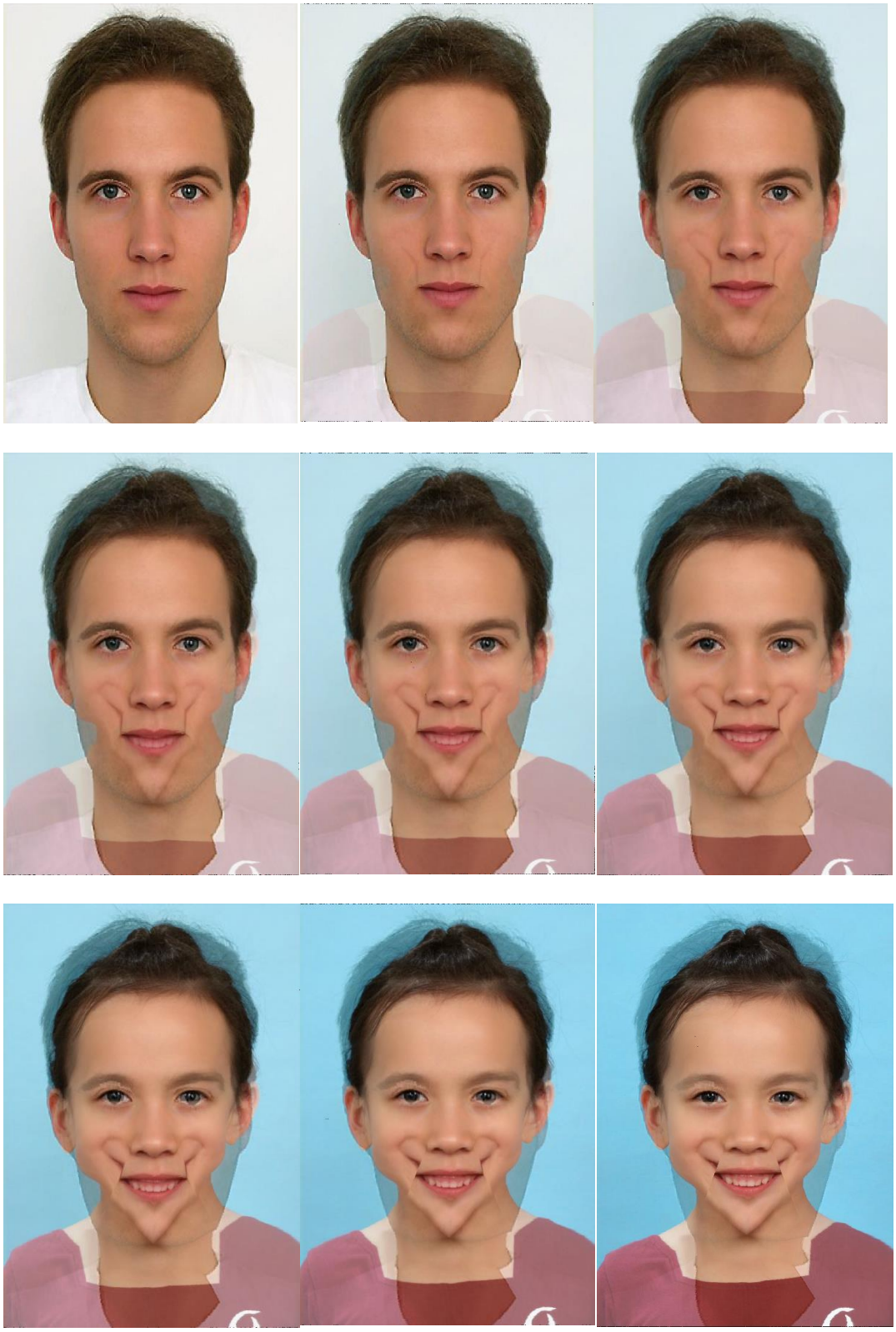
5. 对每一帧进行 morphing

```
CImgList<float> ImageMorphing::morphing() {
    int size = mid_triangle_list[0].size();

    result.push_back(src); // 放入第一帧
    for (int k = 0; k < frame_cnt; k++) {
        CImg<float> middle(dst._width, dst._height, 1, 3, 1);
        cimg_forXY(middle, i, j) {
            CImg<float> x(1, 3, 1, 1, 1), y1(1, 3, 1, 1, 1), y2(1, 3, 1
, 1, 1);
            for (int m = 0; m < size; m++) {
                Point p(i, j);
                if (isInTriangle(p, mid_triangle_list[k][m])) {
                    x(0, 0) = i;
                    x(0, 1) = j;
                    // middle image 到src image 的变换
                    CImg<float> transform1 = computeTransformMatrix(mid
_triangle_list[k][m], src_triangle_list[m]);
                    y1 = transform1 * x;
                    // middle image 到dst image 的变换
                    CImg<float> transform2 = computeTransformMatrix(mid
_triangle_list[k][m], dst_triangle_list[m]);
                    y2 = transform2 * x;
                    // src 和dst 组合得到middle
                    float a = float(k+1)/(frame_cnt+1);
                    middle(i, j, 0) = (1 - a) * src(y1(0, 0), y1(0, 1),
 0) + a * dst(y2(0, 0), y2(0, 1), 0);
                    middle(i, j, 1) = (1 - a) * src(y1(0, 0), y1(0, 1),
 1) + a * dst(y2(0, 0), y2(0, 1), 1);
                    middle(i, j, 2) = (1 - a) * src(y1(0, 0), y1(0, 1),
 2) + a * dst(y2(0, 0), y2(0, 1), 2);
                    break;
                }
            }
        }
        result.push_back(middle); // 放入中间morphing 的每一帧
    }
    result.push_back(dst); // 放入最后一帧
    return result;
}
```

实验结果：

包括初始两张图片及中间的十一帧，一共 13 张图像如下：



分析：由于三角形拆分的不是很多，所以中间帧图片的变换流畅性很差。

解决方法：在查阅了资料后（ https://blog.csdn.net/happy990/article/details/86763223 ）后，了解到更合理的三角形剖分的方法：进行 Delaunay 三角剖分。（这里并没有进行代码实现，但是看学长的博客，他使用这种方法后中间帧的流畅度比我完成的效果好很多）

如果点集 V 的一个三角剖分 T 只包含 Delaunay 边，那么该三角剖分称为 Delaunay 三角剖分。Delaunay 边是指：假设 E 中的一条边 e(两个端点为 a,b)，e 若满足下列条件，则称之为 Delaunay 边：存在一个圆经过 a,b 两点，圆内（注意是圆内，圆上最多三点共圆）不含点集 V 中任何其他的点，这一特性又称空圆特性。最优化：在散点集可能形成的三角剖分中，Delaunay 三角剖分所形成的三角形的最小角最大。从这个意义上讲，Delaunay 三角网是“最接近于规则化的”三角网。

Delaunay 剖分是一种三角剖分的标准，实现它有多种算法。目前常用的一种算法是 Bowyer-Watson 算法，主要步骤如下：

1. 构造一个超级三角形，包含所有散点，放入三角形链表。
2. 将点集中的散点依次插入，在三角形链表中找出其外接圆包含插入点的三角形（称为该点的影响三角形），删除影响三角形的公共边，将插入点同影响三角形的全部顶点连接起来，从而完成一个点在 Delaunay 三角形链表中的插入。
3. 根据优化准则对局部新形成的三角形进行优化。将形成的三角形放入 Delaunay 三角形链表。

4. 循环执行上述第 2 步，直到所有散点插入完毕。