

数字图像处理第一次大作业*

石大川 <sdc17@mails.tsinghua.edu.cn>

2020 年 4 月 20 日

目录

1 Environment	2
2 Image Fusion	2
2.1 Usage	2
2.2 Principle	2
2.3 Result	3
3 Face Morphing	3
3.1 Usage	3
3.2 Principle	4
3.3 Result	4
4 View Morphing	6
4.1 Usage	6
4.2 Principle	6
4.3 Result	8
4.3.1 Test1	8
4.3.2 Test2	11
4.3.3 Test3	11
5 Conclusion	13
6 References	14

*<https://github.com/sdc17/NaiveImageFusionAndMorphing>

1 Environment

所有实验均在 windows10 下完成，需要安装依赖：

```
1 conda install --yes --file requirements.txt
```

其中 opencv-python 和 dlib 可能要用 pip 另外安装。

2 Image Fusion

2.1 Usage

输入 input, target 和 mask 所在的文件位置以及要被拷贝到的横纵坐标

```
1 (dip1) D:\Courses\_DIP\exp1\code>python imagefusion.py -h
2 usage: imagefusion.py [-h] [-i INPUT] [-t TARGET] [-m MASK] [-x PASTEX]
3 [-y PASTEY]
4
5 Image Fusion
6
7 optional arguments:
8 -h, --help show this help message and exit
9 -i INPUT      input image file location
10 -t TARGET     input target file location
11 -m MASK       input mask file location
12 -x PASTEX    where to paste x axes
13 -y PASTEY    where to paste y axes
```

或者直接运行默认的 test1

```
1 (dip1) D:\Courses\_DIP\exp1\code>python imagefusion.py
```

2.2 Principle

人类视觉对局部对比度变化敏感度高于整体亮度变化。

因此把 A 图的一部分拷贝到 B 图上时，让拷贝的边界像素值等于 B 图的像素值，边界内的点保持 A 图的散度即可很大程度上消减拷贝造成的 artifacts.

其中每个点的散度为：

$$\nabla p(i, j) = p(i - 1, j) + p(i + 1, j) + p(i, j - 1) + p(i, j + 1) - 4 * p(i, j)$$

实际计算时，等价于拿 Laplacian 算子卷积：

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

替换边界像素值并保持内部散度不变等价于解方程组：

$$\mathbf{A}\vec{x} = \vec{b}, \quad A \in R^{N \times N}, \vec{x} \in R^{N \times 1}, \vec{b} \in R^{N \times N}$$

其中 N 表示 mask 中包含的像素点数目。

矩阵 \mathbf{A} 由两种情况构成，首先对于 mask 边界的点，只需令 \mathbf{A} 的对角线值为 1，该行其他值均为 0；对于 mask 内部的点，需令 \mathbf{A} 的对角线值为 -4，涉及该点散度计算的四个邻接点对应位置的值为 1。

\vec{b} 也有相应的两种情况，对于 mask 边界的点，取值即为 A 图中对应点的像素值；对于 mask 内部的点，取值即为 B 图对应点的散度值。

分三个通道分别求解上述方程组得到 \vec{x} 后，替换目标图片中相应位置的像素点即可。

2.3 Result

分别输入以下命令进行两次测试

```
1 (dip1) D:\Courses\_DIP\exp1\code>python imagefusion.py -i ./images/part1/test1_src.jpg -t ./images/part1/test1_target.jpg -m ./images/part1/test1_mask.jpg -x 50 -y 100
2
3 (dip1) D:\Courses\_DIP\exp1\code>python imagefusion.py -i ./images/part1/test2_src.png -t ./images/part1/test2_target.png -m ./images/part1/test2_mask.png -x 150 -y 180
```

效果见 [Test1](#) 和 [Test2](#)。



Figure 1: Test1 Image Fusion

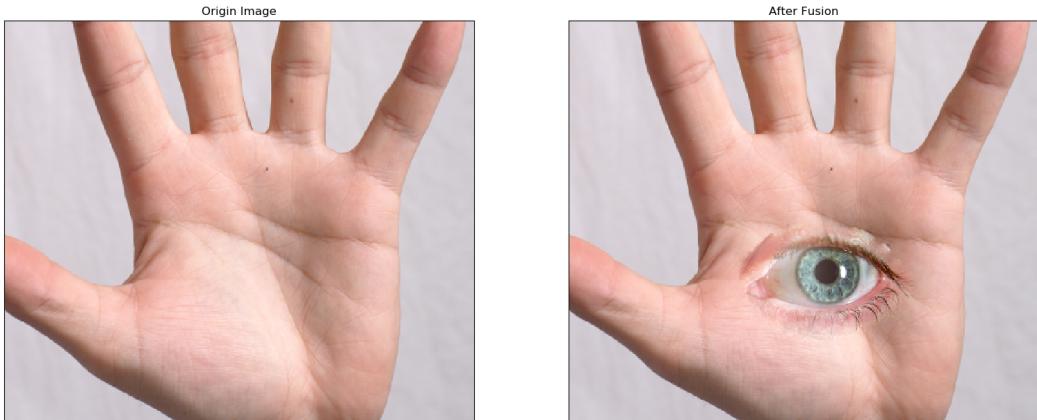


Figure 2: Test2 Image Fusion

可见较好地实现了 Image Fusion 的算法，没有出现明显的 artifacts。

3 Face Morphing

3.1 Usage

输入 input 和 target 所在的文件位置以及要生成的 sequence 的长度（不包含 input 和 target）

```

1 (dip1) D:\Courses\_DIP\exp1\code>python facemorphing.py -h
2 usage: facemorphing.py [-h] [-i INPUT] [-t TARGET] [-s SEQUENCE]
3 [-b {True,False}]
4
5 Face Morphing
6
7 optional arguments:
8 -h, --help      show this help message and exit
9 -i INPUT        input image file location
10 -t TARGET       input target file location
11 -s SEQUENCE    length of the morphing sequence
12 -b {True,False} whether to use bilinear

```

或者直接运行默认的 test1

```

1 (dip1) D:\Courses\_DIP\exp1\code>python facemorphing.py

```

3.2 Principle

首先是做特征点的标定，因为点的数目并不多，所以我拿 labelme 手动标了所有位于脸部和颈部的特征点，后期再添加上图片边界的八个特征点。

接着是做三角剖分，这部分我直接调用了 cv2.Subdiv2D 完成，返回值是一系列三角形的三个顶点的坐标。

分别对 input 和 target 做完三角剖分后，先根据各个特征点的位置合成出 morph 结果的特征点的位置

$$morph(x, y) = (1 - \alpha) \times input(x, y) + \alpha \times target(x, y)$$

再分别做从 input 到 morph 结果对应三角形的 affine warping，和 target 到 morph 结果对应三角形的 affine warping，每个 morph 结果中的三角形根据：

$$morph[x, y, :] = (1 - \alpha) \times input[x, y, :] + \alpha \times target[x, y, :]$$

得到合成后的像素值。其中 affine warping 以及 bilinear 插值的工作在 Image Warping 的作业中已经完成过，这里直接搬过来用，不再赘述原理了。

3.3 Result

分别输入以下命令进行两次测试，其中调整-s 的值可以改变 morph 序列的长度：

```

1 (dip1) D:\Courses\_DIP\exp1\code>python facemorphing.py -i ./images/part2-1/source1.png
2           -t ./images/part2-1/target1.png -s 5 -b True
3
4 (dip1) D:\Courses\_DIP\exp1\code>python facemorphing.py -i ./images/part2-1/source2.png
5           -t ./images/part2-1/target2.png -s 5 -b True

```

测试 1 的三角剖分和 morph 序列分别见[Figure 3](#)和[Figure 4](#)

测试 2 的三角剖分和 morph 序列分别见[Figure 5](#)和[Figure 6](#)

可见较好地实现了 Face Morphing 的算法，artifacts 的程度相较于直接做 Cross-Dissolve 小了很多。

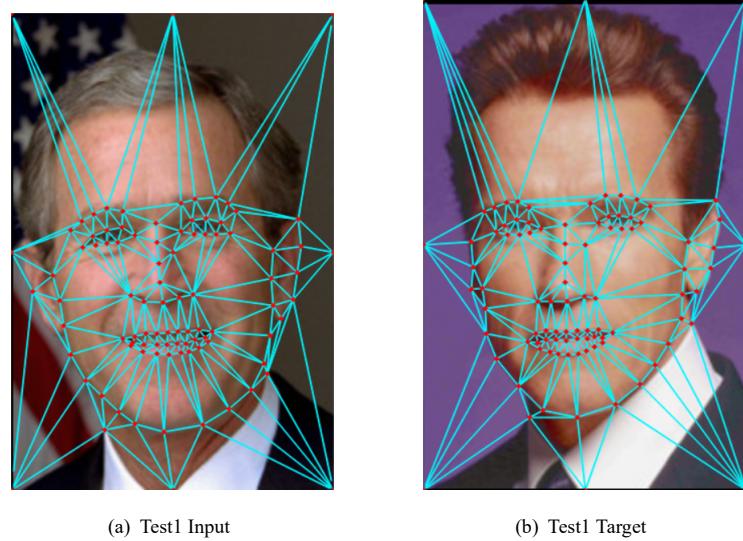


Figure 3: Test1 Delaunay Triangulation



Figure 4: Test1 Face Morphing Sequence

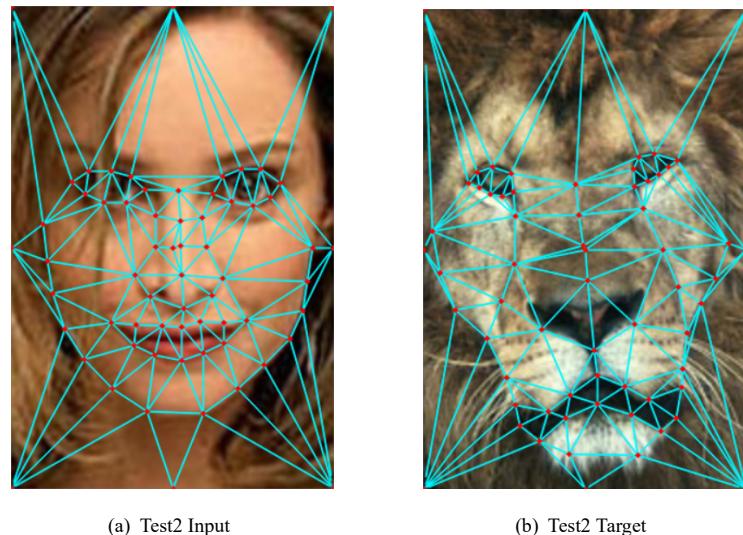


Figure 5: Test2 Delaunay Triangulation



Figure 6: Test2 Face Morphing Sequence

4 View Morphing

4.1 Usage

需要指定 source 和 target 的文件位置，-s 是序列长度，-b 是否使用 Bilinear 插值，-a 是选择手动标 landmark points 还是 dlib 自动检测，-x 和-y 分别代表 PreWarp 后在横纵轴正向上的移位，此外 PreWarp 后的图默认尺寸会被设置为原图的 1.25 倍。

```

1 (dip1) D:\Courses\DIPI\exp1\code>python viewmorphing.py -h
2 usage: viewmorphing.py [-h] [-i INPUT] [-t TARGET] [-s SEQUENCE]
3 [-b {True,False}] [-a {True,False}] [-x SHIFTX]
4 [-y SHIFTY]

5
6 Face Morphing

7
8 optional arguments:
9 -h, --help      show this help message and exit
10 -i INPUT        input image file location
11 -t TARGET        input target file location
12 -s SEQUENCE    length of the morphing sequence
13 -b {True,False} whether to use bilinear
14 -a {True,False} use feature points label or dlib automatical detection
15 -x SHIFTX      shift x pixels for image after prewarping
16 -y SHIFTY      shift y pixels for image after prewarping

```

T 或者直接运行默认的 Test1

```

1 (dip1) D:\Courses\DIPI\exp1\code>python viewmorphing.py

```

4.2 Principle

Feature Points Detection 为了用 8-point 算法[1]计算 fundamental matrix，需要先确定 8 个特征点。这部分我采取了完全手动标点或者从 dlib 识别的结果中挑 8 个点的两种方式，具体使用哪种由后续 warping 结果的质量决定。

Compute Fundamental Matrix 使用 8-point 算法解 Fundamental Matrix，其定义为：

$$\mathbf{V}^T \mathbf{F} \mathbf{U} = \mathbf{0}$$

其中 \mathbf{V} 与 \mathbf{U} 是 source 和 target 中具有配对关系的两个点，记 $\mathbf{U} = (x, y, 1)^T$, $\mathbf{V} = (x', y', 1)^T$ ，可得对应方程：

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \mathbf{0}$$

展开后整理可得：

$$\begin{bmatrix} x'x & x'y & x' & y'x & y'y & y' & x & y & 1 \end{bmatrix} \mathbf{f} = 0$$

当给定 n 个点时将上述方程左端 1×9 的矩阵逐行叠加得到 $n \times 9$ 的矩阵 \mathbf{A} ，此时有：

$$\mathbf{Af} = \mathbf{0}$$

最后一列是固定的 1，这里 A 的自由度是 8。要求解上述方程可以分为三个步骤：

Normalization 不做 Normalization 的 8-point 算法会对噪声比较敏感，讲 8-point 算法的这篇论文其中一项结论就是做了 Normalization 后这个算法求解的效果可以比得上以往的一些常用的迭代解法。

具体过程就是每个点的横纵坐标分别减去横纵坐标的均值，使得原点转移到质心上，再对每个点的横纵坐标乘上一个缩放系数，使得所有点到原点的平均距离为 $\sqrt{2}$ 。

Linear Solution 接下来求线性解，这一步可以对 \mathbf{A} 做 SVD 分解， \mathbf{f} 的解即为 \mathbf{A} 矩阵最小奇异值对应的奇异向量，reshape 成 3×3 就得到了 \mathbf{F}

Constraint Enforcement 基本矩阵要保证奇异性，但是做完 Linear Solution 后得到的 \mathbf{F} 还不一定是奇异的，因此需要增加一个奇异性约束来修正得到的 F 。修正方法不是唯一的，比较简单做法是令：

$$\det \mathbf{F}' = 0$$

将使得

$$\|\mathbf{F} - \mathbf{F}'\|_{Frobenius}$$

最小的 \mathbf{F}' 作为最终的 \mathbf{F} 。具体实现的时候可以先对 \mathbf{F} 做 SVD 分解：

$$\mathbf{F} = \mathbf{U} \mathbf{D} \mathbf{V}^T$$

则

$$\mathbf{F}' = \mathbf{U} * \text{diag}(\mathbf{D}[0], \mathbf{D}[1], 0) * \mathbf{V}^T$$

Denormalization 最后做 Denormalization，也即把 Normalization 的过程反向操作一遍，即可最终得到 U 和 V 之间对应的 fundamental matrix

PreWarp prewarp 的目的是对齐 source 和 target 的视平面，形式化地来说就是根据给定的 \mathbf{F} ，要找到两个 projective warping \mathbf{H}_0 和 \mathbf{H}_1 ，使得

$$\mathbf{H}_1^{-T} * \mathbf{F} * \mathbf{H}_0^{-1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

得到 \mathbf{H}_0 和 \mathbf{H}_1 的方法同样也是很多的，其中一种是借助视距上的 rotation 变换来对齐平面。首先在 source 中选定一条 rotation axis

$$\mathbf{d}_0 = [d_0^x, d_0^y, 0]^T$$

当给定

$$\begin{bmatrix} x & y & z \end{bmatrix}^T = \mathbf{F} \mathbf{d}_0$$

时，那么 target 中对应的 rotation axis 就是

$$\mathbf{d}_1 = \begin{bmatrix} -y & x & 0 \end{bmatrix}^T$$

由 \mathbf{F} 和 \mathbf{F}^T 可以分别得到 epipoles e_0 和 e_1 ，即为 \mathbf{F} 和 \mathbf{F}^T 特征值为 0 对应的单位特征向量。从视图几何上来说，某一个 view 的 epipole 就是另一个 view 光心的 projection。

那么 rotation angle 可以由：

$$\theta_i = -\pi - \tan^{-1} \frac{d_i^y * e_i^x - d_i^x * e_i^y}{e_i^z}$$

给出，此时将使得 source 和 target 图像视平面对齐的矩阵变换分别记为：

$$\mathbf{R}_{\theta_i}^{d_i}, \quad i = 0, 1$$

此外还需要另一个 rotation 来使得 epipolar lines 是水平的：

$$e_{\tilde{i}} = \mathbf{R}_{\theta_i} * e_i, \quad i = 0, 1$$

相应的 rotation angle 由：

$$\phi_i = -\tan^{-1} \frac{e_{\tilde{i}}^y}{e_{\tilde{i}}^x}, \quad i = 0, 1$$

给出。将对应的矩阵变换记为

$$\mathbf{R}_{\phi_i}, \quad i = 0, 1$$

最终的 prewarp 变换即为：

$$\mathbf{H}_i = \mathbf{R}_{\phi_i} \mathbf{R}_{\theta_i}^{d_i}, \quad i = 0, 1$$

Delaunay Triangulation 同 Face Morphing 一节，只不过做剖分的对象变成了 PreWarp 后的图。

Morphing 同 Face Morphing 一节，不再赘述。

PostWarp 最后要将 Morphing 后的结果再做一次 Projective Warping 得到一个合适的视平面。关于 Projective Warping 以及要用到的 Bilinear 插值在 Image Warping 的作业中也已经实现过，这里直接搬过来用了。

这一步特征点的选择 View Morphing 的论文[2]中说手动标，不过我觉得做这种单类别小图像，也即图像的主体部分只有人脸的 Warping 完全可以将 Morphing 后得到的不规则四边形的四个顶点作为特征点，然后做 Projective Warping 到原图大小的四个顶点，这样省掉了手动再标的过程，而且效果也还可以接受。

4.3 Result

4.3.1 Test1

```
(dip1) D:\Courses\_DIP\exp1\code>python viewmorphing.py -i ./images/part2-2/source_1.png
-t ./images/part2-2/target_1.png -s 5 -b True -a False -x 5 -y 200
```



(a) Test1 Source

(b) Test1 Target

Figure 7: Test1 Landmark Points

[Test1](#)我是手动标的8个点,如果从dlib检测的68个点中来挑8个点得到Prewarp的效果不太好。可能是因为脸的视角是斜着的或者图片分辨率太低,dlib检测出来的landmark points偏差比较大,特别是脸颊的轮廓到下巴的部分,同时这个算fundamental matrix的算法数值敏感性又比较高,landmark points偏一点算出来的fundamental martrix就完全不同了,所以我干脆手动标了。

接着是[PreWarp](#)的结果,这里在具体实现上有个问题,是做了Projective Warping后图像的尺寸可能变化很大。对于Test1来说至少要把原图宽度扩大为1.25倍才能装得下PreWarp结果的所有像素点,而且换一张图测试这个比例也会发生变化,可能扩大到原来的数十倍,也可能会缩小到原来的几分之一,所以这一步我目前只能做一遍Warping,根据结果再反过来调整结果的大小。

除了尺寸的变化外,还有横纵坐标的移位对于某些图片也是必要的,例如这里Test1的PreWarp就将沿纵轴正向整体移了200个像素,不然Warping后的右上角纵坐标是负的,仍然无法保留到全部的像素。

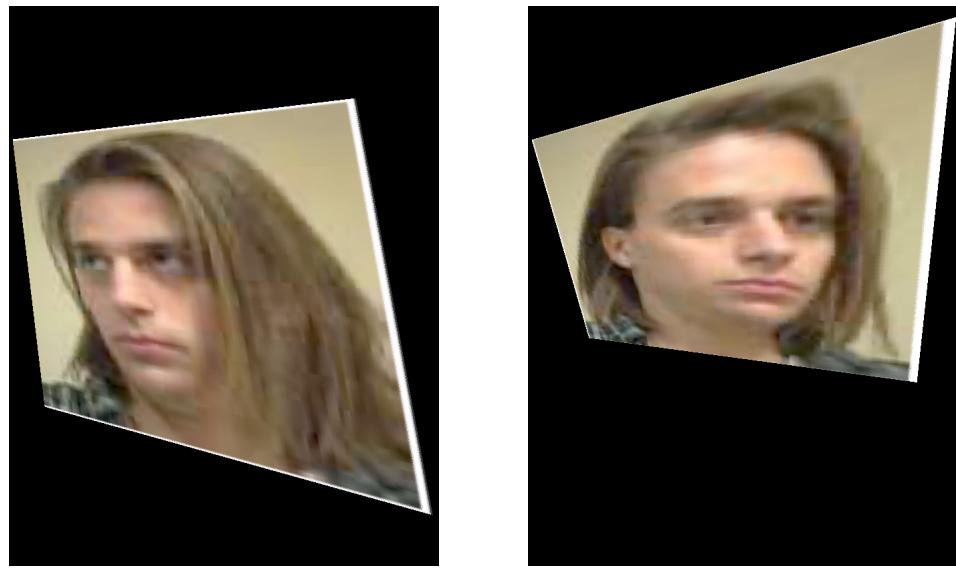
然后再[Delaunay Triangulation](#)

再做[Face Morphing](#)

最后做[PostWarp](#)

总的来说我认为最终效果还是可以接受的,但是当 α 接近0.5时artifacts也是比较明显的。主要的原因是PreWarping后用dlib检测特征点偏差比较大,可以参见[Figure 9](#)的右图,人脸的过分扭曲使得dlib检测的脸颊轮廓到下巴部分的特征点偏差比较大,这导致后续根据三角剖分的结果做Morphing会出现重影的artifact。

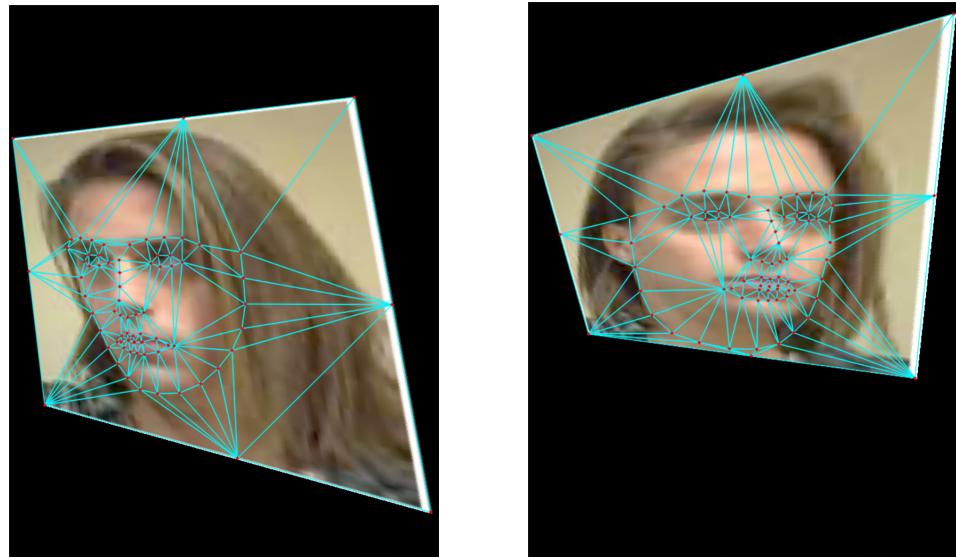
这个问题其实也比较好解决,就是像上一个小节Face Morphing一样做手动标点,这里用的是和Face Morphing完全相同的算法来做的,手动标点做出的结果从Face Morphing一节中可以看出来是相对更好的。但是考虑到整个Face Morphing Pipeline的自动化程度(其实也有一部分原因是六十几个点我懒得再标了),我还是决定这一个步骤使用dlib检测的结果而不是去手动标点。



(a) Test1 Source PreWarp

(b) Test1 Target PreWarp

Figure 8: Test1 PreWarp



(a) Test1 Source Delaunay Triangulation

(b) Test1 Target Delaunay Triangulation

Figure 9: Test1 Delaunay Triangulation



Figure 10: Test1 Morphing



Figure 11: Test1 Final Result with PostWarp

4.3.2 Test2

Test2 的 fundamental matrix 算不准，PreWarp 后结果长这样，这是已经放大过两倍的图，尺寸大约为 1200×1100 ，但是要装下 PreWarp 后的全部像素还差得很远。Test2 我挑了很多组特征点，PreWarp 后需要的图片尺寸还是太大。因此我放弃了 Test2，然后另外做了一个 Test3 作为补充。

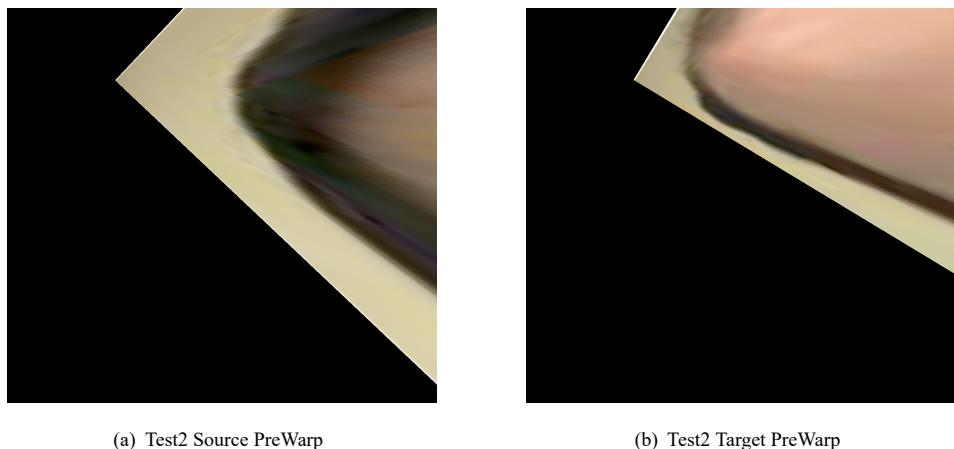


Figure 12: Test2 PreWarp

4.3.3 Test3

```
(dip1) D:\Courses\DIPI\exp1\code>python viewmorphing.py -i ./images/part2-2/source_3.png  
-t ./images/part2-2/target_3.png -s 5 -b True -a True -x 100 -y 100
```

整个 Pipeline 的步骤除了第一步特征点变成了由 dlib 自动检测外，其他的都和 Test1 一样，这里就不再赘述了，直接贴出结果，见[Figure 13](#)到[Figure 16](#)

总的来说效果还可以接受，但是 Test3 也出现了重影的 artifacts，而且这里的原因和 Test1 不太一样。Test1 是因为 landmark 标定不准，这里是因为 landmark 完全没有标头发的部分，所以做三角剖分后头发和背景会在同一个三角形内，此时再做 Morphing 那么两个方向的头发就会重叠在一起造成 artifacts，不过人脸主要部分的 Morphing 效果还是正常的。

这个问题的解决方法仍然是手动标点，把头发的边界也标记到 landmark points 列表中，或者直接把人像从背景里抠出来再做 Morphing。

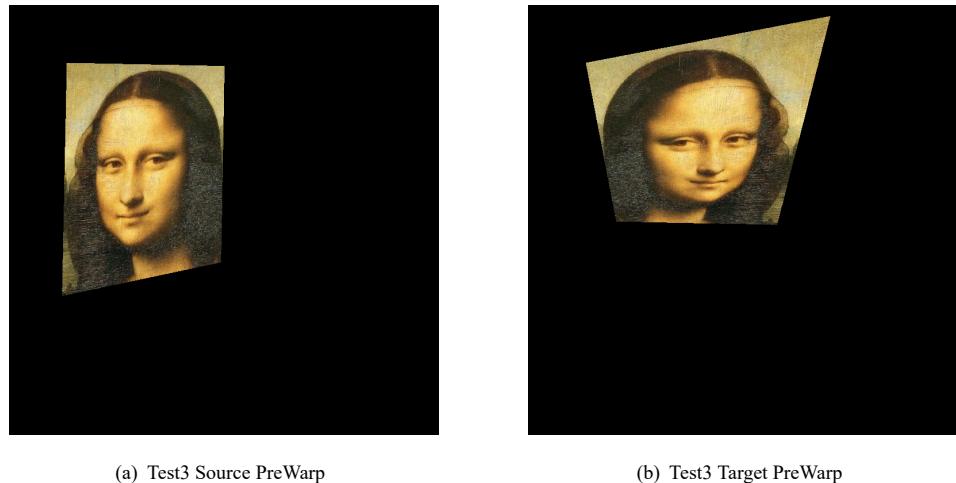


Figure 13: Test3 PreWarp

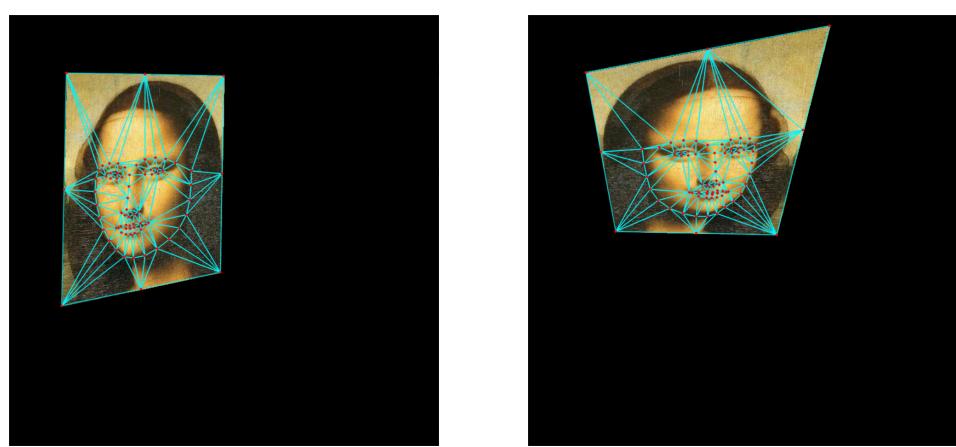


Figure 14: Test3 Delaunay Triangulation

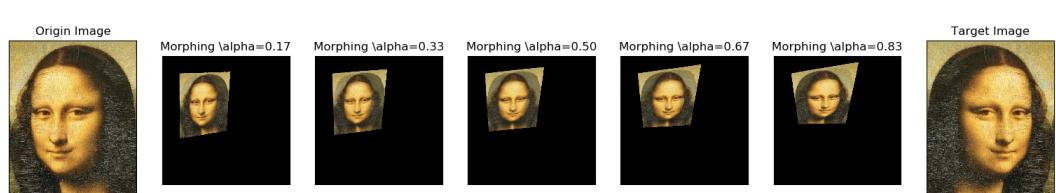


Figure 15: Test3 Morphing



Figure 16: Test3 Final Result with PostWarp

5 Conclusion

这次大作业，特别是 View Morphing 的部分用到了很多课堂上没讲过的内容，例如多视图几何，SIFT 和 RANSAC（开始打算用 SIFT 做特征点标定所以去学了一下，后来发现没有 dlib 效果好就换成了 dlib），fundamental matrix 相关的线性代数，8-point 算法等等，总的来说收获还是很大的。当然 View Morphing 我做的还有欠缺，对于前文提到的一些改进方法之后还可以做进一步的尝试。

6 References

- [1] Hartley, Richard I. "In defense of the eight-point algorithm." IEEE Transactions on pattern analysis and machine intelligence 19.6 (1997): 580-593.
- [2] Seitz, Steven M., and Charles R. Dyer. "View morphing." Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. 1996.