

浙江大学

课程名称: 计算机动画

姓 名: 赵恒

学 院: 计算机科学与技术学院

专 业: 信息安全

学 号: 3210101638

指导教师: 金小刚

2023 年 10 月 29 日

浙江大学实验报告

课程名称: 计算机动画

实验类型: 综合

实验项目名称: 基于 Delaunay 三角剖分的二维人脸 morphing

学生姓名: 赵恒 专业: 信息安全 学号: 3210101638

同组学生姓名: 无 指导老师: 金小刚

实验地点: 玉泉 实验日期: 2023 年 10 月 29 日

0 参考代码和改进

本项目（暂未开源）主要基于 Face Morph Using OpenCV – C++ / Python（官方教程）实现思路，从 0 搭建整个 morphing 流水线（其中一些功能函数继承自官方版本，修改量 $\approx 80\%$ ）

官方源代码 [链接](#) 或参见项目中 reference 文件夹

- 官方代码只给出了 facemorph 文件的 python 和 C++ 版本，我采用参考 faceMorph.py。其中包含了：
 - 函数 readPoints：从 txt 文件中读取事先存好的面部地标；
 - 函数 applyAffineTransform：对配对三角形做仿射变换；
 - 函数 morphTriangle：扭曲以及 alpha 融合两张图片的对应三角区域
- 改进：
 - 给源文件 faceMorph.py 增加参数并改进重构代码；

- 增加面部地标识别和存储；(见 `landmark_detector.py`, `draw_delaunay.py` (该代码可以生成 Delaunay 测量后的图片以及图片对应的 Voronoi 图以便于理解 morphing 原理))
 - 其中 `landmarkdetector.py` 中增加对多脸的选择判定逻辑 (即存在多脸时选择最清晰的人脸作为 morphing 对象)
- 增加基于已识别的面部做裁剪使得变换更具有鲁棒性(在脚本中为可选项); (见 `align_images.py`)
- 将中间帧输出为一个视频文件。(`morphing.sh` 中实现)
- 用 Shell `morphing.sh` 脚本将 morphing 过程形成一个完整的流水线。

代码中只有注释有 **[Official code:]** 的函数为继承自官方教程的代码

1. 实验目的和要求

- 实现基于网格的二维图像 morphing;
- 以基于网格变形的方式把一幅数字图像以一种自然流畅、超现实主义的方式变换到另一幅数字图像。

2. 实验内容和原理

2.1 实验内容

1. 使用 `dlib` 对两张图片做人脸地标识别;
2. 对人脸做 Delaunay 三角剖分;
3. 对两张图片的 Delaunay 三角形做仿射变换，并对两张图片对应的三角形之间做 Alpha blending 融合
 - 根据不同 α 值重复 3 步骤得到一系列中间帧;
4. 将中间帧合成一个视频，即可观察到自然的图像 morphing 过程。

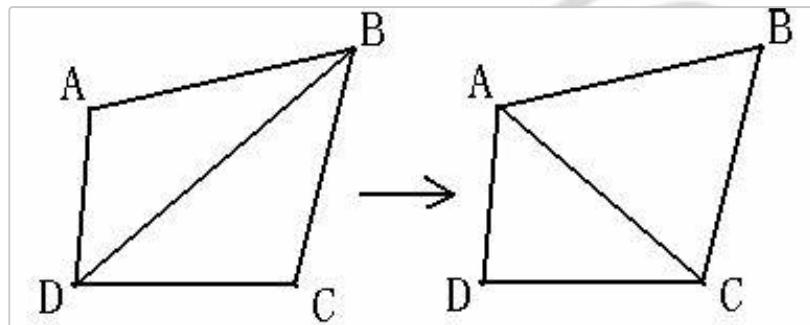
2.2 实验原理

2.2.1 Detecting facial landmarks

- 使用 `dlib` 来检测面部坐标, `dlib` 中的 `facial landmark detector` 基于 `One millisecond face alignment with an ensemble of regression trees`
- 使用 `dlib` 生成 68 对面部特征点, 并将坐标保存到 `txt` 文件中。

2.2.2 Triangular Delaunay Segmentation

Delaunay 三角剖分算法的目的是最大化三角剖分中三角形中最小角, 目的是避免“极瘦”的三角形的出现 (即避免下图左侧的划分方式)



2.2.3 Affine Transformation

- 仿射变换即另外两种简单变换的叠加: 一个是线性变换 R , 一个是平移变换 T 。
- 其计算方法为坐标向量和变换矩阵的乘积, 换言之就是矩阵运算。在应用层面, 仿射变换是图像基于 3 个固定顶点的变换。
- 在二维图像变换中, 一般表示为:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} R_{00} & R_{01} & T_x \\ R_{10} & R_{11} & T_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (1)$$

2.2.4 Alpha Blending

- Alpha Blending 主要用于解决“图层”叠加问题。其中的 `alpha` 表示透明度，范围为 $[0, 255]$ (0 表示全透明， 1 表示不透明)
- 由此对于任意两张图片 1 和 2 ，使用 Alpha Blending 将其融合成图片 3 ，可以表示为

$$\text{RGB}_3 = (1 - \alpha) * \text{RGB}_1 + \alpha * \text{RGB}_2 \quad (2)$$

α 为混合透明度（取值 $[0, 1]$ ）， RGB_i 表示图 i 的像素值

3. 实验器材

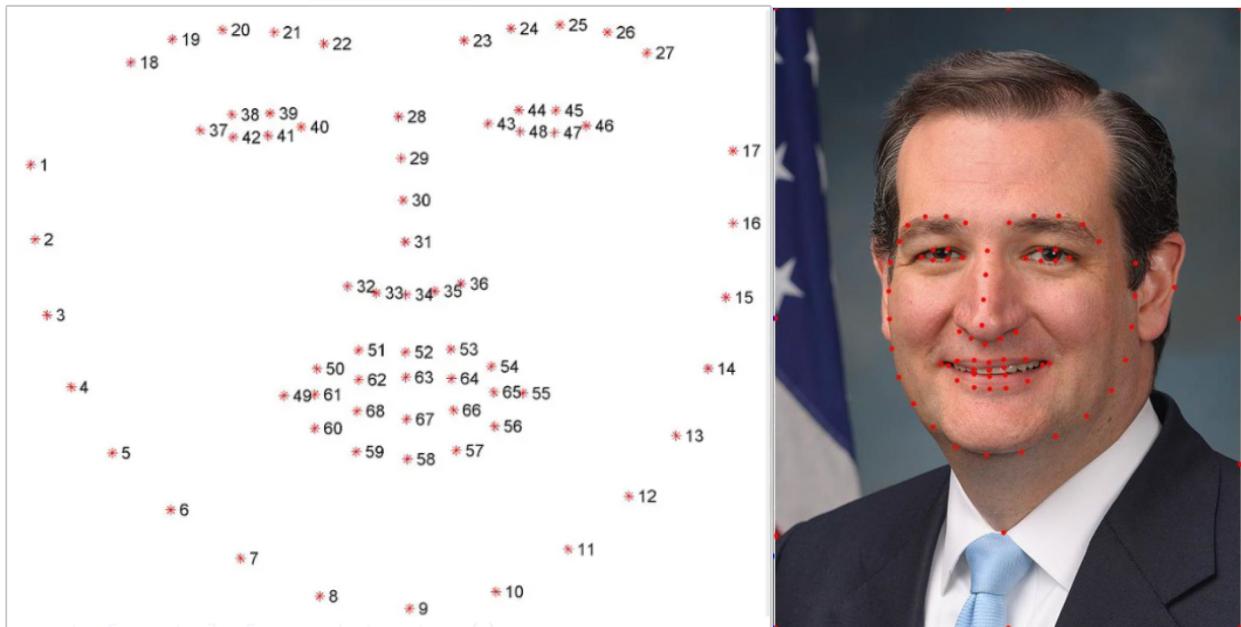
- 「实验编译运行平台」：`python=3.11 & MacOS`
 - 相关依赖包：`opencv-python=4.8.1.78`，`dlib=19.24.2`，`imutils=0.5.4`，`numpy=1.26.1`，`Pillow=10.1.0`，`scipy=1.11.3`（皆为当前实验时刻最新版）
 - 「实验工具软件」：
 - `ffmpeg`：用于将帧图片转换为 `mp4` 文件（version 6.0 built with Apple clang version 15.0.0）
- 虽未尝试在 `Ubuntu` 上运行代码，但理论上也是可迁移的；
- 可以用以下代码安装 `Ubuntu` 依赖（如果没有正常运转的话，可以参考 [官方文档](#)）

```
1 sudo apt-get install build-essential cmake pkg-config
2 sudo apt-get install libx11-dev libatlas-base-dev
3 sudo apt-get install libgtk-3-dev libboost-python-dev
4 sudo apt-get install python-dev python-pip python3-dev python3-
    pip
5 sudo -H pip2 install -U pip numpy
6 sudo -H pip3 install -U pip numpy
7 pip install imutils
8 pip install dlib
```

4. 实验步骤

4.1 Step One

使用 `dlib` 生成 68 对面部特征点，并将坐标保存到 `txt` 文件中。此外，选取一些面部外的点位来帮助更平滑地 `morphing`



实现该部分的关键代码如下：

```
1 # partial-code
2 ...
3 # initialize dlib's face detector (HOG-based) and then create
4 # the facial landmark predictor
5 detector = dlib.get_frontal_face_detector()
6 predictor =
7     dlib.shape_predictor("code/shape_predictor_68_face_landmarks.da-
8 t")
9 ...
9 # loop over the detected faces:
10 for (i, rect) in enumerate(dets):
11     # detect the facial landmarks for the face region,
12     # and convert (x, y)-coordinates to a NumPy array
13     shape = face_utils.shape_to_np(predictor(image_gray, rect))
```

```
14 ...
15 ...
```

- 这里使用 8 个额外的点位 (4 个角, 4 条边的中心点)

实现该部分的代码如下：

```
1 def addAdditionalPoints(face_points, size) :
2     """
3         Append 8 additional points: corners and half way points to the
4         face_points list
5     """
6
7     height = size[0]
8     width = size[1]
9     middle_height = height // 2
10    middle_width = width // 2
11
12    # Corners
13    face_points.append((0, 0))
14    face_points.append((0, height - 1))
15    face_points.append((width - 1, 0))
16    face_points.append((width - 1, height - 1))
17
18    # Half way points
19    face_points.append((0, middle_height))
20    face_points.append((middle_width, 0))
21    face_points.append((width - 1, middle_height))
22    face_points.append((middle_width, height - 1))

23
24    return face_points
```

- 当遇到多个人脸的时候，会判断哪个人脸是最清晰的，并将最合适的人脸当作 morphing 的对象并只保存其面部地标
 - 如下图所示：



实现该部分的关键代码如下：

```
1 # partial-code
2 # calculate the sharpness of the current face
3 sharpness = cv2.Laplacian(image_gray[rect.top():rect.bottom(),
4 rect.left():rect.right()], cv2.CV_64F).var()
5
6 # if this is the first face or if it's sharper than the
7 previous ones, update the variables
8 if sharpness > max_sharpness:
9     max_sharpness = sharpness
10    max_sharpness_shape = shape
11    max_sharpness_rect = rect
12 ...
13 ...
```

4.2 Step Two

根据上一步得到的 76 个点位做 Delaunay 三角测量，输出结果是由点位数组中的点的索引值表示的三角形列表，由此有了两张图片的三角形对应区域；

- Delaunay 三角测量的结果和其对应的 Voronoi 图如下：



实现该部分的关键代码如下：

```
1 # partial-code
2 def build_delaunay(image, points) :
3     """
4     Gets delaunay 2D segmentation and return a list with the the
5     triangles' indexes
6     """
7     subdiv = cv2.Subdiv2D((0, 0, image.shape[1], image.shape[0]))
8     for point in points :
9         subdiv.insert(point)
10
11     triangle_list = subdiv.getTriangleList().astype(np.int32)
12     delaunay_triangles = []
13     for p in triangle_list :
14         vertexes_index = [0, 0, 0]
15
16         for v in range(3) :
17             vertex = (int(p[v * 2]), int(p[v * 2 + 1]))
18             if vertex in points :
19                 vertexes_index[v] = points.index(vertex)
20
21             delaunay_triangles.append(vertexes_index)
22
23     return delaunay_triangles
```

4.3 Step Three

从图 I 变换到图 J ，在建立三角对应列表后，需要计算在 `morphed` 图像中特征点的位置 (x_m, y_m) （共 76 组），使用下列方程：

$$\begin{aligned}x_m &= (1 - \alpha)x_i + \alpha x_j \\y_m &= (1 - \alpha)y_i + \alpha y_j\end{aligned}\tag{3}$$

实现该部分的关键代码如下：

```
1 ...  
2 # Compute weighted average point coordinates  
3 for i in range(0, len(face1_points)):  
4     x = (1 - alpha) * face1_points[i][0] + alpha * face2_points[i]  
        [0]  
5     y = (1 - alpha) * face1_points[i][1] + alpha * face2_points[i]  
        [1]  
6     points.append((x, y))  
7 ...
```

4.4 Step Four

由第三步，我们在图 I （原始图像）， J （目标图像）， M （中间图像）中各有一组 76 个点同时还有在这些点位基础上得到的三角形列表。

1. 选择图 I 中的三角形和图 M 中的相应三角形，并计算将图 I 中三角形的三个角映射到图 M 中相应三角形的三个角的仿射变换。（在 OpenCV 中，该处理过程可以使用 `getAffineTransform` 完成）
2. 最后，重复图 J 和图 M 的过程。

4.5 Step Five

- 对于图 I 和图 J 中的每个三角形，使用第四步计算的仿射变换将三角形内的所有像素转换为图 M ，以获得图 I 和图 J 的扭曲版本。在 OpenCV 中，该处理过程可以使用函数 `warpAffine` 来实现的。
- `WarpAffine` 的处理思路是计算三角形的边界框，使用 `warpAffine` 扭曲边界框内的所有像素，然后掩盖三角形外的像素。（三角形掩码是使用 `fillConvexPoly` 创建的）

实现 Step 4, 5 部分的关键代码如下：

```
1 def applyAffineTransform(src, src_Triangle, dst_Triangle, size)
2 :
3     """
4     [Official code]:
5     Apply affine transform calculated using src_Triangle and
6     dst_Triangle to src and output an image of size.
7     """
8
9     # Given a pair of triangles, find the affine transform.
10    warpMat = cv2.getAffineTransform(np.float32(src_Triangle),
11                                    np.float32(dst_Triangle))
12
13    # Apply the Affine Transform just found to the src image
14    dst = cv2.warpAffine(src, warpMat, (size[0], size[1]), None,
15                         flags=cv2.INTER_LINEAR, borderMode=cv2.BORDER_REFLECT_101)
16
17    return dst
```

4.6 Step Six

在第五步中，我们获得了图 I 和图 J 的扭曲版本。这两个图像可以使用方程 (2) 进行 alpha 混合得到一个变形图像。

$$M(x_m, y_m) = (1 - \alpha)I(x_i, y_i) + \alpha J(x_j, y_j) \quad (4)$$

实现该部分的关键代码如下：

```
1 # Alpha blend rectangular patches
2 imgRect = (1.0 - alpha) * warpImage1 + alpha * warpImage2
```

4.7 Step Seven

通过调整 α 的值重复 3-6 步得到一系列中间帧，并使用 `ffmpeg` 融合成一个视频文件

实现该部分的关键代码如下：

```
1 for (f, a) in enumerate(alpha_values) :
2     alpha = float(a) / 100
3     points = []
4
5     # Compute weighted average point coordinates
6     for i in range(0, len(face1_points)):
7         x = (1 - alpha) * face1_points[i][0] + alpha *
face2_points[i][0]
8         y = (1 - alpha) * face1_points[i][1] + alpha *
face2_points[i][1]
9         points.append((x, y))
10
11    # Allocate space for final output
12    imgMorph = np.zeros(img1.shape, dtype = img1.dtype)
13 d
14    for vertex1, vertex2, vertex3 in delaunay_group :
15        triangle1 = [face1_points[vertex1], face1_points[vertex2],
face1_points[vertex3]]
16        triangle2 = [face2_points[vertex1], face2_points[vertex2],
face2_points[vertex3]]
17        triangle = [points[vertex1], points[vertex2],
points[vertex3]]
18
19        # Morph one triangle at a time.
20        morphTriangle(img1, img2, imgMorph, triangle1, triangle2,
triangle, alpha)
21
22    # Save morphing frame
23    index = str(f).zfill(4)
```

```
24 cv2.imwrite(out_dir1 + '/morph-' + img_name1 + '-' + img_name2  
+ '-' + index + '.png', np.uint8(imgMorph))
```

4.8 Other Implementation

4.8.1 Face Align

当人脸在图片中面积占比较小的似乎，morphing 的效果变得不是很显著，影响观感。可以选择在 morphing 前进行 face align 操作，即使鼻子居中、眼睛水平，然后重新缩放生成的图像

该步的算法主要参考修改自 [GitHub](#) (代码较长，此处略去)

4.8.2 Max Sharpness Face

由于本项目做的是两张图片人脸的 morphing 操作，当一张图片中有多个脸的时候，因为采取的面部地标保存策略是覆写文件。最后 morph 的对象是最后一个识别的人脸，而其往往不是图片中最清晰的。

虽然可以采取手动输入序号来选择将哪张人脸作为 morph 的对象，当会使得程序不完全自动化。因此，我选择对图片中最清晰的人脸做面部地标保存并作为 morphing 的对象以使程序更自动化且更具有鲁棒性。

实现该部分的关键代码如下：

```
1 # initialize variables to keep track of the clearest face found  
so far  
2 max_sharpness = -1  
3 max_sharpness_shape = None  
4 max_sharpness_rect = None  
5  
6 # loop over the detected faces  
7 for (i, rect) in enumerate(dets):  
8     # detect the facial landmarks for the face region,  
9     # and convert (x, y)-coordinates to a NumPy array  
10    shape = face_utils.shape_to_np(predictor(image_gray, rect))  
11
```

```
12 # calculate the sharpness of the current face
13 sharpness = cv2.Laplacian(image_gray[rect.top():rect.bottom(),
14 rect.left():rect.right()], cv2.CV_64F).var()
15
16 # if this is the first face or if it's sharper than the
17 # previous ones, update the variables
18 if sharpness > max_sharpness:
19     max_sharpness = sharpness
20     max_sharpness_shape = shape
21     max_sharpness_rect = rect
22
23 # check if any faces have been found
24 if max_sharpness_shape is None:
25     print('\033[0;31mWarning! no faces have been detected\033[0m')
26     exit()
```

4.8.3 Draw Delaunay

该部分是基于先前识别的面部地标，生成 Delaunay 三角形列表，并根据列表画出对应的 Delaunay Triangle 图和 Voronoi 图来帮助更好地理解该项目是如何基于 Delaunay 三角做 morphing 的。

实现该部分的关键代码如下：

```
1 # Draw delaunay triangles
2 def draw_delaunay(img, subdiv, delaunay_color):
3
4     triangleList = subdiv.getTriangleList()
5     size = img.shape
6     r = (0, 0, size[1], size[0])
7
8     for t in triangleList :
9         pt1 = (int(t[0]), int(t[1]))
10        pt2 = (int(t[2]), int(t[3]))
11        pt3 = (int(t[4]), int(t[5]))
```

```
13     if in_rectangle(r, pt1) and in_rectangle(r, pt2) and
14         in_rectangle(r, pt3) :
15             cv2.line(img, pt1, pt2, delaunay_color, 1,
16                     cv2.LINE_AA, 0)
17             cv2.line(img, pt2, pt3, delaunay_color, 1,
18                     cv2.LINE_AA, 0)
19             cv2.line(img, pt3, pt1, delaunay_color, 1,
20                     cv2.LINE_AA, 0)
21
22
23
24 def draw_voronoi(img, subdiv) :
25     """
26     Draw voronoi diagram
27     """
28
29     (facets, centers) = subdiv.getVoronoiFacetList([])
30
31
32     for i in range(0, len(facets)):
33         ifacet_arr = []
34         for f in facets[i] :
35             ifacet_arr.append(f)
36
37
38         ifacet = np.array(ifacet_arr, np.int64)
39         color = (random.randint(0, 255), random.randint(0, 255),
40                  random.randint(0, 255))
41
42         cv2.fillConvexPoly(img, ifacet, color, cv2.LINE_AA, 0)
43         ifacets = np.array([ifacet])
44         cv2.polylines(img, ifacets, True, (0, 0, 0), 1,
45                       cv2.LINE_AA, 0)
46         cv2.circle(img, (int(centers[i][0])), int(centers[i][1])),
47                     3,
48                     (0, 0, 0), cv2.FILLED, cv2.LINE_AA, 0)
```

5. 实验结果

5.1 实验结果展示：

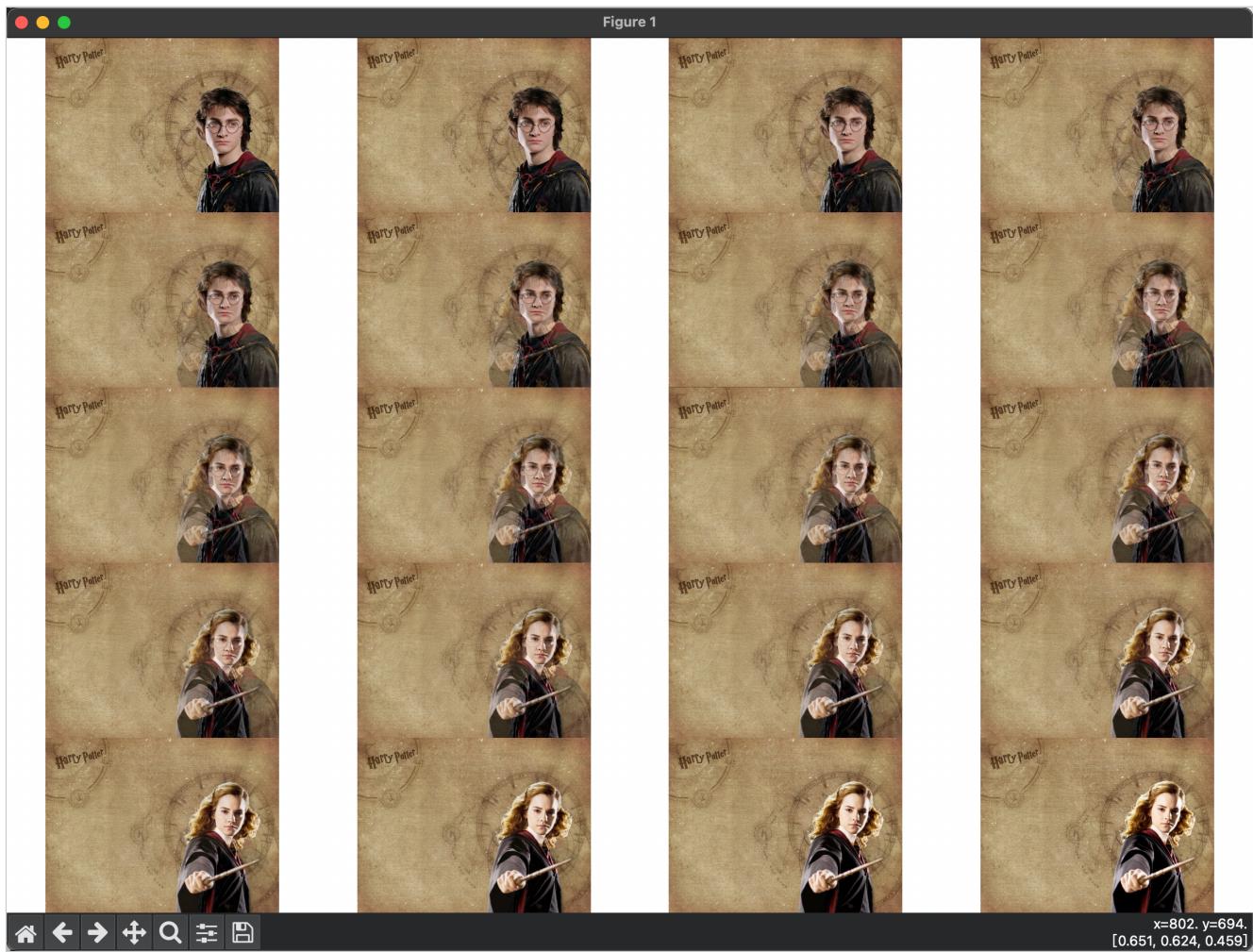
- 直接运行根目录下的 `Shell` 脚本并设置对应参数即可运行程序
- 个别情况下，当前环境只有 `python3` 时，请将 `Shell` 脚本中的 `python` 命令全部替换为 `python3` or 对当前环境中的 `python3` 做一个软链接
- 假设从 `example/harry.jpg` 向 `example/hermione.jpg` 做 morphing
- 最终实验结果见文件夹 `example/` 中的 `gif` 文件

5.2 中间文件：

- `example/harry.txt` 和 `example/hermione.txt` 为保存有两图片面部地标的文件：

	harry.txt	hermione.txt
1	896 472	939 379
2	895 495	937 401
3	898 520	936 425
4	903 544	937 446
5	910 568	944 466
6	921 592	957 485
7	934 613	975 500
8	950 629	992 515
9	973 633	1010 519
10	999 631	1026 519
11	1021 618	1042 509
12	1041 601	1056 495
13	1057 582	1068 480
14	1069 560	1078 463
DEV	1075 534	1085 446
15	1080 507	1089 429
16	1082 479	1092 411
17	899 452	963 375
18	910 441	976 369
19	927 442	991 372
20	943 446	1005 376
21	959 454	1021 383
22		

- `example/morph-harry-hermione-%04d.png` 为中间帧（以生成 20 张中间帧为例）



6. 实验感想与心得

这次作业做的是基于 Delaunay 三角剖分的二维人脸 morphing。最初做这个项目的时候还是比较迷茫的，即便知道实验的原理了，但具体该如何实现还是无从下手。

后来在搜索资料的过程中得知了 OpenCV 库，具有一些内置的函数可以帮忙处理一些操作，并有一个原理较为详细的教程。这时我以为这个项目做起来应该会很简单了，但当把源码下载下来开始的时候，发现事情没有那么简单，官方虽然给出了代码和教程，但代码实际就是调用一些定义好的函数，像 morphing 的前期后期还有细节处理都是缺失的。但是有了方向，就好努力了。通过搜索 python 相关库的使用方法，我先后将 `Landmark_detector`，`draw_delaunay` 搭建起来。（中间过程很艰辛.....）

看到基于面部地标绘制出来的 Delaunay 图时，还是很有成就感的，不仅印证了地标识别保存的正确性，和图交互画图的正确性也得到了印证。下面便正式开启了 morphing 流水线，因为和官方的代码还是有点出入，再将自己的代码融合进去的时候还是遇到了一定的困难，幸运的是因为网络资源丰富，我很快搜索到了函数的正确用法并且完美地输出了 morphing 的中间帧。

下一步便是将中间帧融合成一个视频文件，这里我使用了 `ffmpeg`，效果还是很好的。在项目基本能运转的前提下，我增加了一个 `Shell` 脚本来使得整个项目更加流水线和自动化。最后，由于考虑到多人脸的特殊情况，我决定对代码中面部地标保存的部分做进一步修改，即只保存最清晰的人脸。

这次实验收获很大，不仅让我对 `Delaunay Triangle` 等理论知识有了较为深刻的理解，也锻炼了我的编程能力和发现及解决潜在问题的能力。