

DIP Experiment 1 Report

郑凯文

2021 年 4 月 21 日

1 Image Fusion

1.1 代码说明

`process.py` 主程序，可直接运行，参数说明见代码

1.2 算法实现与效果展示

1.2.1 Naive Poisson

Poisson Image Editing的基本思想是保持融合区域边界颜色一致的前提下，整体调整区域内的颜色，使得区域内的梯度与原先一致。设 Ω 是图A中的一部分，希望将其融合到图B的区域 S 中。设 Ω 原先在A图中的颜色函数为 g , S 在B图中的颜色函数为 f^* , 期望找到融合后 Ω 在B图中的颜色函数 f 满足：

$$\min_f \iint_{\Omega} |\nabla f - \nabla g|^2 \quad s.t. \quad f|_{\partial\Omega} = f^*|_{\partial\Omega} \quad (1)$$

其中 $\partial\Omega$ 为区域 Ω 的边界。在离散化的图像处理中，上式的有限差分离散化将成为一个二次规划问题，可以通过如下方法求解：

定义 N_p 为点 p 的4联通邻居集合，定义区域 Ω 的边界点集为

$$\partial\Omega = \{p \in S \setminus \Omega : N_p \cap \Omega \neq \emptyset\} \quad (2)$$

则 f 可通过如下方程组求解

$$\forall p \in \Omega, \quad |N_p|f_p - \sum_{q \in N_p \cap \Omega} f_q = \sum_{q \in N_p \cap \partial\Omega} f_q^* + \sum_{q \in N_p} v_{pq} \quad (3)$$

其中 $v_{pq} = g_p - g_q$ 。在实现中，程序读取用户提供的mask，自动解析 Ω 和 $\partial\Omega$ 中的点集（图1）并建立系数矩阵。系数矩阵的大小高达 $|\Omega| \times |\Omega|$ ，但每行至多有5个元素非0，因此我采用了`scipy.sparse`库进行储存和求解。

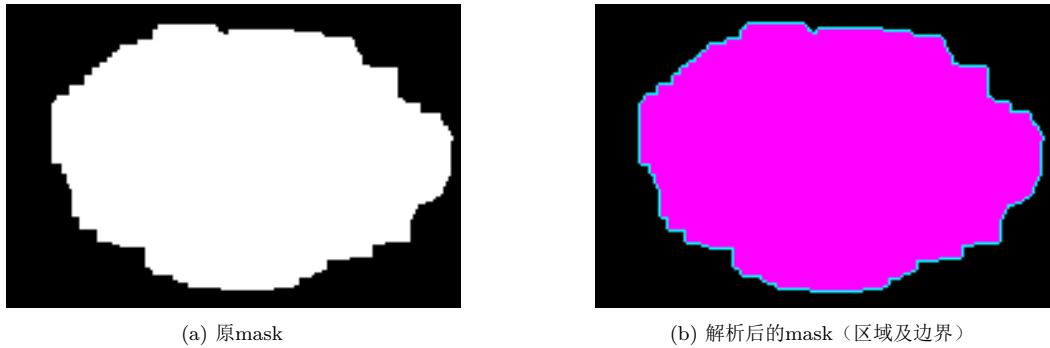


图 1: 提取区域及边界

对于提供的两个样本，效果如图2、3所示。



图 2: Demo1

1.2.2 Poisson with gradient mixture/adaptive gradient

朴素的Poisson方法存在一定的问题，即完全抹去了目标图B在区域 Ω 内的梯度信息，完全以图A的梯度取而代之。这在某些情况下会使效果大打折扣（图4、5）。

它们都出现了融合后不协调的感觉，这是因为目标图中丰富的背景被原图的梯度抹去了，显得模糊而突兀。对于这种情况，我尝试了两种解决方法：

- 使用一定比例的混合梯度 $\alpha \nabla g + (1 - \alpha) \nabla f^*$ 替代 ∇g

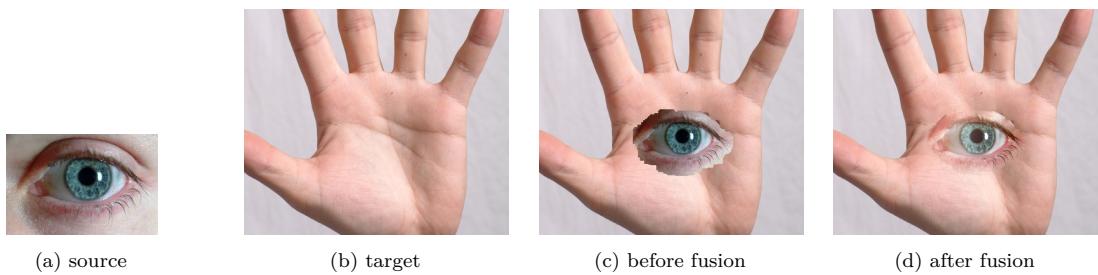


图 3: Demo2

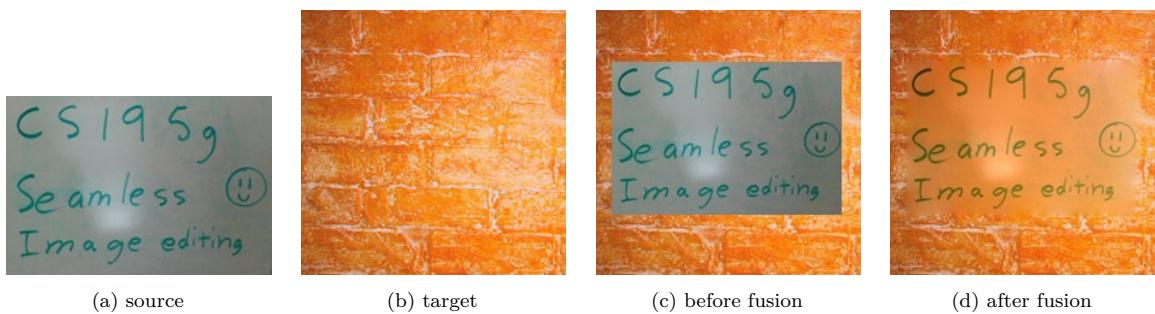


图 4: Demo3

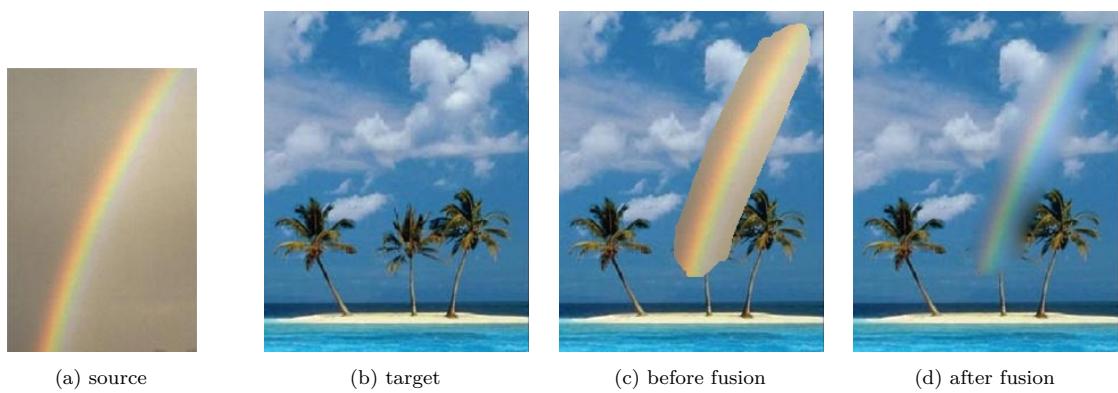


图 5: Demo4

- 使用自适应的梯度：取 ∇f^* 和 ∇g 中绝对值较大者

对于Demo3和Demo4，效果如下：

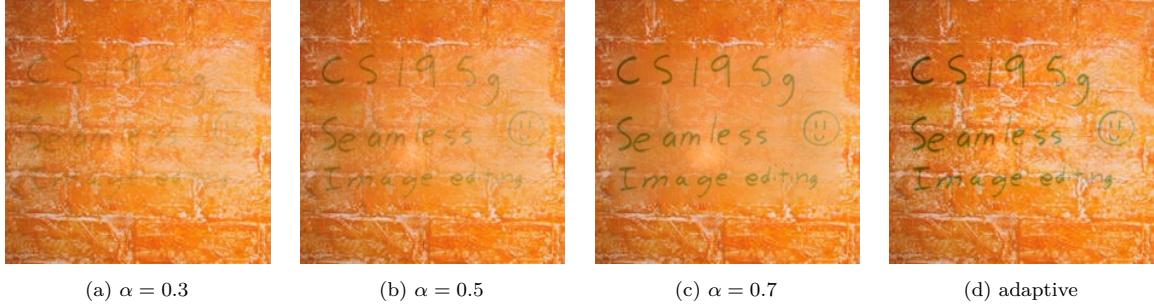


图 6: New Demo3

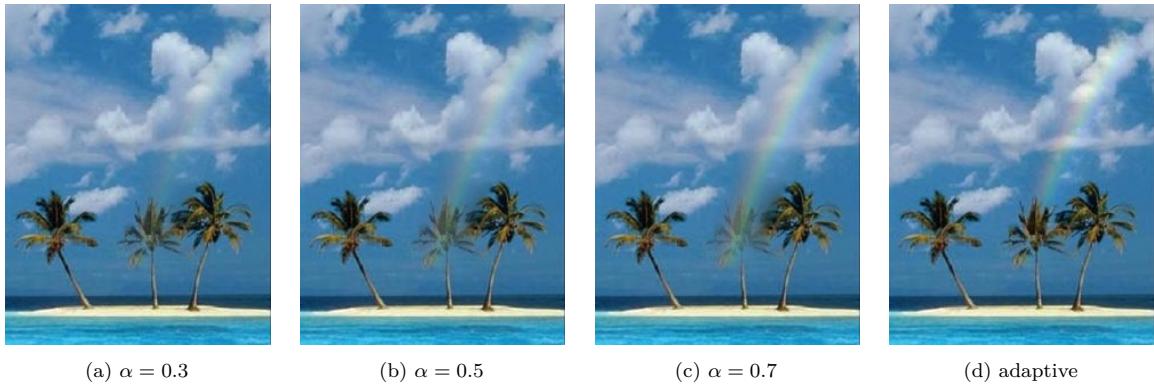


图 7: New Demo4

可以看到，对于Demo3，adaptive的效果最好。对于Demo4，adaptive和适当 α 值的效果都可以接受。综上而言，adaptive策略能够保留更多梯度信息，具有更好的效果。

2 Face Morphing

2.1 代码说明

face++_api.py 调用Face++ API，获取人脸关键点（需要同时提供源和目标，以维持自动去重时点的对应关系）

label.py 这是我利用OpenCV写的一个辅助标注小程序，对于无法自动标注的图像可以利用参照进行手动标注，示意图见图8。左键点击进行标注，右键撤销一步，全部标注完成后按下任意键，程序自动保存标注数据

delaunay.py 我实现了Delaunay三角剖分的Bowyer-Watson算法，它的基本思想是逐个加入点，对于加入的每个点，利用外接圆范围得到多边形边界，将其重建为星形三角网格的结构，复杂度 $O(n^2)$

`process.py` 利用前几个程序得到所需数据后，给定比例进行变换，可输出图片或视频

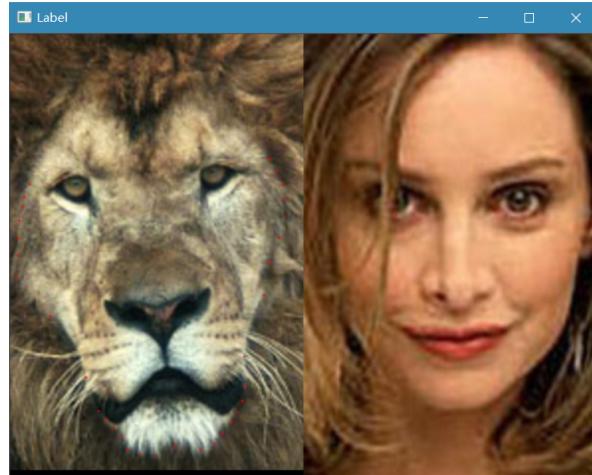


图 8: 手动标注工具（左图红点为已手工标注的点，右图红点为待标注的参照点）

2.2 算法实现与效果展示

2.2.1 Keypoints Deriving && Delaunay Triangulation

利用API或手动标注得到关键点，并进行Delaunay三角剖分。选择部分样本进行可视化，效果如下：

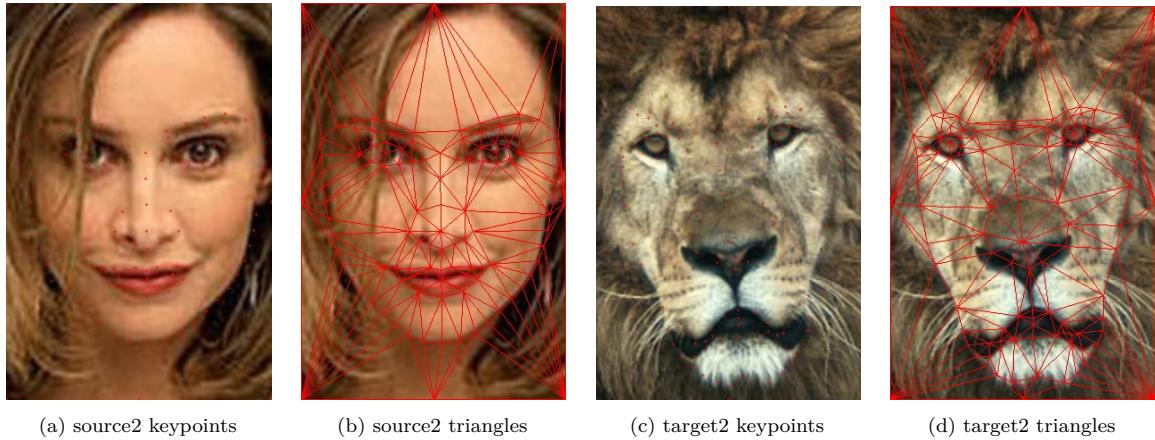


图 9: 可视化点与三角剖分

在进行三角剖分时，为方便下一步的变换，加入了图像边界上的8个点作为辅助关键点。

2.2.2 Affine Transformation

对源和目标进行三角剖分后，需要确定三角区域的对应关系。我这里采用的方法是利用源图三角剖分结果的每个三角形的顶点序号，由序号确定源、目标中的三角形坐标，它们互相对应。这样

做的前提是，源、目标的关键点必须一一对应。

设源三角形区域为 A ，目标三角形区域为 B ，找到中间态三角形区域 $M = (1 - \alpha)A + \alpha B$ ，确定 M 到 A 的仿射变换 T_1 和 M 到 B 的仿射变换 T_2 。这样，开始填充 M 中的每一点，对于某一点 p ， $T_1(p)$ 和 $T_2(p)$ 分别为源和目标图中的对应点，将它们的颜色以 α 为比例混合后得到 p 点的颜色。

图10、11、12展示了一些Morphing Sequence。可以看到，在变换过程中，脸之外的位置出现了虚影，这是由于这些区域没有标注点，只是利用和图像边界的辅助关键点构成的大三角形进行仿射变换，会有不精确的地方。同时，若源和目标有不同的特征（如戴眼镜与否），变换过程中必然有虚影出现。

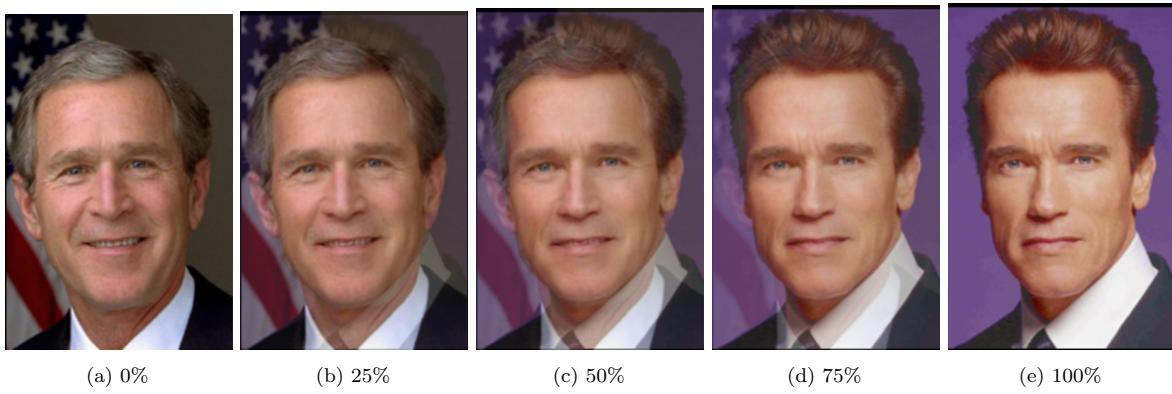


图 10: Demo1

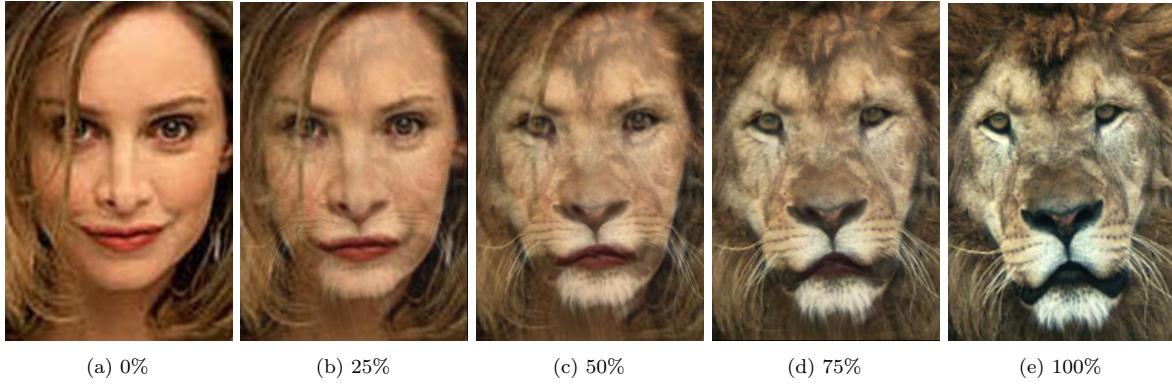


图 11: Demo2

对于这3个Morphing过程，我还相应地生成了3个24帧/秒*5秒的视频，在video目录下。

3 View Morphing

3.1 代码说明

`face++_api.py` 同上节

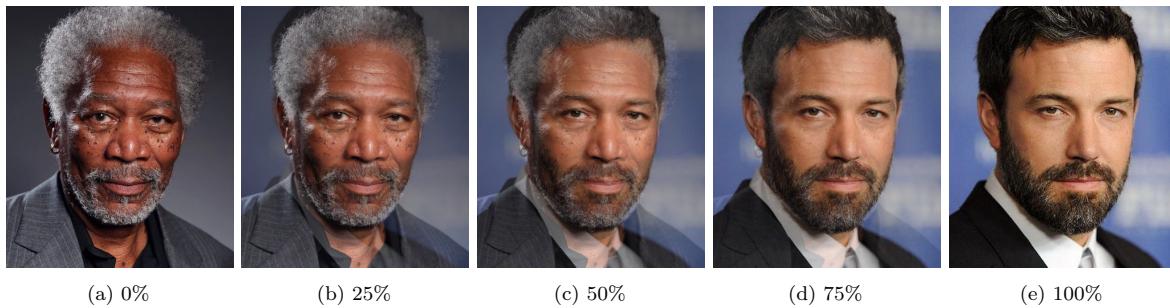


图 12: Demo3

delaunay.py 同上节

prewarp.py Pre-warp过程，输出变换后的图片与关键点

morph_postwarp.py Morph && Post-warp. 利用前3个文件的输出，生成图片或视频

3.2 算法实现与效果展示

算法分为三个阶段：

Pre-warp 通过投影变换，将源和目标变换至图像平面（投影平面）平行的位置

Morph 对Pre-warp后的图像进行morphing，过程同上节

Post-warp 再进行一次投影变换，得到最终图像

3.2.1 Prewarp

对于某个三维物体在不同视角的二维图像 I_0, I_1 , 存在一个基础矩阵 F , 使得对于两幅图像中任何一对对应点 $p_0 \in I_0, p_1 \in I_1$, 满足

$$p_1^T F p_0 = 0 \quad (4)$$

我们希望找到变换 H_0, H_1 , 使得变换后 I_0, I_1 的图像平面平行, 也就是

$$(H_1^{-1})^T F H_0^{-1} = \hat{F} \quad (5)$$

其中 \hat{F} 为基础矩阵的两幅图像的图像平面平行。在实现中，首先利用相对应的关键点求得基础矩阵 F ，这里使用cv2.findFundamentalMat函数。再由 F 求得 H_0, H_1 ，方法如下：

- 取 F, F^T 绝对值最小的特征值作为极点 e_0, e_1

- 沿轴 d_i 对图像 I_i 旋转角度 θ_i , 记这种旋转为 $R_{\theta_i}^{d_i}$ 。对 d_0 合适的选择是与 e_0 垂直, 即 $d_0 = (-e_0^y, e_0^x, 0)$, 令 $Fd_0 = (x, y, z)$, 则选择 $d_1 = (-y, x, 0)$ 。 θ_i 由下式确定

$$\theta_i = \tan^{-1} \frac{e_i^z}{d_i^y e_i^x - d_i^x e_i^y} \quad (6)$$

这样使得图像平面平行。

- 为了减轻下一步Morph的负担, 也为了避免瓶颈问题, 再对图像进行绕 z 轴的旋转 R_{ϕ_i} 使得极线平行。旋转角为

$$\phi_i = -\tan^{-1} \frac{\tilde{e}_i^y}{\tilde{e}_i^x} \quad (7)$$

其中 $\tilde{e}_i = R_{\theta_i}^{d_i} e_i$

- 最终的变换 $H_i = R_{\phi_i} R_{\theta_i}^{d_i}$

得到 H_0, H_1 后对源和目标进行变换。注意这种变换可能将图像像素范围映射到负数或很大的正数, 为了不损失信息, 我没有使用OpenCV的变换函数, 而是手动变换并添加偏移量实现图像大小的自适应匹配, 这样可以看到Pre-warp后的全貌。对于图像的关键点, 要做相同的变换。

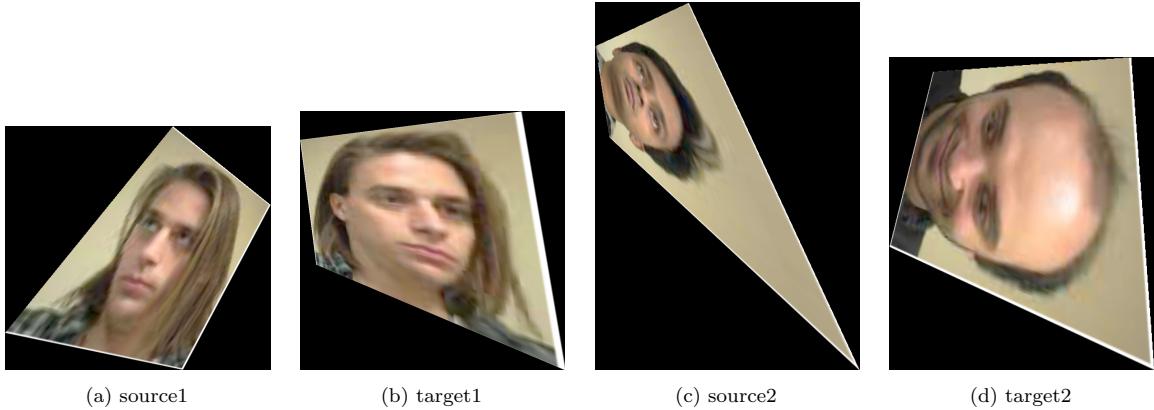


图 13: Prewarp

实际上, 由于关键点分布以及算法的局限, 很容易出现比较异常的情况。对于第二组图片, 我去掉了前25组关键点, 才得到了有效的Pre-warp结果。实际上对于target2的Pre-warp结果是有些问题的, 因为他本身是朝下看的, 变换后却朝下得更厉害了。这应该需要更有效的关键点。

3.2.2 Morph

对Pre-warp的结果进行上节相同的操作, 效果如图14、15。

3.2.3 Postwarp

对上一步的结果进行投影变换来把它“摆正”。由于我保留了Pre-warp后的所有信息, 这一步可以简单地解析Morph后图像的四个角的坐标, 把它们变换到用户给定宽、高后的四个角上。效果如图16、17。

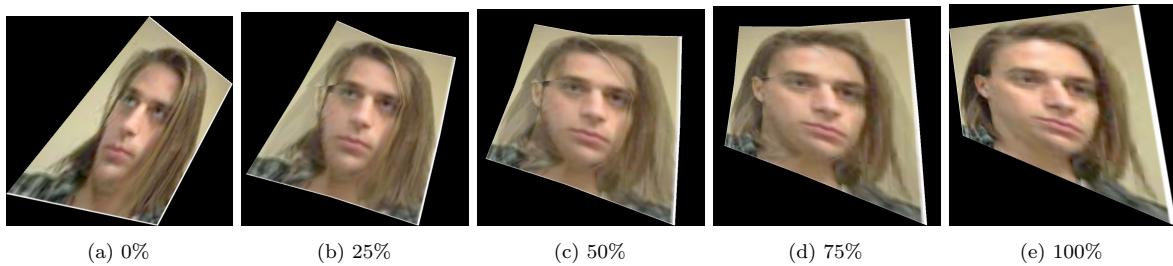


图 14: Morph1

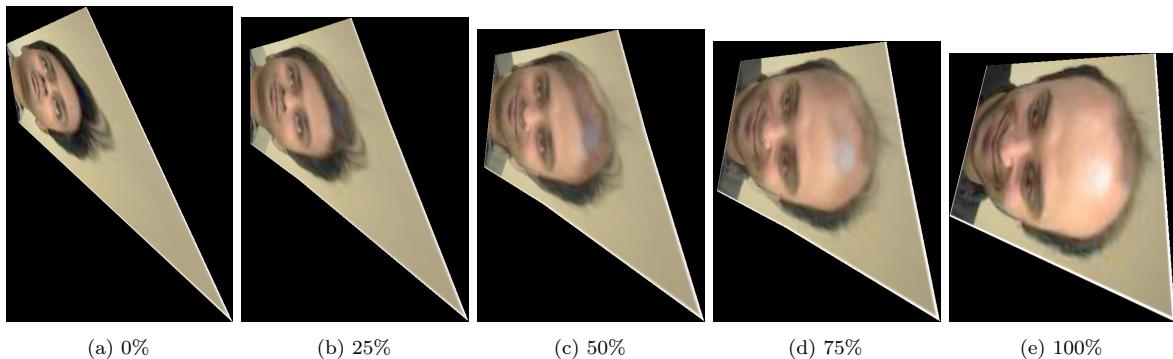


图 15: Morph2

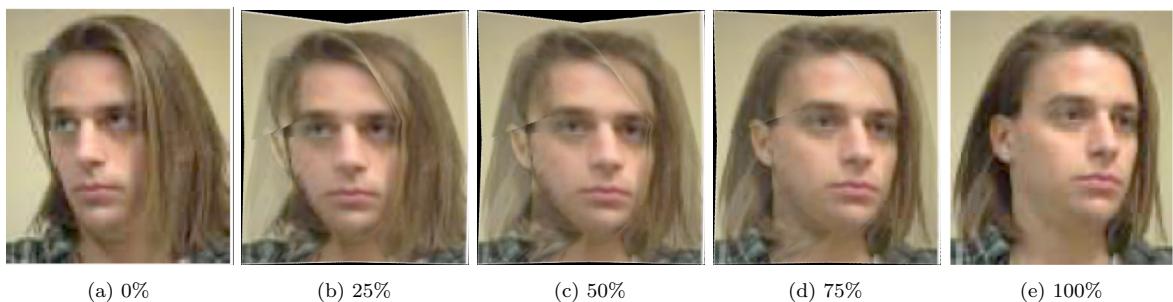


图 16: Postwarp1

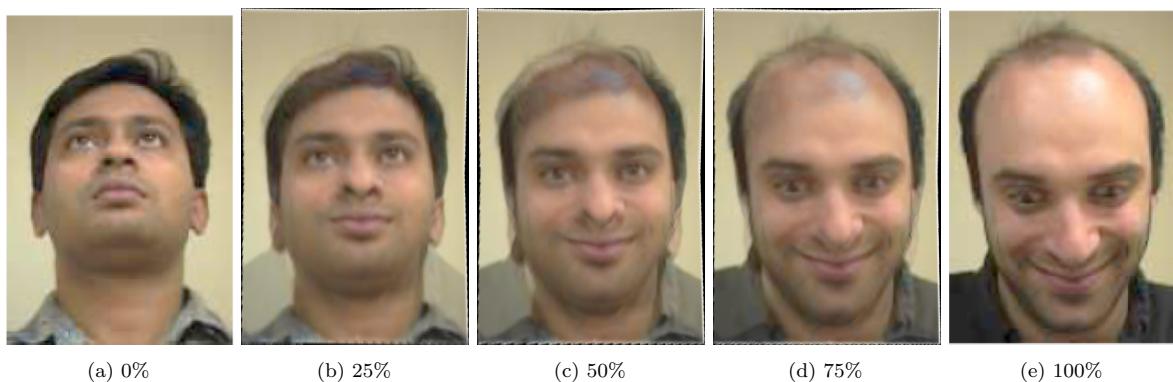


图 17: Postwarp2

我也生成了第一组图片Morph和Post-warp后的变换视频，在video目录下。

注意到第一组图片出现了“裂痕”，我探究后发现是由于源图的三角剖分，对应于目标图后出现了重叠的现象（图18）。这与关键点的选取有关，不过无伤大雅。

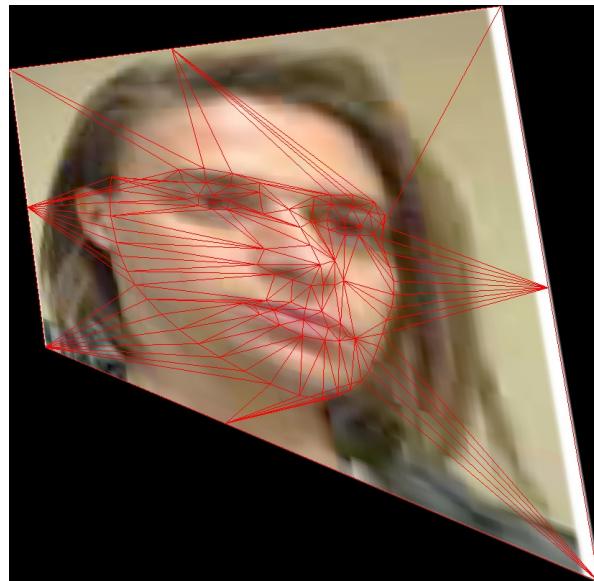


图 18: 重叠的三角形