CIS581: Computer Vision and Computational Photography
Project 1B: Image Gradient Blending
Due: October. 2, 2018 at 3:00 pm

Instructions

- This is an **individual** project. 'Individual' means each student must hand in their **own** answers, and each student must write their **own** code in the homework. It is admissible for students to collaborate in solving problems. To help you actually learn the material, what you write down must be your own work, not copied from any other individual. You **must** also list the names of students (maximum two) you collaborated with.
- You **must** submit your code online on Canvas. We recommend that you can include a README.txt file to help us execute your code correctly. Please place your **code**, **resulting images and videos** into the top level of a single folder (no subfolders please!) named `<Pennkey>_Project1B.zip`
- Your submission folder should include the following:
  - your .m or .py scripts for the required functions.
  - .m or .py scripts for generating the face morphing video.
  - any additional .m files with helper functions you code.
  - the images you used.
  - .avi files generated for each of the morph methods in face morphing.
- This handout provides instructions for two versions of the code: MATLAB and Python. You are free to select **either one of them** for this project.
- Feel free to create your own functions as and when needed to modularize the code. For MATLAB, ensure that each function is in a separate file and that all files are in the same directory. For python, add all functions in a helper.py file and import the file in all the required scripts.
- **Start early!** If you get stuck, please post your questions on Piazza or come to office hours!

1 Gradient Domain Blending

For this part of the project, you will be blending images in the gradient domain as described in the paper Poisson Image Editing by Patrick Perez, Michel Gangnet and Andrew Blake. It is a gradient-domain processing technique with numerous applications such as blending, non-photorealistic rendering, contrast enhancement, texture flattening and tone-mapping.for automatically and seamlessly blending two images together. The paper discusses this technique in Section 3 named Seamless Cloning, which you are strongly advised to thoroughly read and understand before starting this project.

The goal is to seamlessly blend an object from a source image into a target image. The simplest method would be to just copy and paste the pixels from one image directly into the other. However, this will create apparent seams, even if the backgrounds are alike. We need to get rid of these seams without visually tampering the source image.

Human vision is found to be more sensitive to gradients than absolute image intensities. We formulate this problem as finding values for the output pixels that maximally preserve the gradient of the source region without altering any of the background pixels.

To begin with, we define the image that we are changing as the **target image**, the image region that we cut and want to clone as the **source region**, and the pixels in the target image that will be seamlessly cloned with the source image as the **replacement pixels**.

- Image Interpolation using a Guidance Vector Field

$$\min_f \iint_\Omega |\nabla f - \mathbf{v}|^2 \text{ with } f|_{\partial\Omega} = f^*|_{\partial\Omega} \tag{1}$$

where $\nabla = [\frac{\partial}{\partial x}, \frac{\partial}{\partial y}]$ is the gradient operator, f is the function of the blending image, f^* is the function of the target image, $\mathbf{v}$ is the vector field or the gradient field of the source image, Ω is the region of blending and $\partial\Omega$ is the boundary of the blending region.
We solve this interpolation problem (Poisson equations) for each color channel independently.

- Discrete Poisson Solver
The variational problem in equation (6) is discretized to obtain a quadratic optimization problem.

$$\min_{f|_\Omega} \sum_{\langle p,q \bigcap \Omega \neq \emptyset \rangle} (f_p - f_q - v_{pq})^2, \text{ with } f_p = f_p^* \text{ for all } p \in \partial\Omega \tag{2}$$

where N_p is the set of 4-connected neighbors for pixel p, $\langle p,q \rangle$ denote a pixel pair such that $q \in N_p$, f_p is the value of f at p and $v_{pq} = g_p - g_q$ for all $\langle p,q \rangle$.

The solution satisfies the following simultaneous linear equations:

$$\text{for all, } p \in \Omega, |N_p| f_p - \sum_{q \in N_p \bigcap \Omega} f_q = \sum_{q \in N_p \bigcap \partial\Omega} f_q^* + \sum_{q \in N_p} v_{pq} \tag{3}$$

$$|N_p| f_p - \sum_{q \in N_p} f_q = \sum_{q \in N_p} v_{pq} \text{ for pixels interior to } \Omega, \text{ i.e. } N_p \subset \Omega \tag{4}$$

We need to solve for f_p from the given set of simultaneous linear equations. If we form all the f_p as a vector x, then the given set of equations can be converted into a linear system of $Ax = b$. Please do note that not all of f_q is unknown. It is possible that $q \in N_p$ and also $q \in \partial\Omega$, in which case, $f_q = f_q^*$ and it becomes a known parameter.

1.1 Align the Source Image and Create its Mask

First, you need to align the source image and target image. Please use any image editor to adjust the size and position of the source image, ensuring that the region of the target image you want to replace is well-aligned with the source image. Then, save the resized source image and the coordinate of its top left corner as an offset. From now on, source image will refer to the resized source image.

Complete the following function to create an image mask - a logical matrix representing the pixels you want to replace in the source image. A value of 1 means that we will be using the pixel whereas a value of 0 means that the pixel will not be used.
We recommend using MATLAB's function `imfreehand` and `createMask`.

```
[mask]=maskImage(img)
```

- (INPUT) `img`: $h \times w \times 3$ matrix representing the source image.
- (OUTPUT) `mask`: $h \times w$ matrix representing the logical mask

1.2 Index the Pixels

The intensity of the replacement pixels in the target pixel can be found using the linear system $Ax = b$. But, not all the pixels need to be computed. Only the pixels masked as 1 in the logical mask will be used to blend. In order to reduce the number of calculations, you need to index the replacement pixels such that each element in x represents one replacement pixel. As shown in Figure 3, the yellow locations are the replacement pixels (indexed from left to right).

Complete the following function to obtain the indexes of the replacement pixels:

(a) Source Image

(b) Target Image

(c) Blended Image

`[indexes] = getIndexes(mask, targetH, targetW, offsetX, offsetY)`

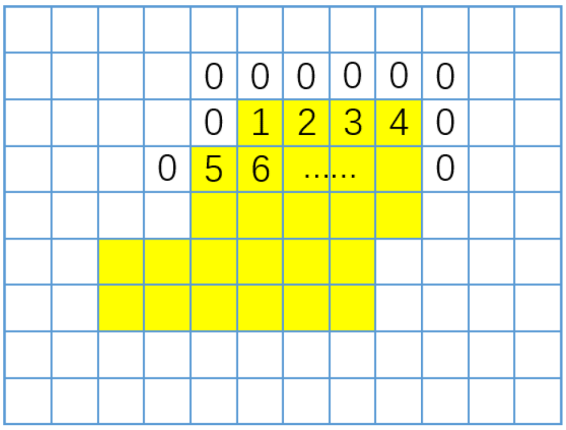
- (INPUT) `mask`: The logical matrix $h \times w$ representing the replacement region.
- (INPUT) `targetH`: The height of the target image, h'
- (INPUT) `targetW`: The width of the target image, w'
- (INPUT) `offsetX`: The x-axis offset of the source image with respect to the target image.
- (INPUT) `offsetY`: The y-axis offset of the source image with respect to the target image.
- (OUTPUT) `indexes`: $h' \times w'$ matrix representing the indices of each replacement pixel. The value 0 means that is not a replacement pixel.

1.3 Compute the Coefficient Matrix

As described in the section 3.2, the intensities of the replacement pixels are obtained by solving $Ax = b$. In this section, you need to generate the the Coefficient Matrix A. Please note that the Coefficient Matrix is of size $N \times N$, where N is the number of replacement pixels. In order to reduce the memory of this matrix, you will have to use a sparse matrix.
Complete the following function to compute the Coefficient Matrix:

`[coeffA] = getCoefficientMatrix(indexes)`

- (INPUT) `indexes`: $h' \times w'$ matrix representing the indices of each replacement pixel.
- (OUTPUT) `coeffA`: an $N \times N$ sparse matrix representing the Coefficient Matrix, where N is the number of replacement pixels.



3.2 Indexing the Pixels

1.4 Compute the Solution Vector

Complete the following function to generate the solution vector b in the linear system $Ax = b$.

```
[solVectorb] = getSolutionVect(indexes, source, target, offsetX, offsetY)
```

- (INPUT) `indexes`: $h' \times w'$ matrix representing the indices of each replacement pixel.
- (INPUT) `source`: $h \times w$ matrix representing one color channel of the source image.
- (INPUT) `target`: $h' \times w'$ matrix representing one color channel of target image.
- (INPUT) `offsetX`: The x-axis offset of the source image with respect to the target image.
- (INPUT) `offsetY`: The y-axis offset of the source image with respect to the target image.
- (OUTPUT) `solVectorb`: $1 \times N$ vector representing the solution vector.

1.5 Seamlessly Clone the Image

Once you have obtained A and b as stated above, solve for vector x. You will need to replace the pixels in question with the updated intensity i.e. clone the image and obtain the resulting image. Complete the following function to obtained the composite image:

```
[resultImg] = reconstructImg(indexes, red, green, blue, targetImg)
```

- (INPUT) `indexes`: $h' \times w'$ matrix representing the indices of each replacement pixel.
- (INPUT) `red`: $1 \times N$ vector representing the intensity of the red channel replacement pixel.
- (INPUT) `green`: $1 \times N$ vector representing the intensity of the green channel replacement pixel.
- (INPUT) `blue`: $1 \times N$ vector representing the intensity of the blue channel replacement pixel.
- (INPUT) `targetImg`: $h' \times w' \times 3$ matrix representing the target image.
- (OUTPUT) `resultImg`: $h' \times w' \times 3$ matrix representing the resulting cloned image.

1.6 Wrapper Function

After you complete all the above functions, you will need to write a wrapper function and a demo script. In this function named `seamlessCloningPoisson.m`, call `getIndexes.m`, `getCoefficientMatrix.m`, `getSolutionVect.m` and `reconstructImg.m` and solve the linear system. We recommend the MATLAB function `mldivide`.

```
[resultImg] = seamlessCloningPoisson(sourceImg, targetImg, mask, offsetX, offsetY)
```

- (INPUT) `sourceImg`: $h \times w \times 3$ matrix representing the source image.
- (INPUT) `targetImg`: $h' \times w' \times 3$ matrix representing the target image.
- (INPUT) `mask`: The logical matrix $h \times w$ representing the replacement region.
- (INPUT) `offsetX`: The x-axis offset of the source image with respect to the target image.
- (INPUT) `offsetY`: The y-axis offset of the source image with respect to the target image.
- (OUTPUT) `resultImg`: $h' \times w' \times 3$ matrix representing the resulting cloned image.

Finally, write a script to generate your blended image using `seamlessCloningPoisson.m` and `maskImage.m`

Please use your creativity while creating cloned images.

2 Test and Submission

- Use different kinds of images. Face to face morphing is mandatory but get creative! Morph objects to objects, even face to objects. Creative morphs will receive the honor of public recognition on Piazza.
- We have provided a test script for MATLAB and Python for this project. Extract the contents of the test script to the same directory as your functions and run `Test_script.m/py` in MATLAB/Python. When grading, we'll be calling your functions in the same manner, so make sure they work as you'd expect on the sample in the test script.
- Collect all your source code files and test images into a folder named as `<Pennkey>_Project1B`. Zip this folder and submit it to Canvas. Any break in this rule will lead to a failure in the test script. Only submit codes pertaining to your language of implementation. For example: If you choose to do the project in Python, do not submit the MATLAB folder containing the MATLAB starter codes.