Yuchen Fu: poisson image editing - seamless cloning

April 20, 2020

[34]:
```
# Import all useful packages into our working enviroment
import cv2
import numpy as np
import matplotlib.pyplot as plt
import numpy as np
from scipy.sparse import linalg as linalg
from scipy.sparse import lil_matrix as lil_matrix
from scipy.sparse.linalg import spsolve, minres
from IPython.display import Image
```

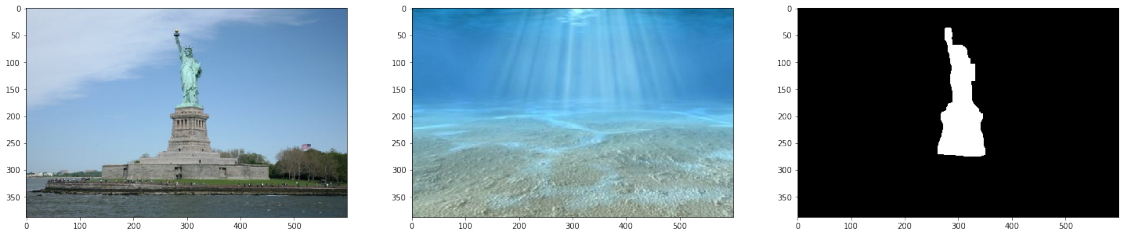
[35]:
```
# Import the desired source, target, and mask into our working enviroment
# Set mask as a grayscale image who only has one channel
raw_source = cv2.imread('resources/2source.jpg',cv2.IMREAD_COLOR)
raw_target = cv2.imread('resources/2target.jpg',cv2.IMREAD_COLOR)
raw_mask = cv2.imread('resources/2mask.jpg', cv2.IMREAD_GRAYSCALE)

source_shape = raw_source.shape[:-1]
target_shape = raw_target.shape[:-1]
mask_shape = raw_mask.shape

# Limit the range of our concern to pictures with the same shape
assert(source_shape == target_shape == mask_shape)

# Show the three pictures that we will be performing seamless cloning on
x, y = plt.subplots(1,3,figsize=(25,25))
y[0].imshow(raw_source[:,:,::-1])
y[1].imshow(raw_target[:,:,::-1])
y[2].imshow(raw_mask, cmap='gray')
```

[35]:
```
<matplotlib.image.AxesImage at 0xa264db748>
```



[36]:
```
# Temporarily set source and target to be the same with the raw ones
# For the above two source and target images we found no need to process␣
 ↪anymore. All three images have equal size.
source=raw_source
target=raw_target
```

[37]:
```
# Preprocess the mask image
# The goal is to convert the mask's range from {0,255} to {0, 1}
mask = np.atleast_3d(raw_mask).astype(np.float) / 255
mask[mask != 1] = 0
```

Now we switch our attention to the center of our problem. Our final objective is to blend the source within the target seamlessly, so the transition must be smooth, meaning the boundary of the source and the target should be the same color (else human eye will detect an edge, or a color difference, immediately). In the language of math, we say: |_=*|_

For the pixels within the source, we need not to maitain its original color, but only its original gradient. As long as the gradient remains the same, or to be precise, as long as we minimize the difference of the former and latter gradients, the new source blended in the target will appear similarly in human eyes compared with the original source.

To formulate all this in math so we can realize it in code, we refer to the following blog written by hjimce. Link: http://blog.csdn.net/hjimce/article/details/45716603

[38]:
```
Image(filename='resources/demo.png')
```

[38]:

现在假设一幅图像为3*3的单通道灰度图像：

1	2	3
4	5	6
7	8	9

我们假设每一点的像素值为V，V(1)表示像素点1的值，那么我们可以定义像素点5的散度的计算公式为：

$$div(5)=[V(2)+V(4)+V(6)+V(8)]-4*V(5)$$

说白了就是通过拉普拉斯卷积核，进行卷积，就可以求解散度了。

0	1	0
1	-4	1
0	1	0

拉普拉斯卷积核

Without mentioning of detailed mathematic explanation, we use the above concept of Laplacian convolution directly. To summarize, each pixel's divv = -4 * that pixel's value + sum(each

of its neigbors' value). Apply this to all pixels within the image, we will find that Poisson Image Blending is in essence a least-squares problem where Ax = B for every pixel under the mask. In other words, for all pixels = 1 in the mask, we want to set up a linear equation such that the gradient for a given pixel is the same in both the source and final images.

Let's now construct our A matrix. Notice we only have to do so once, since A is based on our mask image, not on source or target. For source and target, we have to process it for each of their 3 or more channels. In Ax = b, x is all the pixels, b is the corresponding div for that pixel or a boundary condition, and A will be the coefficient for the values of the pixels. For each pixel, set its coefficient to -4 and all its neigbors' coefficients' to be 1 as we discussed. This means that the diagonal of A will be -4.

[39]:
```
def mask_pixel(mask):
    nonzero = np.nonzero(mask)
    save = zip(nonzero[0], nonzero[1])
    pixel_points = [_ for _ in save]
    return pixel_points
```

[40]:
```
def neighbors(index):
    i,j = index
    return [(i+1,j),(i-1,j),(i,j+1),(i,j-1)]
```

[41]:
```
pixel_points = mask_pixel(mask)
dim = len(pixel_points)
A = lil_matrix((dim,dim))

for i, pixel in enumerate(pixel_points):
    A[i,i] = -4

    for x in neighbors(pixel):
        if x in pixel_points:
            j = pixel_points.index(x)
            A[i,j] = 1
```

Now we have the A matrix, what is left is to create b. As we said, b is the divergence for each pixel. Notice, when that pixel is at the boundary, equate it with the color of the pixel at the target's boundary. We write the process of construction in a function because we have to construct a different b for each channel of our target/source.

To start with, we write a function that can calculate the divergence of a given pixel, which is just the Laplacian formula we discussed earlier.

[42]:
```
def source_div(source, pixel):
    i,j = pixel
    div = -4 * source[i, j] + source[i+1, j] + source[i-1, j] + source[i, j+1]␣
 ↪+ source[i, j-1]
    return div
```

Now we are ready to construct our b vector. First set its dimension, which will be a vector with height = dimension to match our x. Then, enumerate all pixel points and assign the calculated div for that row. Inside the for loop, also check if that pixel is at the boundary. If that is the case, we would need to incorporate the boundary condition in that row of b, which is simply to substract the neighboring target pixel value. That being said, we will first need to write a function that

detects if a pixel is at the boundary or not.

[43]:
```
def boundary(pixel, mask):
    # If that pixel is not even inside the mask, it certainly cannot be at the␣
 ↪boundary and will not be our concern.
    if mask[pixel] == 1:
        for neighbor in neighbors(pixel):
            # For any point in our concern but its neighbors are not, that␣
 ↪point has to be at the boundary
            if mask[neighbor] != 1:
                return True

    # Return False for any other circumstances
    return False
```

[44]:
```
def construct_B(source, target, mask):
    # Determine the size
    b = np.zeros(dim)

    for i,pixel in enumerate(pixel_points):
        # Assign the div to each pixel
        b[i] = source_div(source, pixel)

        # Check if this pixel is at boundary
        if boundary(pixel, mask):
            # If it is, substract values of all the pixels around it
            for neighbor in neighbors(pixel):
                if mask[neighbor] != 1:
                    b[i] -= target[neighbor]

    return b
```

We now have both A and b at hand, simply solve x in the linear system Ax = b using scipy.sparse.linalg.cg, which is designed to solve such linear systems with A being a sparse matrix. After getting x, we know what intensity to give for each pixel inside the mask at the target. So we first construct a new image which is the same as the target, then we change the pixels inside the mask at this new copy, so we do not mess with the original target image.

[45]:
```
def solve(source, target, mask):
    b = construct_B(source, target, mask)
    x = linalg.cg(A, b)
    save = x[0]

    copy = np.copy(target).astype(int)
    for i, pixel in enumerate(pixel_points):
        copy[pixel] = save[i]
    return copy
```

We have finished all the functions required for the blending. Let's now call those functions, giving them our concerned target, source, and mask, for each channels. After solving for each

channels, simply merge each channels' result together using cv2.

[46]:
```
# Save the number of channels, as we have to process the images for each of␣
 ↪their channels and then stack together
channels = source.shape[-1]
```

[47]:
```
solved = [solve(source[:,:,i], target[:,:,i], mask) for i in range(channels)]
```

[48]:
```
final = cv2.merge(solved)
```

[49]:
```
# View our final result
plt.imshow(final[:,:,::-1])
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```