Problem Description

For Lab 3, all the problems were concerning image warping, which was performing some modifications to an image based on user input in this case. Problems included the following:

* Inserting an image into a specific space after performing an affine transformation using homographies. The image then was inserted into the destination image using forward and reverse warping.
* Modifying the given sample code given in class to perform multiple single point warps, using then the new image as the source.
* Implementing a multi-point warping given multiple input points. Very similar to the past problem.
* Implementing a program to morph a face into another. Similar to the example seen in class where a woman’s face and a tiger I believe are morphed together and can be seen overlapping each other.

Algorithms Implemented

For this lab, there were no algorithms implemented other than the ones Dr. Fuentes has provided in his sample code in the class web page.

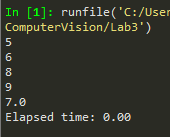
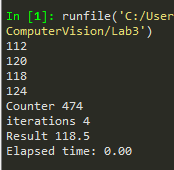
Experimental Results

1) For the first problem were a method needed to perform real valued index on an image, the instructions were a bit ambiguous, so I hope I implemented the expected problem. I used quiz 3 and the pixel averaging power point presentation in the class webpage. What I did was implement a method that would take two X and Y set of points ((0,0), (1,1)) and given those coordinates, I would iterate through the array, keep a counter variable that would add all of the values, and return the average of those given pixel values. As a demonstration I will use the following simple array.

[1,2,3], If I were to give my method the input of (1,1) and (2,2). The values between those points are:

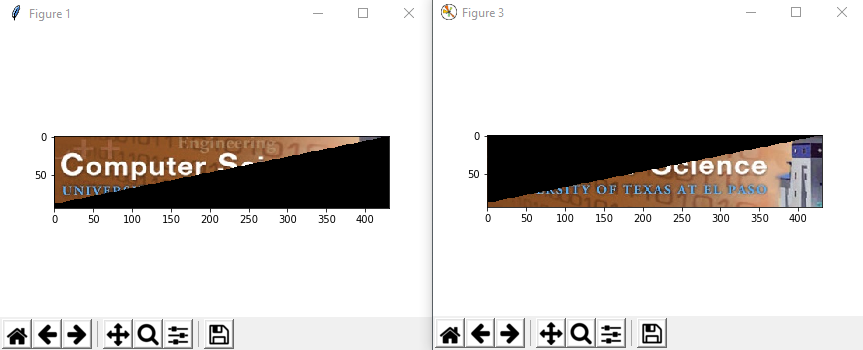
[4,5,6], 5,6,8, and 9. So 5+6+8+9 = 28/4 = 7. To make sure, I print the values the method does traverse

[7,8,9]]) as well as the result the method returns.

Simple Array result Given an actual picture calculating the avg of a region with   
 coords (1,1),(2,2)

2) For the second problem, I was able to successfully break the banner into two different triangles as shown below.



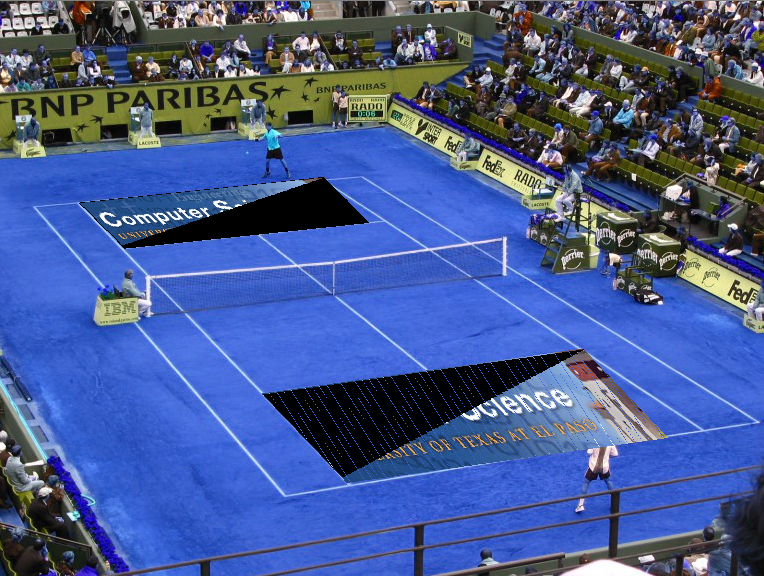
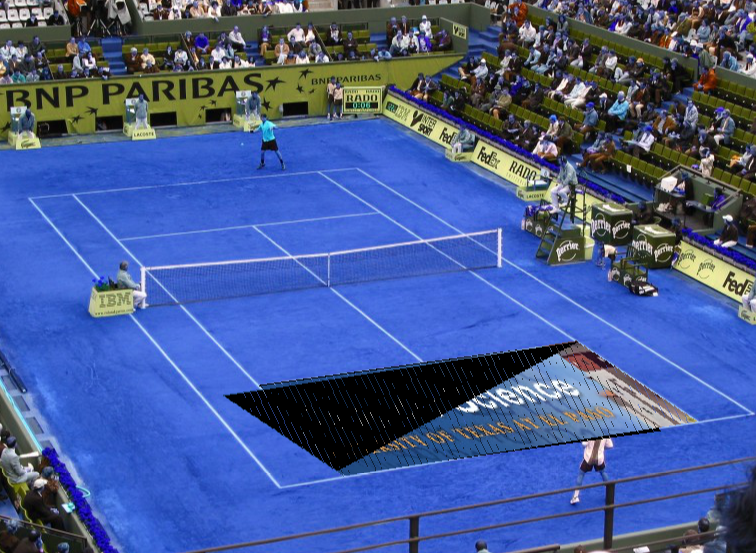
The colors are different since PIL and OpenCV have a different order on RGB/BGR, but since the functionality is the same, I did not change the colorspaces. I broke the triangles by first calculating the row and col size of the original image, and basically what I did was the following. I would take the source img and iterate only to the row and col that I got before, but each time I finished a loop, I would substract the “slope” to the columns, which I got by simply diving the img.shape[1]/img.shape[0]. Everytime I would be setting my second image to the value of the source, and I would delete that pixel from the source by setting it to zero. So by the time the loop ended, I would end with the source img which had the first half set to 0, and the second img would be a copy of the first half before changing those pixels to 0. The code to achieve this is the following.

for y in range (1,row):  
 for x in range (1,col):  
 if(x == col):  
 triangle2[y,x] = img[y,x]  
 continue

else:  
 triangle2[y,x] = img[y,x]  
 img[y,x] = 0

col -= slope

Afterwards, I had to modify Dr. Fuentes code a little bit in order for the ginput to accept 6 different points in the tennis image. To achieve this, I basically duplicated the code twice, and for the second triangle, I applied an np.flip function in order to correct the order of the pixels, since the code only accepted “top left, top right, and bottom left.” If I had left the array without modifying it first, the second triangle would not have aligned correctly as shown below.

The problem I encountered can be clearly seen from the pictures above. I was able to correctly break the triangles and align them properly, but once they overlapped, the region I had set to pixel intensity 0 covers the first triangle. I could not make it to properly eliminate the black triangle but the triangles are properly aligned.

Approaches I tried to eliminate the black triangles. I tried to modify the code given to us by making the second triangle image check its region. If the pixels where it would be drawn had intensity 0, meaning it was black, to replace those pixels with the new ones, therefore replacing the black ones with the “good triangle”. And if the pixels were not 0, meaning there was the first good triangle there, to ignore those pixels. This in my head makes sense, although it may not be the best, most efficient way, I was not able to properly implement it. The way the sample code was made it somewhat difficult for me to modify it, as there were a lot of times I knew what the code was generally doing, but I did not have a solid enough understanding to modify it. For example the following line:

dest\_im = source\_im[source\_coords[0],source\_coords[1],:].reshape(dest\_rows,dest\_cols,3)

I know this is where the dest image is being set equal to the source img, but the way the pixels are benign copied all at once, made it hard for me to do something like

If(dest\_im[y,x] == 0)  
 dest\_im[y,x] == source\_im[y,x]  
else  
 ignore

3) Again, I was unable to actually modify the sample code in order to implement backward mapping. I know the theory that instead of checking the source img, transforming those cords into the new\_coords using the transformation method, and then setting the destination image equal to the source img as

dest\_im = source\_im[source\_coords[0],source\_coords[1],:].reshape(dest\_rows,dest\_cols,3)

It would simply be backwards. First check the dest image desired coordinates, and map those coordinates to the source img. Similar to how in class we saw that we multiply the homography x the desired coordinates, in order to get the source img coordinates and then start correctly mapping the the new\_coords to coords.

4) For the fourth problem, it was very straightforward. The sample code was basically doing everything for us, except that it was only taking 2 points as input, and then it would finish. What I did was only add a while True loop that would infinitely run the program, meaning it would take two points, perform the warp, and then that image would then become the new source. The images shown below display many point warps I did, specially around the ubuntu logo, and the result below that image.

