Report of Lab8

Yuxiang Chen 5110309783

December 17, 2012

Contents

1 The Purpose of This Experiment and My Preparation

I have to say that when I first saw the content of the experiment, I was really scared. Yes, I use the word 'scared', since I thought we had to get all the corners all by ourselves, and then get the most suitable picture by calculating its 128-dimension vectors in lots of scales. But when I read carefully, I find that the first part is telling us a method to get the corner points in a more precise way. And to save time, we can just use the program written by TA using the function, cvGoodFeaturesToTrack, which is based on Harris' method. Then we have to get the main direction of the whole picture, and thus we will get the 128-dimension vector of the picture. Finally, by calculating the similarity of the sift calculator by multiply them, we're supposed to get the most similar picture comparing to the target.

2 The Main Part of the Experiment

In the program, to make it easier to understand, I split the whole program into some functions and then I will explain them in the following part. In the program file, I also make interpretations in the necessary places. And I separate them into a main function and 5 small functions:

2.1 get dir(image,array)

The first function is in the form of get dir(image,array). It can get the main direction of each key point we get in the pictures. And of course it's easy to understand, we first get an 16*16 square, and then get the weight of 36 bins by calculating the gradients. So in this way we're able to find the bin with the biggest weight, which will be treated as the main direction. Here are the codes:

```
def get_dir(image,array):#                    array                    list
    bin=list()
    final_dir=list()#
    len_array=len(array)
    for i in range(0,36):
        bin.append(0)
    for t in range(0,len_array):
        x=int(array[t][0])
        y=int(array[t][1])
        for i in range(0,36):
            bin[i]=0
        for i in range(x-8,x+8):
            for j in range(y-8,y+8):
                if x<=0 or x+8>=image.height-1 or y<=0 or y+8>=image.width-1:#
                    continue
                tempx=image[i+1,j]-image[i-1,j]
                tempy=image[i,j+1]-image[i,j-1]
                sqx=tempx*tempx
                sqy=tempy*tempy
                m=math.sqrt(sqx+sqy)
                theta=get_theta(tempx,tempy)
                bin[int(theta/10)]+=m
        direction=bin.index(max(bin))#                bin
        final_dir.append(direction)
    return final_dir
```

2.2 near(image,array,final dir)

The main directions we get in the above function will be used in this function where we will calculate the 128-dimension calculator. As written in the demo pdf file, we shall first get a 16*16 square whose center is the key point. Then by separating it again into 16 parts, with each part being 4*4, we'll be able to get 16 eight-dimension vectors. And we just get them together to reach the final vector we want. And I use the nearest point method to find the related point of the object-reference-system in the graph-reference-system, since it's easy and also satisfying. We will use the same method as the last function to get the direction distribution in this function.
And below are the codes:

```
def near(image,array,final_dir):#                    1 2 8
    len_array=len(array)
    total_vector=list()#                  1 2 8        l i s t
    for t in range(0,len_array):
        bin=list()#     8
        for i in range(0,8):
            bin.append(0)
        x=int(array[t][0])
        y=int(array[t][1])
        vector=list()
        dir_theta=final_dir[t]*10+5
        pi_theta=float(dir_theta)/360*2*math.pi#   t h e t a
        for i in range(-8,8,4):# 16  4*4          8
            for j in range(-8,8,4):
                for k in range(0,4):
                    for l in range(0,4):
                        realx=x+(i+k)*math.cos(pi_theta)-(j+l)*math.sin(pi_theta
                        #
                        realy=y+(i+k)*math.sin(pi_theta)+(j+l)*math.cos(pi_theta
                        posx=int(realx+0.5)#
                        posy=int(realy+0.5)
                        if posx<=0 or posy<=0 or posx>=image.width-2 or posy>=im
                            continue#
                        tempy=image[posy+1,posx]-image[posy-1,posx]
                        tempx=image[posy,posx+1]-image[posy,posx-1]
                        m=math.sqrt(tempx*tempx+tempy*tempy)
                        theta=get_theta(tempx,tempy)
                        bin[int(float((theta-dir_theta+360))%360/45)]+=m
                        #              8   b i n                    b i n
                        #
                for m in range(0,8):
                    vector.append(bin[m])#     8
                    bin[m]=0#      8
        total_vector.append(vector)#             1 2 8        t o t a l _ v e c t o r
    return total_vector
```

2.3 get vector(total vector)

This part is a quite easy part to understand. After getting the 128-dimension vector in the function 'double linear', we just use this function to make the magnitude of the vector to be 1. And after this part, we can just judge whether two pictures are similar enough by seeing how close the product of their sift calculators are to 1.

```
def get_vector(total_vector):#  1 2 8
    for i in range(0,len(total_vector)):
```

```
            temp=total_vector[i]
            square_sum=0
            for j in range(0,128):
                square_sum+=temp[j]*temp[j]
            length=math.sqrt(square_sum)
            if length==0:
                continue
            for j in range(0,128):
                total_vector[i][j]/=float(length)
        return total_vector
```

2.4 two small assistant function

The first function is 'times(vec1,vec2)', which is used to get the product of two vectors. Obviously, we will use the function to get the product of two sift calculator. And the second function is 'get theta(tempx,tempy)'. We use it to get the gradient-direction, and it's calculated just as the pdf tells us.
Here are the functions:

```
def times(vec1,vec2):#
    result=0
    for i in range(0,128):
        result+=vec1[i]*vec2[i]
    return result

def get_theta(tempx,tempy):#           theta
    if tempx!=0:
         theta=math.atan(float(tempy)/tempx)/(2*math.pi)*360
    else:
        if tempy>0:
            theta=90
        if tempy<0:
            theta=270
        else:
            return 0
    if tempx>0 and tempy<0:
        theta+=360
    if tempx<0 and tempy<0:
        theta+=180
    if tempx<0 and tempy>0:
        theta+=180
    return theta
```

2.5 the main part

In the main part, we need to open two pictures we are going to compare first, then we just use the functions above to get the 128-dimension vectors of all key points in these pictures. Then I just multiply them ,finding the biggest ones and print them out. In my program, I choose the ones with a product bigger than 0.85, and the graph with the largest pairs of these sift calculators bigger than 0.85 is treated as the most suitable one. Following are my program:

```
image1 = cvLoadImage ("3.jpg",0)
image2 = cvLoadImage ("target.jpg",0)
infile1=open("3.txt","r")
infile2=open("target.txt","r")
array1=list()# a r r a y 1
array2=list()
for line in infile1.readlines():
    data=line.strip('\n')
    xandy=data.split()
    x=int(xandy[0])
    y=int(xandy[1])
    temp=[x,y]
    array1.append(temp)
for line in infile2.readlines():
    data=line.strip('\n')
    xandy=data.split()
    x=int(xandy[0])
    y=int(xandy[1])
    temp=[x,y]
    array2.append(temp)
final_dir1=get_dir(image1,array1)
total_vector1=double_linear(image1,array1,final_dir1)#                    1 2 8
compare_vector1=get_vector(total_vector1)#                  1 2 8   s i f t
final_dir2=get_dir(image2,array2)
total_vector2=double_linear(image2,array2,final_dir2)
compare_vector2=get_vector(total_vector2)
length1=len(compare_vector1)#              s i f t
length2=len(compare_vector2)
posx=-1
posy=-1
outfile1=open("3out1.txt","w")
outfile2=open("3out2.txt","w")
for i in range(0,length1):
    maximum=0#   m a x i m u m                              s i f t
    for j in range(0,length2):
        temp=times(compare_vector1[i],compare_vector2[j])
        if temp>maximum:
            posx=i
```

```
39                posy=j
40                maximum=temp
41        if maximum>0.85:#       sift                0.8
42            outfile1.write(str(array1[posx][0])+'\t'+str(array1[posx][1])+'\n')
43            #                   x   y
44            outfile2.write(str(array2[posy][0])+'\t'+str(array2[posy][1])+'\n')
45    infile1.close()
46    infile2.close()
47    outfile1.close()
48    outfile2.close()
```

And I have already attached all the files got in the whole program in this report, including the matched points. After we execute the program, it's easy to find that 3.jpg has the most pairs of matched points with target.jpg. So we can believe that 3.jpg is the one which is most similar to target.jpg in the all five pictures. And I also make a graph with the program offered by TA. But sadly, as this method isn't very precise, we can't get the ideal effect.

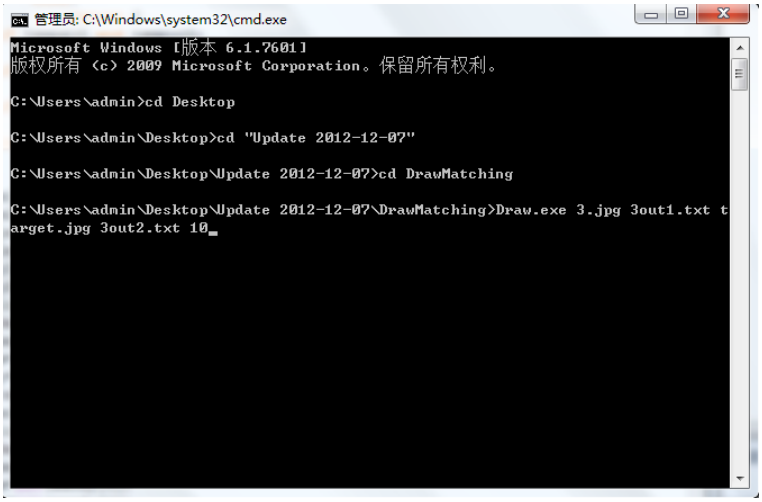

Figure 1: The executing process

3 The Problems I Met and My Thoughts

In this experiment, I certainly met a lot of problems. At first ,I don't understand how to change the coordinates from the object-reference-system to the graph-reference-system, then I have some problems with resizing the pictures. But at last, TA helped us to make the problem easy by eliminating the process of making a graph pyramid. And I also solve my problems by e-mailing TA. So I wanna say 'thank you very much' here to him. And now that I have solved the

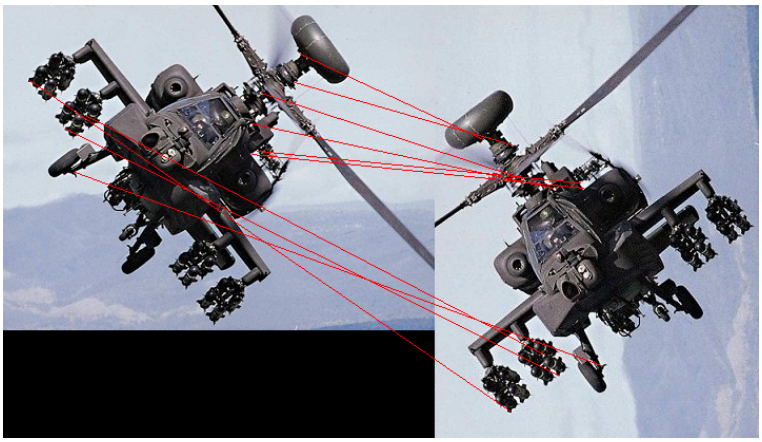

Figure 2: the graph of matched points

problems, I make some interpretations to where I have engaged in.
And of course, I had a mistake at first because I misuse height and width of the image, so I didn't get a good answer then. But later, I correct it and get the result you see.
And after finishing the experiment, I find sift method a very good method to comparing the similarity of two graphs, since it makes us overlook the direction relation with the graph and the object. So it means that however it rotates, we can still know it is the same one with the original one. And I also think the first Gauss method to get the corners maybe more precise than the Harris method, so maybe I will try that method when I have time. And I also thinks this sift method can eliminate the effect of the intensity of the light, since it's based on the gradient. So maybe we can use this method to write a graph-searching engine in the future.