Lab Instructions - Session 11

Geometric Image Transformations

Translation

We want to translate the image with the vector $t = (t_x, t_y)$. Translation is a special case of an affine transformation $x' = Ax + t$ in which $A = 0$. The 2 by 3 matrix $M = [A\ t]$ can represent an affine transformation. Thus, a translation is represented by the 2 by 3 matrix $M = [0\ t]$. Then we can translate the image by applying the affine transformation M using the **cv2.warpAffine** function. Run the following script to translate an image with the vector (tx, ty).

File: **translation.py**

```
import cv2
import numpy as np

I = cv2.imread('karimi.jpg')

# translations in x and y directions
tx = 100
ty = 40

# use an affine transformation matrix (2x3)
M = np.array([[1, 0, tx],
              [0, 1, ty]]).astype(np.float32)

output_size = (I.shape[1],I.shape[0]) # output image size
#output_size = (I.shape[1]+200, I.shape[0]+200)

J = cv2.warpAffine(I,M,output_size)

cv2.imshow('I',I)
cv2.waitKey(0)

cv2.imshow('J',J)
cv2.waitKey(0)

#! use a homography transformation matrix (3x3)
#H = np.array([[1, 0, tx],
#              [0, 1, ty],
#              [0, 0, 1]]).astype(np.float32)
#K = cv2.warpPerspective(I,H, output_size)
#cv2.imshow('K',K)
#cv2.waitKey(0)

cv2.destroyAllWindows()
```

- Change tx and ty and see the result. Set one or both of them to a negative value.
- We have chosen the output image size (**output_size**) as the original image size. Change the output image size (e.g. to the one commented out in the code: `#output_size = ...`) and see the result.

You can also apply translation using a 3 by 3 homography matrix given to the `cv2.warpPerspective` function. Uncomment the following part in the code and display the resulting image **K**.

```
#! use a homography transformation matrix (3x3)
H = np.array([[1, 0, tx],
              [0, 1, ty],
              [0, 0, 1]]).astype(np.float32)
K = cv2.warpPerspective(I,H, output_size)
cv2.imshow('K',K)
```

- Notice that the 3 by 3 matrix **H** is the matrix **M** plus an extra row [0 0 1]. Compare the images **K** and **J** and observe that they are identical.

Euclidean (Rigid) transformation

The following code rotates the image with an angle **th** around the origin (pixel location (0,0)). For a rotation about the origin, the 2 by 3 affine transformation matrix is $M = [R\ 0]$ where R is the 2 by 2 rotation matrix. We can also add a translation vector in which case $M = [R\ t]$

File: **rigid.py**

```
import cv2
import numpy as np

I = cv2.imread('karimi.jpg',0)

tx = 0
ty = 0

th =  20 # angle of rotation (degrees)
th *= np.pi / 180 # convert to radians

M = np.array([[np.cos(th),-np.sin(th),tx],
              [np.sin(th), np.cos(th),ty]])

J = cv2.warpAffine(I,M, (I.shape[1], I.shape[0]) )

cv2.imshow('I',I)
cv2.waitKey(0)

cv2.imshow('J',J)
cv2.waitKey(0)
```

- Why have we converted the rotation angle to radians?
- Change the translation vector elements tx and ty and see the result.

Task 1:

In the above example, we saw how to rotate around the origin (pixel 0,0). We can also rotate around an arbitrary point $c = (c_x, c_y)$ by adding a proper translation vector. This can be done by translating any point with the translation $-c$ (so that c moves to the origin), rotating around the origin, and translating back with the translation vector $+c$. The transformation then becomes $x' = R(x-c) + c = Rx + (c - Rc)$.

The following python code keeps rotating the image **I** around the origin (0,0). Run the code and see the result. You need to change the code so the image is rotated about its center **c**. (**c** has been computed in the code). **You are not allowed to use the `cv2.getRotationMatrix2D` function.**

File: **task1.py**

```
import cv2
import numpy as np

I = cv2.imread('karimi.jpg',0)

# center of the image
c = np.array([[I.shape[1]/2.0], [I.shape[0]/2.0]])

for theta in range(0,360):
    th = theta * np.pi / 180 # convert to radians

    R = np.array([[np.cos(th),-np.sin(th)],
                  [np.sin(th), np.cos(th)]])

    t = np.zeros((2,1)) # you need to change this!

    # concatenate R and t to create the 2x3 transformation matrix
    M = np.hstack([R,t])

    J = cv2.warpAffine(I,M, (I.shape[1], I.shape[0]) )

    cv2.imshow('J',J)

    if cv2.waitKey(10) & 0xFF == ord('q'):
        break
```

Similarity transform

A similarity transformation consists of translation, rotation and global scaling. The transformation matrix is $M = [sR \ t]$. The following code adds a scale factor **s** to the previous examples to apply a similarity mapping.

File: **similarity.py**

```
import cv2
import numpy as np

I = cv2.imread('karimi.jpg')

tx = 100
ty = 60

th =  20 # angle of rotation (degrees)
th *= np.pi / 180 # convert to radians

s = 0.6 # scale factor

M = np.array([[s*np.cos(th),-s*np.sin(th),tx],
              [s*np.sin(th), s*np.cos(th),ty]])

output_size = (I.shape[1], I.shape[0])
J = cv2.warpAffine(I,M,  output_size)

cv2.imshow('I',I)
cv2.waitKey(0)

cv2.imshow('J',J)
cv2.waitKey(0)
```

- Set the scale factor to a number > 1 (e.g. s =2) and see the result.
- How can we change the size of the output image accordingly?

Affine transformation

The affine transformation is in the form of $x' = Ax + t$, where A is an arbitrary 2 by 2 matrix. Thus, $M = [A\ t]$.

File: **affine.py**

```
import cv2
import numpy as np

I = cv2.imread('karimi.jpg')

t = np.array([[30],
              [160]], dtype=np.float32)
A = np.array([[.7, 0.8],
              [-0.3, .6]], dtype=np.float32)


M = np.hstack([A,t])

output_size = (I.shape[1], I.shape[0])
J = cv2.warpAffine(I,M,  output_size)

cv2.imshow('I',I)
cv2.waitKey(0)

cv2.imshow('J',J)
cv2.waitKey(0)
```

- Notice that the parallel line remain parallel.
- Change the elements of matrix `A` and see the results.
- What happens when matrix `A` is diagonal?

Perspective transformation (Homography)

A perspective transformation (projective transformation or homography) can be represented by a 3 by 3 matrix H. Look at the following example. It adds some perspective to the above affine transformation:

File: **perspective.py**

```
import cv2
import numpy as np

I = cv2.imread('karimi.jpg')

t = np.array([[30],
              [160]], dtype=np.float32)
A = np.array([[.7, 0.8],
              [-0.3, .6]], dtype=np.float32)

M = np.hstack([A,t])

# perspective effect
p = np.array([[0.001,0.002, 1]])

H = np.vstack([M,
               p])


output_size = (I.shape[1], I.shape[0])
J = cv2.warpPerspective(I,H,  output_size)

cv2.imshow('I',I)
cv2.waitKey(0)

cv2.imshow('J',J)
cv2.waitKey(0)
```

- Change the values of p[0] and p[1] and see what happens. Set them to 0 or negative values.

Estimating a homography transformation from point correspondences

If we have a set of 2D points $x_1, x_2, \dots, x_m$ in one image and a set of **corresponding** points $y_1, y_2, \dots, y_m$ in the second image, we can estimate a transformation which maps each point x_i to its corresponding point y_i (or sometimes a point close to y_i).

In the next example, we have two photographs of a painting taken from different views. Thus, the relation between them is a homography.

We want to map the first image to the second. We have found four pairs of corresponding points (corners of the frame) in both images. Using four point correspondences we can estimate a perspective transformation matrix **H** using the function **cv2.getPerspectiveTransform**. We then apply the transformation to the first image. Run the following file and see the results.

File: **compute_perspective.py**

```
import cv2
import numpy as np

I1 = cv2.imread('farshchian1.jpg')
I2 = cv2.imread('farshchian2.jpg')

points1 = np.array([(82,14),
                    (242,17),
                    (241, 207),
                    (81, 206)]).astype(np.float32)

points2 = np.array([(46,75),
                    (196,61),
                    (220,227),
                    (76,251)]).astype(np.float32)
```

```
for i in range(4):
    cv2.circle(I1, (points1[i,0], points1[i,1]), 3, [0,0,255],2)
    cv2.circle(I2, (points2[i,0], points2[i,1]), 3, [0,0,255],2)

# compute homography from point correspondences
H = cv2.getPerspectiveTransform(points1, points2)

output_size = (I2.shape[1], I2.shape[0])
J = cv2.warpPerspective(I1,H,  output_size)

cv2.imshow('I1',I1)
cv2.waitKey(0)

cv2.imshow('I2',I2)
cv2.waitKey(0)

cv2.imshow('J',J)
cv2.waitKey(0)
```

- Can you think of an application for this, considering that **I1** has a better quality than **I2**?
- How can you transform **I2** to **I1**?

Task 2: Perspective Correction

Look at the following traffic sign. We want to correct the perspective and extract the sign plate as if we are looking at it from the front. The transformed image **J** must exactly contain the sign plate (and not other parts of the image). We have already found the coordinates of the four corners of the sign plate and stored them in the array `points1`. Complete the task by changing the following code. You need to find the proper transformation matrix **H** and apply it to the image.

File: **task2.py**

```
import numpy as np
import cv2

I = cv2.imread('sign.jpg')

p1 = (135,105)
p2 = (331,143)
p3 = (356,292)
p4 = (136,290)
```

```
points1 = np.array([p1,p2,p3,p4], dtype=np.float32)

n = 480
m = 320
output_size = (n,m)

J = np.zeros((m,n)) # delete this!!

# mark corners of the plate in image I
for i in range(4):
    cv2.circle(I, (points1[i,0], points1[i,1]), 5, [0,0,255],2)

cv2.imshow('I', I)
cv2.waitKey(0)

cv2.imshow('J', J)
cv2.waitKey()
```