CSE 252A Computer Vision I Fall 2018 - Assignment 1 Solutions

Instructor: David Kriegman

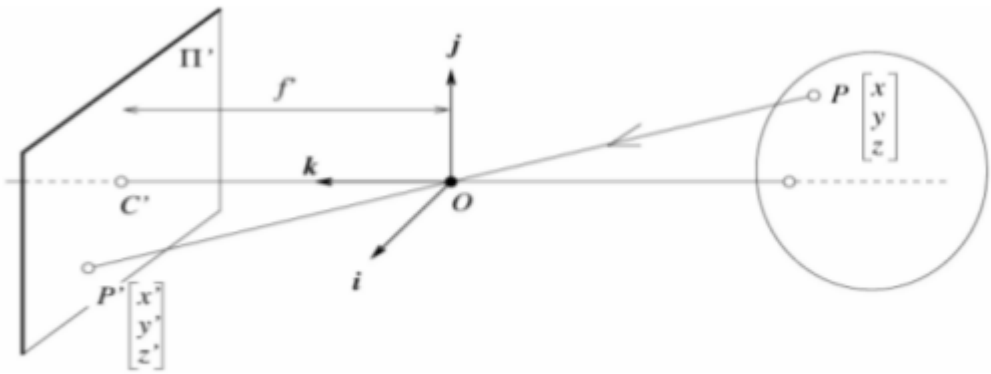
Some of these answers have been built using exemplar submissions from this year's class.

Problem 1: Perspective Projection [5 pts]

Consider a perspective projection where a point

$$P = [\mathrm{x\ y\ z}]^T$$

is projected onto an image plane Π' represented by $k = f' > 0$ as shown in the following figure.



The first second and third coordinate axes are denoted by i, j, k respectively.

Consider the projection of two rays in the world coordinate system

$$Q1 = [7\ \text{-}3\ 1] + t[8\ 2\ 4]$$
$$Q2 = [2\ \text{-}5\ 9] + t[8\ 2\ 4]$$

where $-\infty \leq t \leq -1$.

Calculate the coordinates of the endpoints of the projection of the rays onto the image plane. Identify the vanishing point based on the coordinates.

Solution

From similar triangles, we have

$$P' = \begin{bmatrix} \frac{f'x}{z} \\ \frac{f'y}{z} \end{bmatrix}$$

The points on the rays are represented as,

$$Q1 = \begin{bmatrix} 7+8t \\ -3+2t \\ 1+4t \end{bmatrix}$$
$$Q2 = \begin{bmatrix} 2+8t \\ -5+2t \\ 9+4t \end{bmatrix}$$

The projection of these points on the image place are,

$$Q1' = \begin{bmatrix} \frac{f'(7+8t)}{1+4t} \\ \frac{f'(-3+2t)}{1+4t} \end{bmatrix}$$
$$Q2' = \begin{bmatrix} \frac{f'(2+8t)}{9+4t} \\ \frac{f'(-5+2t)}{9+4t} \end{bmatrix}$$

For endpoints of Q1',

Setting t = -1,

$$Q1' = \begin{bmatrix} \frac{f'}{3} \\ \frac{5f'}{3} \end{bmatrix}$$

Setting t = $-\infty$,

$$Q1' = \begin{bmatrix} 2f' \\ \frac{f'}{2} \end{bmatrix}$$

For endpoints of Q2',

Setting t = -1,

$$Q2' = \begin{bmatrix} \frac{-6f'}{5} \\ \frac{-7f'}{5} \end{bmatrix}$$

Setting t = $-\infty$,

$$Q2' = \begin{bmatrix} 2f' \\ \frac{f'}{2} \end{bmatrix}$$

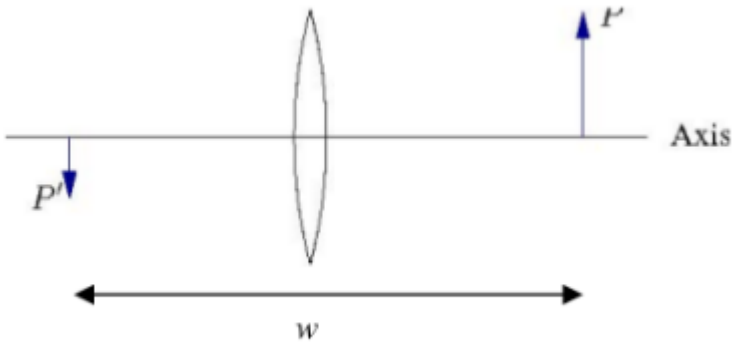
The vanishing point is hence given by $(2f', \frac{f'}{2})$ on the image plane. The rays being parallel, have the same vanishing point.

NOTE :

The end point of Q2 on the image plane is in fact incorrect since Q2 has its end point behind the image plane itself. The ray will still have a distinct end point on the image plane, as seen from the camera, from the point where the ray goes beyond the center of projection but it was not intentional to make the problem tricky and only a couple of students have figured this out. Full credit has been given to anyone that used the right approach even if then end point of Q2 is assumed to be valid.

Problem 2: Thin Lens Equation [5 pts]

An illuminated arrow forms a real inverted image of itself at a distance of $w = 60$ cm, measured along the optical axis of a convex thin lens as shown above. The image is half the size of the object



1. How far from the object must the lens be placed? Whats is the focal length of the lens?
2. At what distance from the center of the lens should the arrow be placed so that the height of the image is the same?
3. What would be the type and location of image formed if the arrow is placed at a distance of 5 cm along the optical axis from the optical center?

Solution

1. Let object distance be given by u and image distance be given by v By using similar triangles,

$\frac{Objectdistance}{ImageDistance} = \frac{Objectsize}{Imagesize} \rightarrow \frac{u}{v} = \frac{2}{1} \rightarrow u = 2v$

Since we know form the question that

$u + v = w = 60$
$\rightarrow u = 40cm, v = 20cm$

Using the formula for focal length, $\frac{1}{f} = \frac{1}{u} + \frac{1}{v}$

$$\rightarrow \frac{1}{f} = \frac{3}{40}$$
$$\rightarrow f = \frac{40}{3}$$

2.The image size and object size is equal when the object distance is equal to twice the focal length. This can be verified using the process in part 1.

3. We know from part 1 that f = 40/3 cm The new object distance, u = 5 cm

Using the same formula, $\frac{1}{f} = \frac{1}{u} + \frac{1}{v}$

$$\rightarrow \frac{40}{3} = \frac{1}{5} + \frac{1}{v}$$

On solving, v = -8 cm

The negative sign indicates that the image is formed on the same side of the lens as the object is placed. This is a magnified virtual image.

Problem 3: Affine Projection [3 pts]

Show that the image of a pair of parallel lines in 3D space is a pair of parallel lines in an affine camera.

Solution

Let the two parallel lines be

$$l: \overrightarrow{p} + t\overrightarrow{u}$$
$$m: \overrightarrow{q} + t\overrightarrow{v}$$

Where $t \in R$ and $\overrightarrow{u} = k\overrightarrow{v}$ for some $k \in R$

Since the direction vector of l is a scalar multiple of direction vector of m, we know that l and m are parallel

Note: It is important for the direction component to not be equal. While for any line $\overrightarrow{p} + t\overrightarrow{u}$, t is free to vary in R, while comparing two lines, using the same $t\overrightarrow{u}$ constrains the direction component to have equal values at all times.

The affine projection of these lines can be given by

$$l': A(\overrightarrow{p} + t\overrightarrow{u}) + \overrightarrow{b}$$
$$= (A\overrightarrow{p} + \overrightarrow{b}) + At\overrightarrow{u}$$
$$= \overrightarrow{p1} + t\overrightarrow{u1}$$
$$m': A(\overrightarrow{q} + t\overrightarrow{v}) + \overrightarrow{b}$$
$$= (A\overrightarrow{q} + \overrightarrow{b}) + At\overrightarrow{ku}$$
$$= \overrightarrow{q1} + kt\overrightarrow{u1}$$

Since the direction vector of l' is a scalar multiple of direction vector of m' we see that the projection of lines are parallel too.

Problem 4: Image Formation and Rigid Body Transformations [10 points]

In this problem we will practice rigid body transformations and image formations through the projective and affine camera model. The goal will be to photograph the following four points

$$^{A}P_1 = [\text{-1 -0.5 2}]^T$$

,

$$^{A}P_2 = [\text{1 -0.5 2}]^T$$

,

$$^{A}P_3 = [\text{1 0.5 2}]^T$$

,

$$^{A}P_4 = [\text{-1 0.5 2}]^T$$

To do this we will need two matrices. Recall, first, the following formula for rigid body transformation

$$^{B}P = {}^{B}_{A}R\,{}^{A}P + {}^{B}O_A$$

Where ^{B}P is the point coordinate in the target (B) coordinate system. ^{A}P is the point coordinate in the source (A) coordinate system. ${}^{B}_{A}R$ is the rotation matrix from A to B, and $^{B}O_A$ is the origin of the coordinate system A expressed in B coordinates.

The rotation and translation can be combined into a single 4 $\times$ 4 extrinsic parameter matrix, P_e, so that $^{B}P = P_e \cdot {}^{A}P$.

Once transformed, the points can be photographed using the intrinsic camera matrix, P_i which is a 3 $\times$ 4 matrix.

Once these are found, the image of a point, ^{A}P, can be calculated as $P_i \cdot P_e \cdot {}^{A}P$.

We will consider four different settings of focal length, viewing angles and camera positions below. For each of these calculate:

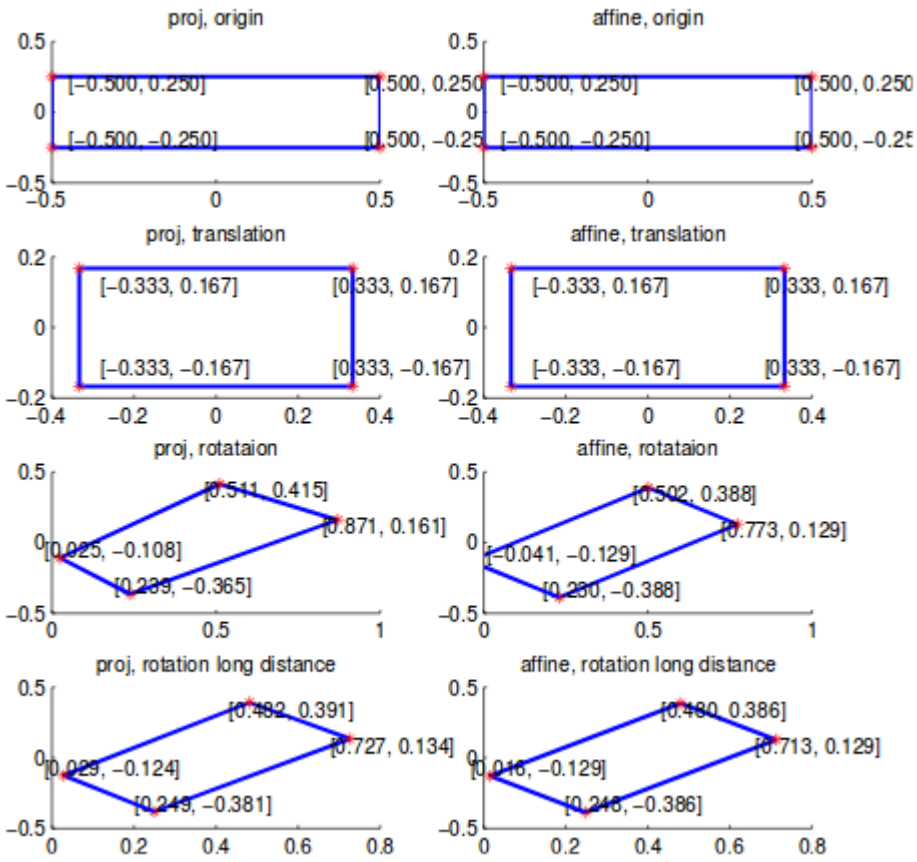
a) Extrinsic transformation matrix,

b) Intrinsic camera matrix under the perspective camera assumption.

c) Intrinsic camera matrix under the affine camera assumption. In particular, around what point do you do the taylor series expansion?

d) Calculate the image of the four vertices and plot using the supplied functions

Your output should look something like the following image (Your output values might not match, this is just an example)



1. [No rigid body transformation]. Focal length = 1. The optical axis of the camera is aligned with the z-axis.
2. [Translation]. ${}^{B}O_{A} = [0\ 0\ 1]^{T}$. Focal length = 1. The optical axis of the camera is aligned with the z-axis.
3. [Translation and Rotation]. Focal length = 1. ${}^{B}_{A}R$ encodes a 30 degrees around the z-axis and then 60 degrees around the y-axis. ${}^{B}O_{A} = [0\ 0\ 1]^{T}$.
4. [Translation and Rotation, long distance]. Focal length = 5. ${}^{B}_{A}R$ encodes a 30 degrees around the z-axis and then 60 degrees around the y-axis. ${}^{B}O_{A} = [0\ 0\ 13]^{T}$.

> You can refer the Richard Szeliski starting page 36 for image formation and the extrinsic matrix.
>
> Intrinsic matrix calculation for perspective and affine camera models was covered in class and can be referred in slide 3 http://cseweb.ucsd.edu/classes/fa18/cse252A-a/lec3.pdf (http://cseweb.ucsd.edu/classes/fa18/cse252A-a/lec3.pdf)

We will not use a full intrinsic camera matrix (e.g. that maps centimeters to pixels, and defines the coordinates of the center of the image), but only parameterize this with f, the focal length. In other words: the only parameter in the intrinsic camera matrix under the perspective assumption is f, and the only ones under the affine assumption are: f, x_0, y_0, z_0, where x_0, y_0, z_0 is the center of the taylor series expansion.

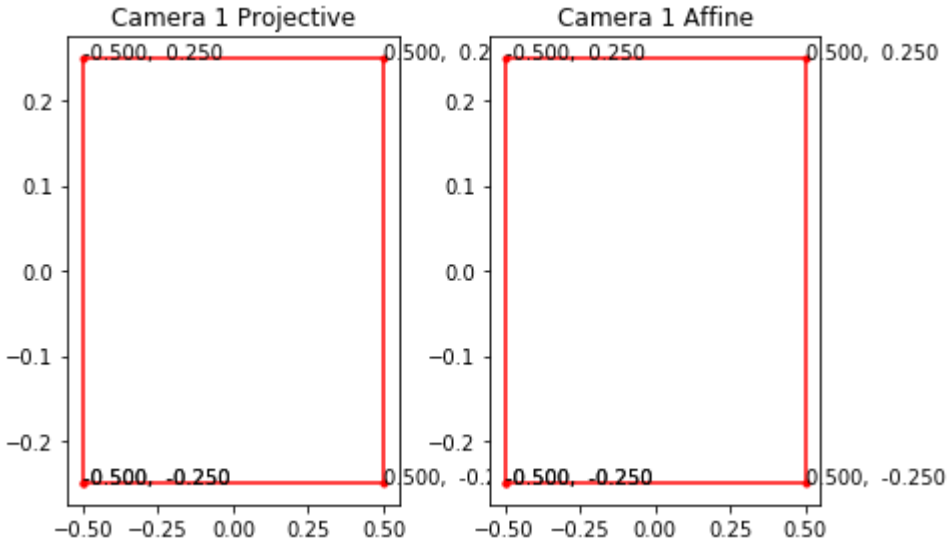
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math


# convert points from euclidian to homogeneous
# Assume we are given a column vector
def to_homog(points):
    # write your code here

    # Add a row of ones as the last coordinate
    add_row = np.ones(points.shape[1])
    points = np.vstack((points,add_row))


    return points

# convert points from homogeneous to euclidian
def from_homog(points_homog):
    # write your code here

    points_homog_out = np.eye(points_homog.shape[0]-1,points_homog.shape[1])
    for col in range(points_homog.shape[1]):
        # Divide each point by the last dimension
        points_homog[:,col] = points_homog[:,col]/points_homog[-1,col]

    # Remove the last row by including all rows but the last row (slice is exc
lusive)
    points_homog_out[:,:] = points_homog[0:-1, :]

    return points_homog_out

# project 3D euclidian points to 2D euclidian
def project_points(P_int, P_ext, pts):
    # write your code here
#     print("Points Matrix:")
#     print(pts)

    output = np.eye(2,4)

    for point in range(pts.shape[1]):
        # Multiply the intrinisic matrix and the extrinsic matrix for combined
 transformation matrix
        P_int_ext = np.matmul(P_int, P_ext)

        # Calculate the projected point
        # convert the point to homogeneous
        output[:,point] = from_homog(np.matmul(P_int_ext,
                                               to_homog(pts[:,point][np.newaxi
s].T))).flatten()

#         print("Output" + str(point) + ":")
#         print(output)

    return output
```

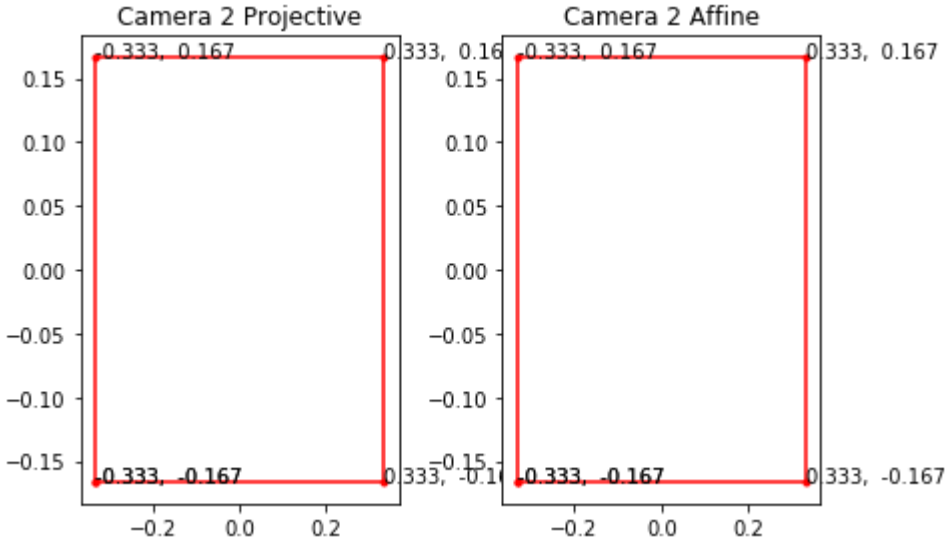
```
# Change the three matrices for the four cases as described in the problem
# in the four camera functions geiven below. Make sure that we can see the for
mula
# (if one exists) being used to fill in the matrices. Feel free to document wi
th
# comments any thing you feel the need to explain.

# This function computes the affine matrix for each camera
# The point that is chosen becomes multipled by the extrinsic parameters
# to obtain x0, y0, and z0
def compute_affine(f, P_ext):
    P_int_affine = np.eye(3,4)
    # Pick point [0 0 2].T for affine (center of all 4 points)
    # Need to apply extrinsic matrix to point, since point is in camera plane
    # From this we obtain new x0, y0, z0

    t_exp_points = np.matmul(P_ext, to_homog(np.array([[0, 0, 2]]).T))
    x0 = t_exp_points[0,0]
    y0 = t_exp_points[1,0]
    z0 = t_exp_points[2,0]

    # print(str(x0) + " " + str(y0) + " " + str(z0))

    # f/z0
    P_int_affine[0,0] = f/z0
    # -fx0/z0^2
    P_int_affine[0,2] = -(f*x0)/(z0**2)
    # fx0/z0
    P_int_affine[0,3] = (f*x0)/z0
    # f/z0
    P_int_affine[1,1] = f/z0
    # -fy0/z0^2
    P_int_affine[1,2] = -(f*y0)/(z0**2)
    # fy0/z0
    P_int_affine[1,3] = (f*y0)/z0

    P_int_affine[2,2] = 0
    P_int_affine[2,3] = 1

    return P_int_affine

def camera1():
    # write your code here
    # Focal Length 1, aligned with z axis
    f = 1
    # The intrinsic camera matrix for camera 1 using perspective camera assump
tion
    # is simply the identity for 3x4, since f = 1
    P_int_proj = np.eye(3,4)

    # No rigid body transformation, so identity is fine
    P_ext = np.eye(4,4)

    P_int_affine = compute_affine(f, P_ext)

    return P_int_proj, P_int_affine, P_ext
```

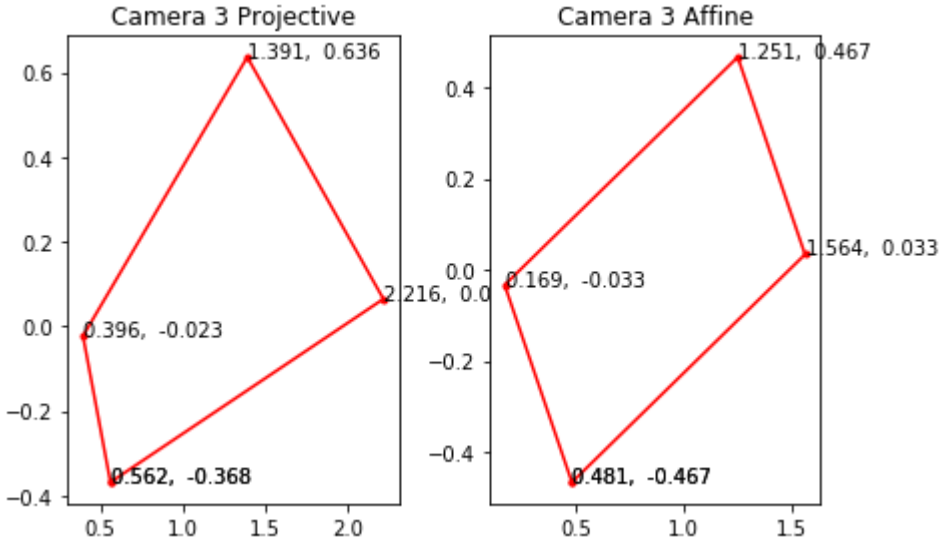
```
def camera2():
    # write your code here
    # Focal Length 1, O = [0 0 1].T, aligned with z axis
    f = 1
    # f = 1, so identity is fine here
    P_int_proj = np.eye(3,4)

    # The extrinsic matrix is [R O; 0 1], size 4x4
    # Rotation is simply identity, O = [0 0 1].T
    P_ext = np.eye(4,4)
    P_ext[:,3] = np.array([0,0,1,1])

    P_int_affine = compute_affine(f, P_ext)

#     print(P_ext)
    return P_int_proj, P_int_affine, P_ext

def camera3():
    # write your code here
    # Focal Length = 1, first 30 degrees about z, then 60 about y
    # O = [0 0 1].T
    f = 1
    # f = 1, identity is fine here
    P_int_proj = np.eye(3,4)

    P_ext = np.eye(4,4)
    R_30z = np.eye(4,4)
    R_60y = np.eye(4,4)

    # cos(30)
    R_30z[0,0] = np.cos(np.deg2rad(30))
    # -sin(30)
    R_30z[0,1] = -np.sin(np.deg2rad(30))
    # sin(30)
    R_30z[1,0] = np.sin(np.deg2rad(30))
    # cos(30)
    R_30z[1,1] = np.cos(np.deg2rad(30))

    # cos(60)
    R_60y[0,0] = np.cos(np.deg2rad(60))
    # sin(60)
    R_60y[0,2] = np.sin(np.deg2rad(60))
    # -sin(60)
    R_60y[2,0] = -np.sin(np.deg2rad(60))
    # cos(60)
    R_60y[2,2] = np.cos(np.deg2rad(60))

    P_ext = np.matmul(R_60y, R_30z)

    P_ext[:,3] = np.array([0,0,1,1])
#     print(P_ext)

    P_int_affine = compute_affine(f, P_ext)

    return P_int_proj, P_int_affine, P_ext
```

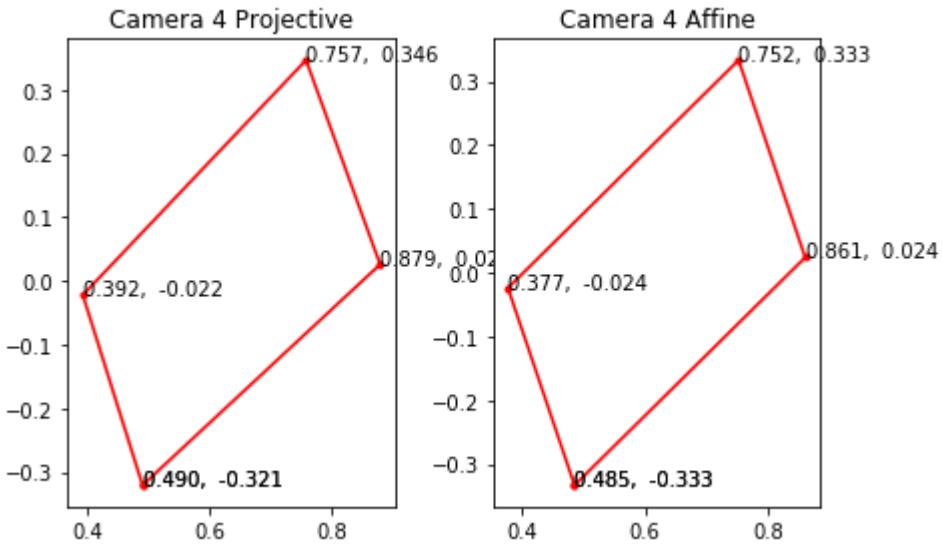
```
def camera4():
    # write your code here
    # Focal length is 5, first 30 degrees about z, then 60 degrees about y
    # O = [0 0 13].T
    f = 5
    P_int_proj = np.eye(3,4)
    # The focal length is 5, so identity is no longer sufficient
    P_int_proj[2,2] = 1/5
#     print(P_int_proj)

    P_ext = np.eye(4,4)
    R_30z = np.eye(4,4)
    R_60y = np.eye(4,4)

    # cos(30)
    R_30z[0,0] = np.cos(np.deg2rad(30))
    # -sin(30)
    R_30z[0,1] = -np.sin(np.deg2rad(30))
    # sin(30)
    R_30z[1,0] = np.sin(np.deg2rad(30))
    # cos(30)
    R_30z[1,1] = np.cos(np.deg2rad(30))

    # cos(60)
    R_60y[0,0] = np.cos(np.deg2rad(60))
    # sin(60)
    R_60y[0,2] = np.sin(np.deg2rad(60))
    # -sin(60)
    R_60y[2,0] = -np.sin(np.deg2rad(60))
    # cos(60)
    R_60y[2,2] = np.cos(np.deg2rad(60))

    P_ext = np.matmul(R_60y, R_30z)

    P_ext[:,3] = np.array([0,0,13,1])

    P_int_affine = compute_affine(f, P_ext)

    return P_int_proj, P_int_affine, P_ext


# Use the following code to display your outputs
# You are free to change the axis parameters to better
# display your quadrilateral but do not remove any annotations

def plot_points(points, title='', style='.-r', axis=[]):
    inds = list(range(points.shape[1]))+[0]
    plt.plot(points[0,inds], points[1,inds],style)

    for i in range(len(points[0,inds])):
        plt.annotate(str("{0:.3f}".format(points[0,inds][i]))+",  "+str("{0:.3
f}".format(points[1,inds][i])),(points[0,inds][i], points[1,inds][i]))

    if title:
        plt.title(title)
    if axis:
        plt.axis(axis)
```

```
    plt.tight_layout()

def main():
    point1 = np.array([[-1,-.5,2]]).T
    point2 = np.array([[1,-.5,2]]).T
    point3 = np.array([[1,.5,2]]).T
    point4 = np.array([[-1,.5,2]]).T
    points = np.hstack((point1,point2,point3,point4))

    for i, camera in enumerate([camera1, camera2, camera3, camera4]):
        P_int_proj, P_int_affine, P_ext = camera()
        plt.subplot(1, 2, 1)
        plot_points(project_points(P_int_proj, P_ext, points), title='Camera %
d Projective'%(i+1))
        plt.subplot(1, 2, 2)
        plot_points(project_points(P_int_affine, P_ext, points), title='Camera
 %d Affine'%(i+1))
        plt.show()

main()
```


Camera 1 Projective
0.500, 0.250
0.500, -0.250
Camera 1 Affine
0.500, 0.250
0.500, -0.250


Camera 2 Projective
0.333, 0.167
0.333, -0.167
Camera 2 Affine
0.333, 0.167
0.333, -0.167


Camera 3 Projective
1.391, 0.636
2.216, 0.0
0.396, -0.023
0.562, -0.368
Camera 3 Affine
1.251, 0.467
1.564, 0.033
0.169, -0.033
0.481, -0.467



1

The points chosen are
camera 1 = (0,0,2)
camera 2 = (0,0,3)
camera 3 = (1.732,0,2)
camera 4 = (1.732,0,14)

2

We see that relative to the camera's coordinate frame the affine camera works best when all the points lie in a small neighbourhood about chosen point. By definition, the centroid is the point which is closest to all the 4 points. Hence we compute the centroid of the figure, to find the point about which we must compute the taylor series expansion.
Note that in camera2,3 and 4, the camera coordinate axis does not coincide with the object coordinate axis. We need to compute the centroid with respect to the points relative to the camera frame of reference.
This is equivalent to finding the centroid in the object frame of reference, and then multiplying it by the extrinsic camera matrix.

3

The projective model is the pinhole camera model. The affine camera model is a first order approximation around the expansion point, if the geometry of the image is similar to that which is around the expansion point, the result is more accurate. The difference is smaller for the last image, because the points are treated as if they are far away, so the object is small in perspective of the image, and the first order approximation around the expansion point is valid for all the points.

Problem 5: Homography [12 pts]

You may use eig or svd routines in python for this part of the assignment.

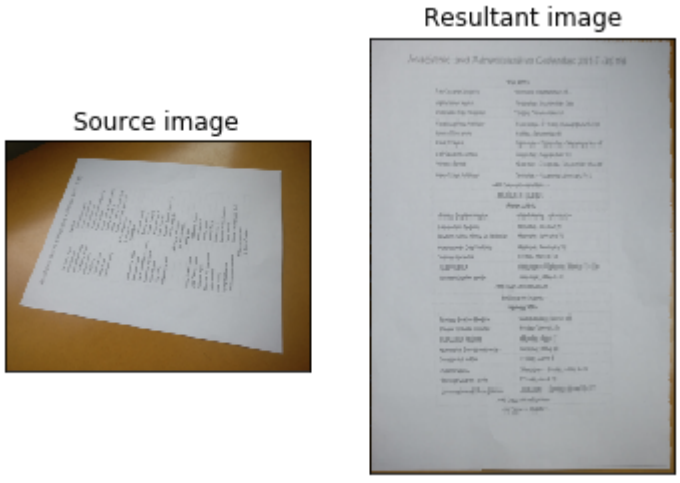
Consider a vision application in which components of the scene are replaced by components from another image scene.

In this problem, we will implement partial functionality of a smartphone camera scanning application (Example: CamScanner) that, in case you've never used before, takes pictures of documents and transforms it by warping and aligning to give an image similar to one which would've been obtained through using a scanner.

The transformation can be visualized by imagining the use of two cameras forming an image of a scene with a document. The scene would be the document you're trying to scan placed on a table and one of the cameras would be your smart phone camera, forming the image that you'll be uploading and using in this assignment. There can also be an ideally placed camera, oriented in the world in such a way that the image it forms of the scene has the document perfectly algined. While it is unlikely you can hold your phone still enough to get such an image, we can use homography to transform the image you take into the image that the ideally placed camera would have taken.

This digital replacement is accomplished by a set of corresponding points for the document in both the source (your picture) and target (the ideal) images. The task then consists of mapping the points from the source to their respective points in the target image. In the most general case, there would be no constraints on the scene geometry, making the problem quite hard to solve. If, however, the scene can be approximated by a plane in 3D, a solution can be formulated much more easily even without the knowledge of camera calibration parameters.

To solve this section of the homework, you will begin by understanding the transformation that maps one image onto another in the planar scene case. Then you will write a program that implements this transformation and use it to warp some document into a well aligned document (See the given example to understand what we mean by well aligned).



To begin with, we consider the projection of planes in images. imagine two cameras C_1 and C_2 looking at a plane π in the world. Consider a point P on the plane π and its projection $p = [\mathrm{u1}, \mathrm{v1}, 1]^T$ in the image 1 and $q = [\mathrm{u2}, \mathrm{v2}, 1]^T$ in image 2.

There exists a unique, upto scale, 3 $\times$ 3 matrix H such that, for any point P:

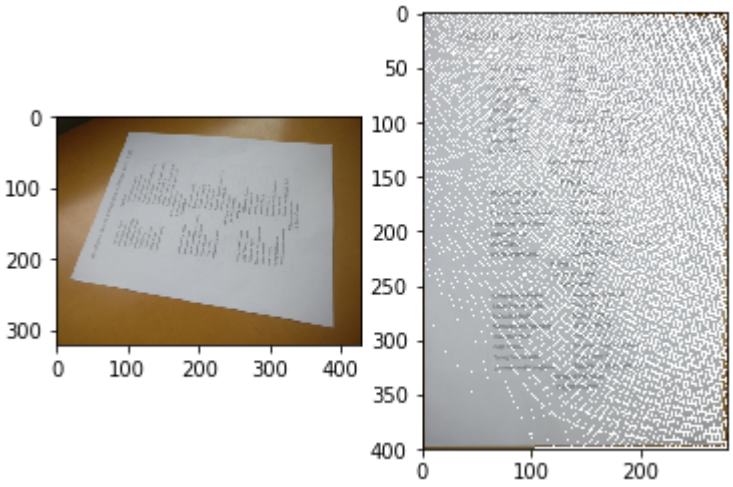
$$q \approx Hp$$

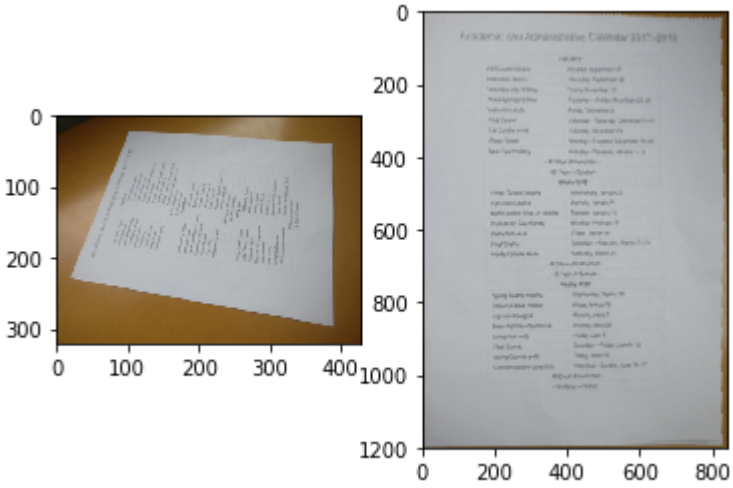
Here $\approx$ denotes equality in homogeneous coordinates, meaning that the left and right hand sides are proportional. Note that H only depends on the plane and the projection matrices of the two cameras.

The interesting thing about this result is that by using H we can compute the image of P that would be seen in the camera with center C_2 from the image of the point in the camera with center at C_1, without knowing the three dimensional location. Such an H is a projective transformation of the plane, called a homography.

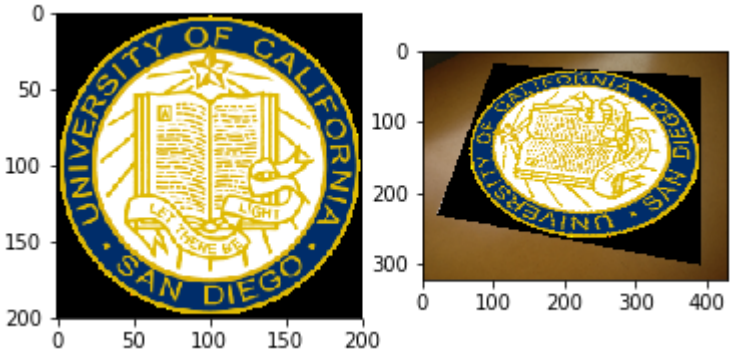
In this problem, complete the code for computeH and warp functions that can be used in the skeletal code that follows.

There are three warp functions to implement in this assignment, example ouputs of which are shown below. In warp1, you will create a homography from points in your image to the target image (Mapping source points to target points). In warp2, the inverse of this process will be done. In warp3, you will create a homography between a given image and your image, replacing your document with the given image.

1. 

2. 

3.

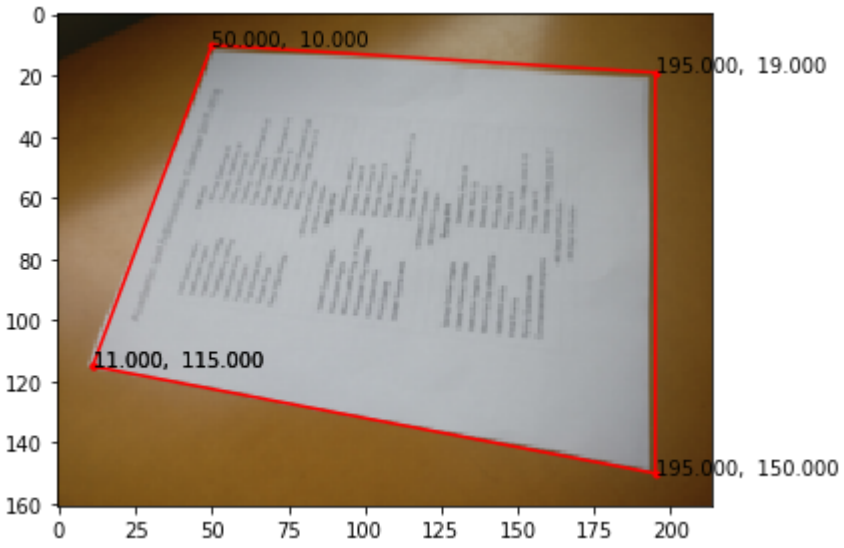


2. In the context of this problem, the source image refers to the image of a document you take that needs to be replaced into the target.
3. The target image can start out as an empty matrix that you fill out using your code.
4. You will have to implement the computeH function that computes a homography. It takes in the point correspondences between the source image and target image in homogeneous coordinates respectively and returns a 3×3 homography matrix.
5. You will also have to implement the three warp functions in the skeleton code given and plot the resultant image pairs. For plotting, make sure that the target image is not smaller than the source image.

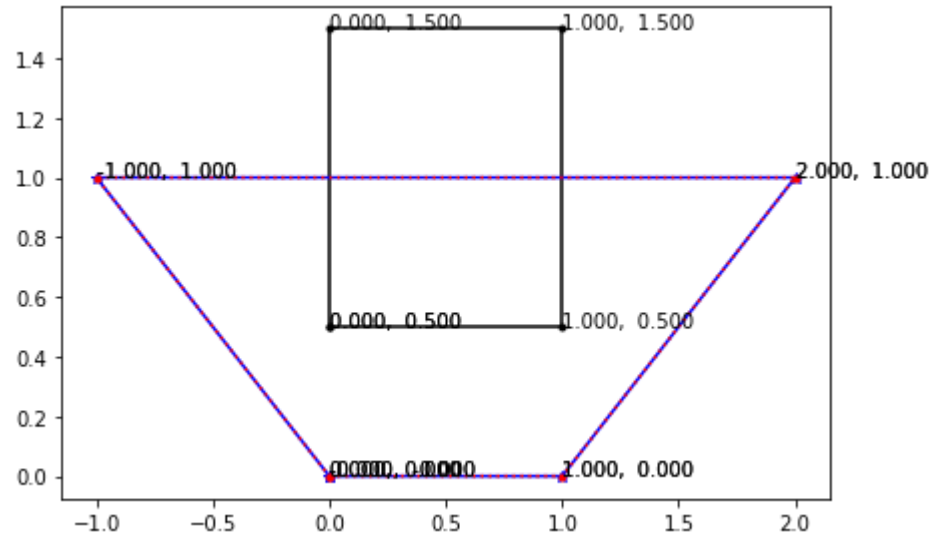
Note: We have provided test code to check if your implementation for computeH is correct. All the code to plot the results needed is also provided along with the code to read in the images and other data required for this problem. Please try not to modify that code.

You may find following python built-ins helpful: numpy.linalg.svd, numpy.meshgrid

In [3]:
```
import numpy as np
from scipy.misc import imread, imresize
from scipy.io import loadmat
import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings('ignore')

# load image to be used - resize to make sure it's not too large
# You can use the given image as well
# A large image will make testing you code take longer; once you're satisfied
 with your result,
# you can, if you wish to, make the image larger (or till your computer memory
 allows you to)
source_image = imresize(imread("photo.jpg"),.1)[:,:,:3]/255.

# display images
plt.imshow(source_image)

# Align the polygon such that the corners align with the document in your pict
ure
# This polygon doesn't need to overlap with the edges perftectly, an approxima
tion is fine
# The order of points is clockwise, starting from bottom left.
x_coords = [11,50,195,195]
y_coords = [115,10,19,150]

# Plot points from the previous problem is used to draw over your image
# Note that your coordinates will change once you resize your image again
source_points = np.vstack((x_coords, y_coords))
plot_points(source_points)

plt.show()
```

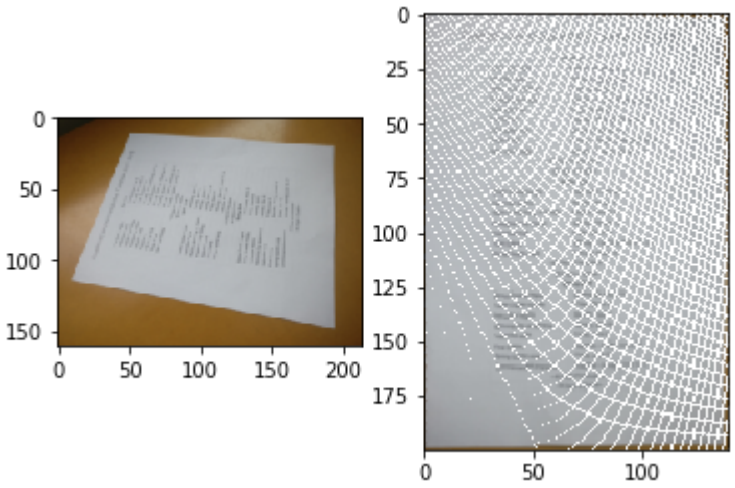


In [4]:
```
def computeH(source_points, target_points):
    # returns the 3x3 homography matrix such that:
    # np.matmul(H, source_points) ~ target_points
    # where source_points and target_points are expected to be in homogeneous

    # Please refer the note on DLT algorithm given at:
    # https://cseweb.ucsd.edu/classes/wi07/cse252a/homography_estimation/homography_estimation.pdf

    # make sure points are 3D homogeneous
    assert source_points.shape[0]==3 and target_points.shape[0]==3

    A = []

    for i in range(4):
        x1 = source_points[0][i]
        y1 = source_points[1][i]
        x2 = target_points[0][i]
        y2 = target_points[1][i]

        row1 = [x1,y1,1.0,0.0,0.0,0.0,-x1*x2,-y1*x2,-x2]
        row2 = [0.0,0.0,0.0,x1,y1,1.0,-x1*y2,-y1*y2,-y2]

        A.append(row1)
        A.append(row2)


    A = np.matrix(A)
    A = np.matmul(A.T,A)
    U,s,V = np.linalg.svd(A)


    h = []
    for i in range(len(U)):
        h.append(U.item(i,-1))

    h = np.reshape(h,(3,3))
    return h

##########################################################
# test code. Do not modify
##########################################################
def test_computeH():
    source_points = np.array([[0,0.5],[1,0.5],[1,1.5],[0,1.5]]).T
    target_points = np.array([[0,0],[1,0],[2,1],[-1,1]]).T
    H = computeH(to_homog(source_points), to_homog(target_points))
    mapped_points = from_homog(np.matmul(H,to_homog(source_points)))

    plot_points(source_points,style='.-k')
    plot_points(target_points,style='*-b')
    plot_points(mapped_points,style='.:r')
    plt.show()
    print('The red and blue quadrilaterals should overlap if ComputeH is implemented correctly.')
test_computeH()
```



The red and blue quadrilaterals should overlap if ComputeH is implemented cor
rectly.

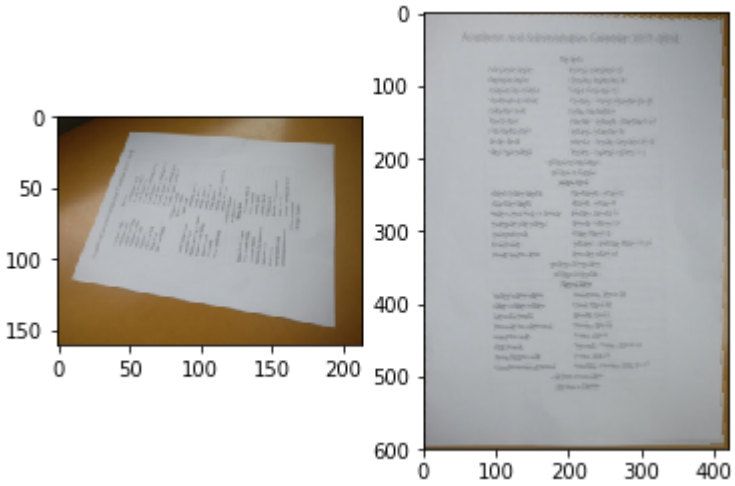
In [6]:
```
def warp(source_img, source_points, target_size):
    # Create a target image and select target points to create a homography an
d then create a warped version
    # of the image based on the homography by filling in the target image.
    # Make sure the new image (of size target_size) has the same number of col
or channels as source image
    assert target_size[2]==source_img.shape[2]


    target_points = np.array([[0,0],[target_size[1],0],[target_size[1],target_
size[0]],[0,target_size[0]]]).T


    H = computeH(to_homog(source_points), to_homog(target_points))
    res_img = np.ones(target_size)

    for i in range(source_img.shape[1]):
        for j in range(source_img.shape[0]):

            new_points = from_homog(np.matmul(H,to_homog(np.array([[i,j]]).T
)))


            x,y = int(new_points[0]),int(new_points[1])
            if x>=0 and x<target_size[1] and y>=0 and y<target_size[0]:
                res_img[y][x] = source_img[j][i]



    return res_img

result = warp(source_image, source_points, (200,140,3))
plt.subplot(1, 2, 1)
plt.imshow(source_image)
plt.subplot(1, 2, 2)
plt.imshow(result)
plt.show()
```



The output of warp1 of your code probably has some striations or noise. The larger you make your target image, the less it will resemble the document in the source image. Why is this happening?

To fix this, implement warp2, by creating an inverse homography matrix and fill in the target image.

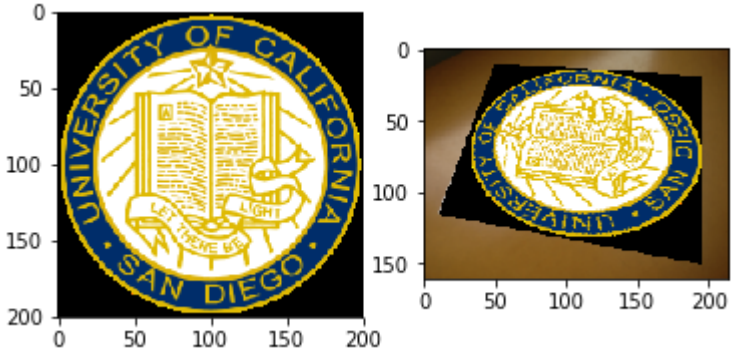
In [7]:
```
def warp2(source_img, source_points, target_size):
    # Create a target image and select target points to create a homography an
d then create a warped version
    # of the image based on the homography by filling in the target image.
    # Make sure the new image (of size target_size) has the same number of col
or channels as source image
    assert target_size[2]==source_img.shape[2]


    target_points = np.array([[0,0],[target_size[1],0],[target_size[1],target_
size[0]],[0,target_size[0]]]).T


    Hinv = computeH(to_homog(target_points), to_homog(source_points))
    res_img = np.ones(target_size)

    for i in range(target_size[1]):
        for j in range(target_size[0]):

            new_points = from_homog(np.matmul(Hinv,to_homog(np.array([[i,j]]).
T)))

            y,x = int(new_points[0]),int(new_points[1])
            if x>=0 and x<source_img.shape[0] and y>=0 and y<source_img.shape[
1]:
                try:
                    res_img[j][i] = source_img[x][y]
                except:
                    print(x,y,i,j)



    return res_img

result = warp2(source_image, source_points, (600,420,3))
plt.subplot(1, 2, 1)
plt.imshow(source_image)
plt.subplot(1, 2, 2)
plt.imshow(result)
plt.show()
```



Try playing around with the size of your target image in warp1 versus in warp2, additionally you can also implement nearest pixel interpolation or bi-linear interpolations and see if that makes a difference in your output.

In warp3, you'll be replacing the document in your image with a provided image. Read in "ucsd_logo.png" as the source image, keeping your document as the target.

In [10]:
```
source_image2 = imread('ucsd_logo.png')[:,:,:3]/255.

def warp3(target_image, target_points, source_image):
    # Create a target image and select target points to create a homography an
d then create a warped version
    # of the image based on the homography by filling in the target image.
    # Make sure the new image (of size target_size) has the same number of col
or channels as source image


    # Selecting the four points on the source image
    source_size = source_image.shape
    source_points = np.array([[0,0],[source_size[1],0],[source_size[1],source_
size[0]],[0,source_size[0]]]).T

    # Target points from previous cells, we choose to inverse mapping because
 of our demonstration
    # from warp1 and warp2 that inverse mapping is the better approach
    Hinv = computeH(to_homog(target_points), to_homog(source_points))


    # Fill in the values in the target image form source image
    for i in range(target_image.shape[1]):
        for j in range(target_image.shape[0]):

            # Transformed points after applying homography
            new_points = from_homog(np.matmul(Hinv,to_homog(np.array([[i,j]]).
T)))

            y,x = int(new_points[0]),int(new_points[1])
            # Check if point exists in source image
            if x>=0 and x<source_size[1] and y>=0 and y<source_size[0]:
                target_image[j][i] = source_image[x][y]



    return target_image

############################################################
# test code. Do not modify
############################################################
result = warp3(source_image, source_points, source_image2)
plt.subplot(1, 2, 1)
plt.imshow(source_image2)
plt.subplot(1, 2, 2)
plt.imshow(result)
plt.show()
```



Note

It was an expectation for the student to learn that inverse mapping is better, which is exactly what we discovered in doing warp1 and warp2. Warp3 with forward mapping also gives a visibly worse output, which is why some students have been given only partial credit for warp3.