CSE 252A (Computer Vision I) · Fall 2020 · Assignment 1

Instructor: David Kriegman

Assignment published on Friday, October 16, 2020

Due on Friday, October 30, 2020 at 11:59 pm Pacific Time

Instructions

- Review the academic integrity and collaboration policies on Canvas. This assignment must be completed individually.
- All solutions must be written in this notebook. Programming aspects of the assignment must be completed using Python (preferably 3.6+).
- If you want to modify the skeleton code, you may do so. The existing code is merely intended to provide you with a framework for your solution.
- You may use Python packages for basic linear algebra (e.g. simple operations from NumPy or SciPy), but you may **not** use packages that directly solve the problem. If you are unsure about using a specific package or function, please ask the instructor and teaching assistants for clarification.
- You must submit, through Gradescope, both (1) this notebook exported as a PDF **and** (2) this notebook as an `.ipynb` file. You must mark every problem in the PDF on Gradescope. If you do not submit both the `.pdf` and `.ipynb`, and/or if you do not mark every problem in the PDF on Gradescope, you may receive a penalty.
- It is highly recommended that you begin working on this assignment early.
- **Late policy:** Assignments submitted late will receive a 10% grade reduction for each day late (e.g. an assignment submitted an hour after the due date will receive a 10% penalty, an assignment submitted 10 hours after the due date will receive a 10% penalty, and an assignment submitted 28 hours after the due date will receive a 20% penalty). Assignments will not be accepted 72 hours after the due date. If you require an extension (for personal reasons only), you must request one as far in advance as possible. Extensions requested close to or after the due date will only be granted for clear emergencies or clearly unforeseeable circumstances.

Problem 1: Perspective Projection and Homogenous Coordinates [10 pts]

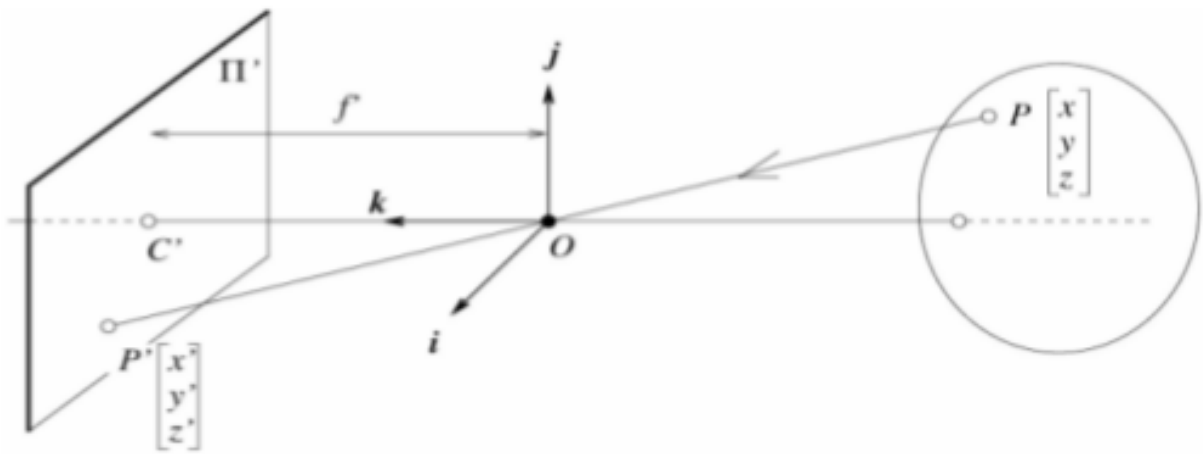
Part 1 [3 pts]

Consider a perspective projection where a point

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

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

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

$$Q1 = [4\ \text{-}1\ 3] + t[5\ 3\ 3]$$

$$Q1 = [4\ \text{-}1\ 3] + t[5\ 3\ 3]$$

$$Q2 = [1\ \text{-}3\ 2] + t[5\ 3\ 3]$$

$$Q2 = [1\ \text{-}3\ 2] + t[5\ 3\ 3]$$

where $-\infty \leq t \leq -2$.

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

Part 2 [6 pts]

Show that:

1) In $\mathbb{R}^3$ R^3 distances are preserved under a rigid transformation.

2) In $\mathbb{R}^2$ R^2 parallel lines remain parallel under an affine transformation.

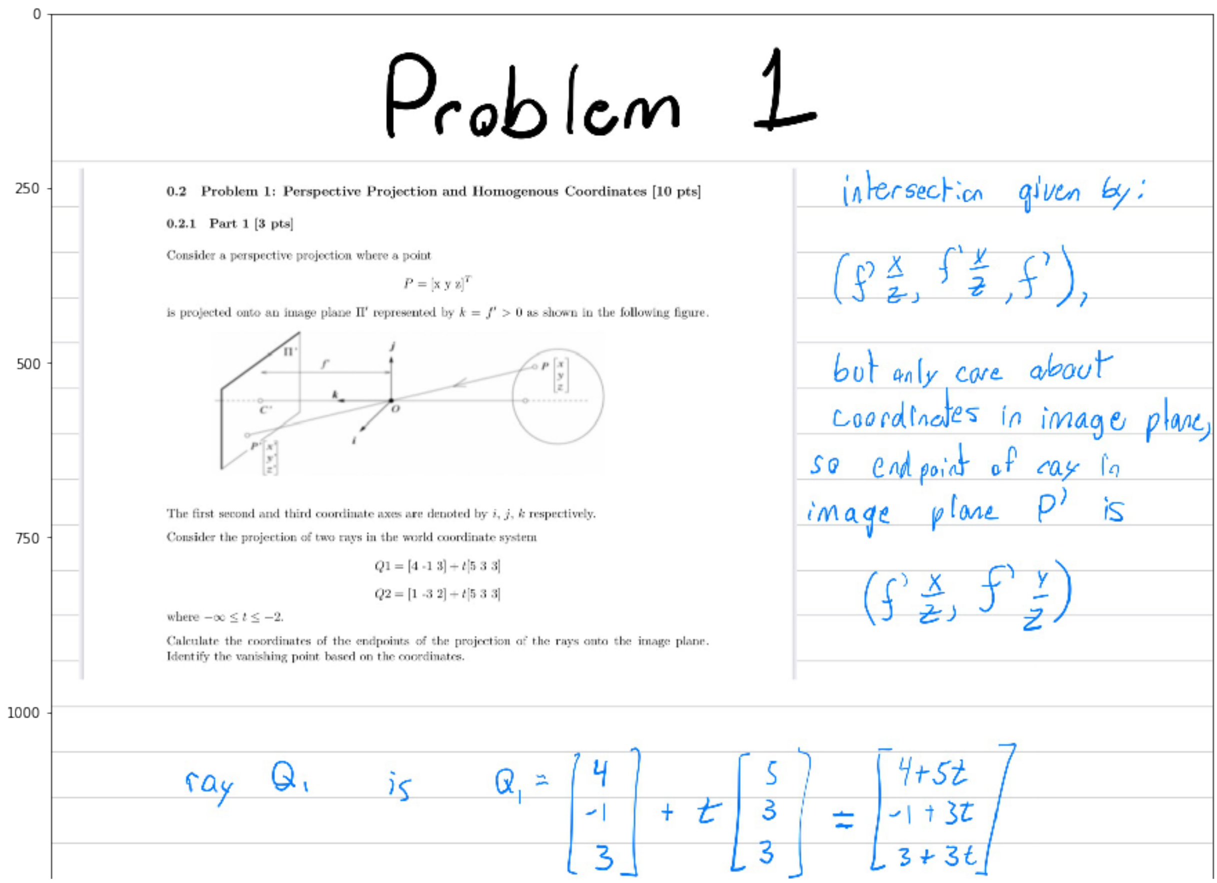
3) If $a, b \in \mathbb{R}^3$ $a, b \in \mathrm{R}^3$ are orthogonal, they are orthogonal after rotating by a rotation matrix $\boldsymbol{R}R$

Part 3 [1 pts]

Given four points forming a square with Cartesian coordinates (0,0), (1,0), (1,1), and (0,1), find a projective transformation, A, which sends any two of the points to infinity while the other two are not sent to infinity. Define the matrix A below and print the result of applying the transformation to each of the four points.

In [112]:

```python
#Problem 1 Part 1.1
import matplotlib.pyplot as plt

WrittenOne = plt.imread('CSE 252A_HW1_1.jpg')                # read a 
WrittenTwo = plt.imread('CSE 252A_HW1_2.jpg')
WrittenThree = plt.imread('CSE 252A_HW1_3.jpg')
WrittenFour = plt.imread('CSE 252A_HW1_4.jpg')
plt.figure(figsize=(20, 20))
plt.imshow(WrittenOne)
plt.show()
```



Problem 1

0.2 Problem 1: Perspective Projection and Homogenous Coordinates [10 pts]

0.2.1 Part 1 [3 pts]

Consider a perspective projection where a point

$$P = [x\ y\ z]^T$$

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

$$Q1 = [4\ {-1}\ 3] + t[5\ 3\ 3]$$

$$Q2 = [1\ {-3}\ 2] + t[5\ 3\ 3]$$

where $-\infty \leq t \leq -2$.

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

intersection given by:

$(f' \frac{x}{z}, f' \frac{y}{z}, f'),$

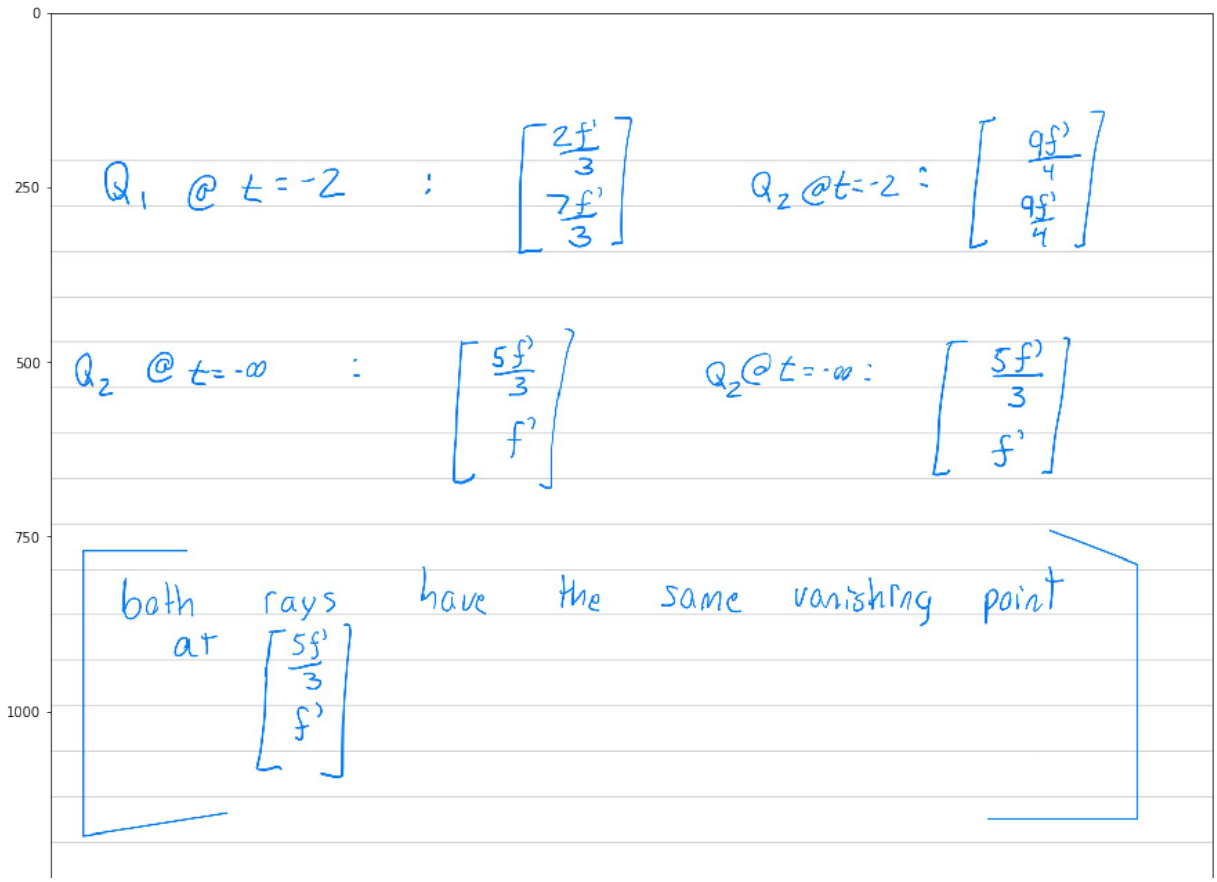
but only care about coordinates in image plane, so endpoint of ray in image plane P' is

$(f' \frac{x}{z}, f' \frac{y}{z})$

ray Q_1 is $Q_1 = \begin{bmatrix} 4 \\ -1 \\ 3 \end{bmatrix} + t \begin{bmatrix} 5 \\ 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 4+5t \\ -1+3t \\ 3+3t \end{bmatrix}$

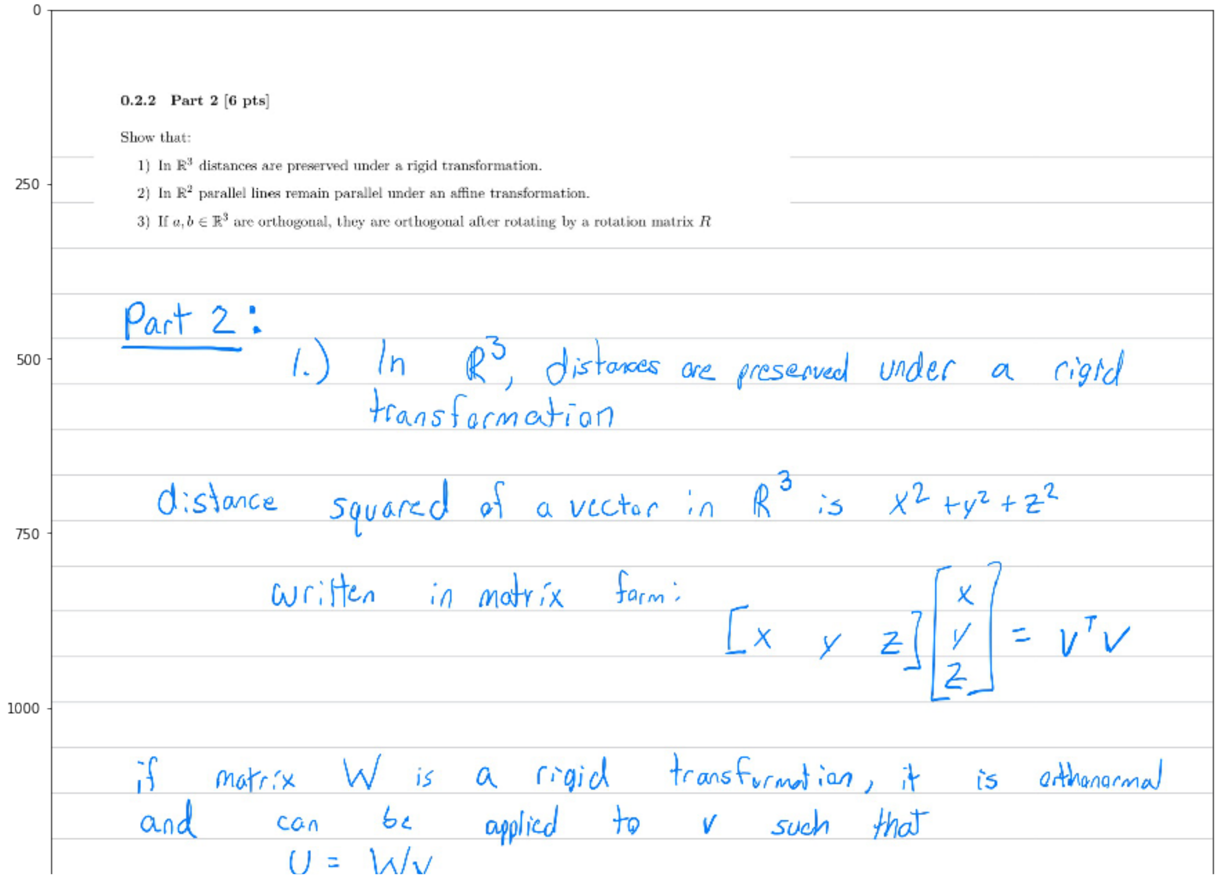
In [113]:

```
#Problem 1 Part 1.2
WrittenTwo = plt.imread('CSE 252A_HW1_2.jpg')
plt.figure(figsize=(20, 20))
plt.imshow(WrittenTwo)
plt.show()
```



In [114]:

```python
#Problem 1 Part 2
WrittenThree = plt.imread('CSE 252A_HW1_3.jpg')
plt.figure(figsize=(20, 20))
plt.imshow(WrittenThree)
plt.show()
```



0.2.2 Part 2 [6 pts]

Show that:

1) In $\mathbb{R}^3$ distances are preserved under a rigid transformation.
2) In $\mathbb{R}^2$ parallel lines remain parallel under an affine transformation.
3) If $a, b \in \mathbb{R}^3$ are orthogonal, they are orthogonal after rotating by a rotation matrix R

Part 2:

1.) In $\mathbb{R}^3$, distances are preserved under a rigid transformation

distance squared of a vector in $\mathbb{R}^3$ is $x^2 + y^2 + z^2$

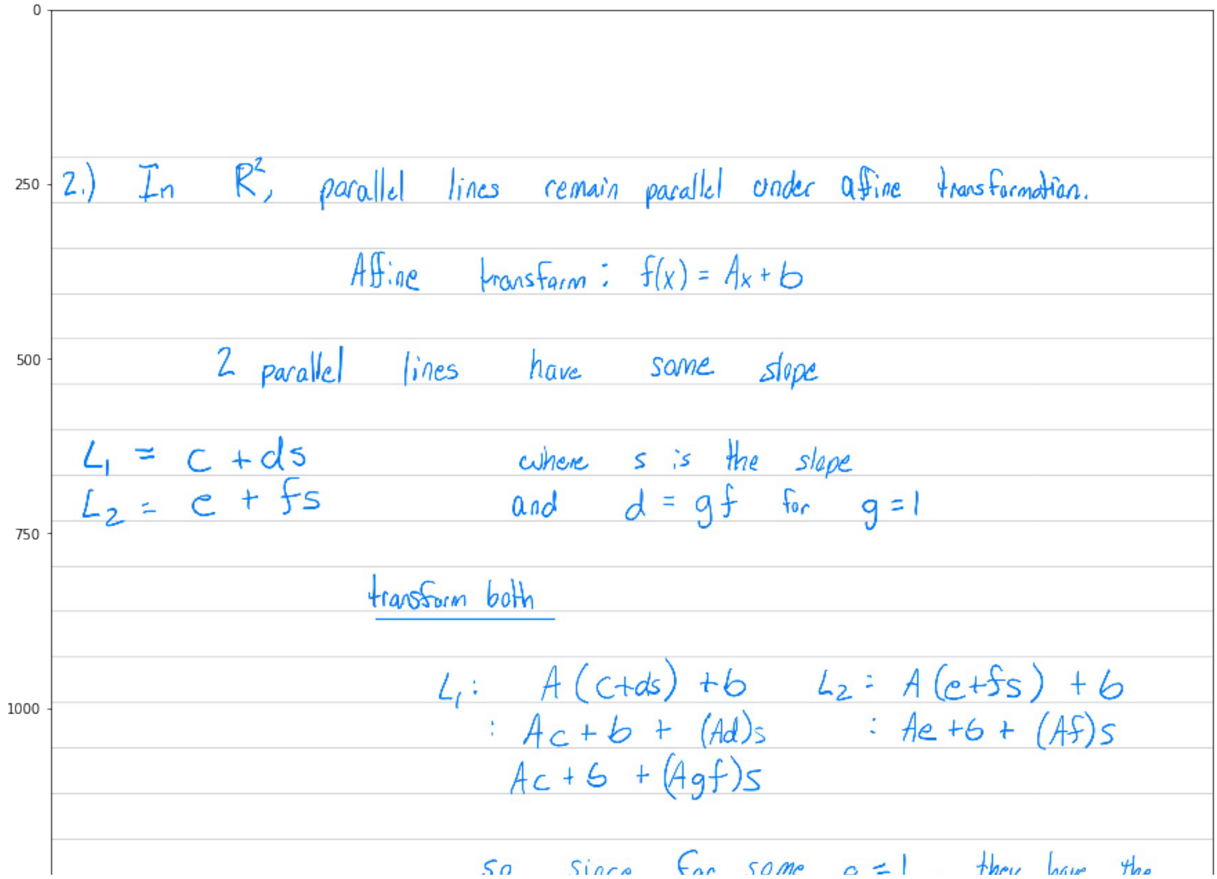
written in matrix form: $\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = v^T v$

if matrix W is a rigid transformation, it is orthonormal and can be applied to v such that

$U = Wv$

In [115]:

```
WrittenFour = plt.imread('CSE 252A_HW1_4.jpg')
plt.figure(figsize=(20, 20))
plt.imshow(WrittenFour)
plt.show()
```



In [116]:

```
#Problem 1 Part 3
import numpy as np

A = np.array([[1,0,-1],
              [0,1, 0],
              [0,0, 1]]) # Fill this in
X = np.array([[0,1,1,0], # Fill this in
              [0,0,1,1],
              [0,1,0,1]])
print("A\n{}\nX\n{}\nAX\n{}".format(A, X, A@X))
```

```
A
[[ 1  0 -1]
 [ 0  1  0]
 [ 0  0  1]]
X
[[0 1 1 0]
 [0 0 1 1]
 [0 1 0 1]]
AX
[[ 0  0  1 -1]
 [ 0  0  1  1]
 [ 0  1  0  1]]
```

Problem 2: Image Formation and Rigid Body Transformations [10 points]

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

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

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

,

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

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

,

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

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

,

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

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

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

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

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

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

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

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

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

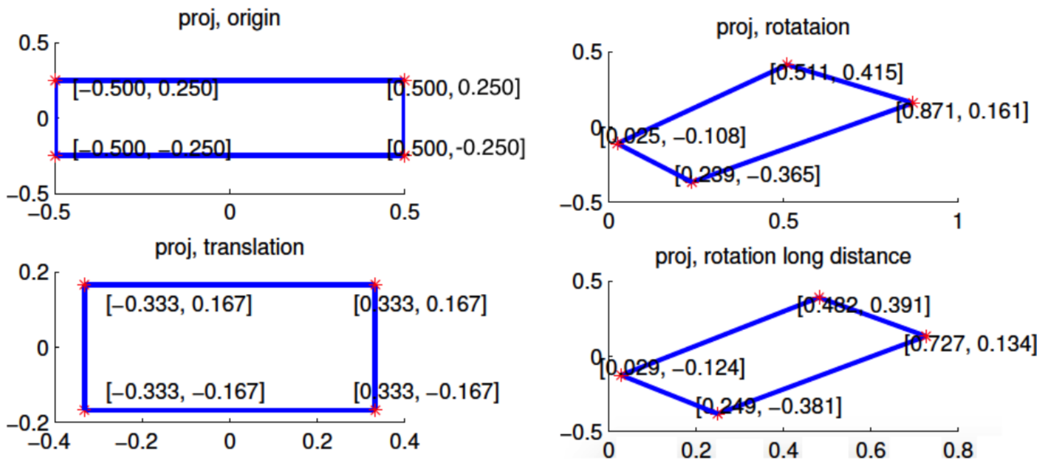
We will consider four different settings of focal length, viewing angles and camera positions below. For each of these calculate:

a) Extrinsic transformation matrix,

b) Intrinsic camera matrix under the perspective camera assumption.

c) 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]. $^{B}O_{A} = [1\ \text{-}1\ 1]^T\ {}^{B}O_{A} = [1\ \text{-}1\ 1]^T$. Focal length = 1. The optical axis of the camera is aligned with the z-axis.
3. [Translation and Rotation]. Focal length = 1. $^{B}_{A}R\,^{B}_{A}R$ encodes a 45 degree rotation around the y-axis followed by a 20 degree rotation around the x-axis. $^{B}O_{A} = [\text{-}1\ 0\ 1]^T\ {}^{B}O_{A} = [\text{-}1\ 0\ 1]^T$.
4. [Translation and Rotation, long distance]. Focal length = 7. $^{B}_{A}R\,^{B}_{A}R$ encodes a 45 degree rotation around the y-axis followed by a 20 degree rotation around the x-axis. $^{B}O_{A} = [\text{-}1\ \text{-}1\ 21]^T\ {}^{B}O_{A} = [\text{-}1\ \text{-}1\ 21]^T$.

> You can refer the Richard Szeliski starting page 36 for image formation and the extrinsic matrix.

> Intrinsic matrix calculation for perspective camera models was covered in class and can be found in lecture 3 https://canvas.ucsd.edu/courses/18894/files/folder/lectures (https://canvas.ucsd.edu/courses/18894/files/folder/lectures)

> You can also refer lecture 2 of the previous year's course as well for further information if you wish! https://cseweb.ucsd.edu/classes/fa19/cse252A-a/lec2.pdf (https://cseweb.ucsd.edu/classes/fa19/cse252A-a/lec2.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 ff, the focal length. In other words: the only parameter in the intrinsic camera matrix under the perspective assumption is ff.

In [2]:

```
import numpy as np
import matplotlib.pyplot as plt
import math


# convert points from euclidian to homogeneous
def to_homog(points):
    # write your code here
    check = np.shape(points)
    homog = np.ones((check[0]+1, check[1]))
    homog[0:check[0], 0:check[1]] = points[:,:]
    return(homog)


# convert points from homogeneous to euclidian
def from_homog(points_homog):
    # write your code here
    check = np.shape(points_homog)
    last_row = points_homog[check[0]-1, :]
    euclid = points_homog/last_row[None,:]
    euclid = euclid[0:check[0]-1, :]
    return(euclid)



# project 3D euclidian points to 2D euclidian
def project_points(P_int, P_ext, pts):
    # write your code here
    pts_homog = to_homog(pts)
    pts_final = np.matmul( P_int, np.matmul(P_ext, pts_homog))

    return from_homog(pts_final)
```

In [3]:

```
# Change the three matrices for the four cases as described in the
# in the four camera functions geiven below. Make sure that we can
```

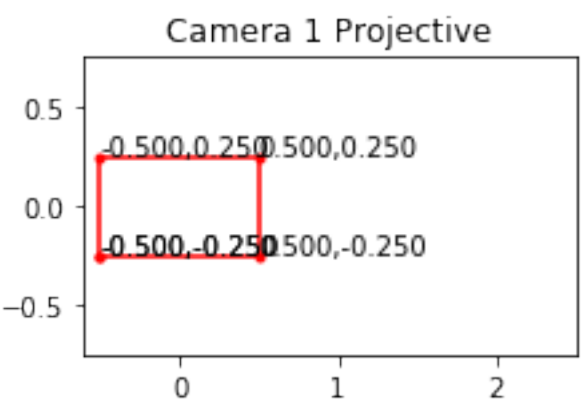
```
# (if one exists) being used to fill in the matrices. Feel free to
# comments any thing you feel the need to explain.

def camera1():
    # write your code here
    '''
        Use transformation matrix as below
        [R t
         0 1]
        where R is 3x3 rotation matrix, t is 3x1 transformation mat

        no rotation so R is 3x3 identity
        no translation so t is [0,0,0].T
    '''
    P_ext = np.array([[1, 0, 0, 0],
                      [0, 1, 0, 0],
                      [0, 0, 1, 0],
                      [0, 0, 0, 1]])

    '''
        intrinsic is given by: ([[f, 0, 0, 0],
                                 [0, f, 0, 0],
                                 [0, 0, 1, 0]])
        here f is focal length = 1
    '''
    f=1
    P_int_proj = np.array([[f, 0, 0, 0],
                           [0, f, 0, 0],
                           [0, 0, 1, 0]])
    return P_int_proj, P_ext

def camera2():
    # write your code here
    '''
        Use transformation matrix as below
        [R t
         0 1]
        where R is 3x3 rotation matrix, t is 3x1 transformation mat

        no rotation so R is 3x3 identity
        translation is [1,-1,1].T, however since it is stated that
        the only way this is possible is if there is only z-axis tr
    '''
    P_ext = np.array([[1, 0, 0, 0],
                      [0, 1, 0, 0],
```

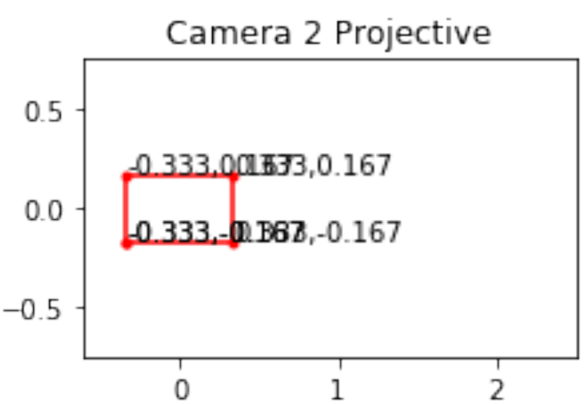
```python
                    [0, 0, 1, 1],
                    [0, 0, 0, 1]])

    '''
        intrinsic is given by: ([[f, 0, 0, 0],
                                 [0, f, 0, 0],
                                 [0, 0, 1, 0]])
        here f is focal length = 1
    '''
    f=1
    P_int_proj = np.array([[f, 0, 0, 0],
                           [0, f, 0, 0],
                           [0, 0, 1, 0]])
    return P_int_proj, P_ext

def camera3():
    # write your code here
    '''
        Use transformation matrix as below
        [R t
         0 1]
        where R is 3x3 rotation matrix, t is 3x1 transformation mat

        rotation
        about x-axis: Rx = ([[1, 0, 0],
                             [0, cos(@), -sin(@)],
                             [0, sin(@), cos(@)]]), where @ is angl

        about y-axis: Ry = ([[cos(@), 0, sin(@)],
                             [0, 1, 0],
                             [-sin(@), 0, cos(@)]]) where @ is angl

        rotation is R = Rx*Ry (matrix mult), because the y rotation

        translation is [-1,0,1].T
    '''
    cos20 = np.cos(np.deg2rad(20))
    sin20 = np.sin(np.deg2rad(20))
    cos45 = np.cos(np.deg2rad(45))
    sin45 = np.sin(np.deg2rad(45))

    Rx = np.array([[1, 0, 0],
                   [0, cos20, -sin20],
                   [0, sin20, cos20]])
    Ry = np.array([[cos45, 0, sin45],
```

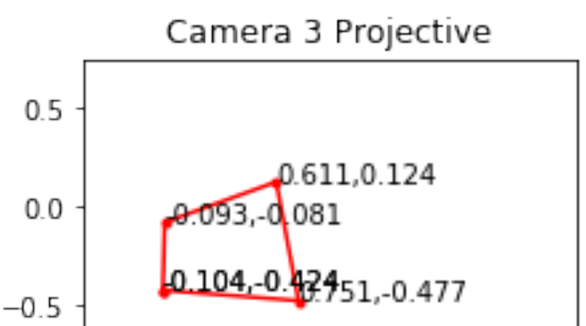
```
                    [0, 1, 0],
                    [-sin45, 0, cos45]])

    R = np.matmul(Rx, Ry)
    bOa = np.array([-1, 0, 1]).T
    P_ext = np.zeros((4,4))
    P_ext[0:3,0:3] = R
    P_ext[0:3, 3] = bOa
    P_ext[3,3] = 1

    '''
        intrinsic is given by: ([[f, 0, 0, 0],
                                 [0, f, 0, 0],
                                 [0, 0, 1, 0]])
        here f is focal length = 1
    '''
    f=1
    P_int_proj = np.array([[f, 0, 0, 0],
                           [0, f, 0, 0],
                           [0, 0, 1, 0]])
    return P_int_proj, P_ext

def camera4():
    # write your code here
    '''
        Use transformation matrix as below
        [R t
         0 1]
        where R is 3x3 rotation matrix, t is 3x1 transformation mat

        rotation
        about x-axis: Rx = ([[1, 0, 0],
                             [0, cos(@), -sin(@)],
                             [0, sin(@), cos(@)]]), where @ is angl

        about y-axis: Ry = ([[cos(@), 0, sin(@)],
                             [0, 1, 0],
                             [-sin(@), 0, cos(@)]]) where @ is angl

        rotation is R = Rx*Ry (matrix mult), because the y rotation

        translation is [-1,0,1].T
    '''
    cos20 = np.cos(np.deg2rad(20))
    sin20 = np.sin(np.deg2rad(20))
```

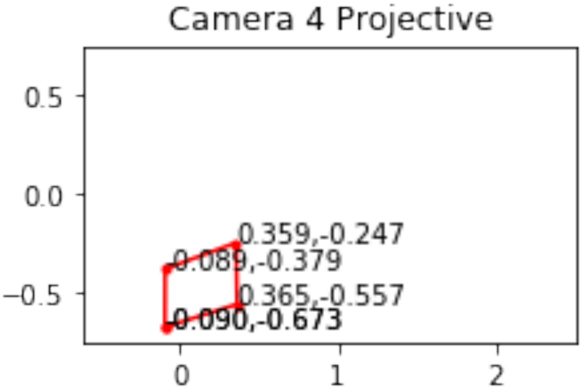
```
    cos45 = np.cos(np.deg2rad(45))
    sin45 = np.sin(np.deg2rad(45))

    Rx = np.array([[1, 0, 0],
                   [0, cos20, -sin20],
                   [0, sin20, cos20]])
    Ry = np.array([[cos45, 0, sin45],
                    [0, 1, 0],
                    [-sin45, 0, cos45]])

    R = np.matmul(Rx, Ry)
    bOa = np.array([-1, -1, 21]).T

    P_ext = np.zeros((4,4))
    P_ext[0:3,0:3] = R
    P_ext[0:3, 3] = bOa
    P_ext[3,3] = 1

    '''
        intrinsic is given by: ([[f, 0, 0, 0],
                                 [0, f, 0, 0],
                                 [0, 0, 1, 0]])
        here f is focal length = 1
    '''
    f=7
    P_int_proj = np.array([[f, 0, 0, 0],
                           [0, f, 0, 0],
                           [0, 0, 1, 0]])
    return P_int_proj, 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]))+","+s

    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_ext = camera()
        plt.subplot(2, 2, i+1)
        plot_points(project_points(P_int_proj, P_ext, points), titl
        plt.show()

main()
```









Problem 3: Homography [12 pts]

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 C_1 and C_2 C_2 looking at a plane π π in the world. Consider a point P P on the plane π π and its projection $p = [\mathrm{u1}, \mathrm{v1}, 1]^T$ $p = [\mathrm{u1}, \mathrm{v1}, 1]^T$ in the image 1 and $q = [\mathrm{u2}, \mathrm{v2}, 1]^T$ $q = [\mathrm{u2}, \mathrm{v2}, 1]^T$ in image 2.

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

$$q \approx Hp$$

$$q \approx Hp$$

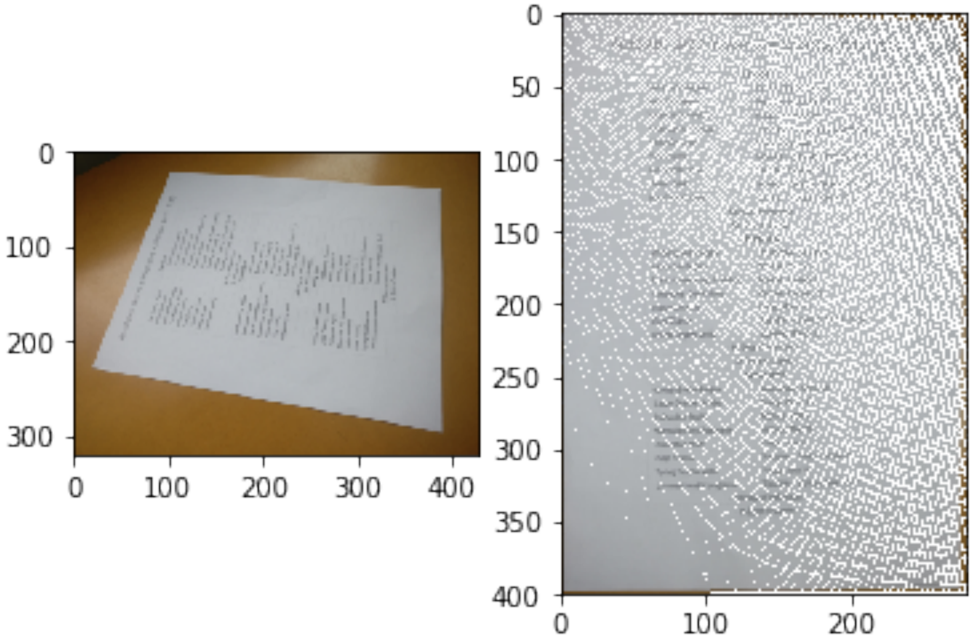
Here $\approx$ $\approx$ denotes equality in homogeneous coordinates, meaning that the left and right hand sides are proportional. Note that H H only depends on the plane and the projection matrices of the two cameras.

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

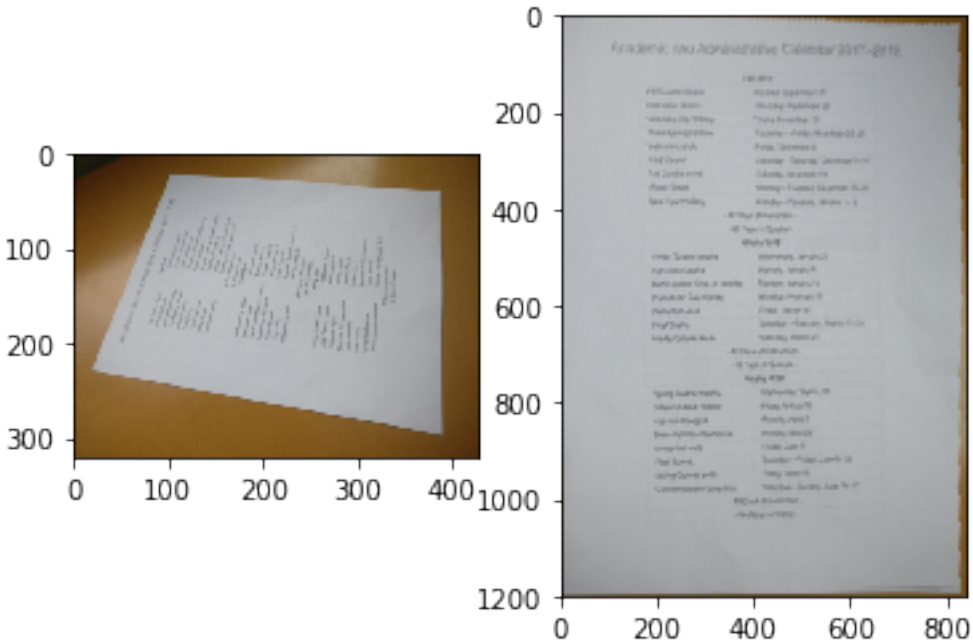
In this problem, complete the code for computeH and warp functions that can be used in the skeletal code that follows.

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 an image of an image being imaged. The goal will be to map the given image onto both the portrait as well as the screen of the tablet imaging the portrait. This will require the computation of two homographies.

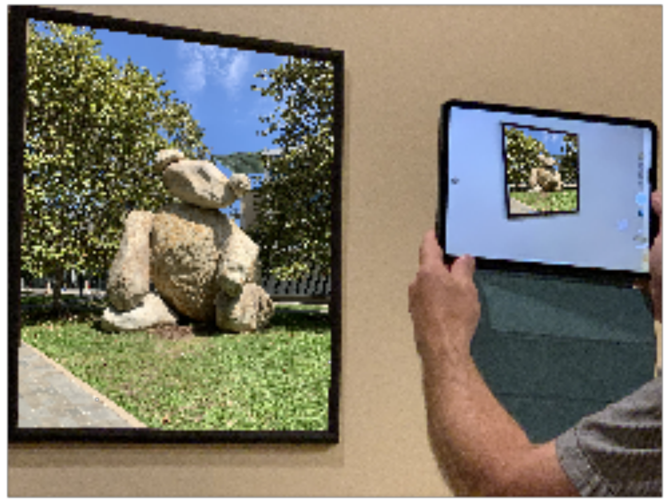
1.



2.



3.



1. 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.
2. The target image can start out as an empty matrix that you fill out using your code.
3. You will have to implement the computeH function that computes a homography. It takes in exactly four point correspondences between the source image and target image in homogeneous coordinates respectively and returns a 3 × × 3 homography matrix.
4. You will also have to implement the three warp functions in the skeleton code given and plot the resultant image pairs. For plotting the results of warps 1 and 2, make sure that the target image is not smaller than the source image.

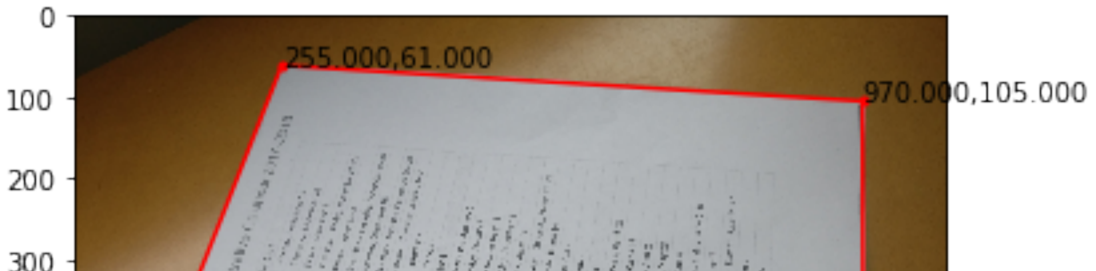
Note: We have provided test code to check if your implementation for computeH is correct. All the code to plot the results needed is also provided along with the code to read in the images and other data required for this problem. Please try not to modify that code.

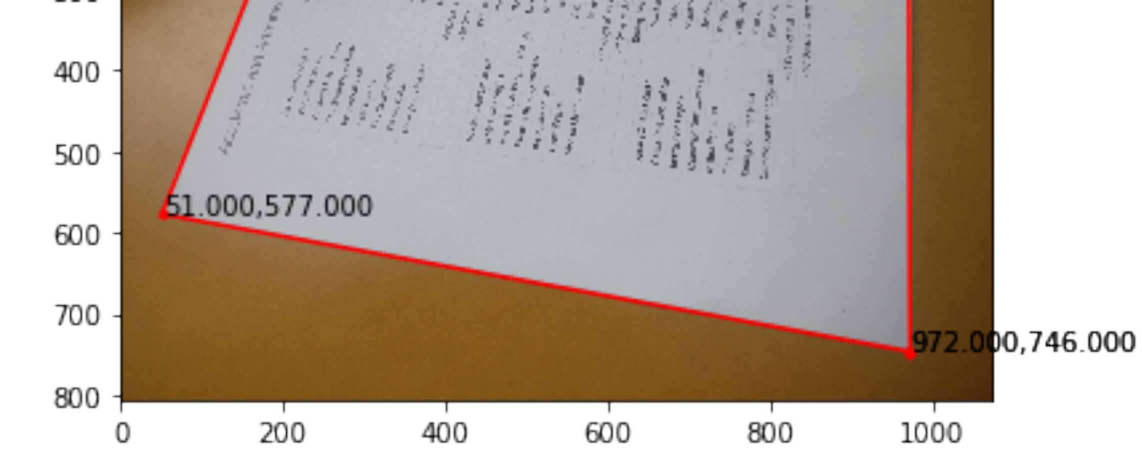
You may find following python built-ins helpful: numpy.linalg.svd, numpy.meshgrid

In [99]:

```
import numpy as np
from PIL import Image

import matplotlib.pyplot as plt
```

```
# 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
# you can, if you wish to, make the image larger (or till your comp

source_image = np.array(Image.open("photo.jpg"))/255





#code to easily downsize without having to change any parameters ot
from skimage.transform import rescale
import cv2
downSize = 2
source_image = rescale(source_image, 1/downSize, anti_aliasing=Fals








# display images
plt.imshow(source_image)

# Align the polygon such that the corners align with the document i
# This polygon doesn't need to overlap with the edges perfectly, an
# The order of points is clockwise, starting from bottom left.
y_coords = [int(1493/downSize),int(1154/downSize),int(122/downSize)
x_coords = [int(1944/downSize),int(102/downSize),int(511/downSize),

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

plt.show()
print (source_image.shape)
```





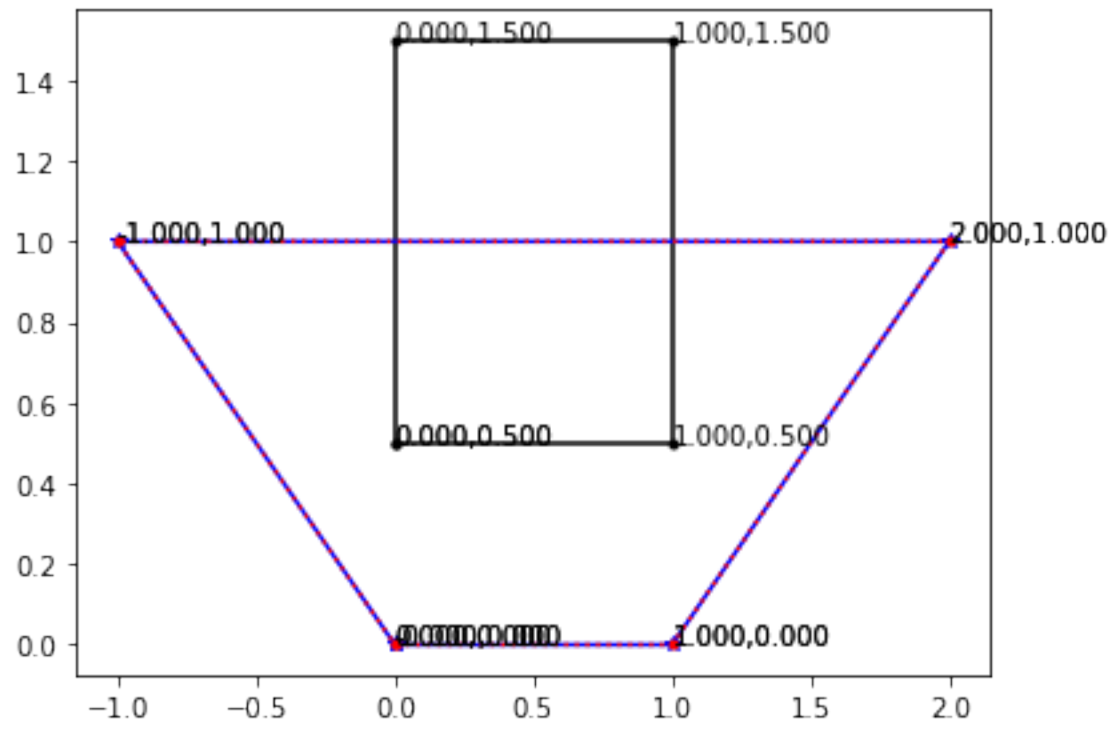

```
(806, 1074, 3)
```

In [100]:

```python
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 h
    # make sure points are 3D homogeneous
    assert source_points.shape[0]==3 and target_points.shape[0]==3
    #Your code goes here

    x1S,x2S,x3S,x4S = source_points[0,0],source_points[0,1],source_
    y1S,y2S,y3S,y4S = source_points[1,0],source_points[1,1],source_
    #source points ^^
    x1T,x2T,x3T,x4T = target_points[0,0],target_points[0,1],target_
    y1T,y2T,y3T,y4T = target_points[1,0],target_points[1,1],target_
    #target points ^^
    A = np.array([[0, 0, 0, -x1S, -y1S, -1, y1T*x1S, y1T*y1S, y1T],
                  [x1S, y1S, 1, 0, 0, 0, -x1T*x1S, -x1T*y1S, -x1T
                  [0, 0, 0, -x2S, -y2S, -1, y2T*x2S, y2T*y2S, y2T
                  [x2S, y2S, 1, 0, 0, 0, -x2T*x2S, -x2T*y2S, -x2T
                  [0, 0, 0, -x3S, -y3S, -1, y3T*x3S, y3T*y3S, y3T
                  [x3S, y3S, 1, 0, 0, 0, -x3T*x3S, -x3T*y3S, -x3T
                  [0, 0, 0, -x4S, -y4S, -1, y4T*x4S, y4T*y4S, y4T
                  [x4S, y4S, 1, 0, 0, 0, -x4T*x4S, -x4T*y4S, -x4T

    u, s, vh = np.linalg.svd(A, full_matrices=True)
    vh = np.transpose(vh)

    H_mtx = np.zeros((3,3)) #Fill in the H_mtx with appropriate val
    H_mtx[0,0:3] = vh[0:3,8]
    H_mtx[1,0:3] = vh[3:6,8]
    H_mtx[2,0:3] = vh[6:9,8]
```

```
    return  H_mtx
#######################################################
# 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))
    print (from_homog(np.matmul(H,to_homog(source_points[:,1].resha
    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 Comput
test_computeH()
```

```
[[1.00000000e+00]
 [4.41370382e-16]]
```

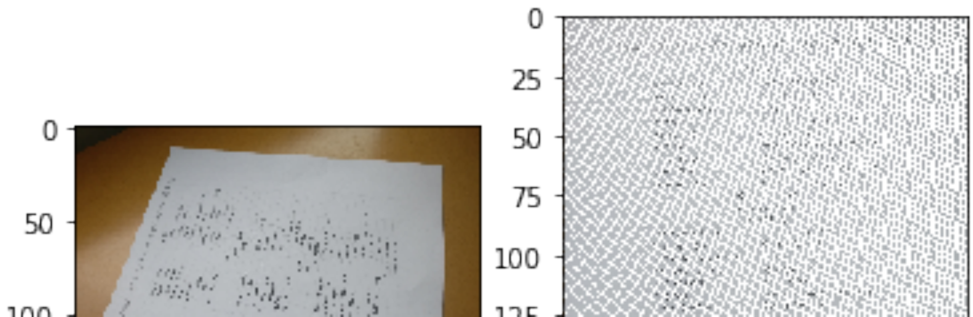


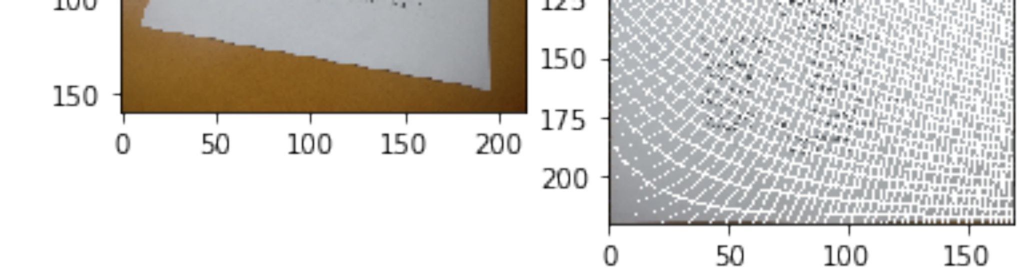

```
The red and blue quadrilaterals should overlap if Comp
uteH is implemented correctly.
```

In [6]:

```
def warp(source_img, source_points, target_size):
    # Create a target image and select target points to create a ho
    # in other words map all source points to target points and the
```

```
    # a warped version of the image based on the homography by fill
    # Make sure the new image (of size target_size) has the same nu
    assert target_size[2]==source_img.shape[2]
    #Your code goes here

    targetSizeX, targetSizeY = target_size[0], target_size[1]
    target_img = np.ones((targetSizeX,targetSizeY,3))
    target_points = np.array([[0,targetSizeX-1],[0,0],[targetSizeY-
    H_mtx = computeH(to_homog(source_points), to_homog(target_point


    for i in range(0, source_img.shape[1]):
        #print(i)
        for j in range(0, source_img.shape[0]):
            tempCoords = to_homog(np.array([[i,j]]).T)
            #tempCoords = to_homog(np.array([[194,21]]).T)
            targetCoords = np.matmul(H_mtx, tempCoords)
            targetCoords = from_homog(targetCoords)
            targetCoords = np.rint(targetCoords)
            xTarget = int(targetCoords[0])
            yTarget = int(targetCoords[1])
            if(yTarget >= 0 and yTarget < targetSizeX and xTarget >
                target_img[yTarget, xTarget, :] = source_img[j, i,
            #target_img[xTarget, yTarget, :] = source_img[i, j, :]


    return target_img

# Use the code below to plot your result
result = warp(source_image, source_points, (int(2200/downSize),int(

plt.subplot(1, 2, 1)
plt.imshow(source_image)
plt.subplot(1, 2, 2)
plt.imsave("myop.png",result)
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?

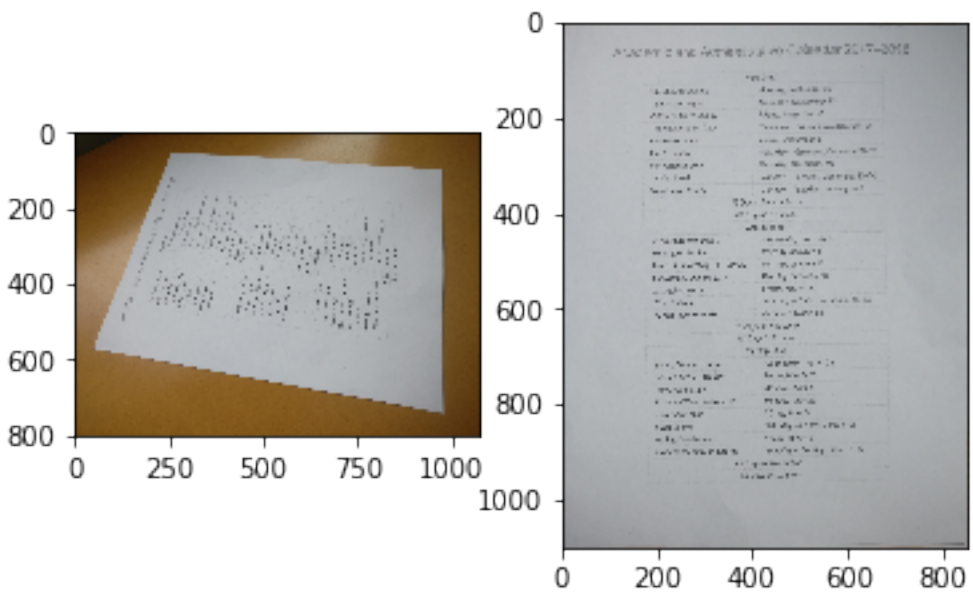
Answer: This is happening because the result starts out as a matrix of all zeros (in my case), and if a pixel isn't filled in, it will remain a zero. Pixels end up not being filled in because the result image is larger than the initial image so not every pixel in the result image can be mapped to a pixel with color data in the initial image. The noisy qualitatively can be thought of as tears that would appear if a peice of cloth the size of the desired region in the original image was stretched to the dimensions of the result.

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

In [101]:

```
def warp2(source_img, source_points, target_size):
    # Create a target image and select target points to create a ho
    # in other words map each target point to a source point, and t
    # of the image based on the homography by filling in the target
    # Make sure the new image (of size target_size) has the same nu

    #Your code goes here
    targetSizeX, targetSizeY = target_size[0], target_size[1]
    target_img = np.zeros((targetSizeX,targetSizeY,3))
    target_points = np.array([[0,targetSizeX-1],[0,0],[targetSizeY-
    H_mtx = computeH(to_homog(target_points), to_homog(source_point

    for i in range(0, targetSizeY):
        #print(i)
        for j in range(0, targetSizeX):
            tempCoords = to_homog(np.array([[i,j]]).T)
            sourceCoords = np.matmul(H_mtx, tempCoords)
            sourceCoords = from_homog(sourceCoords)
            sourceCoords = np.rint(sourceCoords)
            xSource = int(sourceCoords[0])
```

```
            xSource = int(sourceCoords[0])
            ySource = int(sourceCoords[1])
            #if(yTarget >= 0 and yTarget < targetSizeX and xTarget 
            target_img[j, i, :] = source_img[ySource, xSource, :]
            #source_img[ySource, xSource, :] = np.array([[0, 0, 0]]

            #Bicubic interpolation (optional, not used as it crashe
                                        #when attempting high quali
            #target_img = cv2.resize(target_img,None, fx = 2, fy = 

    return target_img

# Use the code below to plot your result
result = warp2(source_image, source_points, (int(2200/downSize),int

plt.subplot(1, 2, 1)
plt.imshow(source_image)
plt.subplot(1, 2, 2)
plt.imshow(result)
plt.imsave("warp2.png",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.

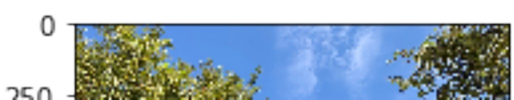
Answer: Warp1 gives the best results in a smaller target image, since more of its pixels are filled in with pixels from the source image; as the target image gets larger, a smaller and smaller percentage of its pixels are filled in, resulting in a lower quality image with more noise. Warp 2 gives the good results in an image roughly the same dimensions (but rotated 90 degrees and slightly larger) than the source image. This is because warp 2 guarentees all the pixels in the target image are filled in. The result was a very good image, and interpolation isn't really necessary to make it better. (see the saved warp2.png that was outputted by the code into the folder). Interpolation (bicubic, very resouce heavy) also crashed the kernel every time it was implemented on anything over than a very downsized image, leading this student to conclude the best approach, if maximum quality was desired, is to use a non-downsized image for warp2, as the result was better than a downsized image that was bicubic-ly interpolated, since the downsizing had to be significant in order for the interpolation to not crash the kernel.

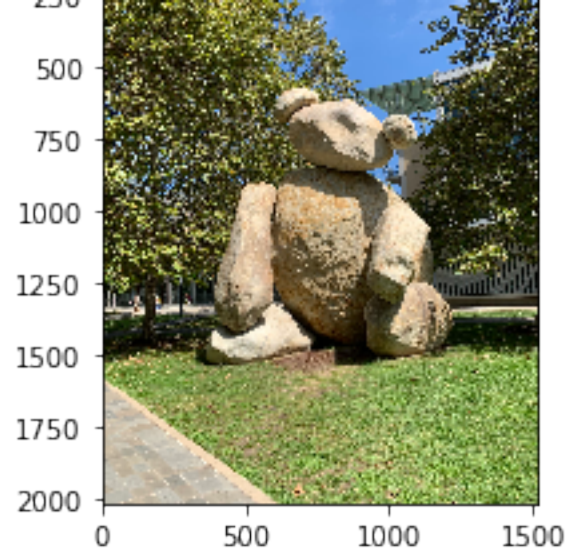
In warp3, you'll be replacing the portrait and image of the portrait in a provided image with another image. Read in "bear.png" as the source image, and "gallery.png" will serve as the target.

In [102]:

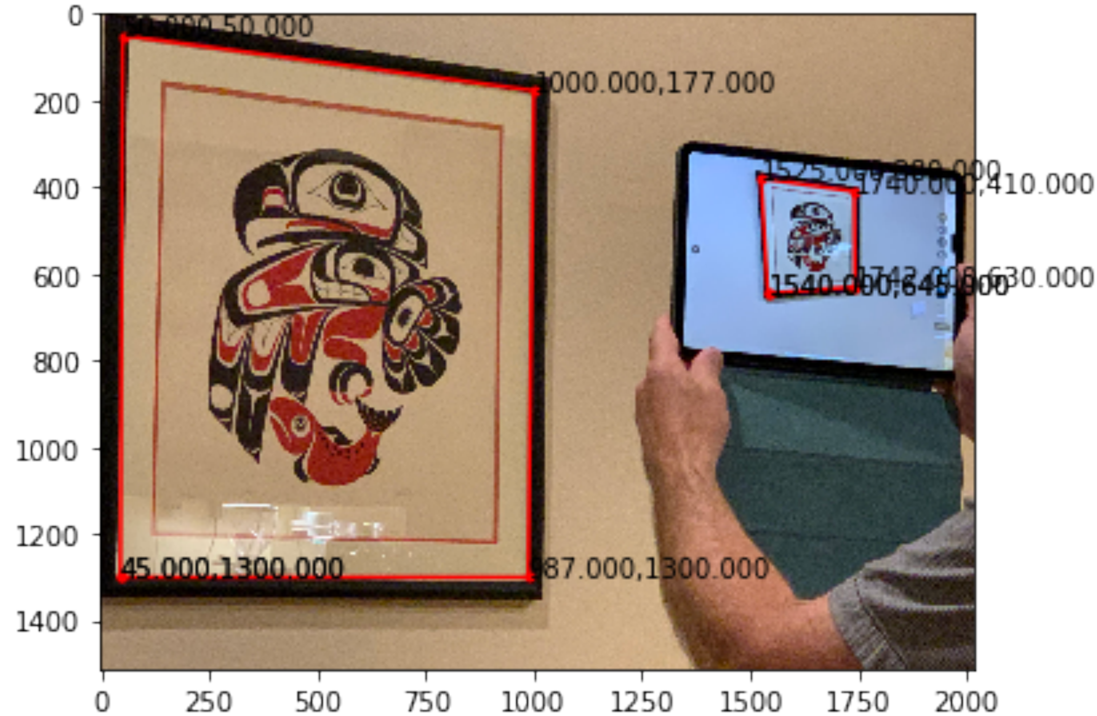
```
# Load the supplied source and target images here

source_image = np.array(Image.open("bear.png"))/255
target_image = np.array(Image.open("gallery.png"))/255




#code to easily downsize without having to change any parameters ot
from skimage.transform import rescale
import cv2
downSize = 2
source_image = rescale(source_image, 1/downSize, anti_aliasing=Fals
target_image = rescale(target_image, 1/downSize, anti_aliasing=Fals
```

```
# display images
plt.imshow(source_image)

plt.show()
print (source_image.shape)

# display images
plt.imshow(target_image)

# Align the polygon such that the corners align with the document i
# This polygon doesn't need to overlap with the edges perfectly, an
# The order of points is clockwise, starting from bottom left.
y_coords_frame = [int(2600/downSize),int(100/downSize),int(355/down
x_coords_frame = [int(90/downSize),int(100/downSize),int(2000/downS

y_coords_ipad = [int(1290/downSize),int(760/downSize),int(820/downS
x_coords_ipad = [int(3080/downSize),int(3050/downSize),int(3480/dow


# Plot points from the previous problem is used to draw over your i
# Note that your coordinates will change once you resize your image
target_frame_points = np.vstack((x_coords_frame, y_coords_frame))
plot_points(target_frame_points)

target_ipad_points = np.vstack((x_coords_ipad, y_coords_ipad))
plot_points(target_ipad_points)

plt.show()
print (target_image.shape)

target_points = np.vstack((x_coords_frame, y_coords_frame, x_coords_
```






```
(2016, 1512, 3)
```

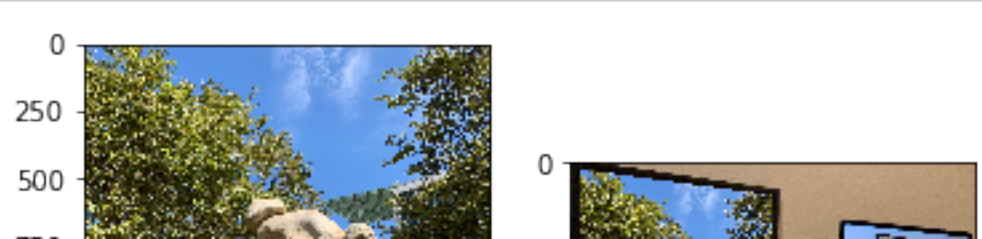


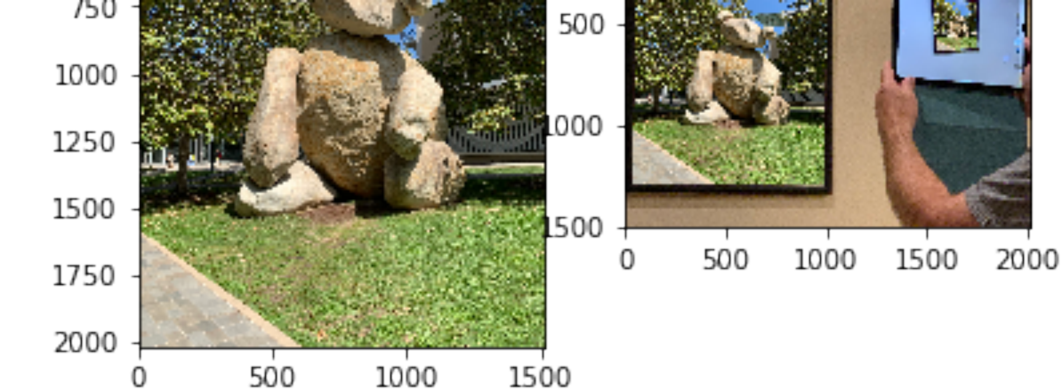

```
(1512, 2016, 3)
```

In [103]:

```
def warp3(target_image, target_points, source_image):
    #Your code goes here

    #First out the image into the Frame
    frame_target_points = target_points[0:2,0:4]
    source_pointsX, source_pointsY = np.shape(source_image)[0], np.
    source_points = np.array([[0,source_pointsX-1],[0,0],[source_po
    H_mtx = computeH(to_homog(source_points), to_homog(frame_target
```

```
    H_mtx = computeH(to_homog(source_points), to_homog(frame_target

    for i in range(0, source_pointsY):
        #print(i)
        for j in range(0, source_pointsX):
            tempCoords = to_homog(np.array([[i,j]]).T)
            targetCoords = np.matmul(H_mtx, tempCoords)
            targetCoords = from_homog(targetCoords)
            targetCoords = np.rint(targetCoords)
            xTarget = int(targetCoords[0])
            yTarget = int(targetCoords[1])
            target_image[yTarget, xTarget, :] = source_image[j, i,

    #Now put the image into the iPad
    frame_target_points = target_points[2:4,0:4]
    source_pointsX, source_pointsY = np.shape(source_image)[0], np.
    source_points = np.array([[0,source_pointsX-1],[0,0],[source_po
    H_mtx = computeH(to_homog(source_points), to_homog(frame_target

    for i in range(0, source_pointsY):
        #print(i)
        for j in range(0, source_pointsX):
            tempCoords = to_homog(np.array([[i,j]]).T)
            targetCoords = np.matmul(H_mtx, tempCoords)
            targetCoords = from_homog(targetCoords)
            targetCoords = np.rint(targetCoords)
            xTarget = int(targetCoords[0])
            yTarget = int(targetCoords[1])
            target_image[yTarget, xTarget, :] = source_image[j, i,
    return target_image


# Use the code below to plot your result
result1 = warp3(target_image, target_points, source_image)
plt.subplot(1, 2, 1)
plt.imshow(source_image)
plt.subplot(1, 2, 2)
plt.imshow(result1)
plt.imsave("warp3.png",result1)
plt.show()
```





Problem 4: Surface Rendering [18 pts]

In this portion of the assignment we will be exploring different methods of approximating local illumination of objects in a scene. This last section of the homework will be an exercise in rendering surfaces. Here, you need use the surface normals and the masks from the provided pickle files, with various light sources, different materials, and using a number of illumination models. For the sake of simplicity, multiple reflections of light rays, and occlusion of light rays due to object/scene can be ignored.

Data

The surface normals and masks are to be loaded from the respective pickle files. For comparison, you should display the rendering results for both normals calculated from the original image and the diffuse components. There are 2 images that we will be playing with--namely one of a sphere and the other of a pear.

Assume that the albedo map is uniform.

Lambertian Illumination

One of the simplest models available to render 3D objections with illumination is the Lambertian model. This model finds the apparent brightness to an observer using the direction of the light source $\mathbf{L}\mathbf{L}$ and the normal vector on the surface of the object $\mathbf{N}\mathbf{N}$. The brightness intensity at a given point on an object's surface, $\mathbf{I_d}\mathbf{I_d}$, with a single light source is found using the following relationship:

$$\mathbf{I_d} = \mathbf{L} \cdot \mathbf{N}(I_l \mathbf{C})$$

$$\mathbf{I_d} = \mathbf{L} \cdot \mathbf{N}(I_l \mathbf{C})$$

where, $\mathbf{C}$ and I_l are the the color and intensity of the light source respectively.

Phong Illumination

One major drawback of Lambertian illumination is that it only considers the diffuse light in its calculation of brightness intensity. One other major component to illumination rendering is the specular component. The specular reflectance is the component of light that is reflected in a single direction, as opposed to all directions, which is the case in diffuse reflectance. One of the most used models to compute surface brightness with specular components is the Phong illumination model. This model combines ambient lighting, diffused reflectance as well as specular reflectance to find the brightness on a surface. Phong shading also considers the material in the scene which is characterized by four values: the ambient reflection constant (k_a), the diffuse reflection constant (k_d), the specular reflection constant (k_s) and α the Phong constant, which is the 'shininess' of an object. Furthermore, since the specular component produces 'rays', only some of which would be observed by a single observer, the observer's viewing direction ($\mathbf{V}$) must also be known. For some scene with known material parameters with M light sources the light intensity $\mathbf{I}_{phong}$ on a surface with normal vector $\mathbf{N}$ seen from viewing direction $\mathbf{V}$ can be computed by:

$$\mathbf{I}_{phong} = k_a\mathbf{I}_a + \sum_{m\in M} \left\{k_d(\mathbf{L}_m \cdot \mathbf{N})\mathbf{I}_{m,d} + k_s(\mathbf{R}_m \cdot \mathbf{V})^{\alpha}\mathbf{I}_{m,s}\right\},$$

$$\mathbf{I}_{phong} = k_a\mathbf{I}_a + \sum_{m\in M} \left\{k_d(\mathbf{L}_m \cdot \mathbf{N})\mathbf{I}_{m,d} + k_s(\mathbf{R}_m \cdot \mathbf{V})^{\alpha}\mathbf{I}_{m,s}\right\},$$

$$\mathbf{R}_m = 2\mathbf{N}(\mathbf{L}_m \cdot \mathbf{N}) - \mathbf{L}_m,$$

$$\mathbf{R}_m = 2\mathbf{N}(\mathbf{L}_m \cdot \mathbf{N}) - \mathbf{L}_m,$$

where $\mathbf{I}_a$, is the color and intensity of the ambient lighting, $\mathbf{I}_{m,d}$ and $\mathbf{I}_{m,s}$ are the color values for the diffuse and specular light of the mth light source.

Rendering

Please complete the following:

1. Write the function `lambertian()` that calculates the Lambertian light intensity given the light direction $\mathbf{L}$ with color and intensity $\mathbf{C}$ and $I_l = 1$, and normal vector $\mathbf{N}$. Then use this function in a program that calculates and displays the specular sphere and the pear using each of the two lighting sources found in Table 1. *Note: You do not need to worry about material coefficients in this model.*
2. Write the function `phong()` that calculates the Phong light intensity given the material constants (k_a, k_d, k_s, α), $\mathbf{V} = (0, 0, 1)^\top$, $\mathbf{N}$ and some number of M light sources. Then use this function in a program that calculates and displays the specular sphere and the pear using each of the sets of coefficients found in Table 2 with each light source individually, and both light sources combined.

Hint: To avoid artifacts due to shadows, ensure that any negative intensities found are set to zero.

Table 1: Light Sources

m	**Location**	**Color (RGB)**
1	$(1, 1, 0)^\top$	$(0.75, 0.75, 0.5)$
2	$(\frac{1}{3}, -\frac{1}{3}, \frac{1}{2})^\top$	$(1, 1, 1)$

Table 2: Material Coefficients

Mat.	k_a	k_d	k_s	α
1	0	0.1	0.75	5
2	0	0.5	0.1	5
3	0	0.5	0.5	10

Part 1. Loading pickle files and plotting the normals [4 pts] (Sphere - 2pts, Pear - 2pts)

In this first part, you are required to work with 2 images, one of a sphere and the other one of a pear. The pickle file normals.pickle is a list consisting of 4 numpy matrices which are

1) Normal Vectors for the sphere with specularities removed (Diffuse component)
2) Normal Vector for the sphere
3) Normal Vectors for the pear with specularities removed (Diffuse component)
4) Normal vectors for the pear

Please load the normals and plot them using the function plot_normals which is provided.

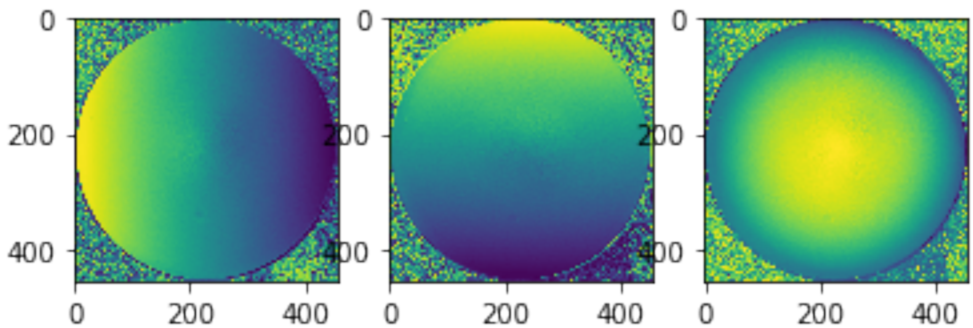
In [52]:

```
def plot_normals(diffuse_normals, original_normals):
    # Stride in the plot, you may want to adjust it to different im
    stride = 5

    normalss = diffuse_normals
    normalss1 = original_normals

    print("Normals:")
    print("Diffuse")
    # showing normals as three separate channels
    figure = plt.figure()
    ax1 = figure.add_subplot(131)
    ax1.imshow(normalss[..., 0])
    ax2 = figure.add_subplot(132)
    ax2.imshow(normalss[..., 1])
    ax3 = figure.add_subplot(133)
    ax3.imshow(normalss[..., 2])
    plt.show()
    print("Original")
    figure = plt.figure()
    ax1 = figure.add_subplot(131)
    ax1.imshow(normalss1[..., 0])
    ax2 = figure.add_subplot(132)
    ax2.imshow(normalss1[..., 1])
    ax3 = figure.add_subplot(133)
    ax3.imshow(normalss1[..., 2])
    plt.show()
```

In [54]:

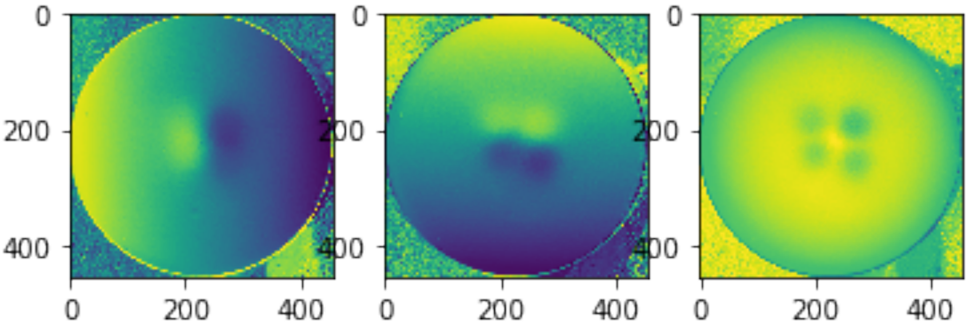
```
#Plot the normals for the sphere and pear for both the normal and d
#1 : Load the different normals
# LOAD HERE
import pickle
import numpy as np
import matplotlib.pyplot as plt
import math

file = open('normals.pkl', 'rb')
normals = pickle.load(file)
file.close()

sphereDiffuse = normals[0]
sphereND = normals[1]
pearDiffuse = normals[2]
pearND = normals[3]



#2 : Plot the normals using plot_normals
#What do you observe? What are the differences between the diffuse
''' The diffuse is smoother on the object itself, while the origina
surface on them, as a result of the specularities where the light c
part of the image to appear almost white, as discussed in class. It
worth noting that the background is different, with the diffuse ver
a speckled background and the originals having a more uniform backg
a result of background specularities removed and replaced resulting
'''
#PLOT HERE
plot_normals(sphereDiffuse, sphereND)
plot_normals(pearDiffuse, pearND)
```

Normals:
Diffuse

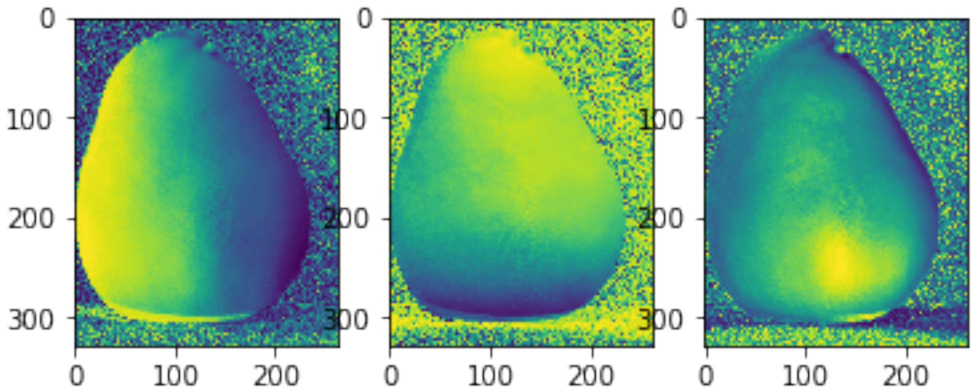


Original

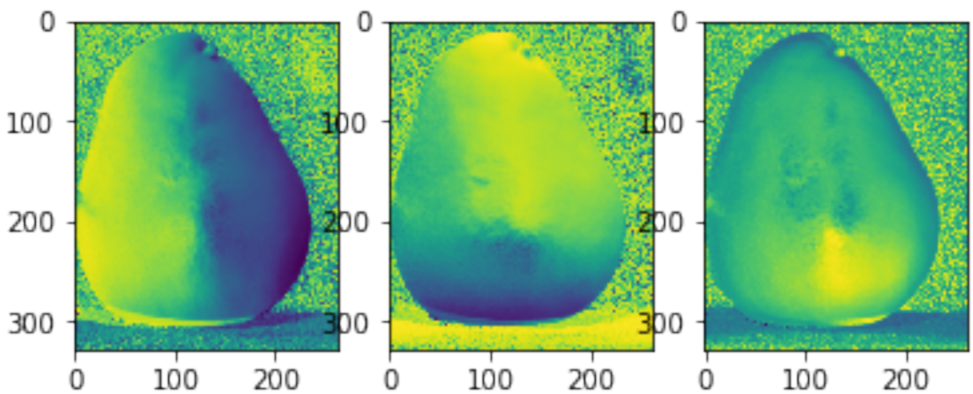
Original



Normals:
Diffuse



Original



Part 2. Lambertian model [6 pts]

Fill in your implementation for the rendered image using the lambertian model.

In [55]:

```
def normalize(img):
    assert img.shape[2] == 3
    maxi = img.max()
    mini = img.min()
    return (img - mini)/(maxi-mini)
```

In [74]:

```
def lambertian(normals, lights, color, intensity, mask):
    '''Your implementation'''
    image = np.ones((normals.shape[0], normals.shape[1], 3))
    lights = lights.T
    for i in range(0, normals.shape[0]):
        #print(i)
        for j in range(0, normals.shape[1]):
            #NC = np.matmul(normals[i,j,:].T, color)
            #Id = np.dot(lights, NC)
            nrml = np.array([normals[i,j,:]]).T
            LdotN = np.dot(lights, nrml)[0,0]
            Id = LdotN * color
            image[i,j,:] = Id[:,0]
    image[:,:,0] = image[:,:,0]*mask
    image[:,:,1] = image[:,:,1]*mask
    image[:,:,2] = image[:,:,2]*mask
    image = normalize(image)
    return (image)
```

Plot the rendered results for both the sphere and the pear for both the original and the diffuse components. Remember to first load the masks from the masks.pkl file. The masks.pkl file is a list consisting of 2 numpy arrays-

1)Mask for the sphere

2)Mask for the pear

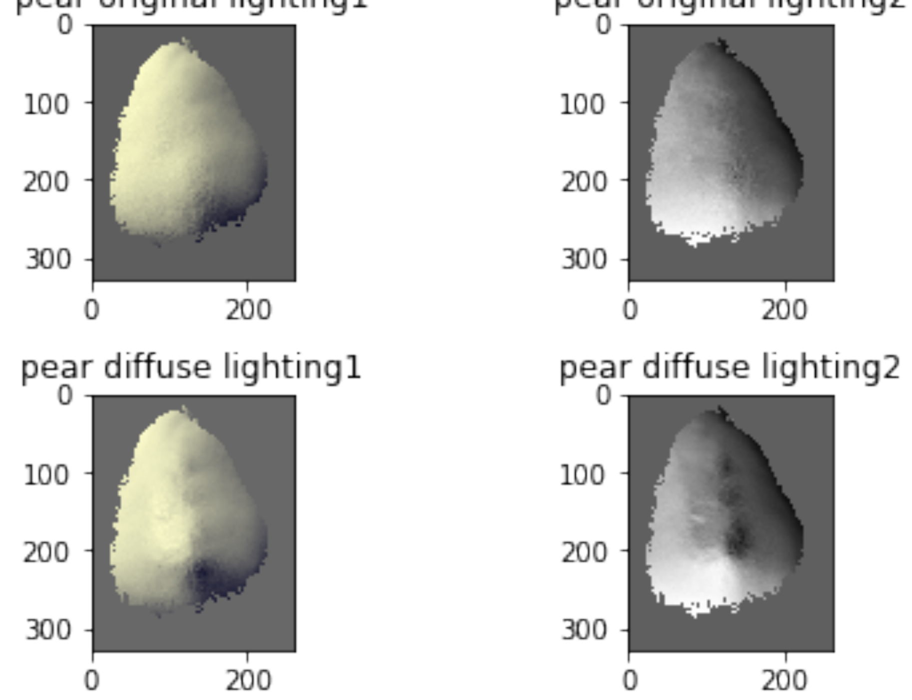
Remember to plot the normalized image using the function normalize which is provided.

In [75]:

```
# Load the masks for the sphere and pear
# LOAD HERE
file = open('masks.pkl', 'rb')
masks = pickle.load(file)
file.close()
```

```
file.close()

sphereMask = masks[0]
pearMask = masks[1]

# Output the rendering results for Pear
dirn1 = np.array([[1.0],[1.0],[0]])
color1 = np.array([[.75],[.75],[.5]])
dirn2 = np.array([[1.0/3],[-1.0/3],[1.0/2]])
color2 = np.array([[1],[1],[1]])

shapesName = ['pear', 'sphere']
settingsName = ['original', 'diffuse']
lightingName = ['lighting1', 'lighting2']


#Display the rendering results for pear for both diffuse and for bo
count = 1
i=0
if (i == 0): mask = pearMask
else: mask = sphereMask
for j in range(0,len(settingsName)):
    if (j == 0):
        if (i == 0): normals = pearDiffuse
        else: normals = sphereDiffuse
    if (j == 1):
        if (i == 0): normals = pearND
        else: normals = sphereND
    #print('j = ' + str(j))
    for k in range(0,len(lightingName)):
        if (k == 0):
            lights = dirn1
            color = color1
        else:
            lights = dirn2
            color = color2
        result = lambertian(normals, lights, color, 1, mask)

        plt.subplot(2, 2, count)
        plt.title(shapesName[i] + ' ' + settingsName[j] + ' ' + lig
        plt.imshow(result)
        count = count + 1
plt.tight_layout(pad=1.0)
plt.show()
```

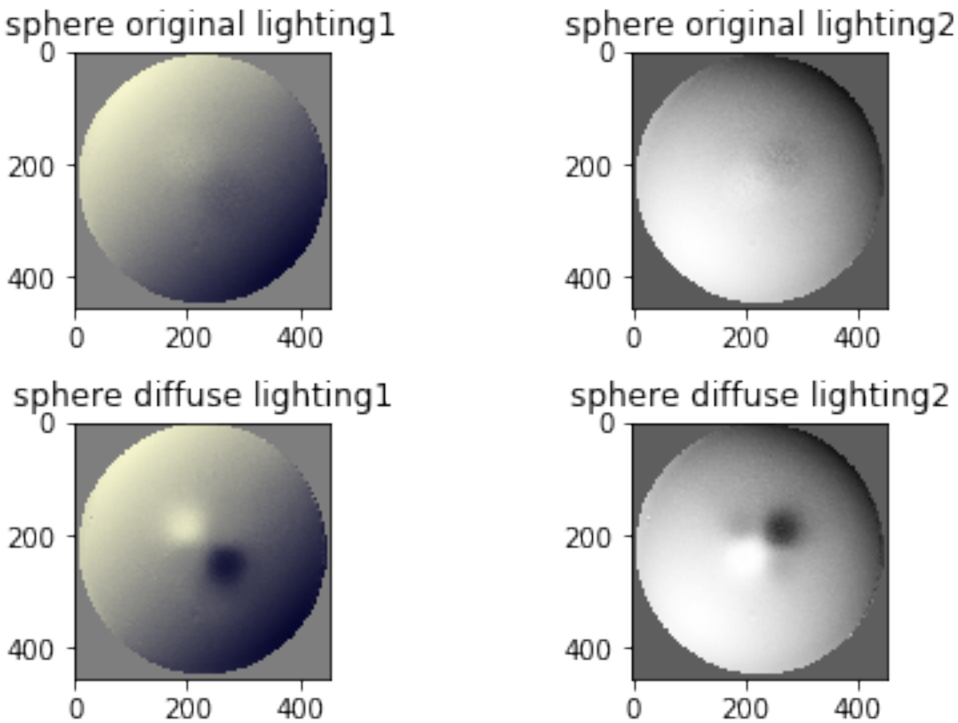
pear original lighting1 pear original lighting2



In [76]:

```
# Output the rendering results for Sphere
dirn1 = np.array([[1.0],[1.0],[0]])
color1 = np.array([[.75],[.75],[.5]])
dirn2 = np.array([[1.0/3],[-1.0/3],[1.0/2]])
color2 = np.array([[1],[1],[1]])

shapesName = ['pear', 'sphere']
settingsName = ['original', 'diffuse']
lightingName = ['lighting1', 'lighting2']

#Display the rendering results for sphere for both diffuse and for
count = 1
i = 1
if (i == 0): mask = pearMask
else: mask = sphereMask
for j in range(0,len(settingsName)):
    if (j == 0):
        if (i == 0): normals = pearDiffuse
        else: normals = sphereDiffuse
    if (j == 1):
        if (i == 0): normals = pearND
        else: normals = sphereND
    #print('j = ' + str(j))
    for k in range(0,len(lightingName)):
        if (k == 0):
            lights = dirn1
```

```
            color = color1
        else:
            lights = dirn2
            color = color2
        result = lambertian(normals, lights, color, 1, mask)

        plt.subplot(2, 2, count)
        plt.title(shapesName[i] + ' ' + settingsName[j] + ' ' + lig
        plt.imshow(result)
        count = count + 1
plt.tight_layout(pad=1.0)
plt.show()
```



Part 3. Phong model [8 pts]

Please fill in your implementation for the Phong model below.

In [77]:

```
def phong(normals, lights, color, material, view, mask):
    '''Your implementation'''
    image = np.ones((normals.shape[0], normals.shape[1], 3))

    numColors = np.shape(color)[1]  #find M
    ka = 0
    kd = material[0]
    ks = material[1]
    alpha = material[2]

    for i in range(0, normals.shape[0]):
        #print(i)
        for j in range(0, normals.shape[1]):
            Iphong = np.zeros((3,1))
            nrml = np.array([normals[i,j,:]]).T
            for k in range(0, numColors):
                L = np.zeros((1,3))
                L[0,:] = lights[:,k] #get the current light directi
                Im = np.zeros((3,1))
                Im[:,0] = color[:,k] #get the current light RGB col

                LdotN = np.matmul(L, nrml)[0,0]

                Rm = 2*nrml*LdotN - L.T

                firstTerm = kd*LdotN*Im
                secondTerm = ks*(np.matmul(Rm.T, view)**alpha)*Im
                SUM = firstTerm + secondTerm
                if (LdotN <= 0):
                    SUM = np.zeros((3,1))
                Iphong = np.add(Iphong, SUM)
            #now do the add to array thing
            image[i,j,:] = Iphong[:,0]
    image[:,:,0] = image[:,:,0]*mask
    image[:,:,1] = image[:,:,1]*mask
    image[:,:,2] = image[:,:,2]*mask
    image = normalize(image)
    return (image)
```

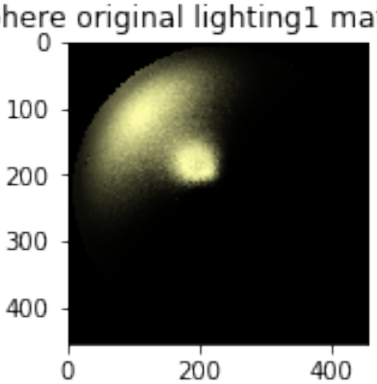
With the function completed, plot the rendering results for the sphere and pear (both diffuse and original compnents) for all the materials and light sources and also with the combination of both the light sources.

In [95]:

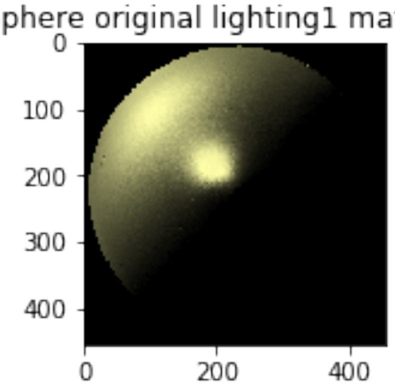
```
# Output the rendering results for sphere
view =  np.array([[0],[0],[1]])
material = np.array([[0.1,0.75,5],[0.5,0.1,5],[0.5,0.5,10]])
lightcol1 = np.array([[1,0.75],[1,0.75],[0,0.5]])
lightcol2 =  np.array([[1.0/3,1],[-1.0/3,1],[1.0/2,1]])
#Display rendered results for sphere for all materials and light so
material1 = material[0,:]
material2 = material[1,:]
material3 = material[2,:]
view = view
lights1 = np.array([lightcol1[:,0]]).T
color1 = np.array([lightcol1[:,1]]).T
lights2 = np.array([lightcol2[:,0]]).T
color2 = np.array([lightcol2[:,1]]).T
lights3 = np.concatenate((lights1, lights2), axis=1)
color3 = np.concatenate((color1, color2), axis=1)
#test = phong(sphereDiffuse, lights2, color2, material1, view, sphe
#plt.imshow(test)
#plt.show()

# 2 - sphere and pear
# 2 - original and deffuse
# 3 - material 1,2 and 3
# 3 - light 1, 2, and 1&2



settingsName = ['original', 'diffuse']
lightingName = ['lighting1', 'lighting2', 'lighting3']
materialName = ['material1', 'material2', 'material3']

##sphere
mask = sphereMask
count = 1
fig = plt.figure(figsize=(8, 20))

for j in range(0,len(settingsName)):
    if (j == 0): normals = sphereND
    if (j == 1): normals = sphereDiffuse
```

```python
    if (j == 1): normals = sphereDiffuse
    #print('j = ' + str(j))
    for k in range(0,len(lightingName)):
        if (k == 0):
            lights = lights1
            color = color1
        if (k == 1):
            lights = lights2
            color = color2
        if (k == 2):
            lights = lights3
            color = color3

        for h in range(0,len(materialName)):
            if (h == 0): material = material1
            if (h == 1): material = material2
            if (h == 2): material = material3
            #print(count)

            result = phong(normals, lights, color, material, view,

            ax = fig.add_subplot(9, 2, count)
            plt.imshow(result)
            plt.title('Sphere' + ' ' + settingsName[j] + ' ' + ligh

            #plt.subplot(9, 2, count)
            #plt.title('sphere' + ' ' + settingsName[j] + ' ' + lig
            #plt.imshow(result)
            count = count + 1
plt.tight_layout(pad=0.1)
plt.show()
```

Sphere original lighting1 material1

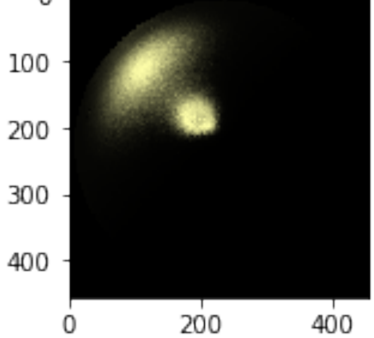


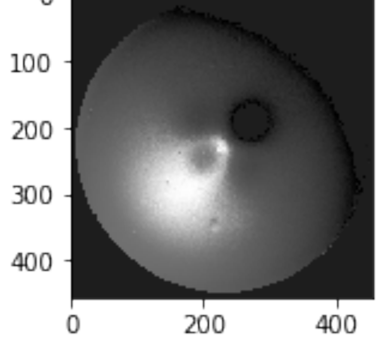
Sphere original lighting1 material2



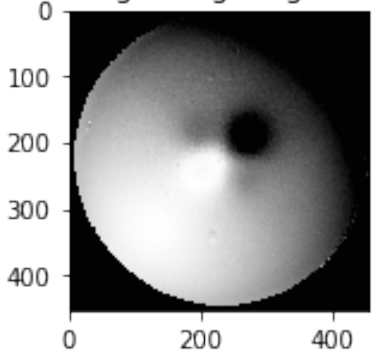
Sphere original lighting1 material3

Sphere original lighting2 material1

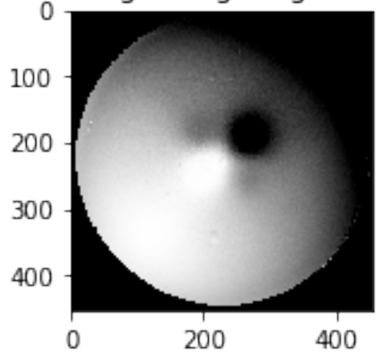




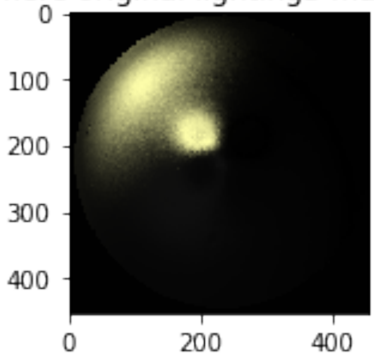
Sphere original lighting2 material2



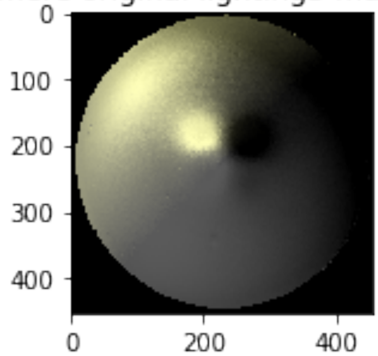
Sphere original lighting2 material3



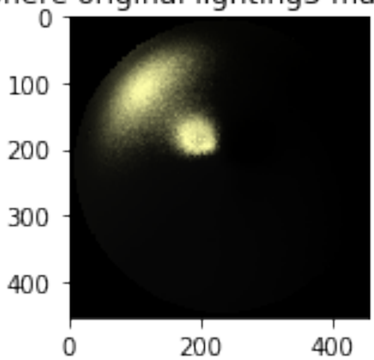
Sphere original lighting3 material1



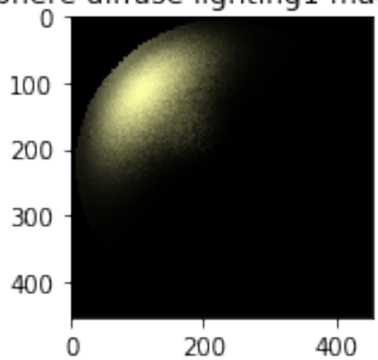
Sphere original lighting3 material2



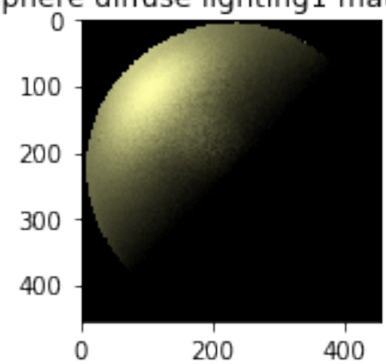
Sphere original lighting3 material3



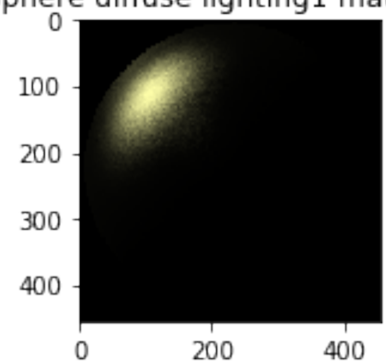
Sphere diffuse lighting1 material1



Sphere diffuse lighting1 material2

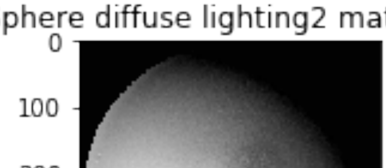


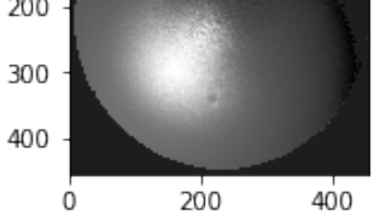
Sphere diffuse lighting1 material3

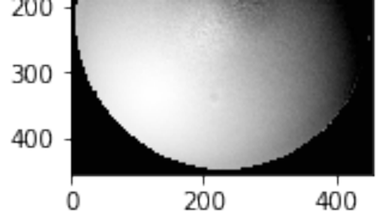


Sphere diffuse lighting2 material1

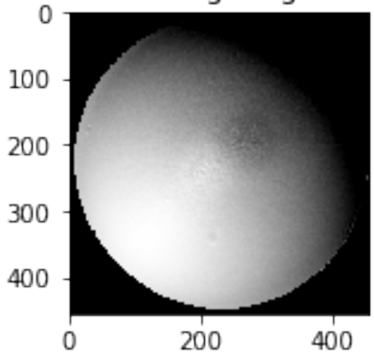
Sphere diffuse lighting2 material2



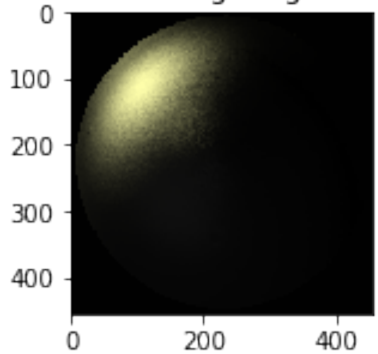




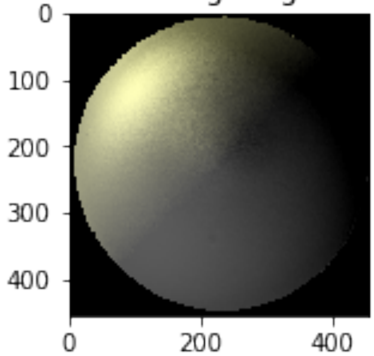
Sphere diffuse lighting2 material3



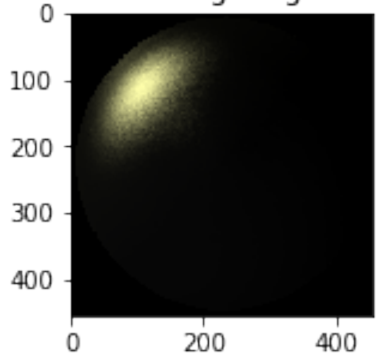
Sphere diffuse lighting3 material1



Sphere diffuse lighting3 material2



Sphere diffuse lighting3 material3

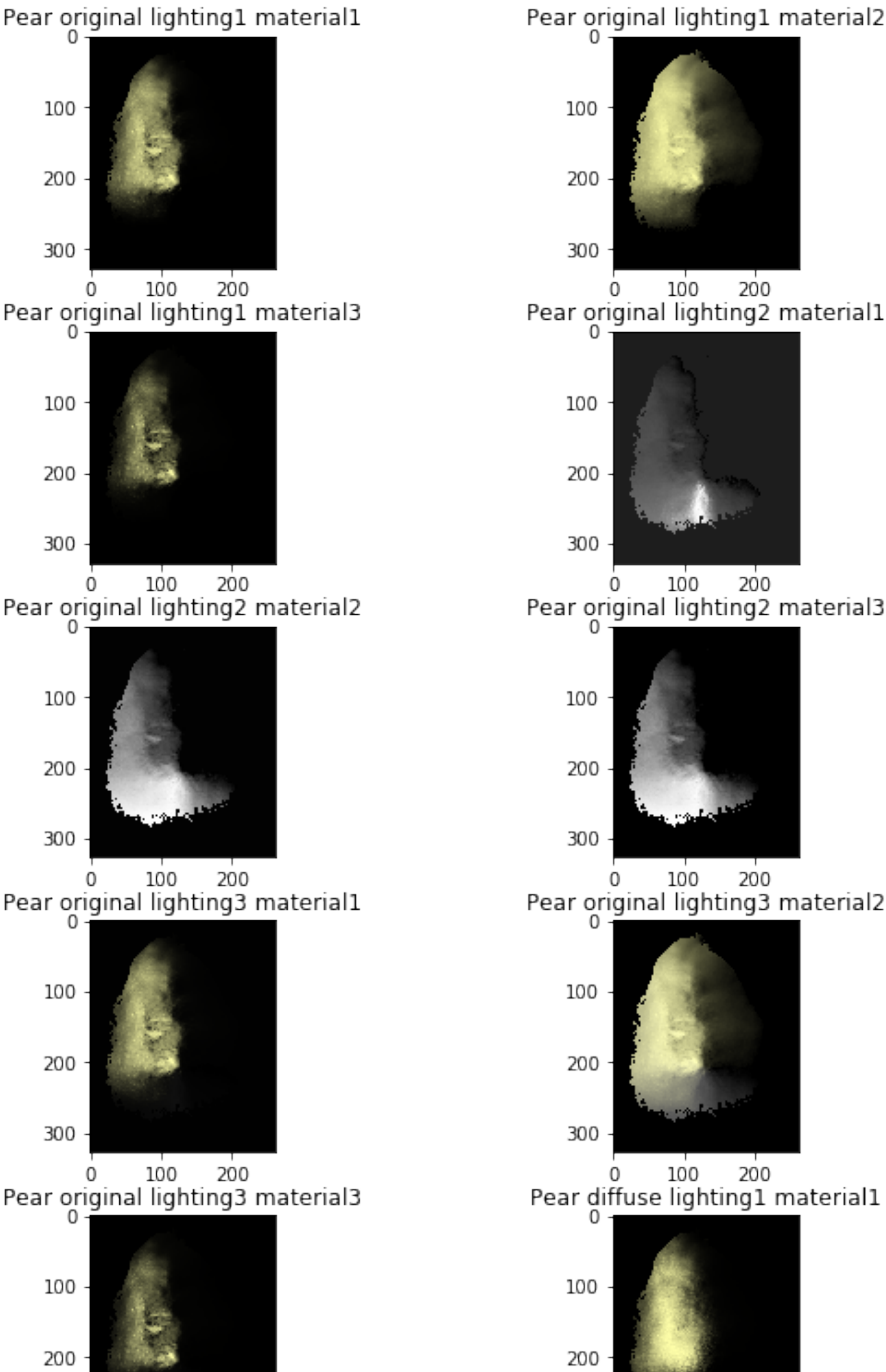


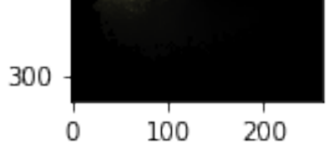
In [96]:

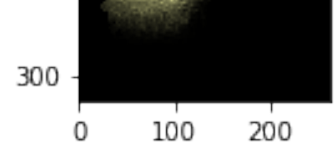
```
# Output the rendering results for pear
view =  np.array([[0],[0],[1]])
material = np.array([[0.1,0.75,5],[0.5,0.1,5],[0.5,0.5,10]])
lightcol1 = np.array([[1,0.75],[1,0.75],[0,0.5]])
lightcol2 =  np.array([[1.0/3,1],[-1.0/3,1],[1.0/2,1]])
#Display rendered results for pear for all materials and light sour
material1 = material[0,:]
material2 = material[1,:]
material3 = material[2,:]
view = view
lights1 = np.array([lightcol1[:,0]]).T
color1 = np.array([lightcol1[:,1]]).T
lights2 = np.array([lightcol2[:,0]]).T
color2 = np.array([lightcol2[:,1]]).T
lights3 = np.concatenate((lights1, lights2), axis=1)
color3 = np.concatenate((color1, color2), axis=1)
#test = phong(sphereDiffuse, lights2, color2, material1, view, sphe
#plt.imshow(test)
#plt.show()
```

```
# 2 - sphere and pear
# 2 - original and deffuse
# 3 - material 1,2 and 3
# 3 - light 1, 2, and 1&2


settingsName = ['original', 'diffuse']
lightingName = ['lighting1', 'lighting2', 'lighting3']
materialName = ['material1', 'material2', 'material3']

##pear
mask = pearMask
count = 1
fig = plt.figure(figsize=(8, 20))

for j in range(0,len(settingsName)):
    if (j == 0): normals = pearND
    if (j == 1): normals = pearDiffuse
    #print('j = ' + str(j))
    for k in range(0,len(lightingName)):
        if (k == 0):
            lights = lights1
            color = color1
        if (k == 1):
            lights = lights2
            color = color2
        if (k == 2):
            lights = lights3
            color = color3

        for h in range(0,len(materialName)):
            if (h == 0): material = material1
            if (h == 1): material = material2
            if (h == 2): material = material3
            #print(count)

            result = phong(normals, lights, color, material, view,

            ax = fig.add_subplot(9, 2, count)
            plt.imshow(result)
            plt.title('Pear' + ' ' + settingsName[j] + ' ' + lighti

            #plt.subplot(9, 2, count)
            #plt.title('Pear' + ' ' + settingsName[j] + ' ' + light
            #plt.imshow(result)
```

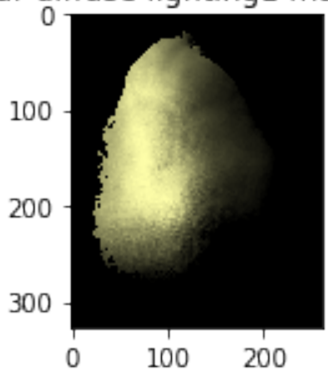
```
            count = count + 1
plt.tight_layout(pad=0.1)
plt.show()
```



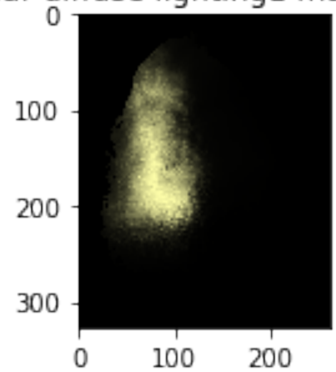




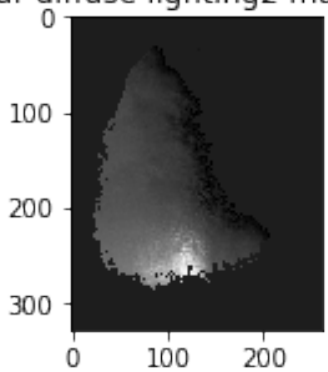
Pear diffuse lighting1 material2



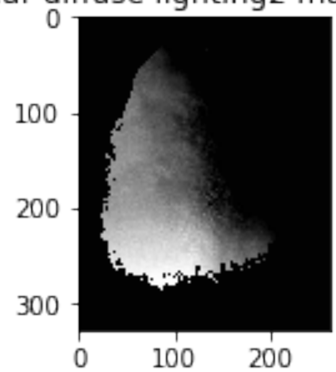
Pear diffuse lighting1 material3



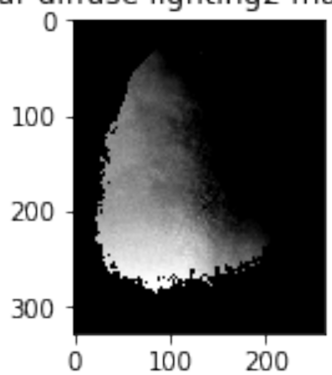
Pear diffuse lighting2 material1



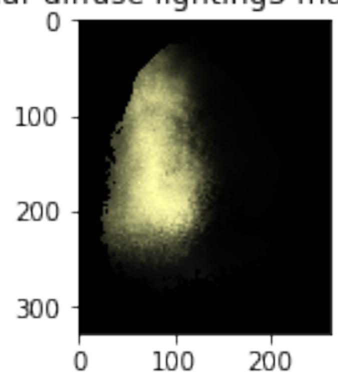
Pear diffuse lighting2 material2



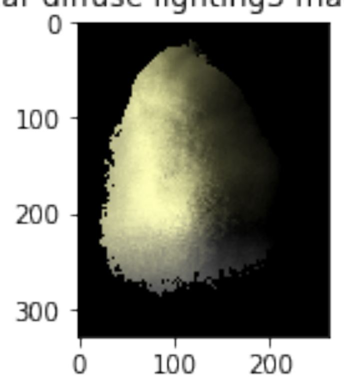
Pear diffuse lighting2 material3



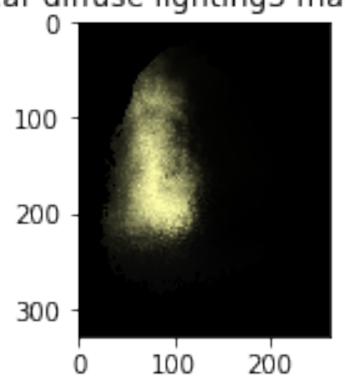
Pear diffuse lighting3 material1



Pear diffuse lighting3 material2



Pear diffuse lighting3 material3



In [ ]:

CSE 252A (Computer Vision I) · Fall 2020 · Assignment 1

Instructor: David Kriegman

Assignment published on Friday, October 16, 2020

Due on Friday, October 30, 2020 at 11:59 pm Pacific Time

Instructions

- Review the academic integrity and collaboration policies on Canvas. This assignment must be completed individually.
- All solutions must be written in this notebook. Programming aspects of the assignment must be completed using Python (preferably 3.6+).
- If you want to modify the skeleton code, you may do so. The existing code is merely intended to provide you with a framework for your solution.
- You may use Python packages for basic linear algebra (e.g. simple operations from NumPy or SciPy), but you may **not** use packages that directly solve the problem. If you are unsure about using a specific package or function, please ask the instructor and teaching assistants for clarification.
- You must submit, through Gradescope, both (1) this notebook exported as a PDF **and** (2) this notebook as an `.ipynb` file. You must mark every problem in the PDF on Gradescope. If you do not submit both the `.pdf` and `.ipynb`, and/or if you do not mark every problem in the PDF on Gradescope, you may receive a penalty.
- It is highly recommended that you begin working on this assignment early.
- **Late policy:** Assignments submitted late will receive a 10% grade reduction for each day late (e.g. an assignment submitted an hour after the due date will receive a 10% penalty, an assignment submitted 10 hours after the due date will receive a 10% penalty, and an assignment submitted 28 hours after the due date will receive a 20% penalty). Assignments will not be accepted 72 hours after the due date. If you require

an extension (for personal reasons only), you must request one as far in advance as possible. Extensions requested close to or after the due date will only be granted for clear emergencies or clearly unforeseeable circumstances.

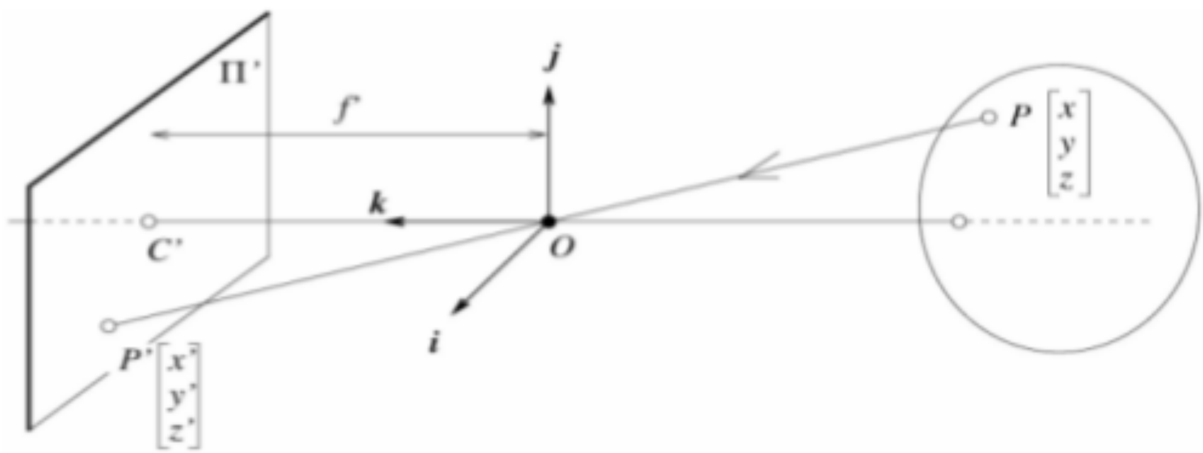
Problem 1: Perspective Projection and Homogenous Coordinates [10 pts]

Part 1 [3 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

$$Q1 = [4\ \text{-}1\ 3] + t[5\ 3\ 3]$$

$$Q2 = [1\ \text{-}3\ 2] + t[5\ 3\ 3]$$

where $-\infty \leq t \leq -2$.

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

Part 2 [6 pts]

Show that:

1) In R^3 distances are preserved under a rigid transformation.

2) In R^2 parallel lines remain parallel under an affine transformation.

3) If $a, b \in \mathrm{R}^3$ are orthogonal, they are orthogonal after rotating by a rotation matrix R

Part 3 [1 pts]

Given four points forming a square with Cartesian coordinates (0,0), (1,0), (1,1), and (0,1), find a projective transformation, A, which sends any two of the points to infinity while the other two are not sent to infinity. Define the matrix A below and print the result of applying the transformation to each of the four points.

In [112]:

Problem 1

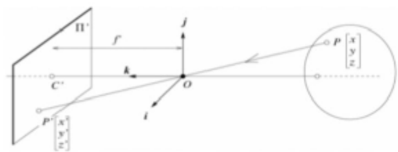
0.2 Problem 1: Perspective Projection and Homogenous Coordinates [10 pts]

0.2.1 Part 1 [3 pts]

Consider a perspective projection where a point

$$P = [x\ y\ z]^T$$

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

$$Q1 = [4\ \text{-}1\ 3] + t[5\ 3\ 3]$$

$$Q2 = [1\ \text{-}3\ 2] + t[5\ 3\ 3]$$

where $-\infty \leq t \leq -2$.

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

intersection given by:

$\left(f'\frac{x}{z}, f'\frac{y}{z}, f'\right)$,

but only care about coordinates in image plane, so endpoint of ray in image plane P' is

$\left(f'\frac{x}{z}, f'\frac{y}{z}\right)$

ray Q_1 is $Q_1 = \begin{bmatrix} 4 \\ -1 \\ 3 \end{bmatrix} + t\begin{bmatrix} 5 \\ 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 4+5t \\ -1+3t \\ 3+3t \end{bmatrix}$

Q_1 coordinates in P': $\left(\frac{f'(4+5t)}{3+3t}, \frac{f'(-1+3t)}{3+3t}\right)$

ray Q_2 is $Q_2 = \begin{bmatrix} 1+5t \\ -3+3t \\ 2+3t \end{bmatrix}$

Q_2 coordinates in P': $\left(\frac{f'(1+5t)}{2+3t}, \frac{f'(-3+3t)}{2+3t}\right)$

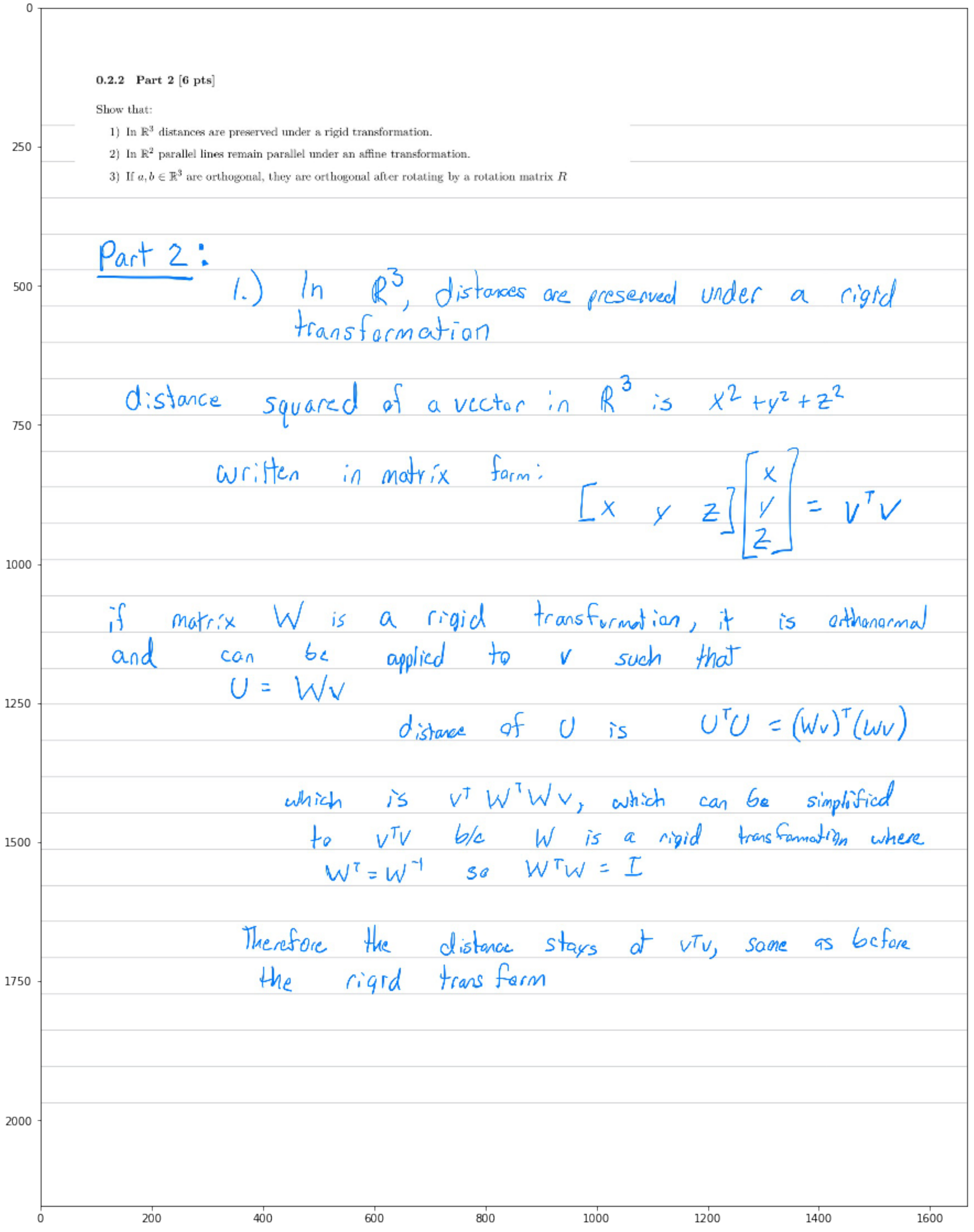
In [113]:

Q_1 @ $t=-2$: $\begin{bmatrix} \frac{2f'}{3} \\ \frac{7f'}{3} \end{bmatrix}$ Q_2 @ $t=-2$: $\begin{bmatrix} \frac{9f'}{4} \\ \frac{9f'}{4} \end{bmatrix}$

Q_2 @ $t=-\infty$: $\begin{bmatrix} \frac{5f'}{3} \\ f' \end{bmatrix}$ Q_2 @ $t=-\infty$: $\begin{bmatrix} \frac{5f'}{3} \\ f' \end{bmatrix}$

$\boxed{\text{both rays have the same vanishing point at } \begin{bmatrix} \frac{5f'}{3} \\ f' \end{bmatrix}}$

In [114]:



0.2.2 Part 2 [6 pts]

Show that:

1) In $\mathbb{R}^3$ distances are preserved under a rigid transformation.
2) In $\mathbb{R}^2$ parallel lines remain parallel under an affine transformation.
3) If $a, b \in \mathbb{R}^3$ are orthogonal, they are orthogonal after rotating by a rotation matrix R

Part 2:

1.) In $\mathbb{R}^3$, distances are preserved under a rigid transformation

distance squared of a vector in $\mathbb{R}^3$ is $x^2 + y^2 + z^2$

written in matrix form: $\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = v^T v$

if matrix W is a rigid transformation, it is orthonormal and can be applied to v such that $U = Wv$

distance of U is $U^T U = (Wv)^T (Wv)$

which is $v^T W^T W v$, which can be simplified to $v^T v$ b/c W is a rigid transformation where $W^T = W^{-1}$ so $W^T W = I$

Therefore the distance stays at $v^T v$, same as before the rigid transform

In [115]:

2.) In $\mathbb{R}^2$, parallel lines remain parallel under affine transformation.

Affine transform: $f(x) = Ax + b$

2 parallel lines have same slope

$L_1 = c + ds$ where s is the slope

$L_2 = e + fs$ and $d = gf$ for $g = 1$

<u>transform both</u>

$L_1: A(c+ds) + b$ $\quad$ $L_2: A(e+fs) + b$

$: Ac + b + (Ad)s$ $\quad$ $: Ae + b + (Af)s$

$Ac + b + (Agf)s$

so since for some $g = 1$, they have the same slope, that is presved in both have same slope for $g = 1$.

3.) If $a, b \in \mathbb{R}^3$ orthog, they are orthog after rotation.

for 2 orthog vectors, dot product $= 0$ so $a^T b = 0$

rotate both to get a_R and b_R

$(Ra)^T(Rb) = a_R^T b_R$

$a^T R^T R b = a_R^T b_R$

$a^T b = 0 = a_R^T b_R$

so dot product of a_R and b_R $= 0$ so orthaganality is preserved by transformation.

In [116]:

```
A
[[ 1  0 -1]
 [ 0  1  0]
 [ 0  0  1]]
X
[[0 1 1 0]
 [0 0 1 1]
 [0 1 0 1]]
AX
[[ 0  0  1 -1]
 [ 0  0  1  1]
 [ 0  1  0  1]]
```

Problem 2: Image Formation and Rigid Body Transformations [10 points]

In this problem we will practice rigid body transformations and image formations through the projective 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 4×4 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 3×4 matrix.

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

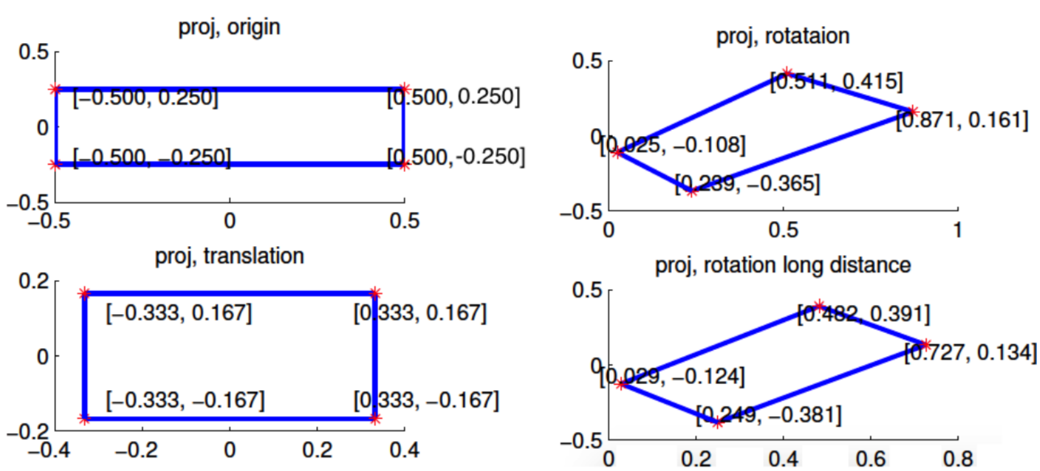
We will consider four different settings of focal length, viewing angles and camera positions below. For each of these calculate:

a) Extrinsic transformation matrix,

b) Intrinsic camera matrix under the perspective camera assumption.

c) 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]. ${}^{B}O_A = [1\ \text{-}1\ 1]^T$. 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 45 degree rotation around the y-axis followed by a 20 degree rotation around the x-axis. ${}^{B}O_A = [\text{-}1\ 0\ 1]^T$.
4. [Translation and Rotation, long distance]. Focal length = 7. ${}^{B}_{A}R$ encodes a 45 degree rotation around the y-axis followed by a 20 degree rotation around the x-axis. ${}^{B}O_A = [\text{-}1\ \text{-}1\ 21]^T$.

> You can refer the Richard Szeliski starting page 36 for image formation and the extrinsic matrix.

> Intrinsic matrix calculation for perspective camera models was covered in class and can be found in lecture 3 https://canvas.ucsd.edu/courses/18894/files/folder/lectures (https://canvas.ucsd.edu/courses/18894/files/folder/lectures)

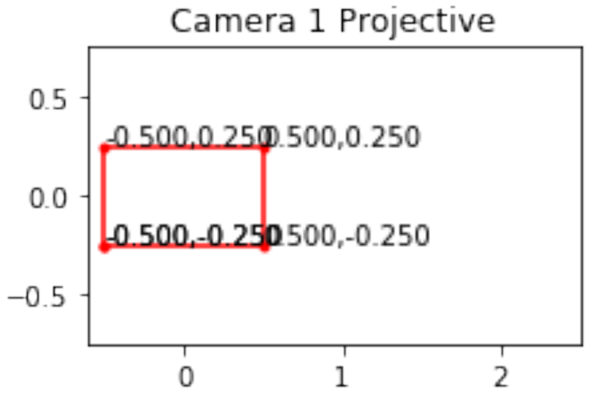
> You can also refer lecture 2 of the previous year's course as well for further information if you wish! https://cseweb.ucsd.edu/classes/fa19/cse252A-a/lec2.pdf (https://cseweb.ucsd.edu/classes/fa19/cse252A-a/lec2.pdf)

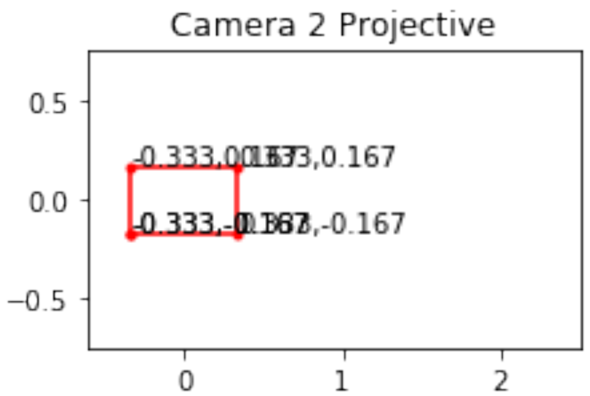
We will not use a full intrinsic camera matrix (e.g. that maps centimeters to pixels, and defines the coordinates of the center of the image), but only parameterize this with f, the focal length. In other words: the only parameter in the intrinsic camera matrix under the perspective assumption is f.

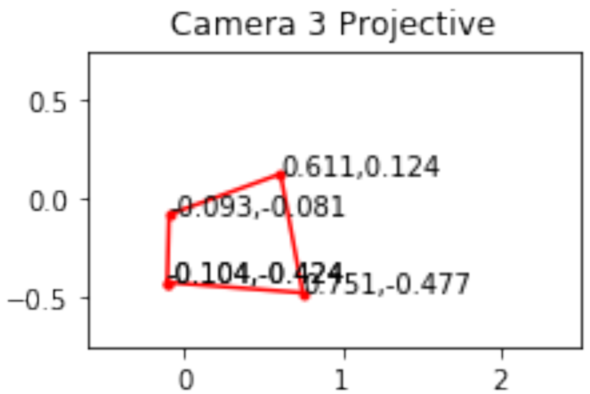
In [2]:

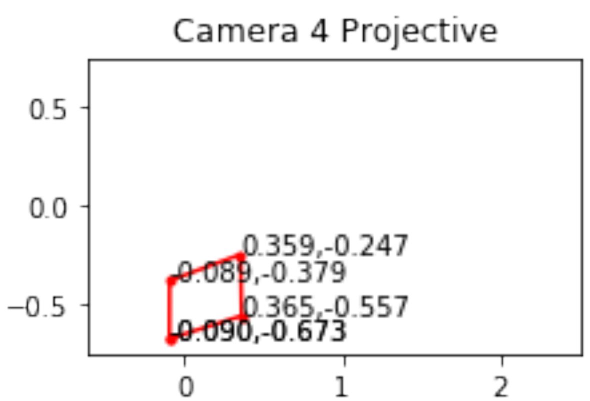
```

```

In [3]:


Camera 1 Projective
-0.500,0.250
0.500,0.250
-0.500,-0.250
0.500,-0.250
0.5
0.0
−0.5
0
1
2


Camera 2 Projective
-0.333,0.167
0.333,0.167
-0.333,-0.167
0.333,-0.167
0.5
0.0
−0.5
0
1
2


Camera 3 Projective
0.611,0.124
-0.093,-0.081
-0.104,-0.424
0.751,-0.477
0.5
0.0
−0.5
0
1
2


Camera 4 Projective
0.359,-0.247
-0.089,-0.379
0.365,-0.557
-0.090,-0.673
0.5
0.0
−0.5
0
1
2

Problem 3: Homography [12 pts]

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:

$$q \approx Hp$$

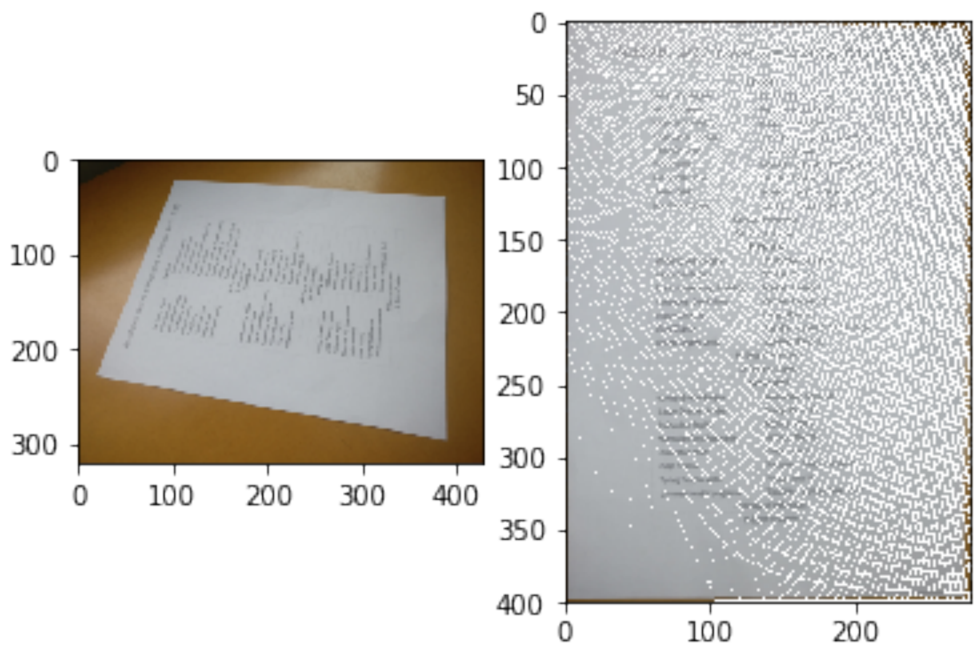
Here $\approx$ denotes equality in homogeneous coordinates, meaning that the left and right hand sides are proportional. Note that H only depends on the plane and the projection matrices of the two cameras.

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.

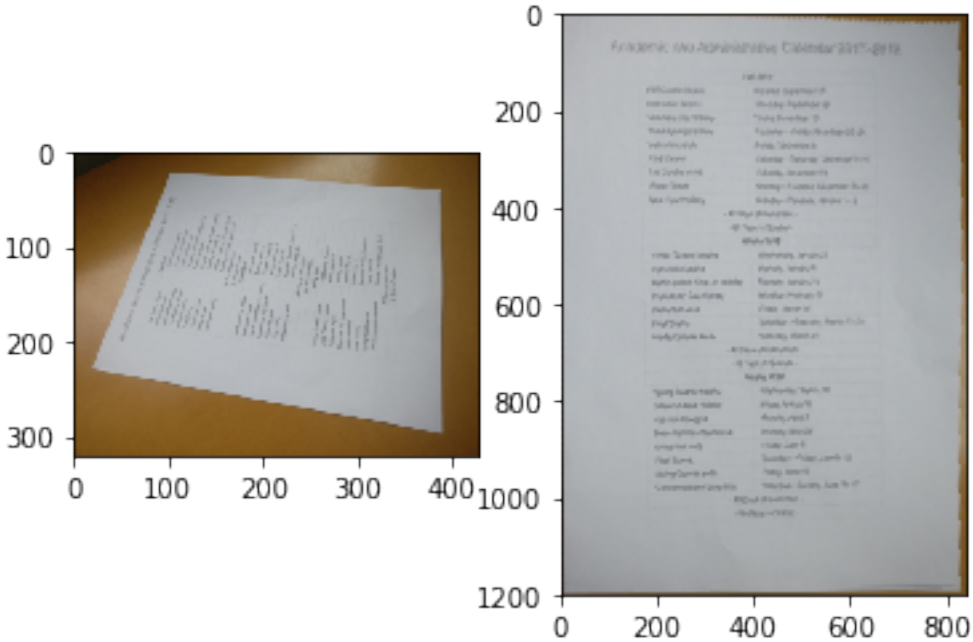
In this problem, complete the code for computeH and warp functions that can be used in the skeletal code that follows.

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 an image of an image being imaged. The goal will be to map the given image onto both the portrait as well as the screen of the tablet imaging the portrait. This will require the computation of two homographies.

1.



2.



3.

1. 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.
2. The target image can start out as an empty matrix that you fill out using your code.
3. You will have to implement the computeH function that computes a homography. It takes in exactly four point correspondences between the source image and target image in homogeneous coordinates respectively and returns a 3×3 homography matrix.
4. You will also have to implement the three warp functions in the skeleton code given and plot the resultant image pairs. For plotting the results of warps 1 and 2, make

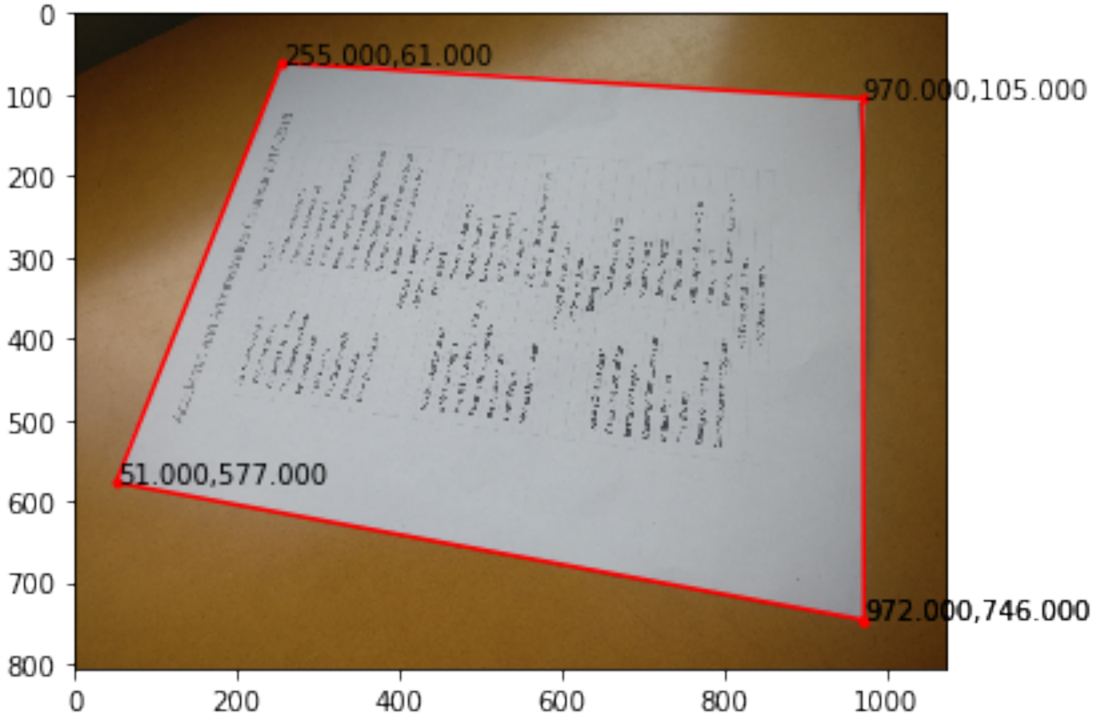
sure that the target image is not smaller than the source image.

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 [99]:

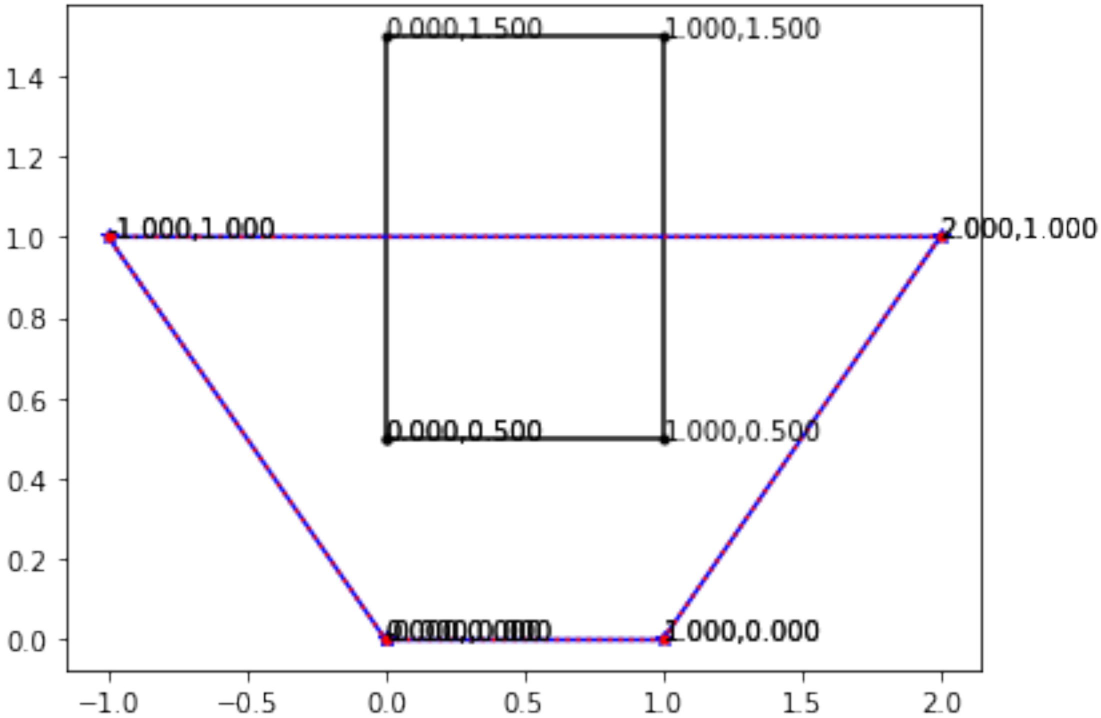
```
```



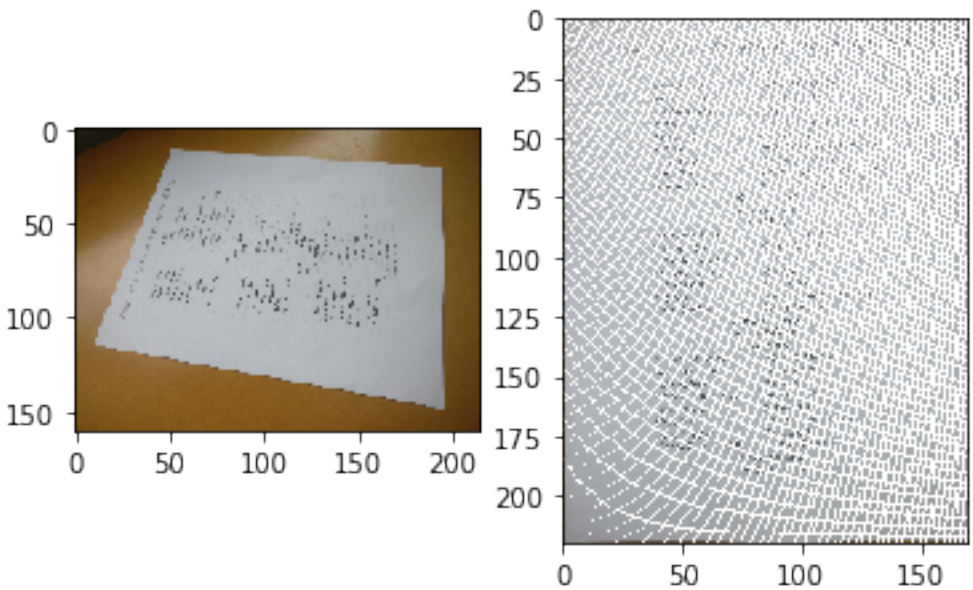

```
(806, 1074, 3)
```

In [100]:

```
[[1.00000000e+00]
 [4.41370382e-16]]
```



In [6]:

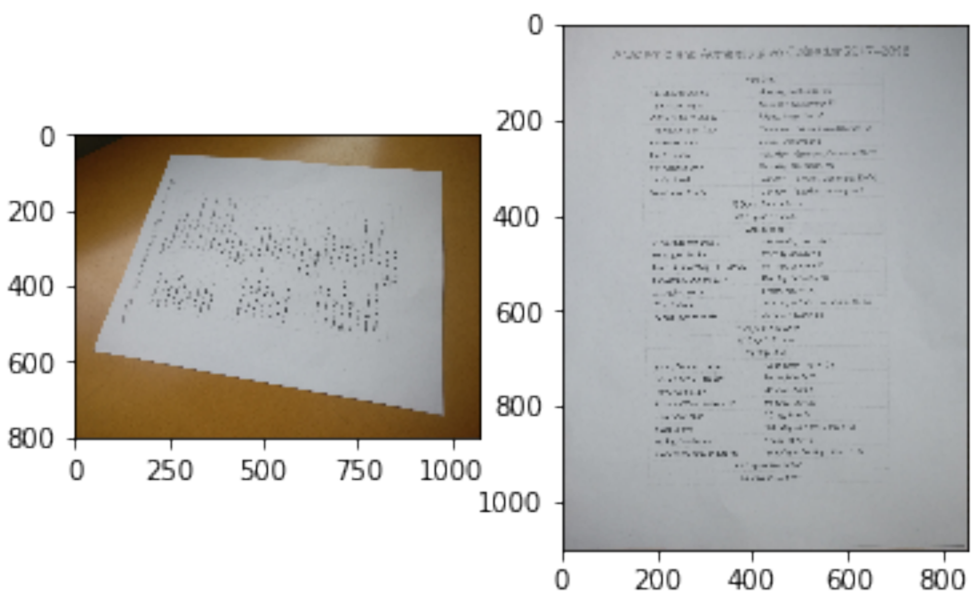


The output of warp1 of your code probably has some striations or noise. The larger you make your target image, the less it will resemble the document in the source image. Why is this happening?

Answer: This is happening because the result starts out as a matrix of all zeros (in my case), and if a pixel isn't filled in, it will remain a zero. Pixels end up not being filled in because the result image is larger than the initial image so not every pixel in the result image can be mapped to a pixel with color data in the initial image. The noisy qualitatively can be thought of as tears that would appear if a peice of cloth the size of the desired region in the original image was stretched to the dimensions of the result.

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

In [101]:

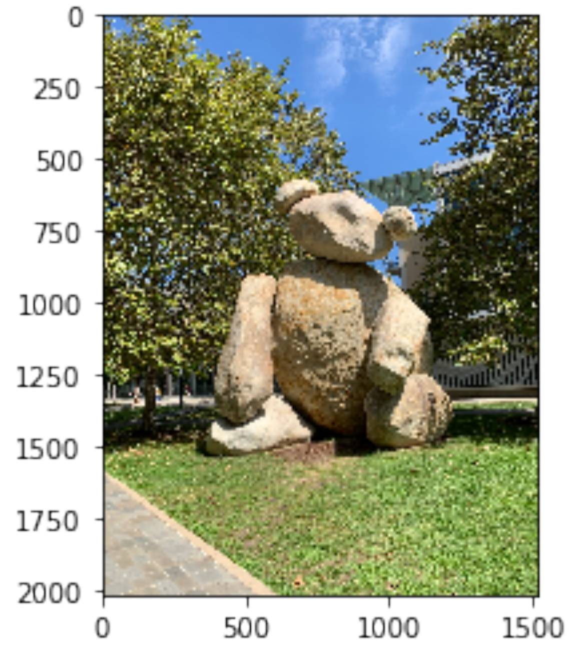


Try playing around with the size of your target image in warp1 versus in warp2, additionally you can also implement nearest pixel interpolation or bi-linear interpolations and see if that makes a difference in your output.

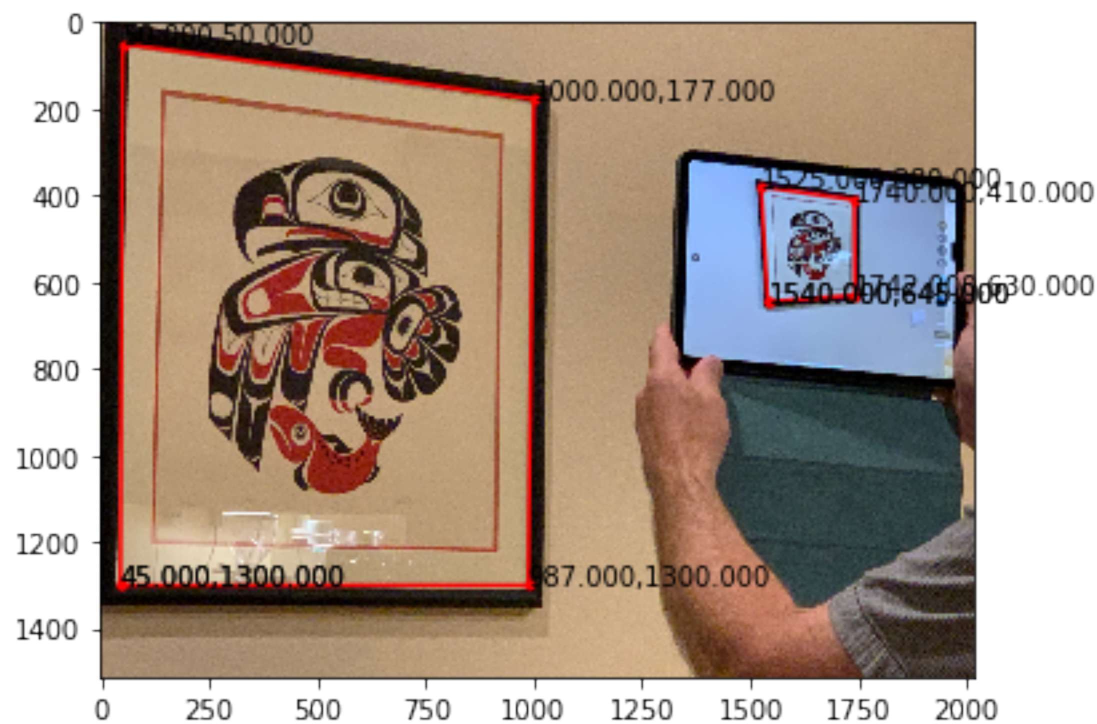
Answer: Warp1 gives the best results in a smaller target image, since more of its pixels are filled in with pixels from the source image; as the target image gets larger, a smaller and smaller percentage of its pixels are filled in, resulting in a lower quality image with more noise. Warp 2 gives the good results in an image roughly the same dimensions (but rotated 90 degrees and slightly larger) than the source image. This is because warp 2 guarentees all the pixels in the target image are filled in. The result was a very good image, and interpolation isn't really necessary to make it better. (see the saved warp2.png that was outputted by the code into the folder). Interpolation (bicubic, very resouce heavy) also crashed the kernel every time it was implemented on anything over than a very downsized image, leading this student to conclude the best approach, if maximum quality was desired, is to use a non-downsized image for warp2, as the result was better than a downsized image that was bicubic-ly interpolated, since the downsizing had to be significant in order for the interpolation to not crash the kernel.

In warp3, you'll be replacing the portrait and image of the portrait in a provided image with another image. Read in "bear.png" as the source image, and "gallery.png" will serve as the target.

In [102]:

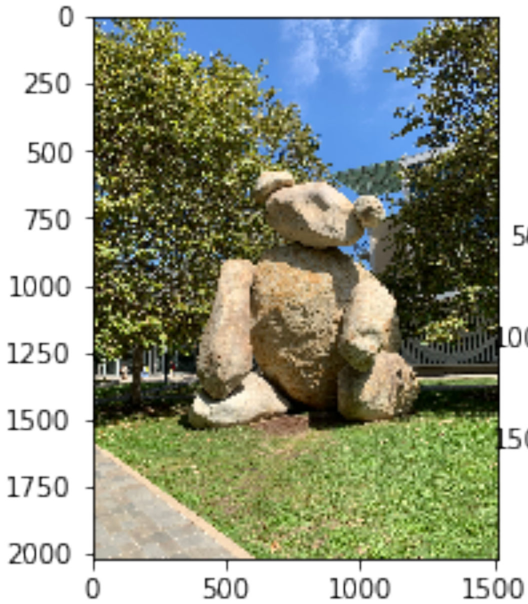


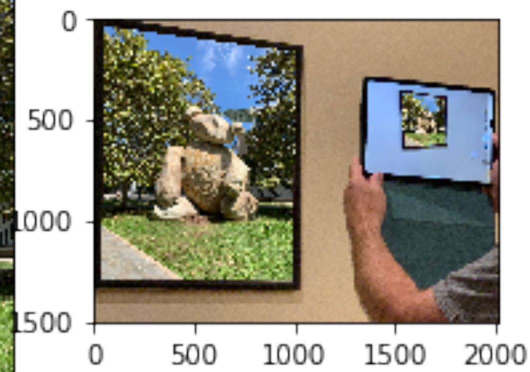
(2016, 1512, 3)



(1512, 2016, 3)

In [103]:





Problem 4: Surface Rendering [18 pts]

In this portion of the assignment we will be exploring different methods of approximating local illumination of objects in a scene. This last section of the homework will be an exercise in rendering surfaces. Here, you need use the surface normals and the masks from the provided pickle files, with various light sources, different materials, and using a number of illumination models. For the sake of simplicity, multiple reflections of light rays, and occlusion of light rays due to object/scene can be ignored.

Data

The surface normals and masks are to be loaded from the respective pickle files. For comparison, you should display the rendering results for both normals calculated from the original image and the diffuse components. There are 2 images that we will be playing with--namely one of a sphere and the other of a pear.

Assume that the albedo map is uniform.

Lambertian Illumination

One of the simplest models available to render 3D objections with illumination is the Lambertian model. This model finds the apparent brightness to an observer using the direction of the light source $\mathbf{L}$ and the normal vector on the surface of the object $\mathbf{N}$. The brightness intensity at a given point on an object's surface, $\mathbf{I_d}$, with a single light source is found using the following relationship:

$$\mathbf{I_d} = \mathbf{L} \cdot \mathbf{N}(I_l \mathbf{C})$$

where, $\mathbf{C}$ and I_l are the the color and intensity of the light source respectively.

Phong Illumination

One major drawback of Lambertian illumination is that it only considers the diffuse light in its calculation of brightness intensity. One other major component to illumination rendering is the specular component. The specular reflectance is the component of light that is reflected in a single direction, as opposed to all directions, which is the case in diffuse reflectance. One of the most used models to compute surface brightness with specular components is the Phong illumination model. This model combines ambient lighting, diffused reflectance as well as specular reflectance to find the brightness on a surface. Phong shading also considers the material in the scene which is characterized by four values: the ambient reflection constant (k_a), the diffuse reflection constant (k_d), the specular reflection constant (k_s) and α the Phong constant, which is the 'shininess' of an object. Furthermore, since the specular component produces 'rays', only some of which would be observed by a single observer, the observer's viewing direction ($\mathbf{V}$) must also be known. For some scene with known material parameters with M light sources the light intensity $\mathbf{I}_{phong}$ on a surface with normal vector $\mathbf{N}$ seen from viewing direction $\mathbf{V}$ can be computed by:

$$\mathbf{I}_{phong} = k_a \mathbf{I}_a + \sum_{m \in M} \left\{ k_d (\mathbf{L}_m \cdot \mathbf{N}) \mathbf{I}_{m,d} + k_s (\mathbf{R}_m \cdot \mathbf{V})^{\alpha} \mathbf{I}_{m,s} \right\},$$

$$\mathbf{R}_m = 2\mathbf{N}(\mathbf{L}_m \cdot \mathbf{N}) - \mathbf{L}_m,$$

where $\mathbf{I}_a$, is the color and intensity of the ambient lighting, $\mathbf{I}_{m,d}$ and $\mathbf{I}_{m,s}$ are the color values for the diffuse and specular light of the mth light source.

Rendering

Please complete the following:

1. Write the function `lambertian()` that calculates the Lambertian light intensity given the light direction $\mathbf{L}$ with color and intensity $\mathbf{C}$ and $I_l = 1$, and normal vector $\mathbf{N}$. Then use this function in a program that calculates and displays the specular sphere and the pear using each of the two lighting sources found in Table 1. *Note: You do not need to worry about material coefficients in this model.*
2. Write the function `phong()` that calculates the Phong light intensity given the material constants (k_a, k_d, k_s, α), $\mathbf{V} = (0, 0, 1)^\top$, $\mathbf{N}$ and some number of M light sources. Then use this function in a program that calculates and displays the specular sphere and the pear using each of the sets of coefficients found in Table 2 with each light source individually, and both light sources combined.

Hint: To avoid artifacts due to shadows, ensure that any negative intensities found are set to zero.

Table 1: Light Sources

m	**Location**	**Color (RGB)**
1	$(1, 1, 0)^\top$	$(0.75, 0.75, 0.5)$
2	$(\frac{1}{3}, -\frac{1}{3}, \frac{1}{2})^\top$	$(1, 1, 1)$

Table 2: Material Coefficients

Mat.	k_a	k_d	k_s	α
1	0	0.1	0.75	5
2	0	0.5	0.1	5
3	0	0.5	0.5	10

Part 1. Loading pickle files and plotting the normals [4 pts] (Sphere - 2pts, Pear - 2pts)

In this first part, you are required to work with 2 images, one of a sphere and the other one of a pear. The pickle file normals.pickle is a list consisting of 4 numpy matrices which are

1) Normal Vectors for the sphere with specularities removed (Diffuse component)
2) Normal Vector for the sphere
3) Normal Vectors for the pear with specularities removed (Diffuse component)
4) Normal vectors for the pear

Please load the normals and plot them using the function plot_normals which is provided.

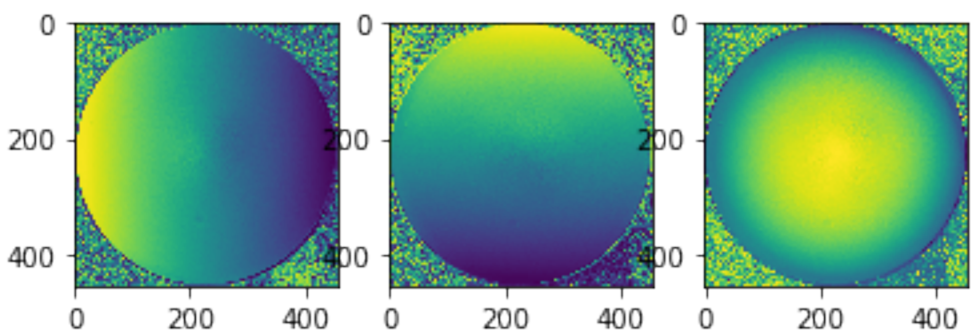
In [52]:

```
```

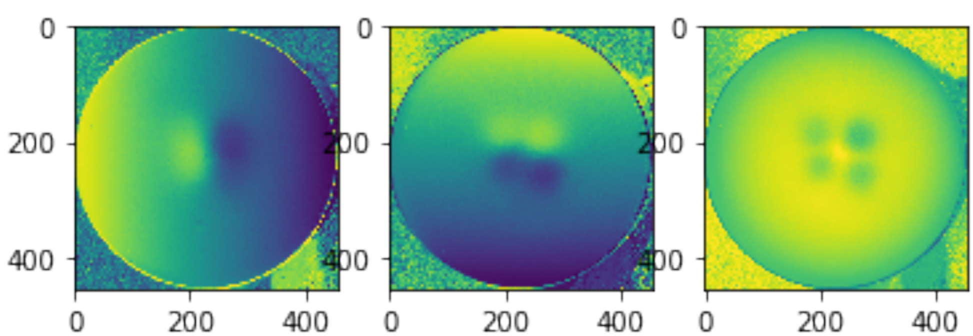
In [54]:

```
```

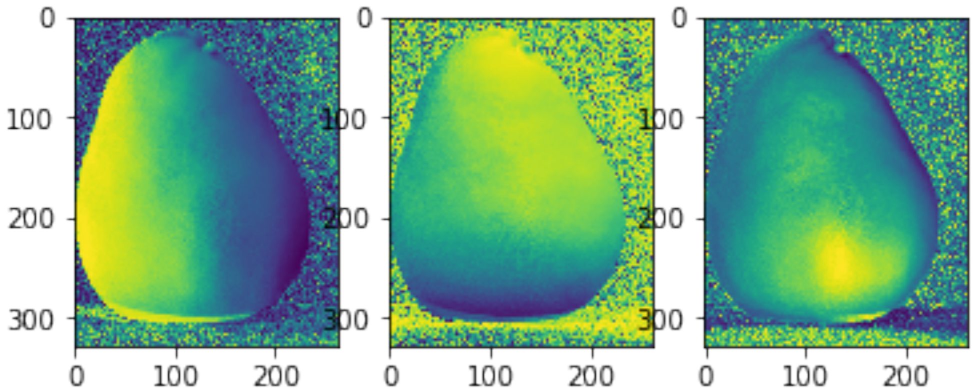
```
Normals:
Diffuse
```



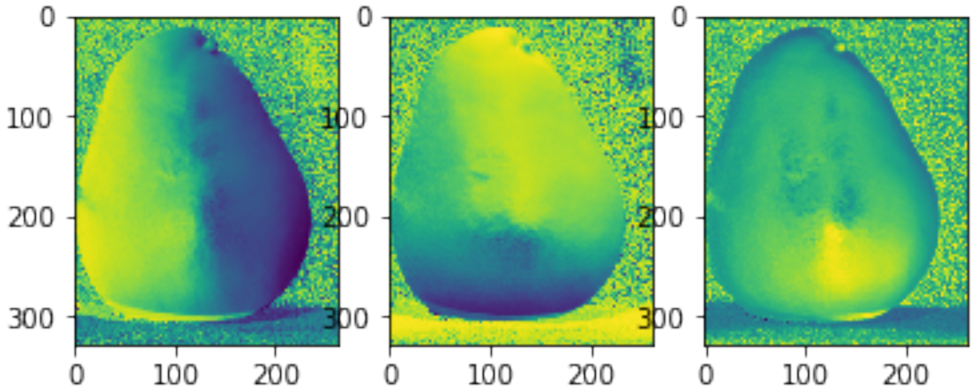

```
Original
```




```
Normals:
Diffuse
```



Original



Part 2. Lambertian model [6 pts]

Fill in your implementation for the rendered image using the lambertian model.

In [55]:

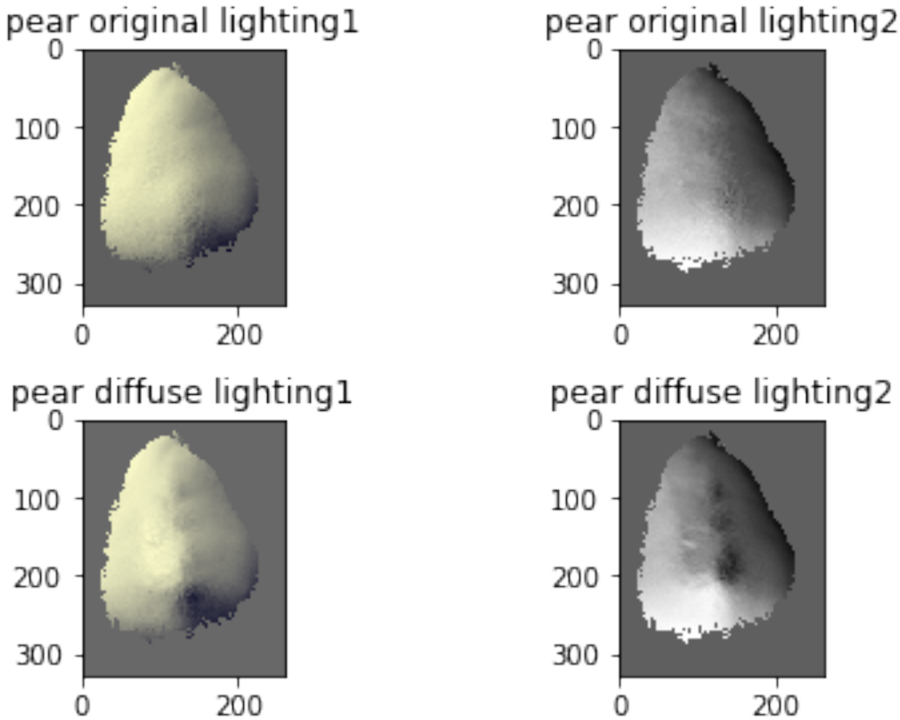
```
```

In [74]:

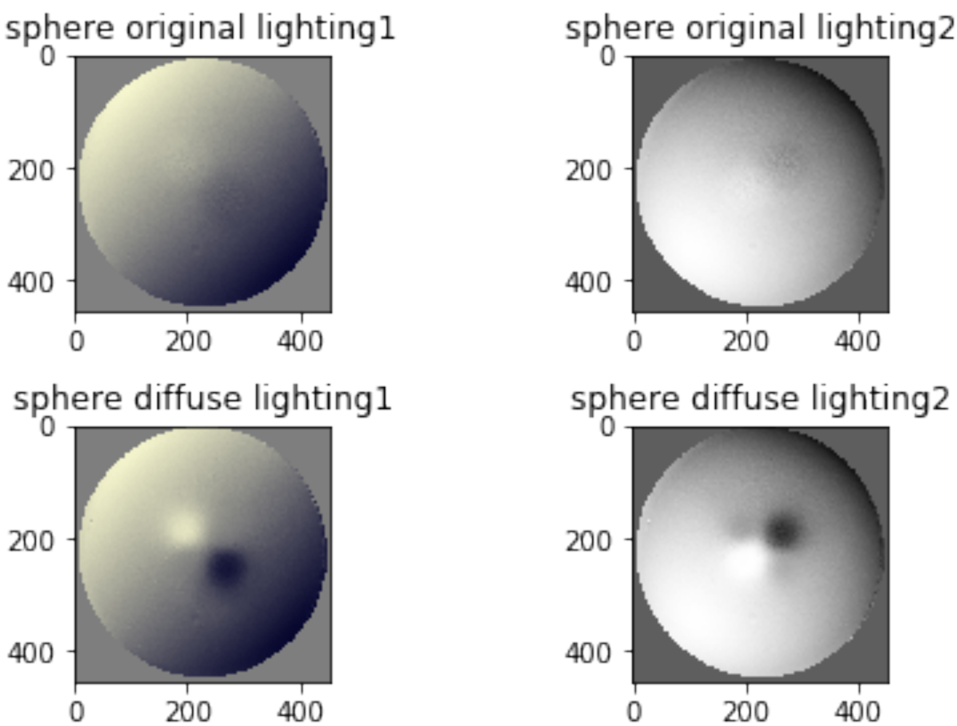
```
```

Plot the rendered results for both the sphere and the pear for both the original and the diffuse components. Remember to first load the masks from the masks.pkl file. The masks.pkl file is a list consisting of 2 numpy arrays-
1)Mask for the sphere
2)Mask for the pear
Remember to plot the normalized image using the function normalize which is provided.

In [75]:



In [76]:



Part 3. Phong model [8 pts]

Please fill in your implementation for the Phong model below.

In [77]:

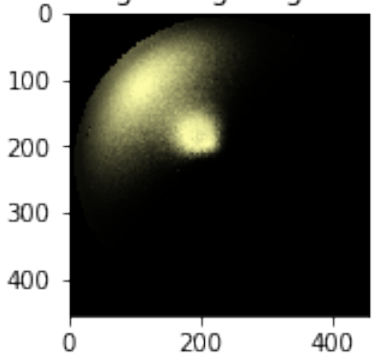
```

```

With the function completed, plot the rendering results for the sphere and pear (both diffuse and original compnents) for all the materials and light sources and also with the combination of both the light sources.

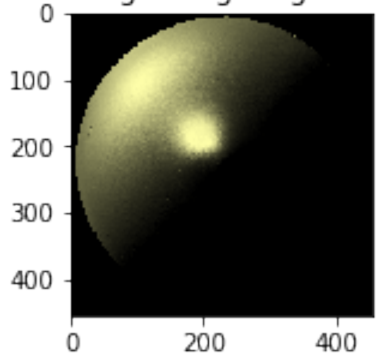
In [95]:

```

```

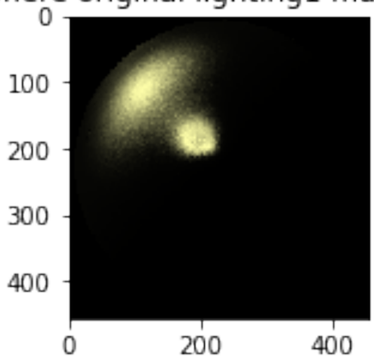
Sphere original lighting1 material1



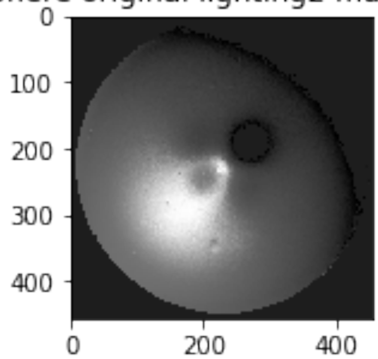
Sphere original lighting1 material2



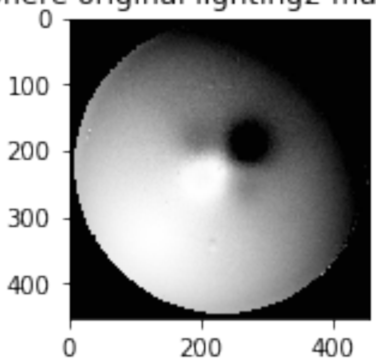
Sphere original lighting1 material3



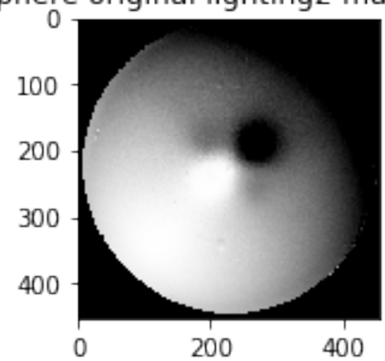
Sphere original lighting2 material1



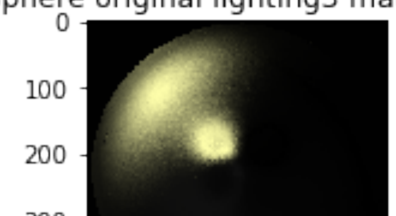
Sphere original lighting2 material2



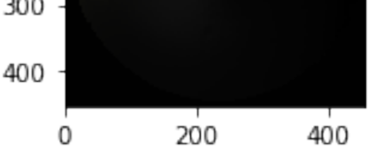
Sphere original lighting2 material3

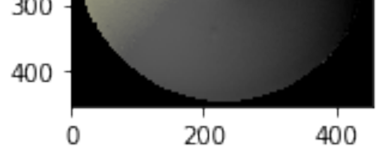


Sphere original lighting3 material1

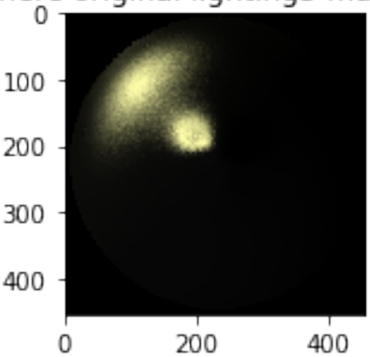


Sphere original lighting3 material2

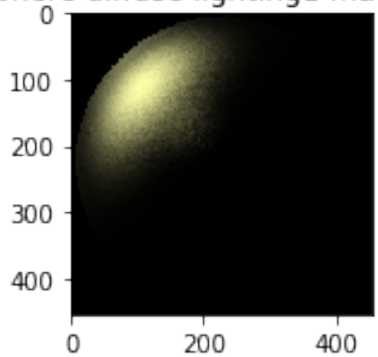




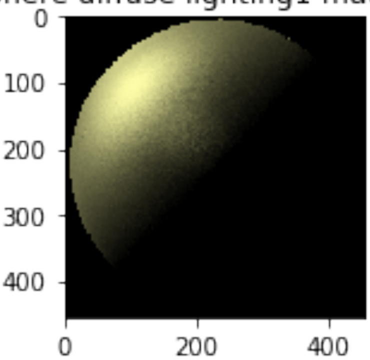
Sphere original lighting3 material3



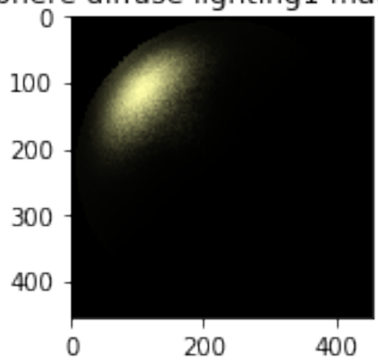
Sphere diffuse lighting1 material1



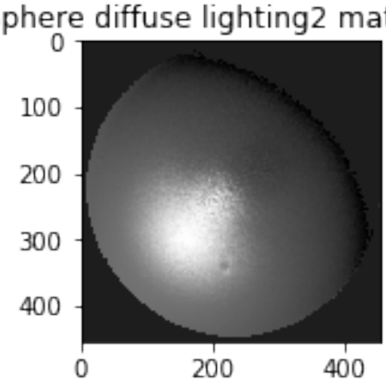
Sphere diffuse lighting1 material2



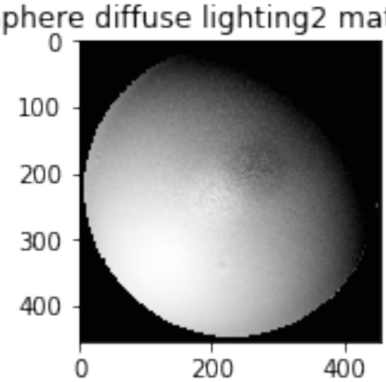
Sphere diffuse lighting1 material3



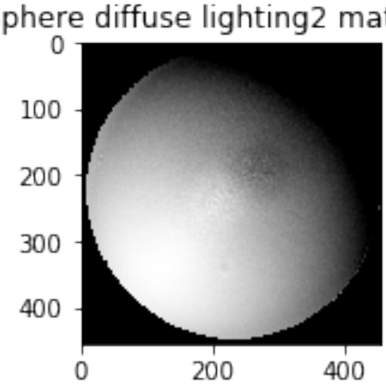
Sphere diffuse lighting2 material1



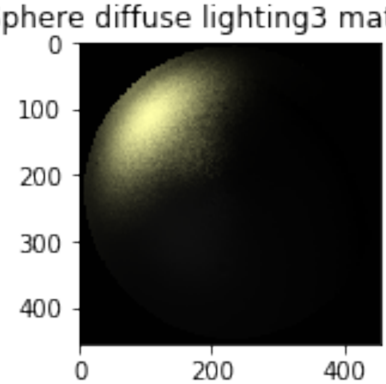
Sphere diffuse lighting2 material2



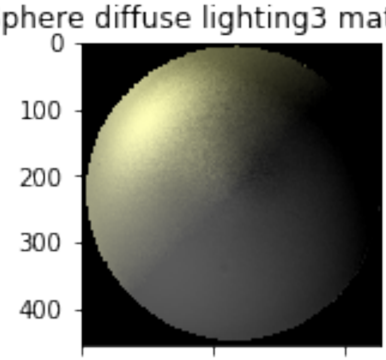
Sphere diffuse lighting2 material3



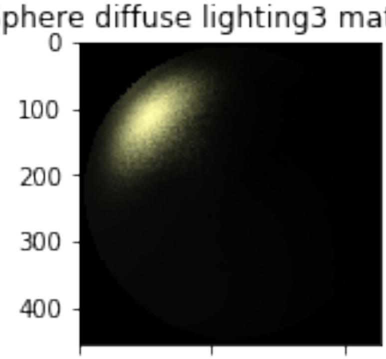
Sphere diffuse lighting3 material1



Sphere diffuse lighting3 material2



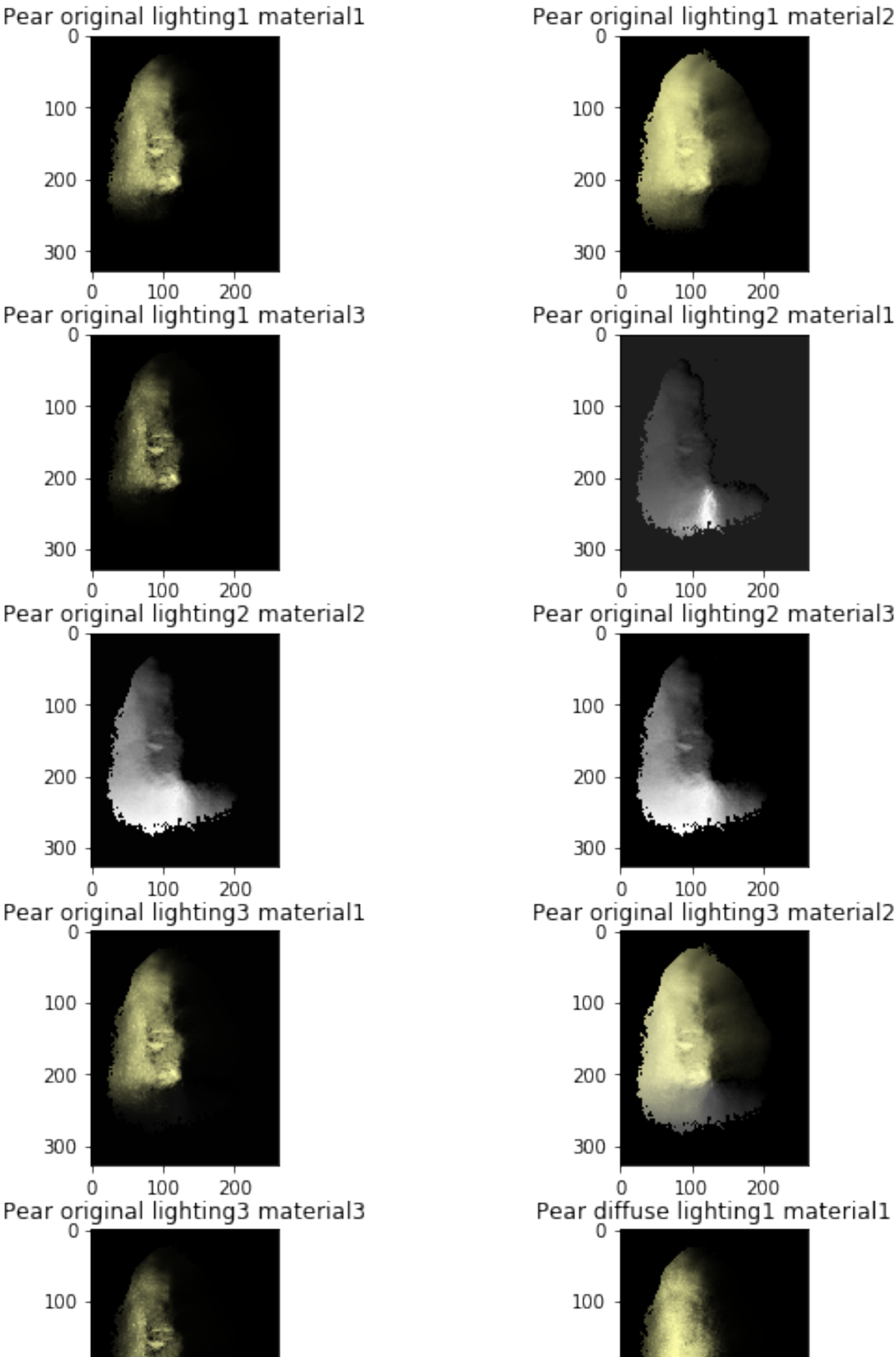
Sphere diffuse lighting3 material3

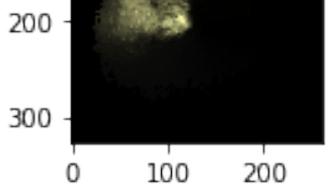


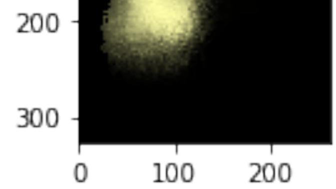
0 200 400 0 200 400

In [96]:

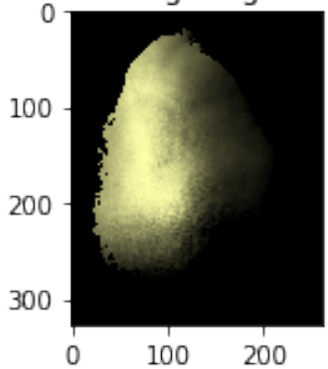
```

```



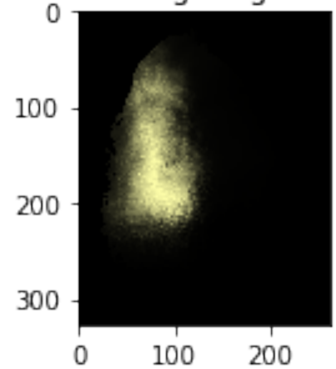




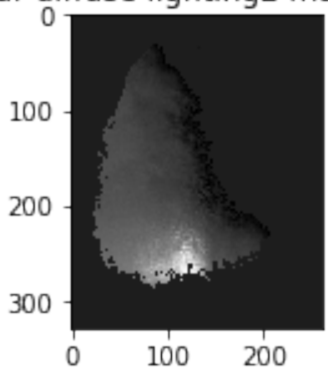
Pear diffuse lighting1 material2



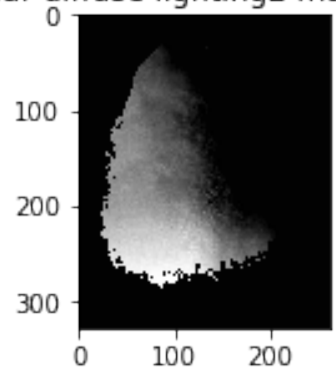
Pear diffuse lighting1 material3



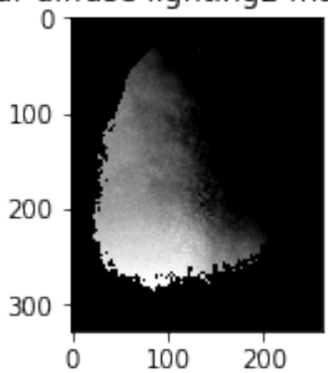
Pear diffuse lighting2 material1



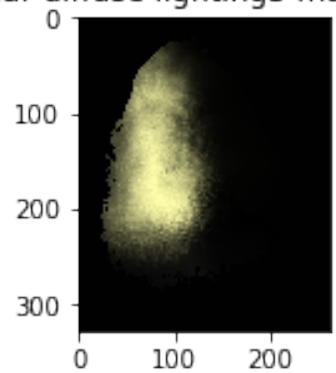
Pear diffuse lighting2 material2



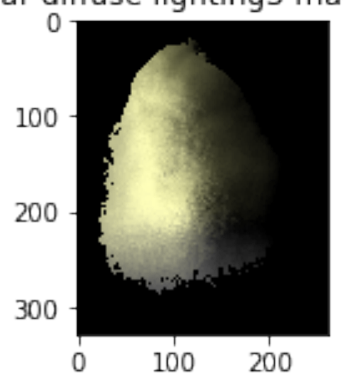
Pear diffuse lighting2 material3



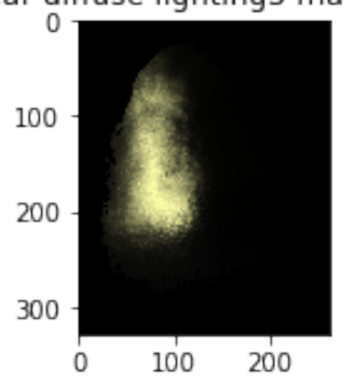
Pear diffuse lighting3 material1



Pear diffuse lighting3 material2



Pear diffuse lighting3 material3



In [ ]: