HW1

October 17, 2021

1 CSE 252A Computer Vision I Fall 2021 - Assignment 1

1.1 Instructor: Ben Ochoa

- Assignment Published On: **Wed, October 6, 2021**.
- Due On: **Wed, October 20, 2021 11:59 PM (Pacific Time)**.

1.2 Instructions

Please answer the questions below using Python in the attached Jupyter notebook and follow the guidelines below:

- This assignment must be completed **individually**. For more details, please follow the Academic Integrity Policy and Collaboration Policy on Canvas.
- All the solutions must be written in this Jupyter notebook.
- After finishing the assignment in the notebook, please export the notebook as a PDF and submit both the notebook and the PDF (i.e. the `.ipynb` and the `.pdf` files) on Gradescope.
- You may use basic algebra packages (e.g. `NumPy`, `SciPy`, etc) but you are not allowed to use the packages that directly solve the problems. Feel free to ask the instructor and the teaching assistants if you are unsure about the packages to use.
- It is highly recommended that you begin working on this assignment early.

Late Policy: Assignments submitted late will receive a 15% grade reduction for each 12 hours late (i.e., 30% per day). Assignments will not be accepted 72 hours after the due date. If you require an extension (for personal reasons only) to a due date, 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.

1.2.1 Problem 1: Geometry (15 points)

Consider a line in the 2D plane, whose equation is given by $a\tilde{x} + b\tilde{y} + c = 0$, where $\boldsymbol{l} = (a, b, c)^\top$ and $\boldsymbol{x} = (\tilde{x}, \tilde{y}, 1)^\top$. Noticing that $\boldsymbol{x}$ is a homogeneous representation of $\tilde{\boldsymbol{x}} = (\tilde{x}, \tilde{y})^\top$, we can view $\boldsymbol{l}$ as a homogeneous representation of the line $a\tilde{x} + b\tilde{y} + c = 0$. We see that the line is also defined up to a scale since $(a, b, c)^\top$ and $k(a, b, c)^\top$ with $k \neq 0$ represents the same line.

1. [6 points] Prove $x^T l + l^T x = 0$, if a point x in homogeneous coordinates lies on the homogeneous line l.

Solution:

If $x^T l = 0$, where $\boldsymbol{x}^T = (\tilde{x}, \tilde{y}, 1)$ and $\boldsymbol{l} = (a, b, c)^T$ because a point x in homogeneous coordinates lies on the homogeneous line l,

then:

$x^T l + l^T x = 0$

$0 + l^T x = 0$

$l^T x = 0$

Hence, $x^T l + l^T x = 0 + 0 = 0$.

2. [2 points] What is the line, in homogenous coordinates, joining the inhomogeneous points $(1, 7)$ and $(5, -8)$.

Solution:

$y = mx + p$, where m is the gradient of a line, and p is the y-intercept of the line.

Point 1 consists a coordinate of (x_1, y_1), which is $(1, 7)$, and Point 2 consists a coordinate of (x_2, y_2), which is $(5, -8)$,

$m = \frac{y_2 - y_1}{x_2 - x_1}$

$m = \frac{-8-7}{5-1} = -\frac{15}{4}$

$y = -\frac{15}{4}x + b$

Sub $x = 1$ and $y = 7$ to the line equation above

$7 = -\frac{15}{4}(1) + b$

$b = \frac{43}{4}$

Hence, $y = -\frac{15}{4}x + \frac{43}{4}$.

$\frac{15}{4}x + y - \frac{43}{4} = 0$.

$15\tilde{x} + 4\tilde{y} - 43 = 0$.

Finally, the line in homogeneous coordinates is $15\tilde{x} + 4\tilde{y} - 43 = 0$.

3. [2 points] Find a vector that is a homogeneous representation of the line that passes through the points (1, 5) and (−3, 2).

Solution:

$y = mx + p$, where m is the gradient of a line, and p is the y-intercept of the line.

Point 1 consists a coordinate of (x_1, y_1), which is $(1, 5)$, and Point 2 consists a coordinate of (x_2, y_2), which is $(-3, 2)$,

$m = \frac{y_2 - y_1}{x_2 - x_1}$

$m = \frac{2-5}{-3-1} = \frac{3}{4}$

$y = \frac{3}{4}x + b$

Sub $x = 1$ and $y = 5$ to the line equation above

$5 = \frac{3}{4}(1) + b$

$b = \frac{17}{4}$

Hence, $y = \frac{3}{4}x + \frac{17}{4}$.

$-\frac{3}{4}x + y - \frac{17}{4} = 0$.

$3\tilde{x} - 4\tilde{y} + 17 = 0$.

Finally, the vector that is a homogeneous representation of the line is $\begin{bmatrix} 3 \\ -4 \\ 17 \end{bmatrix}$.

4. [5 points] Consider the intersection of two lines l_1 and l_2. Prove that the homogeneous point of intersection, x, of two homogeneous lines l_1 and l_2 is $x = l_1 \times l_2$, where $\times$ stands for the vector (or cross) product.

Solution:

Consider a 2D point, $\boldsymbol{x}$ is on both homogeneous line l_1 and homogeneous line l_2, then:

$l_1^T\boldsymbol{x} = 0$ and $l_2^T\boldsymbol{x} = 0$ based on the proof from Problem 1.1, where $\boldsymbol{x}$ is the intersection point of both homogeneous line l_1 and homogeneous line l_2

Since the cross product of l_1 and l_2 produces a vector that is normal to both homogeneous line l_1 and homogeneous line l_2, then:

$x = l_1 \times l_2$, where $\boldsymbol{x}$ is the intersection point of both homogeneous line l_1 and homogeneous line l_2

1.3 Problem 2: Image Formation and Rigid Body Transformations (17 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 $\widetilde{\boldsymbol{X}}_1$ = [-3 -6 4]^T, $\widetilde{\boldsymbol{X}}_2$ = [3 -6 4]^T, $\widetilde{\boldsymbol{X}}_3$ = [3 6 4]^T, $\widetilde{\boldsymbol{X}}_4$ = [-3 6 4]^T in the world coordinate frame. First, recall the following formula for rigid body transformation

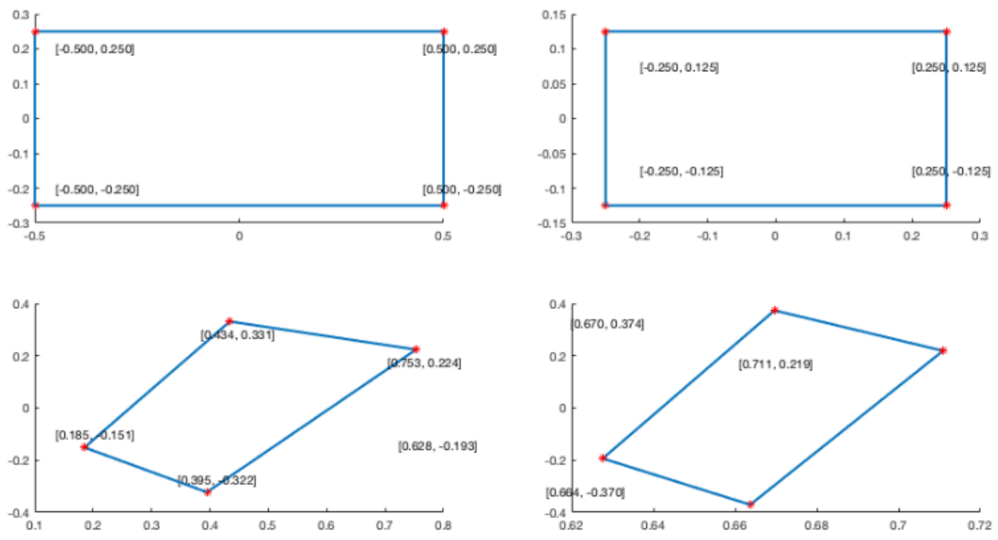
$$\widetilde{\boldsymbol{X}}_{cam} = R\widetilde{\boldsymbol{X}} + \boldsymbol{t}$$

Where $\widetilde{\boldsymbol{X}}_{cam}$ is the point coordinate in the camera coordinate system. $\widetilde{\boldsymbol{X}}$ is a point in the world coordinate frame, and R and $\boldsymbol{t}$ are the rotation and translation that transform points from the world coordinate frame to the camera coordinate frame. Together, R and $\boldsymbol{t}$ are the *extrinsic* camera parameters. Once transformed to the camera coordinate frame, the points can be photographed using the 3×3 camera calibration matrix K, which embodies the *intrinsic* camera parameters, and the canonical projection matrix [I|$\mathbf{0}$]. Given K, R, and $\boldsymbol{t}$, the image of a point $\widetilde{\boldsymbol{X}}$ is $\boldsymbol{x} = \mathrm{K}[\mathrm{I}|\mathbf{0}]\boldsymbol{X}_{\mathrm{Cam}} = \mathrm{K}[\mathrm{R}|\boldsymbol{t}]\boldsymbol{X}$, where the homogeneous points $\boldsymbol{X}_{\mathrm{Cam}} = (\widetilde{\boldsymbol{X}}_{\mathrm{Cam}}^{\top}, 1)^{\top}$ and $\boldsymbol{X} = (\widetilde{\boldsymbol{X}}^{\top}, 1)^{\top}$. We will consider four different settings of focal length, viewing angles and camera positions below.

a). The 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 **plot__points** function (see e.g. output in figure below).



1. [No rigid body transformation]. Focal length = 1. The optical axis of the camera is aligned with the z-axis.
2. [Translation]. Focal length = 1. $\boldsymbol{t} = [0\ 0\ 2]^T$. The optical axis of the camera is aligned with the z-axis.
3. [Translation and Rotation]. Focal length = 1. R encodes a 60 degrees around the z-axis and then 30 degrees around the y-axis. $\boldsymbol{t} = [0\ 0\ 2]^T$.
4. [Translation and Rotation, long distance]. Focal length = 5. R encodes a 60 degrees around the z-axis and then 30 degrees around the y-axis. $\boldsymbol{t} = [0\ 0\ 7]^T$.

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.

For all the four cases, include a image like above. Note that the axis are the same for each row, to facilitate comparison between the two camera models. Note: the angles and offsets used to generate these plots may be different from those in the problem statement, it's just to illustrate how to report your results.

Also, Explain why you observe any distortions in the projection, if any, under this model.

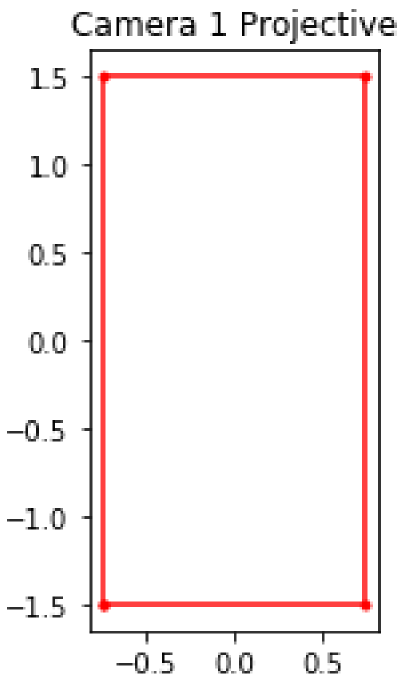
Solution:

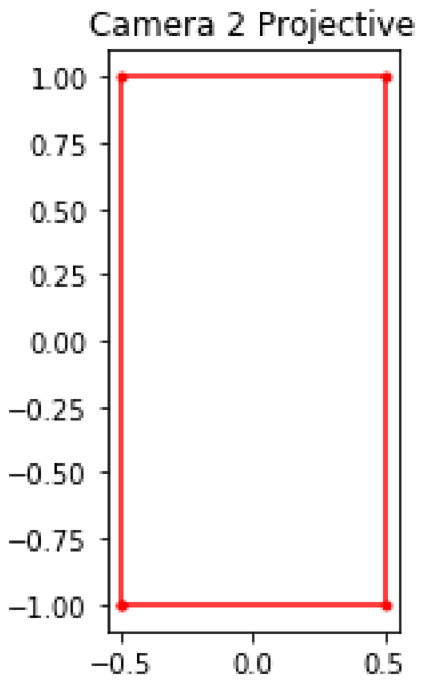
There is a little distortion on the Camera 4 Projective projection due to an increase in the focal length.

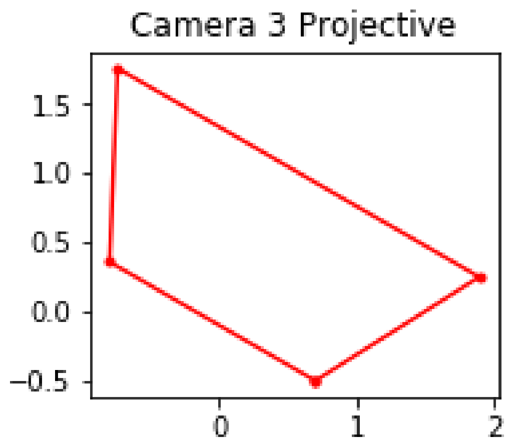
[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
import math
```

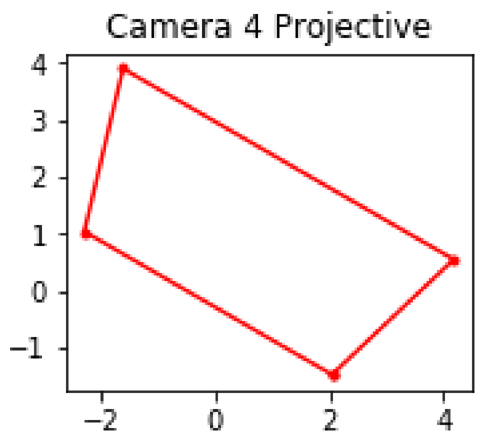
```python
# convert points from euclidean to homogeneous
def to_homog(points): #here always remember that points is a 3x4 matrix
    # write your code here

    # Determine Number of Columns (i.e., number of coordinates)
    num_Points = points.shape[1]

    # Generate an array with ones
    array_of_ONES = np.ones(num_Points)

    # Stack the points matrix with the array with ones.
    homo_points = np.vstack([points, array_of_ONES])

    return homo_points

## Check to_homog() user-defined function
#points = np.array([[-3, 3, 3, -3],[-6, -6, 6, 6],[4, 4, 4, 4]])
#print(points, points.shape)
#print(to_homog(points))


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

    # Change the matrix data type to float
    points_homog = points_homog.astype(float)

    # Extract the last row of the array.
    array_of_Ws = points_homog[-1,:]

    # Determine the number of rows of the matrix.
    num_rows = points_homog.shape[0]

    # Delete the last row of the points_homog matrix.
    euclid_points = np.delete(points_homog, num_rows-1, axis=0)

    # Divide the coordinates with its own w.
    for col in range(len(array_of_Ws)):
        for row in range(euclid_points.shape[0]):
            euclid_points[row, col] = euclid_points[row, col] / array_of_Ws[col]

    return euclid_points

## Check from_homog() user-defined function
```

```
#points_homog = np.array([[-3, 3, 3, -3],[-6, -6, 6, 6],[4, 4, 4, 4], [2, 1, 4,␣
↪3]])
#print(from_homog(points_homog))


# project 3D euclidean points to 2D euclidean
def project_points(P_int, P_ext, pts):
    # write your code here

    # From Geometrix Image Information Slide:
    # x = K [R|t] X
    # x is a 3x1 vector
    # K is a 3x3 matrix
    # [R|t] is a 3x4 matrix
    # X is a 4 x 1 vector

    # Convert the euclidean points to homogeneous points
    points_homog = to_homog(pts)

    # Multiply the Intrinsic Camera Matrix, K with the Extrinsic Camera Matrix,␣
↪[R|t]
    # Which is also known as camera projection matrix.
    P_int_N_ext =  np.dot(P_int, P_ext)

    # Multiply the camera projection matrix with the homogeneous points to␣
↪obtain 3D euclidean points.
    projected_points_homog = np.dot(P_int_N_ext, points_homog)

    # Convert 3D euclidean points to 2D euclidean points
    pts_2d = from_homog(projected_points_homog)

    #return the 2d euclidean points
    #pts_2d=np.zeros([2,1])

    return pts_2d


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

# given the focal length, compute the intrinsic camera matrix

def intrinsic_cam_mat(f):
    """
    K = [f 0 0
         0 f 0
         0 0 1]
    """
    # write your code here
    int_cam_mat = np.eye(3)

    # Generate the intrinsic camera matrix.
    for i in range(int_cam_mat.shape[0]-1):
        int_cam_mat[i,i] = f

    return int_cam_mat

## Check intrinsic_cam_mat() user-defined function.
#print(intrinsic_cam_mat(3))

# Compute the extrinsic camera matrix
def extrinsic_cam_mat(angles, t):
    """
        ext_cam_mat = [R|t]
    """
    # write your code here

    # Initialize ext_cam_mat
    R = np.eye(4)

    # angles is a 1x3 array, [z_angle, y_angle, x_angle]
    # Units are in degree.
    z_angle = angles[0]
    y_angle = angles[1]
    x_angle = angles[2]

    # Rotation about z-axis, R_z
    R_z = np.array([[np.cos(np.deg2rad(z_angle)), -np.sin(np.deg2rad(z_angle)),␣
↪0],\
                    [np.sin(np.deg2rad(z_angle)), np.cos(np.deg2rad(z_angle)),␣
↪0],\
                    [0, 0, 1]])

    # Rotation about y-axis, R_y
    R_y = np.array([[np.cos(np.deg2rad(y_angle)), 0, np.sin(np.
↪deg2rad(y_angle))],\
                    [0, 1, 0],\
                    [-np.sin(np.deg2rad(y_angle)), 0, np.cos(np.
↪deg2rad(y_angle))]])

    # Rotation about x-axis, R_x
    R_x = np.array([[1, 0, 0],\
                    [0, np.cos(np.deg2rad(x_angle)), -np.sin(np.
↪deg2rad(x_angle))],\
                    [0, np.sin(np.deg2rad(x_angle)), np.cos(np.
↪deg2rad(x_angle))]])

    # Rotation matrix, 3x3
    R_mat = np.dot(np.dot(R_z, R_y), R_x)

    # Round-up to 6 decimal points for R_mat because, for example, np.cos(np.pi/
↪2) = 6.123233995736766e-17 which is 0.
    R_mat = np.around(R_mat, decimals = 6)

    # Generate extrinsic matrix of the camera.
    R[:3,:3] = R_mat
    R[:3,3] = t

    # 3 x 4 Identity Matrix
    Iden34 = np.array([[1, 0, 0, 0], [0, 1, 0, 0],[0, 0, 1, 0]])

    # Determine matrix [R|t]
    ext_cam_mat = np.dot(Iden34,R)

    return ext_cam_mat

## Check extrinsic_cam_mat(angles, t) user-defined function
#print(extrinsic_cam_mat([90, 90, 90], [2, 2, 2]))


def camera1():
    """
    replace with your code
    """
    # write your code here
    P_int_proj = intrinsic_cam_mat(1)
    P_ext = extrinsic_cam_mat([0, 0, 0], [0, 0, 0])

    return P_int_proj, P_ext

def camera2():
    """
    replace with your code
    """
    # write your code here
    P_int_proj = intrinsic_cam_mat(1)
    P_ext = extrinsic_cam_mat([0, 0, 0], [0, 0, 2])
    return P_int_proj, P_ext

def camera3():
    """
    replace with your code
    """
    # write your code here
    P_int_proj = intrinsic_cam_mat(1)
    P_ext = extrinsic_cam_mat([60, 30, 0], [0, 0, 2])

    return P_int_proj, P_ext

def camera4():
    """
    replace with your code
    """
    # write your code here
    P_int_proj = intrinsic_cam_mat(5)
    P_ext = extrinsic_cam_mat([60, 30, 0], [0, 0, 7])

    return P_int_proj, P_ext


#######################################################
# test code. Do not modify
#######################################################

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

def main():
    point1 = np.array([[-3,-6,4]]).T
    point2 = np.array([[3,-6,4]]).T
    point3 = np.array([[3,6,4]]).T
    point4 = np.array([[-3,6,4]]).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(1, 2, 1)
        plot_points(project_points(P_int_proj, P_ext, points), title='Camera %d␣
↪Projective'%(i+1), axis=[-1,1,-1,1])
        plt.show()

main()
```




Camera 2 Projective
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−0.5
0.0
0.5


Camera 3 Projective
1.5
1.0
0.5
0.0
−0.5
0
1
2



1.4 Problem 3: Surface Rendering (18 pts)

In this portion of the assignment we will be exploring different methods of approximating local reflectance 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 reflectance models. For the sake of simplicity, multiple reflections of light rays, and occlusion of light rays due to object/scene can be ignored.

1.4.1 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.

1.4.2 Lambertian Reflectance

One of the simplest models available to render 3D objections with reflectance is the Lambertian model. This model finds the apparent brightness to an observer using the direction of the light source **L** and the normal vector on the surface of the object **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, **C** and I_l are the the color and intensity of the light source respectively.

1.4.3 Phong Reflectance

One major drawback of Lambertian reflectance is that it only considers the diffuse light in its calculation of brightness intensity. One other major component to reflectance 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 reflectance 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.

1.4.4 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	$(-\frac{1}{2}, \frac{1}{2}, \frac{1}{2})^\top$	$(1, 1, 1)$
2	$(1, 0, 0)^\top$	$(1, .5, 1)$

Table 2: Material Coefficients

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

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

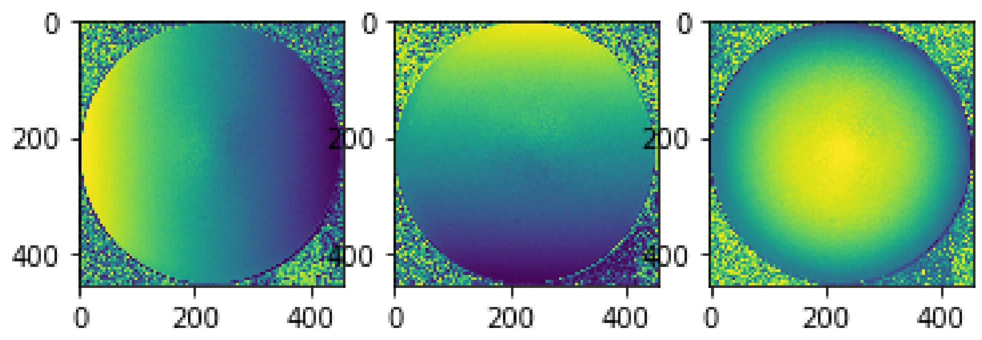
In this first part, you are required to work with 2 images, one of a sphere and the other one of a pear. The pickle file normals.pickle is a list consisting of 4 numpy matrices which are
1) Normal Vectors for the sphere with specularities removed (Diffuse component)
2) Normal Vector for the sphere
3) 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.

[2]:
```
def plot_normals(diffuse_normals, original_normals):
    # Stride in the plot, you may want to adjust it to different images
    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()
```

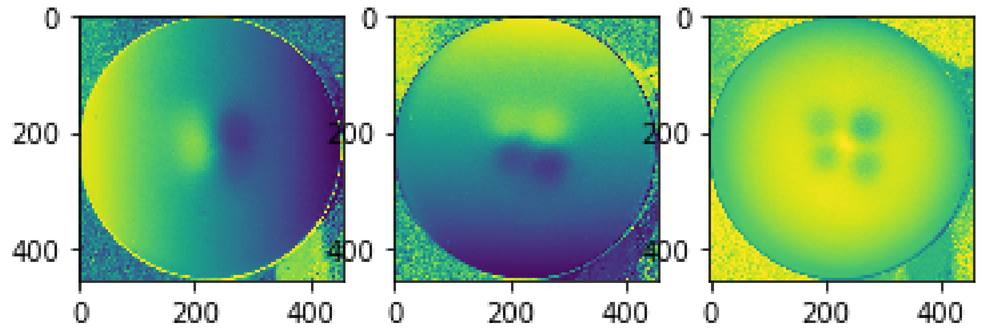
[3]:
```
#Plot the normals for the sphere and pear for both the normal and diffuse␣
↪components.
#1 : Load the different normals
```

```
# LOAD HERE
import pandas as pd

# Load normals pickle file
normals_pickle = pd.read_pickle("normals.pkl")
# Extract Normal Vectors for the sphere with specularities removed (Diffuse␣
↪component).
Sphere_Diffuse_NVs = normals_pickle[0]
# Extract Normal Vector for the sphere.
Sphere_Original_NVs = normals_pickle[1]
# Extract Normal Vectors for the pear with specularities removed (Diffuse␣
↪component)
Pear_Diffuse_NVs = normals_pickle[2]
# Extract Normal Vector for the pear.
Pear_Original_NVs = normals_pickle[3]


#2 : Plot the normals using plot_normals
#What do you observe? What are the differences between the diffuse component␣
↪and the original images shown?
# The surfaces of the sphere and pear from the diffuse images were smoothen␣
↪compare to the surfaces of the sphere and
# pear from the original images. Besides that, on the diffuse images of the␣
↪sphere and pear, the objects were darken
# as the surface normals turn away from the light source as compared to the␣
↪original images of the sphere and pear,
# which also have the same light source.

#PLOT HERE
#Plot normals for the sphere including the original and diffuse versions
plot_normals(Sphere_Diffuse_NVs, Sphere_Original_NVs)
#Plot normals for the pear including the original and diffuse versions
plot_normals(Pear_Diffuse_NVs, Pear_Original_NVs)
```

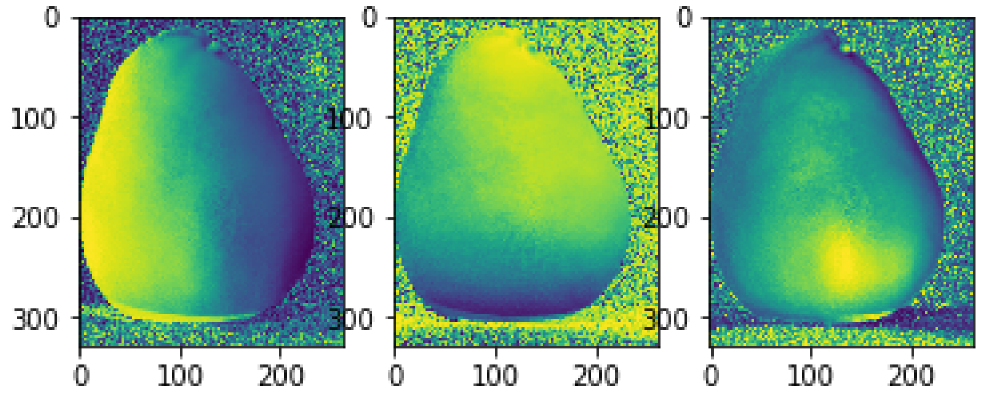
```
Normals:
Diffuse
```



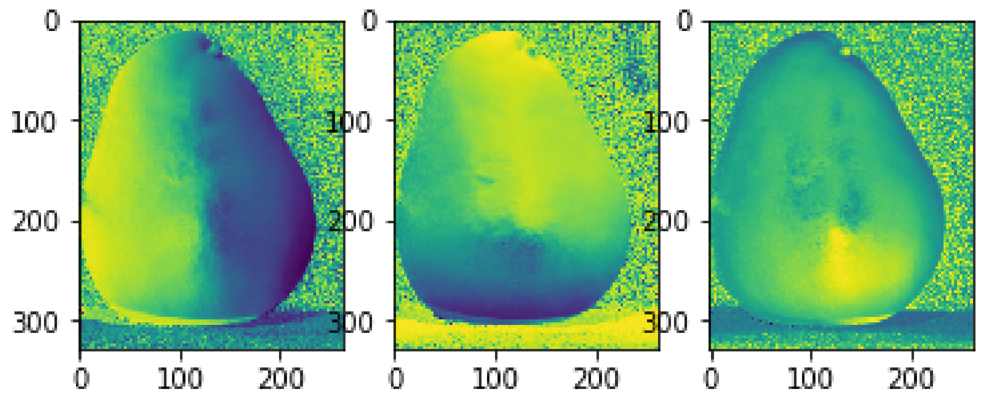
```
Original
```



```
Normals:
Diffuse
```



```
Original
```



1.4.6 Part 2. Lambertian model [6 pts]

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

[4]:
```python
def normalize(img):
    assert img.shape[2] == 3
    maxi = img.max()
    mini = img.min()
    return (img - mini)/(maxi-mini)
```

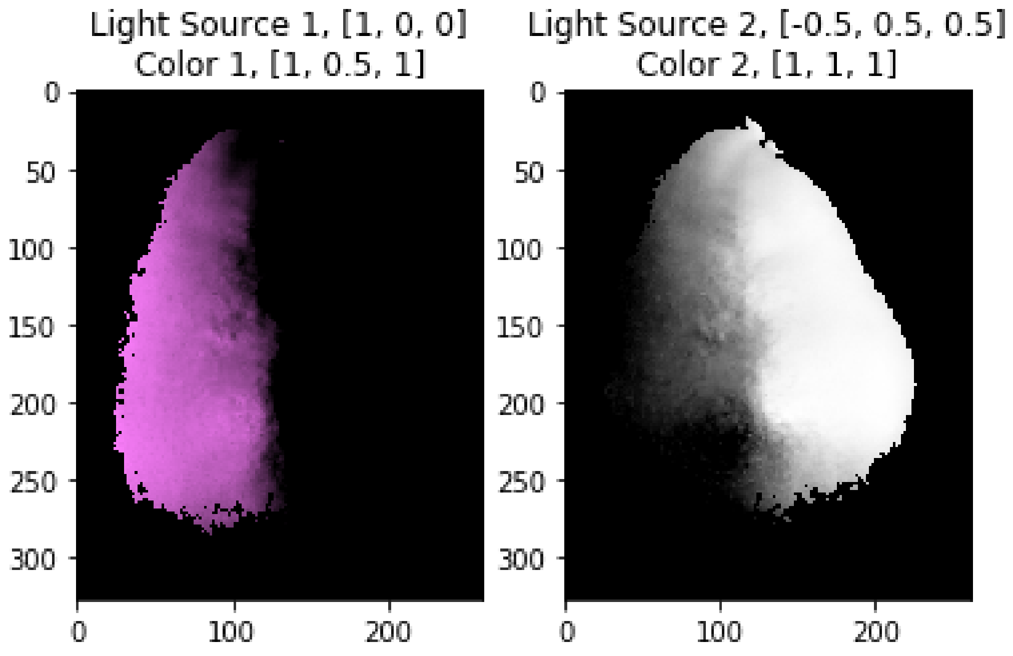
[5]:
```python
def lambertian(normals, lights, color, intensity, mask):
    '''Your implementation'''
    # Lambertian Reflectance Equation
    # I_d = L * N(I_l * C)
    # where,
    # I_d - The brightness intensity at a given point on an object’s surface
    # L - The direction of the light source (3 x 1)
    # N - The normal vector on the surface of the object (1 x 3)
    # I_l - Intensity of the light source (scalar)
    # C - Color (1 x 3)

    # Initialize and define the image array
    image = np.ones((normals.shape[0], normals.shape[1], 3))

    # Reshape mask from H x W to H x W x 1
    mask = mask.reshape(mask.shape[0], mask.shape[1], 1)

    # Calculate the Lambertian Reflectance Intensity
    for height in range(normals.shape[0]):
        for width in range(normals.shape[1]):
            # Get the normal vector
            norm_vec = normals[height, width, :]
            # Reshape the normal vector
            norm_vec = norm_vec.reshape(norm_vec.shape[0], 1)
            # Get the Lambertian Reflectance Intensity
            image[height, width, :] = np.dot(lights.T, norm_vec) * intensity *␣
↪color

    # Turn negative values to zero.
    image[image < 0] = 0

    # Apply mask to the render image.
    image = image * mask

    # Normalize the image
    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.

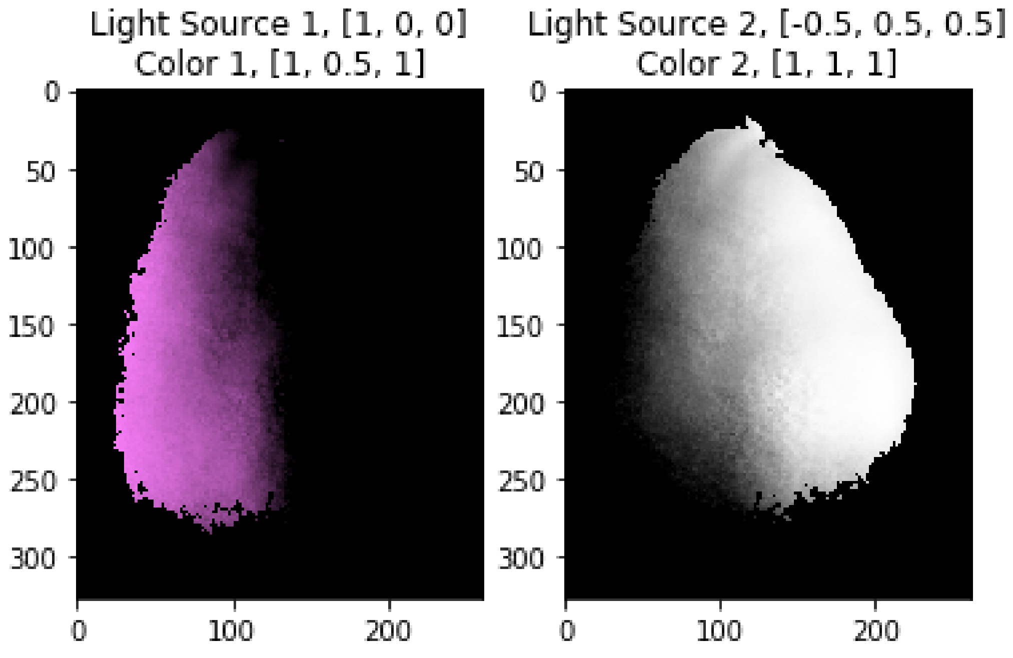
[6]:
```
# Load the masks for the sphere and pear
# LOAD HERE
# Load normals pickle file
masks_pickle = pd.read_pickle("masks.pkl")
# Extract Mask for the sphere
Sphere_Masks = masks_pickle[0]
# Extract Mask for the pear.
Pear_Masks = masks_pickle[1]



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

# Original Pear's Normals
# Light Source 1 and Color 1
Lam_Pear_Original_L1_C1 = lambertian(Pear_Original_NVs, dirn1, color1,␣
↪intensity_LS, Pear_Masks)
# Light Source 2 and Color 2
```

```
Lam_Pear_Original_L2_C2 = lambertian(Pear_Original_NVs, dirn2, color2,␣
 ↪intensity_LS, Pear_Masks)

# Diffuse Pear's Normals
# Light Source 1 and Color 1
Lam_Pear_Diffuse_L1_C1 = lambertian(Pear_Diffuse_NVs, dirn1, color1,␣
 ↪intensity_LS, Pear_Masks)
# Light Source 2 and Color 2
Lam_Pear_Diffuse_L2_C2 = lambertian(Pear_Diffuse_NVs, dirn2, color2,␣
 ↪intensity_LS, Pear_Masks)


#Display the rendering results for pear for both diffuse and for both the light␣
 ↪sources
def plot_Lambertian(original_L1_C1, original_L2_C2, diffuse_L1_C1,␣
 ↪diffuse_L2_C2, obj_Name):

    print(obj_Name + " Lambertian Plots:")

    print("Original")
    figure = plt.figure()
    ax1 = figure.add_subplot(121)
    plt.title("Light Source 1, [1, 0, 0]\nColor 1, [1, 0.5, 1]")
    ax1.imshow(original_L1_C1)
    ax2 = figure.add_subplot(122)
    plt.title("Light Source 2, [-0.5, 0.5, 0.5]\nColor 2, [1, 1, 1]")
    ax2.imshow(original_L2_C2)
    plt.show()

    print("Diffuse")
    figure = plt.figure()
    ax1 = figure.add_subplot(121)
    plt.title("Light Source 1, [1, 0, 0]\nColor 1, [1, 0.5, 1]")
    ax1.imshow(diffuse_L1_C1)
    ax2 = figure.add_subplot(122)
    plt.title("Light Source 2, [-0.5, 0.5, 0.5]\nColor 2, [1, 1, 1]")
    ax2.imshow(diffuse_L2_C2)
    plt.show()

plot_Lambertian(Lam_Pear_Original_L1_C1, Lam_Pear_Original_L2_C2,␣
 ↪Lam_Pear_Diffuse_L1_C1, Lam_Pear_Diffuse_L2_C2,"Pear")
```

```
Pear Lambertian Plots:
Original
```

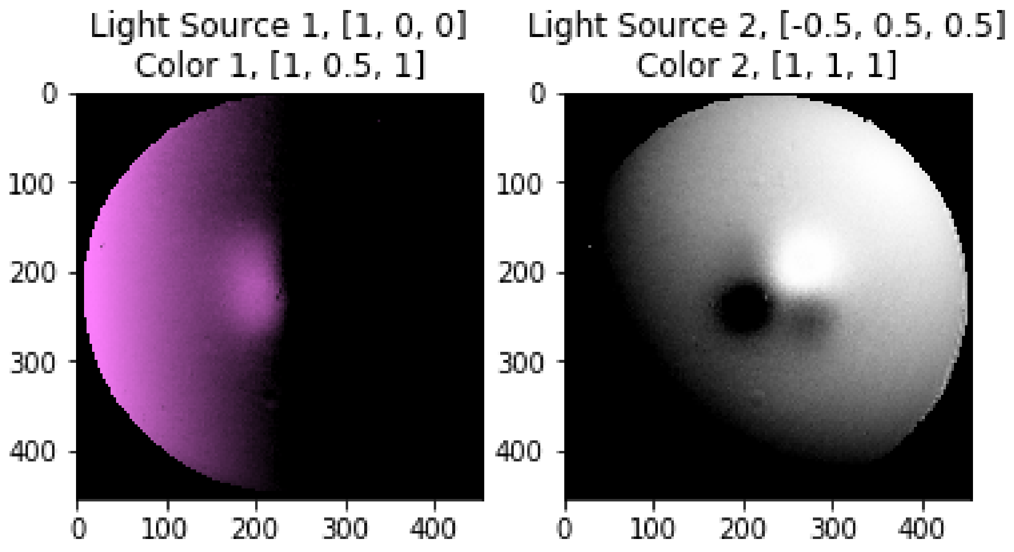

Light Source 1, [1, 0, 0]
Color 1, [1, 0.5, 1]
Light Source 2, [-0.5, 0.5, 0.5]
Color 2, [1, 1, 1]
0
50
100
150
200
250
300
0
100
200

```
Diffuse
```

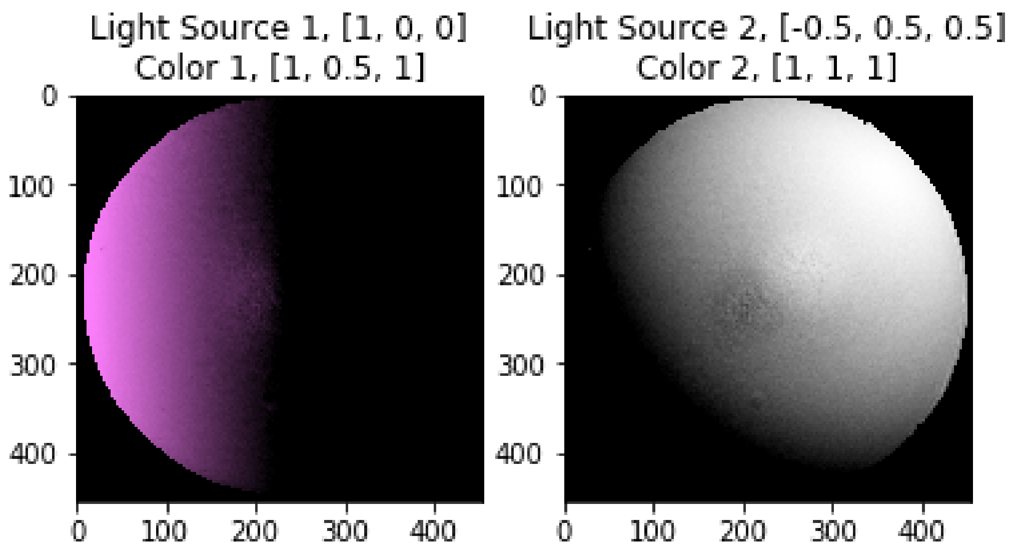

Light Source 1, [1, 0, 0]
Color 1, [1, 0.5, 1]
Light Source 2, [-0.5, 0.5, 0.5]
Color 2, [1, 1, 1]
0
50
100
150
200
250
300
0
100
200

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

# Original Sphere's Normals
# Light Source 1 and Color 1
Lam_Sphere_Original_L1_C1 = lambertian(Sphere_Original_NVs, dirn1, color1,␣
 ↪intensity_LS, Sphere_Masks)
# Light Source 2 and Color 2
Lam_Sphere_Original_L2_C2 = lambertian(Sphere_Original_NVs, dirn2, color2,␣
 ↪intensity_LS, Sphere_Masks)

# Diffuse Sphere's Normals
# Light Source 1 and Color 1
Lam_Sphere_Diffuse_L1_C1 = lambertian(Sphere_Diffuse_NVs, dirn1, color1,␣
 ↪intensity_LS, Sphere_Masks)
# Light Source 2 and Color 2
Lam_Sphere_Diffuse_L2_C2 = lambertian(Sphere_Diffuse_NVs, dirn2, color2,␣
 ↪intensity_LS, Sphere_Masks)


#Display the rendering results for sphere for both diffuse and for both the␣
 ↪light sources
plot_Lambertian(Lam_Sphere_Original_L1_C1, Lam_Sphere_Original_L2_C2,␣
 ↪Lam_Sphere_Diffuse_L1_C1, Lam_Sphere_Diffuse_L2_C2,"Sphere")
```

```
Sphere Lambertian Plots:
Original
```



```
Diffuse
```



1.4.7 Part 3. Phong model [8 pts]

Please fill in your implementation for the Phong model below.

```
[8]: def phong(normals, lights, color, material, view, mask):
         '''Your implementation'''
         # Phong Reflectance Equation
         #      =    *    +    {  * (  ) *   ,  +    * (     )^ *   , }
         #    = 2 *   * (     ) -
         # where,
         #      - Phong Reflectance Intensity
         #       - Ambient Reflection Constant
         #       - Diffuse Reflection Constant
         #       - Specular Reflection Constant
         #       - Phong Constant (Shininess of an Object)
         #       - Observer's Viewing Direction, (0,0,1)
         #       - Known Material Parameters
         #       - Normal Vector of the Object Surface
         #        - The color and intensity of the ambient lighting
         #   ,   - The color values for the diffuse light of the  th light source
         #   ,   - The color values for the specular light of the  th light source

         # Extract the material constants from the material array, [ ,  , ].
         k_d = material[0, 0]
```

```
    k_s = material[0, 1]
    alpha_const = material[0, 2]

    # Initialize and define the image array
    image = np.ones((normals.shape[0], normals.shape[1], 3))

    # Reshape mask from H x W to H x W x 1
    mask = mask.reshape(mask.shape[0], mask.shape[1], 1)

    # Calculate the Phong Reflectance Intensity
    for height in range(normals.shape[0]):
        for width in range(normals.shape[1]):

            # Initialize I_phong for each and every pixels
            I_phong = np.zeros((1,3))

            for m in range(len(lights)):

                # Get the normal vector
                norm_vec = normals[height, width, :]

                # Reshape the normal vector
                norm_vec = norm_vec.reshape(norm_vec.shape[0], 1)

                # Calculate R for the mth light source (1 x 3).
                R_m = 2 * norm_vec.T * np.dot(lights[m], norm_vec) - lights[m]

                # Calculate I_phong_m
                I_phong_m = k_d * np.dot(lights[m], norm_vec) * color[m] + \
                            k_s * np.power(np.dot(R_m, view),alpha_const) *␣
↪color[m]

                # Calculate sum of I_phong_m
                I_phong = I_phong + I_phong_m

            # Get the I_phong for the entire pixels
            image[height, width, :] = I_phong

    # Turn negative values to zero.
    image[image < 0] = 0

    # Apply mask to the render image.
    image = image * mask

    # Normalize the image
    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.

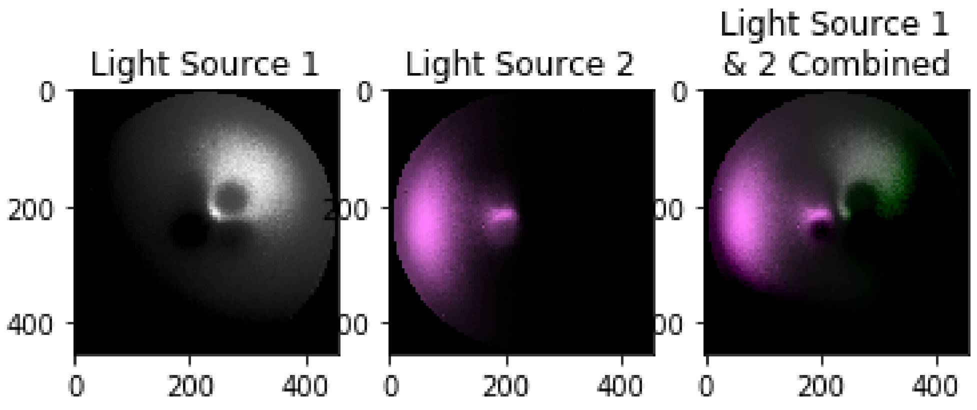
[9]:
```
# Output the rendering results for sphere
view =  np.array([[0],[0],[1]])
material = np.array([[0.1,0.5,5],[0.5,0.1,5],[0.5,0.5,10]])
lightcol1 =  np.array([[-1.0/2,1],[1.0/2,1],[1.0/2,1]])
lightcol2 = np.array([[1,1],[0,0.5],[0,1]])

# Extract the light source vector 1 (1 x 3) and color vector 1 (1 x 3)
light_dir_1 = lightcol1[:,0].T.reshape(1, lightcol1[:, 0].T.shape[0])
color_1 = lightcol1[:,1].T.reshape(1, lightcol1[:, 1].T.shape[0])

# Extract the light source vector 2 (1 x 3) and color vector 2 (1 x 3)
light_dir_2 = lightcol2[:,0].T.reshape(1, lightcol2[:, 0].T.shape[0])
color_2 = lightcol2[:,1].T.reshape(1, lightcol2[:, 1].T.shape[0])

# Split the material matrix into 3 (1 x 3) arrays
material_1 = material[0,:].T.reshape(1,material[0,:].T.shape[0])
material_2 = material[1,:].T.reshape(1,material[1,:].T.shape[0])
material_3 = material[2,:].T.reshape(1,material[2,:].T.shape[0])


### Original Sphere's Normals
## Material 1
# Light Source 1 and Color 1
Phong_Sphere_Original_L1_C1_M1 = phong(Sphere_Original_NVs, light_dir_1,␣
↪color_1, material_1, view, Sphere_Masks)
# Light Source 2 and Color 2
Phong_Sphere_Original_L2_C2_M1 = phong(Sphere_Original_NVs, light_dir_2,␣
↪color_2, material_1, view, Sphere_Masks)
# Combine Both Light Source 1 and 2 including Color 1 and Color 2
Phong_Sphere_Original_L12_C12_M1 = phong(Sphere_Original_NVs, [light_dir_1,␣
↪light_dir_2], [color_1, color_2], material_1, view, Sphere_Masks)

## Material 2
# Light Source 1 and Color 1
Phong_Sphere_Original_L1_C1_M2 = phong(Sphere_Original_NVs, light_dir_1,␣
↪color_1, material_2, view, Sphere_Masks)
# Light Source 2 and Color 2
Phong_Sphere_Original_L2_C2_M2 = phong(Sphere_Original_NVs, light_dir_2,␣
↪color_2, material_2, view, Sphere_Masks)
# Combine Both Light Source 1 and 2 including Color 1 and Color 2
```

```
Phong_Sphere_Original_L12_C12_M2 = phong(Sphere_Original_NVs, [light_dir_1,␣
 ↪light_dir_2], [color_1, color_2], material_2, view, Sphere_Masks)

## Material 3
# Light Source 1 and Color 1
Phong_Sphere_Original_L1_C1_M3 = phong(Sphere_Original_NVs, light_dir_1,␣
 ↪color_1, material_3, view, Sphere_Masks)
# Light Source 2 and Color 2
Phong_Sphere_Original_L2_C2_M3 = phong(Sphere_Original_NVs, light_dir_2,␣
 ↪color_2, material_3, view, Sphere_Masks)
# Combine Both Light Source 1 and 2 including Color 1 and Color 2
Phong_Sphere_Original_L12_C12_M3 = phong(Sphere_Original_NVs, [light_dir_1,␣
 ↪light_dir_2], [color_1, color_2], material_3, view, Sphere_Masks)


### Diffuse Sphere's Normals
## Material 1
# Light Source 1 and Color 1
Phong_Sphere_Diffuse_L1_C1_M1 = phong(Sphere_Diffuse_NVs, light_dir_1, color_1,␣
 ↪material_1, view, Sphere_Masks)
# Light Source 2 and Color 2
Phong_Sphere_Diffuse_L2_C2_M1 = phong(Sphere_Diffuse_NVs, light_dir_2, color_2,␣
 ↪material_1, view, Sphere_Masks)
# Combine Both Light Source 1 and 2 including Color 1 and Color 2
Phong_Sphere_Diffuse_L12_C12_M1 = phong(Sphere_Diffuse_NVs, [light_dir_1,␣
 ↪light_dir_2], [color_1, color_2], material_1, view, Sphere_Masks)

## Material 2
# Light Source 1 and Color 1
Phong_Sphere_Diffuse_L1_C1_M2 = phong(Sphere_Diffuse_NVs, light_dir_1, color_1,␣
 ↪material_2, view, Sphere_Masks)
# Light Source 2 and Color 2
Phong_Sphere_Diffuse_L2_C2_M2 = phong(Sphere_Diffuse_NVs, light_dir_2, color_2,␣
 ↪material_2, view, Sphere_Masks)
# Combine Both Light Source 1 and 2 including Color 1 and Color 2
Phong_Sphere_Diffuse_L12_C12_M2 = phong(Sphere_Diffuse_NVs, [light_dir_1,␣
 ↪light_dir_2], [color_1, color_2], material_2, view, Sphere_Masks)

## Material 3
# Light Source 1 and Color 1
Phong_Sphere_Diffuse_L1_C1_M3 = phong(Sphere_Diffuse_NVs, light_dir_1, color_1,␣
 ↪material_3, view, Sphere_Masks)
# Light Source 2 and Color 2
Phong_Sphere_Diffuse_L2_C2_M3 = phong(Sphere_Diffuse_NVs, light_dir_2, color_2,␣
 ↪material_3, view, Sphere_Masks)
# Combine Both Light Source 1 and 2 including Color 1 and Color 2
```

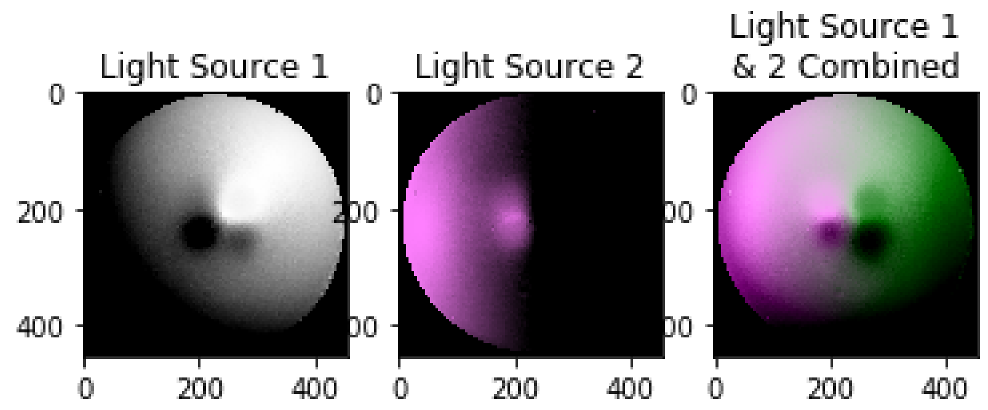
```
Phong_Sphere_Diffuse_L12_C12_M3 = phong(Sphere_Diffuse_NVs, [light_dir_1,␣
↪light_dir_2], [color_1, color_2], material_3, view, Sphere_Masks)


#Display rendered results for sphere for all materials and light sources and␣
↪combination of light sources
def plot_Phong(object_L1_C1, object_L2_C2, object_L12_C12, obj_Name,␣
↪material_num, Ori_or_Dif):

    print(obj_Name + " Phong Model Plots:")

    # Print original or diffuse
    print(Ori_or_Dif)

    # Print which material number
    if material_num == 1:
        print("Material 1, [  = 0.1,   = 0.5,   = 5]" )
    elif material_num == 2:
        print("Material 2, [  = 0.5,   = 0.1,   = 5]" )
    elif material_num == 3:
        print("Material 3, [  = 0.5,   = 0.5,   = 10]" )

    print("Light Source 1, [-0.5, 0.5, 0.5], Color 1, [1, 1, 1]")
    print("Light Source 2, [1, 0, 0], Color 2, [1, 0.5, 1]")

    figure = plt.figure()
    ax1 = figure.add_subplot(131)
    plt.title("Light Source 1")
    ax1.imshow(object_L1_C1)
    ax2 = figure.add_subplot(132)
    plt.title("Light Source 2")
    ax2.imshow(object_L2_C2)
    ax3 = figure.add_subplot(133)
    plt.title("Light Source 1\n& 2 Combined")
    ax3.imshow(object_L12_C12)
    plt.show()

### Original Sphere's Normals
## Material 1
plot_Phong(Phong_Sphere_Original_L1_C1_M1, Phong_Sphere_Original_L2_C2_M1,␣
↪Phong_Sphere_Original_L12_C12_M1, "Sphere", 1, "Original")
## Material 2
plot_Phong(Phong_Sphere_Original_L1_C1_M2, Phong_Sphere_Original_L2_C2_M2,␣
↪Phong_Sphere_Original_L12_C12_M2, "Sphere", 2, "Original")
## Material 3
plot_Phong(Phong_Sphere_Original_L1_C1_M3, Phong_Sphere_Original_L2_C2_M3,␣
↪Phong_Sphere_Original_L12_C12_M3, "Sphere", 3, "Original")
```

```
### Diffuse Sphere's Normals
## Material 1
plot_Phong(Phong_Sphere_Diffuse_L1_C1_M1, Phong_Sphere_Diffuse_L2_C2_M1,␣
↪Phong_Sphere_Diffuse_L12_C12_M1, "Sphere", 1, "Diffuse")
## Material 2
plot_Phong(Phong_Sphere_Diffuse_L1_C1_M2, Phong_Sphere_Diffuse_L2_C2_M2,␣
↪Phong_Sphere_Diffuse_L12_C12_M2, "Sphere", 2, "Diffuse")
## Material 3
plot_Phong(Phong_Sphere_Diffuse_L1_C1_M3, Phong_Sphere_Diffuse_L2_C2_M3,␣
↪Phong_Sphere_Diffuse_L12_C12_M3, "Sphere", 3, "Diffuse")
```

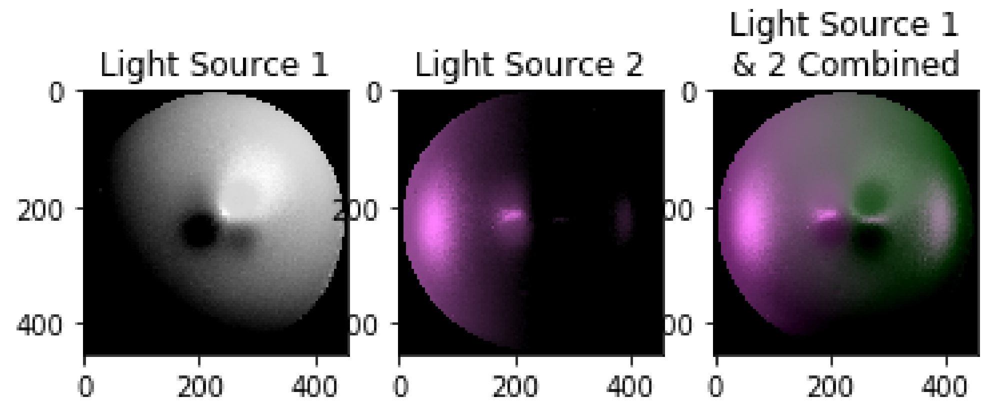
```
Sphere Phong Model Plots:
Original
Material 1, [  = 0.1,    = 0.5,   = 5]
Light Source 1, [-0.5, 0.5, 0.5], Color 1, [1, 1, 1]
Light Source 2, [1, 0, 0], Color 2, [1, 0.5, 1]
```



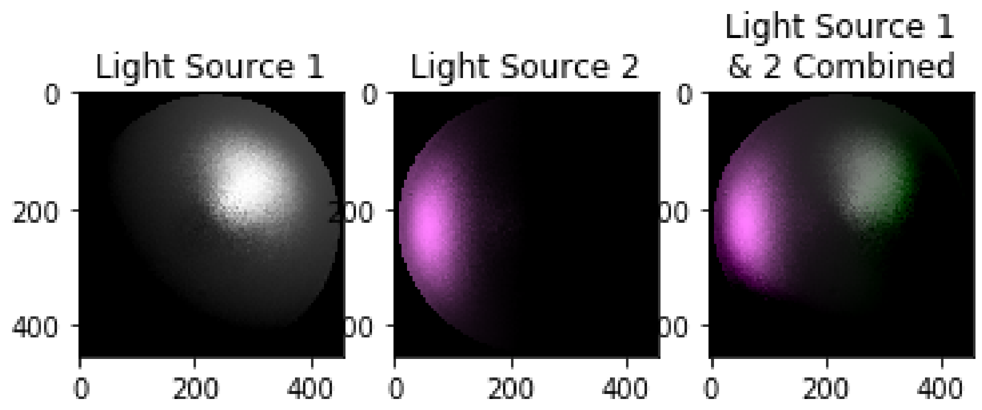
```
Sphere Phong Model Plots:
Original
Material 2, [  = 0.5,    = 0.1,   = 5]
Light Source 1, [-0.5, 0.5, 0.5], Color 1, [1, 1, 1]
Light Source 2, [1, 0, 0], Color 2, [1, 0.5, 1]
```



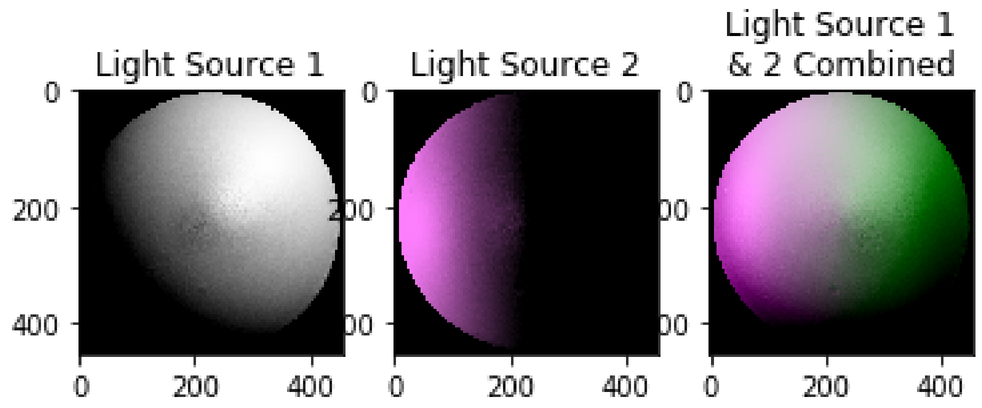
```
Sphere Phong Model Plots:
Original
Material 3, [  = 0.5,   = 0.5,   = 10]
Light Source 1, [-0.5, 0.5, 0.5], Color 1, [1, 1, 1]
Light Source 2, [1, 0, 0], Color 2, [1, 0.5, 1]
```



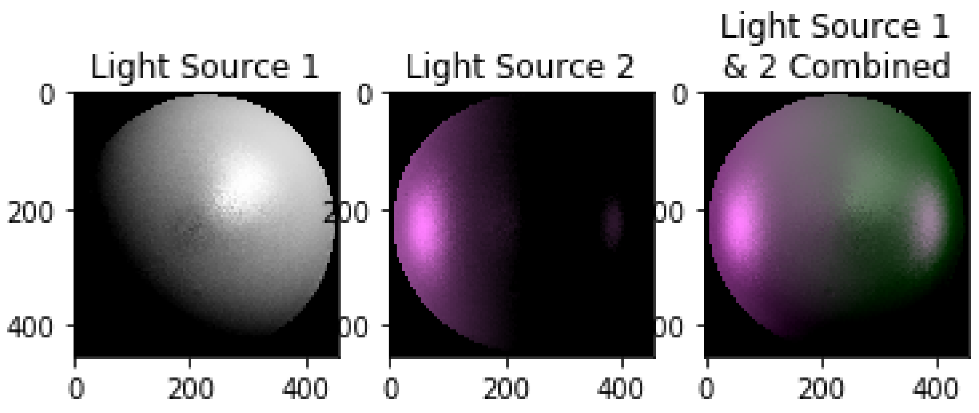
```
Sphere Phong Model Plots:
Diffuse
Material 1, [  = 0.1,   = 0.5,   = 5]
Light Source 1, [-0.5, 0.5, 0.5], Color 1, [1, 1, 1]
Light Source 2, [1, 0, 0], Color 2, [1, 0.5, 1]
```



```
Sphere Phong Model Plots:
Diffuse
Material 2, [  = 0.5,   = 0.1,   = 5]
Light Source 1, [-0.5, 0.5, 0.5], Color 1, [1, 1, 1]
Light Source 2, [1, 0, 0], Color 2, [1, 0.5, 1]
```



```
Sphere Phong Model Plots:
Diffuse
Material 3, [  = 0.5,   = 0.5,   = 10]
Light Source 1, [-0.5, 0.5, 0.5], Color 1, [1, 1, 1]
Light Source 2, [1, 0, 0], Color 2, [1, 0.5, 1]
```

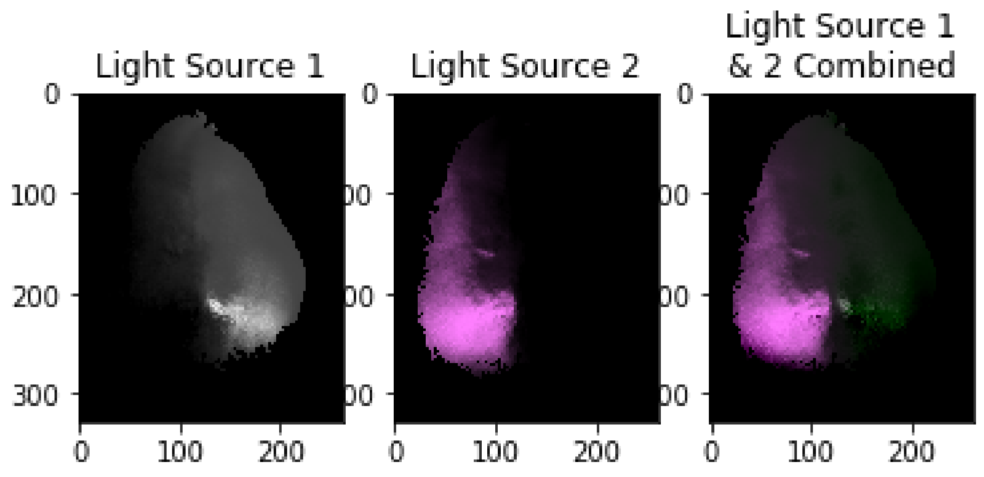


```
[10]: # Output the rendering results for the pear.
view =  np.array([[0],[0],[1]])
material = np.array([[0.1,0.5,5],[0.5,0.1,5],[0.5,0.5,10]])
lightcol1 =  np.array([[-1.0/2,1],[1.0/2,1],[1.0/2,1]])
lightcol2 = np.array([[1,1],[0,0.5],[0,1]])

### Original Pear's Normals
## Material 1
# Light Source 1 and Color 1
Phong_Pear_Original_L1_C1_M1 = phong(Pear_Original_NVs, light_dir_1, color_1,␣
 ↪material_1, view, Pear_Masks)
# Light Source 2 and Color 2
Phong_Pear_Original_L2_C2_M1 = phong(Pear_Original_NVs, light_dir_2, color_2,␣
 ↪material_1, view, Pear_Masks)
# Combine Both Light Source 1 and 2 including Color 1 and Color 2
Phong_Pear_Original_L12_C12_M1 = phong(Pear_Original_NVs, [light_dir_1,␣
 ↪light_dir_2], [color_1, color_2], material_1, view, Pear_Masks)

## Material 2
# Light Source 1 and Color 1
Phong_Pear_Original_L1_C1_M2 = phong(Pear_Original_NVs, light_dir_1, color_1,␣
 ↪material_2, view, Pear_Masks)
# Light Source 2 and Color 2
Phong_Pear_Original_L2_C2_M2 = phong(Pear_Original_NVs, light_dir_2, color_2,␣
 ↪material_2, view, Pear_Masks)
# Combine Both Light Source 1 and 2 including Color 1 and Color 2
Phong_Pear_Original_L12_C12_M2 = phong(Pear_Original_NVs, [light_dir_1,␣
 ↪light_dir_2], [color_1, color_2], material_2, view, Pear_Masks)

## Material 3
# Light Source 1 and Color 1
```

```
Phong_Pear_Original_L1_C1_M3 = phong(Pear_Original_NVs, light_dir_1, color_1,␣
 ↪material_3, view, Pear_Masks)
# Light Source 2 and Color 2
Phong_Pear_Original_L2_C2_M3 = phong(Pear_Original_NVs, light_dir_2, color_2,␣
 ↪material_3, view, Pear_Masks)
# Combine Both Light Source 1 and 2 including Color 1 and Color 2
Phong_Pear_Original_L12_C12_M3 = phong(Pear_Original_NVs, [light_dir_1,␣
 ↪light_dir_2], [color_1, color_2], material_3, view, Pear_Masks)

### Diffuse Pear's Normals
## Material 1
# Light Source 1 and Color 1
Phong_Pear_Diffuse_L1_C1_M1 = phong(Pear_Diffuse_NVs, light_dir_1, color_1,␣
 ↪material_1, view, Pear_Masks)
# Light Source 2 and Color 2
Phong_Pear_Diffuse_L2_C2_M1 = phong(Pear_Diffuse_NVs, light_dir_2, color_2,␣
 ↪material_1, view, Pear_Masks)
# Combine Both Light Source 1 and 2 including Color 1 and Color 2
Phong_Pear_Diffuse_L12_C12_M1 = phong(Pear_Diffuse_NVs, [light_dir_1,␣
 ↪light_dir_2], [color_1, color_2], material_1, view, Pear_Masks)

## Material 2
# Light Source 1 and Color 1
Phong_Pear_Diffuse_L1_C1_M2 = phong(Pear_Diffuse_NVs, light_dir_1, color_1,␣
 ↪material_2, view, Pear_Masks)
# Light Source 2 and Color 2
Phong_Pear_Diffuse_L2_C2_M2 = phong(Pear_Diffuse_NVs, light_dir_2, color_2,␣
 ↪material_2, view, Pear_Masks)
# Combine Both Light Source 1 and 2 including Color 1 and Color 2
Phong_Pear_Diffuse_L12_C12_M2 = phong(Pear_Diffuse_NVs, [light_dir_1,␣
 ↪light_dir_2], [color_1, color_2], material_2, view, Pear_Masks)

## Material 3
# Light Source 1 and Color 1
Phong_Pear_Diffuse_L1_C1_M3 = phong(Pear_Diffuse_NVs, light_dir_1, color_1,␣
 ↪material_3, view, Pear_Masks)
# Light Source 2 and Color 2
Phong_Pear_Diffuse_L2_C2_M3 = phong(Pear_Diffuse_NVs, light_dir_2, color_2,␣
 ↪material_3, view, Pear_Masks)
# Combine Both Light Source 1 and 2 including Color 1 and Color 2
Phong_Pear_Diffuse_L12_C12_M3 = phong(Pear_Diffuse_NVs, [light_dir_1,␣
 ↪light_dir_2], [color_1, color_2], material_3, view, Pear_Masks)



#Display rendered results for pear for all materials and light sources and␣
 ↪combination of light sources
```

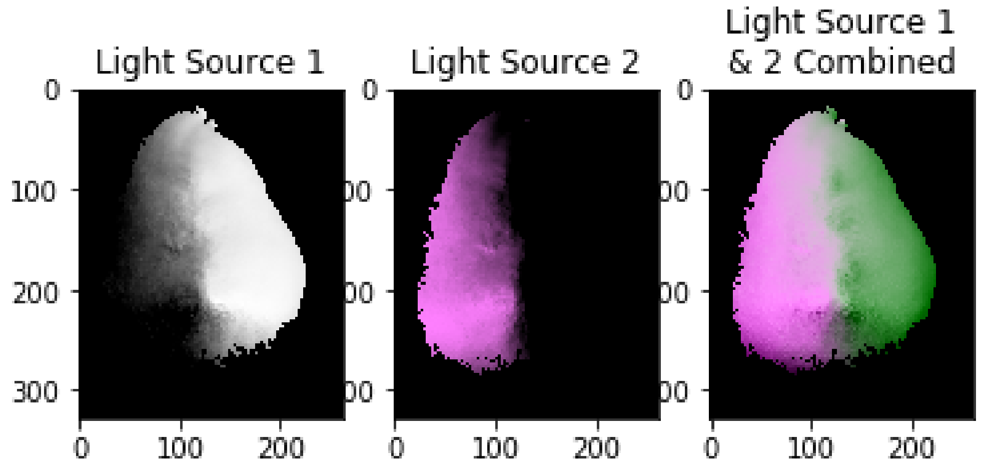
```
### Original Pear's Normals
## Material 1
plot_Phong(Phong_Pear_Original_L1_C1_M1, Phong_Pear_Original_L2_C2_M1,␣
 ↪Phong_Pear_Original_L12_C12_M1, "Pear", 1, "Original")
## Material 2
plot_Phong(Phong_Pear_Original_L1_C1_M2, Phong_Pear_Original_L2_C2_M2,␣
 ↪Phong_Pear_Original_L12_C12_M2, "Pear", 2, "Original")
## Material 3
plot_Phong(Phong_Pear_Original_L1_C1_M3, Phong_Pear_Original_L2_C2_M3,␣
 ↪Phong_Pear_Original_L12_C12_M3, "Pear", 3, "Original")

### Diffuse Sphere's Normals
## Material 1
plot_Phong(Phong_Pear_Diffuse_L1_C1_M1, Phong_Pear_Diffuse_L2_C2_M1,␣
 ↪Phong_Pear_Diffuse_L12_C12_M1, "Pear", 1, "Diffuse")
## Material 2
plot_Phong(Phong_Pear_Diffuse_L1_C1_M2, Phong_Pear_Diffuse_L2_C2_M2,␣
 ↪Phong_Pear_Diffuse_L12_C12_M2, "Pear", 2, "Diffuse")
## Material 3
plot_Phong(Phong_Pear_Diffuse_L1_C1_M3, Phong_Pear_Diffuse_L2_C2_M3,␣
 ↪Phong_Pear_Diffuse_L12_C12_M3, "Pear", 3, "Diffuse")
```

```
Pear Phong Model Plots:
Original
Material 1, [   = 0.1,    = 0.5,    = 5]
Light Source 1, [-0.5, 0.5, 0.5], Color 1, [1, 1, 1]
Light Source 2, [1, 0, 0], Color 2, [1, 0.5, 1]
```

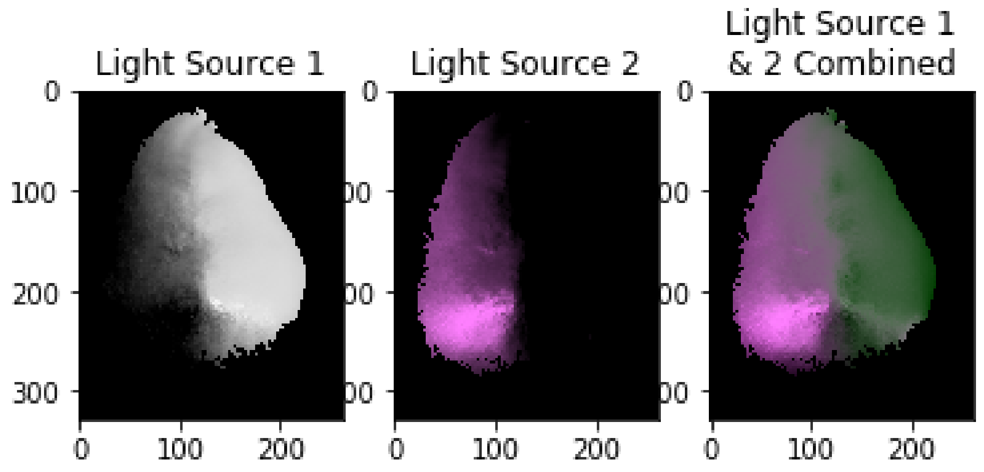


```
Pear Phong Model Plots:
Original
Material 2, [   = 0.5,    = 0.1,    = 5]
```

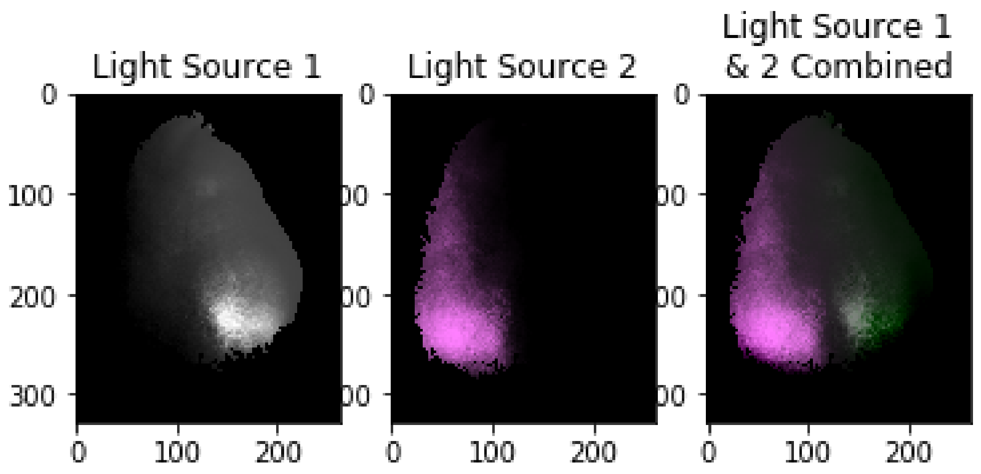
```
Light Source 1, [-0.5, 0.5, 0.5], Color 1, [1, 1, 1]
Light Source 2, [1, 0, 0], Color 2, [1, 0.5, 1]
```



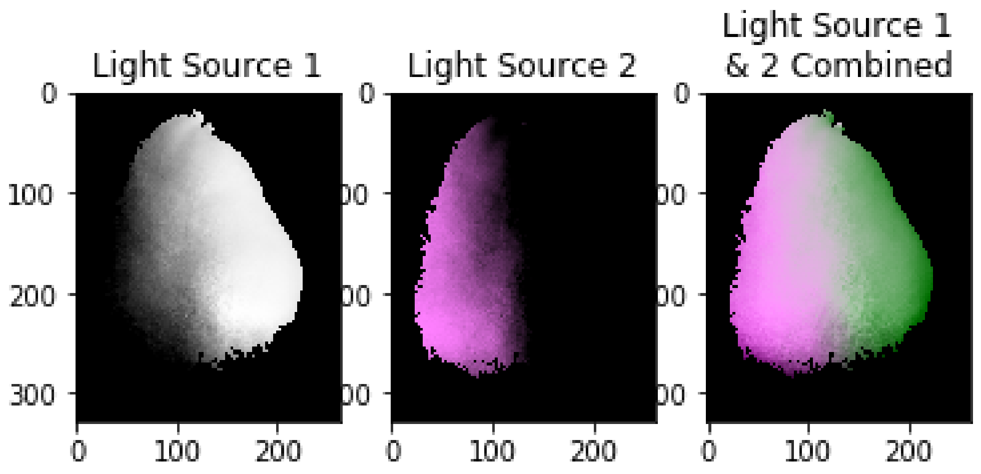
```
Pear Phong Model Plots:
Original
Material 3, [  = 0.5,   = 0.5,   = 10]
Light Source 1, [-0.5, 0.5, 0.5], Color 1, [1, 1, 1]
Light Source 2, [1, 0, 0], Color 2, [1, 0.5, 1]
```



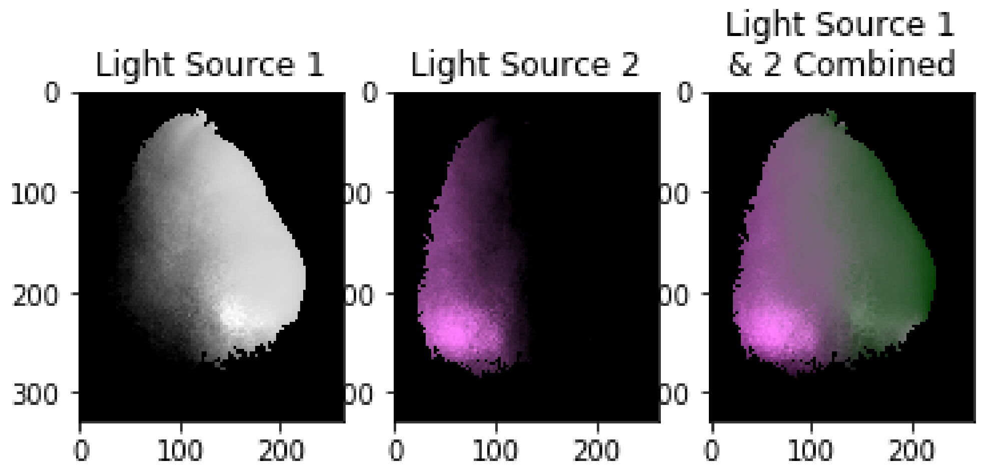
```
Pear Phong Model Plots:
Diffuse
Material 1, [  = 0.1,   = 0.5,   = 5]
Light Source 1, [-0.5, 0.5, 0.5], Color 1, [1, 1, 1]
Light Source 2, [1, 0, 0], Color 2, [1, 0.5, 1]
```



```
Pear Phong Model Plots:
Diffuse
Material 2, [  = 0.5,   = 0.1,   = 5]
Light Source 1, [-0.5, 0.5, 0.5], Color 1, [1, 1, 1]
Light Source 2, [1, 0, 0], Color 2, [1, 0.5, 1]
```



```
Pear Phong Model Plots:
Diffuse
Material 3, [  = 0.5,   = 0.5,   = 10]
Light Source 1, [-0.5, 0.5, 0.5], Color 1, [1, 1, 1]
Light Source 2, [1, 0, 0], Color 2, [1, 0.5, 1]
```



1.5 Problem 4: Photometric Stereo, Specularity Removal (20 pts)

The goal of this problem is to implement a couple of different algorithms that reconstruct a surface using the concept of Lambertian photometric stereo. Additionally, you will implement the specular removal technique of Mallick et al., which enables photometric stereo to be performed on certain non-Lambertian materials.

You can assume a Lambertian reflectance function once specularities are removed. However, note that the albedo is unknown and non-constant in the images you will use.

As input, your program should take in multiple images along with the light source direction for each image. Each image is associated with only a single light, and hence a single direction.

1.5.1 Data

You will use synthetic images and specular sphere images as data. These images are stored in `.pickle` files which have been graciously provided by Satya Mallick. Each `.pickle` file contains

- `im1`, `im2`, `im3`, `im4`, ... images.
- `l1`, `l2`, `l3`, `l4`, ... light source directions.

1.5.2 Part 1: Lambertian Photometric Stereo [8 pts]

Implement the photometric stereo technique described in the lecture. Your program should have two parts:

1. Read in the images and corresponding light source directions, and estimate the surface normals and albedo map.
2. Reconstruct the depth map from the surface normals. You should first try the naive scanline-based "shape by integration" method described in lecture. (You are required to implement this.) For comparison, you should also integrate using the Horn technique which is already implemented for you in the `horn_integrate` function. Note that for good results you will often want to run the `horn_integrate` function with 10000-100000 iterations, which will take

a while. For your final submission, we will require that you run Horn integration for 1000 (one thousand) iterations or more in each case. But for debugging, it is suggested that you keep the number of iterations low.

You will find all the data for this part in `synthetic_data.pickle`. Try using only `im1`, `im2` and `im4` first. Display your outputs as mentioned below.

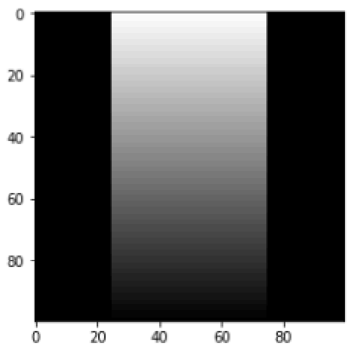
Then use all four images (most accurate).

For **each** of the **two above cases** you must output:

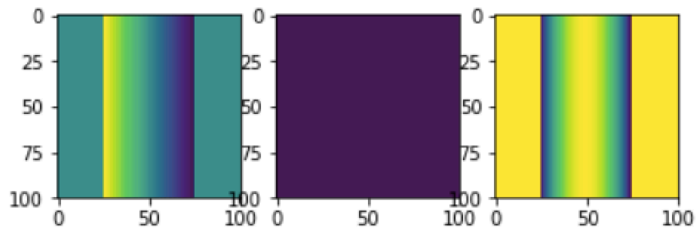
1. The estimated albedo map.
2. The estimated surface normals by showing both
    1. Needle map, and
    2. Three images showing each of the surface normal components.
3. A wireframe of the depth map given by the scanline method.
4. A wireframe of the depth map given by Horn integration.

In total, we expect 2 * 7 = 14 images for this part.

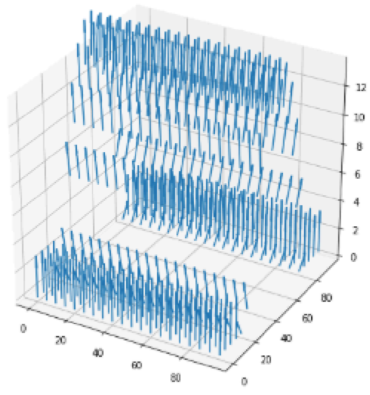
An example of outputs is shown in the figure below. (The example outputs only include one depth map, although we expect two – see above.)



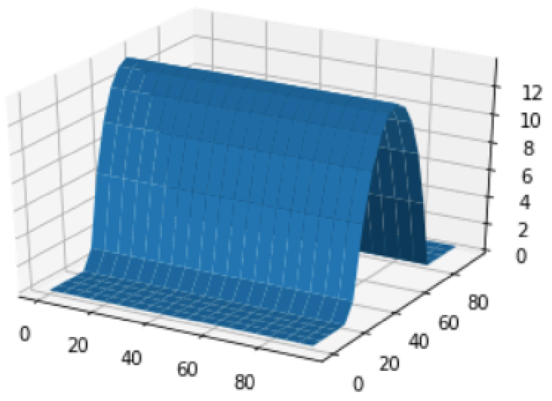
Albedo map



Normals as three separate channels



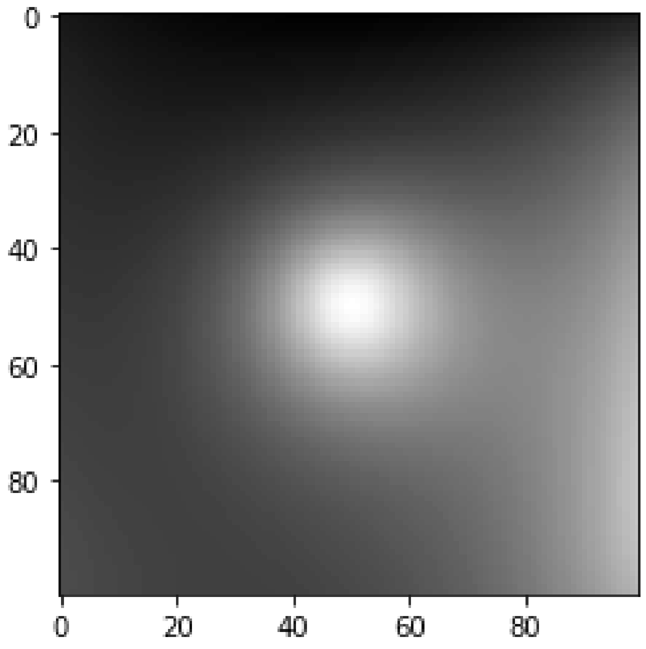
Needle map



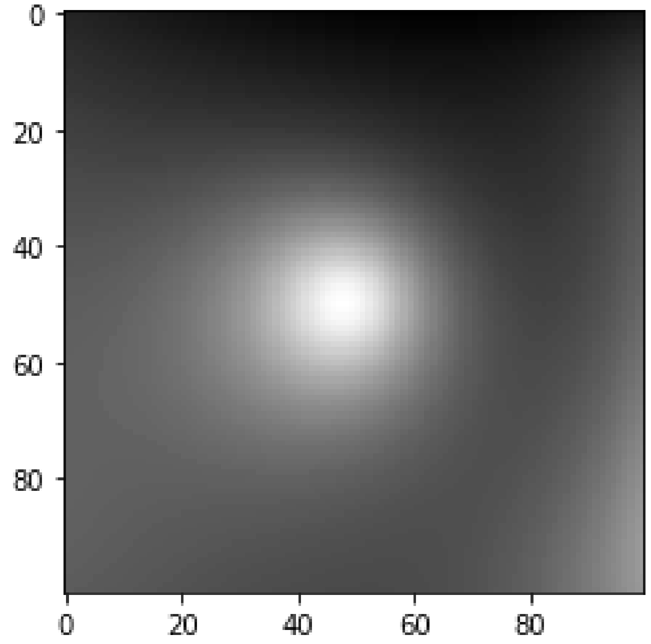
Wireframe of depth map

```
[134]: # Setup
import pickle
import numpy as np
from time import time
from skimage import io
%matplotlib inline
import matplotlib.pyplot as plt

### Example: how to read and access data from a .pickle file
pickle_in = open("synthetic_data.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")

# data is a dict which stores each element as a key-value pair.
print("Keys: ", list(data.keys()))

# To access the value of an entity, refer to it by its key.
for i in range(1, 5):
    print("\nImage %d:" % i)
    plt.imshow(data["im%d" % i], cmap="gray")
    plt.show()
    print("Light source direction: " + str(data["l%d" % i]))
```

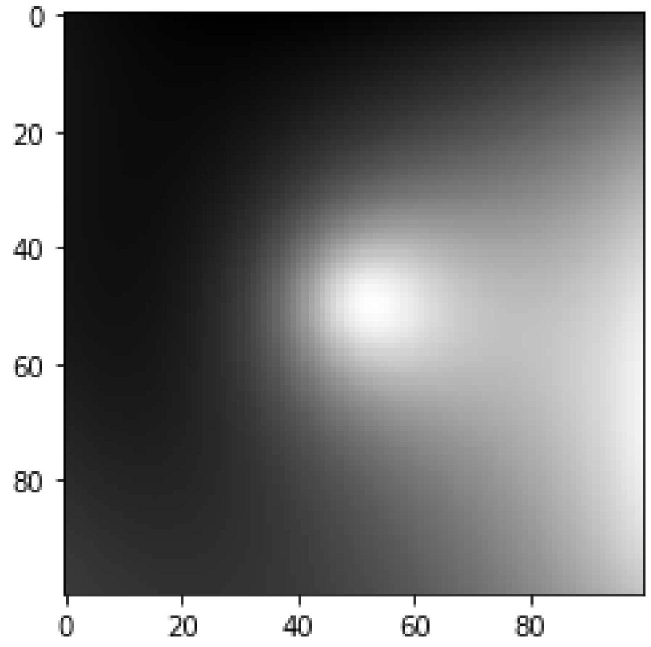
```
Keys:  ['__version__', 'l4', '__header__', 'im1', 'im3', 'im2', 'l2', 'im4',
'l1', '__globals__', 'l3']

Image 1:
```



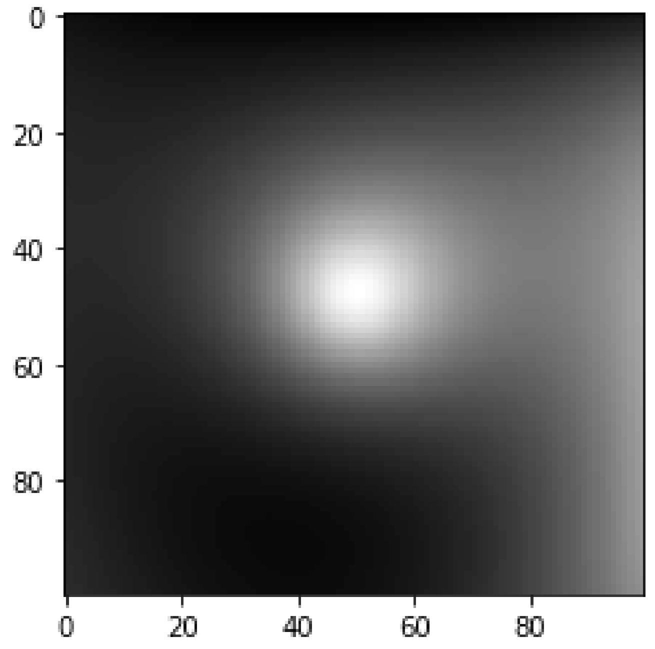
```
Light source direction: [[0 0 1]]

Image 2:
```



```
Light source direction: [[0.2 0.  1. ]]

Image 3:
```



```
Light source direction: [[-0.2  0.   1. ]]

Image 4:
```



```
Light source direction: [[0.  0.2 1. ]]
```

Based on the above images, can you interpret the orientation of the coordinate frame? If we label the axes in order as x, y, z, then the x-axis points left, the y-axis points up, and the z-axis points out of the screen in our direction. (That means this is a left-handed coordinate system. How will this affect the scanline integration algorithm? Hint: if you integrate rightward along the x-axis and downward along the y-axis, you will be doing in opposite directions to the axes, and the partial derivatives you compute may need to be modified.)

Note: as clarification, no direct response is needed for this cell.

[135]:
```python
import numpy as np
from scipy.signal import convolve

def horn_integrate(gx, gy, mask, niter):
    """
    horn_integrate recovers the function g from its partial
    derivatives gx and gy.
    mask is a binary image which tells which pixels are
    involved in integration.
    niter is the number of iterations.
    typically 100,000 or 200,000,
    although the trend can be seen even after 1000 iterations.
    """
    g = np.ones(np.shape(gx))

    gx = np.multiply(gx, mask)
    gy = np.multiply(gy, mask)

    A = np.array([[0,1,0],[0,0,0],[0,0,0]]) #y-1
    B = np.array([[0,0,0],[1,0,0],[0,0,0]]) #x-1
    C = np.array([[0,0,0],[0,0,1],[0,0,0]]) #x+1
    D = np.array([[0,0,0],[0,0,0],[0,1,0]]) #y+1

    d_mask = A + B + C + D

    den = np.multiply(convolve(mask,d_mask,mode="same"),mask)
    den[den == 0] = 1
    rden = 1.0 / den
    mask2 = np.multiply(rden, mask)

    m_a = convolve(mask, A, mode="same")
    m_b = convolve(mask, B, mode="same")
    m_c = convolve(mask, C, mode="same")
    m_d = convolve(mask, D, mode="same")

    term_right = np.multiply(m_c, gx) + np.multiply(m_d, gy)
    t_a = -1.0 * convolve(gx, B, mode="same")
    t_b = -1.0 * convolve(gy, A, mode="same")
    term_right = term_right + t_a + t_b
    term_right = np.multiply(mask2, term_right)

    for k in range(niter):
        g = np.multiply(mask2, convolve(g, d_mask, mode="same")) + term_right

    return g
```

[195]:
```
def photometric_stereo(images, lights, mask, horn_niter):

    """mask is an optional parameter which you are encouraged to use.
    It can be used e.g. to ignore the background when integrating the normals.
    It should be created by converting the images to grayscale, averaging them,
    normalizing to [0, 1] and thresholding (only using locations for which the
    pixel value is above some threshold).

    The choice of threshold is something you can experiment with,
    but in practice something like 0.05 or 0.1 tends to work well.
    You do not need to use the mask for 1a (it shouldn't matter),
    but you SHOULD use it to filter out the background for the specular data␣
↪(1c).
    """

    """ ==========
    YOUR CODE HERE
    ========== """

    # note:
    # images : (n_ims, h, w)
    # lights : (n_ims, 3)
    # mask   : (h, w)

    # Lambertian Photometric Stereo
    # b_T = [e_1, e_2, e_3][s_1, s_2, s_3]^-1
    # e_1, e_2, e_3 are pixel values/intensity (n images x 1)
    # s_1, s_2, s_3 are light sources (n images x 3)
    # b_T (1 x 3)
    # norm = b / |b|
    # albedo, a = |b|
    #######################################
    # b = (S^T * S)^-1 * S^T * e
    # where,
    # S - Light Sources Matrix (3 x n)
    # e - Pixel Values / Intensities (n x 1)

    # Initialize Albedo (H x W)
    albedo = np.ones(images[0].shape)

    # Initialize Normals (H x W x 3)
    normals = np.dstack((np.zeros(images[0].shape),
                         np.zeros(images[0].shape),
                         np.ones(images[0].shape)))

    # Initialize albedo-scaled surface normal in matrix, B
    B = np.ones((images.shape[1], images.shape[2], 3))

    # Determine S^+ = (S^T * S)^-1 * S^T (3 x n images)
    S_plus = np.dot(np.linalg.inv(np.dot(np.transpose(lights),lights)),np.
↪transpose(lights))

    # Calculate the albedo-scaled surface normal, b (3 x 1)
    for height in range(images.shape[1]):
        for width in range(images.shape[2]):
            # Initialize e
            e = np.zeros((images.shape[0],1))

            # Get the Pixel Values, e (n x 1)
            for image_num in range(images.shape[0]):

                # e in (n,) matrix
                e[image_num] = images[image_num, height, width]

            # Reshape e to n x 1
            e = e.reshape(e.shape[0], 1)

            # Determine the albedo-scaled surface normal, b (3 x 1)
            b = np.dot(S_plus,e)

            # Store the b in B matrix
            B[height, width, :] = b.T

    # Calculate the normal for each pixel, n = b / |b|, and albedo, a = |b|
    for height in range(images.shape[1]):
        for width in range(images.shape[2]):

            # Extract each albedo element from the albedo vector
            b1 = B[height, width, 0]
            b2 = B[height, width, 1]
            b3 = B[height, width, 2]

            # Calculate the albedo for a pixel
            b_mag = np.sqrt(np.power(b1,2) + np.power(b2,2) + np.power(b3,2))

            # Store the albedo for a pixel in the albedo array
            albedo[height, width] = b_mag

            # If the mask value for a particular pixel is more than 0,
            if mask[height, width] > 0:

                # Calculate the normal for each pixel
                normals[height, width, :] = B[height, width, :] / b_mag


    # Initialize Height matrix (H x W)
    H = np.ones(images[0].shape)

    # Initialize cumulative h (scalar)
    cum_h = 0

    # Determine the heights of the left most column.
    # |
    # |
    # v
    for height in range(images.shape[1]):

        # If mask is more than 0 for the pixel,
        if mask[height, 0] > 0:

            # Extract each element from the normal vector
            normal_x = normals[height, 0, 0]
            normal_y = normals[height, 0, 1]
            normal_z = normals[height, 0, 2]

            # Calculate q = df/dy
            q = normal_y/normal_z

            # Cumulative h
            cum_h = cum_h + q

            # Assign it to sum of previous H and q
            H[height, 0] = cum_h

    # Determine the rest of rows of the heights
    # ------>
    # ------>
    # ------>
    for height in range(images.shape[1]):

        # Initialize define the starting height as the first element of the␣
↪left most column
        cum_height_rest = H[height, 0]

        for width in range(1, images.shape[2]): # Start from the second column

            # If mask is more than 0 for the pixel,
            if mask[height, width] > 0:

                # Extract each element from the normal vector
                normal_x = normals[height, width, 0]
                normal_y = normals[height, width, 1]
                normal_z = normals[height, width, 2]

                # Calculate p = df/dx
                p = normal_x/normal_z

                # Cumulative cum_height_rest
                cum_height_rest = cum_height_rest + p
                # Assign it to sum of previous H and p
                H[height, width] = cum_height_rest


    ## Horn's Method
    # Integral_image (z_x - p)^2 + (z_y - q)^2 dxdy
    # p = n_x/n_y
    # q = n_y/n_x
    # Initialize H_horn (H x W)
    H_horn = np.ones(images[0].shape)
    # Initialize (H x W) gx and gy (H x W)
    g_x = np.zeros(images[0].shape)
    g_y = np.zeros(images[0].shape)

    # Calculate p and q for each pixel.
    for height in range(images.shape[1]):
        for width in range(1,images.shape[2]):

            # If mask is more than 0 for the pixel,
            if mask[height, width] > 0:

                # Extract each element from the normal vector
                normal_x = normals[height, width, 0]
                normal_y = normals[height, width, 1]
                normal_z = normals[height, width, 2]

                # Calculate p
                g_x[height, width] = normal_x / normal_z

                # Calculate q
                g_y[height, width] = normal_y / normal_z

    # Call horn_integrate function
    H_horn = horn_integrate(g_x, g_y, mask, horn_niter)

    return albedo, normals, H, H_horn
```

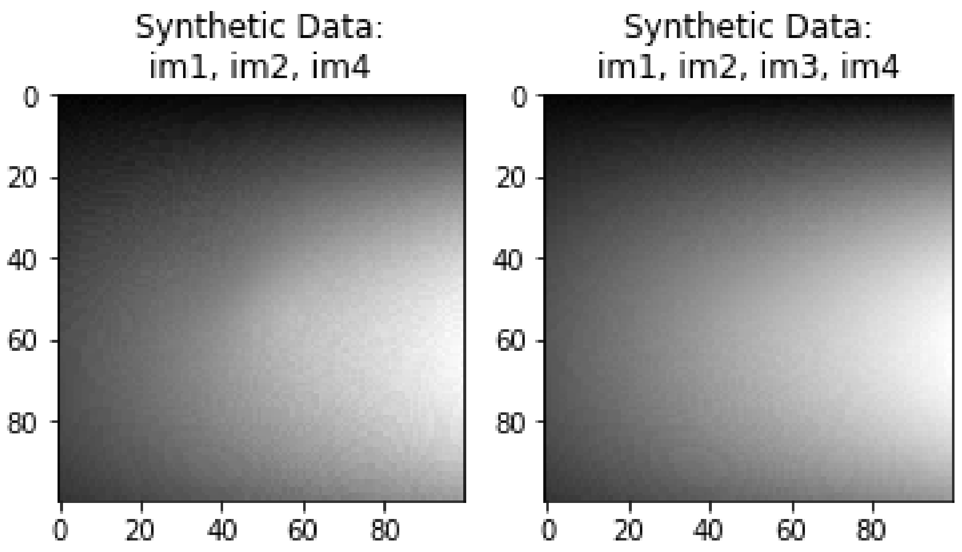
[199]:
```
from mpl_toolkits.mplot3d import Axes3D

pickle_in = open("synthetic_data.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")

lights = np.vstack((data["l1"], data["l2"], data["l4"]))
lights_2 = np.vstack((data["l1"], data["l2"], data["l3"], data["l4"]))

images = []
```

```
images.append(data["im1"])
images.append(data["im2"])
images.append(data["im4"])
images = np.array(images)

images_2 = []
images_2.append(data["im1"])
images_2.append(data["im2"])
images_2.append(data["im3"])
images_2.append(data["im4"])
images_2 = np.array(images_2)

mask = np.ones(data["im1"].shape)

albedo, normals, depth, horn = photometric_stereo(images, lights, mask,␣
 ↪horn_niter = 1000)

albedo_2, normals_2, depth_2, horn_2 = photometric_stereo(images_2, lights_2,␣
 ↪mask, horn_niter = 1000)

# --------------------------------------------------------------------------
# The following code is just a working example so you don't get stuck with any
# of the graphs required. You may want to write your own code to align the
# results in a better layout. You are also free to change the function
# however you wish; just make sure you get all of the required outputs.
# --------------------------------------------------------------------------

def visualize(albedo, albedo_2, normals, normals_2, depth, depth_2, horn,␣
 ↪horn_2):
    # Stride in the plot, you may want to adjust it to different images
    #stride = 15
    stride = 6

    # showing albedo map
    print("Albedo Map")
    fig = plt.figure()
    ax1 = fig.add_subplot(121)
    albedo_max = albedo.max()
    albedo = albedo / albedo_max
    plt.title("Synthetic Data:\nim1, im2, im4")
    ax1.imshow(albedo, cmap="gray")
    ax2 = fig.add_subplot(122)
    albedo_max_2 = albedo_2.max()
    albedo_2 = albedo_2 / albedo_max_2
    plt.title("Synthetic Data:\nim1, im2, im3, im4")
    ax2.imshow(albedo_2, cmap="gray")
    plt.show()
```

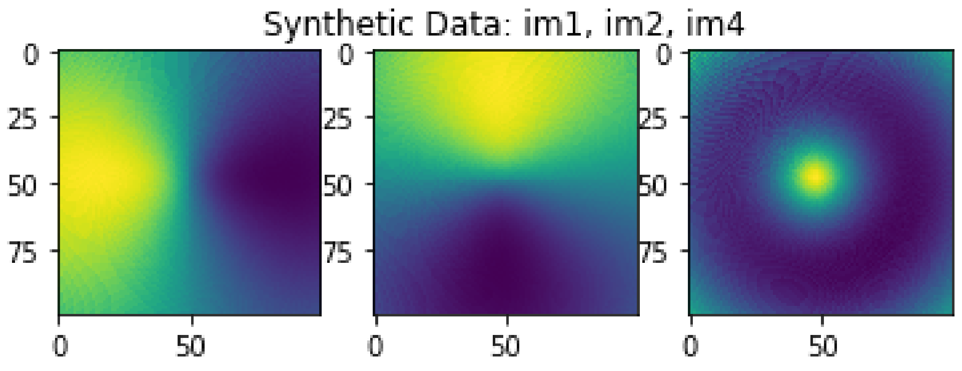
```
# showing normals as three separate channels
print("Normals as Three Separate Channels")
figure = plt.figure()
ax1 = figure.add_subplot(131)
ax1.imshow(normals[..., 0])
ax2 = figure.add_subplot(132)
plt.title("Synthetic Data: im1, im2, im4")
ax2.imshow(normals[..., 1])
ax3 = figure.add_subplot(133)
ax3.imshow(normals[..., 2])
plt.show()

fig = plt.figure()
ax1 = fig.add_subplot(131)
ax1.imshow(normals_2[..., 0])
ax2 = fig.add_subplot(132)
plt.title("Synthetic Data: im1, im2, im3, im4")
ax2.imshow(normals_2[..., 1])
ax3 = fig.add_subplot(133)
ax3.imshow(normals_2[..., 2])
plt.show()

# showing normals as quiver
print("Normals as Quiver")
print("Synthetic Data: im1, im2, im4")
X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals)[0], 6),
                      np.arange(0,np.shape(normals)[1], 6),
                      np.arange(1))
X = X[..., 0]
Y = Y[..., 0]
Z = depth[::stride,::stride].T
NX = normals[..., 0][::stride,::-stride].T
NY = normals[..., 1][::-stride,::stride].T
NZ = normals[..., 2][::stride,::stride].T
fig = plt.figure(figsize=(5, 5))
ax = fig.gca(projection='3d')
plt.quiver(X,Y,Z,NX,NY,NZ, length=10)
plt.show()

print("Normals as Quiver")
print("Synthetic Data: im1, im2, im3, im4")
X_2, Y_2, _ = np.meshgrid(np.arange(0,np.shape(normals_2)[0], 6),
                      np.arange(0,np.shape(normals_2)[1], 6),
                      np.arange(1))
X_2 = X_2[..., 0]
Y_2 = Y_2[..., 0]
```

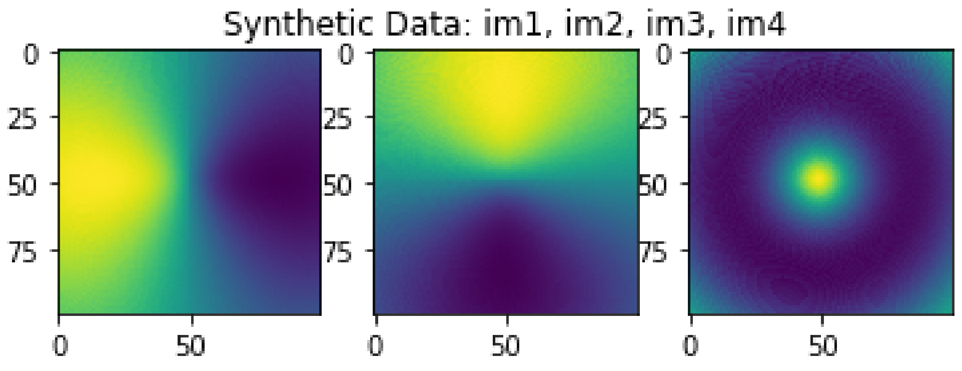
```
    Z_2 = depth_2[::stride,::stride].T
    NX_2 = normals_2[..., 0][::stride,::-stride].T
    NY_2 = normals_2[..., 1][::-stride,::stride].T
    NZ_2 = normals_2[..., 2][::stride,::stride].T
    fig = plt.figure(figsize=(5, 5))
    ax = fig.gca(projection='3d')
    plt.quiver(X_2,Y_2,Z_2,NX_2,NY_2,NZ_2, length=10)
    plt.show()


    # plotting wireframe depth map
    print("Wireframe Depth Map")
    print("Synthetic Data: im1, im2, im4")
    H = depth[::stride,::stride]
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_surface(X,Y, H.T)
    plt.show()

    print("Wireframe Depth Map")
    print("Synthetic Data: im1, im2, im3, im4")
    H_2 = depth_2[::stride,::stride]
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_surface(X_2,Y_2, H_2.T)
    plt.show()

    # plotting Horn's Integration
    print("Horn's Integration")
    print("Synthetic Data: im1, im2, im4")
    H = horn[::stride,::stride]
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_surface(X,Y, H.T)
    plt.show()

    print("Horn's Integration")
    print("Synthetic Data: im1, im2, im3, im4")
    H_2 = horn_2[::stride,::stride]
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_surface(X_2,Y_2, H_2.T)
    plt.show()

visualize(albedo, albedo_2, normals, normals_2, depth, depth_2, horn, horn_2)
```

```
Albedo Map
```

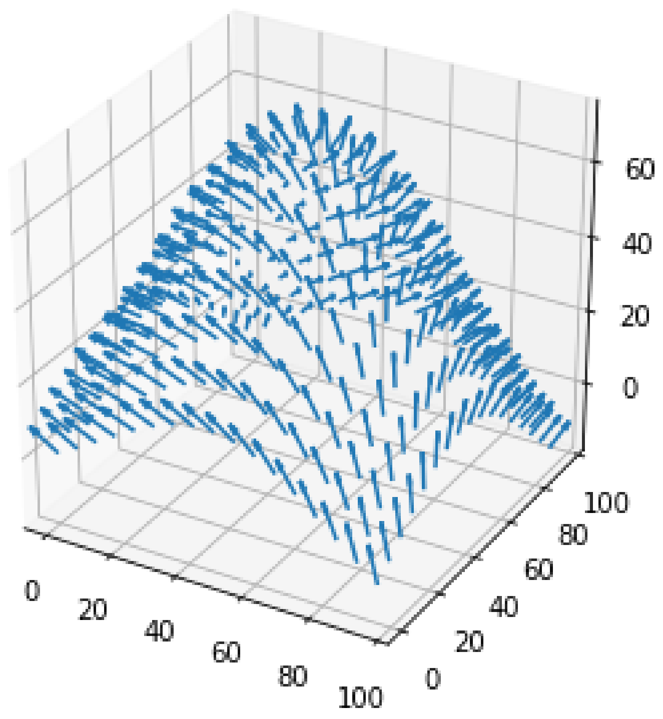


```
Normals as Three Separate Channels
```

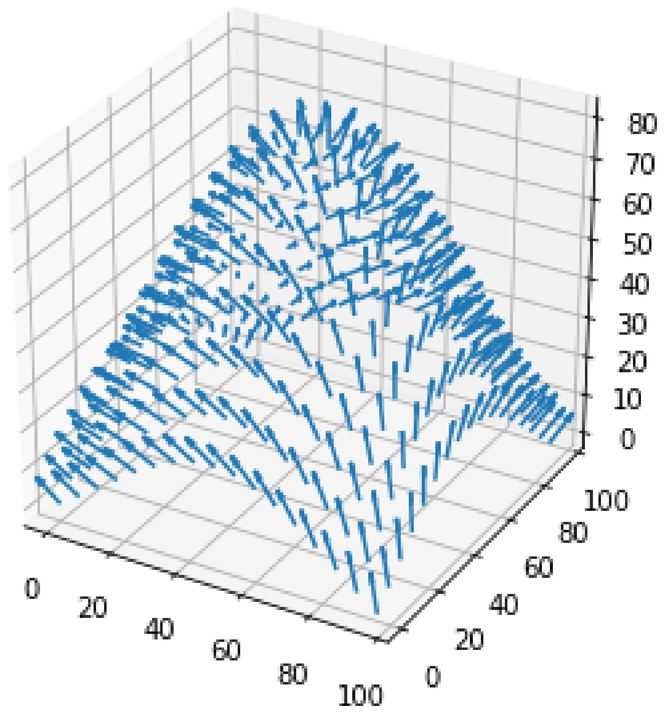




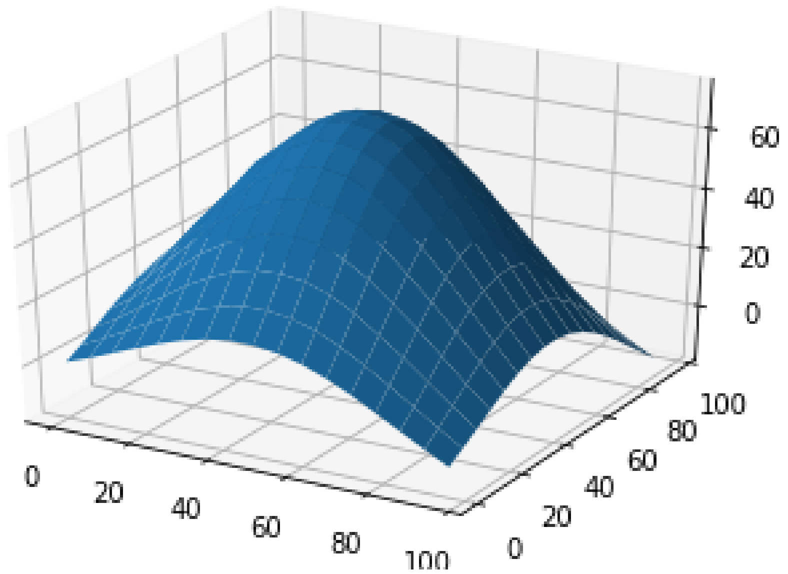
```
Normals as Quiver
Synthetic Data: im1, im2, im4
```



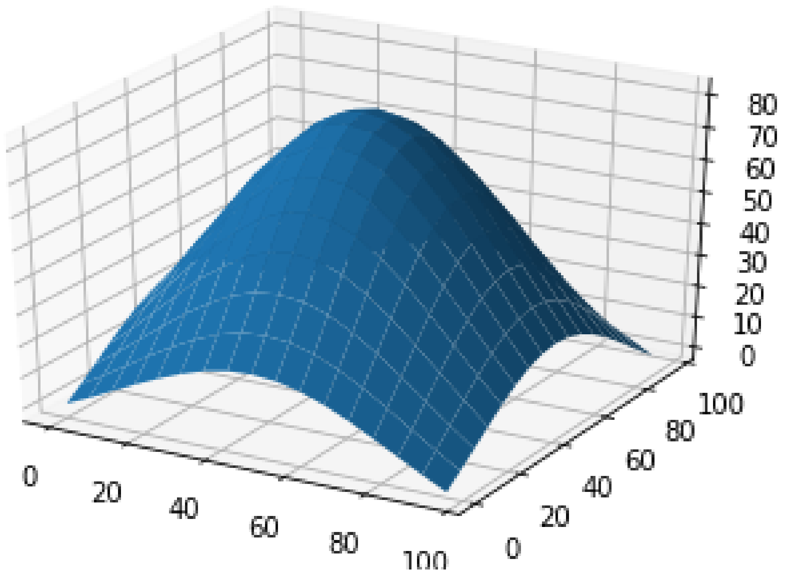
```
Normals as Quiver
Synthetic Data: im1, im2, im3, im4
```



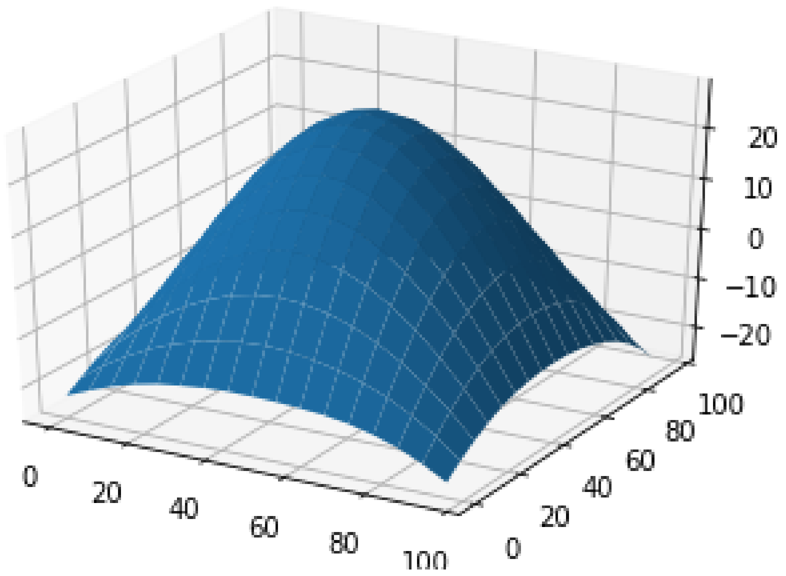
```
Wireframe Depth Map
Synthetic Data: im1, im2, im4
```



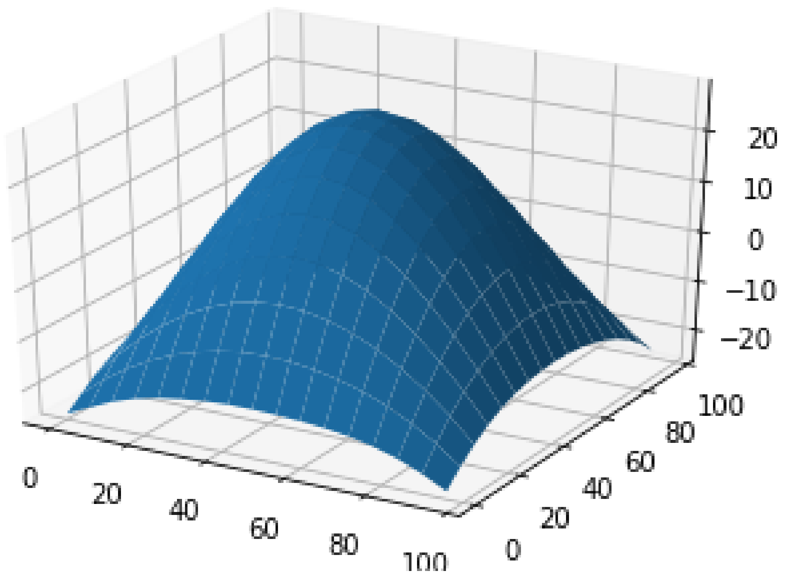
```
Wireframe Depth Map
Synthetic Data: im1, im2, im3, im4
```



```
Horn's Integration
Synthetic Data: im1, im2, im4
```



```
Horn's Integration
Synthetic Data: im1, im2, im3, im4
```



[52]:
```
# Don't forget to run your photometric stereo code on TWO sets of images!
# (One being {im1, im2, im4}, and the other being {im1, im2, im3, im4}.)
```

1.5.3 Part 2: Specularity Removal [6 pts]

Implement the specularity removal technique described in *Beyond Lambert: Reconstructing Specular Surfaces Using Color* (by Mallick, Zickler, Kriegman, and Belhumeur; CVPR 2005).

Your program should input an RGB image and light source color and output the corresponding SUV image.

Try this out first with the specular sphere images and then with the pear images.

For each of the specular sphere and pear images, include

1. The original image (in RGB colorspace).
2. The recovered S channel of the image.
3. The recovered diffuse part of the image. Use $D = \sqrt{U^2 + V^2}$ to represent the diffuse part.

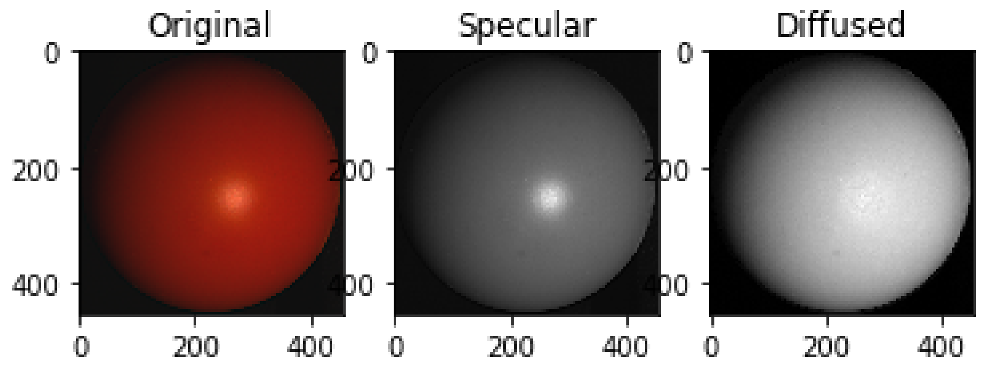
In total, we expect $2 * 3 = 6$ images as outputs for this problem.

Note: You will find all the data for this part in `specular_sphere.pickle` and `specular_pear.pickle`.

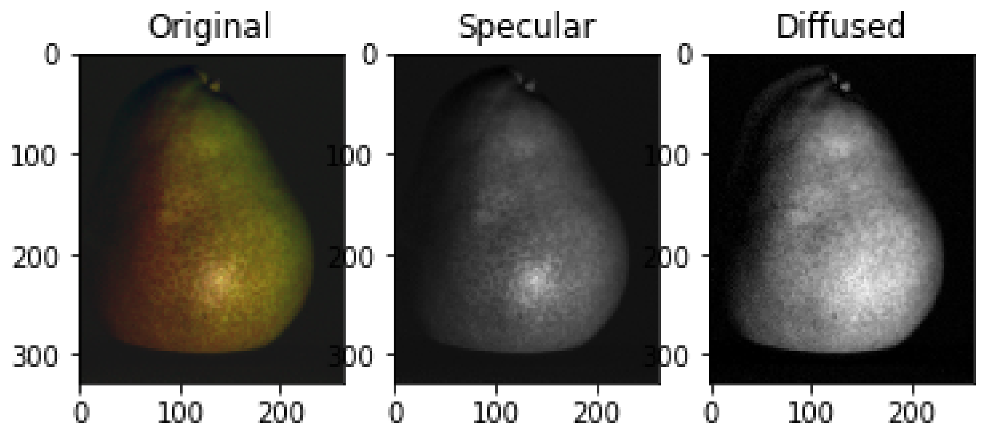
[146]:
```
def get_rot_mat(rot_v, unit=None):
    '''
    Takes a vector and returns the rotation matrix required to align the
    unit vector(2nd arg) to it.
    '''
    if unit is None:
        unit = [1.0, 0.0, 0.0]

    rot_v = rot_v/np.linalg.norm(rot_v)
    uvw = np.cross(rot_v, unit) # axis of rotation

    rcos = np.dot(rot_v, unit) # cos by dot product
    rsin = np.linalg.norm(uvw) # sin by magnitude of cross product

    # normalize and unpack axis
    if not np.isclose(rsin, 0):
        uvw = uvw/rsin
    u, v, w = uvw

    # compute rotation matrix
    R = (
        rcos * np.eye(3) +
        rsin * np.array([
            [ 0, -w,  v],
            [ w,  0, -u],
            [-v,  u,  0]
        ]) +
        (1.0 - rcos) * uvw[:,None] * uvw[None,:]
    )
    return R

def RGBToSUV(I_rgb, rot_vec):
    '''
    Your implementation which takes an RGB image and a vector encoding
    the orientation of the S channel w.r.t. to RGB.
    '''

    """ ==========
    YOUR CODE HERE
    ========== """
    # Data-dependet SUV Color Space
    # I_SUV = [R] * I_RGB
    # [R] = [S|U|V]^t
    # First Row of R is Specular Color
    # Other Rows are orthogonal to S

    # Initialize Specular Matrix (H x W)
    S = np.ones(I_rgb.shape[:2])
    # Initialize Diffuse Matrix (H x W)
    G = np.ones(I_rgb.shape[:2])
    # Obtain Rotation Matrix (3 x 3)
    R = get_rot_mat(rot_vec)
    for height in range(I_rgb.shape[0]):
        for width in range(I_rgb.shape[1]):

            # Extract the RGB from the I_rgb
            rgb_matrix = I_rgb[height, width, :] # (3,)
            rgb_matrix = rgb_matrix.reshape(rgb_matrix.shape[0],1) # (3,1)

            # Calculate the I_SUV
            I_SUV = np.dot(R, rgb_matrix) # (3 x 1)

            # Reshape I_SUV to (3,)
            re_I_SUV = I_SUV.reshape(3) # (3,)

            # Specular Matrix, S
            S[height, width] = re_I_SUV[0] # S Element

            # Diffuse Matrix, G or D = sqrt(U^2 +V^2)
            G[height, width] = np.sqrt(np.power(re_I_SUV[1], 2) + np.
 ↪power(re_I_SUV[2], 2))

    return S, G

# Original Specular Sphere
pickle_in_sphere = open("specular_sphere.pickle", "rb")
data_sphere = pickle.load(pickle_in_sphere, encoding="latin1")
original_sphere = data_sphere["im1"]

# Specular and Diffuse Spheres
S_sphere, G_sphere = RGBToSUV(data_sphere["im1"], np.
 ↪hstack((data_sphere["c"][0][0],
                                                                       ␣
 ↪data_sphere["c"][1][0],
                                                                       ␣
 ↪data_sphere["c"][2][0])))

# Original Specular Pear
pickle_in_pear = open("specular_pear.pickle", "rb")
data_pear = pickle.load(pickle_in_pear, encoding="latin1")
original_pear = data_pear["im1"]

# Specular and Diffuse Pears
S_pear, G_pear = RGBToSUV(data_pear["im1"], np.hstack((data_pear["c"][0][0],
                                        data_pear["c"][1][0],
                                        data_pear["c"][2][0])))

# Plot the object images
```

```python
def plot_Spec_Removal(original_obj, specular_obj, diffuse_obj, obj_Name):

    print(obj_Name)
    fig = plt.figure()
    ax1 = fig.add_subplot(131)
    plt.title("Original")
    ax1.imshow(normalize(original_obj))
    ax2 = fig.add_subplot(132)
    plt.title("Specular")
    ax2.imshow(specular_obj, cmap = "gray")
    ax3 = fig.add_subplot(133)
    plt.title("Diffused")
    ax3.imshow(diffuse_obj, cmap = "gray")
    plt.show()

# Sphere plots
plot_Spec_Removal(original_sphere, S_sphere, G_sphere, "Sphere")

# Pear plots
plot_Spec_Removal(original_pear, S_pear, G_pear, "Pear")
```

```
Sphere
```



```
Pear
```



1.5.4 Part 3: Robust Photometric Stereo [6 pts]

Now we will perform photometric stereo on our sphere/pear images which include specularities. First, for comparison, run your photometric stereo code from 1a on the original images (converted to grayscale and rescaled/shifted to be in the range [0, 1]). You should notice erroneous "bumps" in the resulting reconstructions, as a result of violating the Lambertian assumption. For this, show the same outputs as in 1a.

Next, combine parts 1 and 2 by removing the specularities (using your code from 1b) and then running photometric stereo on the diffuse components of the specular sphere/pear images. Our goal will be to remove the bumps/sharp parts in the reconstruction.

For the specular sphere image set in `specular_sphere.pickle`, using all of the four images (again, be sure to convert them to grayscale and normalize them so that their values go from 0 to 1), include:

1. The estimated albedo map (original and diffuse).
2. The estimated surface normals (original and diffuse) by showing both
    1. Needle map, and
    2. Three images showing each of the surface normal components.
3. A wireframe of depth map (original and diffuse).
4. A wireframe of the depth map given by Horn integration (original and diffuse).

In total, we expect 2 * 7 = 14 images for the 1a comparison, plus 2 * 7 = 14 images for the outputs after specularity removal has been performed. (Thus 28 output images overall.)

[209]:
```
# ------------------------------------------------------------------------
# You may reuse the code for photometric_stereo here.
# Write your code below to process the data and send it to photometric_stereo
# and display the albedo, normals, and depth maps.
# ------------------------------------------------------------------------
#Libraries
```

```
import pickle
import numpy as np
import copy
%matplotlib inline
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

## Mask Threshold
mask_thres = 0.05

### User-defined Function for Merging Channels
def merge_channels(chnl_0, chnl_1, chnl_2):
    """ Function to merge three single channel images to form a color image.
    This function should return a H*W*3 array which merges all three single␣
 ↪channel images
    (i.e. img0, img1, img2) in the input."""

    re_chnl_0 = chnl_0.reshape(chnl_0.shape[0], chnl_0.shape[1],1) # Reshape␣
 ↪red channel from H x W  to H x W x 1
    re_chnl_1 = chnl_1.reshape(chnl_1.shape[0], chnl_1.shape[1],1) # Reshape␣
 ↪green channel from H x W  to H x W x 1
    re_chnl_2 = chnl_2.reshape(chnl_2.shape[0], chnl_2.shape[1],1) # Reshape␣
 ↪blue channel from H x W  to H x W x 1

    return np.concatenate((re_chnl_0, re_chnl_1, re_chnl_2), axis=2) # merge␣
 ↪them to H x W x 3

### User-defined Function for Visualization
def visualize_ori_diff(albedo, albedo_diff, normals, normals_diff, depth,␣
 ↪depth_diff, horn, horn_diff):
    # Stride in the plot, you may want to adjust it to different images
    stride = 5

    # showing albedo map
    print("Albedo Map")
    fig = plt.figure()
    ax1 = fig.add_subplot(121)
    albedo_max = albedo.max()
    albedo = albedo / albedo_max
    plt.title("Original")
    ax1.imshow(albedo, cmap="gray")
    ax2 = fig.add_subplot(122)
    albedo_max_diff = albedo_diff.max()
    albedo_diff = albedo_diff / albedo_max_diff
    plt.title("Diffused")
    ax2.imshow(albedo_diff, cmap="gray")
```

```
plt.show()

# showing normals as three separate channels
print("Normals as Three Separate Channels")
figure = plt.figure()
ax1 = figure.add_subplot(131)
ax1.imshow(normals[..., 0])
ax2 = figure.add_subplot(132)
plt.title("Original")
ax2.imshow(normals[..., 1])
ax3 = figure.add_subplot(133)
ax3.imshow(normals[..., 2])
plt.show()

fig = plt.figure()
ax1 = fig.add_subplot(131)
ax1.imshow(normals_diff[..., 0])
ax2 = fig.add_subplot(132)
plt.title("Diffused")
ax2.imshow(normals_diff[..., 1])
ax3 = fig.add_subplot(133)
ax3.imshow(normals_diff[..., 2])
plt.show()

# showing normals as quiver
print("Normals as Quiver")
print("Original")
X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals)[0], 5),
                      np.arange(0,np.shape(normals)[1], 5),
                      np.arange(1))
X = X[..., 0]
Y = Y[..., 0]
Z = depth[::stride,::stride].T
NX = normals[..., 0][::stride,::-stride].T
NY = normals[..., 1][::-stride,::stride].T
NZ = normals[..., 2][::stride,::stride].T
fig = plt.figure(figsize=(5, 5))
ax = fig.gca(projection='3d')
plt.quiver(X,Y,Z,NX,NY,NZ, length=10)
plt.show()

print("Normals as Quiver")
print("Diffused")
X_diff, Y_diff, _ = np.meshgrid(np.arange(0,np.shape(normals_diff)[0], 5),
                      np.arange(0,np.shape(normals_diff)[1], 5),
                      np.arange(1))
X_diff = X_diff[..., 0]
```

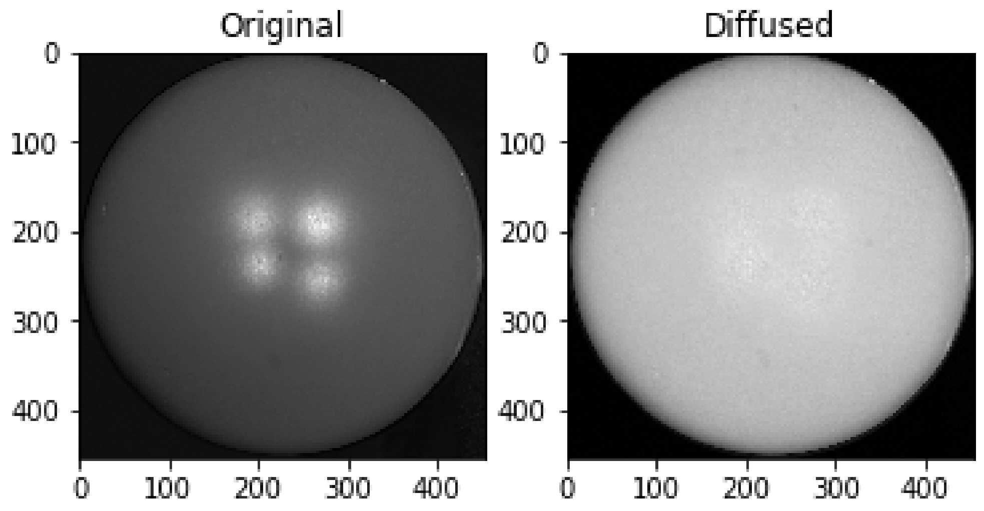
```
    Y_diff = Y_diff[..., 0]
    Z_diff = depth_diff[::stride,::stride].T
    NX_diff = normals_diff[..., 0][::stride,::-stride].T
    NY_diff = normals_diff[..., 1][::-stride,::stride].T
    NZ_diff = normals_diff[..., 2][::stride,::stride].T
    fig = plt.figure(figsize=(5, 5))
    ax = fig.gca(projection='3d')
    plt.quiver(X_diff,Y_diff,Z_diff,NX_diff,NY_diff,NZ_diff, length=10)
    plt.show()


    # plotting wireframe depth map
    print("Wireframe Depth Map")
    print("Original")
    H = depth[::stride,::stride]
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_surface(X,Y, H.T)
    plt.show()

    print("Wireframe Depth Map")
    print("Diffused")
    H_diff = depth_diff[::stride,::stride]
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_surface(X_diff, Y_diff, H_diff.T)
    plt.show()

    # plotting Horn's Integration
    print("Horn's Integration")
    print("Original")
    H = horn[::stride,::stride]
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_surface(X,Y, H.T)
    plt.show()

    print("Horn's Integration")
    print("Diffused")
    H_diff = horn_diff[::stride,::stride]
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_surface(X_diff,Y_diff, H_diff.T)
    plt.show()


### Sphere
```

```
# Open the sphere file
pickle_in_sphere = open("specular_sphere.pickle", "rb")

# Load the sphere file
data_sphere = pickle.load(pickle_in_sphere, encoding="latin1")

# Sphere Light Sources
lights_sphere = np.vstack((data_sphere["l1"], data_sphere["l2"],␣
 ↪data_sphere["l3"], data_sphere["l4"]))

## Convert the colored image to grayscale
# Image 1
im1_sphere_chnl = data_sphere["im1"].dot(np.array([0.299, 0.587, 0.114])) # H x␣
 ↪W
im1_sphere = merge_channels(im1_sphere_chnl, im1_sphere_chnl, im1_sphere_chnl)␣
 ↪# H x W x 3
im1_sphere_normalized = normalize(im1_sphere) # H x W x 3
# Image 2
im2_sphere_chnl = data_sphere["im2"].dot(np.array([0.299, 0.587, 0.114])) # H x␣
 ↪W
im2_sphere = merge_channels(im2_sphere_chnl, im2_sphere_chnl, im2_sphere_chnl)␣
 ↪# H x W x 3
im2_sphere_normalized = normalize(im2_sphere) # H x W x 3
# Image 3
im3_sphere_chnl = data_sphere["im3"].dot(np.array([0.299, 0.587, 0.114])) # H x␣
 ↪W
im3_sphere = merge_channels(im3_sphere_chnl, im3_sphere_chnl, im3_sphere_chnl)␣
 ↪# H x W x 3
im3_sphere_normalized = normalize(im3_sphere) # H x W x 3
# Image 4
im4_sphere_chnl = data_sphere["im4"].dot(np.array([0.299, 0.587, 0.114])) # H x␣
 ↪W
im4_sphere = merge_channels(im4_sphere_chnl, im4_sphere_chnl, im4_sphere_chnl)␣
 ↪# H x W x 3
im4_sphere_normalized = normalize(im4_sphere) # H x W x 3


# Pack the images
images_sphere = []
images_sphere.append(im1_sphere_normalized[:, :, 0]) # H x W
images_sphere.append(im2_sphere_normalized[:, :, 0]) # H x W
images_sphere.append(im3_sphere_normalized[:, :, 0]) # H x W
images_sphere.append(im4_sphere_normalized[:, :, 0]) # H x W
images_sphere = np.array(images_sphere) # n x H x W

## Mask sphere.
```

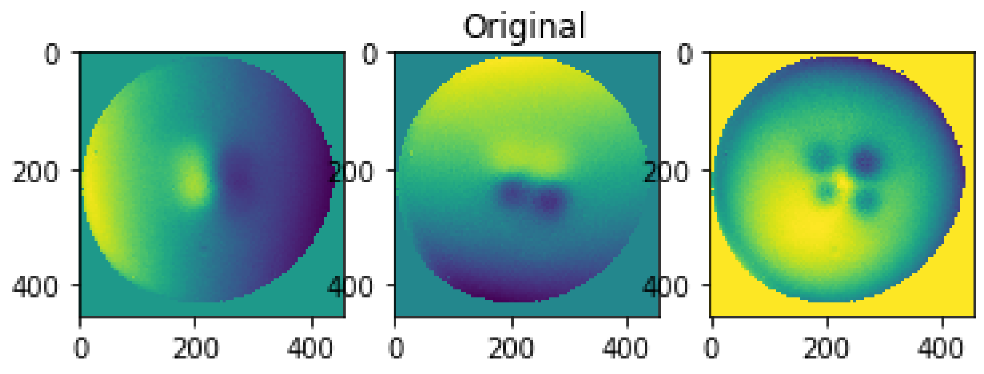
```
mask_sphere = np.ones((im1_sphere_normalized[:, :, 0].shape))

for height in range(mask_sphere.shape[0]):
    for width in range(mask_sphere.shape[1]):
        if im1_sphere_normalized[height, width, 0] <= mask_thres or␣
 ↪im2_sphere_normalized[height, width, 0] <= mask_thres or␣
 ↪im3_sphere_normalized[height, width, 0] <= mask_thres or␣
 ↪im4_sphere_normalized[height, width, 0] <= mask_thres:
            mask_sphere[height,width] = 0

# Orignial Sphere
albedo, normals, depth, horn = photometric_stereo(images_sphere, lights_sphere,␣
 ↪mask_sphere, horn_niter = 25000)

## Diffused Sphere
# Image 1
S_sphere_im1, G_sphere_im1 = RGBToSUV(data_sphere["im1"], np.
 ↪hstack((data_sphere["c"][0][0],
                                        data_sphere["c"][1][0],
                                        data_sphere["c"][2][0])))

# Image 2
S_sphere_im2, G_sphere_im2 = RGBToSUV(data_sphere["im2"], np.
 ↪hstack((data_sphere["c"][0][0],
                                        data_sphere["c"][1][0],
                                        data_sphere["c"][2][0])))
# Image 3
S_sphere_im3, G_sphere_im3 = RGBToSUV(data_sphere["im3"], np.
 ↪hstack((data_sphere["c"][0][0],
                                        data_sphere["c"][1][0],
                                        data_sphere["c"][2][0])))
# Image 4
S_sphere_im4, G_sphere_im4 = RGBToSUV(data_sphere["im4"], np.
 ↪hstack((data_sphere["c"][0][0],
                                        data_sphere["c"][1][0],
                                        data_sphere["c"][2][0])))

# Pack the images
images_sphere_diff = []
images_sphere_diff.append(G_sphere_im1) # H x W
images_sphere_diff.append(G_sphere_im2) # H x W
images_sphere_diff.append(G_sphere_im3) # H x W
images_sphere_diff.append(G_sphere_im4) # H x W
images_sphere_diff = np.array(images_sphere_diff) # n x H x W

# Diffuse Rendered Outputs
```

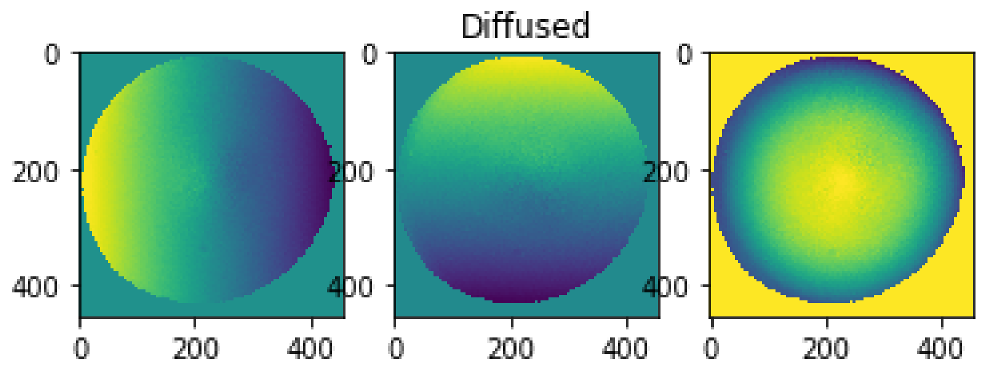
```
albedo_diff, normals_diff, depth_diff, horn_diff =␣
↪photometric_stereo(images_sphere_diff, lights_sphere, mask_sphere,␣
↪horn_niter= 25000)

# Plot
visualize_ori_diff(albedo, albedo_diff, normals, normals_diff, depth,␣
↪depth_diff, horn, horn_diff)
```

```
Albedo Map
```

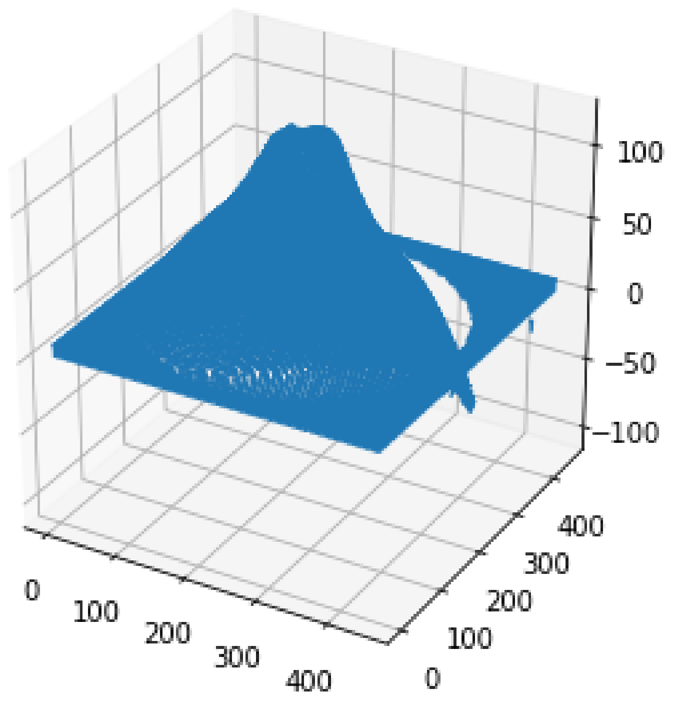


```
Normals as Three Separate Channels
```

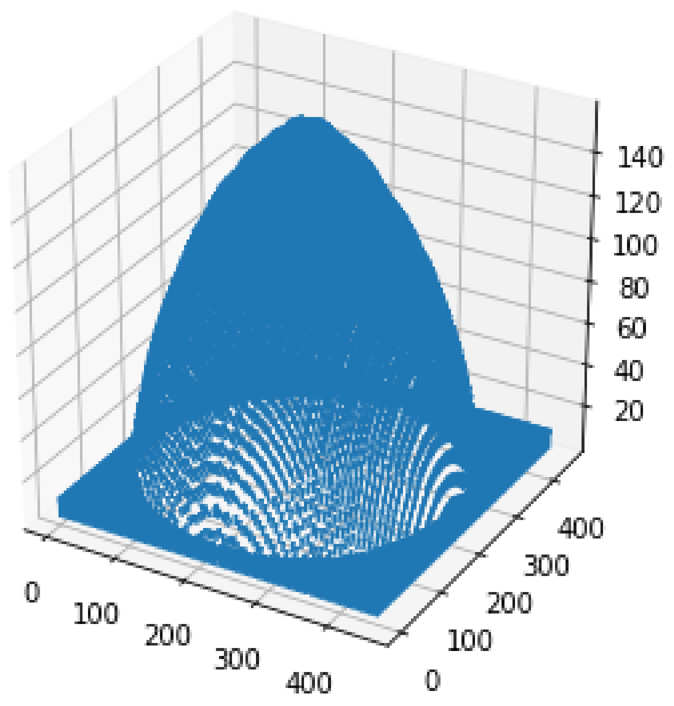




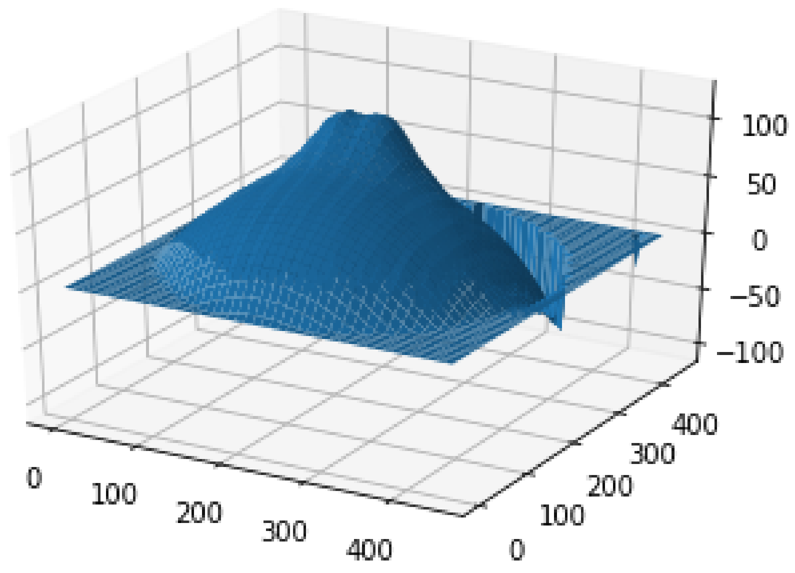
```
Normals as Quiver
Original
```



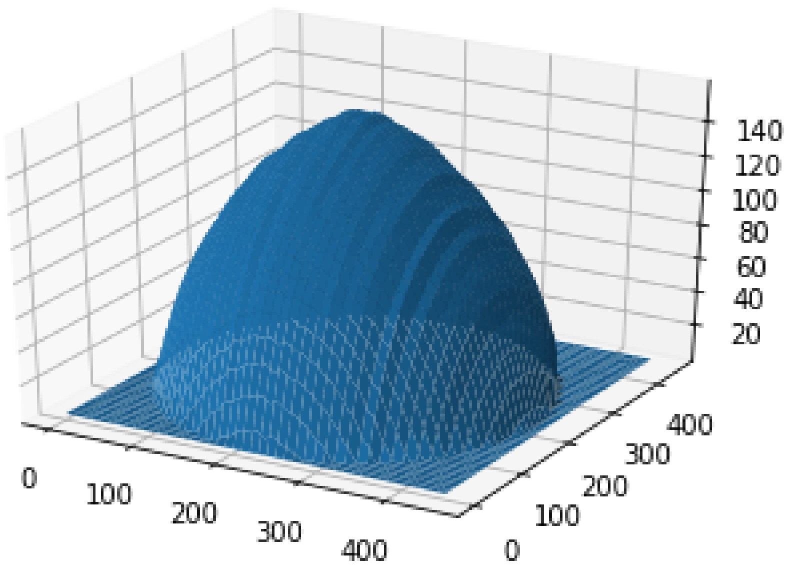
```
Normals as Quiver
Diffused
```



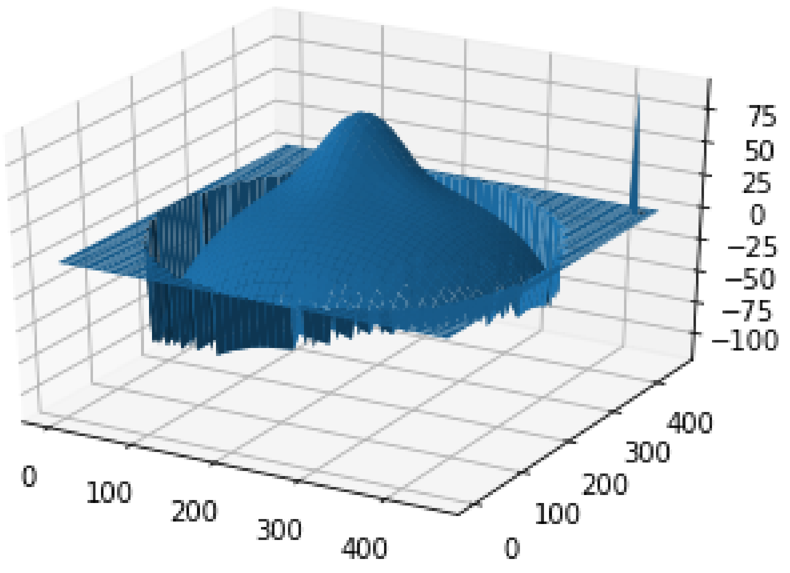
```
Wireframe Depth Map
Original
```



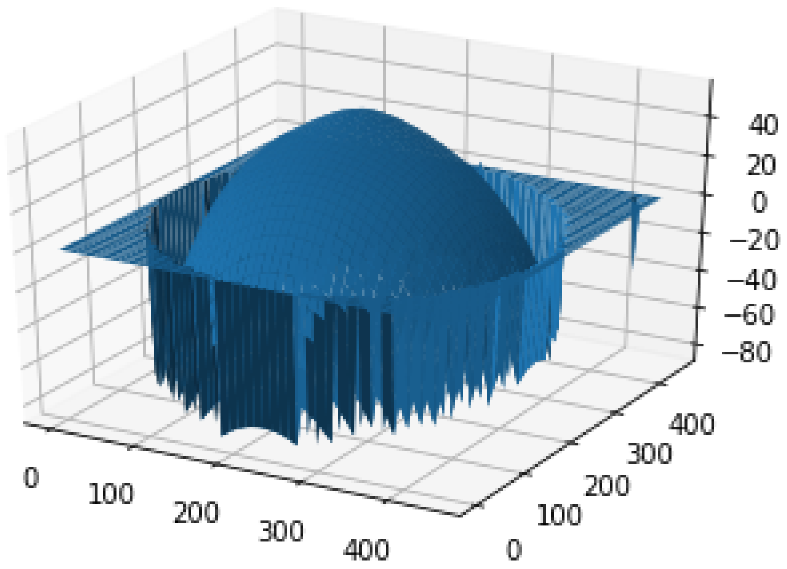
```
Wireframe Depth Map
Diffused
```



```
Horn's Integration
Original
```



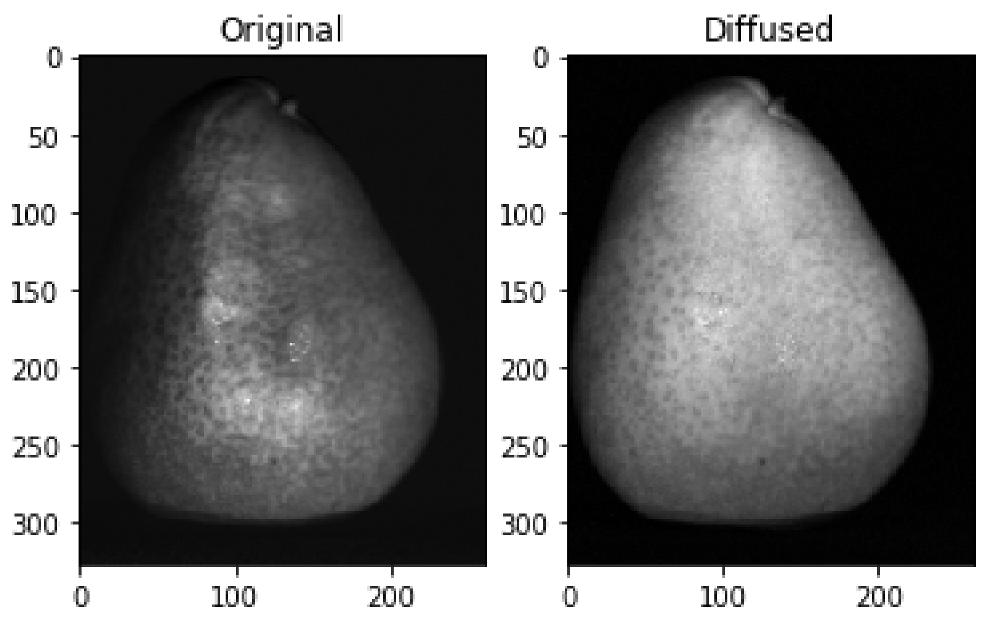
```
Horn's Integration
Diffused
```



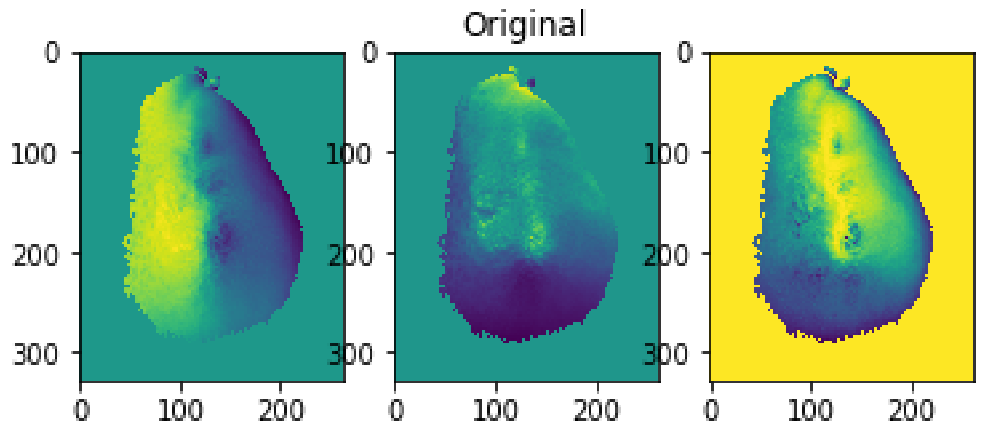
[211]:
```
### Pear
# Open the pear file
pickle_in_pear = open("specular_pear.pickle", "rb")

# Load the pear file
data_pear = pickle.load(pickle_in_pear, encoding="latin1")

# Pear Light Sources
lights_pear = np.vstack((data_pear["l1"], data_pear["l2"], data_pear["l3"],␣
↪data_pear["l4"]))

## Convert the colored image to grayscale
# Image 1
im1_pear_chnl = data_pear["im1"].dot(np.array([0.299, 0.587, 0.114])) # H x W
im1_pear = merge_channels(im1_pear_chnl, im1_pear_chnl, im1_pear_chnl) # H x W␣
↪x 3
im1_pear_normalized = normalize(im1_pear) # H x W x 3
# Image 2
im2_pear_chnl = data_pear["im2"].dot(np.array([0.299, 0.587, 0.114])) # H x W
im2_pear = merge_channels(im2_pear_chnl, im2_pear_chnl, im2_pear_chnl) # H x W␣
↪x 3
im2_pear_normalized = normalize(im2_pear) # H x W x 3
# Image 3
im3_pear_chnl = data_pear["im3"].dot(np.array([0.299, 0.587, 0.114])) # H x W
im3_pear = merge_channels(im3_pear_chnl, im3_pear_chnl, im3_pear_chnl) # H x W␣
↪x 3
```

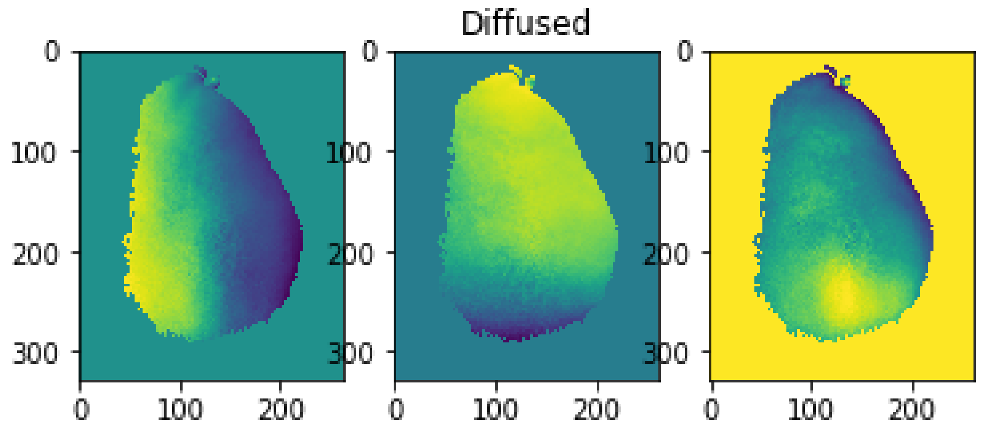
```
im3_pear_normalized = normalize(im3_pear) # H x W x 3
# Image 4
im4_pear_chnl = data_pear["im4"].dot(np.array([0.299, 0.587, 0.114])) # H x W
im4_pear = merge_channels(im4_pear_chnl, im4_pear_chnl, im4_pear_chnl) # H x W␣
 ↪x 3
im4_pear_normalized = normalize(im4_pear) # H x W x 3


# Pack the images
images_pear = []
images_pear.append(im1_pear_normalized[:, :, 0]) # H x W
images_pear.append(im2_pear_normalized[:, :, 0]) # H x W
images_pear.append(im3_pear_normalized[:, :, 0]) # H x W
images_pear.append(im4_pear_normalized[:, :, 0]) # H x W
images_pear = np.array(images_pear) # n x H x W

## Mask pear.
mask_pear = np.ones((im1_pear_normalized[:, :, 0].shape))

for height in range(mask_pear.shape[0]):
    for width in range(mask_pear.shape[1]):
        if im1_pear_normalized[height, width, 0] <= mask_thres or␣
 ↪im2_pear_normalized[height, width, 0] <= mask_thres or␣
 ↪im3_pear_normalized[height, width, 0] <= mask_thres or␣
 ↪im4_pear_normalized[height, width, 0] <= mask_thres:
            mask_pear[height,width] = 0

# Orignial pear
albedo_pear, normals_pear, depth_pear, horn_pear =␣
 ↪photometric_stereo(images_pear, lights_pear, mask_pear, horn_niter = 100000)

## Diffused pear
# Image 1
S_pear_im1, G_pear_im1 = RGBToSUV(data_pear["im1"], np.
 ↪hstack((data_pear["c"][0][0],
                                        data_pear["c"][1][0],
                                        data_pear["c"][2][0])))

# Image 2
S_pear_im2, G_pear_im2 = RGBToSUV(data_pear["im2"], np.
 ↪hstack((data_pear["c"][0][0],
                                        data_pear["c"][1][0],
                                        data_pear["c"][2][0])))
# Image 3
S_pear_im3, G_pear_im3 = RGBToSUV(data_pear["im3"], np.
 ↪hstack((data_pear["c"][0][0],
                                                            data_pear["c"][1][0],
                                                            data_pear["c"][2][0])))
# Image 4
S_pear_im4, G_pear_im4 = RGBToSUV(data_pear["im4"], np.
 ↪hstack((data_pear["c"][0][0],
                                                            data_pear["c"][1][0],
                                                            data_pear["c"][2][0])))

# Pack the images
images_pear_diff = []
images_pear_diff.append(G_pear_im1) # H x W
images_pear_diff.append(G_pear_im2) # H x W
images_pear_diff.append(G_pear_im3) # H x W
images_pear_diff.append(G_pear_im4) # H x W
images_pear_diff = np.array(images_pear_diff) # n x H x W

# Diffuse Rendered Outputs
albedo_pear_diff, normals_pear_diff, depth_pear_diff, horn_pear_diff =␣
 ↪photometric_stereo(images_pear_diff, lights_pear, mask_pear, horn_niter=␣
 ↪100000)

# Plot
visualize_ori_diff(albedo_pear, albedo_pear_diff, normals_pear,␣
 ↪normals_pear_diff, depth_pear, depth_pear_diff, horn_pear, horn_pear_diff)
```

```
Albedo Map
```

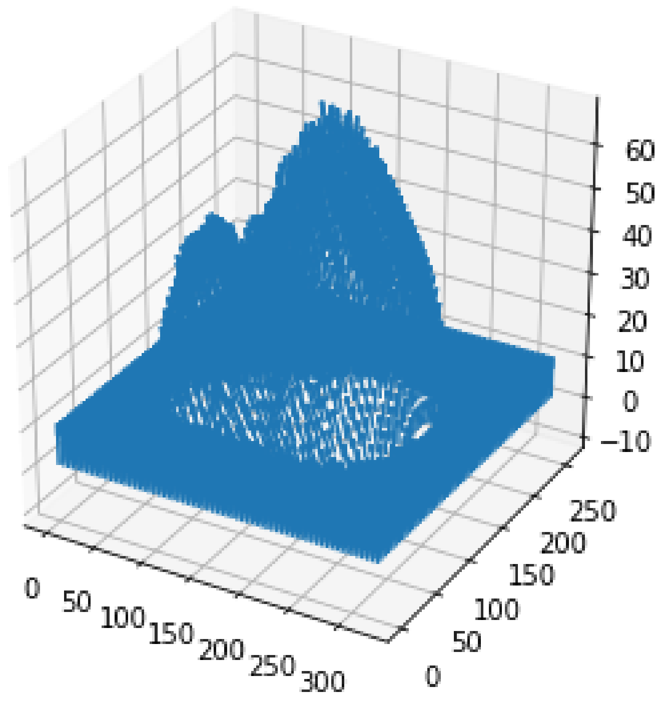


```
Normals as Three Separate Channels
```

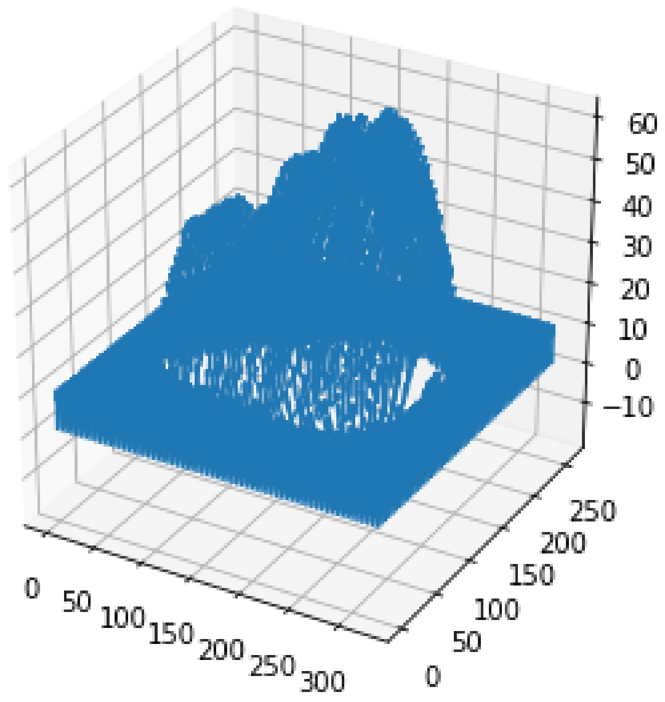




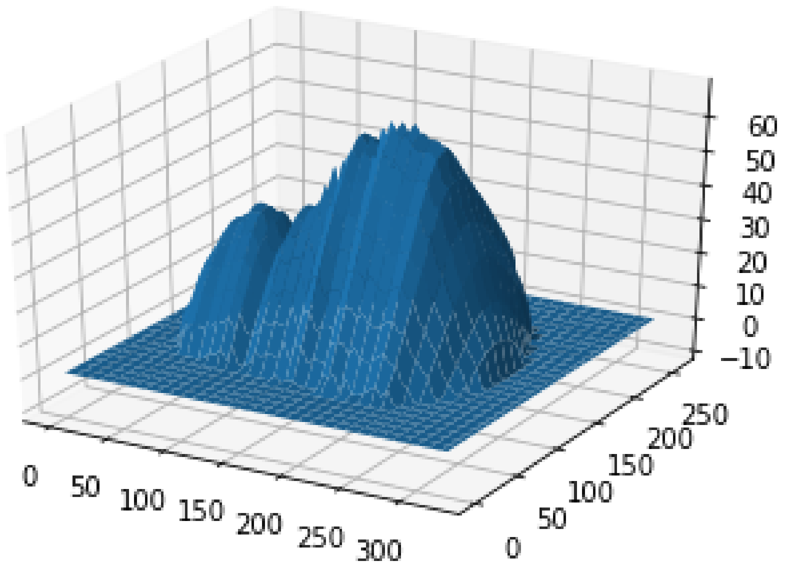
```
Normals as Quiver
Original
```



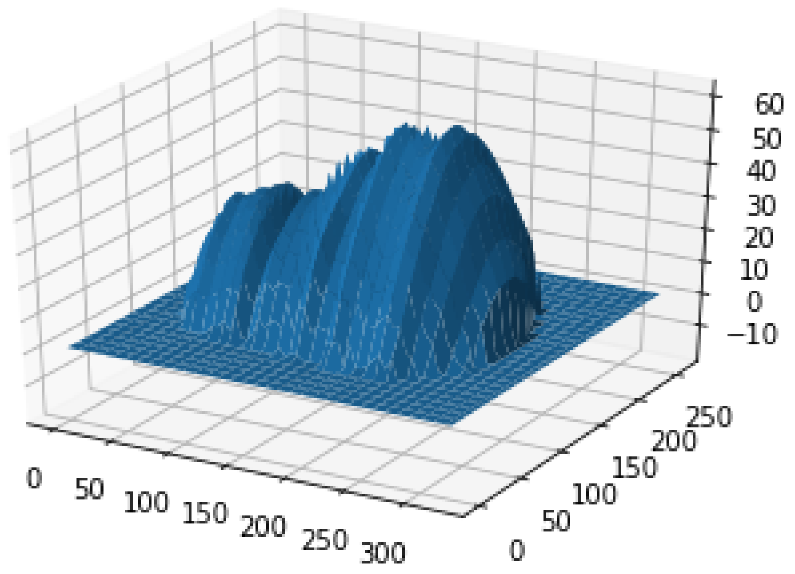
```
Normals as Quiver
Diffused
```



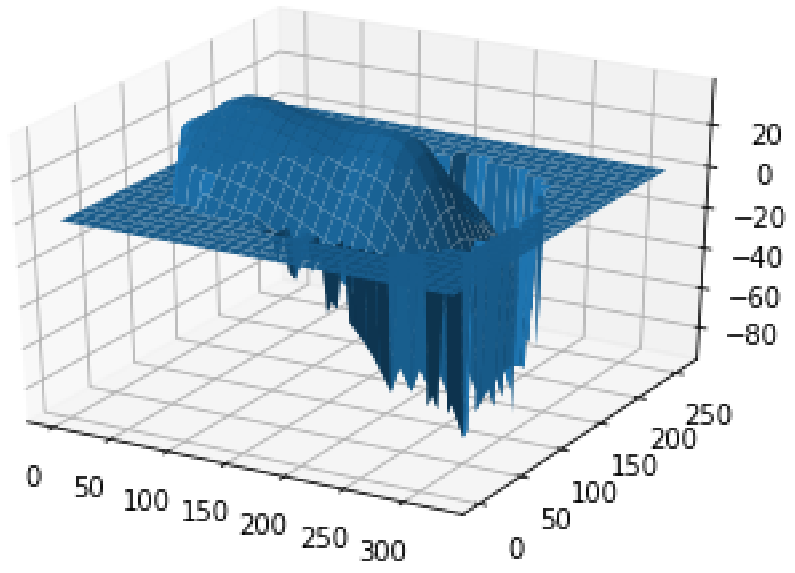
```
Wireframe Depth Map
Original
```



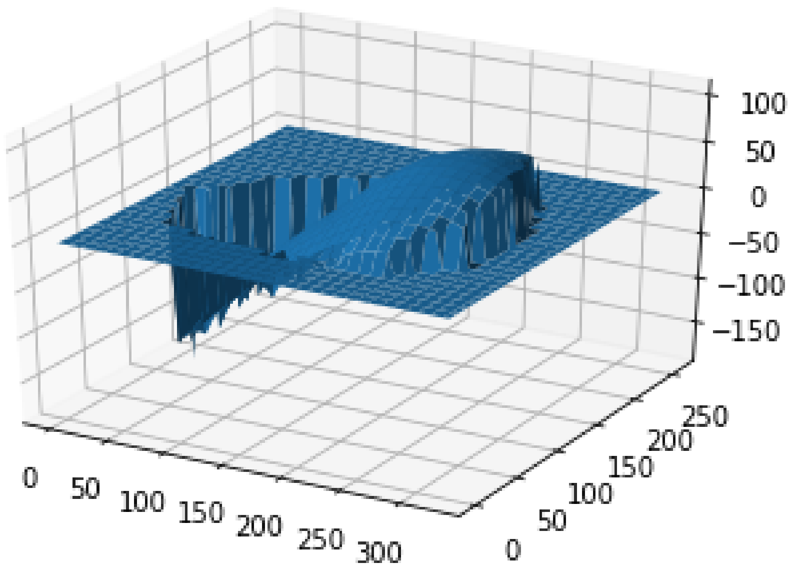
```
Wireframe Depth Map
Diffused
```



```
Horn's Integration
Original
```



```
Horn's Integration
Diffused
```


100
50
0
−50
−100
−150
0
50
100
150
200
250
300
0
50
100
150
200
250

```
[ ]:
```