CSE252A_FA25_assignment_1

October 22, 2025

1 CSE 252A Computer Vision I, FA 2025 - Assignment 1

Instructor: Ben Ochoa

Assigment due: Wed, Oct 22, 11:59 PM

Name: Mehul Maheshwari

PID: A16478228

1.1 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.
- 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.
- You must **submit 3 files: the Notebook, the PDF and the python file** (i.e. the `.ipynb`, the `.pdf` and the `.py` files) on Gradescope. **You must mark each problem on Gradescope in the pdf.**
  - To convert the notebook to PDF, you can choose one way below:
    - You may first export the notebook as HTML, and then print the web page as PDF
      - e.g., in Chrome: File → Save and Export Notebook as → "HTML"; or in VScode: Open the Command Palette by pressing Ctrl+Shift+P (Windows/Linux) or Cmd+Shift+P (macOS), search for Jupyter: Export to HTML
      - Open the saved web page and right click → Print… → Choose "Destination: Save as PDF" and click "Save")
    - If you have XeTex installed on your machine, you may directly export the notebook as PDF: e.g., in Chrome, File → Save and Export Notebook as → "PDF"

    - You may use nbconvert to convert the ipynb file to pdf using the following command `jupyter nbconvert --allow-chromium-download --to webpdf filename.ipynb`
  - To convert the notebook to python file, you can choose one way below:
    - You may directly export the notebook as py: e.g., in Chrome, File → Save and Export Notebook as → "Executable script"; or in VScode: Open the Command Palette and search for Jupyter: Export to Python Script
    - You may use nbconvert to convert the ipynb file to python file using the following command `jupyter nbconvert --to script filename.ipynb --output output_filename.py`
- Please make sure the content in each cell (e.g. code, output images, printed results, etc.) are clearly visible and are not cut-out or partially cropped in your final PDF file.
- While submitting on gradescope, please make sure to assign the relevant pages in your PDF submission for each problem.

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 Problem 1: Geometry (16 points)

Note: The solution **must be** typed using Markdown or Latex, handwritten answers will not be accepted.

Consider a line in the 2D plane, whose equation is given by $a\tilde{x} + b\tilde{y} + c = 0$, where $\mathbf{l} = (a, b, c)^\top$ and $\mathbf{x} = (\tilde{x}, \tilde{y}, 1)^\top$. Noticing that $\mathbf{x}$ is a homogeneous representation of $\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y})^\top$, we can view $\mathbf{l}$ 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 $\lambda(a, b, c)^\top$ with $\lambda \neq 0$ represents the same line.

- 1.1 Prove $\mathbf{x}^\top\mathbf{l} + \mathbf{l}^\top\mathbf{x} = 0$, if a point $\mathbf{x}$ in homogeneous coordinates lies on the homogeneous line $\mathbf{l}$. [4 points]

Answer:

$\mathbf{x}^\top\mathbf{l} + \mathbf{l}^\top\mathbf{x} = [\hat{x}, \hat{y}, 1][a, b, c]^\top + [a, b, c][\hat{x}, \hat{y}, 1]^\top = a\hat{x} + b\hat{y} + c + a\hat{x} + b\hat{y} + c = 2(a\hat{x} + b\hat{y} + c)$. Since we are told that $\mathbf{x}$ lies on the line $\mathbf{l}$, it satisfies the line equation $a\hat{x} + b\hat{y} + c = 0$. Thus, $2(ax + by + c) = 2(0) = 0$. Therefore, $\mathbf{x}^\top\mathbf{l} + \mathbf{l}^\top\mathbf{x} = 0$.

- 1.2 What is the line, in homogenous coordinates, joining the inhomogeneous points $\tilde{\mathbf{x}}_1 = (3, 2)^\top$ and $\tilde{\mathbf{x}}_2 = (4, -1)^\top$. [4 points]

Answer:

First we need to convert the points to homogeneous coordinates:
$x_1 = [3, 2, 1]^T$, $x_2 = [4, -1, 1]^T$.

Then the line joining these points is given by the cross product:

$$l = x_1 \times x_2 = (2 \cdot 1 - 1 \cdot (-1),\ -(3 \cdot 1 - 1 \cdot 4),\ 3 \cdot (-1) - 2 \cdot 4) = (3, 1, -11)$$

The homogeneous line equation is: $3x + y - 11 = 0$

- 1.3 Consider the intersection of two lines $\mathbf{l}_1$ and $\mathbf{l}_2$. Prove that the homogeneous point of intersection, $\mathbf{x}$, of two homogeneous lines $\mathbf{l}_1$ and $\mathbf{l}_2$ is $\mathbf{x} = \mathbf{l}_1 \times \mathbf{l}_2$, where $\times$ is the vector cross product. [4 points]

Answer:

We know that a point lies on a line if $l \cdot x = 0$. Therefore, the intersection point is the point that satisfies both $l_1 \cdot x = 0$ and $l_2 \cdot x = 0$.

Now, we need to show that this same x is the cross product. To do this, we can plug in the cross product into both equations:

$l_1 \cdot (l_1 \times l_2) = 0$ and $l_2 \cdot (l_1 \times l_2) = 0$.

Now consider $x = l_1 \times l_2$. By definition of cross product, x is orthogonal to both l_1 and l_2. Any two lines that are orthogonal necessarily have a dot product of 0. Therefore the conditions $l_1 \cdot (l_1 \times l_2) = 0$ and $l_2 \cdot (l_1 \times l_2) = 0$ are both satisfied. This is what we wanted to show, demonstrating that $x = l_1 \times l_2$ is the point that intersects both lines.

- 1.4 Consider the two lines $\tilde{x} - 3\tilde{y} + 1 = 0$ and $4\tilde{x} - 2\tilde{y} - 2 = 0$. Find their intersection in homogeneous coordinates. Next, convert this homogeneous point $\mathbf{x}$ to inhomogeneous coordinates $\tilde{\mathbf{x}}$ and report the 2D point of intersection. [4 points]

Answer:

First, lets calculate their intersection point in homogeneous coordinates. i could use cross product here, but simple substitution is easier:

x = 3y - 1,
4(3y - 1) - 2y - 2 = 0,
10y - 6 = 0,
y = 6/10 = 0.6,
x = 3(0.6) - 1 = 0.8.

In homogeneous coordinates this is (0.6, 0.8, 1).

Now to convert to inhomogeneous coordinates, we have to do: (x, y, z) → (x/z, y/z).

Therefore the homogeneous coordinate point of intersection is (0.6/1, 0.8/1) = (0.6, 0.8).

1.3 Problem 2: Image Formation and Rigid Body Transformations (20 points)

In this problem we will practice rigid body transformations and image formations through the projective camera model. The goal will be to image the following four points $\tilde{\mathbf{X}}_1 = (-4, -7, 2)^\top$, $\tilde{\mathbf{X}}_2 = (4, -7, 2)^\top$, $\tilde{\mathbf{X}}_3 = (4, 7, 2)^\top$, and $\tilde{\mathbf{X}}_4 = (-4, 7, 2)^\top$ in the world coordinate frame. First, recall the following formula for rigid body transformation

$$\tilde{\mathbf{X}}_{\text{cam}} = \mathtt{R}\tilde{\mathbf{X}} + \mathbf{t}$$

Where $\tilde{\mathbf{X}}_{\text{cam}}$ is a point in the camera coordinate system, $\tilde{\mathbf{X}}$ is a point in the world coordinate frame, and $\mathtt{R}$ and $\mathbf{t}$ are the rotation and translation that transform points from the world coordinate frame

to the camera coordinate frame. Together, $\mathtt{R}$ and $\mathbf{t}$ are the *extrinsic* camera parameters. Once transformed to the camera coordinate frame, the points can be imaged using the 3-by-3 camera calibration matrix $\mathtt{K}$, which embodies the *intrinsic* camera parameters, and the canonical projection matrix $[\mathtt{I} \,|\, \mathbf{0}]$. Given $\mathtt{K}, \mathtt{R}$, and $\mathbf{t}$, the image of a point $\tilde{\mathbf{X}}$ is $\mathbf{x} = \mathtt{K}[\mathtt{I} \,|\, \mathbf{0}]\mathbf{X}_{\text{cam}} = \mathtt{K}[\mathtt{R} \,|\, \mathbf{t}]\mathbf{X}$, where the homogeneous point $\mathbf{X}_{\text{cam}} = (\tilde{\mathbf{X}}_{\text{cam}}^{\top}, 1)^{\top}$ and $\mathbf{X} = (\tilde{\mathbf{X}}^{\top}, 1)^{\top}$. We will consider four different settings of focal length, viewing angles and camera positions. For each of the settings, calculate:

a). The matrix $[\mathtt{R} \,|\, \mathbf{t}]$

b). The camera calibration matrix $\mathtt{K}$

c). The image of the four vertices and plot using the supplied **plot_points** function

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

The four settings are:

1. **[No rigid body transformation].** Focal length = 2. The optical axis of the camera is aligned with the Z-axis.
2. **[Translation].** Focal length = 2. $\mathbf{t} = (0, 0, 2)^{\top}$. The optical axis of the camera is aligned with the Z-axis.
3. **[Translation and Rotation].** Focal length = 2. $\mathtt{R}$ embodies a 30 degree rotation about the Z-axis. $\mathbf{t} = (0, 1, 2)^{\top}$.
4. **[Translation and Rotation, long distance].** Focal length = 4. $\mathtt{R}$ embodies a 45 degree rotation about the Z-axis and then a 30 degree rotation about the X-axis. $\mathbf{t} = (0, 0, 3)^{\top}$.

We will not use a full camera calibration matrix (e.g., that maps centimeters to pixels, and specifies the coordinates of the principal point), but only parameterize this with the focal length f. In other words: the only parameter in the camera calibration 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.

2.1 Coordinate convertion and projection [6 points]

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import math


def to_homog(points_inhomog):
    """ convert points from inhomogeneous to homogeneous

    inputs:
    points_inhomog is a dxn matrix where n is the number of d dimensional␣
 ↪inhomogeneous points
```

```
    (e.g., d = 3 for 3D inhomogeneous points)

    outputs:
    homo_points is a (d+1)xn matrix of n d-dimensional homogeneous points
    """
    #### Write your code here. ####

    # for each point in points_inhomog, we need to add a 1 to the end of the␣
 ↪point
    homo_points = np.vstack((points_inhomog, np.ones((1, points_inhomog.
 ↪shape[1]))))

    return homo_points


def from_homog(points_homog):
    """
    convert points from homogeneous to inhomogeneous

    inputs:
    points_homog is a (d+1)xn matrix of n d-dimensional homogeous points (e.g.,␣
 ↪d = 2
    for 2D homogeneous points)

    outputs:
    inhomo_points is a dxn matrix of n d-dimensional inhomogeous points
    """
    #### Write your code here. ####

    inhomo_points = np.zeros((points_homog.shape[0]-1, points_homog.shape[1]))
    # for each point in points_homog, we need to divide each element by the␣
 ↪last element
    for i in range(points_homog.shape[1]):
        last_element = points_homog[-1, i]
        inhomo_points[:, i] = points_homog[:-1, i] / last_element

    return inhomo_points


def project_points(M_int, M_ext, pts):
    """
    project 3D inhomogeneous points to 2D inhomogeneous points

    inputs:
       M_int - 3x3 intrinsic camera matrix
       M_ext - 3x4 extrinsic camera matrix
       pts - 3xn inhomogeneous points
```

```
    outputs:
       pts_2d - 2xn inhomogeneous points
    """

    #### Write your code here. ####



    # to project the points, we need to do intrinsic *  extrinsic * points

    # since M_ext is 3x4, we have to homogenize the points first
    pts_homog = to_homog(pts)

    projected_points = M_int @ M_ext @ pts_homog

    # now we need to go back to inhomogeneous points
    pts_2d = from_homog(projected_points)

    return pts_2d
```

2.2 Compute the intrisic and extrinsic matrices [6 points]

[3]:
```
def intrinsic_cam_mat(f):
    """
    K = [f 0 0
         0 f 0
         0 0 1]
    """
    # given the focal length, compute the intrinsic camera matrix

    #### Write your code here. ####

    int_cam_mat = np.array([[f, 0, 0], [0, f, 0], [0, 0, 1]])
    return int_cam_mat

def extrinsic_cam_mat(angles, t, order='xyz'):
    """
        ext_cam_mat = [R|t]
    """
    # Compute the extrinsic camera matrix
    #
    # inputs:
    #    angles - a tuple of angles (alpha, beta, gamma), representing the␣
 ↪rotation
    #    angles around x-axis, y-axis, and z-axis repectively in radians.
    #    t - a 3x1 translation vector
```

```
    #
    # outputs:
    #    ext_cam_mat - 3x4 extrinsic camera matrix


    #### Write your code here. ####
    alpha = angles[0]
    rot_x = np.array([
        [1, 0, 0],
        [0, np.cos(alpha), -np.sin(alpha)],
        [0, np.sin(alpha), np.cos(alpha)]
    ])
    beta = angles[1]
    rot_y = np.array([
        [np.cos(beta), 0, np.sin(beta)],
        [0, 1, 0],
        [-np.sin(beta), 0, np.cos(beta)]
    ])
    gamma = angles[2]
    rot_z = np.array([
        [np.cos(gamma), -np.sin(gamma), 0],
        [np.sin(gamma), np.cos(gamma), 0],
        [0, 0, 1]
    ])



    # rotation is order dependent:
    R = np.eye(3)
    for axis in reversed(order):
        if axis == 'x':
            R = rot_x @ R
        elif axis == 'y':
            R = rot_y @ R
        elif axis == 'z':
            R = rot_z @ R


    t = np.array(t).reshape(3,1) # to ensure its a column vector

    ext_cam_mat = np.hstack((R, t))

    return ext_cam_mat
```

2.3 Define the 4 cameras [8 points]

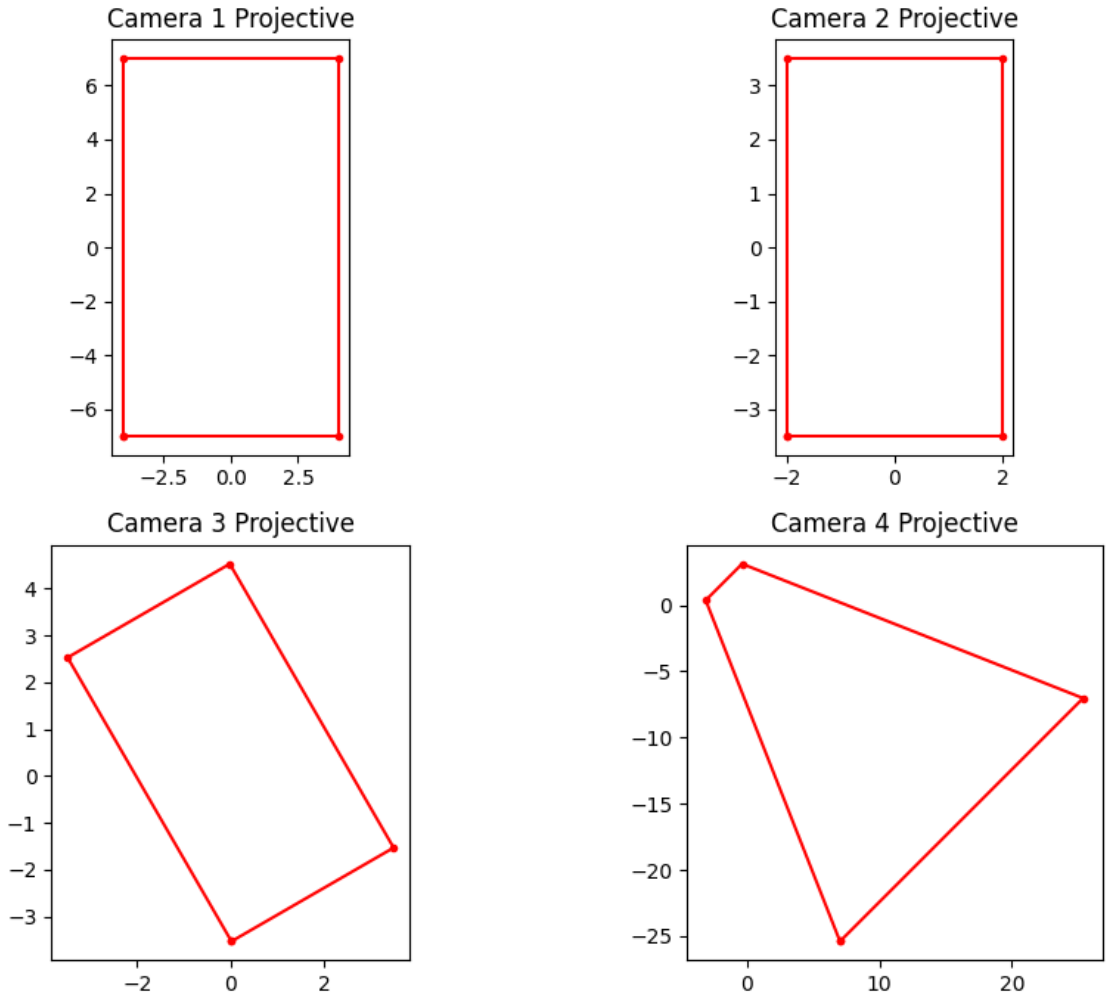
[4]:
```
# Change the three matrices for the four cases as described in the problem
# in the four camera functions given below. Make sure that we can see the␣
 ↪formula
# (if one exists) being used to fill in the matrices. Feel free to document with
# comments any thing you feel the need to explain.


"""
Reacall:
The four settings are:

1. **[No rigid body transformation].** Focal  length  =  2. The  optical  axis ␣
 ↪of  the  camera  is aligned with the $Z$-axis.
2. **[Translation].** Focal  length  =  2. $\boldsymbol{\mathrm{t}} = (0, 0,␣
 ↪2)^\top$. The optical axis of the camera is aligned with the $Z$-axis.
3. **[Translation and Rotation].** Focal length = 2. $\mathtt{R}$ embodies a 30␣
 ↪degree rotation about the $Z$-axis. $\boldsymbol{\mathrm{t}} = (0, 1,␣
 ↪2)^\top$.
4. **[Translation and Rotation, long distance].** Focal length = 4.␣
 ↪$\mathtt{R}$ embodies a 45 degree rotation about the $Z$-axis and then a 30␣
 ↪degree rotation about the $X$-axis. $\boldsymbol{\mathrm{t}} = (0, 0,␣
 ↪3)^\top$.
"""

# IMPORTANT: recall radians = degrees * (pi/180)

def camera1():
    #### Write your code here. ####

    # no rigid body transformation, optical axis is aligned with Z-axis
    f=2
    P_int_proj = intrinsic_cam_mat(f)
    rotation_angles = (0,0,0)
    translation = np.array([[0], [0], [0]])
    P_ext = extrinsic_cam_mat(rotation_angles, translation)
    return P_int_proj, P_ext

def camera2():
    #### Write your code here. ####
    # focal length = 2, translation = (0,0,2), optical axis is aligned with␣
 ↪Z-axis
    f=2
    P_int_proj = intrinsic_cam_mat(f)
    rotation_angles = (0,0,0)
    translation = np.array([[0], [0], [2]])
    P_ext = extrinsic_cam_mat(rotation_angles, translation)
```

```
    return P_int_proj, P_ext

def camera3():
    #### Write your code here. ####
    # focal length = 2, translation = (0,1,2), rotation = 30 degrees about␣
 ↪Z-axis
    f=2
    P_int_proj = intrinsic_cam_mat(f)
    z_angle = 30*np.pi/180
    rotation_angles = (0,0,z_angle)
    translation = np.array([[0], [1], [2]])
    P_ext = extrinsic_cam_mat(rotation_angles, translation)

    return P_int_proj, P_ext

def camera4():
    #### Write your code here. ####
    # focal length = 4, translation = (0,0,3), rotation = 45 degrees about␣
 ↪Z-axis and then 30 degrees about X-axis, translation = (0,0,3)
    f=4
    P_int_proj = intrinsic_cam_mat(f)
    z_angle = 45*np.pi/180
    x_angle = 30*np.pi/180
    rotation_angles = (x_angle, 0, z_angle)
    translation = np.array([[0], [0], [3]])
    P_ext = extrinsic_cam_mat(rotation_angles, translation, order='zx')

    return P_int_proj, P_ext
```

Plot the 4 images of the 4 vertices

[5]:
```
##########################################################
# test code. Do not modify
##########################################################

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

def main():
    point1 = np.array([[-4,-7,2]]).T
    point2 = np.array([[4,-7,2]]).T
```

```
    point3 = np.array([[4,7,2]]).T
    point4 = np.array([[-4,7,2]]).T
    points = np.hstack((point1,point2,point3,point4))

    fig = plt.figure(figsize=(10,7))
    for i, camera in enumerate([camera1, camera2, camera3, camera4]):
        M_int_proj, M_ext = camera()
        ax = fig.add_subplot(2,2,i+1)
        plot_points(ax, project_points(M_int_proj, M_ext, points),␣
↪title='Camera %d Projective'%(i+1), axis=[-1,1,-1,1])
    plt.tight_layout()
    plt.show()

main()
```



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

(Here distortion means the perspective effect that causes the projected rectangle to appear skewed or trapezoidal.)

Answer:

We observe distortions in the projection because the angle at which we view the rectangle has changed, and so how it appears to us when projected down to 2d would changes accordingly. In real life, if you are looking at a piece of paper from close to any edge, it would appear trapezoidal as well.

1.4 Problem 3: Surface Rendering [20 pts]

In this portion of the assignment we will be exploring different methods of approximating local reflectance of objects in a scene. This 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 3.1 Plot the normals [6 pts]

In this part, you are required to plot the normals.

The data needed for this problem are normals and masks of two objects: a sphere and a pear.

Normals are stored in `normals2.pickle`. This file contains a list of 4 arrays: 1) Normal Vectors for the sphere with specularities removed (Diffuse component)
2) Normal Vector for the sphere (Original)
3) Normal Vectors for the pear with specularities removed (Diffuse component)
4) Normal vectors for the pear (Original)

Masks are stored in `masks.pkl`. This file contains a list that stores two arrays: 1) Mask of the sphere 2) Mask of the pear

A mask is a $H \times W$ matrix with each element being either 1 or 0, indicating if a pixel at given location is sphere/pear or not. It is provided for you to remove the background of the rendered sphere/pear.

Please implement the function `plot_normals` with background removed, and plot the normals using this function. You can use subplots to display the two figures — the diffuse object normal and the original object normal — side by side. (Refer to the usage in Problem 2). To convert normal values to colors for display, you must use `normal2rgb` function.

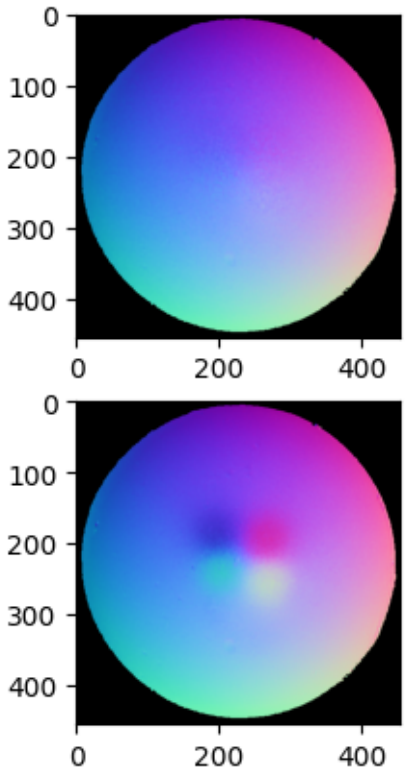
(Don't worry about pickle, we've prepared the code to read the files.)

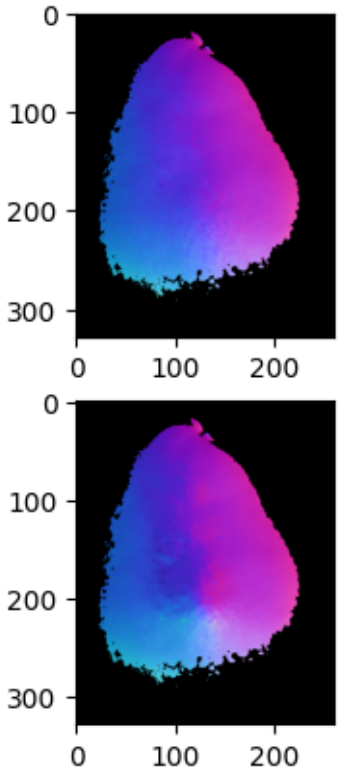
[6]:
```
import numpy as np
import matplotlib.pyplot as plt
################################################
# Don't modify this function
################################################
def normal2rgb(normals):
    return (normals + 1) / 2
```

```
def plot_normals(diffuse_normals, original_normals, mask):
    """
    Plot the normals using subplots. The first subplot is the diffuse normals,
    and the second subplot is the original normals. You must use normal2rgb to
    convert the normals to RGB format before plotting.
    Inputs:
    - diffuse_normals: H x W x 3 array of diffuse normals
    - original_normals: H x W x 3 array of original normals
    - mask: H x W array of binary mask (1 for object, 0 for background)
    """
    #### Write your code here. ####

    # first, get normals in rgb format
    diffuse_normals_rgb = normal2rgb(diffuse_normals)
    original_normals_rgb = normal2rgb(original_normals)

    # then, remove background via mask
    diffuse_normals_rgb[:, :, 0] *= mask
    diffuse_normals_rgb[:, :, 1] *= mask
    diffuse_normals_rgb[:, :, 2] *= mask

    original_normals_rgb[:, :, 0] *= mask
    original_normals_rgb[:, :, 1] *= mask
    original_normals_rgb[:, :, 2] *= mask

    # then, plot the normals
    plt.subplot(2, 1, 1)
    plt.imshow(diffuse_normals_rgb)
    plt.subplot(2, 1, 2)
    plt.imshow(original_normals_rgb)
    plt.show()
```

[7]:
```
#Plot the normals for the sphere and pear for both the normal and diffuse␣
 ↪components.
#1 : Load the different normals
import pickle

with open('normals2.pickle', 'rb') as f:
    data_normal = pickle.load(f)

with open('masks.pkl', 'rb') as h:
    data_mask = pickle.load(h)

#2 : Plot the normals using plot_normals
# What do you observe? What are the differences between the diffuse component␣
 ↪and the original images shown?
```

```
# (Just something to think about, no need to provide an answer)

#### Write your code here. ####
sphere_diffuse = data_normal[0]
sphere_original = data_normal[1]
pear_diffuse = data_normal[2]
pear_original = data_normal[3]

plot_normals(sphere_diffuse, sphere_original, data_mask[0])
plot_normals(pear_diffuse, pear_original, data_mask[1])
```





1.4.2 3.2 Lambertian model [4 pts]

1.4.3 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 unit direction of the light source $\hat{\mathbf{s}}$ and the unit normal vector on the surface of the object $\hat{\mathbf{n}}$. The intensity of the light reflected from the surace with a single light source is found using the following relationship:

$$e(x, y) = a(x, y)\mathbf{C} \max(0, \hat{\mathbf{n}}^\top \mathbf{s})$$

$$\mathbf{s} = s_0 \hat{\mathbf{s}}$$

where, $a(x, y)$ is the albedo of the surface facet imaged by the pixel, $\mathbf{C}$ is the light source color, s_0 is the intensity of the light source, and $\hat{\mathbf{s}}$ is the unit direction to the light source from the surface facet imaged by the pixel.

Fill in your implementation for the rendered image using the lambertian model. You only need to support one light direction.

[8]:
```
# Normalizes the image to be between 0 and 1
def normalize(img):
```

```
    maxi = img.max()
    mini = img.min()
    return (img - mini)/(maxi-mini)
```

[9]:
```
def lambertian(normals, light, color, intensity, mask):
    """
    Implements the Lambertian model to compute the image given the normals,
    light directions, color, intensity, and mask.
    Inputs:
    - normals: H x W x 3 array of surface normals
    - light: 3-element array of light direction
    - color: 3-element array of RGB color of the light
    - intensity: intensity of the light
    - mask: H x W binary mask (1 for object, 0 for background)
    Outputs:
    - image: H x W x 3 array of the rendered image
    """
    #### Write your code here. ####

    # Normalize the light direction for unit vector
    light_dir = light / np.linalg.norm(light)

    # Assume every pixel has uniform albedo of 1
    height, width, _ = normals.shape
    albedo = np.ones((height, width))  # H x W

    # Compute dot product between normals and light direction
    # Reshape light to (1, 1, 3) so it can multiply with normals at each pixel
    light_vector = light_dir.reshape(1, 1, 3)
    dot_product = np.sum(normals * light_vector, axis=2)  # H x W

    # do the max of 0
    dot_product = np.clip(dot_product, 0, None)

    # Apply mask for background pixels
    dot_product *= mask

    # Compute final image by multiplying with light color and intensity
    # Expand dot_product to 3 channels to match RGB
    image = dot_product[:, :, np.newaxis] * color * intensity

    # Normalize image for display
    return normalize(image)
```

1.4.4 3.3 Blinn-Phong model [4 pts]

1.4.5 Blinn-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 Blinn-Phong reflectance model. This model combines ambient lighting, diffused reflectance as well as specular reflectance to find the brightness on a surface. Blinn-Phong shading also considers the material in the scene which is characterized by four values: the ambient value $k_a(x, y)$ of the surface facet imaged by the pixel, the diffuse value $k_d(x, y)$ of the surface facet imaged by the pixel, the specular value $k_s(x, y)$ of the surface facet imaged by the pixel, and the shininess $\alpha(x, y)$ of the surface facet imaged by the pixel. Furthermore, since the specular component produces 'rays', only some of which would be observed by a single observer, the observer's unit viewing direction $\hat{\mathbf{v}}(x, y)$ must also be known. For some scene with known material parameters with M light sources the intensity of the light $e(x, y)$ reflected from light direction $\hat{\mathbf{s}}(x, y)$ to the surface facet with unit normal vector $\hat{\mathbf{n}}(x, y)$ seen from unit viewing direction $\hat{\mathbf{v}}(x, y)$ can be computed by:

$$e(x, y) = \sum_{m \in M} s_a k_a(x, y) + s_m(k_d(x, y) f_d(x, y) + k_s(x, y) f_s(x, y)),$$

$$f_d(x, y) = \max(0, \hat{\mathbf{n}}(x, y)^\top \hat{\mathbf{s}}(x, y)),\ f_s(x, y) = \max(0, \hat{\mathbf{n}}(x, y)^\top \hat{\mathbf{h}}(x, y))^{\alpha(x,y)}$$

$$\hat{\mathbf{h}}(x, y) = \frac{1}{\|\mathbf{h}(x, y)\|} \mathbf{h}(x, y),\ \mathbf{h}(x, y) = \hat{\mathbf{s}}(x, y) + \hat{\mathbf{v}}(x, y)$$

where, for the m-th light source, s_m is the light color multiplied with light intensity. s_a is the ambient light color multiplied with ambient light intensity.

Please fill in your implementation for the Blinn-Phong model below. This time you need to support multiple light sources.

[10]:
```
def blinn_phong(normals, ambient, lights, colors, material, view, mask):
    """
    Implements the Blinn-Phong model to compute the image given the normals,␣
 ↪light directions,
    colors, material properties, view direction, and mask.
    Inputs:
    - normals: H x W x 3 array of surface normals
    - ambient: 3-element array of ambient light color
    - lights: N x 3 array of light directions
    - colors: N x 3 array of RGB colors of the lights
    - material: 4-element array of material properties (ka, kd, ks, a)
    - view: 3-element array of view direction
    - mask: H x W binary mask (1 for object, 0 for background)
    Outputs:
```

```
    - image: H x W x 3 array of the rendered image
    """
    #### Write your code here. ####

    # s^ -> unit light direction
    # v^ -> unit view direction
    # n^ -> unit normal direction
    # h^ -> half vector between s^ and v^
    # ka -> ambient light coefficient
    # kd -> diffuse light coefficient
    # ks -> specular light coefficient
    # a -> shininess coefficient

    # fix the shape of lights and colors
    lights = np.atleast_2d(lights)
    colors = np.atleast_2d(colors)

    # unpack
    H, W, _ = normals.shape
    ka, kd, ks, shininess = material

    # Normalize for s^ and v^
    normals = normals / np.linalg.norm(normals, axis=2, keepdims=True)
    view_dir = view / np.linalg.norm(view)

    # ambient term (base lighting)
    ambient = ambient * ka
    image = np.ones((H, W, 1)) * ambient

    # Recall lights are additive, so we loop through each light and add
    for light, color in zip(lights, colors):
        # Normalize light
        light_dir = light / np.linalg.norm(light)

        # Compute unit half-vector: h^ = s^ + v^
        half = light_dir + view_dir
        half /= np.linalg.norm(half)

        # Compute diffuse term: max(0, n · s)
        diff = np.sum(normals * light_dir.reshape(1,1,3), axis=2)
        diff = np.clip(diff, 0, None)

        # Compute specular term: max(0, n · h)^shininess
        spec = np.sum(normals * half.reshape(1,1,3), axis=2)
        spec = np.clip(spec, 0, None)
        spec = spec ** shininess
```

```
        # Add this light's contribution
        # Multiply by light color and kd/ks
        image += diff[..., None] * color * kd
        image += spec[..., None] * color * ks

    # Apply mask
    image *= mask[..., None]

    return normalize(image)
```

1.4.6 3.4 Render results [6 pts]

In this part, you need to plot the rendered results for Lambertian and Blinn-Phong model. The tables below shows the light directions and materials you'll use.

Table 1: Light Source Directions

Light	Direction	Color (RGB)
1	$(-0.5, 0.5, 0.5)^\top$	$(0.1, 0.4, 0.8)$
2	$(0.3, -0.2, 0.5)^\top$	$(0.7, 0.3, 0.3)$

Table 2: Material Coefficients

Mat	k_a	k_d	k_s	α
1	0.1	0.2	0.6	3
2	0.1	0.6	0.2	3
3	0.1	0.6	0.6	10

For **Lambertian model**, assume light intensity is 1. you're required to render 4 images for sphere (You don't care about materials here.): - Light Source 1, Diffuse Sphere - Light Source 2, Diffuse Sphere - Light Source 1, Original Sphere - Light Source 2, Original Sphere

For **Blinn-Phong model**, Assume light intensity is 1, view direction is `[0,0,1]` and ambient light color is `[1, 1, 1]`. you need to render 9 images each for the diffuse pear and the original pear, corresponding to `[light1, light2, light1+light2]` and the `[material1, material2, material3]`. Therefore, in total, you will need $2 \times 3 \times 3 = 18$ images.

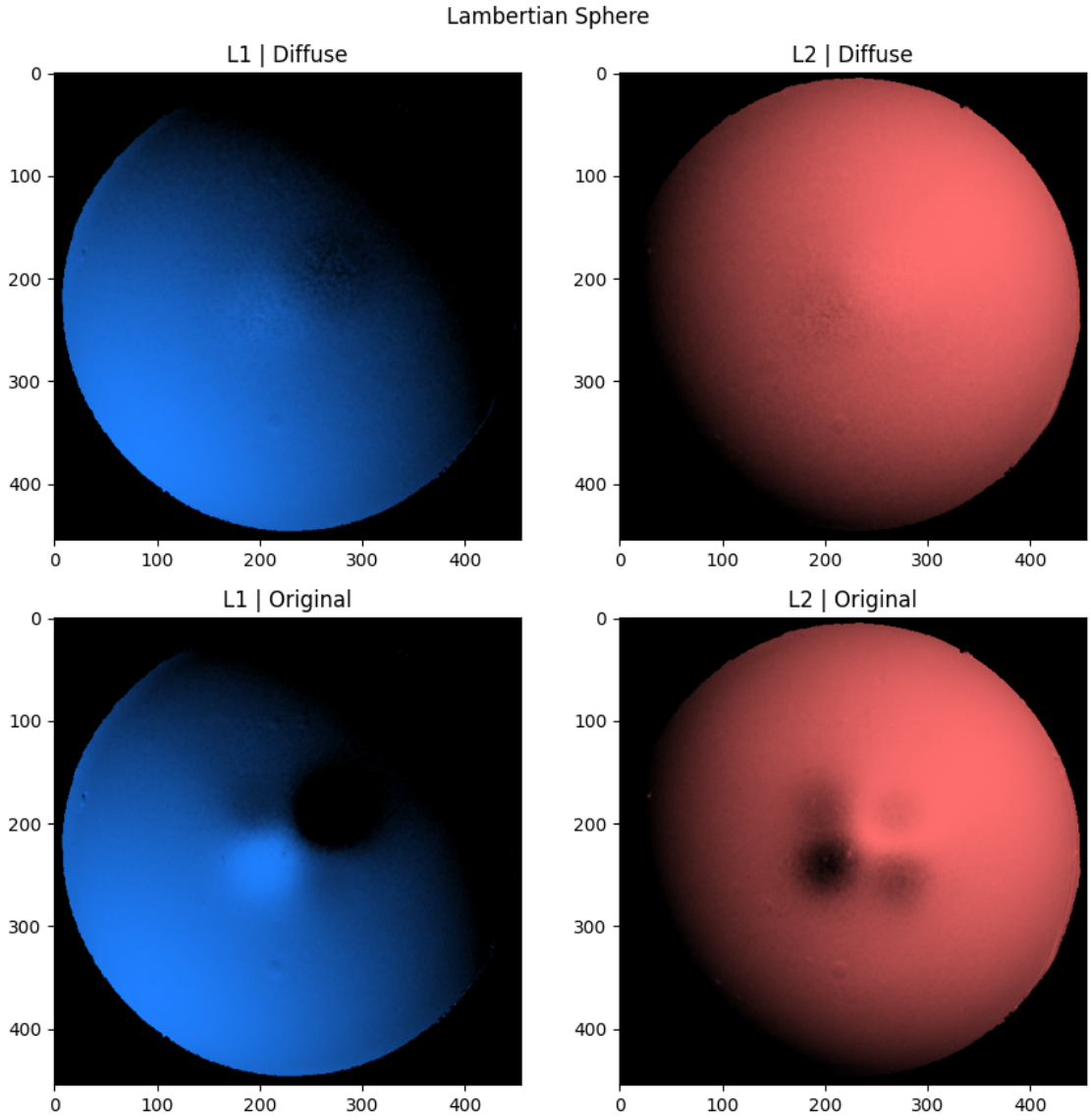
[11]:
```
# Helper method to create subplot
def add_sub_plot(figure, loc, text, img):
    sub1 = figure.add_subplot(loc)
    sub1.title.set_text(text)
    sub1.imshow(img)
```

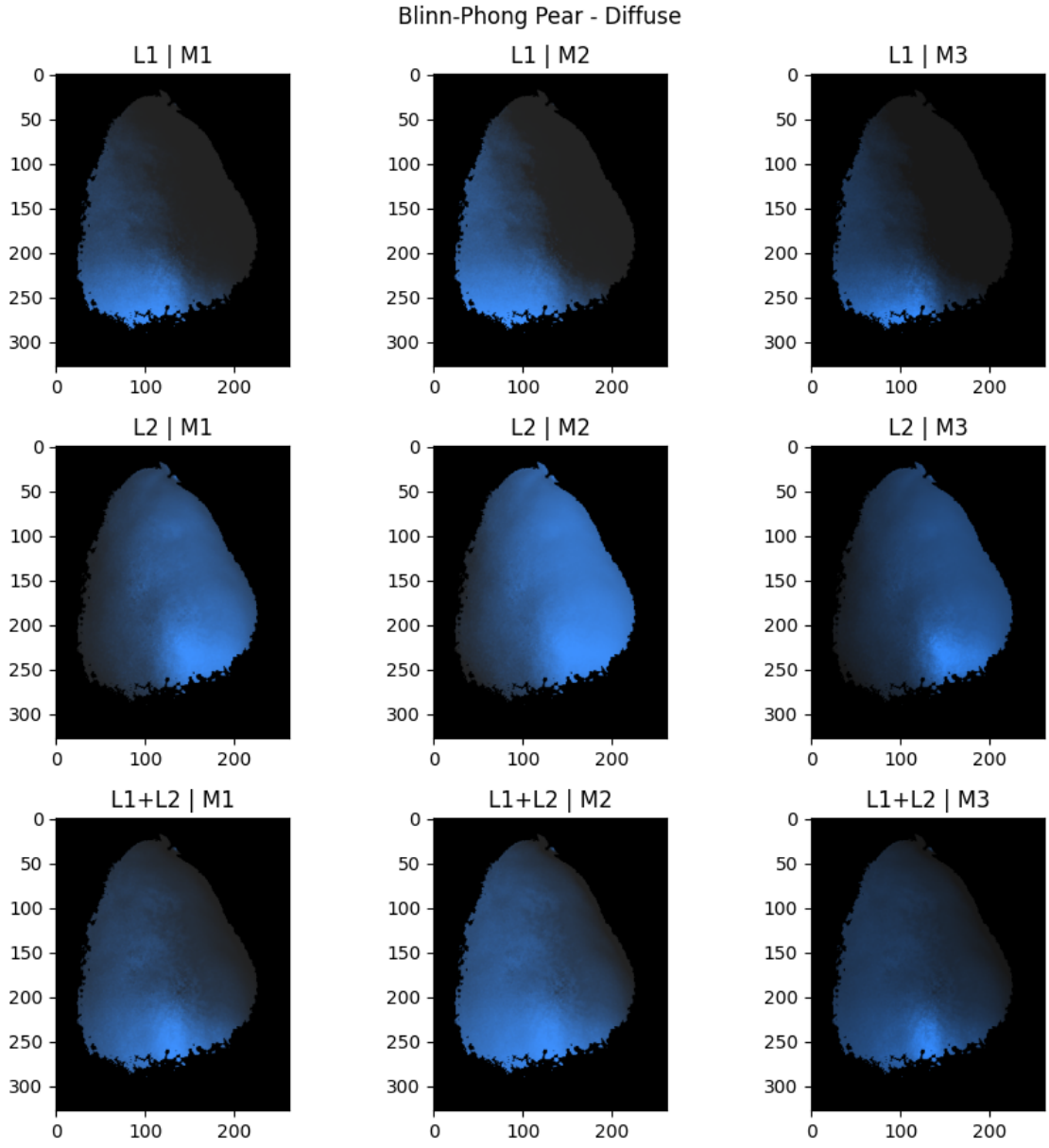
[12]:
```
##################################################
# Don't modify this function
##################################################
def plot_lambertian(results, title):
```

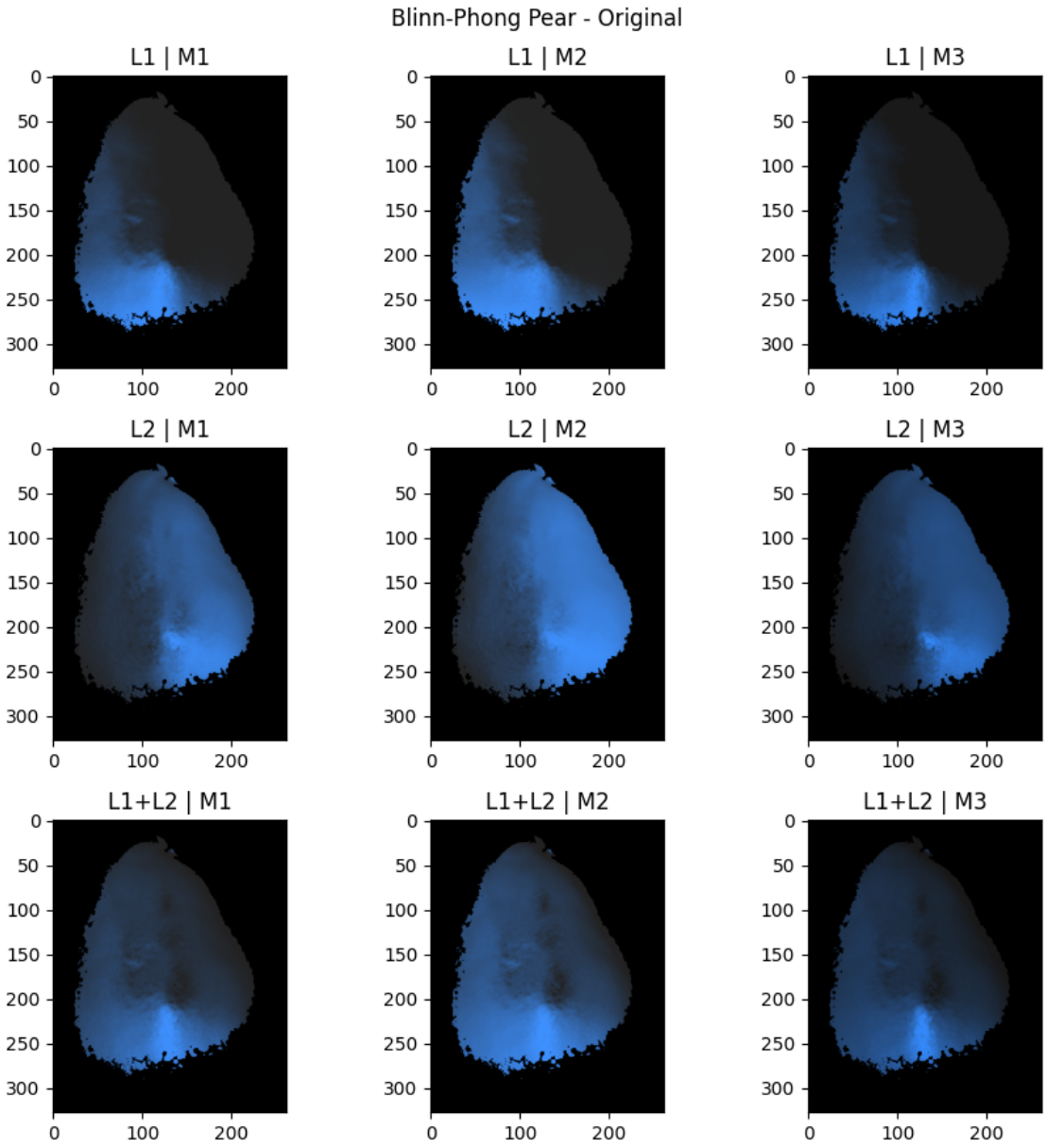
```
    figure = plt.figure(figsize=(9, 9))
    figure.suptitle(title)
    add_sub_plot(figure, 221, 'L1 | Diffuse', results[0])
    add_sub_plot(figure, 222, 'L2 | Diffuse', results[1])
    add_sub_plot(figure, 223, 'L1 | Original', results[2])
    add_sub_plot(figure, 224, 'L2 | Original', results[3])
    figure.tight_layout()
    plt.show()

##################################################
# Don't modify this function
##################################################
def plot_blinnphong(results, title):
    figure = plt.figure(figsize=(9, 9))
    figure.suptitle(title)
    add_sub_plot(figure, 331, 'L1 | M1', results[0])
    add_sub_plot(figure, 332, 'L1 | M2', results[1])
    add_sub_plot(figure, 333, 'L1 | M3', results[2])
    add_sub_plot(figure, 334, 'L2 | M1', results[3])
    add_sub_plot(figure, 335, 'L2 | M2', results[4])
    add_sub_plot(figure, 336, 'L2 | M3', results[5])
    add_sub_plot(figure, 337, 'L1+L2 | M1', results[6])
    add_sub_plot(figure, 338, 'L1+L2 | M2', results[7])
    add_sub_plot(figure, 339, 'L1+L2 | M3', results[8])
    figure.tight_layout()
    plt.show()
```

[13]:
```
#### Write your code here. ####
light1 = np.array([-0.5, 0.5, 0.5])
light2 = np.array([0.3, -0.2, 0.5])

color1 = np.array([0.1, 0.4, 0.8])
color2 = np.array([0.7, 0.3, 0.3])

material1 = np.array([0.1, 0.2, 0.6, 3])
material2 = np.array([0.1, 0.6, 0.2, 3])
material3 = np.array([0.1, 0.6, 0.6, 10])

# Lambertian: def lambertian(normals, light, color, intensity, mask):
results_sphere = []

# light1, diffuse
results_sphere.append(lambertian(sphere_diffuse, light1, color1, 1,␣
 ↪data_mask[0]))
# light2, diffuse
results_sphere.append(lambertian(sphere_diffuse, light2, color2, 1,␣
 ↪data_mask[0]))
```

```
# light1, original
results_sphere.append(lambertian(sphere_original, light1, color1, 1,␣
 ↪data_mask[0]))
# light2, original
results_sphere.append(lambertian(sphere_original, light2, color2, 1,␣
 ↪data_mask[0]))




# Blinn-phong: def blinn_phong(normals, ambient, lights, colors, material,␣
 ↪view, mask):
intesity = 1
view = np.array([0, 0, 1])
ambient = np.array([1, 1, 1])

lights = np.array([light1, light2, light1+light2])
materials = np.array([material1, material2, material3])

results_pear_diffuse = []
results_pear_original = []

for light in lights:
    for material in materials:
        results_pear_diffuse.append(blinn_phong(pear_diffuse, ambient, light,␣
 ↪color1, material, view, data_mask[1]))
        results_pear_original.append(blinn_phong(pear_original, ambient, light,␣
 ↪color1, material, view, data_mask[1]))

plot_lambertian(results_sphere, "Lambertian Sphere")
plot_blinnphong(results_pear_diffuse, "Blinn-Phong Pear - Diffuse")
plot_blinnphong(results_pear_original, "Blinn-Phong Pear - Original")
```


Lambertian Sphere
L1 | Diffuse
L2 | Diffuse
L1 | Original
L2 | Original
0
100
200
300
400


Blinn-Phong Pear - Diffuse
L1 | M1
L1 | M2
L1 | M3
L2 | M1
L2 | M2
L2 | M3
L1+L2 | M1
L1+L2 | M2
L1+L2 | M3
0
50
100
150
200
250
300
0
100
200



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 4.1: 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 10000 (ten 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_data2.pickle`. Try using only `im1`, `im3` and `im4` first. Display your outputs as mentioned below.

Then use all four images (most accurate).

Note: DO NOT normalize the images prior to use in the photemetric stereo algorithm. the images must be used **as-is**.

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.

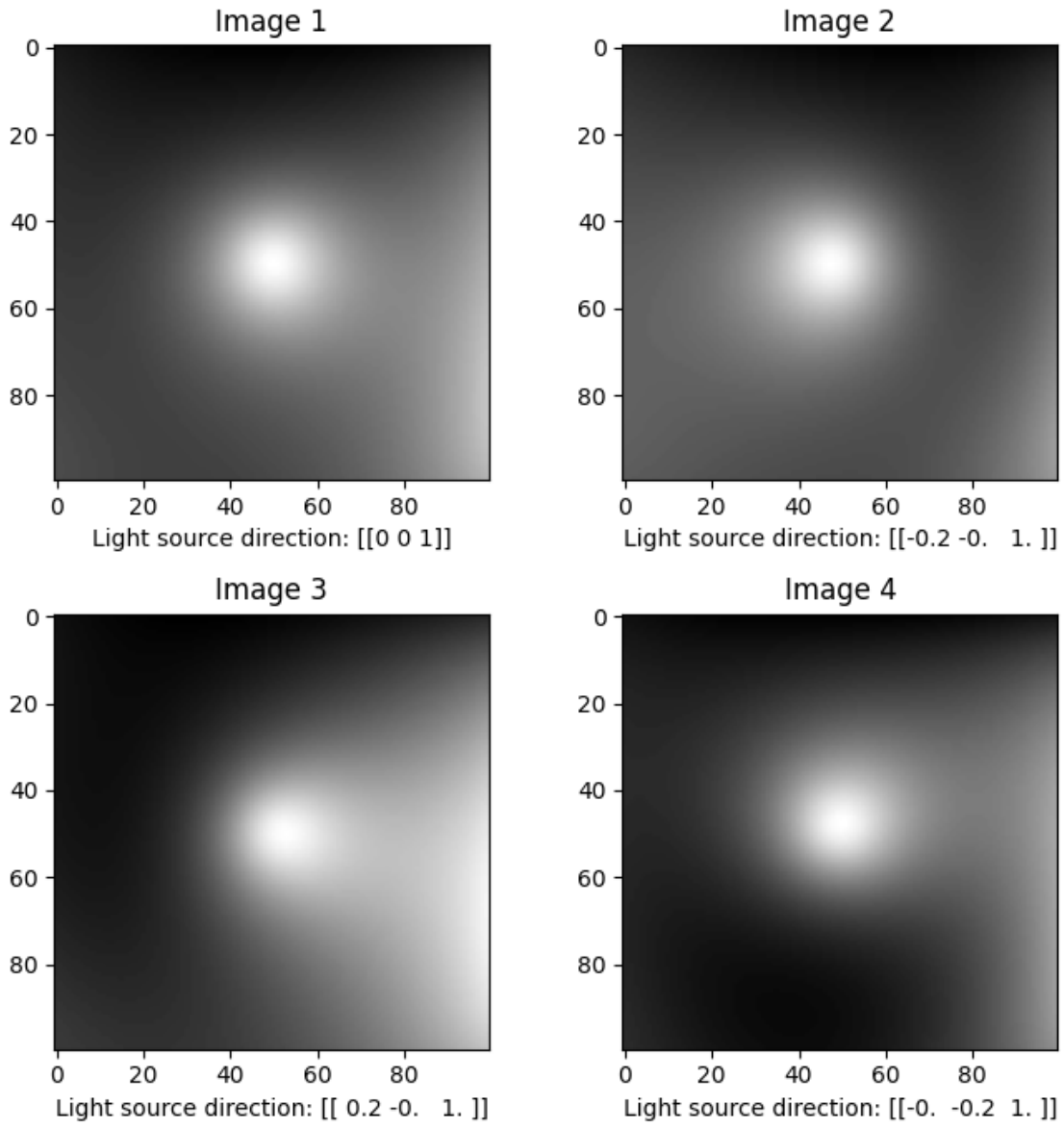
In total, we expect 2 * 7 = 14 images for this part.

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

[14]:
```
# 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_data2.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.
fig = plt.figure(figsize=(7,7))
for i in range(1, 5):
    sub = fig.add_subplot(int('22'+str(i)))
    sub.set_title('Image ' + str(i))
    sub.set_xlabel("Light source direction: " + str(data["l%d" % i]))
    sub.imshow(data["im%d" % i], cmap="gray")
plt.tight_layout()
plt.show()
```

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



Based on the above images, can you interpret the orientation of the coordinate frame? We assume the X-axis **points to the right**, the Y-axis **points down**, and the Z-axis points out of the screen in our direction.

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

[15]:
```
import numpy as np
from scipy.signal import convolve
from numpy import linalg

############################################################
# Do not modify
```

```
#########################################################
def horn_integrate(p, q, mask, niter):
    """
    horn_integrate recovers the function g from its partial
    derivatives p and q.
    p:      the partial derivatives along the horizontal direction, positive␣
 ↪axis to the right
    q:      the partial derivatives along the vertical direction, positive axis␣
 ↪down
    mask:   a binary image which tells which pixels are involved in integration.
    niter:  the number of iterations. typically 10,000 or 100,000,
            although the trend can be seen even after 1000 iterations.
    """
    g = np.ones(np.shape(p))

    p = np.multiply(p, mask)
    q = np.multiply(q, mask)

    # A = np.array([[0,0,0],[0,0,0],[0,1,0]]) #y-1
    # B = np.array([[0,0,0],[0,0,1],[0,0,0]]) #x-1
    # C = np.array([[0,0,0],[1,0,0],[0,0,0]]) #x+1
    # D = np.array([[0,1,0],[0,0,0],[0,0,0]]) #y+1
    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, p) + np.multiply(m_d, q)
    t_a = -1.0 * convolve(p, B, mode="same")
    t_b = -1.0 * convolve(q, 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
```

```
    g -= np.min(g)
    g *= mask
    return g
```

[ ]:
```
'''Function to compute the albedo, normals,
    and height map using the photometric stereo
    method & horn integration'''
def photometric_stereo(images, lights, mask, horn_niter=25000):
    """
    inputs:

    images : (n_ims, h, w) photometric images. Input images should not be␣
 ↪normalized to [0, 1]
             range, use as-is.
    lights : (n_ims, 3) light source directions
    mask   : (h, w) 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 Part 1 (it shouldn't matter, just pass␣
 ↪a mask of all ones),
    but you SHOULD use it to filter out the background for Part 3.

    outputs:

    albedo:   (h, w)  The estimated albedo map.
    normals:  (h, w, 3) The estimated surface normals
    Z:        (h, w)  The depth map given by the scanline method.
    H_horn:   (h, w)  The depth map given by Horn integration.
    """

    #### Write your code here. ####

    n_ims, H, W = images.shape

    H_horn = np.ones(images[0].shape)
    normals = np.ones((H,W,3))
    albedo = np.ones((H,W))
    Z = np.zeros((H,W))
```

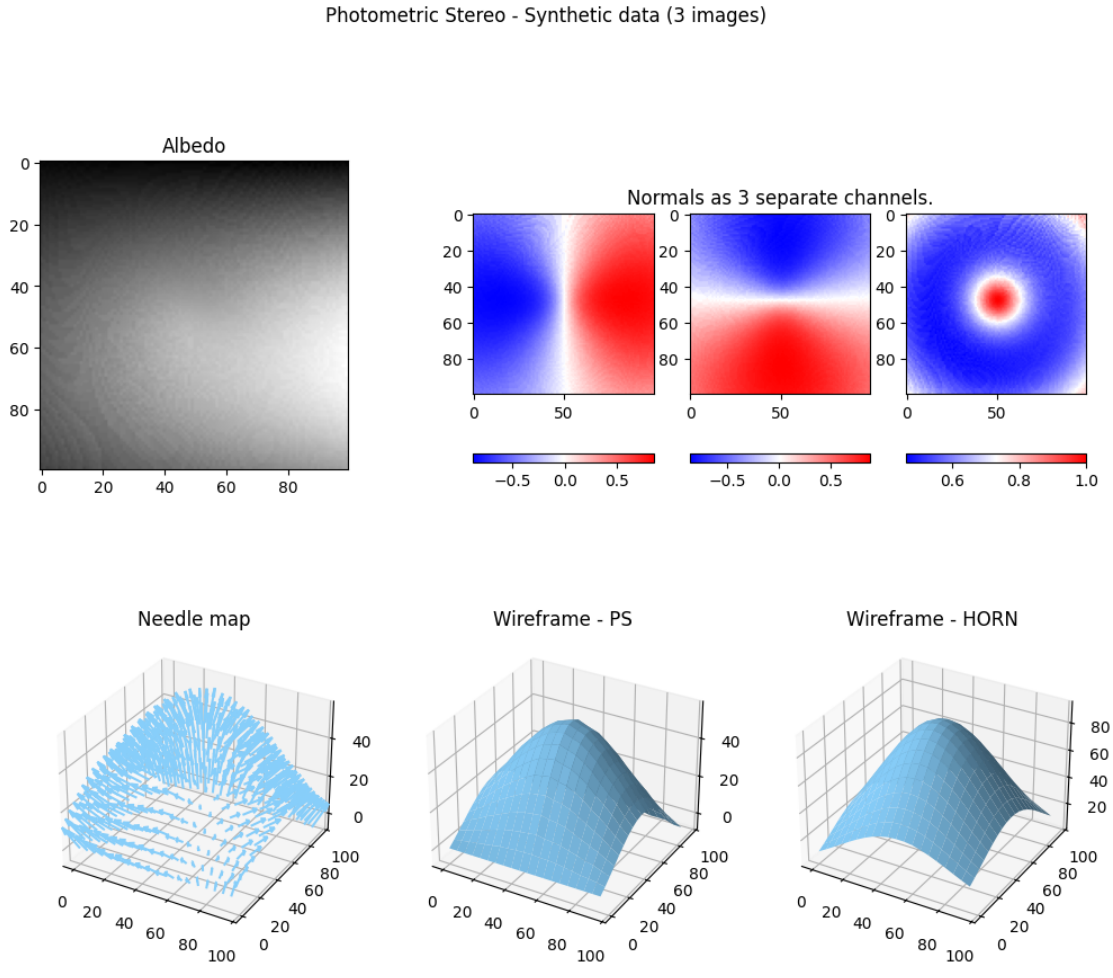
```
    # MUST use 'shapy by integration' method.

    # step 1:construct lambertian reflectance map for each light source.
    # 1) acquire 3 images with known light source directions -> this is given to use as images
    # 2) for each pixel, find (p,q) as the intersection of the three curves

    # linalg.solve doesn't work cuz there are more n_images than lights...
    # have to use least square solution to solve, and pinv solves that
    L_pinv = np.linalg.pinv(lights)
    for x in range(H):
        for y in range(W):
            if mask[x, y] == 0:
                continue
            # get the 3 images for the current pixel
            e = images[:,x,y]
            # solve for g = albedo * normals
            g = np.dot(L_pinv,e)
            # albedo is magnitude of g
            albedo[x,y] = np.linalg.norm(g)
            # normals are components of g
            normals[x,y,:] = g / albedo[x,y]


    # now we have to recover the surface depth map
    # from this normals estimate, we can get p and q, and integrate to get z(x,y)
    # from estimate n = (nx,ny,nz), p = -nx/nz, q = -ny/nz
    p = -normals[...,0] / normals[...,2]
    q = -normals[...,1] / normals[...,2]
    # then integrate p=df/dx along row (x,0) to get f(x,0)

    for x in range(1,H):
        Z[x,0] = Z[x-1,0] + p[x,0]

    # then integrate q=df/dy along each column starting with value of first row.

    for x in range(1, H):
        for y in range(W):
            Z[x, y] = Z[x-1, y] + q[x-1, y]
    # the equation should be z(x,y) = z(x,0) + integral(pdx+qdy) from x=0 to x=x and y=0 to y=y

    # apply mask:
    Z*=mask
    albedo*=mask
```

```
    normals *= mask[..., None]

    H_horn = horn_integrate(p, q, mask, horn_niter)
    return albedo, normals, Z, H_horn
```

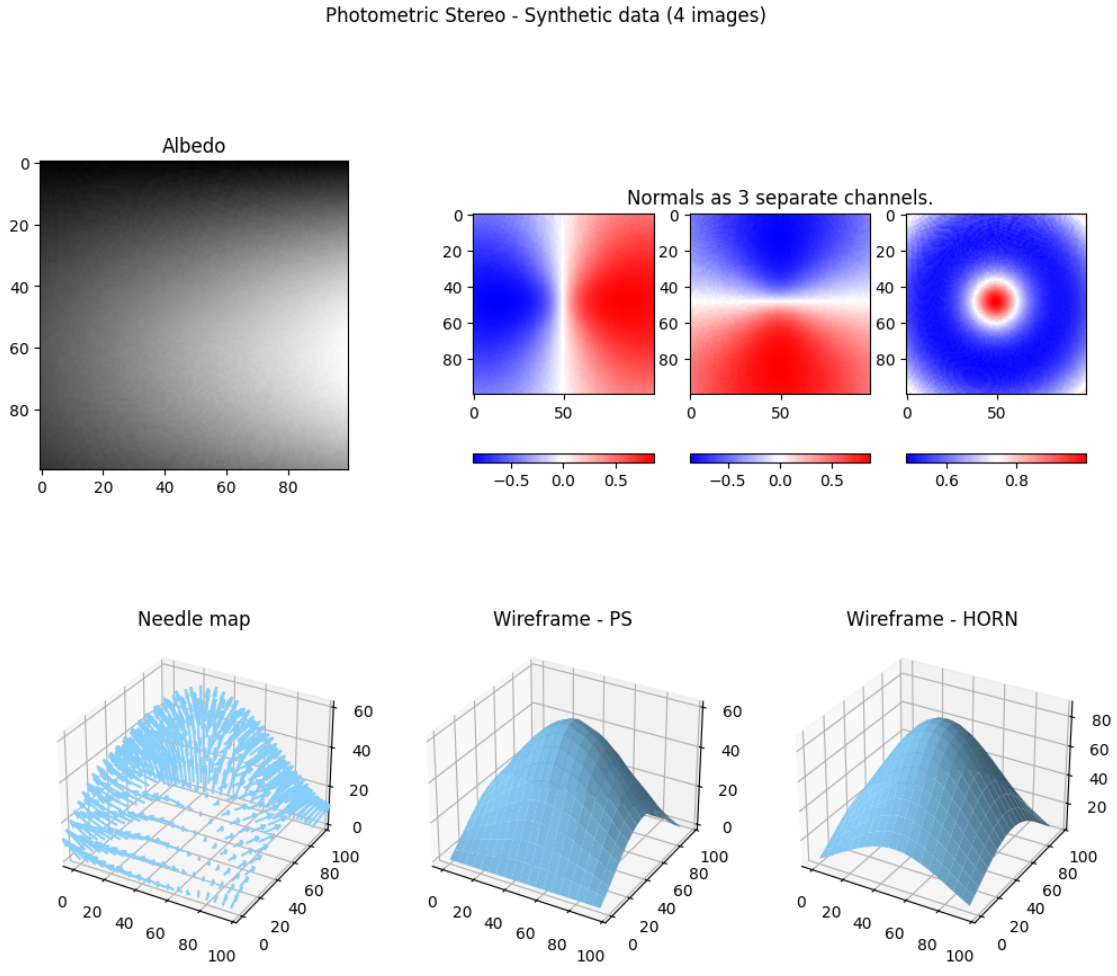
[ ]:
```
# ------------------------------------------------------------------------
# The following code is just a working example so you don't get stuck with any
# of the graphs required. You may want to write your own code to align the
# results in a better layout. You are also free to change the function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

def visualize(albedo, normals, depth, horn, imtitle='', stride = 15):
    # showing albedo map
    fig = plt.figure(figsize=(12,10))
    albedo_max = np.max(albedo)
    if albedo_max > 1e-8:
        albedo = albedo / albedo_max
    else:
        print("  Warning: albedo is zero everywhere.")
    albedo = albedo / albedo_max
    ax0 = fig.add_subplot(231)
    ax0.set_title("Albedo")
    ax0.imshow(albedo, cmap="gray")

    # showing normals as three separate channels
    ax1 = fig.add_subplot(253)
    fig.colorbar(ax1.imshow(normals[..., 0],cmap='bwr'), ax=ax1,␣
 ↪orientation='horizontal')
    ax2 = fig.add_subplot(254)
    ax2.set_title("Normals as 3 separate channels.")
    fig.colorbar(ax2.imshow(normals[..., 1],cmap='bwr'), ax=ax2,␣
 ↪orientation='horizontal')
    ax3 = fig.add_subplot(255)
    fig.colorbar(ax3.imshow(normals[..., 2],cmap='bwr'), ax=ax3,␣
 ↪orientation='horizontal')

    # showing normals as quiver
    X, Y, _ = np.meshgrid(np.arange(0,np.shape(normals)[1], stride),
                          np.arange(0,np.shape(normals)[0], stride),
                          np.arange(1))
    X = X[..., 0]
    Y = Y[..., 0]
    Z = depth[::stride,::stride]
    NX = normals[..., 0][::stride,::stride]
    NY = normals[..., 1][::stride,::stride]
    NZ = normals[..., 2][::stride,::stride]
```

```
    ax4 = fig.add_subplot(234, projection='3d')
    ax4.set_title("Needle map")
    ax4.quiver(X,Y,Z,NX,NY,NZ, length=15, color='lightskyblue')

    # plotting wireframe depth map
    H = depth[::stride,::stride]
    ax5 = fig.add_subplot(235, projection='3d')
    ax5.set_title("Wireframe - PS")
    ax5.plot_surface(X,Y, H, color='lightskyblue')

    H = horn[::stride,::stride]
    ax6 = fig.add_subplot(236,projection='3d')
    ax6.set_title("Wireframe - HORN")
    ax6.plot_surface(X,Y, H, color='lightskyblue')
    fig.suptitle(imtitle)
    plt.show()
```

[18]:
```
## In total, we expect 2 * 7 = 14 images for this part.
## Set-1: USING ONLY im1, im3, im4
#### Write your code here. ####

pickle_in = open("synthetic_data2.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")

lights = np.vstack((data["l1"], data["l3"], data["l4"]))
images = np.array([data["im1"], data["im3"], data["im4"]])
mask = np.ones(data["im1"].shape)


albedo, normals, depth, horn = photometric_stereo(images, lights, mask, 10000)
visualize(albedo, normals, depth, horn, "Photometric Stereo - Synthetic data (3␣
 ↪images)", 5)
```



[19]:
```
## Set-2: USING im1, im2, im3, im4

#### Write your code here. ####

lights = np.vstack((data["l1"], data["l2"], data["l3"], data["l4"]))
images = np.array([data["im1"], data["im2"], data["im3"], data["im4"]])
mask = np.ones(data["im1"].shape)

# Run photometric stereo
albedo, normals, depth, horn = photometric_stereo(images, lights, mask, 10000)

# Visualize results
visualize(albedo, normals, depth, horn, "Photometric Stereo - Synthetic data (4␣
 ↪images)", 5)
```



1.5.3 4.2: 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.

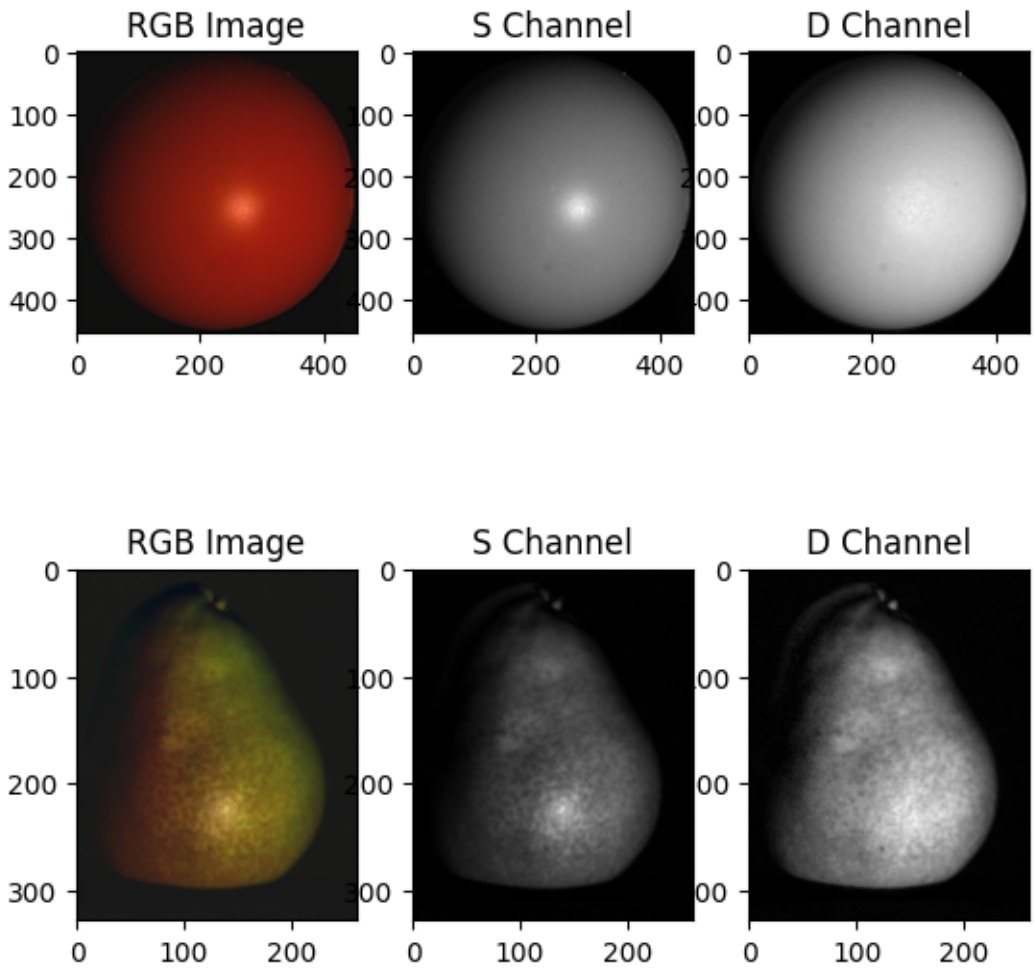
Note: You will find all the data for this part in `specular_sphere2.pickle` and `specular_pear2.pickle`.

[20]:
```
##########################################################
# Do not modify
##########################################################
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
```

[21]:
```
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.
    '''
    #### Write your code here. ####

    S = np.ones(I_rgb.shape[:2])
    D = np.ones(I_rgb.shape[:2])
```

```
    rot_mat = get_rot_mat(rot_vec)

    for i in range(I_rgb.shape[0]):
        for j in range(I_rgb.shape[1]):
            # I_suv = rot_mat * I_rgb
            # S = ||I_suv||
            # D = sqrt(U**2 + V**2)
            I_suv = np.matmul(rot_mat, I_rgb[i,j])
            S[i,j] = np.linalg.norm(I_suv)
            D[i,j] = np.sqrt(I_suv[1]**2 + I_suv[2]**2)


    return S, D
```

[25]:
```
# plot the original image(RGB), the recovered S channel and
# the diffuse part of the image for sphere and pear

def plot_RGB_SUV(filename, imtitle):
    pickle_in = open(filename, "rb")
    data = pickle.load(pickle_in, encoding="latin1")
    #### Write your code here. ####

    # get SUV from image
    S, D = RGBToSUV(data["im1"], np.hstack((data["c"][0][0],
                                            data["c"][1][0],
                                            data["c"][2][0])))

    # plot RGB image
    plt.subplot(1,3,1)
    plt.imshow(normalize(data["im1"]))
    plt.title("RGB Image")

    # plot S channel
    plt.subplot(1,3,2)
    plt.imshow(S, cmap="gray")
    plt.title("S Channel")

    # plot D channel
    plt.subplot(1,3,3)
    plt.imshow(D, cmap="gray")
    plt.title("D Channel")

    plt.show()

plot_RGB_SUV("specular_sphere2.pickle", 'SPECULAR SPHERE')
plot_RGB_SUV("specular_pear2.pickle", 'SPECULAR PEAR')
```



1.5.4 4.3: 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 **Part 1** on the original images (converted to grayscale using the equation in Assignment 0). 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 **Part 1**.

Next, combine parts 1 and 2 by removing the specularities (using your code from **Part 2**) 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.

Note: While creating the masks, please use 0.15 as your threshold for the sphere and 0.1 for the pear. **DO NOT** "normalize" or ortherwise modify the images prior to use in the photometric stereo algorithm. The images must be used **as-is**.

For the specular sphere image set in `specular_sphere2.pickle` and specular pear images set in `specular_pear2.pickle`, using all of four images in each, include:

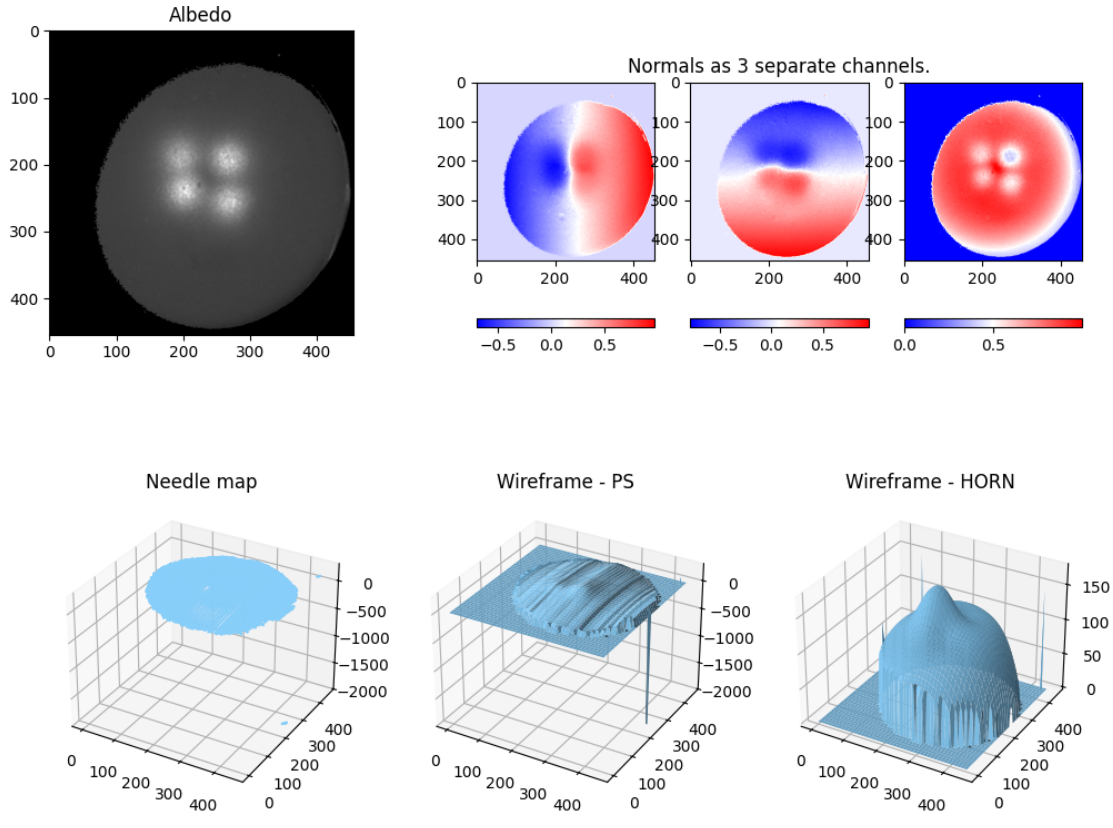
1. The estimated albedo map (original and diffuse).
2. The estimated surface normals (original and diffuse) by showing both
    1. Needle map, and
    2. Three images showing each of the surface normal components.
3. A wireframe of depth map (original and diffuse).
4. A wireframe of the depth map given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

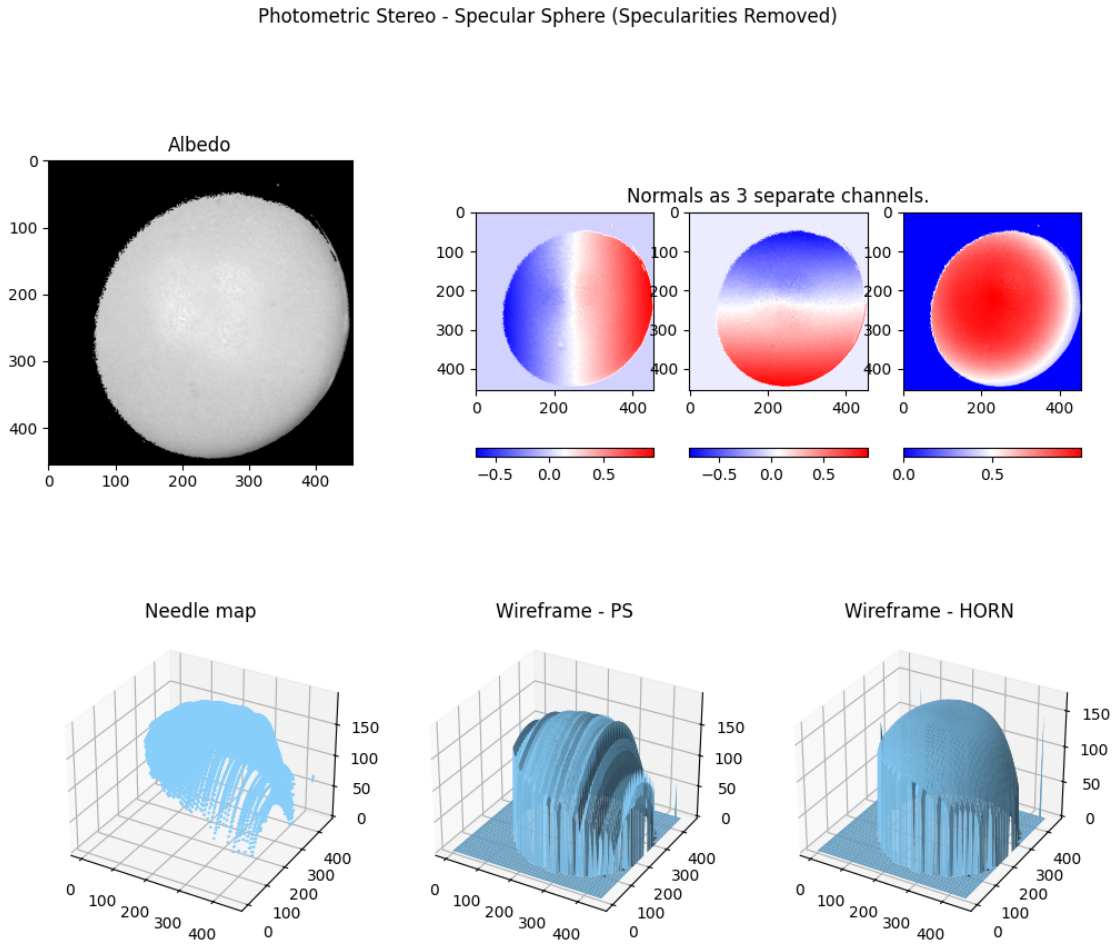
In total, we expect 2 * 7 = 14 images for the **Part 1** comparison, plus 2 * 7 = 14 images for the outputs after specularity removal has been performed. (Thus 28 output images overall.)

[ ]:
```
# ------------------------------------------------------------------------
# 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.
# ------------------------------------------------------------------------

import copy

pickle_in = open("specular_sphere2.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")

# Prepare lights and original mask
lights = np.vstack((data["l1"], data["l2"], data["l3"], data["l4"]))
mask = np.ones(data["im1"].shape[:2])

# Specular Sphere (grayscale)
images = []
for i in range(1, 5):
    img = np.array(data["im"+str(i)])
    gray = np.dot(img[...,:3], [0.21263903, 0.71516871, 0.072192319])
    images.append(gray)
images = np.stack(images, axis=0)

# filter mask using 0.15 as threshold

gray_ref = normalize(images[0])
for i in range(gray_ref.shape[0]):
    for j in range(gray_ref.shape[1]):
        if gray_ref[i, j] <= 0.15:
            mask[i, j] = 0

# Run photometric stereo on original images
albedo, normals, depth, horn = photometric_stereo(images, lights, mask, 10000)
visualize(albedo, normals, depth, horn, "Photometric Stereo - Specular Sphere",␣
↪5)
```

```
# Sphere with specularities removed.

diffuse_images = []

for i in range(1, 5):
    rgb_img = np.array(data["im"+str(i)])
    S, D = RGBToSUV(rgb_img, np.hstack((data["c"][0][0], data["c"][1][0],␣
 ↪data["c"][2][0])))
    diffuse_images.append(D)
diffuse_images = np.stack(diffuse_images, axis=0)

albedo, normals, depth, horn = photometric_stereo(diffuse_images, lights, mask,␣
 ↪10000)
visualize(albedo, normals, depth, horn, "Photometric Stereo - Diffuse Sphere␣
 ↪(Specularities Removed)", 5)
```

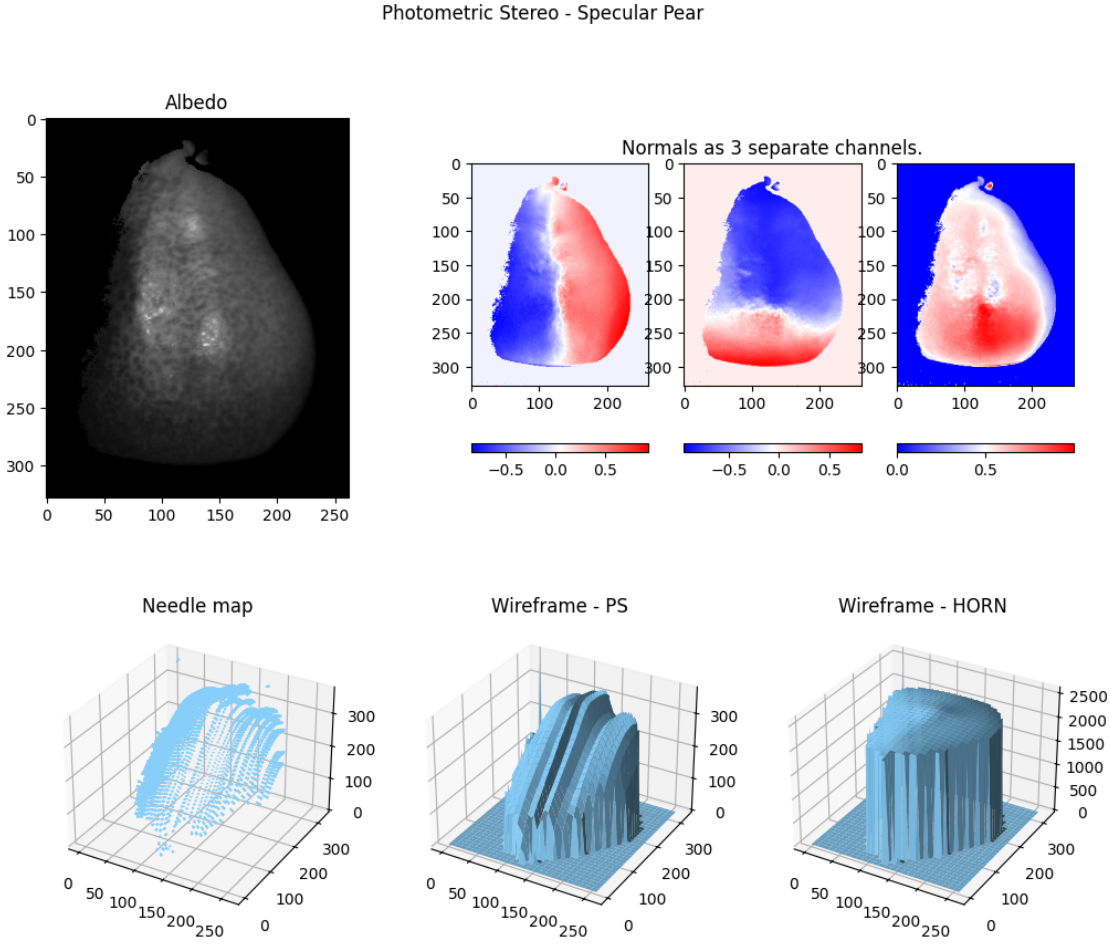
Photometric Stereo - Specular Sphere

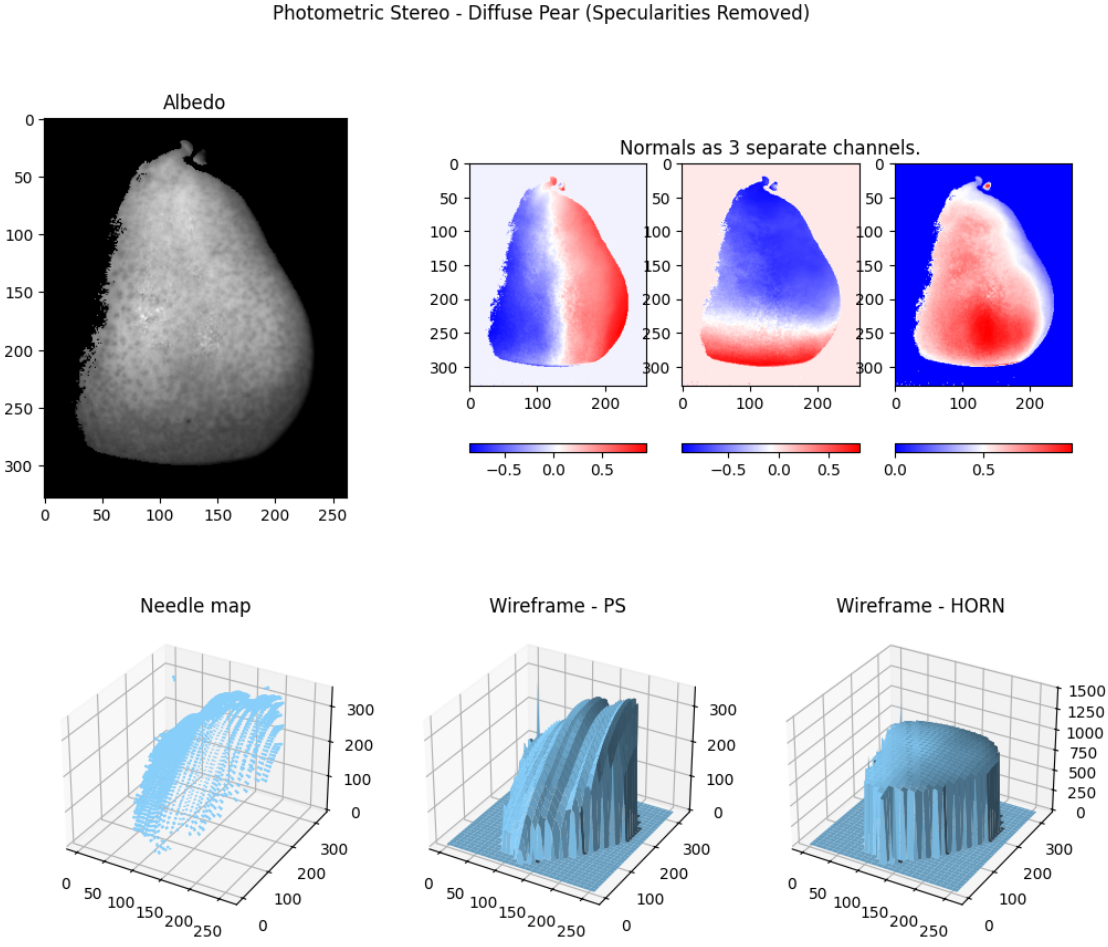




[35]:
```
pickle_in = open("specular_pear2.pickle", "rb")
data = pickle.load(pickle_in, encoding="latin1")
#### Write your code here. ####

# Prepare lights and original mask
lights = np.vstack((data["l1"], data["l2"], data["l3"], data["l4"]))
mask = np.ones(data["im1"].shape[:2])

# Specular Pear (grayscale)
images = []
for i in range(1, 5):
    img = np.array(data["im"+str(i)])
    gray = np.dot(img[...,:3], [0.21263903, 0.71516871, 0.072192319])
    images.append(gray)
images = np.stack(images, axis=0)

# filter mask using 0.1 as threshold
```

```
gray_ref = normalize(images[0])
for i in range(gray_ref.shape[0]):
    for j in range(gray_ref.shape[1]):
        if gray_ref[i, j] <= 0.095:
            mask[i, j] = 0

# Run photometric stereo on original images
albedo, normals, depth, horn = photometric_stereo(images, lights, mask, 10000)
visualize(albedo, normals, depth, horn, "Photometric Stereo - Specular Pear", 5)

# Pear with specularities removed.

diffuse_images = []

for i in range(1, 5):
    rgb_img = np.array(data["im"+str(i)])
    S, D = RGBToSUV(rgb_img, np.hstack((data["c"][0][0], data["c"][1][0],␣
 ↪data["c"][2][0])))
    diffuse_images.append(D)
diffuse_images = np.stack(diffuse_images, axis=0)

albedo, normals, depth, horn = photometric_stereo(diffuse_images, lights, mask,␣
 ↪10000)
visualize(albedo, normals, depth, horn, "Photometric Stereo - Diffuse Pear␣
 ↪(Specularities Removed)", 5)
```


Photometric Stereo - Specular Pear
Albedo
Normals as 3 separate channels.
Needle map
Wireframe - PS
Wireframe - HORN


Photometric Stereo - Diffuse Pear (Specularities Removed)
Albedo
Normals as 3 separate channels.
Needle map
Wireframe - PS
Wireframe - HORN