16-720 HW6: Photometric Stereo

Pablo Agustin Ortega Kral (portegak)

Utils and Imports

Importing all necessary libraries.

In [1]:
```
import numpy as np
from matplotlib import pyplot as plt
from skimage.color import rgb2xyz
import warnings
from scipy.ndimage import gaussian_filter
from matplotlib import cm
from skimage.io import imread
from scipy.sparse import kron as spkron
from scipy.sparse import eye as speye
from scipy.sparse.linalg import lsqr as splsqr
import os
import shutil
```

Downloading the data

In [2]:
```
if os.path.exists('data'):
  shutil.rmtree('data')

os.mkdir('data')
!wget 'https://docs.google.com/uc?export=download&id=13nA1Haq6bJz0-h_7NmovvSRrRD76qiF0' -O data/data.zip
!unzip "data/data.zip"
os.system("rm data/data.zip")
data_dir = 'data/'
```

```
--2025-04-24 21:25:41--  https://docs.google.com/uc?export=download&id=13nA1Haq6bJz0-h_7NmovvSRrRD76qiF0
Resolving docs.google.com (docs.google.com)... 172.253.62.138, 172.253.62.100, 172.253.62.101, ...
Connecting to docs.google.com (docs.google.com)|172.253.62.138|:443... connected.
HTTP request sent, awaiting response... 303 See Other
Location: https://drive.usercontent.google.com/download?id=13nA1Haq6bJz0-h_7NmovvSRrRD76qiF0&export=downloa
d [following]
--2025-04-24 21:25:41--  https://drive.usercontent.google.com/download?id=13nA1Haq6bJz0-h_7NmovvSRrRD76qiF0
&export=download
Resolving drive.usercontent.google.com (drive.usercontent.google.com)... 142.251.16.132, 2607:f8b0:4004:c1
7::84
Connecting to drive.usercontent.google.com (drive.usercontent.google.com)|142.251.16.132|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 6210854 (5.9M) [application/octet-stream]
Saving to: ‘data/data.zip’

data/data.zip       100%[===================>]   5.92M  --.-KB/s    in 0.09s

2025-04-24 21:25:45 (62.6 MB/s) - ‘data/data.zip’ saved [6210854/6210854]

Archive:  data/data.zip
  inflating: data/sources.npy
  inflating: data/input_5.tif
  inflating: data/input_7.tif
  inflating: data/input_6.tif
  inflating: data/input_4.tif
  inflating: data/input_1.tif
  inflating: data/input_2.tif
  inflating: data/input_3.tif
```

Utils Functions.

In [3]:
```
def integrateFrankot(zx, zy, pad = 512):

    """
    Question 1 (j)
```

```
    Implement the Frankot-Chellappa algorithm for enforcing integrability
    and normal integration

    Parameters
    ----------
    zx : numpy.ndarray
        The image of derivatives of the depth along the x image dimension

    zy : tuple
        The image of derivatives of the depth along the y image dimension

    pad : float
        The size of the full FFT used for the reconstruction

    Returns
    ----------
    z: numpy.ndarray
        The image, of the same size as the derivatives, of estimated depths
        at each point

    """

    # Raise error if the shapes of the gradients don't match
    if not zx.shape == zy.shape:
        raise ValueError('Sizes of both gradients must match!')

    # Pad the array FFT with a size we specify
    h, w = 512, 512

    # Fourier transform of gradients for projection
    Zx = np.fft.fftshift(np.fft.fft2(zx, (h, w)))
    Zy = np.fft.fftshift(np.fft.fft2(zy, (h, w)))
    j = 1j

    # Frequency grid
    [wx, wy] = np.meshgrid(np.linspace(-np.pi, np.pi, w),
                           np.linspace(-np.pi, np.pi, h))
    absFreq = wx**2 + wy**2

    # Perform the actual projection
    with warnings.catch_warnings():
        warnings.simplefilter('ignore')
        z = (-j*wx*Zx-j*wy*Zy)/absFreq

    # Set (undefined) mean value of the surface depth to 0
    z[0, 0] = 0.
    z = np.fft.ifftshift(z)

    # Invert the Fourier transform for the depth
    z = np.real(np.fft.ifft2(z))
    z = z[:zx.shape[0], :zx.shape[1]]

    return z


def enforceIntegrability(N, s, sig = 3):

    """
    Question 2 (e)

    Find a transform Q that makes the normals integrable and transform them
    by it

    Parameters
    ----------
    N : numpy.ndarray
        The 3 x P matrix of (possibly) non-integrable normals

    s : tuple
        Image shape

    Returns
    -------
    Nt : numpy.ndarray
        The 3 x P matrix of transformed, integrable normals
```

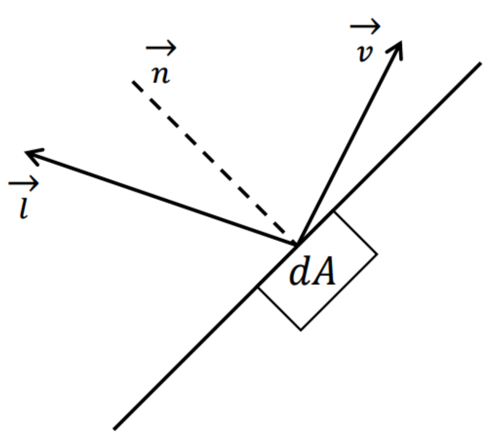
```
    """

    N1 = N[0, :].reshape(s)
    N2 = N[1, :].reshape(s)
    N3 = N[2, :].reshape(s)

    N1y, N1x = np.gradient(gaussian_filter(N1, sig), edge_order = 2)
    N2y, N2x = np.gradient(gaussian_filter(N2, sig), edge_order = 2)
    N3y, N3x = np.gradient(gaussian_filter(N3, sig), edge_order = 2)

    A1 = N1*N2x-N2*N1x
    A2 = N1*N3x-N3*N1x
    A3 = N2*N3x-N3*N2x
    A4 = N2*N1y-N1*N2y
    A5 = N3*N1y-N1*N3y
    A6 = N3*N2y-N2*N3y

    A = np.hstack((A1.reshape(-1, 1),
                   A2.reshape(-1, 1),
                   A3.reshape(-1, 1),
                   A4.reshape(-1, 1),
                   A5.reshape(-1, 1),
                   A6.reshape(-1, 1)))

    AtA = A.T.dot(A)
    W, V = np.linalg.eig(AtA)
    h = V[:, np.argmin(np.abs(W))]

    delta = np.asarray([[-h[2],  h[5], 1],
                        [ h[1], -h[4], 0],
                        [-h[0],  h[3], 0]])
    Nt = np.linalg.inv(delta).dot(N)

    return Nt

def plotSurface(surface, suffix=''):

    """
    Plot the depth map as a surface

    Parameters
    ----------
    surface : numpy.ndarray
        The depth map to be plotted

    suffix: str
        suffix for save file

    Returns
    -------
        None

    """
    x, y = np.meshgrid(np.arange(surface.shape[1]),
                       np.arange(surface.shape[0]))
    fig = plt.figure()
    #ax = fig.gca(projection='3d')
    ax = fig.add_subplot(111, projection='3d')
    surf = ax.plot_surface(x, y, -surface, cmap = cm.coolwarm,
                           linewidth = 0, antialiased = False)
    ax.view_init(elev = 60., azim = 75.)
    plt.savefig(f'figs/faceCalibrated{suffix}.png')
    plt.show()

def plotSurfaceGrid(surface, suffix='', ax = None):

    """
    Plot the depth map as a surface

    Parameters
    ----------
    surface : numpy.ndarray
        The depth map to be plotted

    suffix: str
```

```
        suffix for save file

    Returns
    -------
        None

    """
    render = ax is None
    x, y = np.meshgrid(np.arange(surface.shape[1]),
                       np.arange(surface.shape[0]))
    if render:
        fig = plt.figure()
        #ax = fig.gca(projection='3d')
        ax = fig.add_subplot(111, projection='3d')
    surf = ax.plot_surface(x, y, -surface, cmap = cm.coolwarm,
                           linewidth = 0, antialiased = False)

    ax.view_init(elev = 60., azim = 75.)
    ax.set_title(suffix)
    if render:
        plt.savefig(f'figs/faceCalibrated{suffix}.png')
        plt.show()

def loadData(path = "../data/"):

    """
    Question 1 (c)

    Load data from the path given. The images are stored as input_n.tif
    for n = {1...7}. The source lighting directions are stored in
    sources.mat.

    Paramters
    ---------
    path: str
        Path of the data directory

    Returns
    -------
    I : numpy.ndarray
        The 7 x P matrix of vectorized images

    L : numpy.ndarray
        The 3 x 7 matrix of lighting directions

    s: tuple
        Image shape

    """
    I = None
    L = None
    s = None

    L = np.load(path + 'sources.npy').T

    im = imread(path + 'input_1.tif')
    P = im[:, :, 0].size
    s = im[:, :, 0].shape

    I = np.zeros((7, P))
    for i in range(1, 8):
        im = imread(path + 'input_' + str(i) + '.tif')
        im = rgb2xyz(im)[:, :, 1]
        I[i-1, :] = im.reshape(-1,)

    return I, L, s

def displayAlbedosNormals(albedos, normals, s):

    """
    Question 1 (e)

    From the estimated pseudonormals, display the albedo and normal maps

    Please make sure to use the `coolwarm` colormap for the albedo image
```

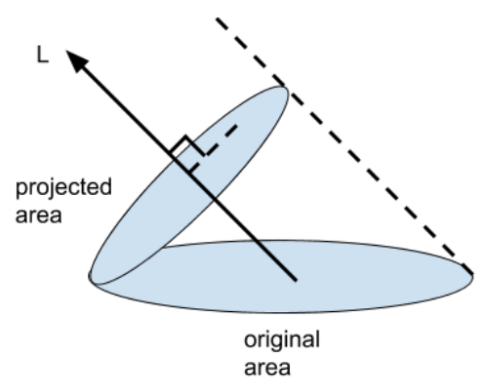
```
    and the `rainbow` colormap for the normals.

    Parameters
    ----------
    albedos : numpy.ndarray
        The vector of albedos

    normals : numpy.ndarray
        The 3 x P matrix of normals

    s : tuple
        Image shape

    Returns
    -------
    albedoIm : numpy.ndarray
        Albedo image of shape s

    normalIm : numpy.ndarray
        Normals reshaped as an s x 3 image

    """
    albedoIm = None
    normalIm = None

    albedoIm = albedos.reshape(s)
    normalIm = (normals.T.reshape((s[0], s[1], 3))+1)/2

    plt.figure()
    plt.imshow(albedoIm, cmap = 'gray')

    plt.figure()
    plt.imshow(normalIm, cmap = 'rainbow')

    plt.show()

    return albedoIm, normalIm
```

Q1: Calibrated photometric stereo (75 points)

Q 1 (a): Understanding n-dot-l lighting (5 points)



(a) Geometry of photometric stereo



(b) Projected area

Fig 2.a depicts the interaction of vectors respect to the surface of an object. Vector $\vec{l}$ represents direction of the lighting source of the scene; $\vec{n}$ is the normal of the surface and $\vec{v}$ is the camera view vector. The area iluminated depends on the angle between the light source vector and the surface normal.

The dot product between the vectors arises as it precisely describes the cosine of the angle of the two vectors, scaled by their magnitude, allowing to compactly use them to model the radiance. In particular if $\vec{l}$ and $\vec{n}$ are unit vectors then $\vec{n} \cdot \vec{l} = \cos \theta_{\vec{l}\vec{n}}$.

The cosine of the angle, and thus the dot product $\vec{n} \cdot \vec{l}$, geometrically represents the scaling factor between the original surface dA and the surface projected in the view of the light vector $\hat{dA}$.

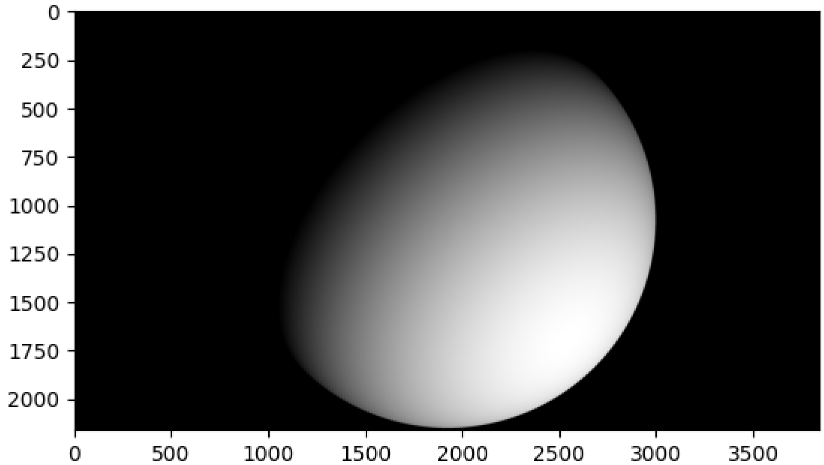
$$\hat{dA} = dA\vec{n} \cdot \vec{l}$$

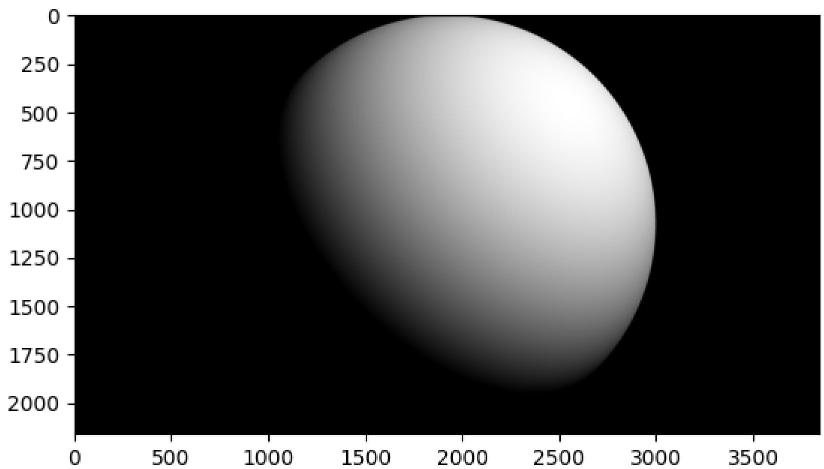
Notice that the radiance does not dependend on where we view the object from, rather it only depends of the relationship between the object's surface and the light source. As regardless of the viewpoint, the scattered light will be the same.

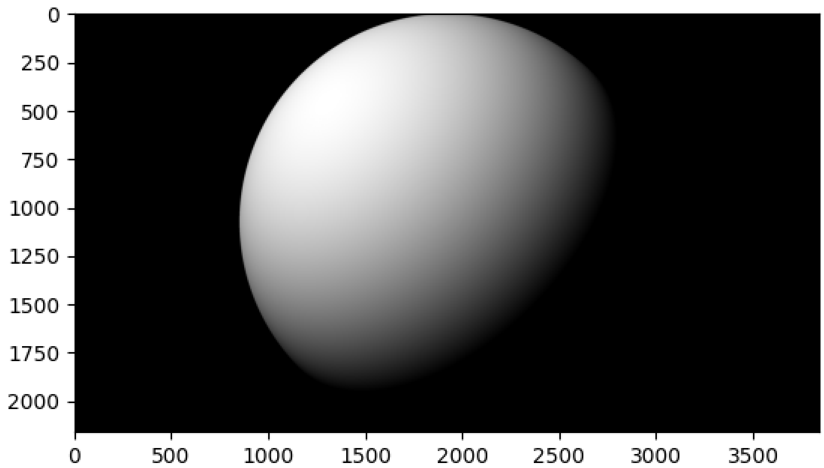
Q 1 (b): Rendering the n-dot-l lighting (10 points)

In [ ]:
```
def renderNDotLSphere(center, rad, light, pxSize, res):

    """
    Question 1 (b)

    Render a hemispherical bowl with a given center and radius. Assume that
    the hollow end of the bowl faces in the positive z direction, and the
    camera looks towards the hollow end in the negative z direction. The
    camera's sensor axes are aligned with the x- and y-axes.

    Parameters
    ----------
    center : numpy.ndarray
        The center of the hemispherical bowl in an array of size (3,)

    rad : float
        The radius of the bowl

    light : numpy.ndarray
        The direction of incoming light

    pxSize : float
        Pixel size

    res : numpy.ndarray
        The resolution of the camera frame

    Returns
    -------
    image : numpy.ndarray
        The rendered image of the hemispherical bowl
    """

    [X, Y] = np.meshgrid(np.arange(res[0]), np.arange(res[1]))
    X = (X - res[0]/2) * pxSize*1.e-4
    Y = (Y - res[1]/2) * pxSize*1.e-4
    Z = np.sqrt(rad**2+0j-X**2-Y**2)
    X[np.real(Z) == 0] = 0
    Y[np.real(Z) == 0] = 0
    Z = np.real(Z)

    image = None
    # Get light vector
    light_vec = light/np.linalg.norm(light)
    # Get surface normals
    cx , cy, cz = center
    X = (X - cx)/rad
    Y = (Y - cy)/rad
    Z = (Z - cz)/rad
    normals = np.array([X, Y, Z])
    normals /= np.linalg.norm(normals)
    # Get dot product
    n_dot_l = np.dot(normals.T, light_vec)
    n_dot_l[n_dot_l < 0] = 0
    # Build the image
    image = n_dot_l.astype(np.float32)
    image = image.T
    return image

# Part 1(b)
radius = 0.75 # cm
center = np.asarray([0, 0, 0]) # cm
```

```
pxSize = 7 # um
res = (3840, 2160)

light = np.asarray([1, 1, 1])/np.sqrt(3)
image = renderNDotLSphere(center, radius, light, pxSize, res)
plt.figure()
plt.imshow(image, cmap = 'gray')
plt.imsave('figs/1b-a.png', image, cmap = 'gray')

light = np.asarray([1, -1, 1])/np.sqrt(3)
image = renderNDotLSphere(center, radius, light, pxSize, res)
plt.figure()
plt.imshow(image, cmap = 'gray')
plt.imsave('figs/1b-b.png', image, cmap = 'gray')

light = np.asarray([-1, -1, 1])/np.sqrt(3)
image = renderNDotLSphere(center, radius, light, pxSize, res)
plt.figure()
plt.imshow(image, cmap = 'gray')
plt.imsave('figs/1b-c.png', image, cmap = 'gray')

I, L, s = loadData(data_dir)
```







Q 1 (c): Initials (10 points)

In [5]:
```
#NOTE:  I = L.T B
U, S ,Vh = np.linalg.svd(I, full_matrices=False)
print("Singular values: ", S)
```

```
Singular values:  [79.36348099 13.16260675  9.22148403  2.414729    1.61659626  1.26289066
  0.89368302]
```

1. Why should I be rank 3?

Theoritically given that the lighting direction have dimensions $L \in \mathbb{R}^{3\times N}$ and the psuedo normal matrix has one column per point making it shape $B \in \mathbb{R}^{3\times P}$, then there are a maximum of three linearly independent components forming I meaning we should expect a rank of 3.

2. Do the singular values agree with this?

A: We can see that the all 7 singular values are different than 0, meaning that it is not purely speaking rank 3. However we can see that the first 3 values are significantly larger than the rest; while the remaining singular values are closer to 0. This occurs due to inherent noise in real world photographs; as well as departures from the Lambertain assumptions made during modelling.

Q 1 (d) Estimating pseudonormals (20 points)

In [6]:
```
def estimatePseudonormalsCalibrated(I, L):

    """
    Question 1 (d)

    In calibrated photometric stereo, estimate pseudonormals from the
    light direction and image matrices

    Parameters
    ----------
    I : numpy.ndarray
        The 7 x P array of vectorized images

    L : numpy.ndarray
        The 3 x 7 array of lighting directions

    Returns
    -------
    B : numpy.ndarray
        The 3 x P matrix of pesudonormals
    """


    A = L.T
    y = I.copy()
```

```
    B, _, _, _ = np.linalg.lstsq(A, y, rcond=None)

    return B

# Part 1(e)
B = estimatePseudonormalsCalibrated(I, L)
print("Pseudonormals shape: ", B.shape)
```

```
Pseudonormals shape:  (3, 159039)
```

A

Given that $I = L^T B$ we have a linear system $Ax = y$ that we can solve directly, where $A = L^T$ and $y = I$. Note that in this simple implementation one could run into memory issues because of the size of the matrixes. In the case of my system, I was able to load everything without a problem.

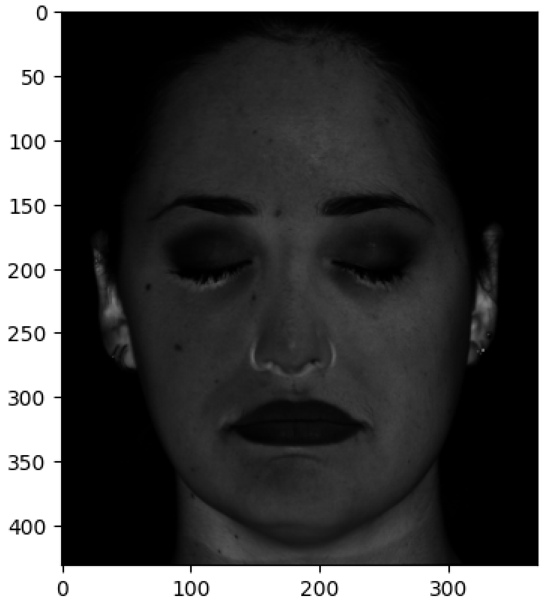
Q 1 (e) Albedos and normals (10 points)

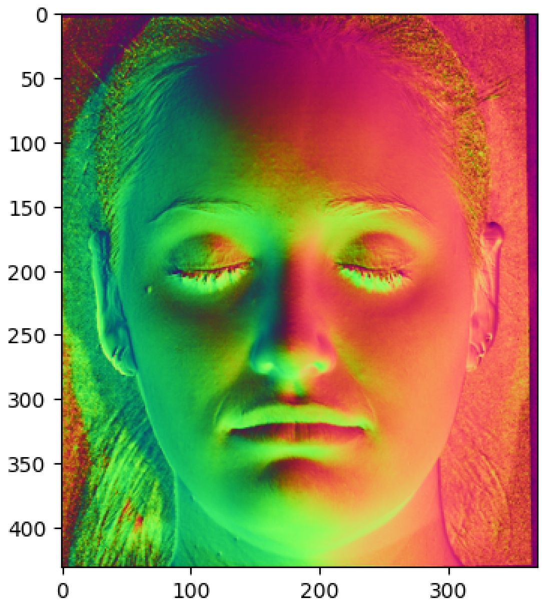
A:

Surprisingly, higher albedo values are observed in areas where we typically would see shadows caused by the nose, ears and neck. Given that we are estimating the albedo from the pseudonormals by taking their scale as the the albedo value ($B = an \therefore a = ||B|| \text{and} b = \frac{B}{||B||}$); this means regions with high magnitude of the pseudonorm, have high albedo.

When solving for the psuedo, there are multiple factors that could cause the norm to be large. Given that we are using a simple Lambertian model, this might produce incorrect estimation of shadows while omitting other considerations that impact realworld lighting, leading to an overestimation of the pseudonormal to fit the perceived intensities in shadows and noisy regions.

In [7]:
```
def estimateAlbedosNormals(B):

    '''
    Question 1 (e)

    From the estimated pseudonormals, estimate the albedos and normals

    Parameters
    ----------
    B : numpy.ndarray
        The 3 x P matrix of estimated pseudonormals

    Returns
    -------
    albedos : numpy.ndarray
        The vector of albedos

    normals : numpy.ndarray
        The 3 x P matrix of normals
    '''

    albedos = None
    normals = None

    # Albedos are the scale of the pseudonormals
    albedos = np.linalg.norm(B, axis=0)
    # Normals are the normalized pseudonormals
    normals = B/albedos
    return albedos, normals


# Part 1(e)
albedos, normals = estimateAlbedosNormals(B)
albedoIm, normalIm = displayAlbedosNormals(albedos, normals, s)
plt.imsave('figs/1f-a.png', albedoIm, cmap = 'gray')
plt.imsave('figs/1f-b.png', normalIm, cmap = 'rainbow')
```





Q 1 (f): Normals and depth (5 points)

A:

The face is defined as a surface $S = \begin{bmatrix} x \\ y \\ f(x,y) \end{bmatrix}$. We wish to establish a relationship between the surface normal and the partial derivatives of the $f(x)$. We know that the normals will always be perpendicular to the tangent planes of the surface; the partial derivatives describe vectors on these tangent plane. We can exploit this relationship to express the normals in terms of the partials.

1. The tangent vectors in the x and y are obtained by taking the corresponding partials,

$$S_x = \begin{bmatrix} 1 \\ 0 \\ f_x \end{bmatrix}$$

$$S_y = \begin{bmatrix} 0 \\ 1 \\ f_y \end{bmatrix}$$

2. To obtain a vector in the direction of the normal, that is perpendicular to both S_x and S_y we take their cross product.

$$S_x \times S_y = \begin{bmatrix} (0)(f_y) - f_x(1) \\ f_x(0) - (1)(f_y) \\ (1)(1) - (0)(0) \end{bmatrix}$$

$$S_x \times S_y = \begin{bmatrix} -f_x \\ -f_y \\ 1 \end{bmatrix}$$

3. We know that n is proportional to $S_x \times S_y$, to ger rid of the notion of scale we can normalize $S_x \times S_y$ such that $n = \mathrm{norm}(S_x \times S_y)$

$$n = \frac{(-f_x - f_y + 1)}{\sqrt{f_x^2 + f_y^2 + 1}}$$

$$n_1 = \frac{(-f_x)}{\sqrt{f_x^2 + f_y^2 + 1}}$$

$$n_2 = \frac{(-f_y)}{\sqrt{f_x^2 + f_y^2 + 1}}$$

$$n_3 = \frac{(1)}{\sqrt{f_x^2 + f_y^2 + 1}}$$

4. We can use n_3 to write the partials in terms of the components of the normal

$$\frac{n_1}{n_3} = \frac{(-f_x)}{\sqrt{f_x^2 + f_y^2 + 1}} \frac{\sqrt{f_x^2 + f_y^2 + 1}}{1} = -f_x$$

$$\frac{n_2}{n_3} = \frac{(-f_y)}{\sqrt{f_x^2 + f_y^2 + 1}} \frac{\sqrt{f_x^2 + f_y^2 + 1}}{1} = -f_y$$

5. Therfore,

$$f_x = -\frac{n_1}{n_3}$$

$$f_y = -\frac{n_2}{n_3}$$

Q 1 (g): Understanding integrability of gradients (5 points)

Given $g = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$.

0. Calculate the $g_x(x_i, y_j) = g(x_i + 1, y_j) - g(x_i, y_j)$ and $g_y(x_i, y_j) = g(x_i, y_j + 1) - g(x_i, y_j)$

$$g_x = \begin{bmatrix} 2-1 & 3-2 & 4-3 \\ 6-5 & 7-6 & 8-7 \\ 10-9 & 11-10 & 12-11 \\ 14-13 & 15-14 & 16-15 \end{bmatrix}$$

$$g_x = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$g_y = \begin{bmatrix} 5-1 & 6-2 & 7-3 & 8-4 \\ 9-5 & 10-6 & 11-7 & 12-8 \\ 13-9 & 14-10 & 15-11 & 16-12 \end{bmatrix}$$

$$g_y = \begin{bmatrix} 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \end{bmatrix}$$

1. Use g_x to construct the first row of g, then use g_y to construct the rest of g

For the first row, and using $g(0,0) = g_x(0,0)$ we obtain

$$\hat{g} = \begin{bmatrix} 1 & 1+1=2 & 2+1=3 & 3+1=4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

For the rest,

$$\hat{g} = \begin{bmatrix} 1 & 1+1=2 & 2+1=3 & 3+1=4 \\ 1+4=5 & 2+4=6 & 3+4=7 & 4+4=8 \\ 5+4=9 & 6+4=10 & 7+4=11 & 8+4-12 \\ 9+4=13 & 10+4=14 & 11+4=15 & 12+4=16 \end{bmatrix}$$

We see that .

2. Use g_y to construct the first column of g, then use g_x to construct the rest of g.

$$\hat{g} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1+4=5 & 0 & 0 & 0 \\ 5+4=9 & 0 & 0 & 0 \\ 9+4=13 & 0 & 0 & 0 \end{bmatrix}$$

Now applying g_x iteratively,

$$\hat{g} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1+4=5 & 6 & 7 & 8 \\ 5+4=9 & 10 & 11 & 12 \\ 9+4=13 & 14 & 15 & 16 \end{bmatrix}$$

In this case, $\hat{g} = g$. Confirming that g_x and g_y are integrable.

- What could make gradients non-integrable?

In general, gradients are integrable if they are derived from a function that follows Clairut's Theorem, that is that **the second order mixed partial derivatives are equal**

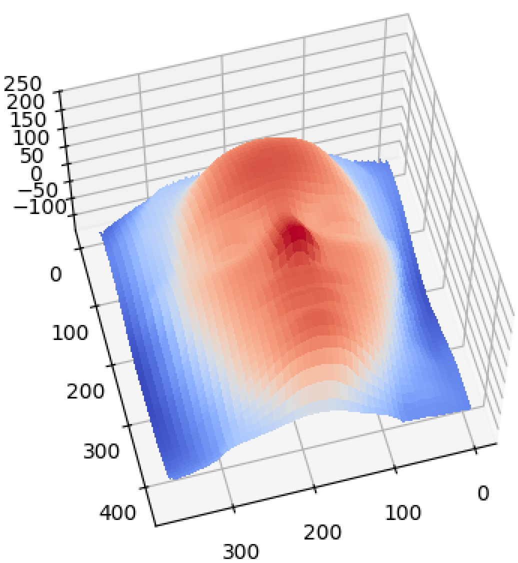
$$\frac{\partial^2 f}{\partial x} = \frac{\partial^2 f}{\partial y}$$

This allows us to disregard the order in which we integrate, which is we start with the row or column yielded the same result when reconstructing g. We see that indeed for the provided example g if we again diferentiate g_x and g_y these are equal.

To make it non-integrable, we could asume some noise in the derivation process that results in noisy gradientes, such that their seconf order mixed derivates are no longer similar.

Q 1 (h): Shape estimation (10 points)

In [8]:
```
def estimateShape(normals, s):

    """
    Question 1 (h)

    Integrate the estimated normals to get an estimate of the depth map
    of the surface.

    Parameters
    ----------
    normals : numpy.ndarray
        The 3 x P matrix of normals

    s : tuple
        Image shape

    Returns
    ----------
    surface: numpy.ndarray
        The image, of size s, of estimated depths at each point

    """

    surface = None
    n1, n2, n3 = normals[0, :], normals[1, :], normals[2, :]

    # Estimate depth partials
    depth_dx = -n1/n3
    depth_dy = -n2/n3
    depth_dx = depth_dx.reshape(s)
    depth_dy = depth_dy.reshape(s)

    # Use Frankot-Chellappa to get surface
    surface = integrateFrankot(depth_dx, depth_dy)

    return surface


# Part 1(h)
surface = estimateShape(normals, s)
plotSurface(surface)
```



Q2: Uncalibrated photometric stereo (50 points)

Q 2 (a): Uncalibrated normal estimation (10 points)

Given that in the uncalibrated scenario we do not have acess to the ligh sources, we now have to factorize I to estimate the light sources $\hat{L}$ and the psuedo normals $\hat{B}$.

We can leverage single value decomposition to factor I as

$$I = U\Sigma V^T$$

To get the best rank 3 aproximation of I, we simply pick the top 3 singular values from Σ, giving us that $\Sigma_3 = \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix}$.

To enable us to express I as two factors, we can write $\Sigma_3 = \Sigma^{1/2}\Sigma^{1/2}$. Thus,

$$I = U\Sigma^{1/2}\Sigma^{1/2}V^T$$

By grouping terms we have,

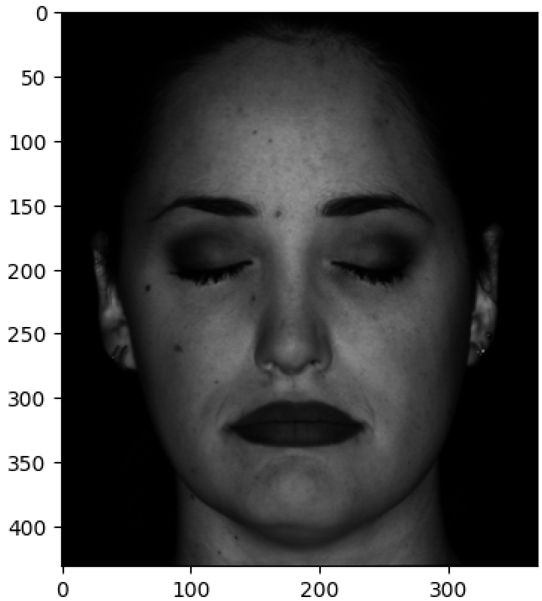
$$I = (U\Sigma^{1/2})(\Sigma^{1/2}V^T)$$

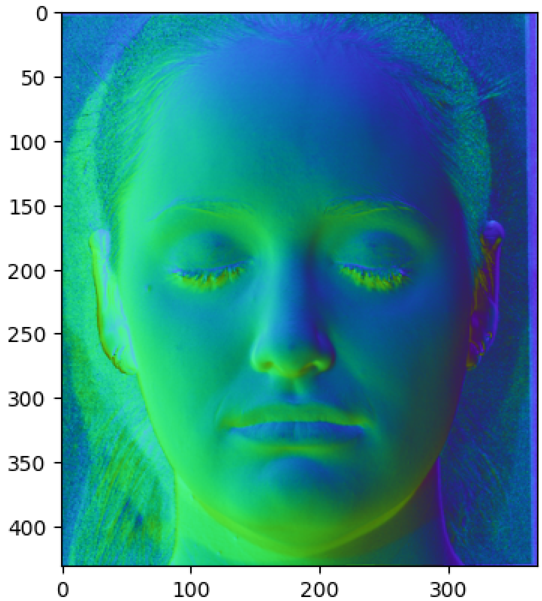
The light sources are aproximate by $\hat{L}^T = (U\Sigma^{1/2})$, while the psuedonormals are approximated by $\hat{B} = (\Sigma^{1/2}V^T)$.

Q 2 (b): Calculation and visualization (10 points)

In [9]:
```python
def estimatePseudonormalsUncalibrated(I):

    """
    Question 2 (b)

    Estimate pseudonormals without the help of light source directions.

    Parameters
    ----------
    I : numpy.ndarray
        The 7 x P matrix of loaded images

    Returns
    -------
    B : numpy.ndarray
        The 3 x P matrix of pesudonormals

    L : numpy.ndarray
        The 3 x 7 array of lighting directions

    """

    B = None
    L = None

    # Get SVD
    U, S, Vh = np.linalg.svd(I, full_matrices=False)

    # Pick the first 3 singular values
    U       = U[:, :3]
    S_rank3 = np.diag(S[:3])
    Vh      = Vh[:3, :]

    # Split singular value matrix in two matrix
    s_sqrt = np.sqrt(S_rank3)

    L_hat = U @ s_sqrt  # U \Sigma^{1/2}
    B_hat = s_sqrt @ Vh # \Sigma^{1/2}V^T

    aprox_I = L_hat @ B_hat
    error = np.linalg.norm(I - aprox_I)
    assert error < 5, "Error is too high"

    L = L_hat.T
    B = B_hat
    return B, L
```

```
# Part 2 (b)
I, L, s = loadData(data_dir)
B, LEst = estimatePseudonormalsUncalibrated(I)
albedos, normals = estimateAlbedosNormals(B)
albedoIm, normalIm = displayAlbedosNormals(albedos, normals, s)
plt.imsave('figs/2b-a.png', albedoIm, cmap = 'gray')
plt.imsave('figs/2b-b.png', normalIm, cmap = 'rainbow')
```





Q 2 (c): Comparing to ground truth lighting

A: As seen in the print below, the true lighting directions are vastly different than the estimated ones. This is because the choosen factorization scheme only garantees that the aproximated I image will be the same, however it offers no garantees on the choosen factors.

To obtain factors that more closely resemble the true lightning, we can play around with the factorization. For instance, we can apply symetric transformations to each of the factors, such that we permute and scale the the matrixes but still mantain the same output. Note that for this to work, the choosen transformation must be invertible and a 3×3 matrix.

$$I = (\hat{L}T)^T(T^{-1}\hat{B})$$

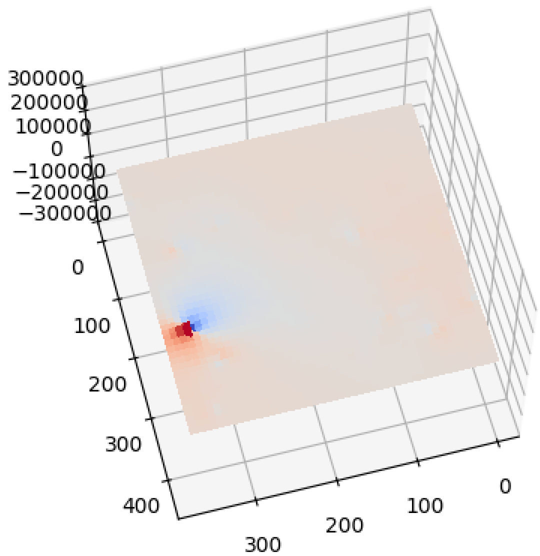
In [10]:
```
np.set_printoptions(precision=4, suppress=True)
print("True light directions (L):\n", L)
print("\nEstimated light directions (LEst):\n", LEst)
```

```
True light directions (L):
 [[-0.1418  0.1215 -0.069   0.067  -0.1627  0.      0.1478]
 [-0.1804 -0.2026 -0.0345 -0.0402  0.122   0.1194  0.1209]
 [-0.9267 -0.9717 -0.838  -0.9772 -0.979  -0.9648 -0.9713]]

Estimated light directions (LEst):
 [[-2.9927 -3.87   -2.408  -3.745  -3.5914 -3.3867 -3.3525]
 [ 0.9478 -2.3171  0.4991 -0.626   2.3257  0.4661 -0.7927]
 [ 1.8793  1.0146  0.4294 -0.0173 -0.3108 -0.9127 -1.883 ]]
```

Q 2 (d): Reconstructing the shape, attempt 1 (5 points)

It does not look like a face. As explained in c), given that the aproximation has no garantees the quantities will ressemble the real conditions, there is ambiguity in the process. In this case, the estimated normal did not yield integrable partials.

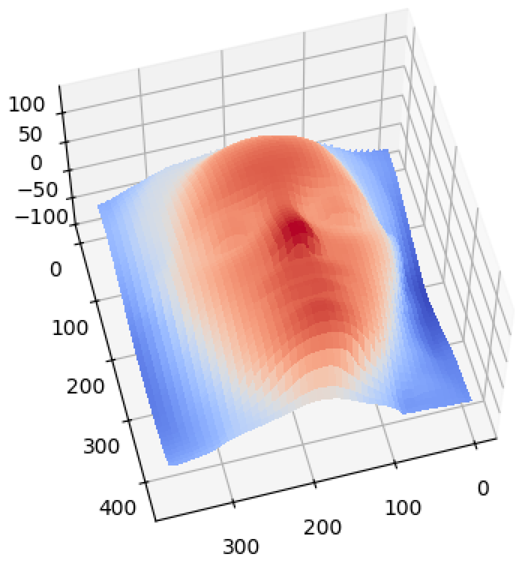
In [11]:
```
# Part 2 (d)
surface = estimateShape(normals, s)
plotSurface(surface)
```



Q 2 (e): Reconstructing the shape, attempt 2 (5 points)

A: The reconstructed face more closely resembles the calibrated reconstruction. However, in general there is less distinction with the background, and in the edges the shape of face is less defined. For example, looking at the forhead, in the calibrated reconstruction the "roundness" is much more apparent; while in the uncalibrated case, it appears more flat.

In [12]:
```
# Part 2 (e)
# Your code here
int_pseudonormals = enforceIntegrability(B, s)
albedos, normals = estimateAlbedosNormals(int_pseudonormals)
surface_integrable = estimateShape(normals, s)
plotSurface(surface_integrable, suffix='integrable')
```



Q 2 (f): Why low relief? (5 points)

A

As described in [1], the bas-relief ambiguity refers to the difficulty to distinguish the true depth (or relief) of a surface. In uncalibrated photometric stereo, this can cause reconstructed surfaces to appear more or less flat than the actually are. To remedy this, one can apply transformations that warp the surface such that

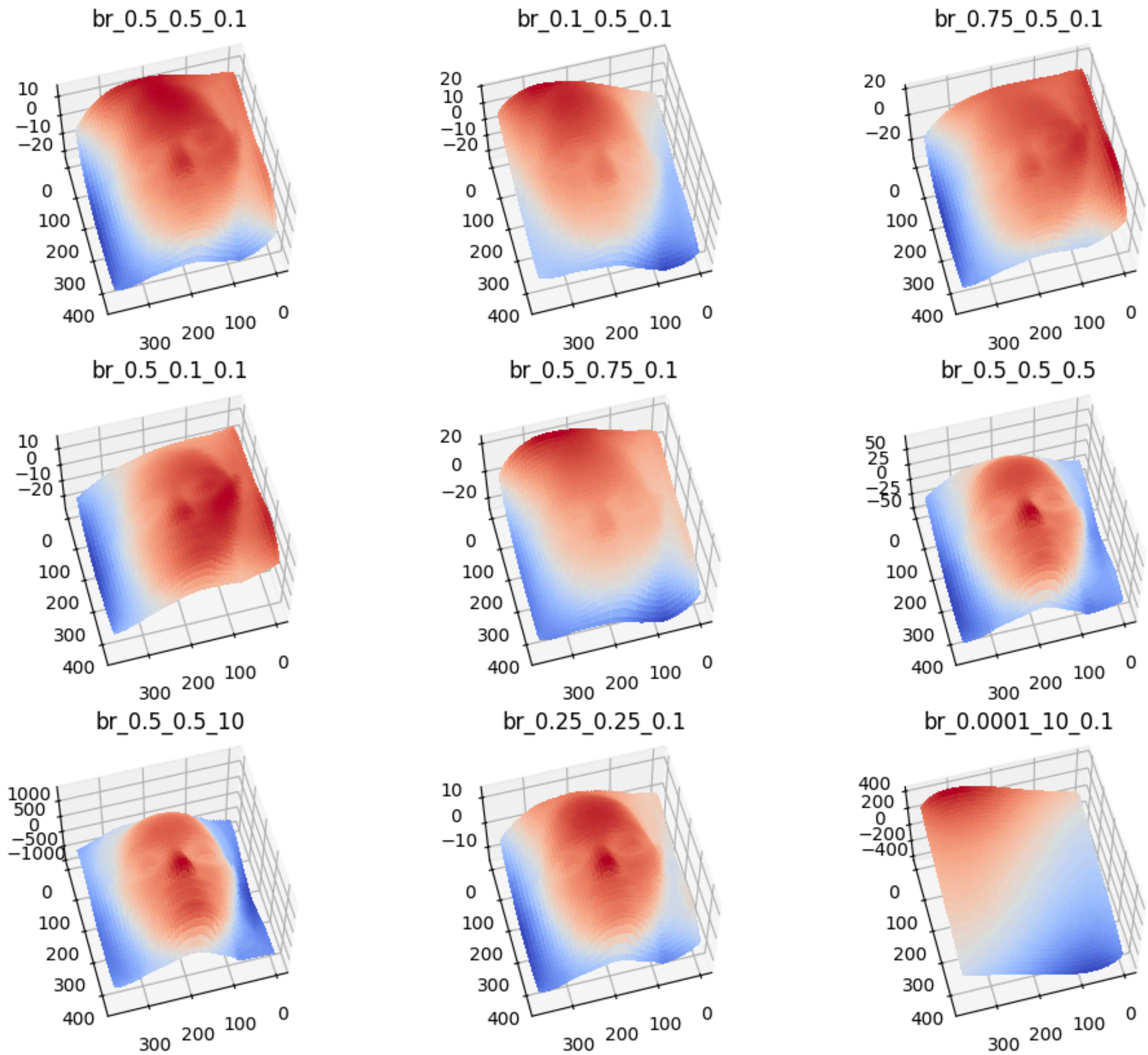
$$\hat{f}(x, y) = \lambda f(x, y) + \mu x + \nu y$$

- λ affects the height of the surfaces, changing the overall scale of the relief.
- μ scales the inclination of the surface, biasing towards the x-axis.
- ν scales the inclination, biasing towards the y-axis.

[1] P. N. Belhumeur, D. J. Kriegman, and A. L. Yuille, “The bas-relief ambiguity,” in Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition, San Juan, Puerto Rico: IEEE Comput. Soc, 1997, pp. 1060–1066. doi: 10.1109/CVPR.1997.609461.

In [13]:
```
def plotBasRelief(B, mu, nu, lam, ax):

    """
    Question 2 (f)

    Make a 3D plot of of a bas-relief transformation with the given parameters.

    Parameters
    ----------
    B : numpy.ndarray
        The 3 x P matrix of pseudonormals

    mu : float
        bas-relief parameter

    nu : float
        bas-relief parameter

    lambda : float
        bas-relief parameter

    Returns
    -------
        None

    """
    P = np.asarray([[1, 0, -mu/lam],
                                        [0, 1, -nu/lam],
                                        [0, 0,   1/lam]])
```

```
    Bp = P.dot(B)
    surface = estimateShape(Bp, s)
    plotSurfaceGrid(surface, suffix=f'br_{mu}_{nu}_{lam}', ax=ax)

# keep all outputs visible
from IPython.display import Javascript
display(Javascript('''google.colab.output.setIframeHeight(0, true, {maxHeight: 5000})'''))



cols = 3
fig, axes = plt.subplots(3, cols, figsize=(12, 10),
                         subplot_kw={'projection': '3d'})

# Base case
mu, nu, lam = 0.5, 0.5, 0.1
plotBasRelief(int_pseudonormals, mu, nu, lam, axes[0, 0])

# Vary mu (x-axis)
plotBasRelief(int_pseudonormals, 0.1, nu, lam, axes[0, 1])
plotBasRelief(int_pseudonormals, 0.75, nu, lam, axes[0, 2])

# Vary nu (y-axis)
plotBasRelief(int_pseudonormals, mu, 0.1, lam, axes[1, 0])
plotBasRelief(int_pseudonormals, mu, 0.75, lam, axes[1, 1])

# Vary lam (z-axis)
plotBasRelief(int_pseudonormals, mu, nu, 0.5, axes[1, 2])
plotBasRelief(int_pseudonormals, mu, nu, 10, axes[2, 0])

# Vary mu and nu (inclination)
plotBasRelief(int_pseudonormals, 0.25, 0.25, lam, axes[2, 1])
plotBasRelief(int_pseudonormals, 0.0001, 10, lam, axes[2, 2])
```



Q 2 (g): Flattest surface possible (5 points)

A: for the flattest surface posible we want no biasing in the inclinations so $\mu = 0$ and $\nu = 0$. We want to reduce the scale of the image as much as possible, without setting it to 0, as this would result in a non-exisiting surface. For example, we can achieve a very falt surface with $\mu = 0, \nu = 0, \lambda = 1e^{-100}$

Q 2 (h): More measurements

A: Acquiring more images will not resolve the ambiguity. Though adding more images might result in a better shape estimation, the ambuity arises from the fact that in uncalibrated photometric stereo, the surface can only be determined up to a certain rotation and skew.

This is because the underlying mechanism to estimate the lightning sources and the pseudonormals remains the same. While adding more images might produce less noisy singular valeus, the factors themselves are not guaranteed to be faithfull to the real conditions under which the photographs where captured.