Computer Vision

Anthony +22421378

Introduction

Given that the intensity of a single pixel in a single image does not provide information about the surface orientation or surface normal *n* of a particular object. **Photometric stereo** uses multiple images of an object taken under different lighting conditions from different angles to obtain the surface orientation. With images of an object taken under different lighting conditions *S* gives us a measure of some intensity value of each pixel in that lighting condition. Given the measure of the image intensity value *I* produced by each light source S_i Looking at these intensity values helps to overcome the ambiguity in estimating the surface orientation. Thus, **Photometric stereo** is a way to estimate the 3D shape of an object from images of the same object taken with different light sources oriented at different positions.

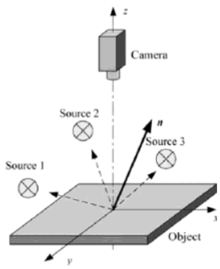


Figure 1 : Principle of photometric stereo.

Basic idea

Step 1: Acquire *K* Images with *K* Known Light Sources

- Capture *K* images of the object, each under a different known light source direction.

Step 2: Construct Reflectance Map for Each Source Direction

- Using the known source direction and **BRDF**, construct a reflectance map for each source direction.

Step 3: Compute Surface Normal at Each Pixel

- For each pixel location *(x,y)*, find *(p,q)* as the intersection of *K* reflectance curves from different light sources.
- The point *(p,q)* gives the surface normal at pixel *(x,y)*.

Photometric Stereo : Lambertian Case

In performing photometric stereo with the Lambertian model of diffusive reflection, I model the equations so that the image intensity is measured at a point *(x,y)* of the object or at an image pixel under three different light sources, as follows:

$$I = \frac{\rho}{\pi} n \cdot s$$

Where I is the image intensity of each light source S_i. Where n is the unit normal vector where is a normal surface. ρ is the albedo of the image, where π can be neglected. In the image intensity values are given as a vector that has been measured at each point in the image. Where S is a known $m*m$ matrix for m different light sources that we know. From this set of equations, we have two unknown variables, the image albedo and the surface normal. Where it is speculated that the image albedo is not constant but varies from point to point. With these measurements and set of assumptions, we can solve the matrix equations for n and albedo. The solution is given below:

$$I(x,y) = \rho(x,y) \cdot L \cdot N(x,y)$$

Equation for compute Surface normal and albedo from above equation:

$$N(x,y) = (L^T L)^{-1} \cdot L^T \cdot I(x,y)$$

$$\rho(x,y) = \| N(x,y) \|$$

Handling shadows and Highlights in Photometric Stereo

1. For each pixel *(x,y)*, the intensity values from all images corresponding to different lighting direction.

$$I = \{I_1(x,y), I_2(x,y), \ldots, I_n(x,y)\}$$

2. Sort the intensity value for each pixels and discard a percentage of the darkest and brightest values, which likely correspond to shdaows and highlights.

$$Num\_to\_discard = [\frac{percentage \cdot n}{100}]$$

3. Apply the Lambertian photometric stereo model to the remaining valid intensity values based on the second step.

Code Implementation

Load_datasetdir.py: this file uses for normalize dataset and loading dataset (images ,filename ,light direction and light intensities, mask)

```
def numerical_sort(value):

    nums = re.compile(r"(\d+)")
    parts = nums.split(value)
    parts[1::2] = map(int, parts[1::2])
    return parts

# convert image to vector
def imread_image(list_object=None, I=None, flag=-1, scale=1):
    # checking list object
    if list_object == None:
        print("The list of object is empty")
        exit()
    # list of images name
    image_names = list_object
    # sorted file paths numerical
    image_names = sorted(image_names, key=numerical_sort)
    image_list = []

    # reading images and normalize images
    for i, path in enumerate(image_names):
        image = cv2.resize(cv2.imread(path, flag), None, fx=scale, fy=scale)
        # normalize image
        if np.any(I):
            image = image / I[i]
        image_list.append(image)
    return image_list
```

```
# loading images and convert image from image into vector (images + mask)

def loader_dataset(object: str):

    data_path = "./pmsData/"
    # loading light direction
    L_direction = loadtxt(data_path + object + "/light_directions.txt")
    # loading light intensities
    intensities = loadtxt(data_path + object + "/light_intensities.txt")
    # load file name
    file = loadtxt(data_path + object + "/filenames.txt", dtype=str)
    filenames = []
    for i in range(len(file)):
        temp = data_path + object + "/" + file[i]
        filenames.append(temp)
    # transform image into vector
    images = imread_image(filenames, I=intensities)
    # loading mask image into vector
    mask = imread_image([data_path + object + "/mask.png"])[0]
    return images, mask, L_direction
```

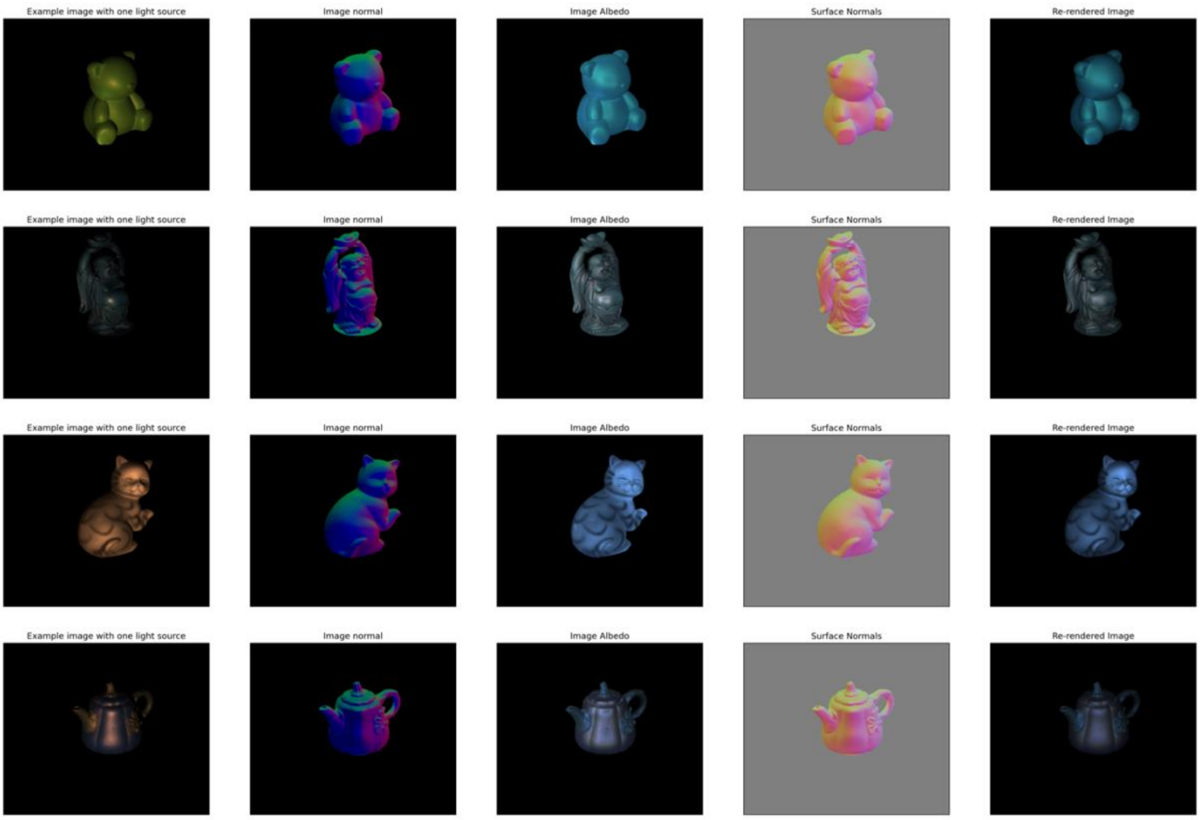
Pms.py : Algorithm PMS and Handling shadows and Highlights

```
import numpy as np
import cv2


def discard_extremes(I, percentage: int):
    num_image = len(I)
    # sort intensities for each pixels
    I_sorted = np.sort(I)
    num_to_discard = int(num_image * percentage / 100)
    for i in range(num_to_discard):
        I_sorted[i] = 0    # Discard darkest values
        I_sorted[-(i + 1)] = 0    # Discard brightest values
    return I_sorted


def pms(images, L_list, percentage):
    print("PMS algorihtm")
    L = np.array(L_list)    # convert L_list into np array
    L_transpose = L.T    # transpose of L
    height, weight = images[0].shape[:2]
    print(height, weight)
    # intilize image normal and image albedo
    image_normal = np.zeros((height, weight, 3))
    image_albedo = np.zeros((height, weight, 3))
```

```
    # intilize intensity
    size_L = len(L_list)
    I = np.zeros((size_L, 3))
    print(weight)
    for x in range(weight):
        for y in range(height):
            for i in range(len(images)):

                I[i] = images[i][y][x]
            # handling shadow and outlier
            I = discard_extremes(I, percentage)
            # solve surface normal
            temp1 = np.linalg.inv(np.dot(L_transpose, L))    # (S^T . S)^-1
            temp2 = np.dot(L_transpose, I)    # S^T . I
            N = np.dot(temp1, temp2).T
            # compute albedo rho
            rho = np.linalg.norm(N, axis=1)
            image_albedo[y][x] = rho

            # compute rgb using luminosity
            N_gray = N[0] * 0.0722 + N[1] * 0.7152 + N[2] * 0.2126
            Nnorm = np.linalg.norm(N)
            if Nnorm == 0:
                continue
            image_normal[y][x] = N_gray / Nnorm
    # Normalize and visualize the normal image
    image_temp = image_normal
    image_normal_rgb = ((image_normal * 0.5 + 0.5) * 255).astype(np.uint8)
    image_normal_rgb = cv2.cvtColor(image_normal_rgb, cv2.COLOR_BGR2RGB)

    image_albedo = (image_albedo / np.max(image_albedo) * 255).astype(np.uint8)
    print(image_albedo)

    # Re-rendered image
    view_dir = np.array([0, 0, 1])    # View direction
    light_dir = view_dir    # Illumination direction is the same as the view direction
    light_dir = light_dir / np.linalg.norm(
        light_dir
    )    # Normalize the light direction if necessary
    Re_rendered_image = re_render(image_normal, image_albedo, light_dir)
    return image_normal, image_albedo, image_normal_rgb, Re_rendered_image


def re_render(image_normal, image_albedo, light_dir):
```

```
    height, width, _ = image_normal.shape
    re_rendered_image = np.zeros((height, width, 3), dtype=np.float32)    # RGB output
    light_dir = light_dir / np.linalg.norm(light_dir)
    print("Re-rendering image")
    for y in range(height):
        for x in range(width):
            normal = image_normal[y][x]
            albedo = image_albedo[y][x]

            normal_norm = np.linalg.norm(normal)
            if normal_norm != 0:
                normal = normal / normal_norm
            dot_product = np.dot(light_dir, normal)
            dot_product = max(dot_product, 0)
            re_rendered_image[y][x] = albedo * dot_product
    re_rendered_image = (re_rendered_image).astype(np.uint8)

    return re_rendered_image
```

Result



Different using deal with shadows and highlights and not using

Deal with shadows and highlights

Not using deal with shadows and highlights

Discussion

The biggest problem in the implemented algorithm was dealing with shadow and outliers present in the input images, which lowered the accuracy of the surface normal and albedo recovery (Selection percentage). The best datasets were obtained with this algorithm on high-quality, well-lightened pictures where diffuse surfaces and low noise allowed more exact results to be derived. It struggled with shiny surfaces or noisy data, where only very large pixel values that were rejected still caused distortion in such cases as pictures with specular reflections or pictures poorly lightened. To make the algorithm even better, the incorporation of a model for outlier detection enhancement would yield more robust and accurate results, specifically in real applications.`