MIT CSAIL

6.819/6.869 Advances in Computer Vision

Spring 2022

Problem Set 2

Jiahui Tang

jiahuita@mit.edu

Problem 1

(a)

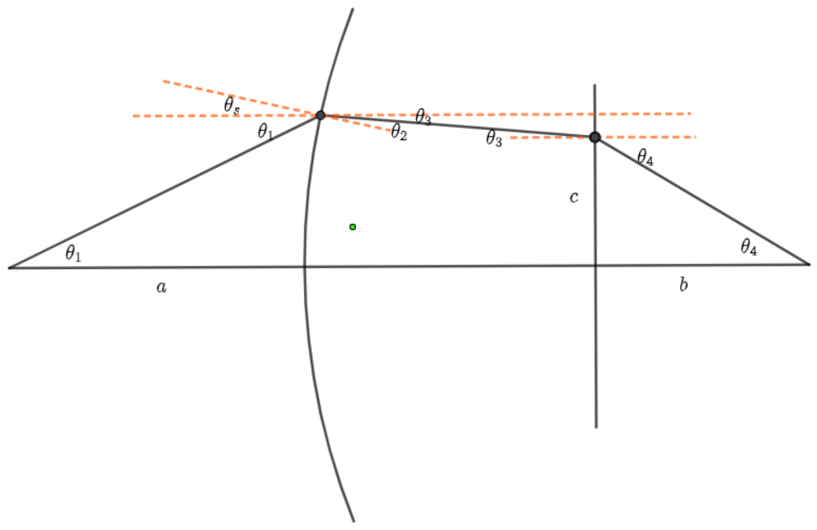

Figure 1: Diagram of a plano-convex lens (borrowed from *Yuanbiao Wang* from Piazza discussion)

As we assume the index of refraction of air is 1, and denote the index of refraction of lens glass as n, by Snell's Law, we can obtain the below five equations.

Angle	**Description**	**Relation**	**Reason**
θ_1	initial angle from optical axis	$\theta_1 = \frac{c}{a}$	small angle approx.
θ_2	angle of refracted ray wrt front sfc norm	$n\theta_2 = \theta_1 + \theta_s$	Snell's Law, small angle approx.
θ_3	angle of refracted ray wrt back sfc norm	$\theta_s = \theta_2 + \theta_3$	geometry in front surface
θ_4	angle of ray exiting lens wrt back sfc norm	$\theta_4 = n\theta_3$	Snell's Law, small angle approx
θ_3	angle of refracted ray wrt back sfc norm	$\theta_4 = \frac{c}{b}$	small angle approx.

Table 1: Relations between angles of rays passing through the plano-convex lens

(b) If we start from the relation in the above table from θ_4, and substitute for each angle using the relation in each line above, up through $\theta_1 = \frac{c}{a}$, we can algebraically eliminate the angle θ_1 through θ_4 to find the condition on θ_s which allows for the desired focusing to occur:

$$\theta_s = \frac{c}{n-1}(\frac{1}{a} + \frac{1}{b}) \tag{1}$$

As we know from textbook, for a spherical lens surface, curving according to a radius R, we have $sin(\theta_s) = \frac{c}{R}$. For small angles θ_s, this reduces to

$$\theta_s = \frac{c}{R} \tag{2}$$

Combing above two formula, with the *Lensmaker's Formula*

$$\frac{1}{f} = \frac{1}{a} + \frac{1}{b} \tag{3}$$

We could obtain that

$$\frac{1}{f} = \frac{1}{a} + \frac{1}{b} = \frac{n-1}{R} \tag{4}$$

Thus, we can obtain that the *lens focal length, f* is

$$\boxed{f = \frac{R}{n-1}} \tag{5}$$

Problem 2

(a) **Triangulation**

According to the geometry for the stereo system and similar triangle, we have

$$\frac{X_c}{u_c} = \frac{Y_c}{v_c} = \frac{Z_c}{f} \tag{6}$$
$$\frac{X_p}{u_p} = \frac{Y_p}{v_p} = \frac{Z_p}{f} \tag{7}$$
$$X_p = X_c - d \tag{8}$$
$$Z_p = Z_c \tag{9}$$
$$Y_p = Y_c \tag{10}$$
$$\tag{11}$$

and

$$\frac{d}{Z_c} = \frac{d + u_p - u_c}{Z_c - f} \tag{12}$$

$$u_c = \frac{fX_c}{Z_c} \tag{13}$$

$$v_c = \frac{fY_c}{Z_c} \tag{14}$$

Solve for the above, overall we have:

$$X_c = \frac{du_c}{u_c - u_p} \tag{15}$$

$$Y_c = \frac{dv_c}{u_c - u_p} \tag{16}$$

$$Z_c = \frac{df}{u_c - u_p} \tag{17}$$

$$X_p = X_c - d \tag{18}$$

$$Z_p = Z_c \tag{19}$$

$$Y_p = Y_c \tag{20}$$

$$\tag{21}$$

(b) **Simulating a projected laser**

The fixed linear transformation could be represented by the below transformation matrix in homogeneous coordinates

$$T_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & f \\ 0 & 0 & f & 0 \end{bmatrix}, T_2 = \begin{bmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, T_3 = \begin{bmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{bmatrix}, T = T_3T_2T_1$$

Codes are attached below:

```
def getTransformationMatrices(f, d):

    T1 = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, f], [0, 0, f, 0]])

    T2 = np.array([[1, 0, 0, d], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]])

    T3 = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1/f, 0]])

    T = np.matmul(np.matmul(T3, T2), T1)

    return T, T1, T2, T3
```

(c) **Laser scan of a scene**

Scanning from left to right: the laser path doesn't wiggle in the vertical direction. Camera view and projector view in the dark are similar.

Scanning from top to bottom: the laser path wiggle in the horizontal direction from the camera's view. Camera view wiggles comparing to projector view.

So when we transform of projector image to camera image, we are transform using matrix $T = T_3T_2T_1$ to transform $(u_p, v_p, 1/Z_p)$ to (u_c, v_c).

The result would be in the form of $\begin{bmatrix} 1 & 1 & df & d \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1/Z_p \\ 1 \end{bmatrix} = \begin{bmatrix} df/Z_p \\ v_p \\ 1 \end{bmatrix}$

If we are scanning from top to bottom, u_c is dependent of variable d, thus it will wiggle given different depth at that pixel. If we are scanning from left tot right, v_c equals to v_p, independent of other variables, thus we can see the camera view and projector view in the dark are similar without wiggles in vertical directions.

(d) **Inferring camera depth**

Using previous equation from (a), where we get

$$Z_c = \frac{df}{u_c - u_p}$$

```
def inferDepthFromMatchedCoords(uv_c, uv_p, f, d):
    Z_c = flatten(np.array([d*f/(uv_c[i][0]-uv_p[i][0]) for i in range(len(uv_c))]))
    return Z_c
```

Below is a screenshot of obtained result.

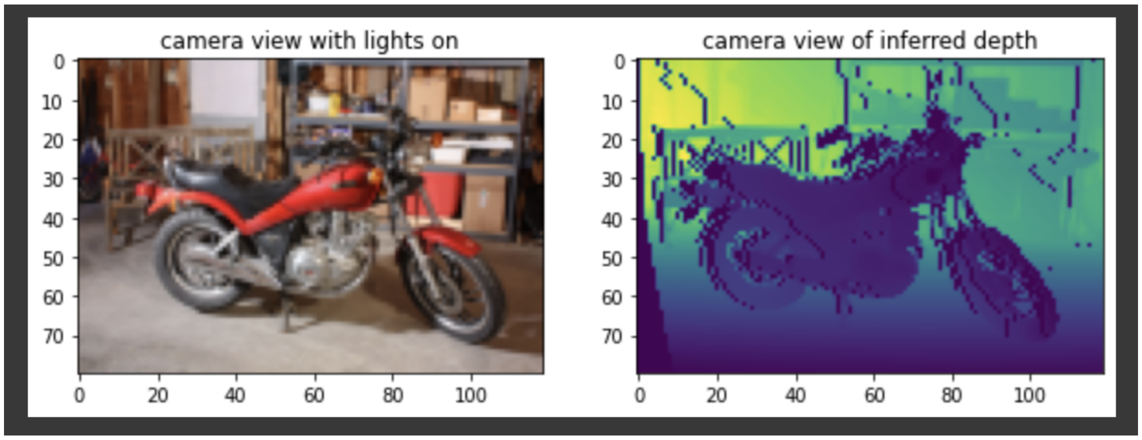

Figure 2: Screenshot for inferring camera depth of motorcycle

From this graph, I think it did a decent job in inferring depth of motorcycle and the objects in the background. However, it failed to infer depth in some area with occlusions, e.g dark colors in motorcycle surroundings and lower left border of the picture, indicating no light can be projected there. Besides, due to resolution of projector, as we calculating by iterating across pixels in the projector's virtual image plane, this depth view is more pixeled and low resolution with more graininess.

(e) **Projecting structured light**

Code as below

```
def getImgCoordinatePairs(Z_p_img, T, img_size, cx_p, cy_p, cx_c, cy_c):

    x, y = (np.meshgrid(np.arange(img_size[1]),  np.arange(img_size[0])))

    # xy_p is Nx2
    xy_p = np.concatenate([x.reshape(-1, 1), y.reshape(-1, 1)], axis = 1)
    # Z is N*1, flatten from Z_p_img
    xy_c = transform_xy_p_to_xy_c(xy_p, flatten(Z_p_img), cx_p, cy_p, cx_c, cy_c, T)

    return xy_c, xy_p
```

The result of stripe lights rendered on camera view are attached below:

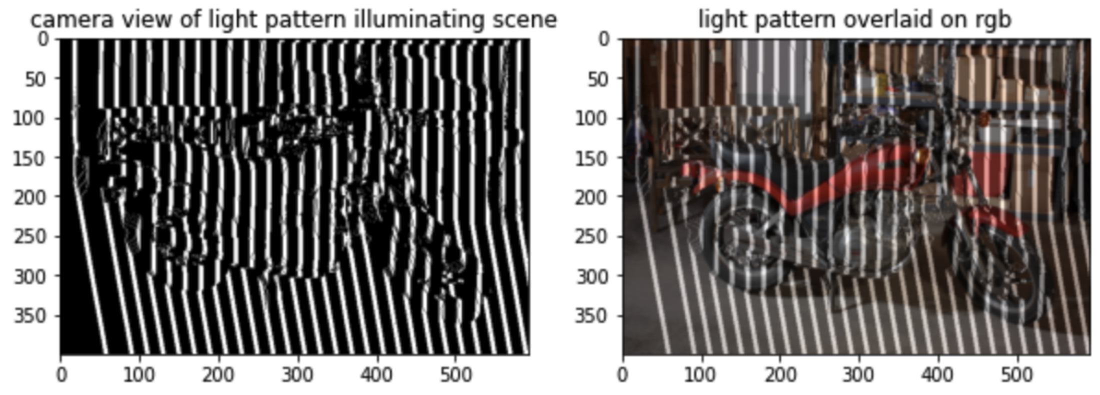
camera view of light pattern illuminating scene
light pattern overlaid on rgb
0
50
100
150
200
250
300
350
0
100
200
300
400
500

Figure 3: render illuminated image of vertical lights

For alternative pattern of stripes, I illuminate it with horizontal lights by transpose the light matrix. The result looks like below.

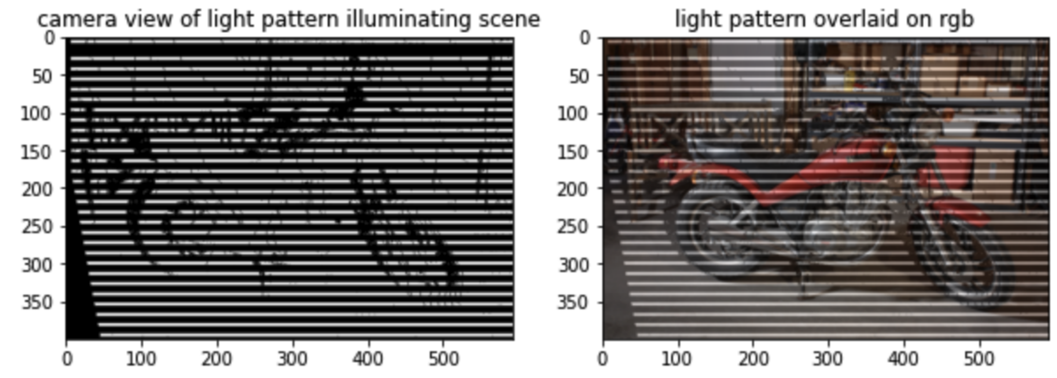

Figure 4: render illuminated image of horizontal lights

Occlusion appears near the two motorcycle wheels and its border, and the left lower corner, where the regions of camera image showing black color. And project lights are blocked by occlusion, and the depth of image cannot be inferred from the view of projector. It is particularly obvious in vertical lights (similar to what we discussed in c, x_c depends of depth, and thus it fluctuates and be blocked by occlusion where depth is unsure).

(f) **Inferring depth from structured light**

I choose to generate light pattern with intensity increases linearly as a function of x_p coordinates in the light matrix. It shows the light get increasingly bright and intense from left to right, with the right most part the brightest. In decoder function F, Y would be the same from projector view and camera view. X will be taken from value of light.

```
def getStructuredLight(img_size):
    L_p_img = np.zeros((img_size[0], img_size[1]))
    # light changes with x_p coordinates
    for i in range(L_p_img.shape[0]):
       for j in range(L_p_img.shape[1]):
         L_p_img[i, j] += j
    return L_p_img

def F(L_c, xy_c):
    # L_c is Nx1
    # xy_c is Nx2
    # xy_p should be Nx2

    xy_p = np.concatenate([L_c.reshape(-1, 1), np.array([i[1] for i in xy_c]).reshape(-1, 1)
    return xy_p
```

An image of the decoded path can be seen below, where the lights are getting brighter from left to right. We can see the depth are inferred in the picture pretty decently, with higher depth around motorcycle. Thus, the F function could directly take the y coordinates, with x coordinates taken from L_c.

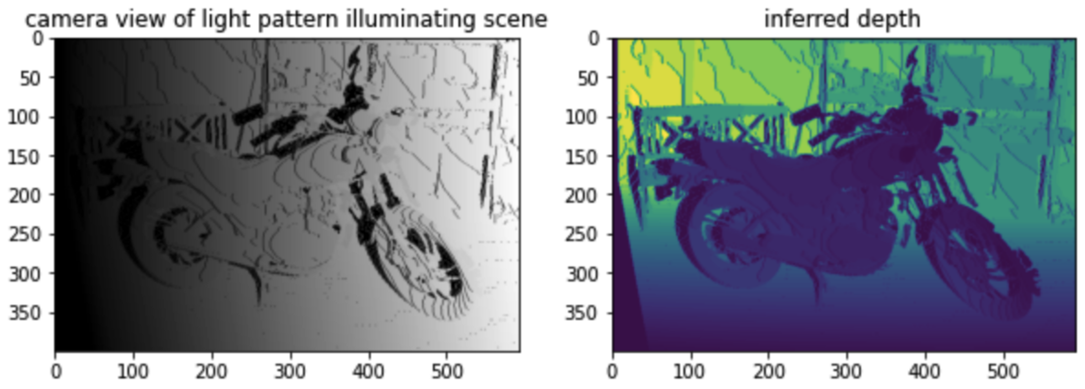

Figure 5: Decoded depth

Other light pattern paired with decoder F: Still, we want to project a pattern of light such that a simple function of each pixel intensity (or patch) tells us the x_p coordinate. We could create light patterns that increase not linearly, but as an exponential or polynomial function of x_p. For example, let $L_p = x_p^2$. In this way, we could set decoder to be $x_p = \sqrt{x_c}$, and again take y from the same camera coordinates y.