ELE510 Image Processing with robot vision: LAB, Exercise 2, Image Formation.

Daniel Fylling

Abstract

This exercise explores basic fundamentals in image processing such as "PSF" -point spread function, filter / sensor alternatives for image capturing and how to use scaling coefficients to translate real world points into a digital format.

The concept of transforming scene points to camera sensor coordinates using a transformation matrix was introduced. A Python function was provided to calculate image points from input scene points, considering the field of view and pixel count.

Both a self-developed solution and a matrix-based solution were presented, with a comparison indicating similar results. Both functions were made able to identify points outside the field of view. The two last points in the given input were identified as outside the field of view.

Additionally, a performance comparison showed a significant difference in execution time between the self-developed and matrix-based solutions, with the self-developed, array-based, solution being approximately 30% faster.

Overall, this exercise deepened understanding of image formation, camera parameters, and transformation matrices in the context of image processing and robot vision.

Problem 1

a) What is the meaning of the abbreviation PSF? What does the PSF specify?

The point spread function specifies the shape that a point will take on the image plane. Also called impulse response. I we imagine a single ray of light travelling through the optics of a camera, then by the time it reaches the optical sensors it will have spread out to some degree. This will depend on the quality of the equipment and the manual adjustment of the current focus.

b) Use the imaging model shown in Figure 1. The camera has a lens with focal length $f = 40\text{mm}$ and in the image plane a CCD sensor of size $10\text{mm} \times 10\text{mm}$. The total number of pixels is 5000×5000. At a distance of $z_w = 0.5\text{m}$ from the camera center, what will be the camera's resolution in pixels per millimeter?

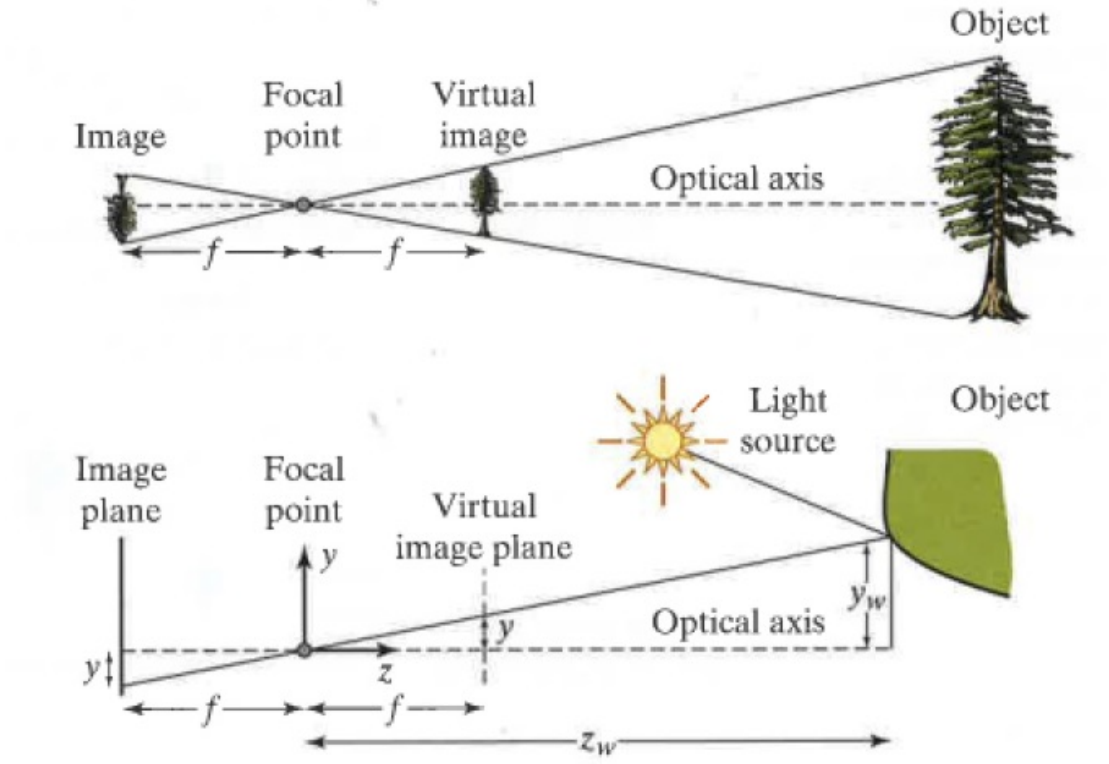

Figure 1: Perspective projection caused by a pinhole camera. Figure 2.23 in [2].

$$Resolution = Numberofpixels/Fieldofview$$

$$FOV = (y * z_w)/f$$

$$Resolution = (5000 * 40)/(10 * 500) = 40pixels/mm$$

c) Explain how a Bayer filter works. What is the alternative to using this type of filter in image acquisition?

A Bayer filter has an sensor mosaic where each tile is only sensitive to a ceratain range of frequencies; red, green and blue. A de-mosaicing / interpolating algorithm is applied to approximate the missing values.

The main alternative is for each sensor tile to accept all wavelengths of light - then split the wavelengths internally to catch the intensity of each. Splitting the incoming light is done by use of prisms in each reseptor.

d) Briefly explain the following concepts: Sampling, Quantization, Gamma Compression.

Sampling - related to filters as discussed in c). Formally it is defined as the process of discretizing the continuous wavelength function into pixel information. Essential for converting an analog image or input to a digital form.

Quantization - Convert the input signal from a particular sensor into a given gray scale. As our eyes are more sensitive to green light and less to blue light, the weighting from

each component may be skewed if the aim is for the image to look most natural to the human eye.

Gamma Compression - transforming input intensity levels by use of a certain function. Useful also because of the logarithmic nature of our senses. Light that is physically 10 times brighter in terms of energy (w/m2), may only appear twice as bright to the human eye. Can also be used to correct or make better use of the available light spectrum to enhance an image.

Problem 2

Assume we have captured an image with a digital camera. The image covers an area in the scene of size $1.024\mathrm{m} \times 0.768\mathrm{m}$ (The camera has been pointed towards a wall such that the distance is approximately constant over the whole image plane, *weak perspective*). The camera has 4096 pixels horizontally, and 3072 pixels vertically. The active region on the CCD-chip is $8\mathrm{mm} \times 6\mathrm{mm}$. We define the spatial coordinates (x_w, y_w) such that the origin is at the center of the optical axis, x-axis horizontally and y-axis vertically upwards. The image indexes (x, y) is starting in the upper left corner. The solutions to this problem can be found from simple geometric considerations. Make a sketch of the situation and answer the following questions:

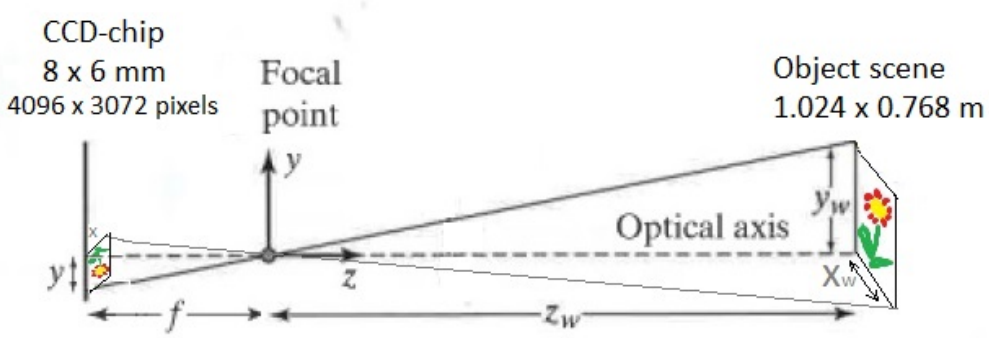

Figure 2: Sketch of scene projected onto virtual image plane, with origin in the center of optical axis.

a) What is the size of each sensor (one pixel) on the CCD-chip?

$8mm/4096 \times 6mm/3072 \ = 2nm \times 2nm$

b) What is the scaling coefficient between the image plane (CCD-chip) and the scene? What is the scaling coefficient between the scene coordinates and the pixels of the image?

The scaling coefficient between image plane and scene is defined as f / z_w for weak perspective. Since we don't have these values, but everything here scales linearly, we can calculate the scaling for each x- and y-directions and compare them to see if they match:

$\alpha_x = 1024mm \div 8mm = 128$

$\alpha_y = 768mm \div 6mm = 128$

The scaling factors in both directions 128, and hence match.

The scaling coefficient between scene and pixels are:

$\alpha_x = 4096pixels \div 1.024m = 4000pixels/m$

$\alpha_y = 3072pixels \div 0.768m = 4000pixels/m$

Problem 3

Translation from the scene to a camera sensor can be done using a transformation matrix, T.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = T \begin{bmatrix} x_w \\ y_w \\ 1 \end{bmatrix} \tag{1}$$

where

$$T = \begin{bmatrix} \alpha_x & 0 & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

α_x and α_y are the scaling factors for their corresponding axes.

Write a function in Python that computes the image points using the transformation matrix, using the parameters from Problem 2. Let the input to the function be a set of K scene points, given by a $2 \times K$ matrix, and the output the resulting image points also given by a $2 \times K$ matrix. The parameters defining the image sensor and field of view from the camera center to the wall can also be given as input parameters. For simplicity, let the optical axis (x_0, y_0) meet the image plane at the middle point (in pixels).

Test the function for the following input points given as a matrix:

$$\mathbf{P}_{in} = \begin{bmatrix} 0.512 & -0.512 & -0.512 & 0.512 & 0 & 0.35 & 0.35 & 0.3 & 0.7 \\ 0.384 & 0.384 & -0.384 & -0.384 & 0 & 0.15 & -0.15 & -0.5 & 0 \end{bmatrix} \tag{3}$$

Comment on the results, especially notice the two last points!

In [ ]:
```
# Import the packages that are useful inside the definition of the weakPerspecti
import math
import numpy as np
import matplotlib.pyplot as plt
```

I first solved this by finding my own method to build understanding, then I made the function as described in the problem text. In the end, both functions do the same thing

My thought process for the formulas below are as follows:

1. Project the pixel grid onto the scene.
2. Transform input position from having origo in the center of the scene to the upper left corner. Notice that y-input is reversed to achieve this.
3. Divide by respective FOV to normalize and multiply by pixel amount in each direction to scale position to pixel grid.
4. Round down to nearest whole number as pixels are discrete.

The solution function will check that input points are within FOV and respond to points outside.

$$x_p = floor(\frac{x_w + FOV_x * 0.5}{FOV_x} \times N_{pixels-x}) \tag{4}$$

$$y_p = floor(\frac{-y_w + FOV_y * 0.5}{FOV_y} \times N_{pixels-y}) \tag{5}$$

In [ ]:
```
"""
Function that takes in input:
- FOV: field of view, [x-direction, y-direction] in meters
- sensorsize: size of the sensor, [x-direction, y-direction] in mm
- n_pixels: camera pixels, [x-direction, y-direction]
- p_scene: K input points (2xK matrix) in meters

and return the resulting image points given the 2xK matrix
"""
def weakPerspective_self(FOV, sensorsize, n_pixels, p_scene):

    # Creating a mask to identify input points that are outside FOV:
    mask_x = np.logical_or(((p_scene[0] + FOV[0]/2)/FOV[0] > 1),
                           ((p_scene[0] + FOV[0]/2)/FOV[0] < 0))
    mask_y = np.logical_or(((p_scene[1] + FOV[1]/2)/FOV[1] > 1),
                           ((p_scene[1] + FOV[1]/2)/FOV[1] < 0))
    mask = np.logical_or(mask_x, mask_y)
    if sum(mask) > 0:
        print(f'{sum(mask)} point(s) are  outside FOV, at\
              location(s),{np.where(mask)[0]}')

    # Calculating pixel positions based on input
    p_pixel_x = np.floor((p_scene[0] + FOV[0]/2)/FOV[0] * n_pixels[0])
    p_pixel_y = np.floor((-p_scene[1] + FOV[1]/2)/FOV[1] * n_pixels[1])

    return np.array([p_pixel_x, p_pixel_y])
```

In [ ]:
```
"""
Function that takes in input:
- FOV: field of view, [x-direction, y-direction] in meters
- sensorsize: size of the sensor, [x-direction, y-direction] in mm
- n_pixels: camera pixels, [x-direction, y-direction]
- p_scene: K input points (2xK matrix) in meters

and return the resulting image points given the 2xK matrix
"""
def weakPerspective(FOV, sensorsize, n_pixels, p_scene):
```

```
    # Creating a mask to identify input points that are outside FOV:
    mask_x = np.logical_or(((p_scene[0] + FOV[0]/2)/FOV[0] > 1),
                           ((p_scene[0] + FOV[0]/2)/FOV[0] < 0))
    mask_y = np.logical_or(((p_scene[1] + FOV[1]/2)/FOV[1] > 1),
                           ((p_scene[1] + FOV[1]/2)/FOV[1] < 0))
    mask = np.logical_or(mask_x, mask_y)
    if sum(mask) > 0:
        print(f'{sum(mask)} point(s) are  outside FOV, at\
              location(s) {np.where(mask)[0]}')

    # Constructing Transformation matrix
    alpha = np.append(n_pixels / FOV, 1)
    T = np.diag(alpha)
    T[:2,2] = n_pixels / 2

    # Preparing input matrix
    ones_row = np.ones((1, p_scene.shape[1]))
    P_in = np.vstack((p_scene, ones_row))

    P_out = np.dot(T, P_in)

    return P_out[:2]
```

In [ ]:
```
# The above function is then called using the following parameters:

# Parameters
FOV = np.array([1.024, 0.768])
sensorsize = np.array([8, 6])
n_pixels = np.array([4096, 3072])
p_scene_x = np.array([0.512, -0.512, -0.512, 0.512, 0, 0.35, 0.35, 0.3, 0.7])
p_scene_y = np.array([0.384, 0.384, -0.384, -0.384, 0, 0.15, -0.15, -0.5, 0])
p_scene = np.array([p_scene_x, p_scene_y])
```

In [ ]:
```
# Call to the weakPerspective_self() function
pimage = weakPerspective_self(FOV, sensorsize, n_pixels, p_scene)

# Result:
print(pimage)
```

```
2 point(s) are  outside FOV, at location(s) [7 8]
[[4096.    0.    0. 4096. 2048. 3448. 3448. 3248. 4848.]
 [   0.    0. 3072. 3072. 1536.  936. 2136. 3536. 1536.]]
```

In [ ]:
```
# Call to the weakPerspective() function
pimage = weakPerspective(FOV, sensorsize, n_pixels, p_scene)

# Result:
print(pimage)
```

```
2 point(s) are  outside FOV, at location(s) [7 8]
[[4096.    0.    0. 4096. 2048. 3448. 3448. 3248. 4848.]
 [3072. 3072.    0.    0. 1536. 2136.  936. -464. 1536.]]
```

Comment on the results, especially notice the two last points!

1. Both functions transform the input coordinates into the pixel space as intended. We notice again that the new origin in the pixel space was chosen as upper left corner for the self-made function and lower left corner for the matrix solution.

2. By just looking at the input values of the last two coordinate points we can tell that they are outside the FOV area. Both functions have no problems calculating these values, so a specific check was made to identify which points are outside the field of view, if any.

3. One thing to note is that the sensorsize is not used by any of the functions, and is irrelevant for this transformation.

Out of curiosity I ran timeit function to see which solution runs faster (last 2 points were omitted to not trigger the function to print the location):

In [ ]:
```
%timeit weakPerspective_self(FOV, sensorsize, n_pixels, p_scene[:,:7])
%timeit weakPerspective(FOV, sensorsize, n_pixels, p_scene[:,:7])
```

```
34.5 µs ± 2.86 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
44.8 µs ± 5.51 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

In [ ]:
```
44.8/34.5
```

Out[ ]:
```
1.298550724637681
```

As can be seen below the difference is significant, with the self-made function running approximately 30% faster than the matrix based solution. I suspect that this is related to the number of multiplications made with "ones" and "zeros" in the matrix version.

Delivery (dead line) on CANVAS: 15-09-2023 at 23:59

Contact

Course teacher

Professor Kjersti Engan, room E-431, E-mail: kjersti.engan@uis.no

Teaching assistant

Saul Fuster Navarro, room E-401 E-mail: saul.fusternavarro@uis.no

Jorge Garcia Torres Fernandez, room E-401 E-mail: jorge.garcia-torres@uis.no

References

[1] S. Birchfeld, Image Processing and Analysis. Cengage Learning, 2016.

[2] I. Austvoll, "Machine/robot vision part I," University of Stavanger, 2018. Compendium, CANVAS.