3D Data Acquisition using a Structured Light Technique

Faculdade de Engenharia
Universidade do Porto
Porto, PT

Abstract

This article presents the implementation of a system that calculates the 3D coordinates of points in an image lit by a shadow plane, by using structured light techniques. This type of systems have numerous practical applications, from industrial quality inspection to medical procedures. The proposed solution for the problem is fully described, from the camera calibration process, the light projection system calibration, the edge and line detection algorithms and the final calculation of 3D coordinates. We explain why this method is reliable and efficient, namely due to calculating the shadow plane equation instead of assuming its position in space. Some problems that were encountered along the way are described, and finally the overall results and tests of the system are analysed.

1 Introduction

Structured light techniques are an effective method for 3D data acquisition in situations where the surfaces to be measured do not have feature points. In this project we apply these techniques by projecting a shadow plane in the scene, which, combined with image processing and camera calibration methods, allow us to calculate the real world 3D coordinates of points lit by the shadow plane, from their 2D coordinates in the acquired images.

2 Setup

The setup for image acquisition [9] is comprised of a Canon EOS 250D, calibrated to take 2400x1600 resolution images, that are then converted to a 1200x800 format.

The camera is secured in a robust tripod to reduce as much movement as possible between the acquisition of the calibration images and the testing images (from button clicking or accidental vibrations).

For the shadow plane, a thin plastic pole is used, which allows for a single line to be extracted using morphological transformations, dilations and erosions (explained in further detail in section 3.3).

The light source used is the flash from a Xiaomi Pocophone F1 Smartphone, that is rotated 90º in relation to the camera, to make sure that the light beam is as thin as possible when hitting the plastic pole, resulting in sharper shadows. To elevate and immobilize this light source, an ironing board was used.

3 Proposed solution

3.1 Camera calibration

3.1.1 Intrinsic calibration

The first step is to find the intrinsic values and distortion factors of our camera. This was done by taking a sample of 20 images (see annexes) of a chessboard pattern (9x6) in different positions, rotations and orientations. These are converted into grayscale and the pattern is detected via OpenCV's findChessboardCorners, which takes the images and returns the respective image points of each chessboard. These image points are later refined by the cornerSubPix function, which grants the points sub-pixel accuracy. We then associate those image points with the object points (which is a unit coordinate matrix multiplied by the square length (2.5cm)), and store all these respective points in two arrays, *objp* and *imgp*. Using the calibrateCamera function, with the points collected from all 20 images and their positions in the real world, we obtain the camera's intrinsic values [1], distortion factors [2], rotation vector and translation vector. These latter two are discarded, since in this phase we are only interested in the camera's properties, and the camera is moved in latter phases. Below we can see the intrinsic matrix, as well as the distortion factors obtained (rounded to the thousandths).

$$\begin{bmatrix} 976.651 & 0.000 & 612.270 \\ 0.000 & 976.631 & 396.649 \\ 0.000 & 0.000 & 1.000 \end{bmatrix}$$

Figure 1: Intrinsic Matrix

$$\begin{bmatrix} -0.187 & 0.135 & 0.001 & 0.001 & 0.047 \end{bmatrix}$$

Figure 2: Distortion Factors

By looping through the object points and trying to re-project them in their specific images with the calculated values, we obtain a (initial) re-projection error of *0.0375*px. However, we will not be using the rvecs and tvecs obtained in this step, so the real re-projection error is calculated in the next subsection.

3.1.2 Extrinsic calibration

Having the intrinsic parameters of the camera, as well as its distortion factors, we must now move the camera to its final position and rotation, facing the scene we want to observe. In order to calculate the pose of the camera, we again use an image of a 9x6 chessboard on the surface of our scene, and call the functions findChessboardCorners and cornerSubPix in the same manner described in the previous subsection. However, this time we give the object points and image points as parameters to the solvePNPRansac function along with the intrinsic matrix and the distortion coefficients. This in turn gives us the camera's pose in the form of a rotation vector (a Rodrigues vector, that we later transform into a rotation matrix [3]) and a translation vector [4].

$$\begin{bmatrix} 0.999 & 0.002 & 0.017 \\ -0.008 & 0.929 & 0.371 \\ -0.015 & -0.371 & 0.929 \end{bmatrix}$$

Figure 3: Rotation Matrix

$$\begin{bmatrix} -11.002 & -6.868 & 39.242 \end{bmatrix}$$

Figure 4: Translation Vector

With these values, we can now re-project 3d points into the 2d canvas, and see how far they are from their actual position. With this, we obtain an average re-projection error of *0.0614*px for every point re-projected in the image.

3.2 Light projection system calibration

The goal of this stage is to use a light source to create a shadow line that is projected both on the object and on the area where the object is placed (floor, table, etc). We can detect the points on the shadow line and use them to calculate the equation of the shadow plane that those points belong to. Using this plane equation, together with the perspective projection matrix that was calculated before, we can convert the 2D image points that belong to the shadow line into 3D points in the real world.

We determined the plane's equation by using an object with a known height. To calculate this equation we need at least 3 non-collinear points (meaning that they cannot all be in the same line) that are on the projected shadow line. As we can see in this image [12], the shadow line is going over both the floor (z = 0 cm) and the object, whose height is known (z = 5.56 cm, in this case). Knowing these planes to which the points belong, and also knowing the perspective projection matrix (camera calibration was done previously), we can calculate the 3D coordinates of these points.

With these 3D coordinates, all that is left is to calculate the equation of the shadow plane. That is done using the Scikit-Spatial Python library,

more specifically the Plane.best_fit() method, that takes in a set of 3D points and outputs the coefficients of the best fit plane for all of them [5].

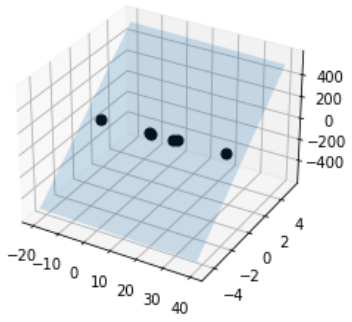


Figure 5: Plane generated by calculating a best fit for the set of input points, using Scikit-Spatial.

This plane equation is then used to update the 2D to 3D conversion matrix, which will allow us to calculate the 3D coordinates of any point in the image intersected by the shadow plane.

3.3 Line detection

The line detection phase is done by applying several operations in sequence to the image, detailed in this section. This image must contain: the object from which to measure coordinates; the area where the object is placed; a horizontal shadow line that goes over both the object and the area.

3.3.1 Edge Detection

The first step is **Edge Detection**, that is used to highlight the edge points of an image. The original colored image is converted to grayscale using the cv.cvtColor() method, to facilitate the next operations. A bilateral filter is applied by using cv.bilateralFilter(), smoothing the image. This helps reducing noise while preserving the edges of objects, improving results in latter steps. Next, the edges of the smoothed image are identified using the Canny edge detector (cv.Canny()). The result is a binary image in which all the edge points are identified as white.

3.3.2 Edge Enhancement

Having identified all the edge points, we perform **Edge Enhancement** to isolate the shadow lines. It may happen that even with the hysteresis thresholding that is performed in the Canny edge detector, the edge points in the resulting image are not fully connected, and do not form lines. To mitigate that problem, we perform a closing morphological operation, using cv.morphologyEx(), to close the gaps between edge points and form lines. This is followed by an erosion operation (cv.erode()) that discard the outer points of those lines and creates finer edges.

3.3.3 Hough Line Transform and Line Bundling

With the shadow lines isolated in the image, we now need to determine their equations. We use the **Hough Transform** technique for this mean, by calling the cv.HoughLinesP() method. Since this step returns overlapping lines, we cluster and merge them, taking into account properties such as the position and slope. All of these operations result in continuous lines in the image, that only stop when a change in height is detected in the shadow line (e.g. the shadow line stops being projected on the floor and starts being projected on an object, or vice-versa). Figures 6 and 7 depict the line bundling process.

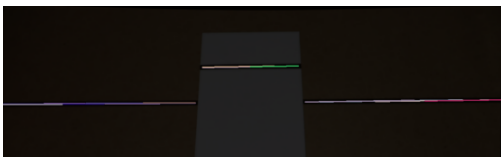


Figure 6: Detected Hough lines before the line bundling process. Some lines are similar and overlap, and could be merged to create a bigger, continuous line.

Figure 7: Hough lines after the line bundling process. The result are single lines that only stop when a change in height occurs in the shadow line.

3.4 Calculation of 3D coordinates

Having the image points of the object that are intercepted by the shadow plane, we can calculate their 3D coordinates in the real world.

The 3D coordinates can be calculated by solving the following system of equations:

$$\begin{cases} C_{11}wx + C_{12}wy + C_{13}wz + C_{14}w = i \\ C_{21}wx + C_{22}wy + C_{23}wz + C_{24}w = j \\ C_{31}wx + C_{32}wy + C_{33}wz + C_{34}w = 1 \\ Awx + Bwy + Cwz + Dw = 0 \end{cases} \quad (1)$$

Where *Cnm* are the factors that compose the perspective projection matrix (calculated in the camera calibration step), *(A, B, C, D)* are the coefficients of the shadow plane (calculated following the method described on section 3.2, *(i, j)* are the 2D coordinates of a point in the image, *(x, y, z)* are the corresponding 3D coordinates in the real world, and w is a multiplication factor.

By solving this system we get the results *(wx, wy, wz, w)*; by dividing the first 3 components by the multiplication factor w we get the 3D coordinates.

4 Considerations

4.1 Efficacy

In this section, the intermediate methods will be analyzed in terms of their efficacy and adequacy.

Regarding the line detection phase, in order to correctly identify the shadow line, a large set of filtering and detection operations could have been performed. After testing several options, like the use of mean and Gaussian filtering with kernels of different sizes, we arrived at the conclusion that the best results were achieved using a bilateral filter for image smoothing, that preserved the edges of the picture, followed by a Canny operation for edge detection. The parameters chosen for these operations were carefully taken into consideration and again, various alternatives were tested with the goal of choosing the best one.

Regarding the shadow plane calibration, there were two main approaches for calculating the plane equation. One was to immobilize the shadow plane as best we could and make it as perpendicular as possible to the area where the object is placed, so we could infer that the shadow plane equation was x = 0. This solution is not very reliable because the plane would need to be at an exact position and angle, and even minor errors and displacements could make a big difference in the final result. The second alternative, which was the one we selected, is to calculate the shadow plane equation by using an object with a known height, and using at least 3 non-collinear points in the shadow line to determine the

plane's equation. This solution is a lot more efficient, versatile and accurate, since it does not depend on the exact position of the light/shadow source. One can confirm this by analysing the practical results obtained using this technique: the errors between the calculated 3D coordinates of points, when compared to their real position, were very small.

4.2 Problems

4.2.1 Large image resolution

The first problem that we encountered was related to the resolutions of the images that we were using. Initially, the Canon EOS 250D shown in Figure 1 was calibrated to take 6000x4000 resolution images. That turned out to be an issue because OpenCV's Canny algorithm picks up considerable noise on images of high resolution, which was decreasing the performance of the subsequent steps. We ended up changing the camera settings to capture 2400x1600 resolution images, that are then converted to a 1200x800 format, which turned out to give better results.

4.2.2 Thick shadow line

For the shadow plane, a broomstick was initially used. However, the projected shadow was too thick, which resulted in the detection of not one shadow line, but two (the edges of the broomstick shadow in the image). This hindered how general the solution was - instead of choosing the detected edges for the next phases, we had to manually choose only the edge points that represented either the top line or the bottom line of the shadow.

To solve this issue, the group settled for a thin plastic pole, that produced a thinner shadow. This allowed for a single shadow line to be extracted, using morphological transformations (explained in further detail in section 3.3). The only small downside is that the pole bends very slightly, introducing some errors when calculating the 3D coordinates.

4.2.3 Noise in Canny edge detector

Even when lowering the resolution of the images, it sometimes occurred that the Canny edge detector was identifying a lot of edge points that constituted noise, as they were not part of the projected shadow and were not relevant for the problem. To ignore and filter out those points, we performed a Hough Line Transform stage after the Canny edge detection stage, which allowed to only select straight lines that belonged to the shadow.

4.2.4 Overlapping Hough lines

After the Hough lines phase, we noticed that, for example, in an image were there is only one object, even though we should identify 3 separate lines for the shadow (shadow on the floor to the left of the object, shadow on top of the object, shadow on the floor to the right of the object), the number of detected Hough lines was greater than that. Since some lines were very close and overlapping each other, we decided to use a Line Bundler algorithm (explained in 3.3.3), that detected overlapping lines and merged them. With this we were able to obtain perfect results, identifying each shadow line only once.

4.2.5 Shadow line sharpness

During image acquisition, we noticed that the rotation of the smartphone light in relation to the camera played a big role in terms of the sharpness of the resulting shadow. When the smartphone faced the camera, the shadow was foggy, given that the light cone was wider. Therefore, it was necessary to ensure that the camera was always at a 90º degree angle, resulting in a thinner light cone and, therefore, sharper shadows that are easier to detect.

5 Results

5.1 Intrinsic parameters

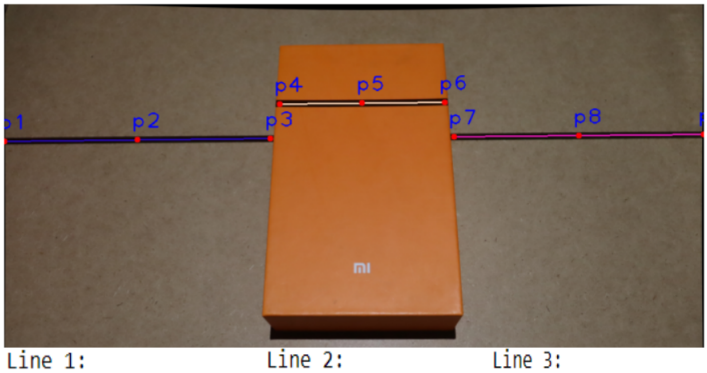
When calculating the intrinsic parameters, a re-projection error of 0.0614px was obtained. Since the images are 1200x800px, totalling 960000px, this gives us a relative error of 6.4e−6 pixels. We can conclude that this is a negligible error and that the chessboard pattern images used for calibration cover a sufficient amount of angles and orientations relative to the camera to make the method reliable.

5.2 Light projection system calibration

After calculating the light projection plane, 6 points were used to calculate the real world coordinates, resulting in an average error of 0.04px. This negligible error shows that the chosen approach is reliable.

5.3 Calculation of 3D coordinates

The main objective of this project is to accurately estimate object dimensions. Taking the orange box as the object to analyze, the results are the following:



Line 1:	Line 2:	Line 3:
p1: (-13.7,0.24,-0.04)	p4: (6.01,0.3,4.06)	p7: (17.31,0.33,0.08)
p2: (-4.41,0.26,0.05)	p5: (11.1,0.31,4.11)	p8: (25.75,0.35,0.09)
p3: (4.76,0.29,0.05)	p6: (16.2,0.33,4.09)	p9: (34.13,0.38,0.11)

Figure 8: Point estimation in the orange box.

The shadow points on top of the object, P4(6.01, 0.3, 4.06), P5(11.1, 0.31, 4.11) and P6(16.2, 0.33, 4.09), allow us to calculate an average value of the box's height: **4.08 cm**. Given that the real world value is roughly 4.1 cm, the error is equal to **0.49%**.

It's also possible to calculate the width of the box through the sum of the distances between P4-P5 and P5-P6. This yields **10.19 cm**, which, compared to the real world value of 10.9 cm, results in an error of **6.51%**.

These results prove our method to be very reliable, with low error rates, especially when estimating the object height. More results for various objects can be viewed in the annex, as well as their respective height graphs. The jupyter notebook in the submission can also be ran, by changing only the arguments in the "image" section, to view a full analysis of other objects.

6 Conclusion

We can safely affirm that the main goal of the project has been fulfilled with success: the system is capable of, given a 2D image of an object that is illuminated by a shadow line, calculate the 3D coordinates of points on that line with fairly good accuracy. All stages of the process were successfully implemented, from the calibration of the camera using its intrinsic and extrinsic parameters, to light projection system calibration, and calculation of the 3D coordinates. The various intermediate operations were carefully tested and chosen, as well as their parameters and values, in order to generate the best results. The overall solution can also be considered efficient and reliable, given the fact that the shadow plane equation is calculated given the various points that belong to the shadow line. In general, we can conclude that both the general and specific aims of the project were achieved.´

References

[1] OpenCV *Camera Calibration*.
https://opencv-pythontutroals.readthedocs.io/en/latest/py/tutorials/py/calib3d/py/calibration/py/calibration.html

[2] OpenCV *Pose Estimation*.
https://docs.opencv.org/4.5.1/d7/d53/tutorial/py/pose.html

[3] Oleg Dats *Hough Lines Bundling*
https://stackoverflow.com/questions/45531074/how-to-merge-lines-after-houghlinesp

[4] D. Kim, S. Lee, H. Kim, S. Lee *Wide-angle laser structured light system calibration with a planar object*
International Conference on Control, Automation and Systems 2010, Oct. 27-30, 2010, Gyeonggi-do, Korea, pp. 1879-1882

[5] H. Luo, J. Xu, N.H. Binh , S. Liu , C. Zhang, K. Chen *A simple calibration procedure for structured light system*
Optics and Lasers in Engineering, vol. 57, June 2014, pp. 6-12

7 Annex

7.1 Results

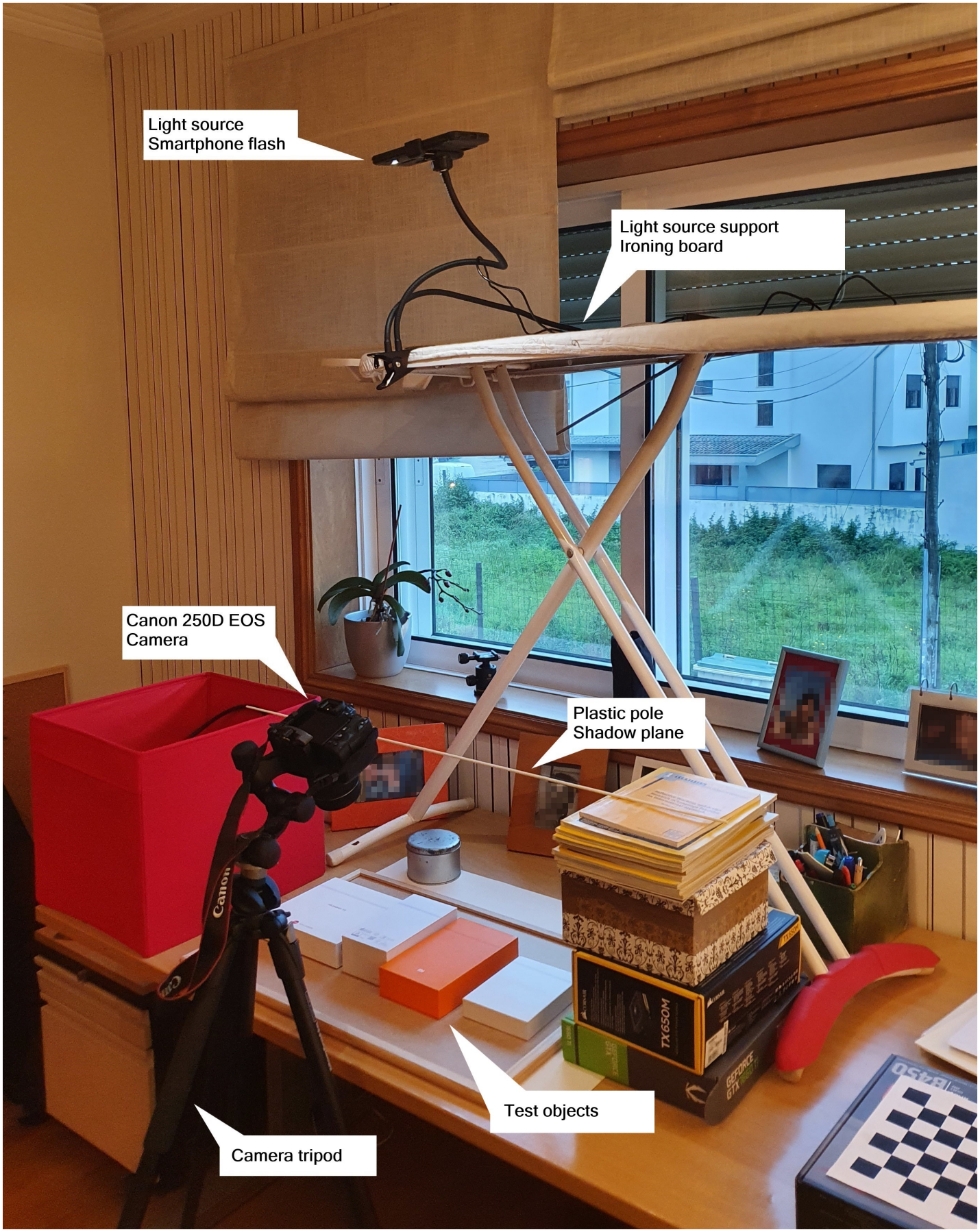


Figure 9: Image acquisition setup and its components.

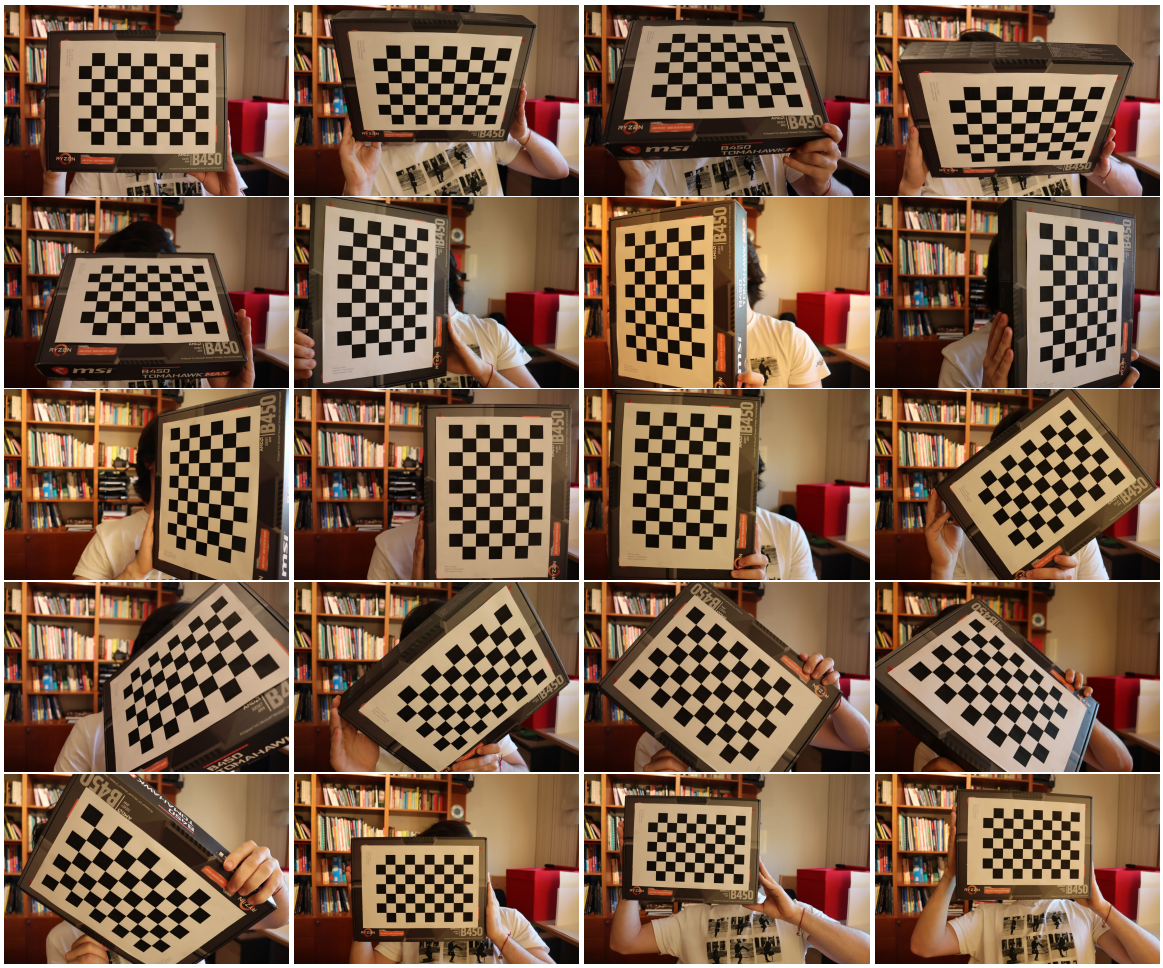


Figure 10: Images used for intrinsic calibration

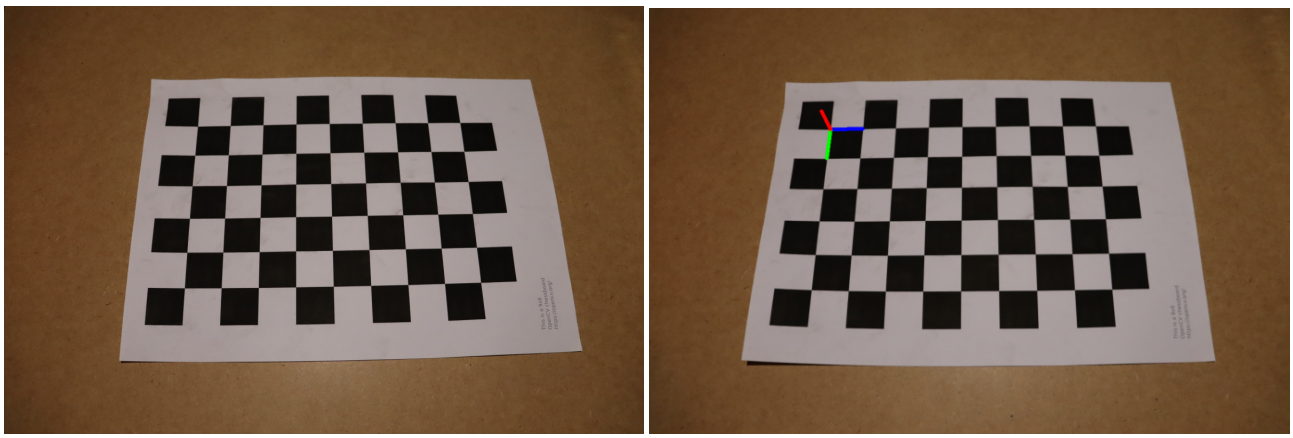


Figure 11: Image used for extrinsic calibration:a) Without Axisb) With Axis

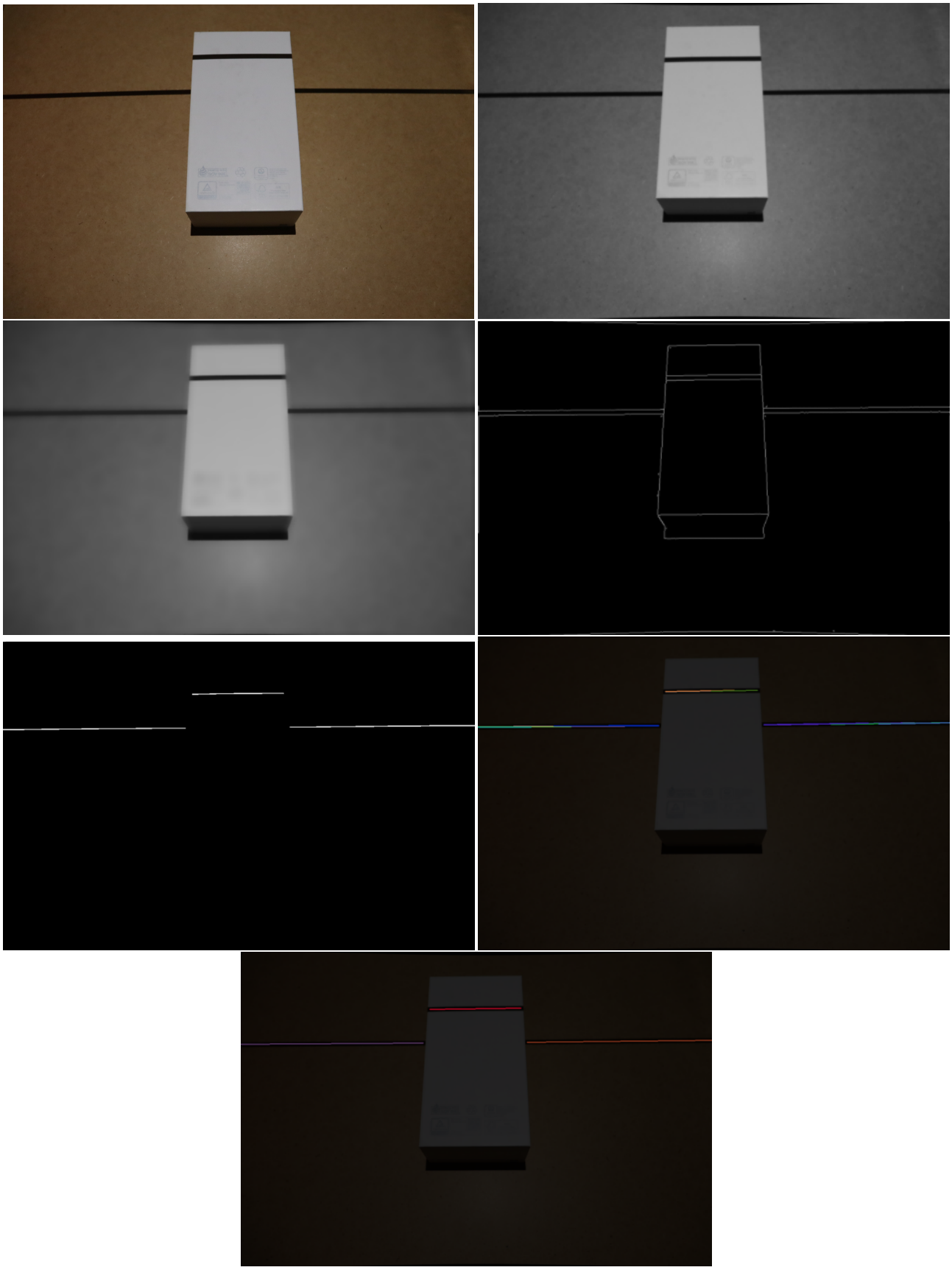


Figure 12: Line detection process:
a) Original
b) Grayscale
c) Bilateral Filter
d) Canny filter
e) Morphological Closing
f) Hough Lines
g) Line Bundling

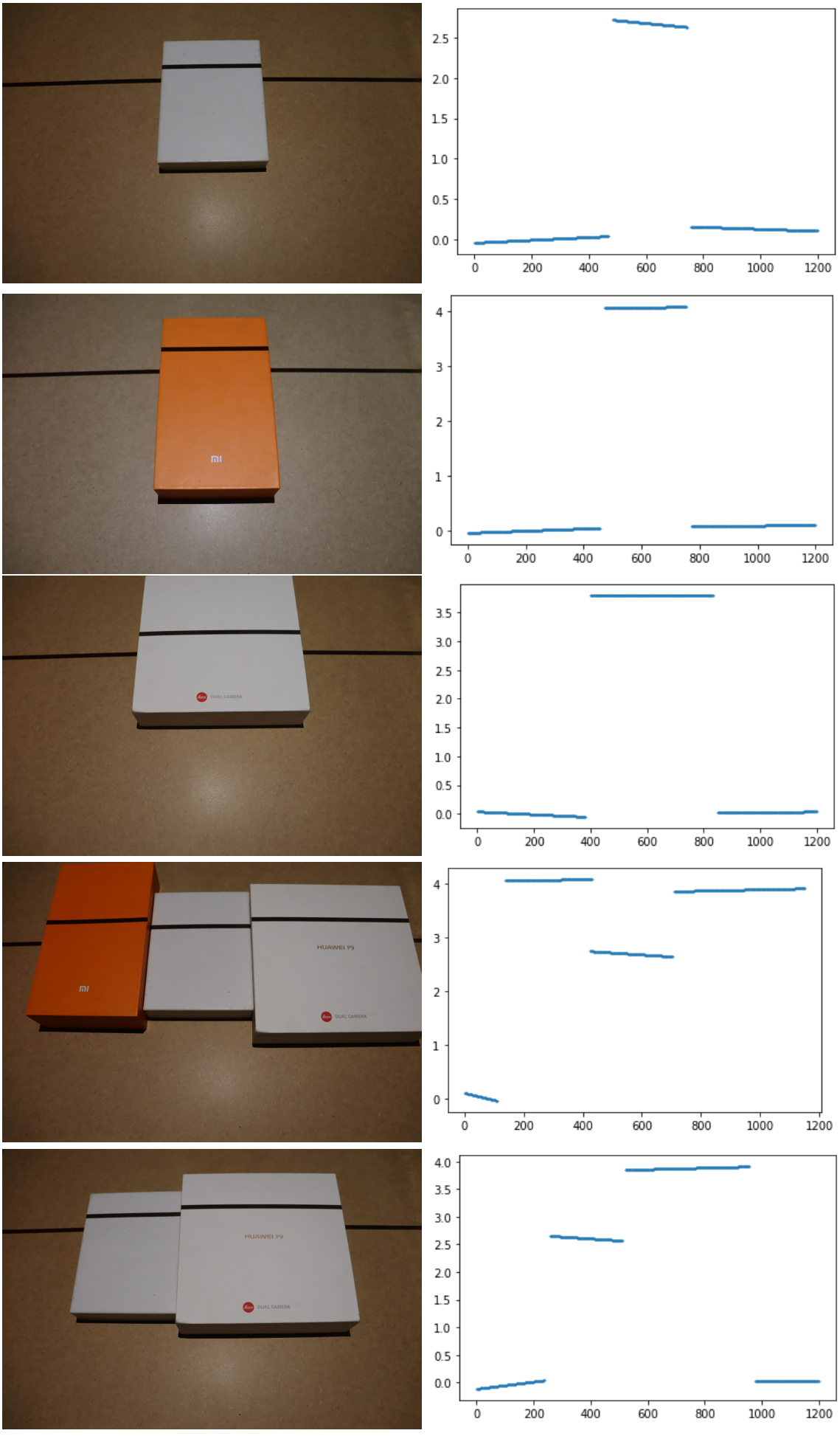


Figure 13: Heightmaps of different images

7.2 Code

```
import numpy as np
import cv2 as cv
import glob
import math
import random
from matplotlib import pyplot as plt
from skspatial.objects import Plane, Points
from skspatial.plotting import plot_3d
from sympy import Matrix, init_printing

### HELPERS - Drawing

# plot images in a grid
def plot_grid(images, rows, columns, figsize=30):
    display_columns = 4
    fig, axes = plt.subplots(rows, columns, figsize=(figsize,figsize))

    for img, ax in zip(images, axes.ravel()):
        ax.imshow(cv.cvtColor(img, cv.COLOR_BGR2RGB))
        ax.axis('off')
    fig.tight_layout()

# plot images in a row
def plot_row(images, figsize=30):
    fig, axes = plt.subplots(1, len(images), figsize=(figsize,figsize))

    for img, ax in zip(images, axes.ravel()):
        ax.imshow(cv.cvtColor(img, cv.COLOR_BGR2RGB))
        ax.axis('off')
    fig.tight_layout()

# plot a single image
def plot_img(image, figsize=10):
    fig, ax = plt.subplots(1, 1, figsize=(figsize,figsize))
    ax.imshow(cv.cvtColor(image, cv.COLOR_BGR2RGB))
    ax.axis('off')
    fig.tight_layout()

# draw axis in a image
def draw_axis(img, corners, imgpts):
    imgCopy = np.copy(img)
    corner = tuple(corners[0].ravel().astype(int))
    imgCopy = cv.line(imgCopy, corner, tuple(imgpts[0].ravel().astype(int)), (255,0,0), 5)
    imgCopy = cv.line(imgCopy, corner, tuple(imgpts[1].ravel().astype(int)), (0,255,0), 5)
    imgCopy = cv.line(imgCopy, corner, tuple(imgpts[2].ravel().astype(int)), (0,0,255), 5)
    return imgCopy

# draw input lines on an img
def draw_lines(img, lines, no_bg=True):
    imgCopy = np.copy(img)
    if no_bg:
        imgCopy = imgCopy * 0
    for line in lines:
        x1, y1, x2, y2 = line[0]
        cv.line(imgCopy, (x1, y1), (x2, y2), (random.random() * 255,random.random() * 255,random.random() * 255),2)
    return imgCopy

### LINE DETECTION

# undistort an image
def undistort_img(img, mtx, dist):
    h, w = img.shape[:2]
    newcameramtx, roi = cv.getOptimalNewCameraMatrix(mtx, dist, (w,h), 1, (w,h))
    newimg = cv.undistort(img, mtx, dist, None, newcameramtx)
    return newimg

# detect edges from an image using Canny
def edge_detection(img):
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY) # convert to gray color
    filtered = cv.bilateralFilter(gray, 30, 50, 50) # blur
    edges = cv.Canny(filtered, 20, 30) # edge detection
    return [gray, filtered, edges]

# enhance shadow edges from an img
def edge_enhancement(img):
    kernel = np.ones((3,3), np.uint8)
    closing = cv.morphologyEx(img, cv.MORPH_CLOSE, kernel, iterations=7) # closing to close small features
    erosion = cv.erode(closing, kernel, iterations=5) # erode to have a finer edge
    return erosion

# detect lines from an image using HoughLinesP
def hough_line_detection(img):
    rho = 1  # distance resolution in pixels of the Hough grid
    theta = np.pi / 180  # angular resolution in radians of the Hough grid
    threshold = 100  # minimum number of votes (intersections in Hough grid cell)
    min_line_length = 100  # minimum number of pixels making up a line
    max_line_gap = 50  # maximum gap in pixels between connectable line segments
    upper_layer = 100 # lines above this layer will not be considered
    lower_layer = 1100 # lines below this layer will not be considered

    # "lines" is an array containing endpoints of detected line segments
    lines = cv.HoughLinesP(img, rho, theta, threshold, np.array([]), min_line_length, max_line_gap)

    # filter lines from top and bottom image edges, due to setup
    new_lines = []
    for line in lines:
        for x1,y1,x2,y2 in line:
            # image is 1200x800, ignore undistort-induced edges
            if not y1 > lower_layer and not y1 < upper_layer:
                new_lines.append(line)
    return new_lines

#-------------------------#
#   HOUGH LINES BUNDLER   #
#-------------------------#
class HoughBundler:
    # Based on: https://stackoverflow.com/questions/45531074/how-to-merge-lines-after-houghlinesp
    # Class that clusters similar/overlapped Hough Lines and merges them

    def get_slope(self, line):
        # https://en.wikipedia.org/wiki/Atan2
        slope = math.atan2(abs((line[0] - line[2])), abs((line[1] - line[3])))
        return math.degrees(slope)

    def is_different_line(self, line_new, groups, min_distance_to_merge, min_angle_to_merge):
        for group in groups:
            for line_old in group:
                if self.get_distance_between_lines(line_old, line_new) < min_distance_to_merge:
                    slope_new = self.get_slope(line_new)
                    slope_old = self.get_slope(line_old)
                    if abs(slope_new - slope_old) < min_angle_to_merge:
                        group.append(line_new)
                        return False
        return True

    def point_to_line_dist(self, point, line):
        # http://local.wasp.uwa.edu.au/~pbourke/geometry/pointline/source.vba
        px, py = point
        x1, y1, x2, y2 = line

        def get_line_length(x1, y1, x2, y2):
            line_length = math.sqrt(math.pow((x2 - x1), 2) + math.pow((y2 - y1), 2))
            return line_length

        line_length = get_line_length(x1, y1, x2, y2)
        if line_length < 0.00000001:
            point_to_line_dist = 9999
            return point_to_line_dist

        u1 = (((px - x1) * (x2 - x1)) + ((py - y1) * (y2 - y1)))
        u = u1 / (line_length * line_length)

        if (u < 0.00001) or (u > 1):
            # closest point does not fall within the line segment, take the shorter distance to an endpoint
            ix = get_line_length(px, py, x1, y1)
            iy = get_line_length(px, py, x2, y2)
            if ix > iy:
                point_to_line_dist = iy
            else:
                point_to_line_dist = ix
        else:
            # intersecting point is on the line, use the formula
            ix = x1 + u * (x2 - x1)
            iy = y1 + u * (y2 - y1)
            point_to_line_dist = get_line_length(px, py, ix, iy)

```

```
        return point_to_line_dist

    def get_distance_between_lines(self, line1, line2):
        dist1 = self.point_to_line_dist(line1[:2], line2)
        dist2 = self.point_to_line_dist(line1[2:], line2)
        dist3 = self.point_to_line_dist(line2[:2], line1)
        dist4 = self.point_to_line_dist(line2[2:], line1)

        return min(dist1, dist2, dist3, dist4)

    def merge_lines_pipeline(self, lines):
        groups = []  # all lines groups are here

        # Parameters to play with
        min_distance_to_merge = 10
        min_angle_to_merge = 30

        def getFirstPointX(line):
            return line[0]
        lines = sorted(lines, key=getFirstPointX)

        # first line will create a new group every time
        groups.append([lines[0]])

        # if line is different from existing groups,
     create a new group
        for line_new in lines[1:]:
            if self.is_different_line(line_new, groups,
     min_distance_to_merge, min_angle_to_merge):
                groups.append([line_new])

        return groups

    def merge_lines_segments(self, lines):
        # Sort lines cluster and return first and last
     coordinates
        slope = self.get_slope(lines[0])

        # special case
        if(len(lines) == 1):
            return [lines[0][:2], lines[0][2:]]

        # [[1,2,3,4],[]] to [[1,2],[3,4],[],[]]
        points = []
        for line in lines:
            points.append(line[:2])
            points.append(line[2:])
        # if vertical
        if not 45 < slope < 135:
            #sort by y
            points = sorted(points, key=lambda point:
     point[1])
        else:
            #sort by x
            points = sorted(points, key=lambda point:
     point[0])

        # return first and last point in sorted group
        # [[x,y],[x,y]]
        return [points[0], points[-1]]


    # convert bundler line format to the hough lines
     format
    def convert_line_format(self, bundled_lines):
        newlines = []
        for line in bundled_lines:
            x2, y2, x1, y1 = line[0][0], line[0][1], line
     [1][0], line[1][1]
            if x2 < x1:
                x1, y1, x2, y2 = x2, y2, x1, y1
            newlines.append(np.array([[x1,y1,x2,y2]],
     dtype=np.int32))

        def getFirstPointX(line):
            return line[0][0]
        newlines = sorted(newlines, key=getFirstPointX)

        return newlines

    def process_lines(self, lines):
        lines_x = []
        lines_y = []
        # for every line of cv.HoughLinesP()
        for line_i in [l[0] for l in lines]:
                slope = self.get_slope(line_i)
                # if vertical
                if 45 < slope < 135:
                    lines_y.append(line_i)
                else:
                    lines_x.append(line_i)

        lines_y = sorted(lines_y, key=lambda line: line
     [1])
        lines_x = sorted(lines_x, key=lambda line: line
     [0])
        merged_lines_all = []

        # for each cluster in vertical and horizontal
     lines leave only one line
        for i in [lines_x, lines_y]:
                if len(i) > 0:
                    groups = self.merge_lines_pipeline(i)
                    merged_lines = []
                    for group in groups:
                        merged_lines.append(self.
     merge_lines_segments(group))

                    merged_lines_all.extend(merged_lines)

        return self.convert_line_format(merged_lines_all)

### 3D/2D CONVERSIONS

# 3D -> 2D
def world_to_image(projMtx, x, y, z):
    res = np.dot(projMtx, [[x],[y],[z],[1]])
    return res/res[2]

# 2D -> 3D
def image_to_world(matrix, i, j):
    res = np.linalg.solve(matrix, [[i], [j], [1], [0]])
    [[x], [y], [z], [_]] = res/res[3]
    return [x, y, z]

### IMAGES

# Images used for the camera Intrinsic calibration
intrinsic_calib_imgs = glob.glob('images/calib/*.jpg')

# Image used for the camera Extrinsic calibration
extrinsic_calib_img = cv.imread('images/exp2/chess.jpg')

# Image used to calibrate the shadow plane
OBJ_HEIGHT = 5.56 # known height of the object in the
     image
shadow_plane_calib_img = cv.imread('images/exp2/calib.jpg'
     )

# Image with object to calculate 3D coordinates
original_image = cv.imread('images/exp2/test5.jpg')

### PREPARATION

# pattern size
pattern = ((9,6))

# termination criteria
criteria = (cv.TERM_CRITERIA_EPS + cv.
     TERM_CRITERIA_MAX_ITER, 30, 0.001)

# prepare object points, like (0,0,0), (1,0,0), (2,0,0)
     ....,(6,5,0)
objp = np.zeros((pattern[0] * pattern[1], 3), np.float32)
objp[:,:2] = np.mgrid[0:pattern[0],0:pattern[1]].T.reshape
     (-1,2) * 2.5

# Arrays to store object points and image points from all
     the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

### CAMERA CALIBRATION

display_images = []
for i, fname in enumerate(intrinsic_calib_imgs): # go
     through all images
    img = cv.imread(fname)
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY) # convert
     to graycolor

    # Find the chess board corners
    ret, corners = cv.findChessboardCorners(gray, pattern,
      None)

    # If found, add object points, image points (after
     refining them)
    if ret == True:
        objpoints.append(objp)

        corners2 = cv.cornerSubPix(gray,corners, (11,11),
     (-1,-1), criteria)
        imgpoints.append(corners)

        # Draw and display the corners
        cv.drawChessboardCorners(img, pattern, corners2,
     ret)
        display_images.append(img)

# calibrate the camera
ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera(
     objpoints, imgpoints, gray.shape[::-1], None, None)
print("Intrinsic Matrix:")
display(Matrix(mtx))
print("Distortion factors:")
display(Matrix(dist))
```

```

# plot images
display_columns = 4
plot_grid(display_images, math.ceil(len(
    intrinsic_calib_imgs)/display_columns),
    display_columns)

### RESET VARIABLES

# pattern size
pattern = ((9,6))

# termination criteria
criteria = (cv.TERM_CRITERIA_EPS + cv.
    TERM_CRITERIA_MAX_ITER, 30, 0.001)
objp = np.zeros((pattern[0]*pattern[1],3), np.float32)
objp[:,:2] = np.mgrid[0:pattern[0],0:pattern[1]].T.reshape
    (-1,2) * 2.5
axis = np.float32([[3,0,0], [0,3,0], [0,0,-3]]).reshape
    (-1,3)

### CALCULATE CAMERA POSE

grayextrinsicCalibImage = cv.cvtColor(extrinsic_calib_img,
     cv.COLOR_BGR2GRAY) # convert to graycolor
ret, corners = cv.findChessboardCorners(
    grayextrinsicCalibImage, pattern, None)
if ret != True:
    print("Failed to calibrate extrinsic parameters")
    exit()
corners2 = cv.cornerSubPix(grayextrinsicCalibImage,
    corners, (11,11), (-1,-1), criteria)
ret, rvecs, tvecs, _ = cv.solvePnPRansac(objp, corners2,
    mtx, dist)

rmtx, _ = cv.Rodrigues(rvecs)

print("Rotation Matrix:")
display(Matrix(cv.Rodrigues(rvecs)[0]))
print("Translation Vector:")
display(Matrix(tvecs))

### REPROJECTION ERROR

imgpoints2, _ = cv.projectPoints(objp, rvecs, tvecs, mtx,
    dist)
error = cv.norm(corners2, imgpoints2, cv.NORM_L2)/len(
    imgpoints2)
print("Reprojection Error: {}px".format(error))

### DRAW IMAGE WITH AXIS

imgpts, jac = cv.projectPoints(axis, rvecs, tvecs, mtx,
    dist)
axisimg = draw_axis(extrinsic_calib_img, corners2, imgpts)

plot_img(axisimg, figsize=10)

### SHADOW PLANE CALIBRATION

# Undistort image
img = undistort_img(shadow_plane_calib_img, mtx, dist)
plot_img(img, figsize=10)

# Edge detection
gray, filtered, edges = edge_detection(img) # detect edges
plot_row([gray, filtered, edges])

# Edge enhancement
enhanced = edge_enhancement(edges) # enhance shadow edges
plot_img(enhanced, figsize=10)

### HOUGH LINES DETECTION
# line detection
lines = hough_line_detection(enhanced)

# draw
imgLines = draw_lines(img, lines)
imgLines = cv.addWeighted(img, 0.2, imgLines, 0.8, 0)
plot_img(imgLines, figsize=8)
# output
print(f'Number of lines: {len(lines)}')
for line in lines:
    print(line)

### BUNDLE LINES
# merge lines
bundled_lines = HoughBundler().process_lines(lines)

# draw
imgBundled = draw_lines(img, bundled_lines)
imgBundled = cv.addWeighted(img, 0.2, imgBundled, 0.8, 0)
plot_img(imgBundled, figsize=10)

# output
print(f'Number of lines: {len(bundled_lines)}')
for line in bundled_lines:
    print(line)

### CALCULATE PROJECTION MATRIX
# calculate projection matrix (projMtx) -> 3D to 2D
rmtx, _ = cv.Rodrigues(rvecs)
rotTransMtx = np.zeros((3,4))
rotTransMtx[:,:-1] = rmtx
rotTransMtx[:,-1:] = tvecs

projMtx = np.dot(mtx, rotTransMtx)

print("Projection Matrix:")
display(Matrix(projMtx))

### CALCULATE IMAGE POINTS
# initialize list to hold image coordinates of points in
    the shadow line
img_points_shadow = []
# initialize list to hold world coordinates of points in
    the shadow line
obj_points_shadow = []

# lines on the left and right are on the plane z = 0
inverseMtxFloor = np.zeros((4,4))
inverseMtxFloor[:-1,:] = projMtx
inverseMtxFloor[-1:,:] = [[0, 0, 1, 0]]

print("Floor Projection Matrix:")
display(Matrix(inverseMtxFloor))

# right line
[[i1, j1, i2, j2]] = bundled_lines[2]
img_points_shadow.append([i1, j1])
obj_points_shadow.append(image_to_world(inverseMtxFloor,
    i1, j1))
img_points_shadow.append([i2, j2])
obj_points_shadow.append(image_to_world(inverseMtxFloor,
    i2, j2))

# left line
[[i1, j1, i2, j2]] = bundled_lines[0]
img_points_shadow.append([i1, j1])
obj_points_shadow.append(image_to_world(inverseMtxFloor,
    i1, j1))
img_points_shadow.append([i2, j2])
obj_points_shadow.append(image_to_world(inverseMtxFloor,
    i2, j2))

# the center line is on the plane z = OBJ_HEIGHT
inverseMtxObj = np.zeros((4,4))
inverseMtxObj[:-1,:] = projMtx
inverseMtxObj[-1:,:] = [[0, 0, 1, OBJ_HEIGHT]]

print("Box Projection Matrix:")
display(Matrix(inverseMtxObj))

[[i1, j1, i2, j2]] = bundled_lines[1]
img_points_shadow.append([i1, j1])
obj_points_shadow.append(image_to_world(inverseMtxObj, i1,
     j1))
img_points_shadow.append([i2, j2])
obj_points_shadow.append(image_to_world(inverseMtxObj, i2,
    j2))

# output list of non-coplanar points that are in the
    shadow line
print(obj_points_shadow)

### SHADOW PLANE EQUATION
# calculate shadow plane equation
plane_points = Points(obj_points_shadow)
plane = Plane.best_fit(plane_points)

# plot the plane and the points
plot_3d(
    plane_points.plotter(c='k', s=50, depthshade=False),
    plane.plotter(alpha=0.2, lims_x=(-30, 30), lims_y
    =(-5, 5)),
)

# calculate final matrix for 2D -> 3D conversion, now with
     the shadow plane's equations
final_shadow_plane = plane.cartesian()
inverseMtxFinal = np.zeros((4,4))
inverseMtxFinal[:-1,:] = projMtx
inverseMtxFinal[-1:,:] = [final_shadow_plane]

### CALCULATE ERROR
# test if the final matrix is able to correctly calculate
    the 3D coordinates of points on the shadow line

# calculate distance between the expected point and the
    calculated point
def calc_error(expected, calculated):
    return np.sqrt(np.sum((np.array(expected) - np.array(
    calculated)) ** 2, axis=0))

# for the previously collected points, calculate world
    coordinates and check the error
```

```
for idx in range(len(img_points_shadow)):
    i, j = img_points_shadow[idx]
    expected = obj_points_shadow[idx]
    calculated = image_to_world(inverseMtxFinal, i, j)

    print(f'Image: ({i}, {j})')
    print(f'Expected: ({expected[0]}, {expected[1]}, {expected[2]})')
    print(f'Calculated: ({calculated[0]}, {calculated[1]}, {calculated[2]})')
    print(f'Error: {calc_error(expected, calculated)}')
    print('--')

### NEW IMAGE - DETERMINING OBJECT HEIGHT
# Undistort
img = undistort_img(original_image, mtx, dist)
plot_img(img, figsize=10)

# Edge detection
gray, filtered, edges = edge_detection(img) # detect edges
plot_row([gray, filtered, edges])

# Edge enhancement
enhanced = edge_enhancement(edges) # enhance shadow edges
plot_img(enhanced, figsize=10)

### HOUGH LINES DETECTION
# line detection
lines = hough_line_detection(enhanced)

# draw
imgLines = draw_lines(img, lines)
imgLines = cv.addWeighted(img, 0.2, imgLines, 0.8, 0)
plot_img(imgLines, figsize=10)

# output
print(f'Number of lines: {len(lines)}')
for line in lines:
    print(line)

### BUNDLE LINES
# merge lines
bundled_lines = HoughBundler().process_lines(lines)

# draw
imgBundled = draw_lines(img, bundled_lines)
imgBundled = cv.addWeighted(img, 0.2, imgBundled, 0.8, 0)
plot_img(imgBundled, figsize=10)

# output
print(f'Number of lines: {len(bundled_lines)}')
for line in bundled_lines:
    print(line)

### CALCULATE LINE HEIGHTS AND PLOT GRAPH
# given a line, return a list with all points from x1 to x2 separated by 1 unit
def get_line_points(line):
    [[x1, y1, x2, y2]] = line

    # line equation
    slope = (y2 - y1) / (x2 - x1)
    b = y1 - x1 * slope

    # calculate coordinates for all points in the line
    pnts = []
    for i in range(x1, x2 + 1):
        j = i * slope + b
        pnts.append([i, j])
    return pnts


# calculate height of shadow line points
pixel_hor = []
pnt_height = []
for line in bundled_lines:
    for i, j in get_line_points(line):
        [_, _, h] = image_to_world(inverseMtxFinal, i, j)
        pixel_hor.append(i)
        # negative of the calculated height, because in the coordinate system the height is negative
        pnt_height.append(-h)

# display height of all pixels on the line with scatterplot
plt.scatter(pixel_hor, pnt_height, s=0.5)
plt.show()

# draw the detected lines in the original image
imgLines = draw_lines(img, bundled_lines, no_bg=False)

# configuration of the text to appear in the image
font = cv.FONT_HERSHEY_PLAIN
fontScale = 1.4
text_color = (255, 0, 0)
text_thickness = 2
text_offset = (-100, 30)

# configuration of the points to appear in the image
radius = 5
point_thickness = cv.FILLED # negative to fill
point_color = (0, 0, 255)

# function to draw a point in the image
def draw_point(img, i, j, img_caption, print_caption, offset=text_offset):
    [x, y, z] = image_to_world(inverseMtxFinal, i, j)
    z = -z # invert z coordinate as explained, in order to have positive value
    cv.putText(img, f"{img_caption}", (i+offset[0], j+offset[1]), font, fontScale, text_color, text_thickness)
    print(f"{print_caption}({round(x,2)},{round(y,2)},{round(z,2)})")
    cv.circle(img, (i,j), radius, point_color, point_thickness)

# function to draw 3 points of a line, endpoints and middle
def draw_line_points(img, line, start_point_id, line_id, offset=text_offset):
    start_point_id += 1

    [[i1, j1, i2, j2]] = line
    print(f"Line {line_id}: ")
    draw_point(img, i1, j1, f"p{start_point_id}",f"p{start_point_id}: ", offset)
    draw_point(img, int((i1+i2)/2), int((j1+j2)/2), f"p{start_point_id+1}", f"p{start_point_id+1}: ", offset)
    draw_point(img, i2, j2, f"p{start_point_id+2}", f"p{start_point_id+2}: ", offset)
    print()

# draw
for i in range(len(bundled_lines)):
    draw_line_points(imgLines, bundled_lines[i], i*3, i+1, (-10, -20))

plot_img(imgLines, figsize=10)

### SAMPLE LINE POINTS WITH 3D COORDINATES
# draw the detected lines in the original image
imgLines = draw_lines(img, bundled_lines, no_bg=False)

# configuration of the text to appear in the image
font = cv.FONT_HERSHEY_PLAIN
fontScale = 1.4
text_color = (255, 0, 0)
text_thickness = 2
text_offset = (-100, 30)

# configuration of the points to appear in the image
radius = 5
point_thickness = cv.FILLED # negative to fill
point_color = (0, 0, 255)

# function to draw a point in the image
def draw_point(img, i, j, img_caption, print_caption, offset=text_offset):
    [x, y, z] = image_to_world(inverseMtxFinal, i, j)
    z = -z # invert z coordinate as explained, in order to have positive value
    cv.putText(img, f"{img_caption}", (i+offset[0], j+offset[1]), font, fontScale, text_color, text_thickness)
    print(f"{print_caption}({round(x,2)},{round(y,2)},{round(z,2)})")
    cv.circle(img, (i,j), radius, point_color, point_thickness)

# function to draw 3 points of a line, endpoints and middle
def draw_line_points(img, line, start_point_id, line_id, offset=text_offset):
    start_point_id += 1

    [[i1, j1, i2, j2]] = line
    print(f"Line {line_id}: ")
    draw_point(img, i1, j1, f"p{start_point_id}",f"p{start_point_id}: ", offset)
    draw_point(img, int((i1+i2)/2), int((j1+j2)/2), f"p{start_point_id+1}", f"p{start_point_id+1}: ", offset)
    draw_point(img, i2, j2, f"p{start_point_id+2}", f"p{start_point_id+2}: ", offset)
    print()

# draw
for i in range(len(bundled_lines)):
    draw_line_points(imgLines, bundled_lines[i], i*3, i+1, (-10, -20))

plot_img(imgLines, figsize=10)
```