3D Data Acquisition using a Structured Light Technique

Faculdade de Engenharia da Universidade do Porto
Porto, Portugal

Abstract

3D scanning is the process of analyzing a real-world object or environment to collect data on its shape and appearance that can be used to construct digital 3D models. A 3D scanner can be based on many different technologies and techniques, each with its own limitations, advantages, and costs. One of those techniques is a 3D scanner based on a Structured-light system.

Structured-light 3D scanners project a pattern of light on the subject and look at the deformation of the pattern on the subject. The pattern is projected onto the subject using either an LCD projector or other stable light source. A camera, offset slightly from the pattern projector, looks at the shape of the pattern and calculates the distance of every point in the field of view.

An implementation of a 3D data acquisition system based on structured light, using a single sheet of light/shadow is presented and described in this report. Practical experimental results are also provided to analyze the efficacy of the proposed methods.

1 Introduction

Computer vision has become very important when it comes to the acquisition of 3D data from surfaces or/and creating 3D models of objects. Structured light 3D scanning has become one of the most popular methods of 3D scanning because of how easily it can be done relative to the quality of results achieved. Because of this, structured light techniques for 3D scanning have several applications that span across several industries. Some of its more notable applications include:

- Reverse engineering of objects to produce CAD data;
- Volume measurement of intricate engineering parts;
- Documentation of cultural and historical artifacts;
- Motion and environment capture for augmented reality games;
- Body measurements for fashion retailing.

This project aims to use the theoretical knowledge about computer vision techniques such as image processing techniques, camera calibration, and structured light techniques to calculate real world coordinates (3D coordinates) based on 2D coordinates. It focuses on the acquisition of an image of a 3D object, with know dimensions, illuminated by a light/shadow plane and the calculation of the 3D coordinates of the points lit by that plane .

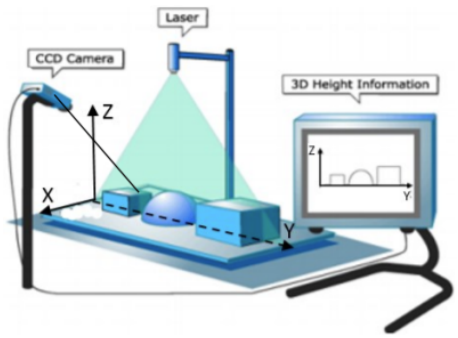


Figure 1: Structured-light 3D data acquisition system

2 Description of the Global Proposed Solution

The implemented solution for this problem involves 6 main stages. These seven stages are camera calibration for the intrinsic/extrinsic parameters, acquisition setup, light project system calibration, image acquisition, line detection and calculation of 3D coordinates.

2.1 Camera calibration - intrinsic/extrinsic parameters

The first step to calibrate our setup was to find the intrinsic and extrinsic parameters of the camera.

The intrinsic parameters are specific to a camera and include information like the focal length (fx,fy) and optical centers (cx,cy). These parameters are part of the camera matrix. The camera matrix is unique to a specific camera so, once calculated, it can be reused on other images taken by the same camera.

$$camera\ matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 2: Camera matrix

The extrinsic parameters correspond to rotation and translation vectors, which translates the coordinates of a 3D point to a coordinate system.

To find all these parameters, we had to provide some sample images of a well-defined pattern. For that, we used images from different perspectives of a chessboard with 7x7 internal corners. After this, we needed to find some specific points in it. These points are the coordinates in the image and in the real world (image points and object points) that helped us to determine the camera matrix and the distortion coefficients.

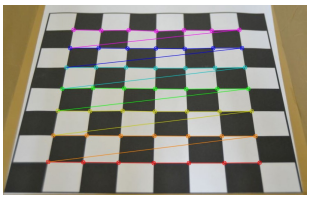


Figure 3: Chess Board Corners

Subsequent to having found the camera matrix and the distortion coefficients, we calculated the pose of the camera with a new chessboard image (placed where we took the photos of the object lit by shadow plane) to find out the rotation and perspective projection matrices. These matrices will be useful for when we get into the calculation of 3D points.

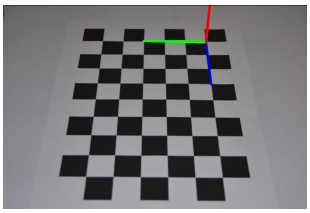


Figure 4: Chess Board Axis

2.2 Acquisition setup

After the process of acquiring the intrinsic and extrinsic parameters of the camera, the setup for the experience was placed as illustrated in Figure 1.

2.3 Light projection system calibration

Two options were proposed for this step:

- Make the shadow plane perpendicular to the table on which the object is placed (plane x=0, figure 1), in which case the obtained measures may not be very accurate.

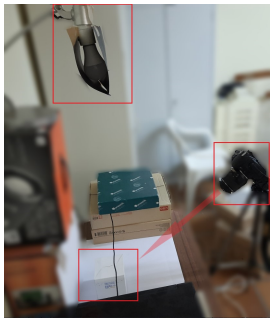


Figure 5: Real Setup

- Develop an improved solution, in which the equation of the shadow plane is calibrated.

We decided to dive into the second option in which we obtain a more accurate and generic result. For that, we had to calculate the location of 3 points of the shadow plane to find its equation. We will go more in-depth about this on the Calculation of 3D coordinates step, where we explain how to calculate any other real world point of the plane.

2.4 Image acquisition

An image of a cube, with a known size (10x10x10 cm), being lit by the shadow plane was acquired on a flat surface by the camera.

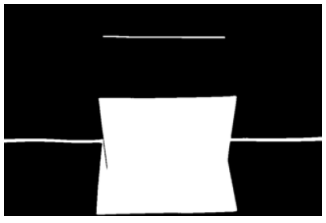
Figure 6: Object lit by the shadow plane

2.5 Line detection

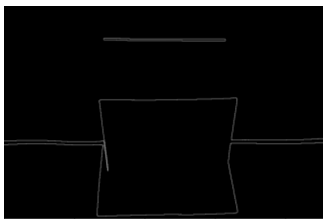
In this step we are meant to detect the shadow line in the acquired image and obtain the coordinates of its edge points. To do this we performed several steps:

- **Threshold**: First, we convert the image to gray. This is a necessary step, since the process of thresholding requires it.

Then we apply a threshold value for every pixel. If the pixel value is smaller than the threshold, it is set to 0, otherwise it is set to a maximum value.

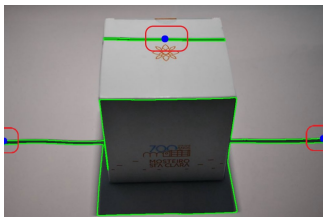


- **Laplacian**: Here, by using the Laplacian Gradient Filter, we were able to detect the edges.



- **Find contours**: Next, we find the contours of the lines where contours can simply be explained as a curve joining all the continuous points (along the boundary), having the same color or intensity.

- **Find 2D points**: To define the shadow plane we need 3 of its points. We select the leftmost and rightmost point from the shadow line and also the middle point in the shadow line in the upper face of the cube.



2.6 Calculation of 3D coordinates

After calculating the 2D coordinates of the previous points, we need to calculate the real-world coordinates of those points so that we can calculate the shadow plane equation. To determine those points we need to use the Perspective Matrix, the plane coefficients, and each point calculated in previous steps. Also, we consider that the two 2D points that are on the extremity of the image to be on the plane z = 0 and that the central point to be on the plane z = 10.

After calculating the shadow plane equation, we use it to calculate the height map of our cube using the shadow plane equation coefficients. What we are left with is a graph that shows the differences in height along the shadow line.

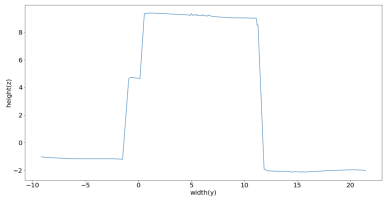


Figure 7: Cube's height map

3 Efficacy of the Used Methods

This section presents the main problems and their respective solution or problem minimization.

3.1 Main Problems

1. **Calculation of shadow line** - the shadow line is not correctly calculated if there are other shadows in the image. This might lead to an incorrect calculation of the height map of the object because the other shadows might be mistaken for the shadow line.

2. **Distorted images** - the images are not undistorted using the camera matrix. This leads to errors in the calculation of the shadow plane because the outward most pixels are in fact distorted and, as such, don't correspond to their real positions. The reason the images are not undistorted is because the undistortion process creates black spaces on the borders of the image, that would later be picked up as shadow lines.

3. **Shadow creation** - the object used to create the shadow line is not perfectly straight. This can be seen in all the images and creates a "wavelike" effect on the calculated height map. This also causes the shadow plane to be miscalculated because the points used to calculate the plane might be part of an imperfection of the shadow creating object.

4. **Height map presentation** - the presentation of the height map does not allow for changes in the referential calculated for the calibration image. This is due to the fact that the height map is calculated along the y axis.

3.2 Proposed solutions

1. **Calculation of shadow line** - To solve this problem there are two solutions:

   (a) Obtain a better image of the object with the shadow line and no other shadows. This is hard to perform since the reduction of shadows on the object implies the use of another light source which will reduce the visibility of the shadow line. This also depends on the used object due to the fact that the object itself casts shadows on the surface plane.

   (b) Remove unwanted shadows programmatically. This solution requires the use of specific methods to remove the unwanted shadows which introduces hard-coded features that do not work for every image with the same results.

2. **Distorted images** - a possible solution to this problem would be to crop the undistorted image, but this can lead to the loss of information regarding the obtained image.

3. **Shadow creation** - this problem can be fixed by using another object to create the shadow line, but due to resource limitations this could not be done without increasing the thickness and the blur of the line.

4. **Height map presentation** - This situation can be solved using the i-axis pixels instead of the y coord, which is always dependent on the referential's position.

4 Status of the Proposed Method

4.1 Degree of Fulfillment of the Aims

- **Acquisition setup**: Some of the calibration images used in the project were not taken using the best methodology because the chess pattern was not on the same surface level as the final object. This can lead to a bigger re-projection error which will be talked about in the Performance section.

- **Camera calibration**: The group was able to obtain the intrinsic and extrinsic parameters by applying cv2.cameraCalibrate and, subsequently, we used cv2.getOptimalNewCameraMatrix which allowed us to obtain a more precise camera matrix (intrinsic parameters), that lead to better results.

- **Light projection system calibration**: For this part it was developed an improved solution in which the equation of the light/shadow plane is calibrated. This was achieved introducing two different planes to the algorithm and selecting three points (at least one of each plane).

- **Image acquisition**: As referred before in the main problems section, the image acquisition wasn't perfect. However, within the group's possibility, the obtained images were good enough to calculate the height map but presented some errors by having too many shadows and an imperfect shadow line.

- **Line detection**: The group was able to create a line detection algorithm through the use of the cv2.threshold, cv2.Laplacian and cv2.findContours methods which output a variety of lines. Out of these lines, it is assumed that the top line is always correspondent to the shadow line which introduces some errors that can latter be seen in the height map acquisition.

- **Calculation of 3D coordinates**: This step was fully done, but showed an error which caused a deviation in the height of the shadow line. Since the incident shadow line on the object was also affected by this problem in the same manner and with the same deviation, this allows the height of the object to be calculated correctly afterwards.

5 Performance

The re-projection error is a geometric error corresponding to the image distance between a projected point and a measured one. It is used to quantify how closely an estimate of a 3D point recreates the point's true projection; By our calculation, the re-projection error gave us 10%. This re-projection error was calculated using the same images used in the camera calibration section.

As previously stated, the height map calculation is determined by calculating the height of the points in the shadow line. The result given by the height map, is very accurate, since the object has 10cm in height and the obtained value is 10.5cm.This gives us a 5% error margin which is comprehensible since our re-projection error is 10%. The one performance problem we detected, as stated before, is the calculation of the shadow line that can have some errors if there are other shadows in the image besides the shadow line.

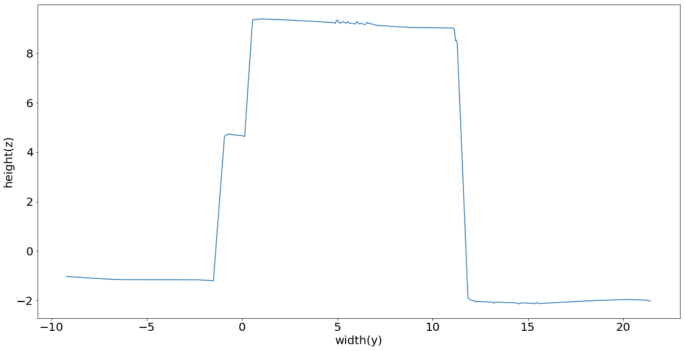


Figure 8: Cube's height map

Moreover, the performance is also affected by the quality and size of the camera/images. Despite these edge cases where the program has trouble calculating the height map, it proved to have great accuracy overall.

References

- https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html
- https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html
- https://en.wikipedia.org/wiki/David_Laserscanner
- https://docs.opencv.org/3.3.0/d9/d0c/group__calib3d.html

Calculate X, Y, Z Real World Coordinates from Image Coordinates using OpenCV and Structured Light

April 23, 2021

Faculdade de Engenharia da Universidade do Porto

It is recommended to view this notebook in nbviewer for the best overall experience

You can also execute the code on this notebook using Jupyter Notebook, Binder or Google Colab (no local installation required)

1 Table of contents

2 Introduction

go back to the top

Structured light techniques for 3D data acquisition play a central role in many 3D data acquisition applications, namelly when the surfaces to be measured do not have feature points or when it is necessary to obtain dense 3D data. They are used in numerous applications: industrial (ex: dimensional control or quality inspection), reverse engineering, urban (ex: road inspection) and medical, are just a few examples.

These techniques are based on the acquisition of an image of a scene over which a light pattern is projected; this pattern ranges from a single light ray or a single light sheet to a set of parallel

sheets or a pseudo-random pattern. Frequently, laser light is used to simplify the detection of the projected patterns.

In this work we will have the opportunity of implementing a 3D data acquisition system based on structured light, using a single sheet of light/shadow

3 Required libraries

go back to the top

The primary libraries that we'll be using are:

- numpy: Provides a fast numerical array structure and helper functions.
- cv2: OpenCV provides a real-time optimized Computer Vision library, tools, and hardware.
- glob: Is used to retrieve files/pathnames matching a specified pattern.
- matplotlib: Basic plotting library in Python, with capabilities of showing images.
- sympy: Library for symbolic mathematics and for solving equations.

[1]:
```
import os
import numpy as np
import cv2
import glob
import math
import matplotlib.pyplot as plt
from sympy import symbols, Eq, solve, poly
```

[2]:
```
# number of interior squares of chess Board
n_grid = 7

# chessboard square size in cm
square_size = 2.5

# termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.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((n_grid*n_grid,3), np.float32)
objp[:,:2] = np.mgrid[0:n_grid,0:n_grid].T.reshape(-1,2)

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

line_detection_path = 'images/lineDetection'
calibration_path = 'images/calibration'
calibration_images = glob.glob(os.path.join(calibration_path,'*.JPG'))
```

4 Camera calibration

4.1 Intrinsic parameters

Intrinsic parameters are specific to a camera. They include information like focal length (fx,fy) and optical centers (cx,cy). The focal length and optical centers can be used to create a camera matrix, which can be used to remove distortion due to the lenses of a specific camera. The camera matrix is unique to a specific camera, so once calculated, it can be reused on other images taken by the same camera.

4.2 Extrinsic parameters

Extrinsic parameters corresponds to rotation and translation vectors which translates a coordinates of a 3D point to a coordinate system.

[3]:
```
for fname in calibration_images:
    img = cv2.imread(fname)

    # TODO: Why do we put thin in gray? Does it affect results?
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    # Find the chess board corners
    ret, corners = cv2.findChessboardCorners(img, (n_grid,n_grid), None)

    # If found, add object points, image points (after refining them)
    if ret:
        obj_points.append(objp)
        img_point = cv2.cornerSubPix(gray, corners, (11, 11), (-1,-1), criteria)
        img_points.append(img_point)

# After acquiring the object and image points we need to calibrate the camera.␣
 ↪For that we use the function, cv2.calibrateCamera() that returns the camera␣
 ↪matrix, distortion coefficients, rotation and translation vectors.
_, camera_matrix, dist, rvecs, tvecs = cv2.calibrateCamera(obj_points,␣
 ↪img_points, gray.shape[::-1], None, None)
```

5 Re-projection Error

Re-projection error gives a good estimation of just how exact the found parameters are. The closer the re-projection error is to zero, the more accurate the parameters we found are.

[4]:
```
total_error = 0
for i in range(len(obj_points)):
    img_points_tes, _  = cv2.projectPoints(obj_points[i], rvecs[i], tvecs[i],␣
 ↪camera_matrix, dist)
    error = cv2.norm(src1=img_points[i], src2=img_points_tes, normType=cv2.
 ↪NORM_L2) / len(img_points_tes)
```

```
    total_error += error

print("mean error: {}".format(total_error / len(obj_points)))
```

```
mean error: 0.10136241510516156
```

6 Undistortion

Now we can take an image and undistort it using the distortion coeficients.

[5]:
```
imgcal = cv2.imread(os.path.join('images', 'img_cal.jpg'))
h,  w = imgcal.shape[:2]
new_camera_mtx, roi=cv2.getOptimalNewCameraMatrix(camera_matrix, dist, (w,h), 1,␣
 ↪(w,h))

# undistort
undst = cv2.undistort(imgcal, camera_matrix, dist, None, new_camera_mtx)

# crop the image
x,y,w,h = roi
undst = undst[y:y+h, x:x+w]
cv2.imwrite('images/calibresult.png', undst)
```

[5]:
```
True
```

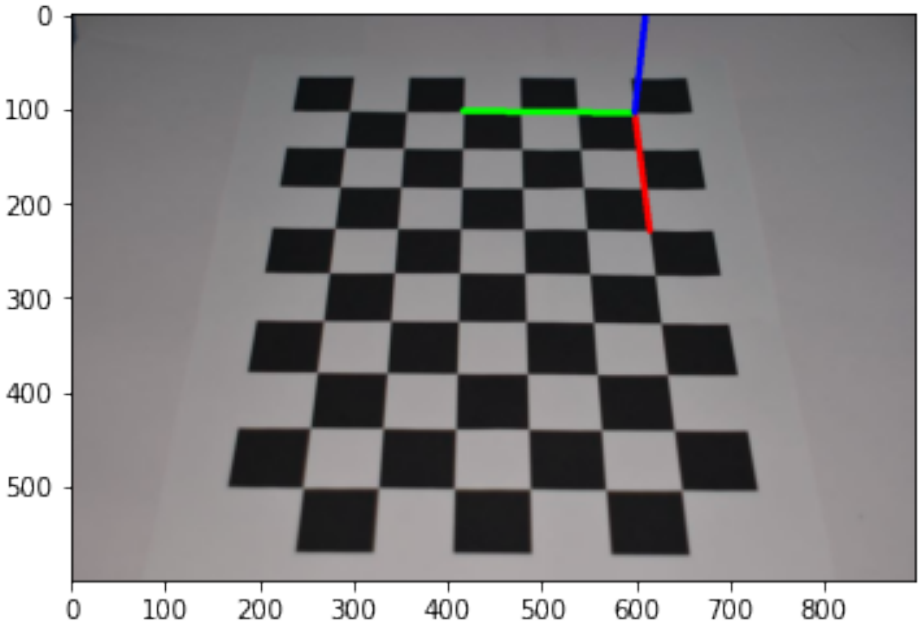
7 Perspective Projection Matrix

We create the Perspective Projection Matrix using the Camera Matrix (obtained in the function calibrateCamera), the Rotation Matrix (which we are going to calculate using Rodrigues function), and the Translation Vector (later obtained in the SolvePnP function), where all these functions are a part of the cv2 library

[6]:
```
#The draw function implemented next draws the reference axis in our selected␣
 ↪image
def draw(img, corners, imgpts):
    corner = tuple(corners[0].ravel())
    img = cv2.line(img, corner, tuple(imgpts[0].ravel()), (255,0,0), 5)
    img = cv2.line(img, corner, tuple(imgpts[1].ravel()), (0,255,0), 5)
    img = cv2.line(img, corner, tuple(imgpts[2].ravel()), (0,0,255), 5)
    return img
```

[7]:
```
axis = np.float32([[3,0,0], [0,3,0], [0,0,-3]]).reshape(-1,3)

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
#The np.zeros matrix was changed from (n_grid, n_grid) = (7,7) to (9,6)
#because the new chessboard image has 9*6 inners corners
```

```
objp2 = np.zeros((9*6,3), np.float32)
objp2[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
obj_points2 = [] # 3d point in real world space
img_points2 = [] # 2d points in image plane.

gray = cv2.cvtColor(undst, cv2.COLOR_BGR2GRAY)
ret, corners = cv2.findChessboardCorners(gray, (9,6),None)
if ret:
    corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
    # Find the rotation and translation vectors.
    ret, rvecs1, tvecs1, inlier = cv2.solvePnPRansac(objp2, corners2,␣
 ↪camera_matrix, dist)

    # project 3D reference into image plane
    imgpts, jac = cv2.projectPoints(axis, rvecs1, tvecs1, camera_matrix, dist)
    ref_img = draw(undst, corners, imgpts)
    plt.imshow(ref_img)
    plt.show()
    rotM, _= cv2.Rodrigues(rvecs1)
```



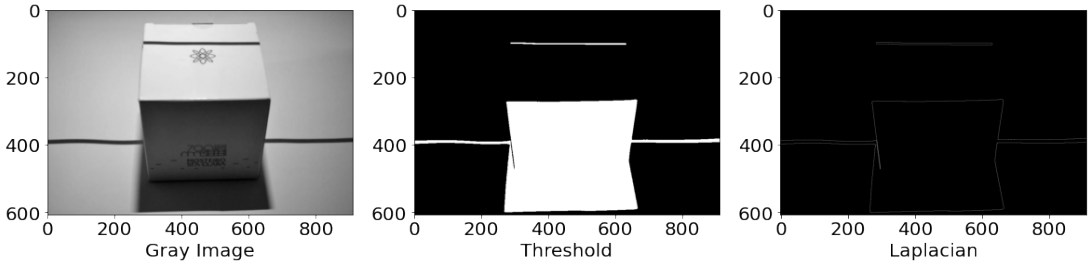
[8]:
```
#Prespective Projection Matrix -
K = new_camera_mtx
```

```
#In this block, we homogenize the matrices that are going to be a part of the␣
 ↪Prespective Projection Matrix (PPMatrix) because
# Using homogeneous coordinates, Rotation and Translation can be expressed by a␣
 ↪single matrix
Homog_K = np.array([[K[0][0], K[0][1], K[0][2], 0], [K[1][0], K[1][1], K[1][2],␣
 ↪0], [K[2][0], K[2][1], K[2][2], 0]])
Homog_R = np.array([[rotM[0][0], rotM[0][1], rotM[0][2], 0],[rotM[1][0],␣
 ↪rotM[1][1], rotM[1][2], 0], [rotM[2][0], rotM[2][1], rotM[2][2], 0],[0, 0, 0,␣
 ↪1]])
Homog_T = np.array([[1, 0, 0, tvecs1[0][0]],[0, 1, 0, tvecs1[1][0]], [0, 0, 1,␣
 ↪tvecs1[2][0]], [0, 0, 0, 1]])
Homog_Ext = np.matmul(Homog_T, Homog_R)
PPMatrix = np.matmul(Homog_K, Homog_Ext)
```

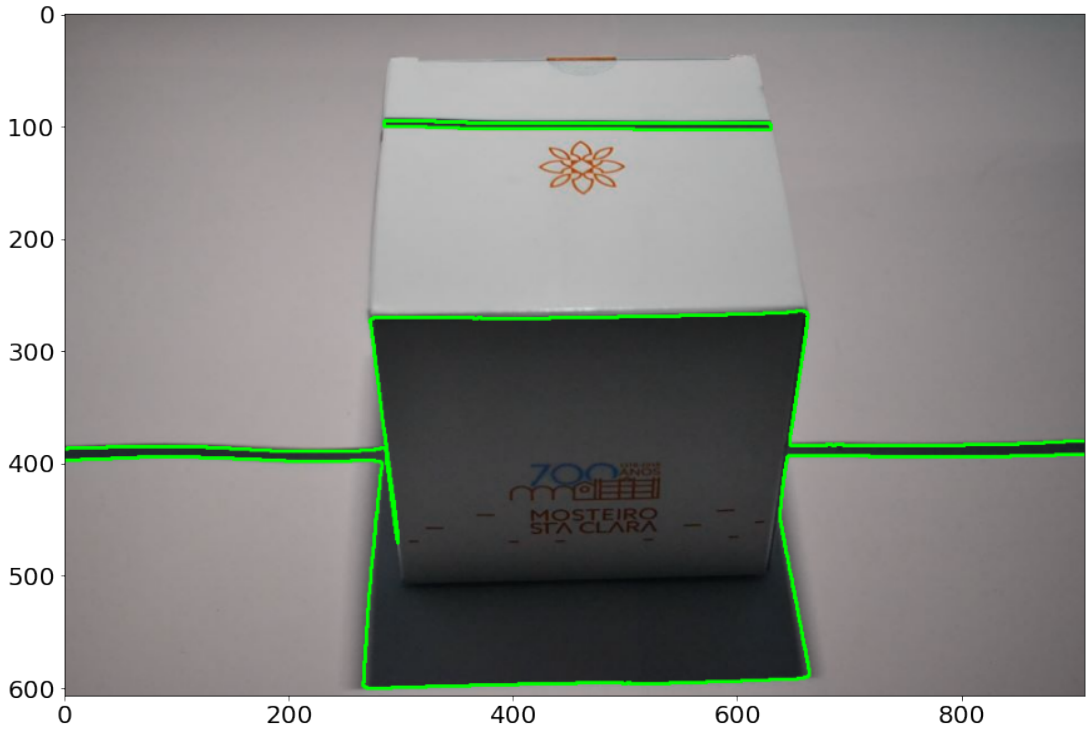
8 Line Detection

[9]:
```
# Gray image
img_shadow_plane = cv2.imread(os.path.join(line_detection_path, 'img4.JPG'))
gray = cv2.cvtColor(img_shadow_plane, cv2.COLOR_BGR2GRAY)

gray = cv2.GaussianBlur(gray, (5,5), 0)

# Threshold
low_threshold = 80
high_threshold = 120
_,thresh = cv2.threshold(gray, low_threshold, high_threshold, cv2.THRESH_BINARY)
thresh = 255 - thresh

thresh = cv2.erode(thresh, kernel=(1, 1), iterations=2)

# Laplacian
laplacian = cv2.Laplacian(thresh, cv2.CV_8U)

plt.rcParams["figure.figsize"] = (20, 10)
plt.rcParams.update({'font.size': 20})
fig, ((ax1, ax2, ax3)) = plt.subplots(1, 3)

ax1.imshow(gray, cmap='gray')
ax1.set_xlabel('Gray Image')

ax2.imshow(thresh, cmap='gray')
ax2.set_xlabel('Threshold')

ax3.imshow(laplacian, cmap='gray')
ax3.set_xlabel('Laplacian')
```

```
plt.show()
```



[10]:
```
# Then, use findContours to get the contours. You can adjust the parameters for␣
↪better performance.

contours2, _ = cv2.findContours(laplacian, cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
cont2 = cv2.drawContours(img_shadow_plane, contours2, -1, (0,255,0), 2)

plt.imshow(cont2, cmap='gray')
plt.show()
```



Here, we defined some variables in order to filtrate some of the overlapping lines obtained with

findContours. Basically, what happens is that we select 3 points on top of the shadow line: the highest left point, the highest right point and the highest middle point.

Also, xl, yl, xr, yr, xm, ym are defined with the value -1 and width+1 because they are supposed to represent pixels, and by assigning the value -1 and width + 1, there is no risk of actually missing a point value because these values do not belong in the image.

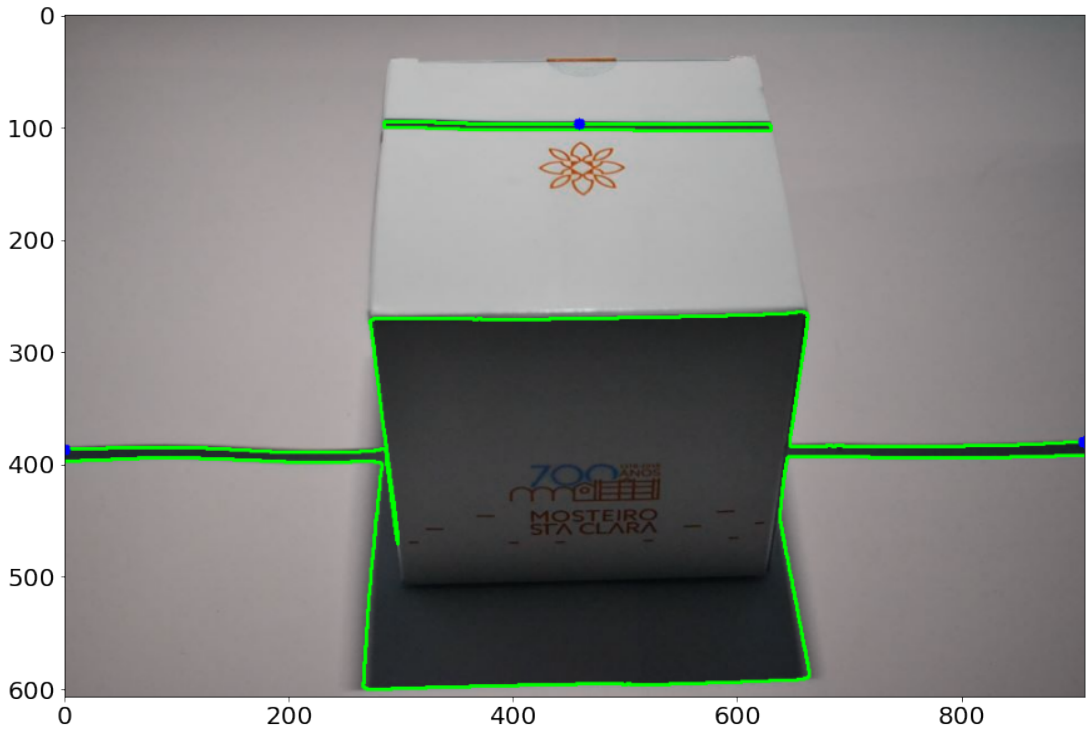
In order to regularize the obtained pixels we assure that the (xm, ym) pixel refers to the line in the upper plane of our image, (xl, yl) to the left line on the lower plane and (xr, yr) to the right line on the lower plane

[11]:
```
width = img_shadow_plane.shape[1]
mid_point = math.ceil(width/2)

xl=width+1
yl=width+1
xr=-1
yr=width+1
xm=-1
ym=width+1


for contour in contours2:
    for point in contour:
        if point[0][0] <= xl:
            if point[0][0] == xl:
                if point[0][1] <= yl:
                    xl = point[0][0]
                    yl = point[0][1]
            else:
                xl = point[0][0]
                yl = point[0][1]
        if point[0][0] >= xr:
            if point[0][0] == xr:
                if point[0][1] <= yr:
                    xr = point[0][0]
                    yr = point[0][1]
            else:
                xr = point[0][0]
                yr = point[0][1]
        if (mid_point-5 < point[0][0] < mid_point+5) and point[0][1] < ym:
            xm = point[0][0]
            ym = point[0][1]
```

Now, it's only necessary to obtain the A, B, C, D and a, b, c, d variables of each pixel plane in order to obtain the pixel values in our world coordinate. For this, we only need to add the value of a selected plane which, in our case, was z=10 for (x10, y10) and z=0 for the other 2 points.

[12]:
```
#1º Pixel
#(xm,ym) it's in the plane z1=10

cv2.circle(img_shadow_plane, (xm,ym), 5, (0,0,255), -1)

A1 = PPMatrix[2][0]*xm - PPMatrix[0][0]
B1 = PPMatrix[2][1]*xm - PPMatrix[0][1]
C1 = PPMatrix[2][2]*xm - PPMatrix[0][2]
D1 = PPMatrix[0][3] - PPMatrix[2][3]*xm

a1 = PPMatrix[2][0]*ym - PPMatrix[1][0]
b1 = PPMatrix[2][1]*ym - PPMatrix[1][1]
c1 = PPMatrix[2][2]*ym - PPMatrix[1][2]
d1 = PPMatrix[1][3] - PPMatrix[2][3]*ym

z1=-10/square_size
x1,y1 = symbols('x1 y1')
eq1 = Eq(A1*x1 + B1*y1 + C1*z1 - D1, 0)
eq2 = Eq(a1*x1 + b1*y1 + c1*z1 - d1, 0)

sol1 = solve((eq1,eq2), (x1,y1))

#2º Pixel
#(xl,yl) it's in the plane z2=0

cv2.circle(img_shadow_plane, (xl,yl), 5, (0,0,255), -1)

A2 = PPMatrix[2][0]*xl - PPMatrix[0][0]
B2 = PPMatrix[2][1]*xl - PPMatrix[0][1]
C2 = PPMatrix[2][2]*xl - PPMatrix[0][2]
D2 = PPMatrix[0][3] - PPMatrix[2][3]*xl

a2 = PPMatrix[2][0]*yl - PPMatrix[1][0]
b2 = PPMatrix[2][1]*yl - PPMatrix[1][1]
c2 = PPMatrix[2][2]*yl - PPMatrix[1][2]
d2 = PPMatrix[1][3] - PPMatrix[2][3]*yl

z2=0
x2,y2 = symbols('x2 y2')
eq3 = Eq(A2*x2 + B2*y2 + C2*z2 - D2, 0)
eq4 = Eq(a2*x2 + b2*y2 + c2*z2 - d2, 0)

sol2 = solve((eq3,eq4), (x2,y2))

#3º Pixel
#(xr,yr) it's in the plane z2=0
```

```
cv2.circle(img_shadow_plane, (xr,yr), 5, (0,0,255), -1)


A3 = PPMatrix[2][0]*xr - PPMatrix[0][0]
B3 = PPMatrix[2][1]*xr - PPMatrix[0][1]
C3 = PPMatrix[2][2]*xr - PPMatrix[0][2]
D3 = PPMatrix[0][3] - PPMatrix[2][3]*xr

a3 = PPMatrix[2][0]*yr - PPMatrix[1][0]
b3 = PPMatrix[2][1]*yr - PPMatrix[1][1]
c3 = PPMatrix[2][2]*yr - PPMatrix[1][2]
d3 = PPMatrix[1][3] - PPMatrix[2][3]*yr

z3=0
x3,y3 = symbols('x3 y3')
eq5 = Eq(A3*x3 + B3*y3 + C3*z3 - D3, 0)
eq6 = Eq(a3*x3 + b3*y3 + c3*z3 - d3, 0)

sol3 = solve((eq5,eq6), (x3,y3))

plt.imshow(img_shadow_plane)
plt.show()
```



Now, we have 3 points: (x1, y1, z1); (x2, y2, z2); (x3, y3, z3); which is the number of points we need to calculate the shadow plane! This means that we only need to do some equation solving to obtain the A, B and C plane variables (to the variable D it is going to be attributed a constant value of 1).

[13]:
```
varA, varB, varC= symbols('A B C')

#D is considered = 1 to calculate the shadow plane
D = 1

eqABC1 = Eq(varA*sol1[x1] + varB*sol1[y1] + varC*z1, D)
eqABC2 = Eq(varA*sol2[x2] + varB*sol2[y2] + varC*z3, D)
eqABC3 = Eq(varA*sol3[x3] + varB*sol3[y3] + varC*z3, D)
solABC = solve((eqABC1, eqABC2, eqABC3), (varA, varB, varC))
A, B, C = solABC[varA], solABC[varB], solABC[varC]

print('(A, B, C, D)')
print('{}, {}, {}, {})'.format(A, B, C, D))
```

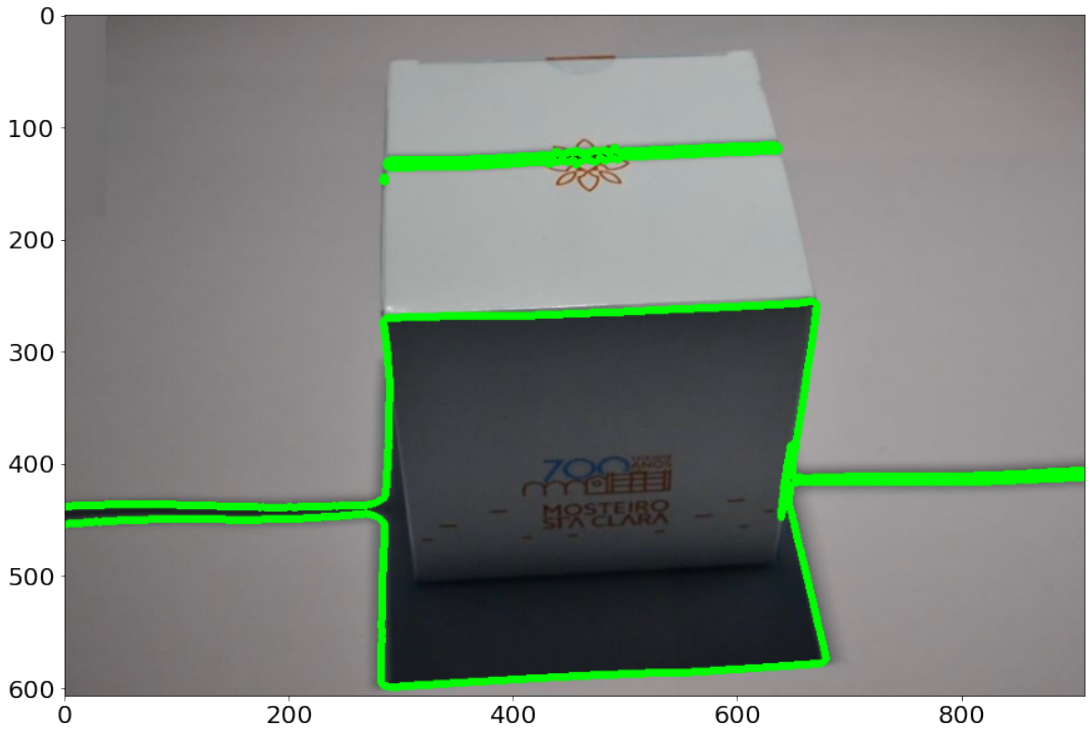
```
(A, B, C, D)
0.164415409206625, -0.00315164790714390, -0.0417105986946354, 1)
```

[14]:
```
#This function calculates the 3D coordinates given the Perspective Projection␣
 ↪Matrix (PPM), the plane coefficients (A, B, C, D) and the
#pixel point we want to convert (i, j)
def calculate3DPointCoords(PPM, i, j, A, B, C, D):
    x_var, y_var, z_var = symbols('X Y Z')
    calcX = Eq((PPM[2][0] * i - PPM[0][0]) * x_var + (PPM[2][1]*i - PPM[0][1]) *␣
 ↪y_var + (PPM[2][2]*i - PPM[0][2]) * z_var - PPM[0][3] + PPM[2][3]*i, 0)
    calcY = Eq((PPM[2][0] * j - PPM[1][0]) * x_var + (PPM[2][1]*j - PPM[1][1]) *␣
 ↪y_var + (PPM[2][2]*j - PPM[1][2]) * z_var - PPM[1][3] + PPM[2][3]*j, 0)
    calcZ = Eq(A * x_var + B * y_var + C * z_var - D, 0)
    sol = solve((calcX, calcY, calcZ), (x_var, y_var, z_var))

    return sol[x_var], sol[y_var], -sol[z_var]
```

Here, we get a new image with the same shadow plane and apply findContours to obtain the line representation of that plane.

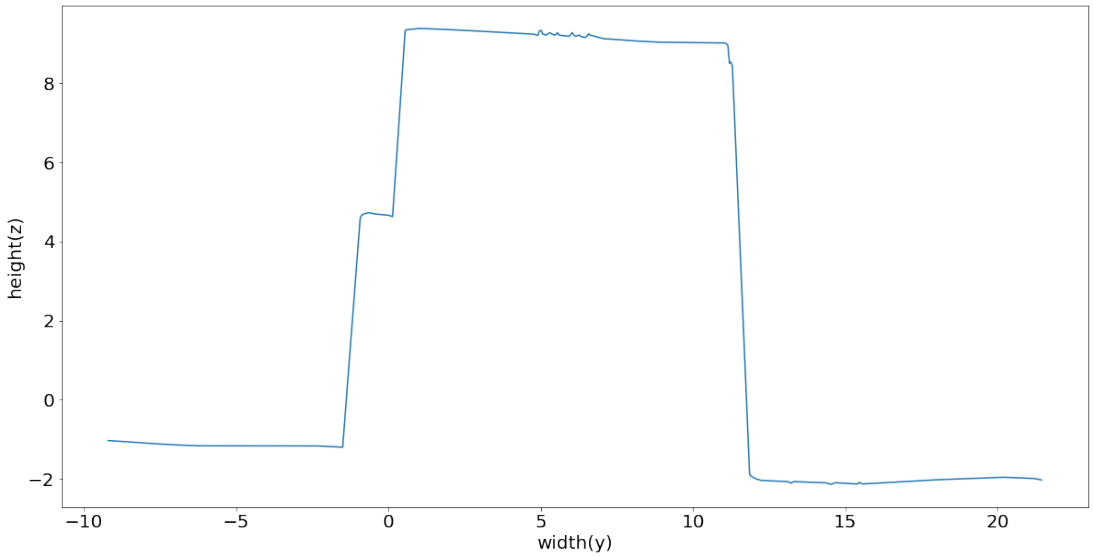
[15]:
```
# Gray image
img_to_represent = cv2.imread(os.path.join(line_detection_path, 'img1.JPG'))
gray_rep = cv2.cvtColor(img_to_represent, cv2.COLOR_BGR2GRAY)

gray_rep = cv2.GaussianBlur(gray_rep, (5,5), 0)

# Threshold
low_threshold = 80
high_threshold = 120 #low_threshold * 3
```

```
_,thresh_rep = cv2.threshold(gray_rep, low_threshold, high_threshold, cv2.
 ↪THRESH_BINARY)
thresh_rep = 255 - thresh_rep

kernel = cv2.getStructuringElement(cv2.MORPH_RECT, ksize=(2, 2))

# Remove noise
thresh_rep = cv2.erode(thresh_rep, kernel=(1,1), iterations=2)

# Laplacian
laplacian_rep = cv2.Laplacian(thresh_rep, cv2.CV_8U)


contours, _ = cv2.findContours(laplacian_rep, cv2.RETR_TREE, cv2.
 ↪CHAIN_APPROX_NONE)
cont = cv2.drawContours(img_to_represent, contours, -1, (0,255,0), 5)
plt.imshow(cont, cmap='gray')
plt.show()
```



After having found the contours of the new image, it is now possible to calculate the height map of the object using the shadow plane coefficients (A, B, C, D) found earlier.

[16]:
```
# Find highest pixel in contours
height = laplacian_rep.shape[0]
maximum = np.ones(width, dtype=int)*height

for contour in contours:
    for point in contour:
        i = point[0][0]
        j = point[0][1]
        if maximum[i] > j:
           maximum[i] = j

# Compress maximum array and calculate 3D points into heightMap
heightMap = []
curr_max = -1

for i, max_val in enumerate(maximum):
    if max_val != curr_max and max_val != height:
        heightMap.append(calculate3DPointCoords(PPMatrix, i, max_val, A, B, C,␣
 ↪D))
        curr_max = max_val

# Create a height histogram with calculated height map
histHeight = [0.0, 1.0] * len(heightMap)
histHeight = np.array(histHeight).reshape((len(heightMap), 2))

for i, point in enumerate(heightMap):
    histHeight[i][0] = point[2] * square_size
    histHeight[i][1] = point[1] * square_size

plt.plot(histHeight[:,1], histHeight[:,0])
plt.xlabel('width(y)')
plt.ylabel('height(z)')
plt.show()
```



References - https://docs.opencv.org/3.3.0/dc/dbb/tutorial_py_calibration.html - https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html - https://docs.opencv.org/3.3.0/d9/d0c/group__calib3d.html