VCOM 1st Project Report: 3D Data Acquisition using a Structured Light Technique

Bernardo Santos
up201706534@edu.fe.up.pt
David Silva
up201705373@edu.fe.up.pt
Laura Majer
up202010647@edu.fe.up.pt
Luís Cunha
up201706736@edu.fe.up.pt

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

Abstract

In this work we use a structured light technique to implement a system for 3D data acquisition, namely information about the height of objects, using household items. The technique consists of casting a pattern of light/shadow over the objects, capturing an image of said objects and, after detecting and extracting the pattern of the line, using the perspective projection matrix to compute the real world coordinates of the pattern points. With the image coordinate of those points and the projection matrix we are only able to obtain the line of sight of the image point, but by constraining the plane (light or shadow) to which the points of the pattern belong we are able to obtain the real world coordinates of each of the points of the pattern. We explore several computer vision techniques, including edge detection, data acquisition, and camera calibration. We are able to measure objects that do not have significant texture with a satisfactory precision, as well as introduce some degree of automation to the selection of suitable parameters for the algorithms used.

1 Introduction

Structured light techniques can be used for the acquisition of 3D data with many real world applications such as dimension inspection of objects for quality control. In this work we implement a simple structured light technique using common household items to measure an object.

In Section 2.1, we will describe the set-up used for image acquisition. In Sections 2.2 to 2.6 we will discuss the methods used for the several steps of the measuring process. In Section 3 we analyse the implemented system, as well as the challenges faced. Finally, in Section 4 we draw some final remarks regarding the fulfillment of the work's goals.

2 Methodology

2.1 Experimental setup

The general setup and materials used in this methodology can be seen in Figure 1. The objects to be measured (1) are illuminated by a suspended LED light (2). This light casts a shadow onto the object using a thin stick (3). The objects are positioned on a large sheet of white paper (8) which aids the shadow point extraction.

The light's type and height were chosen with the goal of illuminating the scene sufficiently and producing a thin, crisp shadow onto the object. These characteristics are important in order to facilitate the shadow's detection, which will be detailed further ahead. Note that the same experiment can be done using a light-plane (e.g. using a laser), however the detection parameters and light conditions would need to be changed. Our experiments will focus on the use of a shadow-plane.

The pictures are recorded using a camera (4) that is positioned using a tripod (5). Positioning the camera is paramount to ensure that the target object(s) and the shadow are captured correctly, leading to good shadow extraction results. The calibration of the camera's intrinsic and extrinsic parameters is done using a chessboard pattern (7).

Finally, as will be explained ahead, in order to calibrate the shadow plane we need an object of known dimensions (6).

For better results the camera should be close to the object and able to see its top. The plane where the object is should be uncluttered and have a uniform color, which facilitates the processing of the image later on. The results are also improved by the usage of higher picture resolution. We decided on using 1280 x 720 because it allowed for a crisp image while also maintaining good algorithm performance.

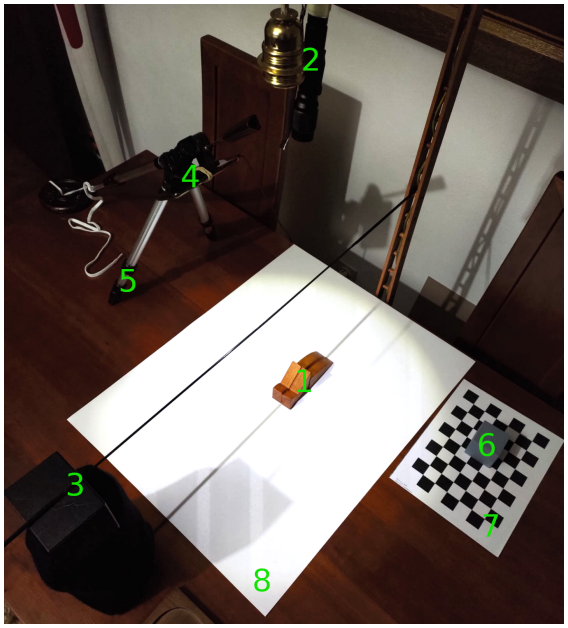

Figure 1: Experimental setup and materials used in the methodology

2.2 Intrinsic parameter calibration

The camera's intrinsic parameters [4] are composed of the camera's focal lengths (f_x, f_y) in pixels, optical center (c_x, c_y) in pixels, and sensor skew (s). These parameters are invariant to the cameras position, thus they can be calculated only once, provided that neither the camera nor its zoom changes. Henceforth we'll refer to the matrix (Equation (1)) formed by the intrinsic parameters as I_m. This step also allows the extraction of the camera's distortion coefficients (k_1, k_2, p_1, p_2, p_3) which help correct any tangential and radial distortion effects. These values allow us to convert 3D camera coordinates into 2D image coordinates.

$$I_m = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

This calibration was done with the aid of chessboard detection [3]. We started by assigning 3D coordinates to each of the inner coordinates of the chessboard (from top to bottom, left to right). This step assigned the coordinates $(0,1,0)$ to the first corner, $(0,2,0)$ to the second one and so on, which, in turn, defines the plane where the chessboard is as the $Z = 0$ plane. Then, using the *findChessboardCorners* function of the *OpenCV* library, together with the *cornerSubPix* function, we were able to find sub-pixel image coordinates of each of the corners. Having this match between 3D real-world coordinates and 2D image coordinates allows us to find matrix I_m and the distortion coefficients using the *calibrateCamera* function from *OpenCV*.

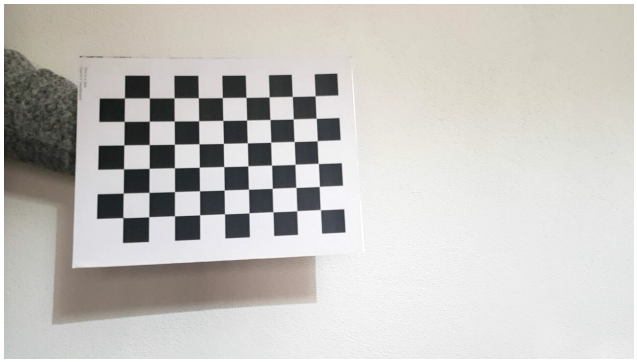

Figure 2: Chessboard image capture for intrinsic parameter calibration

Since we're only interested in calibrating the camera's intrinsic parameters, we used images similar to Figure 2 where the camera's axis is perpendicular to the chessboard, and it covers most of the camera's FOV. Furthermore, several images are used to calibrate these parameters, each having the chessboard in a different orientation.

In the end, the calculated re-projection error, which is a way of measuring the quality of the parameter estimation was around 0.05.

2.3 Extrinsic parameter calibration

Calibration of the camera's extrinsic parameters is done to account for the position of the camera relative to the scene were the objects are. As such, this calibration must be done each time the camera's position changes.

The extrinsic parameters are the translation (T) and rotation (R) matrices that represent the coordinate system transformations from 3D real world coordinates to 3D camera coordinates.

$$T = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} \qquad R = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \tag{2}$$

The rotation matrix may sometimes be represented using a Rodrigues Transformation [4] which requires only three values.

The algorithm used for extracting these parameters is very similar to the one described in the previous section. We start by assigning 3D coordinates to the inner corners of the chessboard and creating a match between these points and 2D image coordinates using the *findChessboardCorners* and *cornerSubPix* functions [1]. Then a *Perspective-n-Point* (*PnP*) algorithm is applied in order to calculate the R and T matrices. In this problem, we opted for using a variation of the *PnP* algorithm using *RANSAC* (*solvePnPRansac* function of the *OpenCV* library) because it makes it more resilient to outliers [2].

This process was applied on images similar to Figure 3. As opposed to the intrinsic calibration the camera is now positioned at an angle, according to the needs of the methodology. We measured a re-projection error of about 0.12.

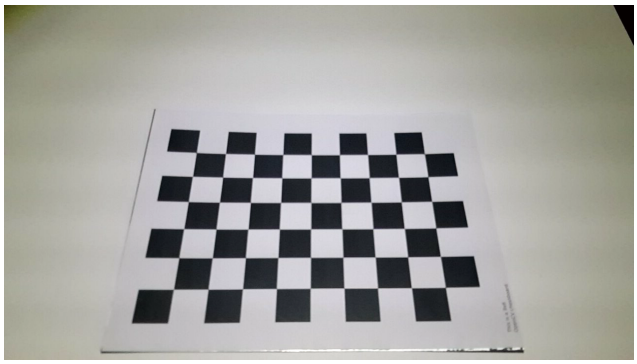

Figure 3: Chessboard image capture for extrinsic parameter calibration

2.4 Shadow points extraction

The extraction of the points of the shadow pattern from the captured images is a very important step of the whole process, since it is in this step that we obtain the points of which we want to extract information, and it is also used in the shadow plane calibration.

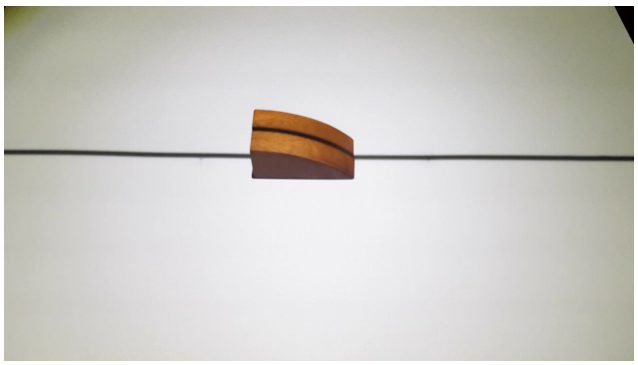

Figure 4: Original image of wood (slightly textured) object

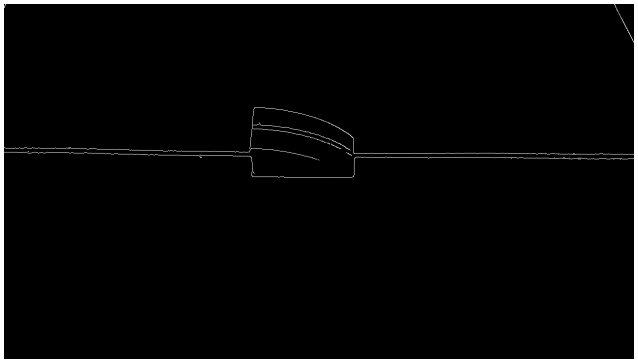

Figure 5: Result of applying Canny edge detector, with low threshold of 96 and high threshold of 110

Our methodology consisted of detecting edges on the image, and applying morphological operations to try to isolate one of the contours of the shadow line. Other approaches could consist of applying segmentation algorithms to identify the shadow, or obtaining two images, one with the shadow and other without it, and subtracting the one without the shadow to the other. The used process consists of four steps, described here.

2.4.1 Step 1: Edge detection

The first step consists of applying an edge detection algorithm. We experimented using the Sobel operator and the Canny edge detector. The latter provided satisfactory results, so we decided to use it. In this step, we detect both the edges of the shadow as well as other edges, such as those of the object being measured. We also experimented applying filters to the original image, such as the bilateral filter, in an attempt to reduce noise on the image while keeping the edges, which did not prove to improve our results. The Gaussian filter applied as part of the Canny edge detector was enough. The result of this step can be seen in Figure 5.

2.4.2 Step 2: Morphological operations

From the previous phase, both top and bottom edges of the shadow line are detected. Also, it can be seen that those edges are closer to each other than any other resulting edges. With this in mind, we applied a dilation operation with a square kernel until both edges of the shadow joined, resulting in Figure 17. After this we apply an erosion operation, also with a square kernel but of different size, until the points which do not belong to the shadow disappear, leaving only the pixels placed between both edges of the shadow line, resulting in Figure 18.

2.4.3 Step 3: Filtering the edge detection result

Initially, we considered the result of the previous step as the final result. However, it had some down sides, such as the result of the erosion having more than one pixel width across the whole image, which was not desired, especially for the plane calibration step where all the detected

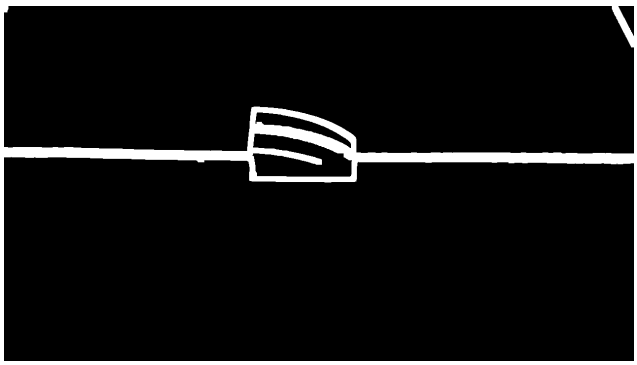

Figure 6: Result of applying the first dilation operation, with a square kernel of size 11

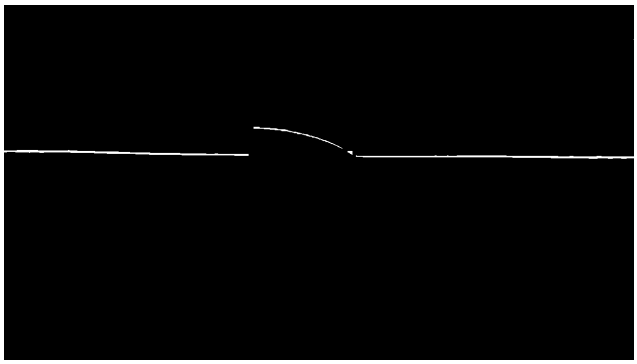

Figure 7: Result of applying the erosion operation, with a square kernel of size 17

pixels should pertain to the same plane. We tried to address this issue using the Hough transform to detect the lines from the shadow line edges, but abandoned the idea as it would not work for objects with sides that are not planar.

To improve the obtained lines we added an additional step consisting of applying a dilation operation to the result of the previous step, with a square kernel where only the bottom half elements were set to 1. The goal of this operation is for the white component resulting of the previous step to grow upwards, as seen on Figure 26. We then apply a *bitwise and* operation between it and the result of the step of edge detection, resulting in removing every edge from it except the top edge of the shadow line (Figure 20).

2.4.4 Step 4: Remove small components

As it may happen that a few pixels which do not belong to the shadow line "survive" the erosion operation, we also use the *connectedComponents* OpenCV function to find components which have an area (number of pixels) smaller than a given number, and remove them, resulting in Figure 10.

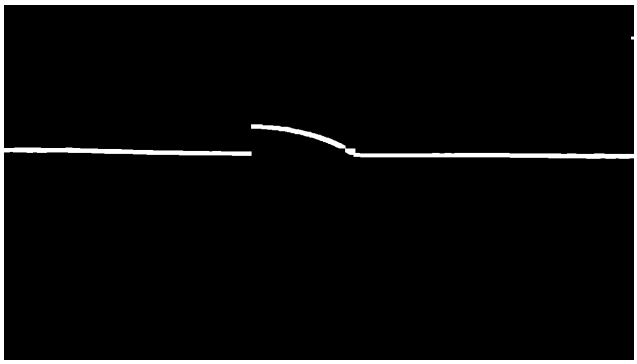

Figure 8: Result of applying dilation with a kernel with the goal of dilating the white line upwards

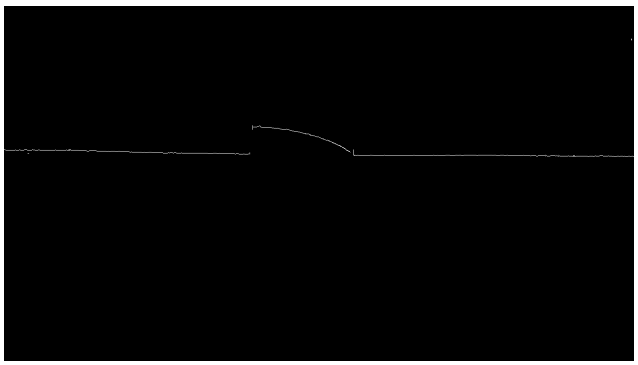

Figure 9: Result of doing a *bitwise and* between the result of the upwards dilation and the output of Canny edge detector

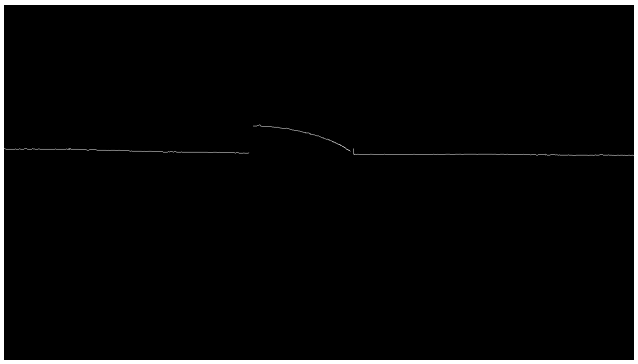

Figure 10: Result of removing connected components with an area smaller than 10

2.4.5 Parameter selection

A challenging aspect of the shadow points extraction is the selection of suitable parameters for use in the Canny edge detector, as well as the kernel sizes of the morphological operations, as different parameters provide different results in different images.

To try to automate the selection of suitable parameters, we had an approach of doing an exhaustive search within a set of parameters and measuring the quality of the obtained result using a simple evaluation function. We then use the image obtained which minimizes the evaluation function. The evaluation function results from the observation that the result we are interested in ideally has one white pixel per column, *i. e.* the line has width of one pixel. So, we compute the score using Equation 2.4.5, being W is the width of the image and n_j is the number of white pixels along column j.

$$F_\beta = (1+\beta^2) \cdot \frac{precision \cdot recall}{} \tag{3}$$

The lower and upper bound of the parameters in which the search is conducted was decided taking into account the insights we gathered by manually tweaking the parameters for our images, such as that for the images with which we experimented the thresholds used in the Canny algorithm gave best results in the range between 100 and 150, and the size of the kernel used in the erosion gave best results when it was at larger than the dilation kernel by at least 4. In the Canny detector, we always use aperture size 3. We also empirically defined a step from which to use the parameters, taking into account the trade-off between using a large step, which may miss a parameter that gives better results, and a smaller step, which takes too long to search.

The images presented in this section, as well as in Appendix A, were obtained using the parameters selected with the method described here.

2.5 3D coordinate extraction

Joining the two matrices referenced in Section 2.3 we arrive at:

$$\begin{bmatrix} R & T \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \end{bmatrix} \tag{4}$$

And combining this matrix with the intrinsic matrix defined in Section 2.2, we can define the *perspective projection matrix* (*PPM*).

$$PPM = I_m \cdot \begin{bmatrix} R & T \end{bmatrix} \tag{5}$$

The *PPM* matrix can be used to find the 2D image coordinates (i, j) of any point (x, y, z) in the 3D world space. This transformation takes the following form.

$$\begin{bmatrix} wi \\ wj \\ w \end{bmatrix} = PPM \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{6}$$

Notice that the *PPM* matrix is not square, so it can't be inverted. As such, trying to find the 3D world coordinates of a a given image point results in an equation of a straight line (the line of sight through the pixel). The system can be solved by adding an extra constraint to the system, in our case, since we are trying to find 3D coordinates of the points in shadow we use the equation of the shadow plane. In Section 2.6, we'll explore how we can extract the coefficients of the shadow plane.

The function *getWhitePoint3DCoords* uses the perspective matrix and the equation of the shadow plane to find the 3D coordinates of the shadow points. Figure 11 shows the plot of the detected coordinates, both from the positive x axis and from the positive z axis. The figure is cropped as to include a more detailed view of the center of the detected points (where the object stands).

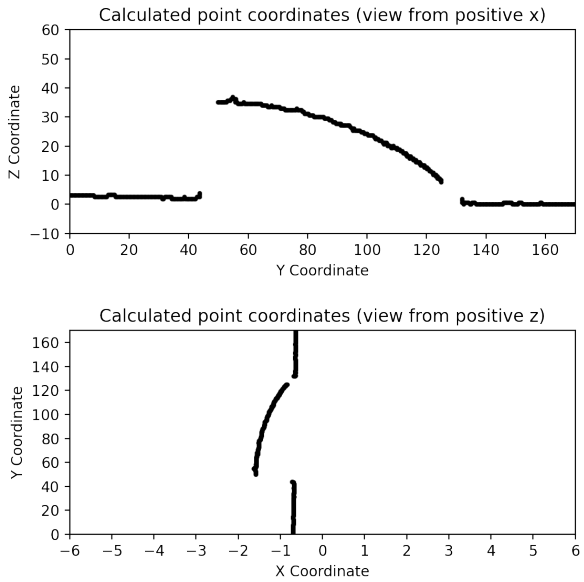

Figure 11: Object measurement result

We're able to see that the measurements obtained are very close to the actual size of the object (85 mm wide and 35 mm tall). Furthermore, the basis of the object, which lies on the $z = 0$ plane is at most 3 mm distant from this value. On the top-view we can see that the top plane points drift about 1 mm at most from the bottom plane points.

2.6 Shadow plane calibration

A plane can be defined using four coefficients as per Equation 7. Calculating these coefficients can be done using three non co-linear points. One way to obtain these points is to find the 3D coordinates of shadow points cast onto different planes with known coefficients (using these plane equations as the 4$^{\text{th}}$ constraint explained in the previous section) and from the acquired points try to estimate the coefficients of the shadow plane using any plane fitting method.

$$ax + by + cz + d = 0 \tag{7}$$

In this methodology we decided to acquire the points by projecting the shadow onto a cube with known dimensions as seen in Figure 12. Notice that the points in the table and the ones on top of the cube are non collinear.

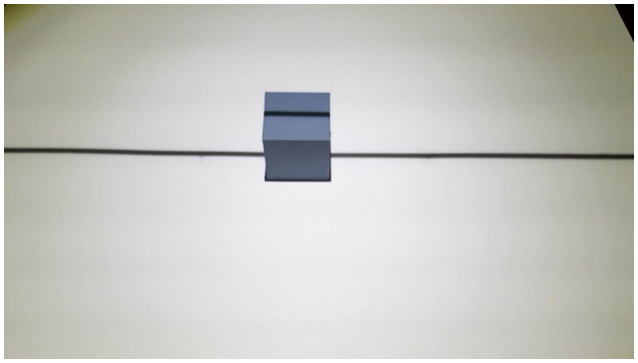

Figure 12: Cube used to calibrate the shadow plane

Next we process the image as described in Section 2.4 in order to extract the shadow points. The next step is to separate the points that are on top of the cube and on the basis of the table. As we know that the points either belong to the top of the cube, or the top the table, then they belong to one of two straight lines. We start by using the *RANSAC* algorithm implemented in the *Scikit-learn* library to isolate the points which belong to one of the lines, as the points belonging to the other line will be deemed outliers. Then, we remove the points of the first line (the inliers) from the original points and use the *RANSAC* algorithm again, this time to identify the points from the second line. To make sure that different lines are identified in the two *RANSAC* usages, we force that the y-intercept of the second usage is different (by some margin) than the first y-intercept. The results of this process are shown in Figures 13 and 14.

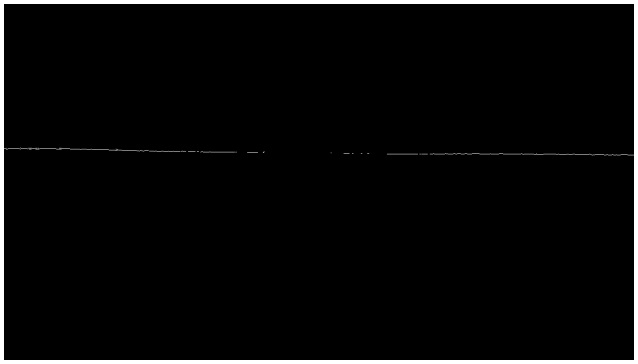

Figure 13: Shadow points from basis plane ($z = 0$)

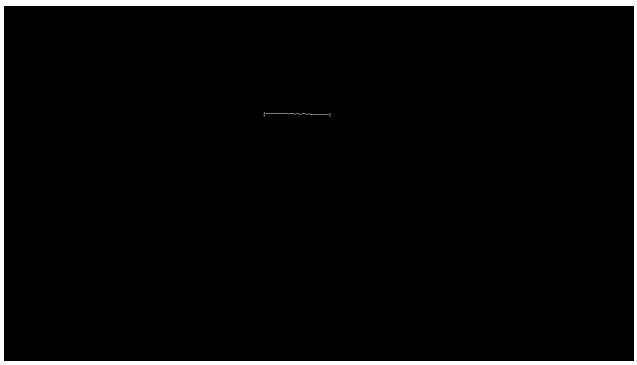

Figure 14: Shadow points from elevated plane ($z = cubeheight$)

Using the method described in Section 11 we calculate the 3D coordinates of each shadow point in the top plane (using the $z = cubeheight$ constraint) and of the basis plane (using the $z = 0$ constraint).

The first attempts to estimate the plane coefficients were made using the least squares method (*lstsq*) of the *numpy* library. These were not very successful as this method is very sensitive to outliers. In an attempt to mitigate this issue only the centermost points i.e. the ones closest to the object were used. However, this still was unable to improve the results.

In the following attempts the *RANSAC* method from *Sklearn* was used, which is resistant to outliers. This improved the results by a lot, allowing us to obtain the results seen in the previous section.

However, the non-determinism of this method, coupled with the discrepancy in the number of points between the basis and elevated plane, sometimes the algorithm would fit either the $z = 0$ or $z = cubeheight$ plane. We fixed this by using a custom *is_data_valid* function in the RANSAC method, which rejects the set of points used to approximate the plane if they have the same z coordinate.

3 Analysis

The obtained results revealed satisfactory, being able to measure a complex object with an error of only a few millimeters. Moreover, we were able to iteratively improve the methodology to require less user input. In its current state, the user only needs to provide images for intrinsic and extrinsic calibration, an image of a known object with the shadow plane cast onto it and an image of the target object hit by the same shadow.

Some of the downsides of the presented solution are that the automated selection of the shadow point extraction parameters is relatively slow, which could hurt the applicability of the system in real life situations.

Another aspect where the obtained results could be improved would be by acquiring the images with a better camera, particularly with an higher resolution. Despite this, we still standby the resolution that was used because it provided enough quality to showcase the pipeline while achieving a good algorithm performance.

4 Conclusion

Ultimately, we consider that the goals of the project were achieved, since we were able to experiment hands on the several steps of a computer vision pipeline, from image acquisition and camera calibration, to image processing and information extraction. We observed the importance of the environment in which the images are acquired. While facing some of the challenges posed by the work, we also experimented with several techniques of performing the same task, and concluded about their applicability on the task at hand.

References

[1] Pose estimation. URL `https://docs.opencv.org/4.5.1/d7/d53/tutorial_py_pose.html`.

[2] Joseph Howse and Joe Minichino. *Learning OpenCV 4 computer vision with Python 3: get to grips with tools, techniques, and algorithms for computer vision and machine learning*. Packt Publishing, 2020.

[3] Alexander Mordvintsev and Abid K. Camera calibration, 2013. URL `https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html`.

[4] Richard Szeliski. *Computer Vision: algorithms and applications*. Springer Nature, 2020.

A Shadow point extraction results

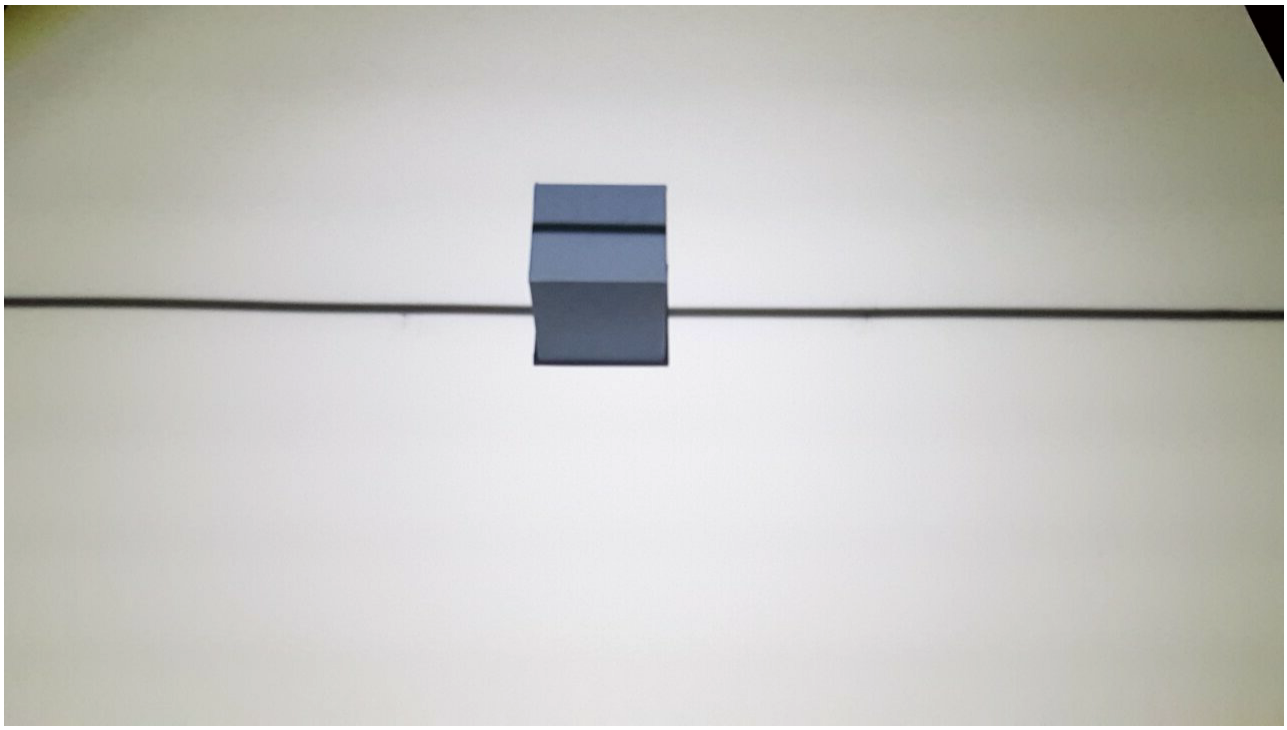

Figure 15: Captured image of a cube with our system

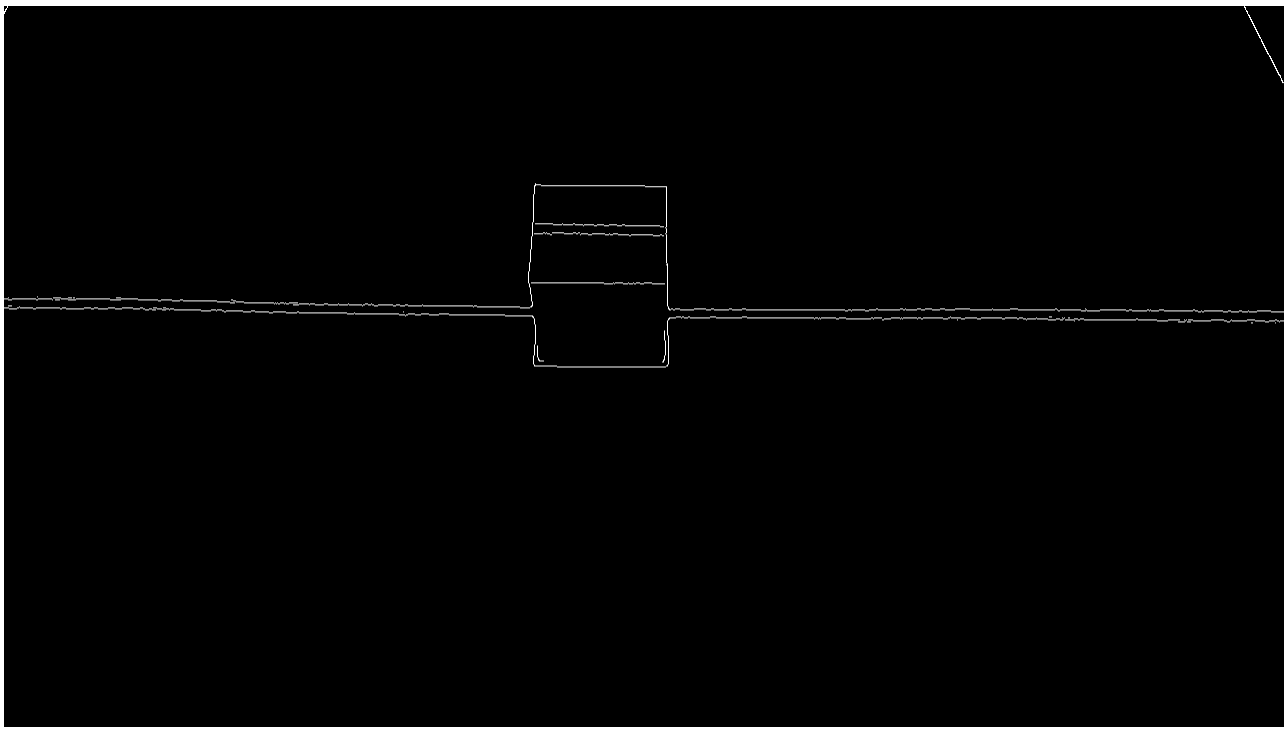

Figure 16: Result of using the Canny edge detector on Figure 15, with low threshold of 84 and high threshold of 140

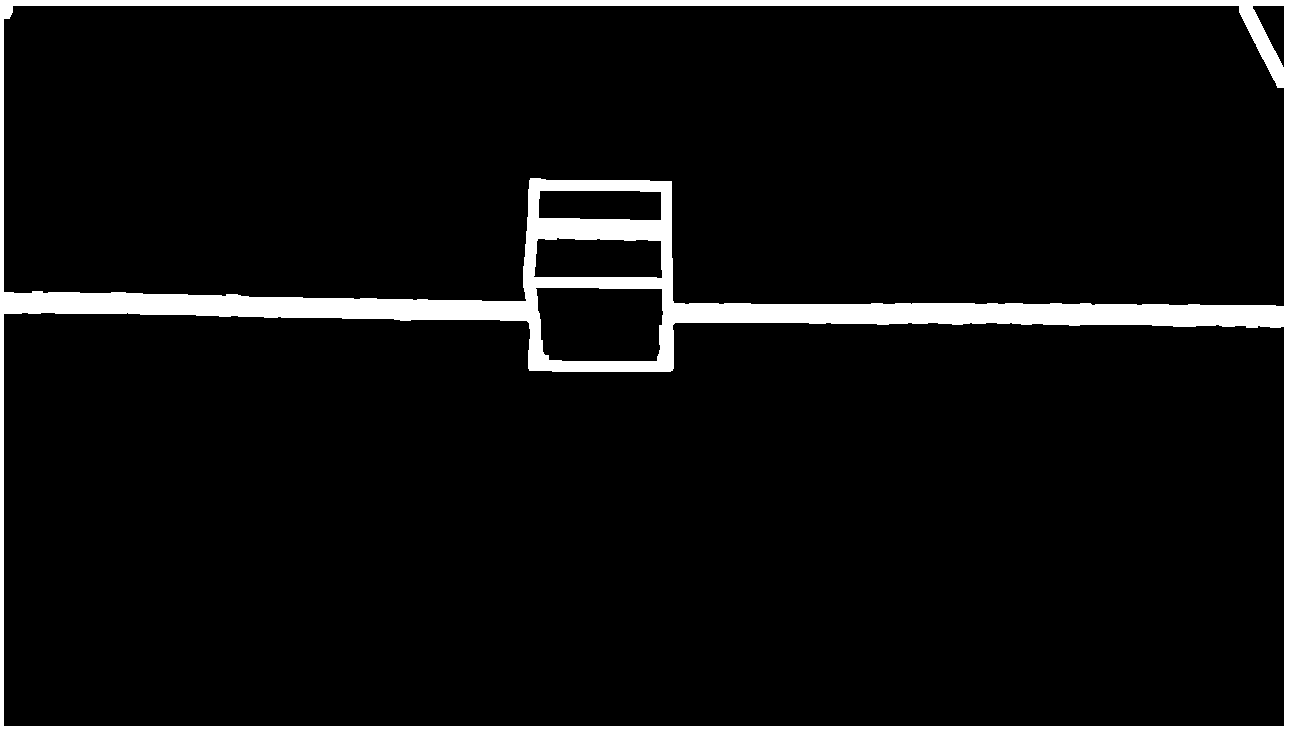

Figure 17: Result of applying dilation with a square kernel of size 11 on Figure 16

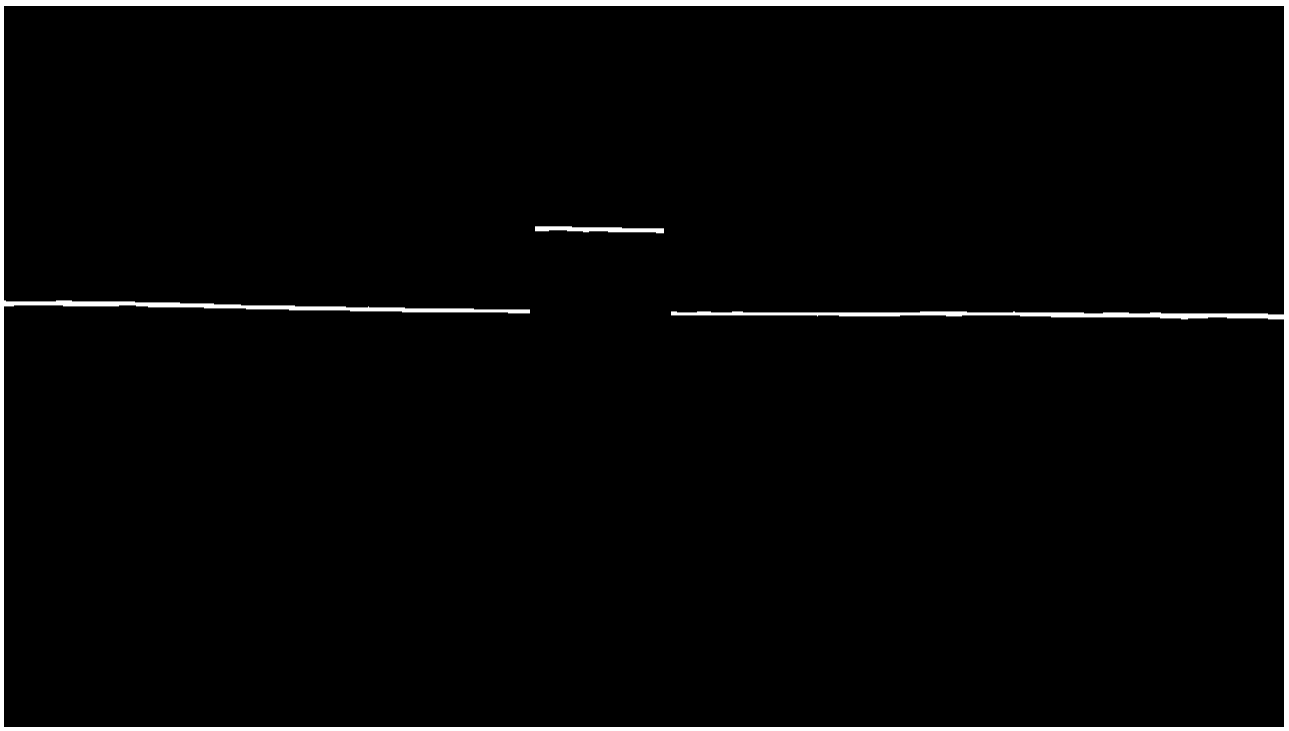

Figure 18: Result of applying erosion with a square kernel of size 17 on Figure 17

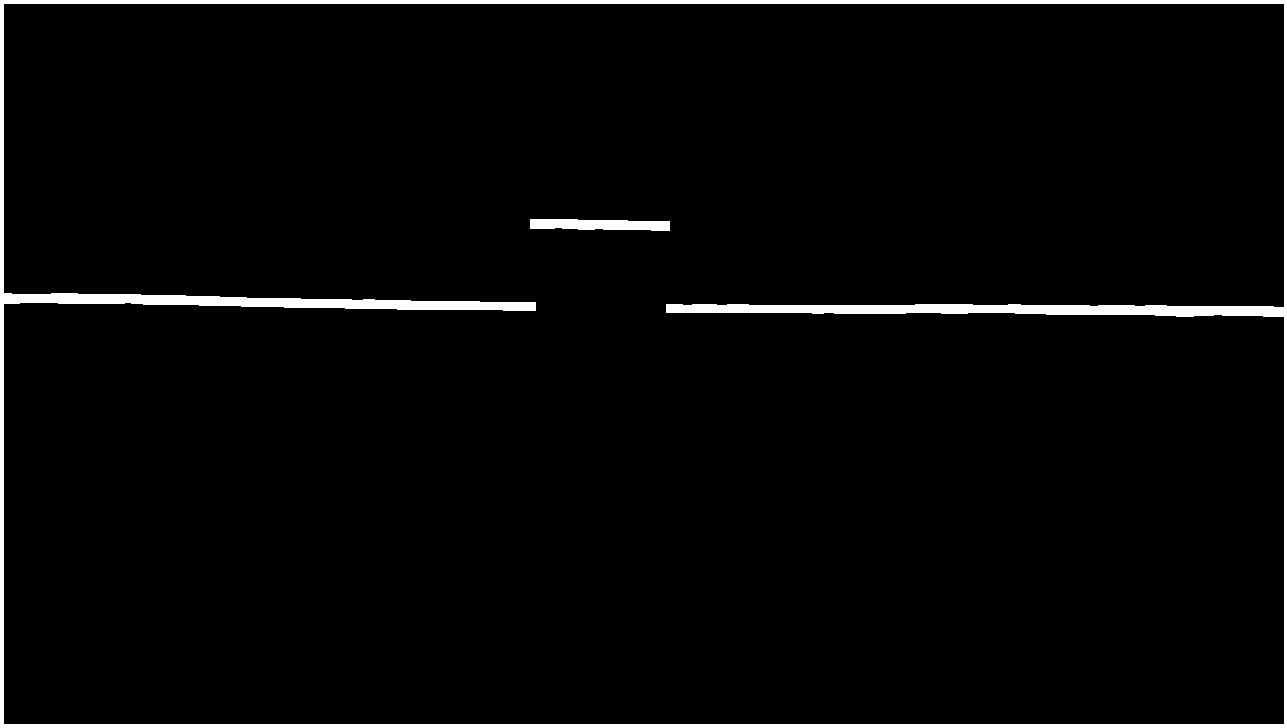

Figure 19: Result of applying grown the Figure 18 upwards with a dilation

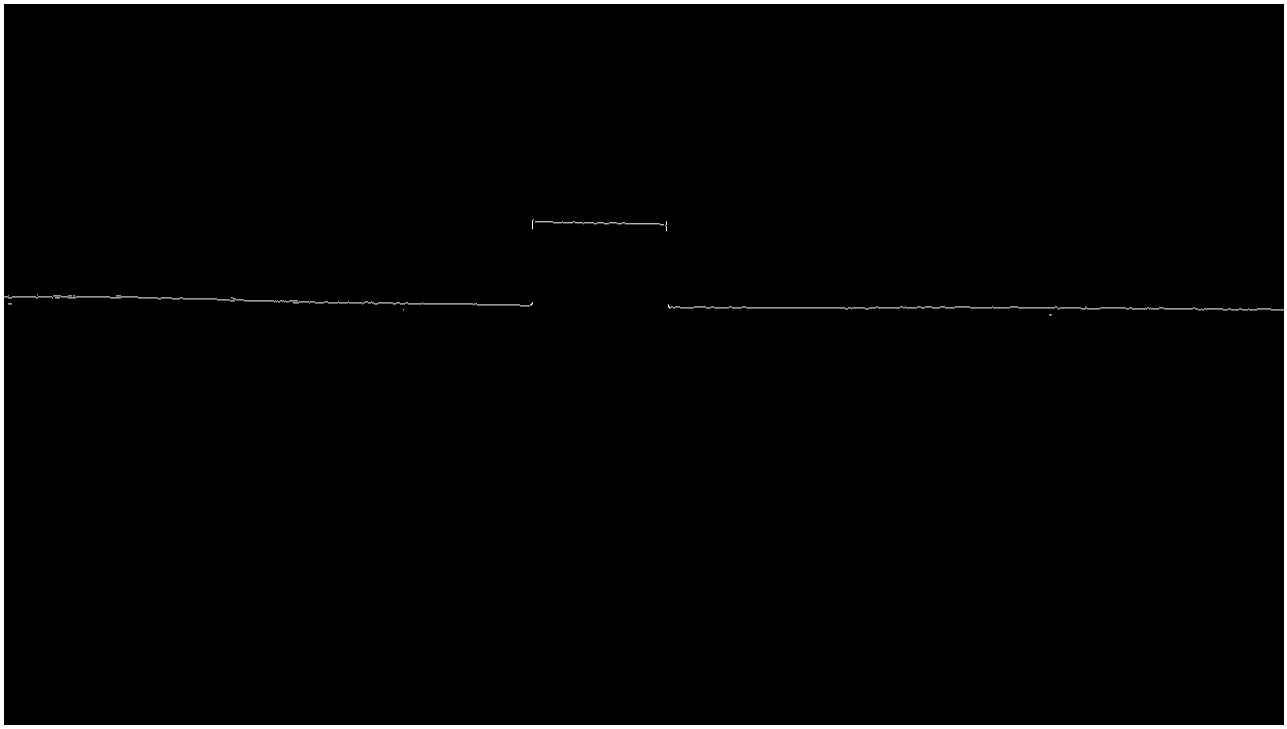

Figure 20: Result of a *bitwise and* between Figure 19 and Figure 16

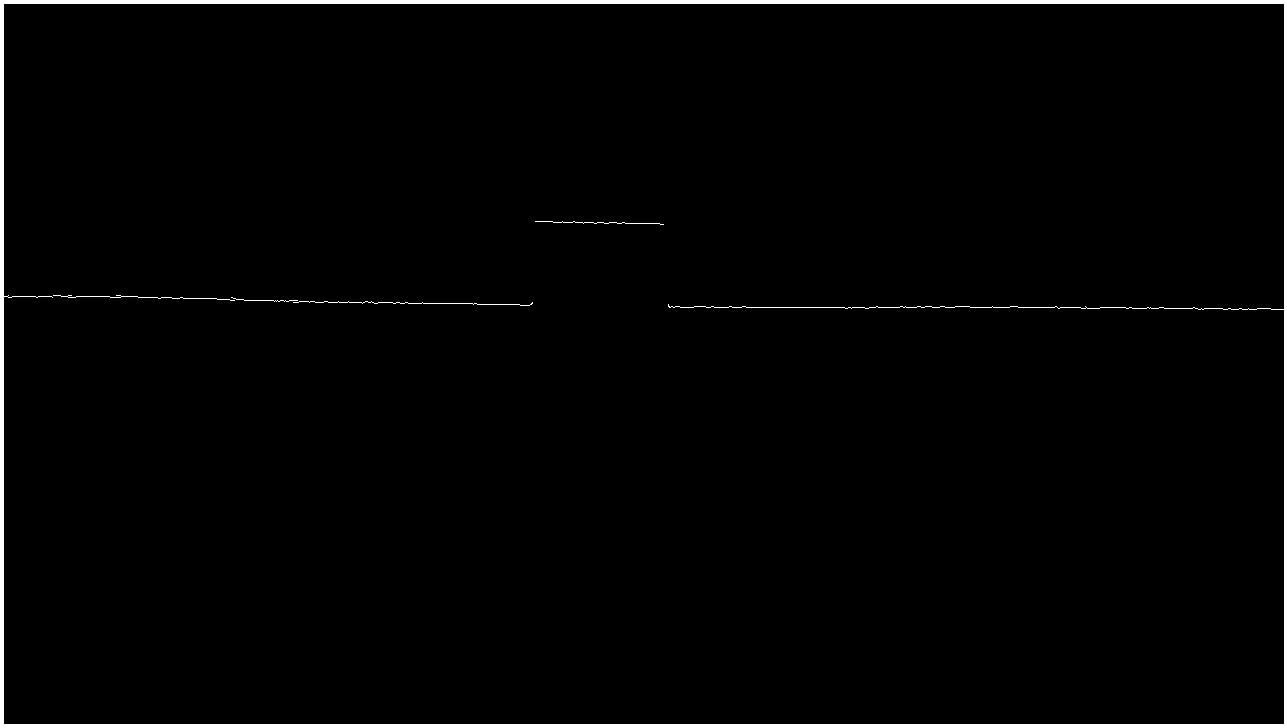

Figure 21: Final result of the shadow extraction step on Figure 15, result of removing connected components with less than 11 pixels from Figure 20

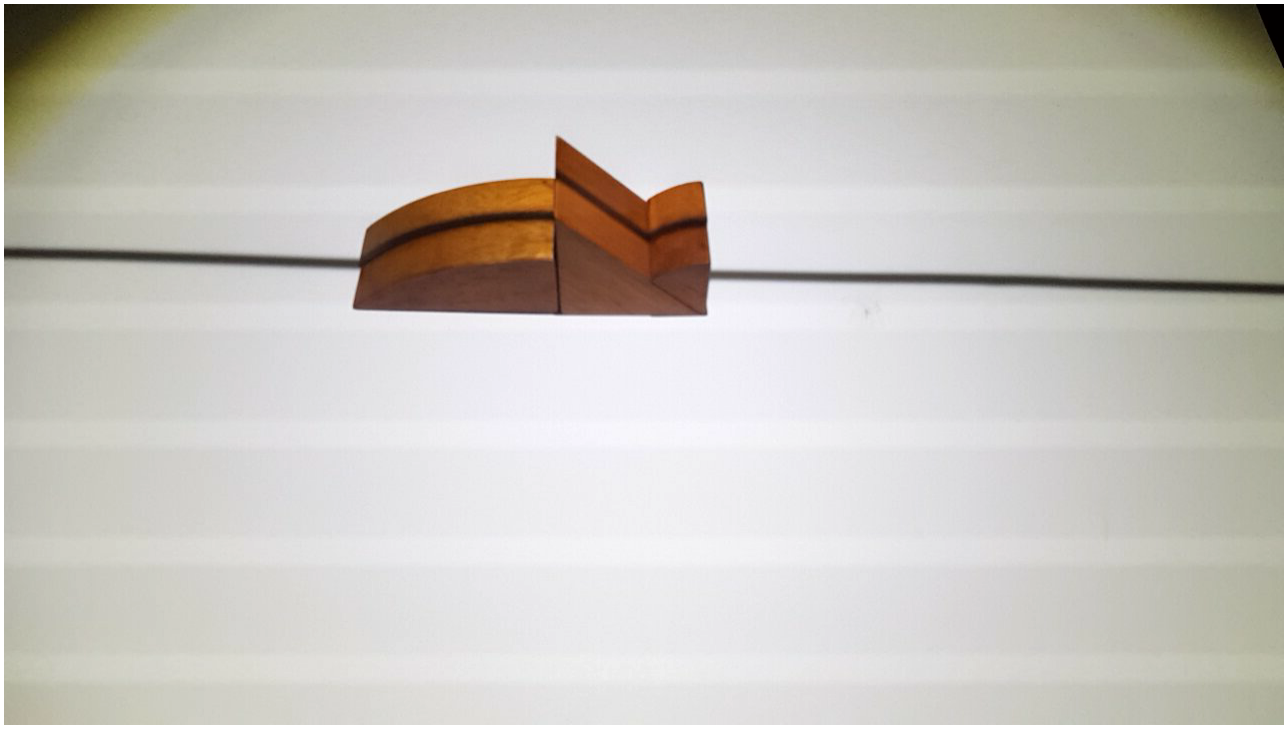

Figure 22: Captured image of a more complex object with our system

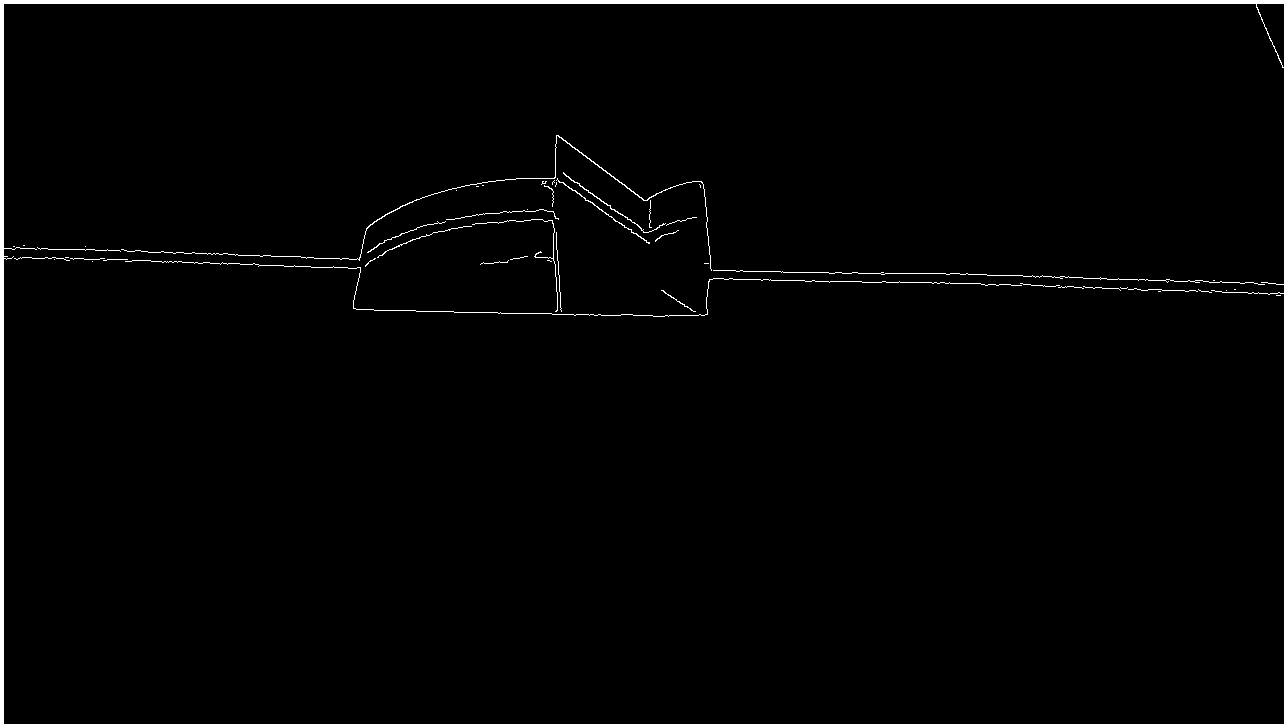

Figure 23: Result of using the Canny edge detector on Figure 22, with low threshold of 72 and high threshold of 115

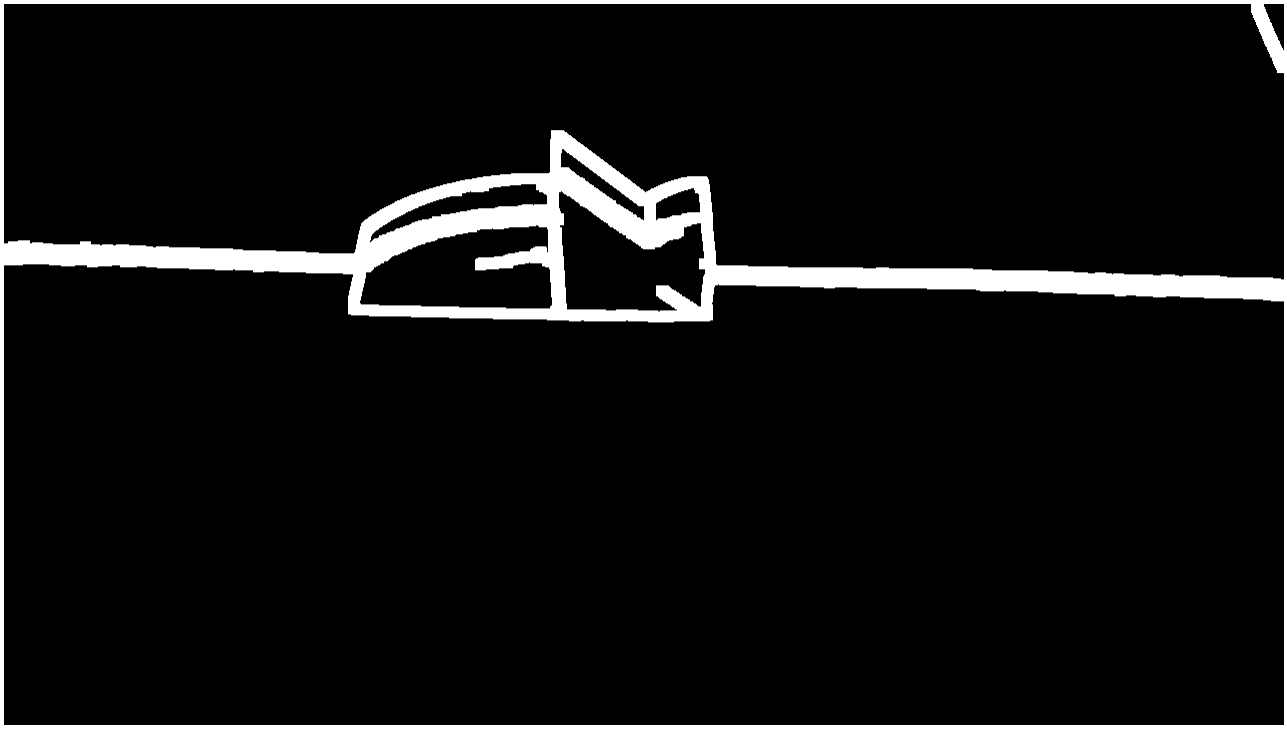

Figure 24: Result of applying dilation with a square kernel of size 11 on Figure 23

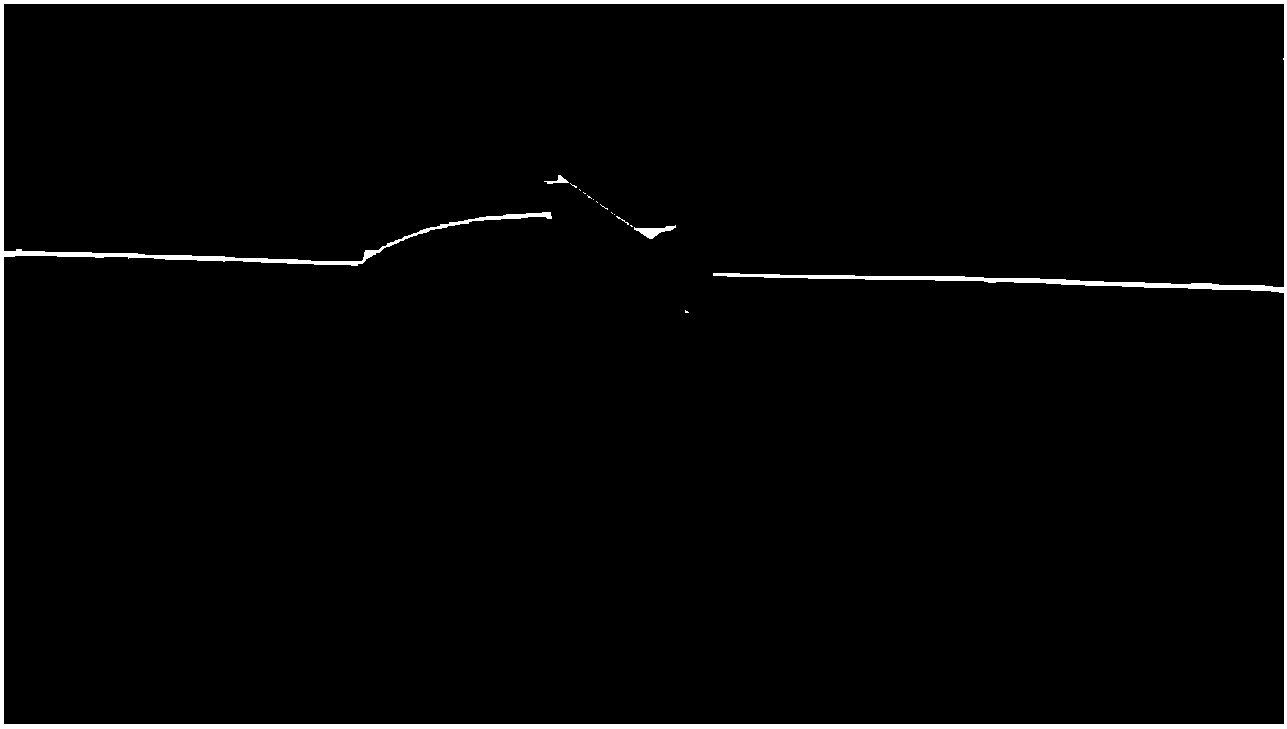

Figure 25: Result of applying erosion with a square kernel of size 17 on Figure 24

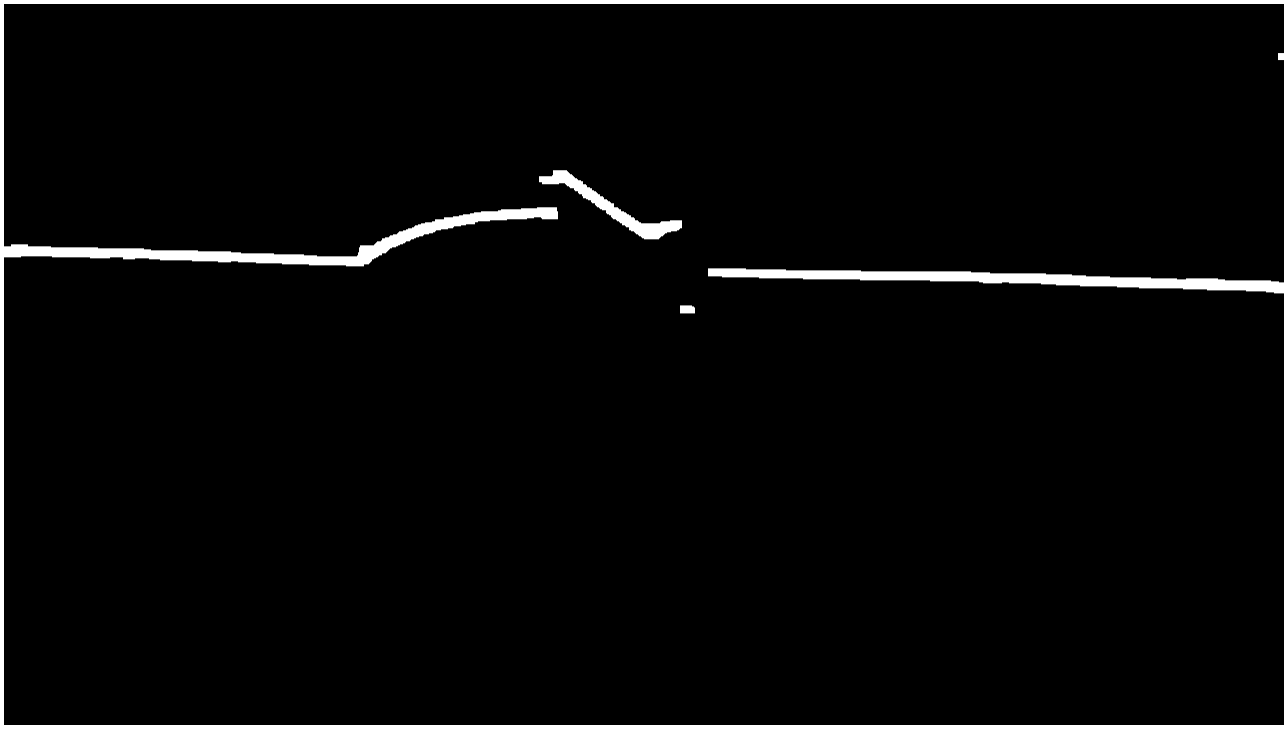

Figure 26: Result of applying grown the Figure 25 upwards with a dilation

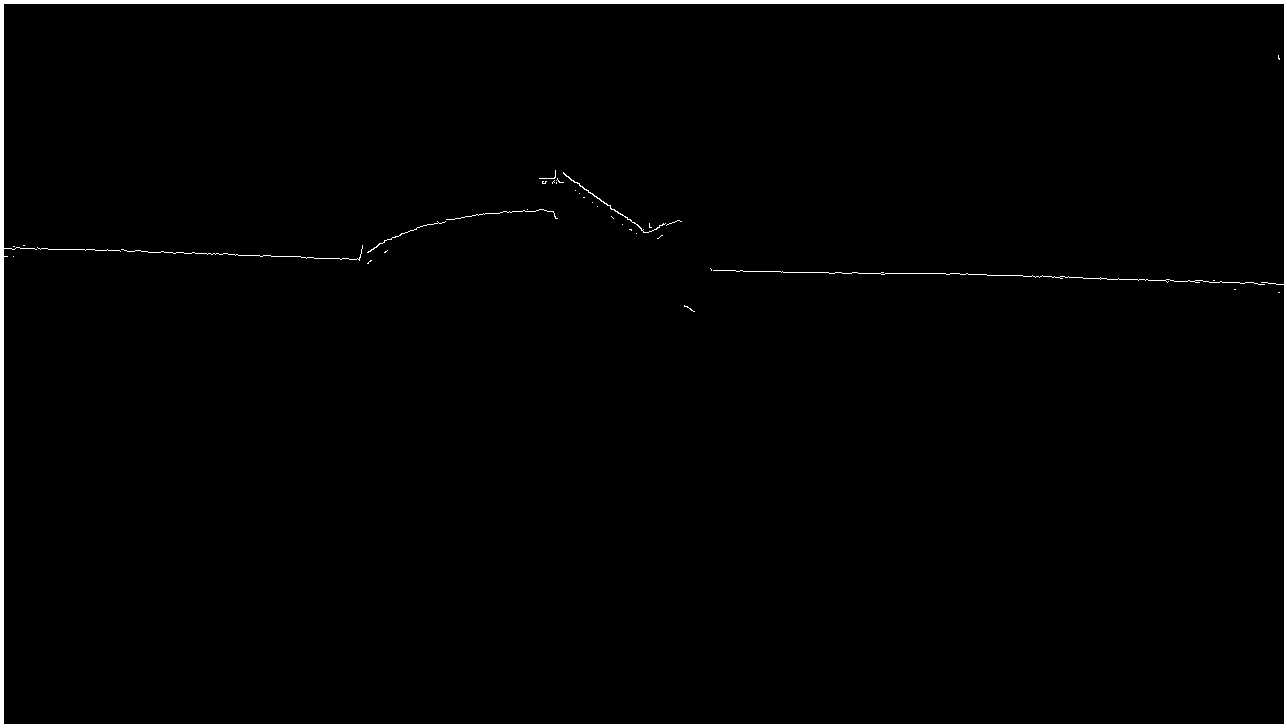

Figure 27: Result of a *bitwise and* between Figure 26 and Figure 23

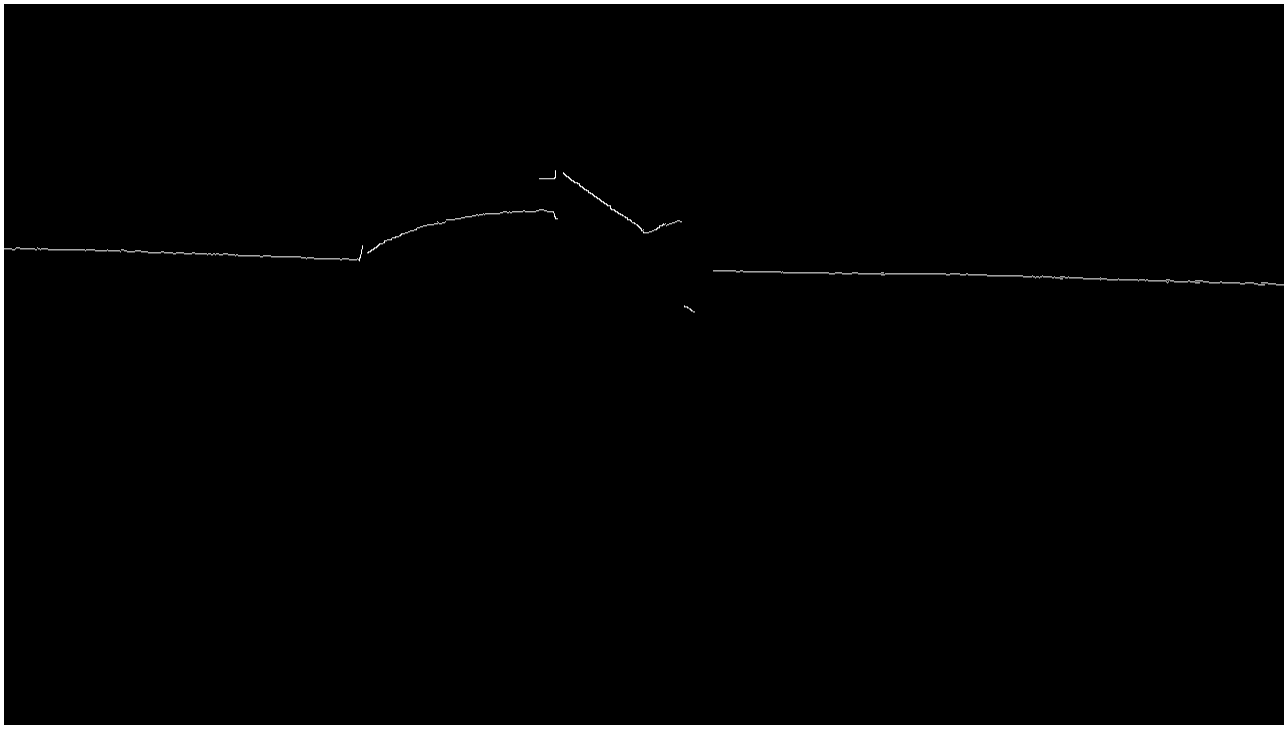

Figure 28: Final result of the shadow extraction step on Figure 22, result of removing connected components with less than 12 pixels from Figure 27

B 3D point calculation - results for cube

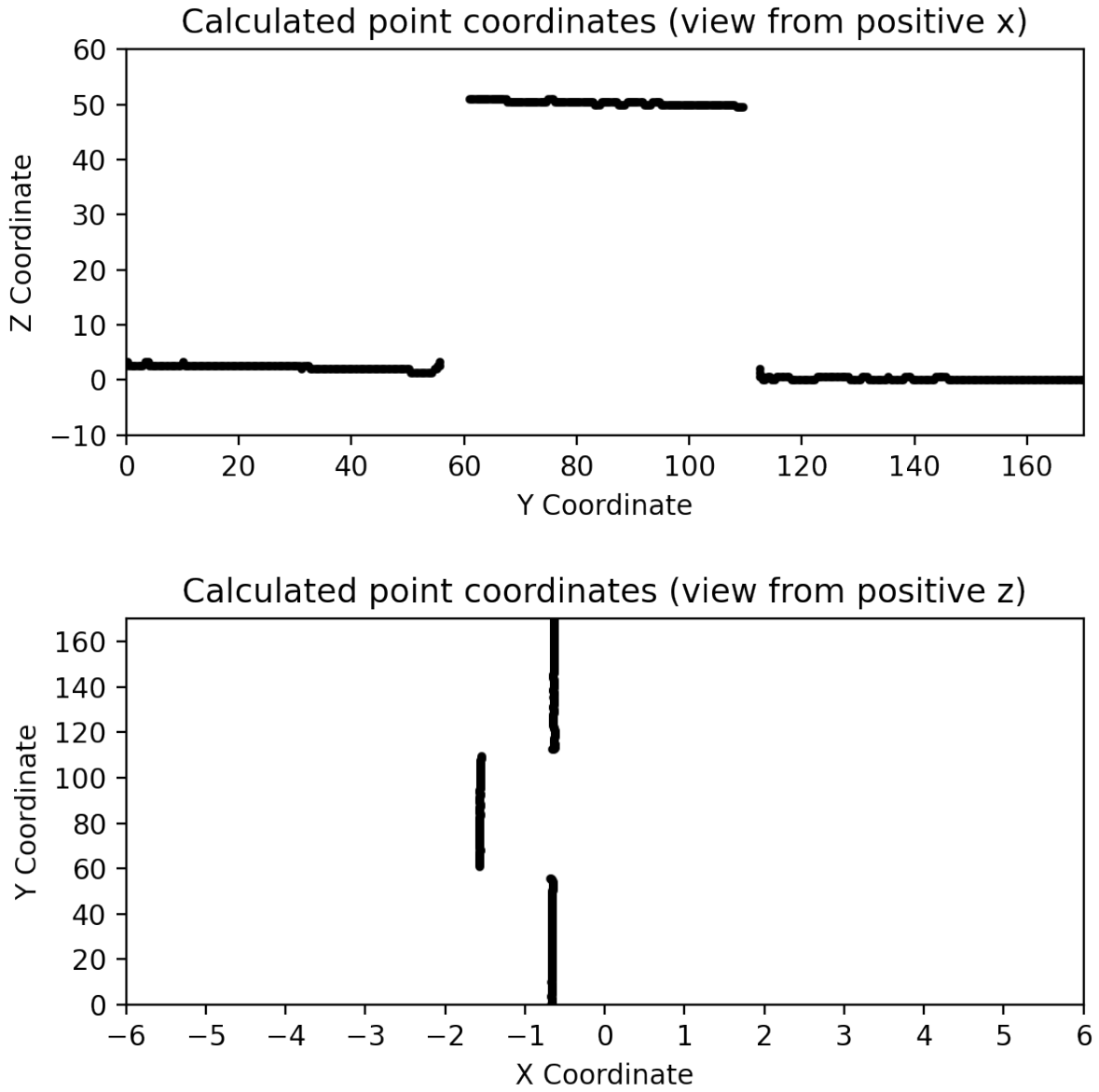

Figure 29: Final result of calculating the 3D coordinates of a cube with 5cm height.

C script.py (better viewed on the experiments.ipynb notebook)

```
# To add a new cell, type '# %%'
# To add a new markdown cell, type '# %% [markdown]'
# %%
import numpy as np
import cv2
from functions import *
from edge_detection import *
from matplotlib import pyplot as plt


# %%
WORKING_FOLDER = 'data/calibration/david/20-04-2021'
INTRINSIC_PATH = 'data/calibration/david/intrinsic'
RAW_TARGET_OBJECT_IMAGE_NAME = 'frame_2021-04-18␣16:50:52.088963.png'
RAW_CALIBRATING_OBJECT_IMAGE_NAME = 'frame_2021-04-18␣16:51:45.830721.png'
CHESSBOARD_SQUARE_LENGTH_MM = 24
CHESSBOARD_DIMENSIONS = (9,6, CHESSBOARD_SQUARE_LENGTH_MM)
PLANE_CALIBRATION_OBJECT_HEIGHT_MM = 50
UNC_COMPONENTS_MASK_SIZE = 8
```

```
# Configure plt plot sizes for notebook
plt.rcParams['figure.figsize'] = [6, 6]
plt.rcParams['figure.dpi'] = 200

# %% [markdown]
# ## Intrinsic parameter calibration

# %%
object_points, image_points, image_dimensions = chessboardPointExtraction(CHESSBOARD_DIMENSIONS,
    INTRINSIC_PATH)
ret, intrinsic_matrix, distortion_coefs, rotation_vecs, translation_vecs = cv2.calibrateCamera(
    object_points, image_points, image_dimensions,None,None)
print('\nImage Dimensions\n', image_dimensions)
print('\nIntrinsic Matrix\n', intrinsic_matrix)
print('\nDistortion Coefficients\n', distortion_coefs)
print(f'\nReprojection error: {calculateReprojectionError(object_points, image_points,
    rotation_vecs, translation_vecs, intrinsic_matrix, distortion_coefs)}')

# %% [markdown]
# ## Extrinsic parameter calibration

# %%
object_points, image_points, _ = chessboardPointExtraction(CHESSBOARD_DIMENSIONS, f'{WORKING_FOLDER
    }/extrinsic')
object_points_reshaped = np.array(object_points).reshape((-1,3))
image_points_reshaped = np.array(image_points).reshape((-1,2))

# Calculate extrinsic parameter matrices (translation and rotation) using PnP RANSAC
ret, rotation_vecs, translation_vecs, _ = cv2.solvePnPRansac(object_points_reshaped,
    image_points_reshaped, intrinsic_matrix, distortion_coefs)
print(f'Reprojection error: {calculateReprojectionError(object_points, image_points, rotation_vecs,
    translation_vecs, intrinsic_matrix, distortion_coefs)}')

# %% [markdown]
# ## Projection Calculation

# %%
perspective_projection_matrix = calculatePpmMatrix(intrinsic_matrix, rotation_vecs,
    translation_vecs)

# %% [markdown]
# ## Shadow/Light Plane Calibration

# %%

plane_calib_img_raw = cv2.imread(f'{WORKING_FOLDER}/{RAW_CALIBRATING_OBJECT_IMAGE_NAME}',cv2.
    IMREAD_GRAYSCALE)

# Extract shadow points from raw image (shadow points in white)
plane_calib_img_processed, _ = extract_shadow_points_auto(plane_calib_img_raw)

# Split the bottom plane shadow and the top plane shadow
processed_base_plane_points, processed_elevated_plane_points = splitTopBottomPoints(
    plane_calib_img_processed)

fig, subplots = plt.subplots(1,2)
subplots[0].set_title('Proc. base plane points')
subplots[0].imshow(processed_base_plane_points, cmap='gray' , vmin=0, vmax=255)

subplots[1].set_title('Proc. elevated plane points')
subplots[1].imshow(processed_elevated_plane_points, cmap='gray', vmin=0, vmax=255)


# %%
# 'Floor' points
floor_plane_constraint = [0,0,1,0] # z=0
base_points = getWhitePoint3DCoords(processed_base_plane_points, [floor_plane_constraint],
    perspective_projection_matrix)

# Object top points
```

```
elevated_plane_constraint = [0,0,1,PLANE_CALIBRATION_OBJECT_HEIGHT_MM]
elevated_points = getWhitePoint3DCoords(processed_elevated_plane_points, [elevated_plane_constraint
    ], perspective_projection_matrix)

plane_3d_points = list(base_points)
plane_3d_points.extend(elevated_points)


# %%
# Calculate plane coeficients
plane_coefs = calculatePlaneCoefs(np.array(plane_3d_points))
plane_coefs

# %% [markdown]
# ## Target object point extraction

# %%
target_object_points_raw = cv2.imread(f'{WORKING_FOLDER}/{RAW_TARGET_OBJECT_IMAGE_NAME}')
target_object_points_raw_gray = cv2.cvtColor(target_object_points_raw, cv2.COLOR_BGR2GRAY)

# Detect shadow line
target_object_points_processed, _ = extract_shadow_points_auto(target_object_points_raw_gray)

# Remove unconnected components from processed image (aimed at removing some of the noise)

plt.title('Target␣object␣shadow/light␣points')
plt.imshow(target_object_points_processed, cmap='gray')


# %%
# Calculate y and z for points in shadow
points = getWhitePoint3DCoords(target_object_points_processed, [plane_coefs],
    perspective_projection_matrix)

points_x = [point[0] for point in points]
points_y = [point[1] for point in points]
points_z = [point[2] for point in points]


# %%
figure, subplots = plt.subplots(2)
subplots[0].set_title('Calculated␣point␣coordinates␣(view␣from␣positive␣x)')
subplots[0].set_xlabel('Y␣Coordinate')
subplots[0].set_ylabel('Z␣Coordinate')
subplots[0].set_xlim([0,170])
subplots[0].set_ylim([-10,60])
subplots[0].set_xticks(np.arange(0, 180, 20))
subplots[0].set_yticks(np.arange(-10, 70, 10))
subplots[0].scatter(points_y, points_z, color='black', marker='.', linewidths=0.1)

subplots[1].set_title('Calculated␣point␣coordinates␣(view␣from␣positive␣z)')
subplots[1].set_xlabel('X␣Coordinate')
subplots[1].set_ylabel('Y␣Coordinate')
subplots[1].set_xlim([-6,6])
subplots[1].set_ylim([0,170])
subplots[1].set_xticks(np.arange(-6, 7, 1))
subplots[1].set_yticks(np.arange(0, 180, 20))
subplots[1].scatter(points_x, points_y, color='black', marker='.', linewidths=0.1)
figure.tight_layout(pad=2)


# %%
```

D functions.py

```
import cv2
import numpy as np
import glob
from tqdm import tqdm
from sklearn.linear_model import RANSACRegressor, LinearRegression

# Find the chessboard corners subpixel coordinates of a given set of images


def chessboardPointExtraction(chessboard_dimensions, frame_path):
    # termination criteria
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

    # prepare object points, like (0,0,0), (0,1,0), (0,2,0) ....,(5,6,0). The coordinates are
        adjusted according to the chessboard square side length
    sample_object_points = np.zeros(
        (chessboard_dimensions[0] * chessboard_dimensions[1], 3), np.float32)

    sample_object_points[:, :2] = np.mgrid[0:chessboard_dimensions[0],
                                           0:chessboard_dimensions[1]].T.reshape(-1, 2) *
                                              chessboard_dimensions[2]

    # Swap axis so that the z axis is perpendicular to the chessboard
    sample_object_points[:, [1, 0]] = sample_object_points[:, [0, 1]]

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

    images = glob.glob(f'{frame_path}/*.png')
    print(f'Found␣{len(images)}␣calibration␣images')

    for fname in tqdm(images):
        image = cv2.imread(fname)
        grayscale_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        # Find the chess board corners
        ret, corners = cv2.findChessboardCorners(
            grayscale_image, (chessboard_dimensions[0], chessboard_dimensions[1]), None)

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

            # Subpixel location
            improved_corners = cv2.cornerSubPix(
                grayscale_image, corners, (11, 11), (-1, -1), criteria)
            image_points.append(improved_corners)
        else:
            print(f'Could␣not␣find␣chessboard␣corners␣in␣image␣{fname}')
    print('\nCalibration␣over')
    return object_points, image_points, grayscale_image.shape[::-1]


def calculateReprojectionError(objpoints, imgpoints, rvecs, tvecs, imatrix, distortion):
    mean_error = 0
    for i in range(len(objpoints)):

        # Calibrate camera returns several rotation and translation vectors
        # but solvePnP only returns one

        rot_vecs = rvecs if len(rvecs) == 3 else rvecs[i]
        tra_vecs = tvecs if len(tvecs) == 3 else tvecs[i]

        calculted_image_points, _ = cv2.projectPoints(
            objpoints[i], rot_vecs, tra_vecs, imatrix, distortion)

        error = cv2.norm(imgpoints[i], calculted_image_points,
                         cv2.NORM_L2)/len(calculted_image_points)
```

```
        mean_error += error

    return mean_error/len(objpoints)

# Get xyz coordinates of an i,j image point given a perpective projection matrix and a set of
    aditional constraints
# Constraints is a list where each element is a list [A,B,C,D] representing a constraint of type Ax
    + By + Cz = D


def get_xyz_coords(i, j, ppm, constraints=[[1, 0, 0, 0]]):
    k1 = ppm[2] * i
    k2 = ppm[2] * j
    k3 = k1 - ppm[0]
    k4 = k2 - ppm[1]
    a = [k3[0:-1], k4[0:-1]]
    b = [-k3[-1], -k4[-1]]

    for constraint in constraints:
        a.append(np.array(constraint[0:-1]))
        b.append(np.array(constraint[-1]))
    res = np.linalg.solve(a, b)

    return res


def calculatePpmMatrix(intrinsic_matrix, rotation_vecs, translation_vecs):

    # rotation_vecs is in Rodrigues forms, and needs to be converted to a 3x3 matrix
    rotation_matrix = cv2.Rodrigues(rotation_vecs)[0]

    # create [R|T] matrix
    extrinsic_matrix = np.concatenate(
        (rotation_matrix, translation_vecs), axis=1)

    # multiply the intrinsic and extrinsinc matrix
    perspective_projection_matrix = intrinsic_matrix @ extrinsic_matrix

    return perspective_projection_matrix


# The points that are used to calculate the shadow plane are in different z planes. This function
# only allows the regression to be done with points that are not on the same z plane. Ideally, the
# meausred points with the same z coordinate should be colinear and, thus, shouldn't be considered,
# but to to imperfections in the 3D coordinate this is not the case.
def validateData(_, b):
    return not(b[0] == b[1] and b[1] == b[2])

# Calculate the coeffiecents of a plane that fits the given <points>. The estimation is done using
# RANSAC. The algorithm expects points in to different z planes. The coeficients are returned in
    the
# form [A,B,C,D] where the plane is of the form Ax+BY+CZ = -D


def calculatePlaneCoefs(points):
    xy = points[:, :2]
    z = points[:, 2]

    # estimate Ax + By + D = Z (C = 1)
    # validateData ensures that points used to estimate the plane are in different Z planes
    # without this the algorithm was considering all points in other planes as outliers
    reg = RANSACRegressor(base_estimator=LinearRegression(fit_intercept=True),
                          is_data_valid=validateData,
                          residual_threshold=0.01,
                          max_trials=1000).fit(xy, z)

    return list(np.append(reg.estimator_.coef_, [-1, -reg.estimator_.intercept_]))


# Calculate the 3D coordinates of the write 2D <points>, given a set of extra constraints (see
    get_xyz_coords) and a
```

```
# perspective projection matrix
def getWhitePoint3DCoords(points, constraints, ppm):
    white_pixel_coords = cv2.findNonZero(points)
    res = [get_xyz_coords(pixel[0, 0], pixel[0, 1], ppm, constraints)
           for pixel in white_pixel_coords]

    return np.array(res)


def validateDataLine(intercept):
    def validateData(model, a, b):
        return abs(intercept - model.intercept_) > 25

    return validateData


# Calculate straight line that fits <points>. Returns the slope, y intercept and inlier mask
def calculateLineCoefs(points, is_model_valid=None):
    x = points[:, :1]
    y = points[:, 1]

    # estimate Ax + B = y
    reg = RANSACRegressor(base_estimator=LinearRegression(fit_intercept=True),
                          is_model_valid=is_model_valid
                          ).fit(x, y)

    return reg.estimator_.coef_, reg.estimator_.intercept_, reg.inlier_mask_


def splitTopBottomPoints(src):
    points = cv2.findNonZero(src)

    # Fit line to either the top or bottom points
    _, intercept_first, inliers = calculateLineCoefs(np.reshape(points, (-1, 2)))

    # Extract first line points
    first_set = np.zeros_like(src, dtype=np.uint8)
    for pt in points[inliers]:
        first_set[pt[0, 1], pt[0, 0]] = 255

    points_top = points[np.logical_not(inliers)]

    # Fit line to remaining points
    _, intercept_second, inliers = calculateLineCoefs(np.reshape(
        points_top, (-1, 2)), is_model_valid=validateDataLine(intercept_first))

    # Extract second line points
    second_set = np.zeros_like(src, dtype=np.uint8)
    for pt in points_top[inliers]:
        second_set[pt[0, 1], pt[0, 0]] = 255

    # The bottom line will have a lower intercept than the top one
    if intercept_first < intercept_second:
        top = first_set
        bottom = second_set
    else:
        top = second_set
        bottom = first_set

    return bottom, top
```

E edge_detection.py

```
import os
import math
import argparse
import cv2 as cv
import numpy as np
from functions import splitTopBottomPoints


def extract_shadow_points(img, high_thresh, low_thresh, dilate, erode, after_canny=None,
    after_dilate=None):
    # Size of the Sobel kernel used to compute the derivatives in the Canny edge detector
    aperture = 3

    # Apply Canny edge detector
    img_canny = after_canny
    if after_dilate is None and after_canny is None:
        img_canny = cv.Canny(img, low_thresh, high_thresh,
                             apertureSize=aperture)

    # Dilate with a square kernel of size "dilate", in attempt to join both edges of the shadow
    img_dilated = after_dilate
    if after_dilate is None:
        img_dilated = cv.morphologyEx(
            img_canny, cv.MORPH_DILATE, np.ones((dilate, dilate)))

        # Erode with a square kernel of size "erode", in attempt to remove every white pixel,
            except for the ones in the center of the shadow
    img_morph = cv.morphologyEx(img_dilated, cv.MORPH_ERODE,
                                np.ones((erode, erode)))

    # Apply a dilation operation to the center of the shadow obtained in the previous step. The
        kernel
    # used is a square kernel, larger than the used in the previous dilate operation by 1, with
        only
    # the elements of the bottom half of the kernel set to 1, and the rest to 0.
    #
    # Example:
    # 0 0 0         0 0 0 0
    # 0 0 0   or    0 0 0 0
    # 1 1 1         1 1 1 1
    #               1 1 1 1
    #
    # Applying this dilation results in the "growth" of the detected line upwards.
    kernel_size = dilate + 1

    # Obtain kernel
    kernel = np.zeros((kernel_size, kernel_size), dtype=np.uint8)
    kernel[int(kernel_size/2):, :] = 1

    # Compute dilation with the obtained kernel
    img_dilate_up = cv.morphologyEx(img_morph, cv.MORPH_DILATE, kernel)

    # Compute the final result by considering only the pixels that are both white in the original
        output
    # of the canny edge detector and white in the image obtained in the previous step. This selects
         pixels
    # that are from the "top" edge of the shadow,
    # which are the extracted shadow points used in later stages.
    final_res = cv.bitwise_and(img_canny, img_dilate_up)

    return final_res, img_dilate_up, img_morph, img_dilated, img_canny


def remove_unconnected_points(img, area_threshold):
    # Find connected components from the image and the number of pixels in each of the components

    connectivity = 8  # find 8-way connected components

    num_labels, labels_im, stats, _ = cv.connectedComponentsWithStats(
```

```
        img, connectivity, stats=cv.CC_STAT_AREA)

    # remove the components with less than "area_threshold" pixels
    for i in range(num_labels):
        if stats[i, cv.CC_STAT_AREA] < area_threshold:
            img[labels_im == i] = 0

    return img


def evaluate(img):
    """
    Compute score of the obtained shadow points.
    The score consists of the the summation of, for each column of the image, the absolute value of
        the
    number of white points minus 1
    """
    score = 0
    for i in range(img.shape[1]):
        score += abs(np.count_nonzero(img[:, i]) - 1)
    return score


def extract_shadow_points_auto(img):
    """
    Extract shadow points from image, by automatically finding set of hyperparameters that obtain a
        good
    result according to the evaluate function above.
    """
    best = {
        'params': {
            'low_threshold': 0,
            'high_threshold': 0,
            'dilate': 0,
            'erode': 0,
        },
        'steps': {
            'after_canny': np.zeros_like(img),
            'after_dilate': np.zeros_like(img),
            'after_morph': np.zeros_like(img),
            'after_dilate_up': np.zeros_like(img),
        },
        'result': np.zeros_like(img),
        'score': math.inf
    }

    # high threshold values
    for ht in range(100, 150, 5):
        # low threshold values
        for lt in range(int(0.6*ht), int(0.9*ht), 3):
            after_canny = None
            # dilation kernel size values
            for d in range(9, 15, 2):
                after_dilate = None
                # erosion kernel size values
                for e in range(d+4, d+8, 2):

                    # apply shadow points extraction algorithm
                    res, after_dilate_up, after_morph, after_dilate, after_canny =
                        extract_shadow_points(
                        img, ht, lt, d, e, after_canny=after_canny, after_dilate=after_dilate)

                    # evaluate shadow points and see if it's the best so far
                    curr_value = evaluate(res)

                    if curr_value < best['score']:
                        best['params']['dilate'] = d
                        best['params']['erode'] = e
                        best['params']['high_threshold'] = ht
                        best['params']['low_threshold'] = lt
                        best['score'] = curr_value
```

```
                        best['result'] = res
                        best['steps']['after_canny'] = after_canny
                        best['steps']['after_morph'] = after_morph
                        best['steps']['after_dilate'] = after_dilate
                        best['steps']['after_dilate_up'] = after_dilate_up

    return best


######
##
# Auxiliary functions for manually tweaking the hyperparameters, using trackbars
##
######
edge_detection_window = 'Step 1. Canny edge detector'
morph_window = 'Step 2. Dilate -> Erode'
dilate_up_window = 'Step 3. Dilate up'
final_window = 'Final Result'


def display(final_result, after_dilate_up, after_morph_ops, after_dilate, after_edge_detection):
    cv.imshow(edge_detection_window, after_edge_detection)
    cv.imshow(morph_window, after_morph_ops)
    cv.imshow(dilate_up_window, after_dilate_up)
    cv.imshow(final_window, final_result)


# Initial values for the trackbars
low_threshold = 100
high_threshold = 150
dilate = 10
erode = 18


def on_low_threshold(v):
    global low_threshold
    low_threshold = v


def on_high_threshold(v):
    global high_threshold
    high_threshold = v


def on_dilate(v):
    global dilate
    dilate = v


def on_erode(v):
    global erode
    erode = v


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Extract shadow points.')
    parser.add_argument('--path', dest='img_path',
                        type=str, help='Path to image.')
    parser.add_argument('--auto', action='store_true',
                        help='whether to find parameters automatically or use a trackbar to define
                            them manually')

    args = parser.parse_args()
    print(args)
    if args.img_path is None:
        img_path = 'data/cube.png'
    else:
        img_path = args.img_path

    name = os.path.splitext(os.path.basename(img_path))[0]
    img = cv.imread(
```

```
    img_path, cv.IMREAD_GRAYSCALE)

if args.auto:
    best = extract_shadow_points_auto(img)

    low_threshold = best['params']['low_threshold']
    high_threshold = best['params']['high_threshold']
    dilate = best['params']['dilate']
    erode = best['params']['erode']
    final_res = best['result']

    img_canny = best['steps']['after_canny']
    img_morph = best['steps']['after_morph']
    img_dilated = best['steps']['after_dilate']
    img_dilate_up = best['steps']['after_dilate_up']

    cv.imshow(final_window, best['result'])
    print(best['params'])

    top, bottom = splitTopBottomPoints(final_res)

    cv.imwrite('{}_final_result_{}_{}_{}_{}.png'.format(name,
                                                        low_threshold, high_threshold, dilate,
                                                            erode), final_res)
    cv.imwrite('{}_after_canny_{}_{}_{}_{}.png'.format(name,
                                                       low_threshold, high_threshold, dilate,
                                                           erode), img_canny)
    cv.imwrite('{}_after_dil_{}_{}_{}_{}.png'.format(name,
                                                     low_threshold, high_threshold, dilate,
                                                         erode), img_dilated)
    cv.imwrite('{}_after_ero_{}_{}_{}_{}.png'.format(name,
                                                     low_threshold, high_threshold, dilate,
                                                         erode), img_morph)
    cv.imwrite('{}_after_dil_up_{}_{}_{}_{}.png'.format(name,
                                                        low_threshold, high_threshold, dilate,
                                                            erode), img_dilate_up)

    cv.imwrite('top.png', top)
    cv.imwrite('bottom.png', bottom)
    cv.waitKey(0)
    exit()

# Max values for the trackbars
morph_max = 30
threshold_max = 300

cv.namedWindow(edge_detection_window)
cv.namedWindow(morph_window)
cv.namedWindow(dilate_up_window)
cv.namedWindow(final_window)

# Trackbars for controlling hyperparameters
cv.createTrackbar('Low Threshold', edge_detection_window,
                  low_threshold, threshold_max, on_low_threshold)
cv.createTrackbar('High Threshold', edge_detection_window,
                  high_threshold, threshold_max, on_high_threshold)
cv.createTrackbar('Dilate', morph_window,
                  dilate, morph_max, on_dilate)
cv.createTrackbar('Erode', morph_window,
                  erode, morph_max, on_erode)

while True:
    final_res, img_dilate_up, img_morph, img_dilated, img_canny = extract_shadow_points(
        img, high_threshold, low_threshold, dilate, erode)

    display(final_res, img_dilate_up, img_morph, img_dilated, img_canny)

    key = cv.waitKey(100)
    if key == ord('q'):
        break
    elif key == ord('s'):
```

```
        cv.imwrite('{}_final_result_{}_{}_{}_{}.png'.format(name,
                                                            low_threshold, high_threshold,
                                                               dilate, erode), final_res)
        cv.imwrite('{}_after_canny_{}_{}_{}_{}.png'.format(name,
                                                           low_threshold, high_threshold,
                                                              dilate, erode), img_canny)
        cv.imwrite('{}_after_dil_{}_{}_{}_{}.png'.format(name,
                                                         low_threshold, high_threshold, dilate,
                                                            erode), img_dilated)
        cv.imwrite('{}_after_ero_{}_{}_{}_{}.png'.format(name,
                                                         low_threshold, high_threshold, dilate,
                                                            erode), img_morph)
        cv.imwrite('{}_after_dil_up_{}_{}_{}_{}.png'.format(name,
                                                            low_threshold, high_threshold,
                                                               dilate, erode), img_dilate_up)

cv.destroyAllWindows()
```