CENTROID AND PLANAR ROTATION CALCULUS USING PARALLEL COMPUTATIONAL GEOMETRY FOR REGULAR SHAPED OBJECTS

Alexandru Stan[1] , Diana Maria Cotorobai[1]
[1]National Institute of Research and Development for Mechatronics and Measurement Technique – INCDMTM, Bucharest
Emails: alexandrustan71.sa@gmail.com , diana.cotorobai22@gmail.com

Abstract - Over the last decades, Computer Science has had a large impact on many areas of science, from theoretical studies of algorithms to the practical issues of implementing computing systems in hardware and software. In the era of digitalization, computer science was the starting ramp for this trend and plays an important role for computational algorithms. This paper describes a real-time object detection and tracking using Computational Geometry, embedded within an Autonomous Mobile Waste Collection Robot. This technique is applied on the system for sorting solid waste, perform the grabbing function with a vacuum gripper and release it in the waste container. The algorithm is implemented using Python and OpenCV, a real time optimized computer vision library. It implements a variety of algorithms such as morphological operations, Gaussian image processing, centroid computation and contour algorithms. The main reason for using parallel algorithms is to facilitate the computer to gain high-level understanding from digital images and incorporate it in the system with a high precision, in a short time, using as few processing cores as possible. The results show that the detection works optimal in a supervised and controlled environment in order to avoid erroneous results and the fact that the following solution can match industrial solutions performance-wise.

Keywords: Digital Image Processing, Bitmap Image, Computational Geometry Algorithms, Bilateral Filtering, HSV Colour Space.

1. Introduction

Digital image processing is nowadays a strong tool for solving real world problems in many fields, including engineering.

The motivation behind this activity is the need to understand and communicate with the computing machines for conceiving programs which facilitates day by day activities. It is a powerful tool for improving the appearance of image and represent it with all the information needed for specific projects. The entire process starts from receiving visual information and then giving out all the description of the scene [1], as illustrated in Figure 1.1 - Image Processing System - IPS, applied to the algorithm developed in the paper.

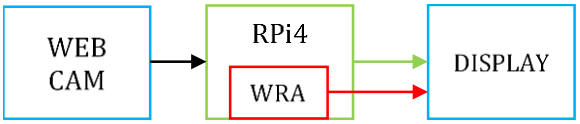

Figure 1.1 – Image Processing System
RPi4 – Raspberry Pie 4 desktop computer;
WRA – Waste Recognition Algorithm

Therefore, the IPS's workflow is as follows:

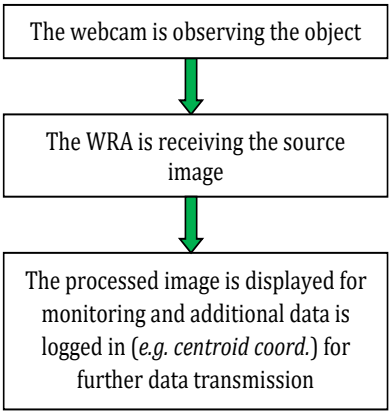

Figure 1.2 – IPS's workflow

Before developing the WRA, the hardware components must be installed within the actual Autonomous Mobile Waste Collection Robot. Therefore, the actual scenery is presented in the following image.

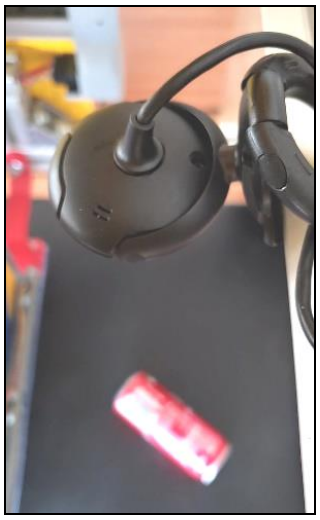

Figure 2.1 – Webcam, provisory position – noncontrolled scenery

As it is shown in Figure 2.1, the black webcam is oriented downwards, normal to the gray wolf colored background. Therefore, the waste in the background can be seen and processed by the webcam and WRA respectively. In this instance, the webcam is connected straight to a computer in order to ease the algorithm's conception.

2. Waste Recognition Algorithm

The WRA has the following tasks:
- Process the observed object in order to output an accurate blob;
- Enclose the processed object within a rectangle;
- Calculate centroid coordinates;
- Calculate angle relative to the Y axis.

In order to obtain the desired result, several image processing methods are used, such as:
- Bilateral filtering;
- Color-space changing;
- Segmentation;
- Morphological transformations;

2.1 HSV Color Space

The WRA will operate within the HSV color-space, Figure 2.1.1, due to its facile color representation and more extensive color information in opposition to BGR or RGB color-spaces.

The HSV model defines a color space in terms of the following components: Hue-Saturation-Value[1].

The colors, or hues, are modeled as an angular dimension rotating around a central, vertical axis, which represents the value channel. Values go from dark (0 at the bottom) to light at the top. The third axis, saturation, defines the shades of hue from least saturated, at the vertical axis, to most saturated furthest away from the center [2].

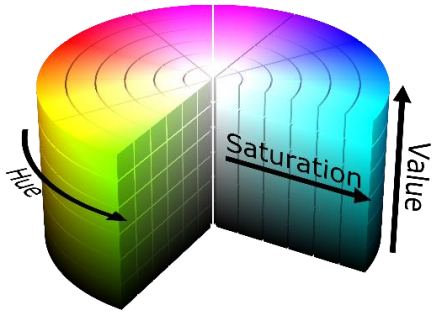

Figure 2.1.1 – HSV Color-space [3]

Therefore, the Hue is represented in degrees ranging from 0° to 360°. In order to fit into the an 8-bit unsigned integer format in OpenCV, the degrees must be divided by two to result a number that resides in the [0, 179] interval, therefore $H \in [0,179]$

The Saturation represents the actual radius of the HSV cylinder, meaning that the bigger the value, the more saturated the color is, and $S \in [0,255]$.

The Value represents amount of emitted light by the color, $V \in [0,179]$ [4].

2.2 WRA's Workflow

In order to obtain the required data, the WRA must run through the following processing stages:

- *IMAGE/VIDEO AQUIRING*

The video feed is acquired by using a core function, cv2.VideoCapture(0) and a transfer variable, *Work_Frame,* is attached to the previous function. The next step is assigning the raw capture to a variable to be processed, *Live_image*:

```
Work_Frame = cv2.VideoCapture(0)
Ret, Live_mage = Work_Frame.read()
```

As result, the algorithm outputs the raw and unprocessed image as shown in the following Figure. 2.2.1:

Figure 2.2.1 – Raw image output

➢ **IMAGE ENHANCEMENT**

Throughout this stage, the original image is enhanced by using a 2D convolution method, the Bilateral Filtering. The purpose of this convolution is to smooth the image in order to ease the recognition process. Bilateral Filtering is the most suited method this particular case for effectively removing noise while keeping object's edges sharp. By using the *cv2.bilateralFilter()* command line, the filter is applied to *Live_image* variable.

Live_image_BF =
cv2.bilateralFilter(Live_image, 15, 100, 100)

As result, after the filtering stage, the algorithm outputs Figure 2.2.3:

Figure 2.2.2 – Raw image output

Figure 2.2.3 – Filtered image output

As it is shown in the above figures, unlike Figure 2.2.2, Figure 2.2.3 has a considerable amount of background noise eliminated, the object itself has sharper edges and the scribblings within the object's perimeter are blurred out.

➢ **CHANGING COLOR SPACE**

As said before, working in HSV color space allows more explicit data to be acquired and processed. In order to change the color space, WRA uses the following function *cv2.cvtColor('source', cv2.COLOR_BGR2HSV)* as follows:

Live_image_HSV =
cv2.cvtColor(Live_image_BF,cv2.COLOR_BGR2HSV)

Next, the algorithm outputs the new variable, *Live_image_HSV, Figure 2.2.4.*

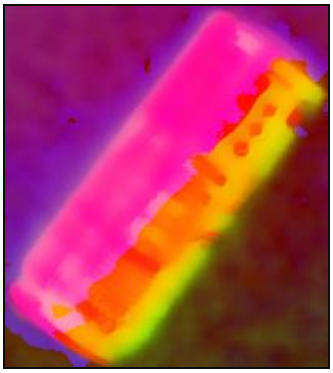

Figure 2.2.4 – Live_image_BF in HSV color space

For the purpose of acquiring the required data, a threshold must be applied to *Live_image_HSV.* Firstly, the lower and upper ranges for each component, Hue, Saturation and Value, must be set.

The function *cv2.inRange* verifies if the array elements of the image lie between the elements of the other two arrays, the lower and upper ranges. For a three-channel input array, the function checks as follows:

$$dst(I) = lowerbH(I)_0 \le src(I)_0 \le upperbH(I)_0 \wedge$$
$$lowerbS(I)_1 \le src(I)_1 \le upperbS(I)_1 \wedge$$
$$lowerbV(I)_2 \le src(I)_2 \le upperbV(I)_2 \wedge$$

Where

- *dst* is the output array of the same size as *src* ;
- *src* is the input array;
- *lowerb* is the inclusive lower boundary array for H, S and V;
- *upperb* is the inclusive upper boundary array for H, S and V.

Therefore, in order to control H, S and V in real-time, a control panel with sliders was created as follows, Figure 2.2.5.

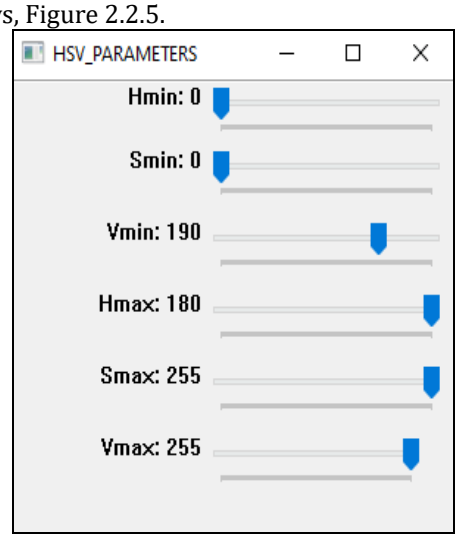

Figure 2.2.5 – HSV components control panel

As Figure 2.2.5 shows, the Hue and Saturation operate in full 8-bit ranges, while Value is limited in the interval [150, 255]. Any value under the lower limit will generate background noise in this particular case. The command lines used for applying a binary threshold are as follows:

```
lower_range =
np.array([Hmin, Smin, Vmin])
upper_range =
np.array([Hmax, Smax, Vmax])

Live_image_HSV_THR =
cv2.inRange(Live_image_HSV, lower_range, upper_range)
```

As result, *Live_image_HSV_THR* outputs a binary image where the required white blob is displayed, Figure 2.2.6.

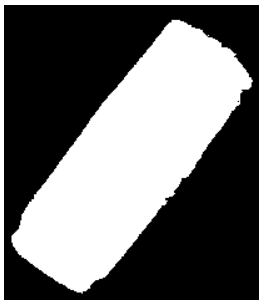

Figure 2.2.6 – Live_image_HSV_THR blob

- **MORPHOLOGICAL TRANSFORMATIONS**

Morphological transformations are simple processes based altogether on binary images. In general, it needs two inputs, the original image and a structuring element, *kernel*, that decides the nature of the operation [5]. These transformations are used to process the binary image in order to obtain the clearest image possible by following the sequence formed by two methods, *Opening* – background processing- and *Closing – foreground processing.*

Beforehand, a *kernel* for *Opening* and *Closing,* respectively, is needed. There are three types of *kernels*: rectangle, elliptical and cross *kernels*. For this particular algorithm, the most suitable *kernel* for the *Opening* is the *rectangle kernel,* while for *Closing* the most suitable *kernel* is the *elliptical kernel.* Therefore, *kernels* are declared as follows:

```
kernel_OPING = cv2.getStructuringElement
(cv2.MORPH_RECT, (8,8)

kernel_CLING = cv2.getStructuringElement
(cv2.MORPH_ELLIPSE, (20,20)
```

The *Opening rectangle kernel* has the following matrix shape:

$$kernelOPING_{8x8} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \quad (1)$$

while the *Closing elliptical kernel* has the following matrix shape:

$$kernelCLING_{20x20} = \begin{pmatrix} 0 & 0 & 1 & \cdots & 1 & 0 & 0 \\ 1 & 1 & 1 & \cdots & 1 & 1 & 1 \\ \vdots & \vdots & \vdots & 1 & \vdots & \vdots & \vdots \\ 1 & 1 & 1 & \cdots & 1 & 1 & 1 \\ 0 & 0 & 1 & \cdots & 1 & 0 & 0 \end{pmatrix} \quad (2)$$

Firstly, the *Opening* method is useful in removing existent or future noise (white spots) caused by parasite reflections on the background. The method is applied to *Live_image_HSV_THR* by using the following command line:

```
HSV_MORPH_OPND =
cv2.morphologyEx(Live_image_HSV_THR,
cv2.MORPH_OPEN, kernel_OPING)
```

Secondly, the *Closing* method is useful in smoothing the foreground, the perimeter edges of the blob, in order to ease the contour finding process. The method is applied to *HSV_MORPH_OPND* by using the following command line:

```
HSV_MORPH_CLSD =
cv2.morphologyEx(HSV_MORPH_OPND,
cv2.MORPH_CLOSE, kernel_CLING)
```

In order to perceive the improvement, *Live_image_HSV_THR* and *HSV_MORPH_CLSD* are displayed in parallel, Figure 2.2.6 and Figure 2.2.7.

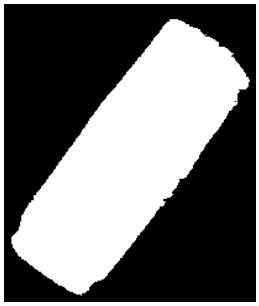

Figure 2.2.6 - *Live_image_HSV_THR*

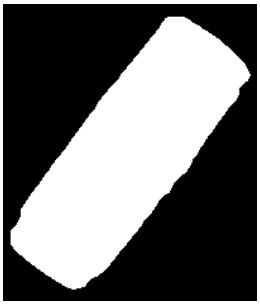

Figure 2.2.7 –*HSV_MORPH_CLSD*

➢ **CONTOUR**

Furthermore, *cv2.findContours()* function is applied to *HSV_MORPH_CLSD* in order to facilitate the contour detection.

```
contours = cv2.findContours(HSV_MORPH_CLSD,
cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
```

As result, the detected contour in *HSV_MORPH_CLOSED* is superposed over *Live_Image* as follows, Figure 2.2.8.

```
HSV_MORPH_CNT = cv2.drawContours(Live_image,
cnt, -1, (0, 255, 0), 3)
cnt = contours[0]
```

Figure 2.2.8 – HSV_MORPH_CNT

➢ **CENTROID**

Firstly, a bounding rotated rectangle is needed in order to consider the rotation of the object relative to Y/vertical axis. This type of rectangle is drawn with minimum area and it also needs 4 points as corners. Therefore, the command lines are the following:

```
rangle = cv2.minAreaRect(cnt)
box = cv2.boxPoints(rangle)
box = np.int0(box)
IMG_RANGLE =
cv2.drawContours(IMG_RANGLE, [box],-1,
(255,255,0),3)
```

As result, the bounding rectangle is drawn in *IMG_RANGLE*, Figure 2.2.9, where the actual perimeter of the rectangle is represented by turquoise color and the 4 corners by green color:

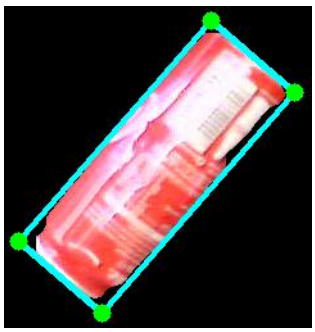

Figure 2.2.9 – Bounding rotated rectangle

Secondly, in order to calculate the centroid coordinates, the function *cv2.moments()* is used as follows:

```
M = cv2.moments(box)
cx = int(M['m10'] / M['m00']
cy = int(M['m01'] / M['m00']
```

Where *cx* and *cy* represent the rectangle's centroid coordinates relative to the top-left corner of the frame's field of view. In the next figure, the centroid is represented by a blue dot, Figure 2.2.10.

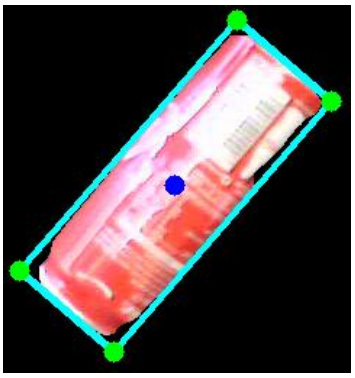

Figure 2.2.10 – Centroid location

In addition to this, the rotation of the object relative to the Y axis is calculated by using the slope of the yellow median transversal line described by the magenta and orange dots, Figure 2.2.11.

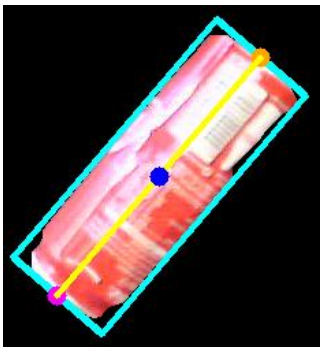

Figure 2.2.11 – Yellow median transversal line

Therefore the slope is calculated by using the following formula:

$$slope = \frac{c\square ange\ in\ y}{c\square ange\ in\ x} = \frac{L2_y - L1_y}{L1_x - L2_x}$$

Where $(L1_x,\ L1_y)$ – coordinates of the magenta dot and $(L2_x,\ L2_y)$ – coordinates of the orange dot

The angle relative to the y axis calculated by using the following equation:

$$\propto = 90^\circ - \left(\frac{\arctan(slope)\cdot 180^\circ}{\pi}\right) \quad (3)$$

As it can be seen in Figure 2.2.12, the turquoise vertical line, pictured on the left hand side of the frame, represents the Y axis, while the yellow horizontal line represents the X axis and the origin is in the top-left corner of the frame, represented by the red dot. The real-time calculated coordinates of the centroid and orientation of the object are to be found in the top-right corner. For this particular image, the algorithm outputs the following piece of information:

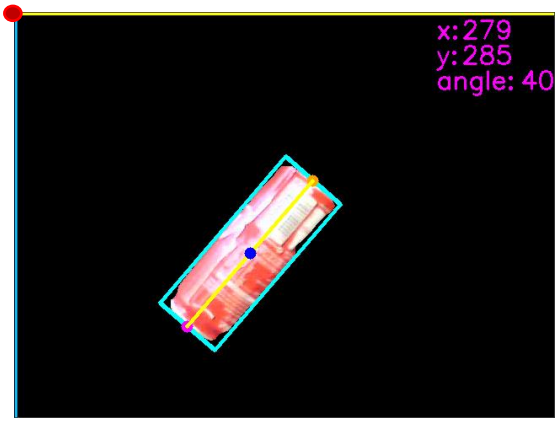

Figure 2.2.12 – Centroid coordinates and orientation

3. Conclusions

This paper has as objective the elaboration of an image processing algorithm that is able to answer to a specific set of needs, such as calculating the centre coordinates and planar orientation of a regular shaped object, such as metal cans, small-medium sized boxes and bottles.

Having this specific piece of information set and ready, further actions can be performed, such as:

- Implementing the algorithm within a Raspberry Pi powered robotic system, e.g. robotic serial/parallel arm, indoor mobile robots, inspection drones;
- Interfacing the Raspberry Pi platform with an industrial "pick up and place" robot solution, such as planar gantries, SCARA robots and-so-forth.

Acknowledgements

This work was supported by a grant of the Romanian Ministry of Research and Innovation, CCCDI – UEFISCDI, project number PN-III-P1-1.2-PCCDI-2017-0086 / contract no. 22 PCCDI /2018, within PNCDI III.

References

[1] (2008) HSV Color Space. In: Li D. (eds) Encyclopedia of Microfluidics and Nanofluidics. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-48998-8_656
[2] https://commons.wikimedia.org/wiki/File:HSV_color_solid_cylinder.png
[3] https://realpython.com/python-opencv-color-spaces/
[4] Fairchild, Mark D. (2005). *Color Appearance Models* (2nd ed.). Addison-Wesley.
[5] https://docs.opencv.org/master/d9/d61/tutorial_py_morphological_ops.html