DE GRUYTER OPEN

Print ISSN: 1312-2622; Online ISSN: 2367-5357
DOI: 10.1515/itc-2018-0012

Computer Vision Based on Raspberry Pi System

M. Abdulhamid, O. Odondi, M. AL-Rawi

Key Words: *Computer vision; Raspberry Pi system.*

Abstract. *The paper focused on designing and developing a Raspberry Pi based system employing a camera which is able to detect and count objects within a target area. Python was the programming language of choice for this work. This is because it is a very powerful language, and it is compatible with the Pi. Besides, it lends itself to rapid application development and there are online communities that program Raspberry Pi computer using python. The results show that the implemented system was able to detect different kinds of objects in a given image. The number of objects was also generated displayed by the system. Also the results show an average efficiency of 90.206% was determined. The system is therefore seen to be highly reliable.*

1. Introduction

Object counting is an important image processing technique that is applicable in many industrial applications. Some examples of these applications include: counting the number of products passing a conveyor belt, counting the number of cars passing through a given road at a given time, or counting the number of a particular species in a wildlife park.

Cameras have become a standard hardware and a required feature in many mobile devices. These developments have moved computer vision from a niche tool to an increasingly common tool for very many applications such as facial recognition programs, gaming interfaces, industrial automation, biometrics, medical image analysis, and planetary exploration.

Raspberry Pi is one such mobile device that comes with a built in camera slot. There are a number of applications that can be achieved through the Pi camera. Hobbyist uses it to develop gaming programs and robotic applications. They direct a robot using a given set of image instructions such as, turn left, or right, or stop.

Computer vision is the automated extraction of information from images. Such information includes: 3D models, object detection and recognition, grouping and searching information, image warping, de-noising among others. Intelligent Transportation Society of America (ITSA) defines computer vision as the process of using an image sensor to capture images, then using a computer processor to analyze these images to extract information of interest.

Computer vision is used in a wide variety of real-world applications such as Optical Character Recognition (OCR) to read handwritten postal codes, rapid machine parts inspection in production plants for quality assurance, using stereo vision with specialized illumination to measure tolerances on aircraft wings or auto body parts, and looking defects in steel castings using X-ray vision. Computer vision is also used in object recognition for automated check points in retail, automotive safety by detecting unexpected obstacles such as pedestrians on the street, under conditions where active vision techniques such as radar do not work, and in medical imaging.

Raspberry Pi is defined as a low cost, credit-card sized computer that plugs into a computer monitor or TV, and uses a standard keyboard and mouse. It is presented as a little device that enables people to experience computing and learn programming languages such as Scratch and Python. Essentially, it can perform anything that one would expect a desktop computer or laptop to perform. Some works which use Raspberry Pi in computer vision are found in literatures [1-6].

2. Design Procedure

2.1. Hardware Requirements

This work is accomplished using the following hardware components: Raspberry Pi, Pi camera, and power supply.

2.1.1. Raspberry Pi and SD Card

The design of this work uses PI (Model B+). In order to efficiently execute the work, Raspbian Jessie OS was installed in a 16 GB Secure Digital (SD) card. As opposed to Raspbian Jessie Lite OS and Whizzy OS, Raspbian Jessie gives a Graphical User Interface (GUI) experience. Therefore, it was not imperative to use Putty to access the Raspberry Pi remotely. With Remote Desktop Protocol (xRDP) installed in the Pi, one can connect to the Raspberry Pi remotely using the Windows Remote Desktop Connection application. It was developed by Raspberry Pi foundation in UK to be used for the advancement of computer science education. The second version of the Raspberry Pi is used here.

2.1.2. Raspberry Pi Camera

The Raspberry Pi camera board plugs directly into the Camera Serial Interface (CSI) connector on the Raspberry Pi. The Raspberry Pi camera module attaches to Raspberry Pi by way of a 15 pin Ribbon cable to the dedicated 15-pin Mobile Industry Processor Interface (MIPI) CSI which was designed especially for interfacing to cameras. It is able to deliver a clear 5 megapixel resolution image or 1080p High-definition (HD) video recording at 30 frames/sec.

2.1.3. Power Supply

The power supply on Raspberry Pi is quite simple. It powers through a Micro Universal Serial Bus (USB) connection which is capable of supplying at least 700 mA at 5 v.

2.2. Ethernet Cable

There are various ways of accessing the Raspberry Pi. It is not possible to work on the Raspberry Pi on its own as it does not have either a monitor or a keyboard. It is therefore important to have an Audio-visual/High-Definition Multimedia Interface (AV/HDMI) display and a keyboard. However, it can also be accessed remotely by connecting it to a laptop or a desktop using an Ethernet cable. The later method was adopted for its convenience.

2.3. Software Used

The software used to successfully realize the objectives of the work include: Python, Open source Computer Vision software (OpenCV), Microsoft Office Visio and Word, Raspbian Jessie OS and Remote Desktop Connection application in Microsoft Windows 8.1 which granted remote access to Pi.

2.4. Modeling

The overall design is visualized at hardware level by block diagram of *figure 1* while the flowchart of *figure 2* gives the steps used in implementing the system.

2.4.1. Block Diagram

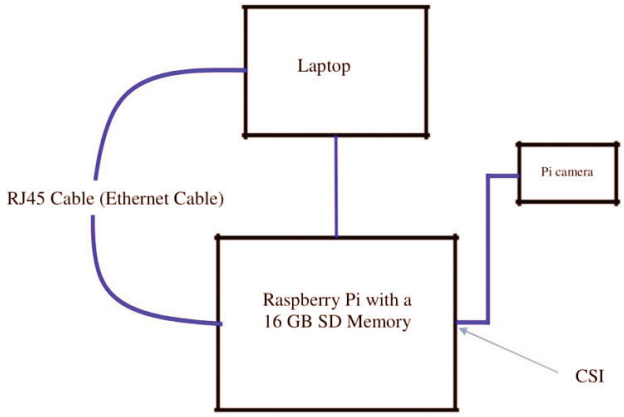


Figure 1. Block diagram showing the integration of the hardware

The integration of different hardware components is as shown by the block diagram of figure 1. The figure shows the integration of various hardware components. The Pi camera was inserted into the CSI slot provided on the Pi, an Ethernet cable is connected to the Ethernet ports of both Pi and the laptop to access Pi remotely. The micro USB 5 V 700 mA was used to power Raspberry Pi.

2.4.2. Methodology

There are several ways in which this task could be achieved. However, design settled on the solution shown in the flowchart of *figure 2* below. The critical steps in the design were as outlined.

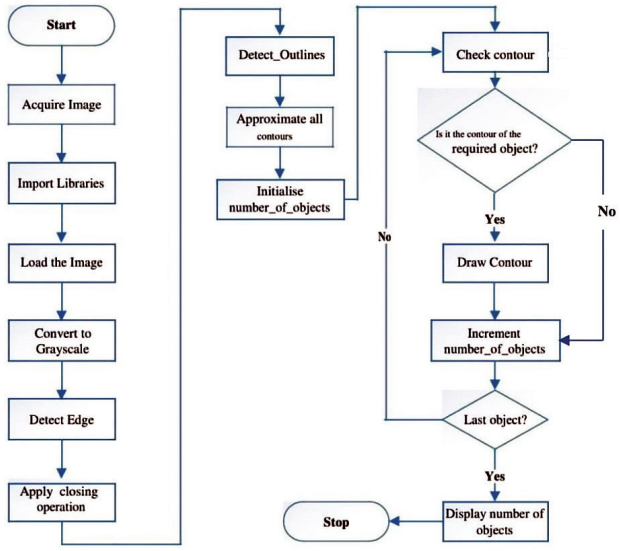


Figure 2. Flow chart of the algorithm

2.4.2.1. Image Acquisition

This step involves image capture using the Pi camera mounted on the Raspberry Pi computer and saving it to Pi root directory. There are several ways of capturing an image using a Pi camera for example the command *"raspistill -o my_image.jpg"* captures and saves a Joint Photographic Experts Group (JPEG) image as *my image* to the root directory of the Raspberry Pi.

2.4.2.2. Image Processing

The first step in image processing is to import the necessary OpenCV libraries (numpy and cv2) before load the image off disk using *cv2.imready* (*"image.jpg"*) function. This function reads the image that was captured so that it can be processed. The next step is grayscale conversion. This is achieved using the *cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)* where cv2.COLOR_BGR2GRAY is the flag which converts a BGR image to a GRAY image.

It is necessary to blur the grayscale image slightly to reduce high frequency (Gaussian) noise and increase the efficiency of the algorithm. This was realised using *cv2.GaussianBlur(gray, (3, 3), 0)*. The function takes four arguments. The first is the image, the second shown in the bracket are the height and the width of the standard deviation. Finally, we also need to specify sigmaX and sigmaY. If only sigmaX is specied, sigmaY is assumed to be the same as sigmaX. In this case, sigmaX was zero which implied that the two sigmas were calculated from the kernel. There are other functions in openCV that can also be used to blur an image such as averaging *(cv2.blur())*, median blurring *(cv2.medianBlur())* and bilateral filtering *(cv2.bilateralFilter())* but Gaussian blurring was used because it is highly effective in removing Gaussian noise from the image.

The next step is to detect the edges of objects in the image. Canny edge detection algorithm was used because it is widely accepted as the optimal detection algorithm. The openCV function *cv2.Canny(gray, 10, 250)* was used with 10 and 250 as the minimum and maximum values respectively.

In many cases, some outlines of the objects in the images are not "clean" and complete. There are usually gaps in between the outlines that must be closed if objects were to be successfully detected. To solve this, a "closing" operation was applied to close the gaps between the white pixels in the image. Two morphological transformation functions: *cv2.getStructuringElement(), and cv2.morphologyEx()* were used. *cv2.morpholgyEx()* takes three inputs. The first one is the image while second is the morphological function, for example, cv2.MORPH_OPEN, or cv2.MORPH_GRADIENT, or cv2.MORPH_CLOSE, etc. depending on what we want to achieve. In this case cv2.MORPH_CLOSE is the argument passed since there is need to close the small holes inside the foreground objects, or small black points on the object. Lastly, the third input is by default kernel. On the other hand *getStructuringElement()* takes two arguments – the first one is dependent on the shape of kernel required, e.g. a rectangle or a cross, while the second argument specifies the matrix e.g. (7,7) for a 7X7 matrix.

The next step was to detect contours or outlines of the objects in the image. A contour is simply a curve joining all the continuous points (along the boundary), having same colour or intensity. *cv2.findContours()* function is used. It takes three inputs: the first one is the image, the second one is the contour retrieval mode and the third is one is the contour approximation method. The argument cv2.CHAIN_APPROX_SIMPLE is preferred to cv2.CHAIN_APPROX_NONE since the latter saves all the points of the contour while the former only save the end points of the contour thus saves on memory space. For a rectangle, only four points are saved.

To check if a contour is of the required object or not, it is necessary to loop over each of the contours one by one. The functions *cv2.arcLength(c, True),* where the first argument is the contour, while the second argument specifies whether the shape is a closed contour (if passed True) or just a curve, and *cv2.approxPolyDP(c, 0.02 * peri, True)* were used. The first argument passed is the counter and the next is epsilon which is usually a percentage of the arc length *('arcLength')*. The third argument is as in the case of *arcLength()* function. Finally, the processed image is displayed on the screen using the *drawContours()* function.

2.4.3. Assumptions

The code assumed that a circle is a polygon with more than several edges. Any circular object in any image was therefore processed and displayed as an object with more than four edges.

3. Results

3.1. Single Object

3.1.1. Rectangular Object

Figures 3 to *7* show the steps involved in detecting and counting rectangular object. *Figure 3* shows the raw image captured while *figure 4* shows the same image in gray scale.

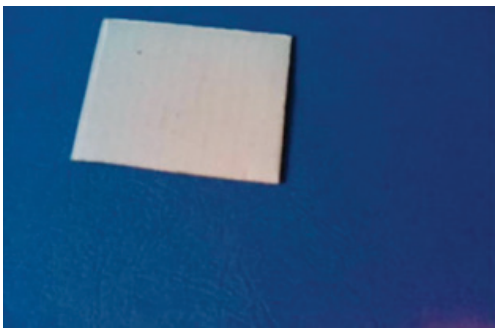


Figure 3. Rectangular image

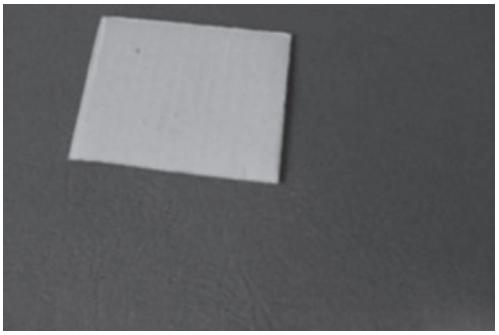


Figure 4. Grayscale Image of *figure 3*

Figure 5 shows the result of canny edge detection while *figure 6* is the result of closing operation. While a clear cut difference between them may not be easily detected by the human eye, a keen look at the vertex of *figure 6* (pointed to by an arrow) shows that a tiny gap at the same corner of *figure 5* was closed. Latter results will clearly show this. Finally, *figure 7* shows the output image displayed after image processing.

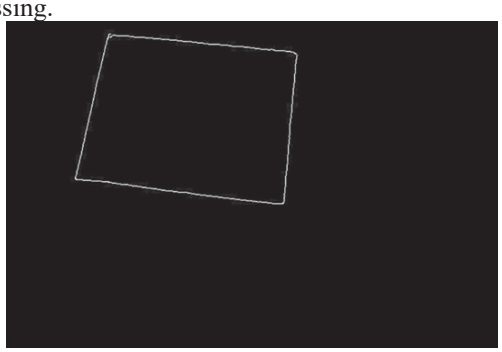


Figure 5. Result of edge detection

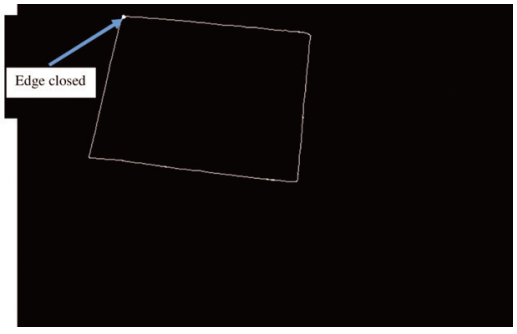


Figure 6. Closed form of *figure 3*

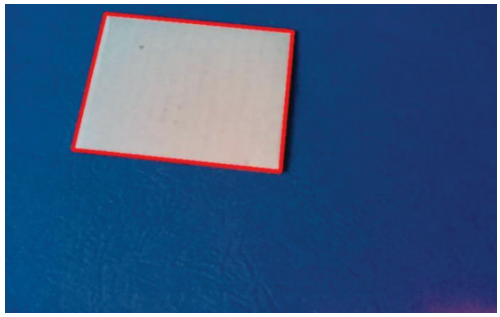


Figure 7. Ouput of *figure 3*

In addition to the output, the code also retuned of the number of objects detected on the terminal upon successful completion. In this case, the following result was generated by the system:

```
There isn't any triangular object in the im-
age.

There is 1 rectangular object in the image.

There isn't any object with more than four
edges in the image.

I am glad to let you know that there is 1 ob-
ject in this image.
```

3.1.2. Triangular Object

Similar observations were made as in the case of the one triangular object. *Figure 8* shows the test image while *figure 9* is the blurred grayscale image of *figure 8*. The feedback from the terminal specifies that only one triangular object has been identified. Again there is a small gap as shown in *figure 10* which is closed before the contour is drawn. *Figure 11* is the result of closing operation while *figure 12* shows the output image displayed after image processing.

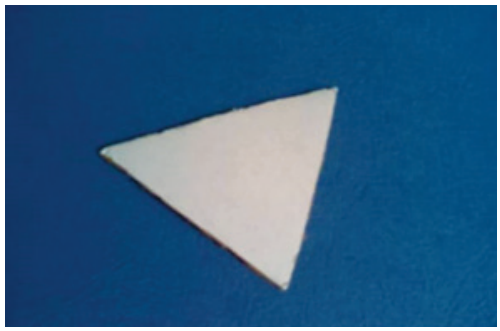


Figure 8. Triangular object

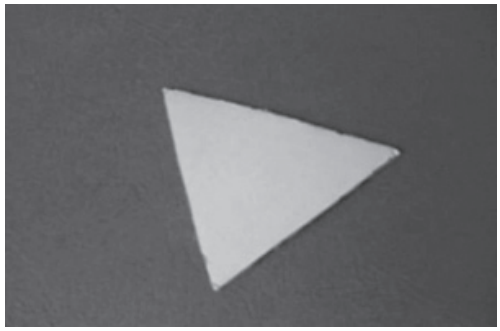


Figure 9. Grayscale Image of *figure 8*

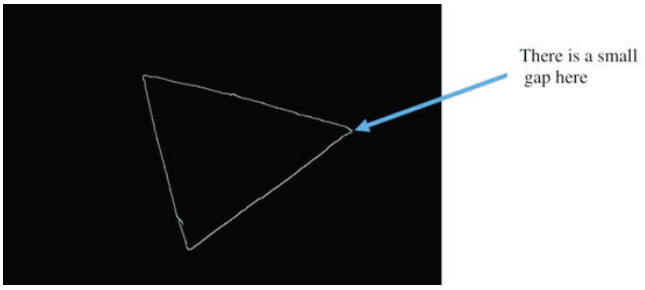


Figure 10. Edged image

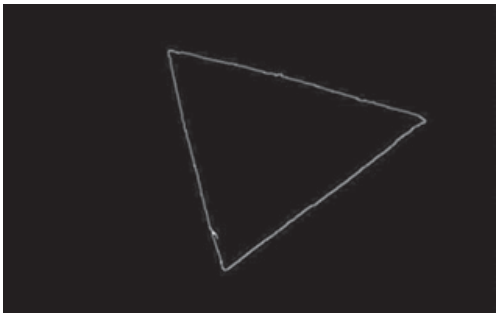


Figure 11. Closed image

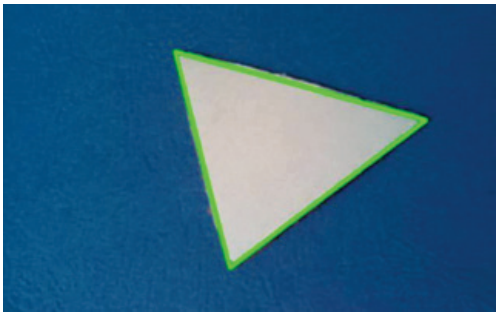


Figure 12. System generated output

The system generated the following feedback when the code was executed to analyze *figure 8*. The result shows that only one triangular object was detected.

```
There is 1 triangular object in the image.

There isn't any rectangular object in the
image.

There isn't any object with more than four
edges in the image.

I am glad to let you know that there is 1 ob-
ject in this image.
```

3.2. Multiple Objects

3.2.1. Human Vision versus Computer Vision

The test data of *figure 13* was used to evaluate the performance of the code when there are different kinds of objects. While there are six rectangular shaped objects by human, the code only identified four of them. Other objects were accurately identified.

An interesting observation is noted here. Since the objects were cut without a ruler and by using a scissor i.e. freehand, the code was able to detect an extra vertex, which

a human eye would not have detected at the bottom right corner of the image. This object is therefore identified as a polygon as shown in *figure 17*. The high level of efficiency is attributed to the fact that computer vision is based on evaluation at pixel level while the human vision is usually based on pre-recorded information. It is also necessary to note that the code classifies coins as polygons with more than four edges. This is because any circular shaped object seen as a polygon.

The object pointed to by the arrow in *figure 13* was not detected by the code. The only parameter that stands out for this object is its small size relative to the other "human perceived rectangles". This is a typical error that would make human vision preferred to computer vision.

Figure 14 shows the result obtained when *figure 13* was converted to a grayscale image and blurring it. *Figure 15* is the result of Canny edge detection algorithm, while *figure 16* is the result of closing and clearly shows the need for performing closing to an image before determining the type of the image. After edge detection, the twenty shillings coin (the coin on the extreme left of the test image – *figure 13*) is seen to have some gaps within its contours which without performing the closing operation could have been considered as objects within the coin.

The following result was generated by the system when the command python *countdiffobjects.py* was run in the terminal to process *figure 13*.

```
There are 2 triangular objects in the image.

There are 4 rectangular objects in the image.

There are 6 objects with more than four edges
in the image.

I am glad to let you know that there are 12
objects in this image.
```

Figure 13. Image with several objects

Figure 14. Grayscale image

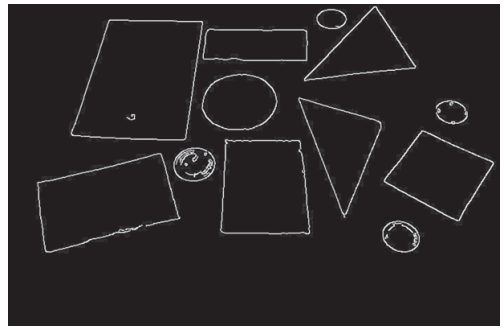


Figure 15. Result of edge detection on *figure 13*

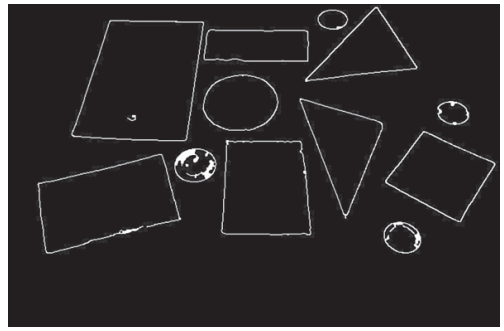


Figure 16. Result of "Closing"operation on *figure 13*

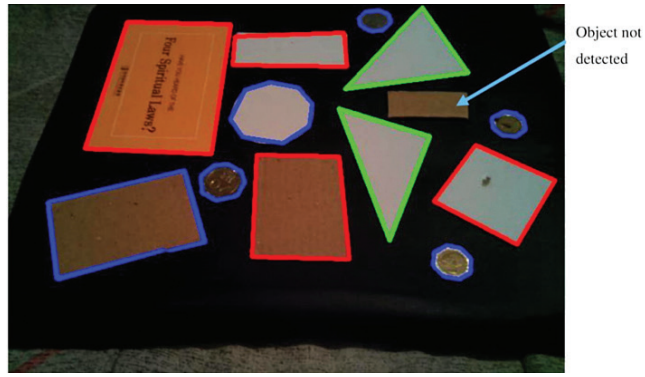


Figure 17. Output of *figure 13*

3.2.2. Optimal Performance

The success of the code depended on the inter-object spacing and the distance from the object to the camera lens (focus). The code was found to be most reliable with a better trade-off between inter-object spacing and focus. Besides, the intensity of the background color affected the performance of the code. Using the specimen of *figure 18*, which was captured when the distance between the objects was optimal and with a better focus, the best result was realized.

Figure 18 is the object under test while *figure 19* is its grayscale image. *Figure 20* and *figure 21* are the edged and closed images respectively. It was observed that all objects were successfully detected because of the sharp contrast in the color of objects and that of the background.

When the image of figure 18 was processed, the output image of *figure 22* was displayed. All the objects were detected and counted. The rectangular and triangular objects were each five while objects with more than four edges were seven. In total therefore, seventeen objects were identified.

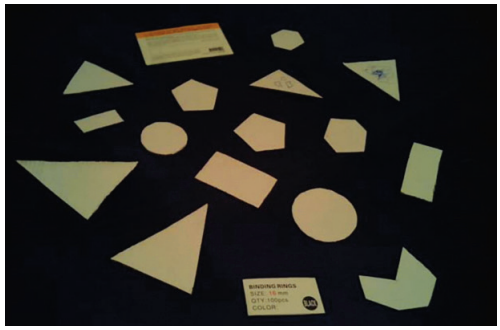


Figure 18. Light colored objects ina black background

Figure 19. Gray image of *figure 18*

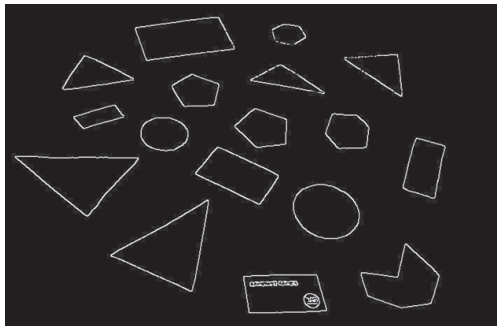


Figure 20. Edged image of *figure 18*

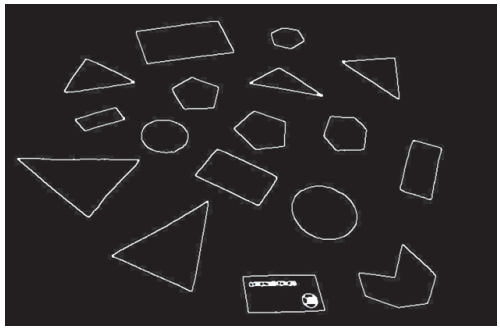


Figure 21. Closed image of *figure 18*

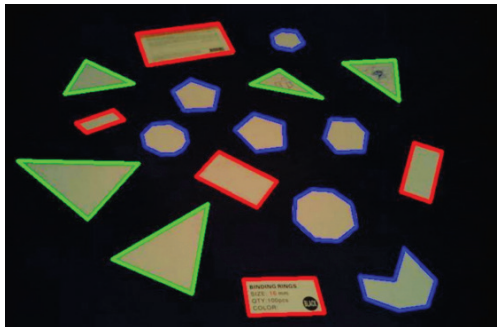


Figure 22. Output of *figure 18*

The system generated the following results when *figure 18* was processed:

```
There are 5 triangular objects in the image.

There are 5 rectangular objects in the image.

There are 7 objects with more than four edges
in the image.

I am glad to let you know that there are 17
objects in this image.
```

The distance from the object to the camera is crucial in the performance of this code. As above figures show, the rectangular object at the bottom of those figures is ultimately detected as one object but when a close-up of the same image is taken, the words and the circular drawing within the same object are detected too.

3.3. Effect of Background Color and Inter-object Spacing on the System

The accuracy of the code was seen to reduce considerably when the objects were close to each other. This section illustrates how inter-object spacing and color intensity of the background in relation to that of the object affected the reliability of the code.

Figure 23 was used to illustrate the two phenomena. It was observed that the spacing of the object, especially when there is no sharp contrast between background color and the color of object, two or more objects were identified as one.

When the image is converted to grayscale, the two pentagons pointed to by the red arrow in *figure 23*, were seen to be of almost the same intensity as the background (see *figure 24*). Their outline were therefore not detected as shown in *figure 25* and *figure 26*. Equally, objects that are close to one another were also detected as one object as shown in *figure 27*. Since the rectangular object and the triangular object adjacent to it (pointed to by the blue arrow in *figure 23*) are very close to each other, their outlines were jointly detected as a single object (see *figure 27*). This is attributed to the fact that the color intensity of the images and background are almost the same in addition to them being close to each other.

The image was taken when the Pi camera was very closer to the object than the previous one. As such, the texture of the background affected the accuracy of the system as shown by the detection of non-object edges as shown in *figure 25*. Two triangles on the extreme left side, and the leftmost rectangular object (*figure 23*) were therefore detected as one object as shown in *figure 27*.

The following result was obtained from the system when *figure 23* was analyzed.

```
There are 2 triangular objects in the image.

There are 2 rectangular objects in the image.

There are 6 objects with more than four edges
in the image.

I am glad to let you know that there are 10
objects in this image.
```

The result shows that only 10 objects were detected. In reality there were 17 objects as identified by human eye in the same image. The result shows the limitations of the system.

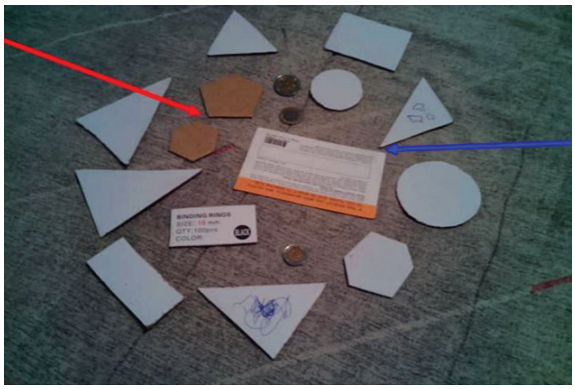


Figure 23. Image with objects close to one another

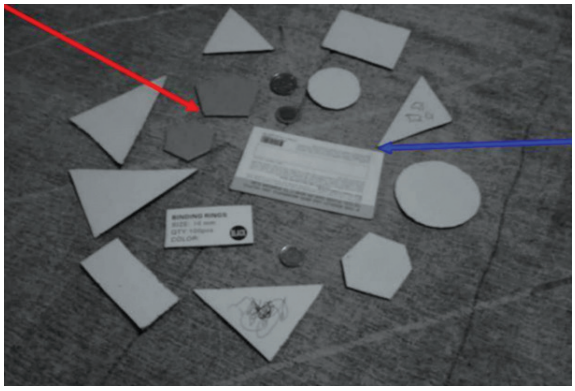


Figure 24. Grayscale image of *figure 23*

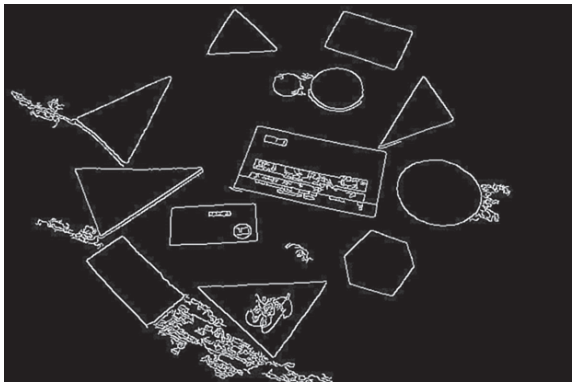


Figure 25. Edged image of *figure 23*

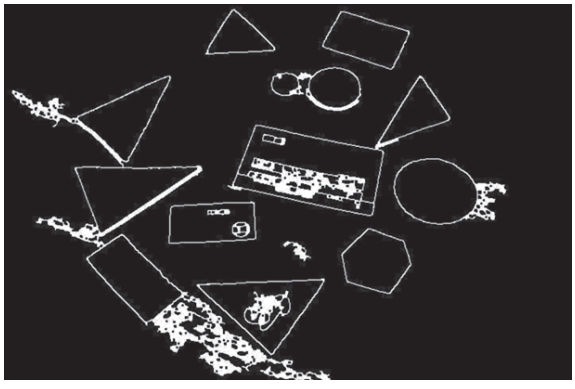


Figure 26. Closed image of *figure 23*

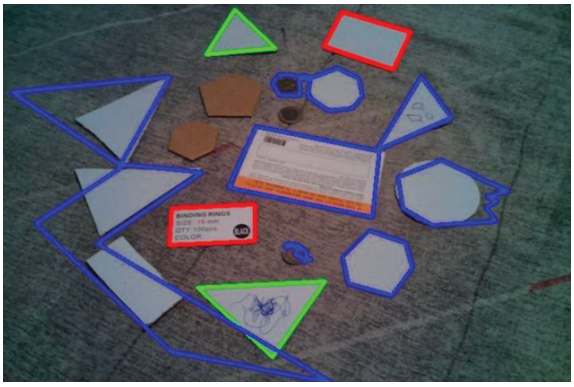


Figure 27. Output of *figure 23*

3.4. Effect of Shadow on Code

The objects with shadow in the image could not be detected especially if there was no sharp contrast in color intensity with respect with the background. When the image of *figure 28* was analyzed to detect rectangular shaped objects, errors were noted. This further showed other limitations of the code.

The rectangular box object had a shadow as shown in both *figure 28* and *figure 29* with a blue arrow. When Canny edge detector algorithm was applied, the edge with shadow was not detected. Besides, the eraser and the coin were not detected due to the effects of the background color. *Figure 30* and *figure 31* show the edged image and closed image of *figure 28* respectively.

As shown on the output generated by the system (*figure 32*), only one rectangular object is detected.

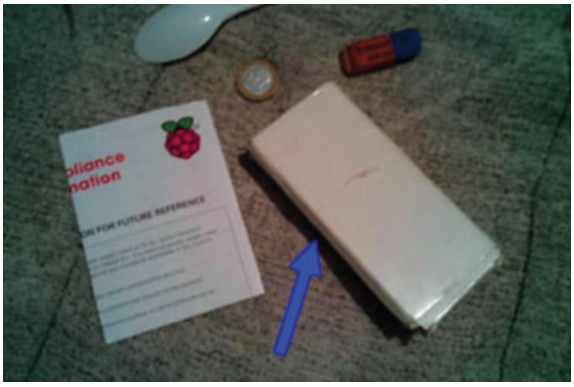


Figure 28. Image with shadow

Figure 29. Grayscale image of *figure 28*

Figure 30. Edged image of *figure 28*

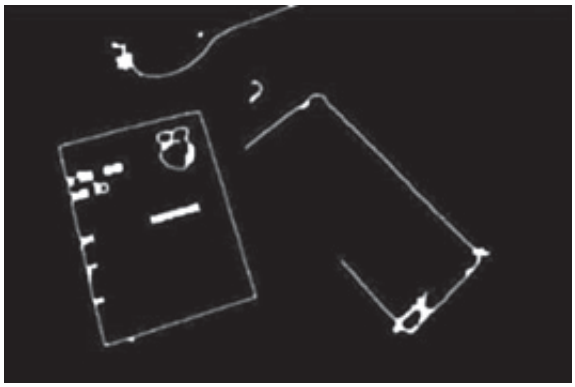


Figure 31. Closed image of *figure 28*

Figure 32. Output of *figure 28*

3.5. Effect of Close-up Image on Code

The closing operation works perfectly as long as the gaps between the objects are small. When the gaps exceed a threshold, the code identifies even the wittings on the objects. The following result was generated by the system when the code was executed for the image of *figure 33*. *Figure 34* shows the output image generated by the system.

```
There isn't any triangular object in the image.

There is 1 rectangular object in the image.

There are 32 objects with more than four edges in the image.

I am glad to let you know that there are 33 objects in this image.
```

By the human eye, this is a serious error since only one rectangular object was examined.

Figure 33. Close-up image

Figure 34. Output when a close-up is taken

4. Analysis

The functionality of the system is dependent on a number of factors which include: the background color in relation to the color of the object in the mage, the proximity of the camera to the object and by extension the texture of the background, the inter-object distance as well as the shadow as of the objects as was shown by the test results.

The test results for five objects – as listed in the *table* – were analyzed to determine the efficiency of the code.

Table 1. Efficiency analysis

Figure	Number of Objects	Objects Detected by Pi Camera Efficiency
3	1	100%
8	1	100%
13	12	92.31%
18	17	100%
23	10	58.72%
Total Efficiency		451.03
Average Efficiency		90.206

From the *table*, the worst case scenario was experienced when *figure 23* was analyzed. In this case, only 10 out of 17 objects were identified. The figures don't tell the whole story as revealed in *figure 27* where it is clearly shown that

only wrong objects were detected. This is because most objects were close to each other, the camera was closer to the object than other images that were analyzed and intensities of the background color and the objects were similar.

5. Conclusion

The work in this paper demonstrated the potential of the Raspberry Pi based system using Python as the programming language. The system was designed employing a camera which is able to detect and count objects within a target area. Different tests cases analyzed revealing the performance of the system. The reliability of the system was found out to depend on: number of objects within an image, the background color in relation to the color of the object, distance between objects, shadow of objects, and distance from the lens of the camera to the specimen (focus). The system was also able to differentiate between objects in an image based on their shapes. Rectangular objects, triangular objects and objects with more than four edges were easily detected. However, circular objects could not be detected as such partly because the camera was tilted at an angle and to a larger extent due to the assumptions outlined in the design section. The contours of circular detected as several edges with many vertices.

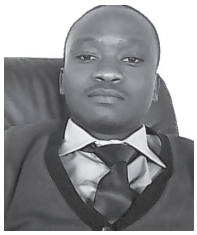
References

1. Senthilkumar, G., K. Gopalakrishnan and V. Sathish Kumar. Embedded Image Capturing System Using Raspberry Pi system. – *International Journal of Emerging Trends and Technology in Computer Science*, 3, 2014, No. 2, 213-215.
2. Md. Maminul Islam, Md. Sharif Uddin Azad, Md. Asfaqul Alam, N. Hassan. Raspberry Pi and Image Processing Based Electronic Voting Machine (EVM). – *International Journal of Scientific and Engineering Research*, 5, 2014, No. 1, 1506-1510.
3. Odondi, O. Computer Vision through the Raspberry-PI: Counting Objects. Graduation Project, University of Nairobi, Kenya, 2016, 1-62.
4. Sandin, V. Object Detection and Analysis Using Computer Vision. Graduation Project, Chalmers University of Technology, Sweden, 2017, 1-53.
5. Jana, S., S. Borkar. Autonomous Object Detection and Tracking Using Raspberry Pi'. – *International Journal of Engineering Science and Computing,* 7, 2017, No. 7, 14151-14155.
6. Nikam, A., A. Doddamani, D. Deshpande, S. Manjramkar. Raspberry Pi Based Obstacle Avoiding Robot. – *International Research Journal of Engineering and Technology*, 4, 2017, No. 2, 917-919.

Manuscript received on 27.09.2018

Mohanad Abdulhamid *was born in Iraq, 1969. He received B.Sc. degree in electrical engineering from AL-Mustansiryia University, Iraq, in 1990, M.Sc. degree in communication and electronics engineering from Baghdad University, Iraq, in 1993, and Ph.D. degree in telecommunication engineering from Bandung Institute of Technology, Indonesia, in 1999. He is currently Assistant Professor at the Department of Electrical Engineering, University of AL-Hikma, Iraq. His research interests include digital communications and digital signal processing.*

Contacts:
AL-Hikma University, Iraq
e-mail: moh1hamid@yahoo.com



Otieno Odondi *was born in Kenya in 1992. He received B.Sc. degree in electrical engineering from University of Nairobi, Kenya, in 2016. Currently, he is working as assistant lecturer at Department of Electrical Engineering, University of Nairobi, Kenya.*

Contacts:
University of Nairobi, Kenya
e-mail: researcher12018@yahoo.com

Muaayed AL-Rawi *was born in Iraq, 1971. He received B.Sc. degree in electrical and nuclear engineering from Baghdad University, Iraq, in 1992, and M.Sc. degree in communication and electronics engineering from Jordan University of Science and Technology, Jordan, in 1999. He had worked as nuclear and electrical engineer for several years at Iraqi Atomic Energy Organization, Iraq. Currently he is lecturer at Department of Electrical Engineering, AL-Mustansiryia University, Iraq. His research interests include biomedical engineering, digital communications and digital signal processing.*

Contacts:
AL-Mustansiryia University, Iraq
e-mail: muaayed@yahoo.com