

Counting Objects In Movement Using Raspberry PI & OpenCV

hackster.io/phfbertoleti/counting-objects-in-movement-using-raspberry-pi-opencv-015ba5

Pedro Henrique Fonseca Bertoleti




Things used in this project

Hardware components



Raspberry Pi 3 Model B

× 1 

Webcam

× 1

An ordinary USB webcam (any model Linux-compatible fits well here)

Power Source for Raspberry PI

A power-source (5V/3A recommended) with micro-USB connection. Some smartphone and tablet chargers fits well here too (in this case, observe if it's output current is compatible to what recommended for Raspberry PI)

Software apps and online services



[OpenCV](#)

Story

Introduction

Computer vision, doubtless, is a fantastic thing! Using this, a computer gains the capability to "see" and sensing better the environment around, what allows the development of complex, useful and cool applications. Applications such as face detecting and recognizing, object tracking and object detection are more and more present in our day-to-day activities, thanks to computer vision advances.

Considering how advanced and acessible are computer vision frameworks and tools, the application described in this article fits well: using a simple Raspberry PI and a cost-free and open-source computer vision framework called OpenCV to count objects in movement, more precisely how much objects go in and out of a certain monitored zone. Too complicated to understand? Check the following animated gif that show this project in action:

Project in action!

Are you curious about how doing it? Check it out!

Before the project - necessary setups

Before procceding to topics related to how this project works, some preparations are nedded in order to allow Raspberry PI run it:

Ensure you have penty of free space (for this, I strongly recommend you using 16GB SDCards at least)

Ensure your Raspberry PI is fully operational: Raspbian Distro installed (or another one that may interest you), full access to Raspberry PI User-Interface (phisically via monitor or via VNC remote access) and with Internet access.

Connect the USB webcam to one of the USB ports available

Still before the project - OpenCV installation

With Raspberry PI fully operational to this project, now it's time to install OpenCV! In my opinion, one of the most effective way for doing this is downloading OpenCV source-code and compilling it on Raspberry PI (it'll avoid incompatibilities that may occur when using pre-compiled OpenCV packages - the kind of problem that drives you crazy). It may take some time to get done, but the chances of getting odd errors are minimal.

In order to make this process easier for you, I wrote a shell-script that automatically downloads, compilles and install OpenCV in your Raspberry PI. Got interested on it? Follow the procedure bellow:

1. Get the script from my Github in this link:

<https://github.com/phfbertoleti/ScriptToCompileInstallOpenCV/blob/master/CompileInstallOpenCV.sh>

2. In your Raspberry PI, in CLI (Command Line Interface / Linux terminal), give full permissions to this script using chmod command:

```
chamod +777 CompileInstallOpenCV.sh
```

3. Execute it by doing the following command:

```
sudo ./CompileInstallOpenCV.sh
```

4. Done! Once it takes some relevant time (maybe some hours), it's a nice opportunity to drink some coffee.

Project overview

As it was said before, this project consists in a counter of objects in movement using computer vision (in this case, OpenCV running on Raspberry PI). About this counting, this project can count how much objects get in and out from the monitored zone.

For instance, this kind of monitoring can be very useful in retail stores. This can be very useful to determine how often custumers go to a certain department or corridor and, based on this, define how to distribute their products in a better way (to raise the sells).

Monitored zone

The monitored zone concept is simple: it means the image seen on webcam image

stream. To monitor it, some limit lines are required, to determine the entrance or exit from objects. These limit lines consist of virtual lines plotted over this images by OpenCV. There are two limit lines:

Entrance line: reference line that defines the limit for an object to enter in monitored zone

Exit line - reference line that defines the limit for an object to exit of monitored zone.

Therefore, as the in/out control is made based on entrance and exit lines (and these lines are plotted over webcam image), they can be easily defined and modified. Observe the monitored zone representation shown in figure 1.

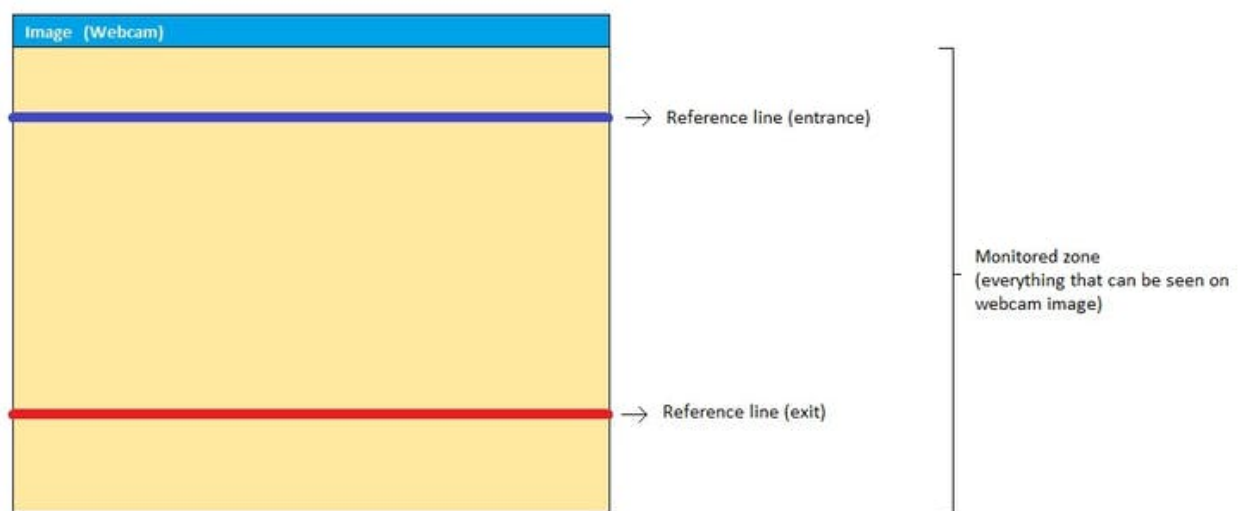


Figure 1 - monitored zone and entrance and exit lines

How can objects in movement be counted?

Here begins the image processing itself.

The increment of a counter (entrance ou exit counter) happens when the **centroid of the object** crosses the entrance or exit lines. This approach makes the count independent from object's shape. This is very good, since the kind of objects that will cross the lines are unknown (it can be an adult, a child, a vehicle, an animal and so on).

The object counting is based on the following characteristics:

Only objects in movement will be detected and considered (and, for consequence, counted)

The increment of counter only happens when a object crosses one of the reference lines (entrance or exit lines)

The direction / orientation of object's movement doesn't matter.

Therefore, considering these three characteristics, there is an ensurance that all objects getting in or out of the monitored zone will be counted.

The counting of objects getting in and out of the monitored zone is done as described below:

To count objects getting in the monitored zone: all object's centroids that crosses the blue line / entrance line and came from red line / exit line (in another words, all object's centroids that were located between the two lines and move in blue line direction) are counted as objects that got in monitored zone. This is shown in figure 2.a.

To count objects getting out the monitored zone: all object's centroids that crosses the red line / exit line and came from blue line / entrance line (in another words, all object's centroids that were located between the two lines and move in red line direction) are counted as objects that got out monitored zone. This is shown in figure 2.b.



Figure 2 - objects's movement direction detection

Important: as capture and frame processing aren't instantaneous (it takes a little - but relevant -time to get an image from webcam stream and proccess it), there's a real chance of not getting the exact moment of object's centroid line crossing. Based on this, there must be a tolerance of line crossing, a "pixel range" before and after each line. This tolerance zone acts like an entrance and/or exit line extension, so the counting won't be so dependent from exact moment that an object crosses a line, but if it's in line tolerance zone or not.

To put it on practical terms, in this project this tolerance consists of 2-pixel range up and down each line (entrance and exit lines). Therefore, if object's centroid is located inside a line tolerance, it's considered that this object crossed that line. Observe figure 3.

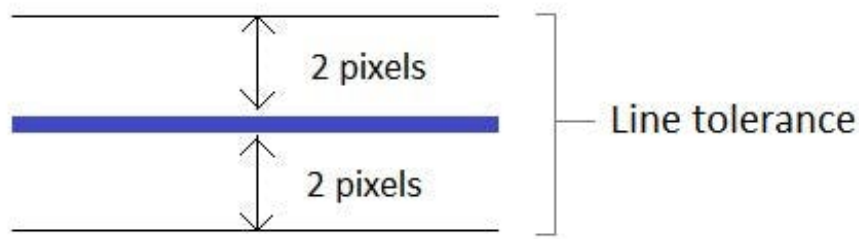


Figure 3 - line tolerance

Getting deeper: how can object movement is detected in an image stream?

Now it's time to getting deeper in image processing stuff: how to get some webcam stream images and detect that something have moved there.

It consists in five steps:

1. To highlight the object in movement

As defined in classical physics, a reference is necessary to infer that something is moving or if it's standing still. Here, to determine if something has moved, it's pretty much the same: every single webcam stream captured frame will be compared to a reference frame. If something is different, something has been moved. It's simple as it sounds. This reference frame must be captured in the most perfect conditions (nothing moving, for example).

In the image processing's world, this comparison between a captured frame and a reference frame consists in a technic called **background subtraction**. Background subtraction consists on literally subtract pixel-to-pixel color information from the captured frame and the reference frame. So, the resulting image from this process will highlight / show with more details only what is different between these two frames (or, what have moved / got movement) and everything else will be black in image (the color of zero value on a gray-scale pixel).

Important: lighting conditions and quality of webcam image captured (due to capture sensors quality) can slightly vary from frame to frame. It implies that the "equal parts" from reference frame and another frames won't be total black after background subtraction. Despite of this behavior, there's no serious consequences in the next steps image processing in this project.

In order to minimize image processing time, before doing a background subtraction, captured frame and reference frame are converted to an gray scale image. But.. why? It's a computing efficiency issue: a image that presents multiple colors (color image) has three informations per pixel: Red, Blue and Green color components (the old but gold RGB standard). So, mathematically, each pixel can be defined as a three-value array,

each one representing a color component. Therefore, extending it to the whole image, the final image will be actually the mix of three image components: Red, Blue and Green image components. To Process it, a lot of work is required! However, in gray-scale images, each pixel has only one color information. So, the processing of an color image is three times slower than in gray-scale image case (at least three times, depending on what technique is involved). And there's more: for some purposes (like this project), process all the colors isn't necessary or important at all. Therefore, we came to the conclusion: gray-scale images usage is highly recommended for image processing purpose.

After background subtraction, it's necessary to apply Gaussian Blur filter. The Gaussian Blur filter applied over background subtracted image smoothes all contours of the moving detected object. For sure, it'll be helpful in the next steps of image processing.

Observe the result of a background subtraction and Gaussian filter applying in figure 4.

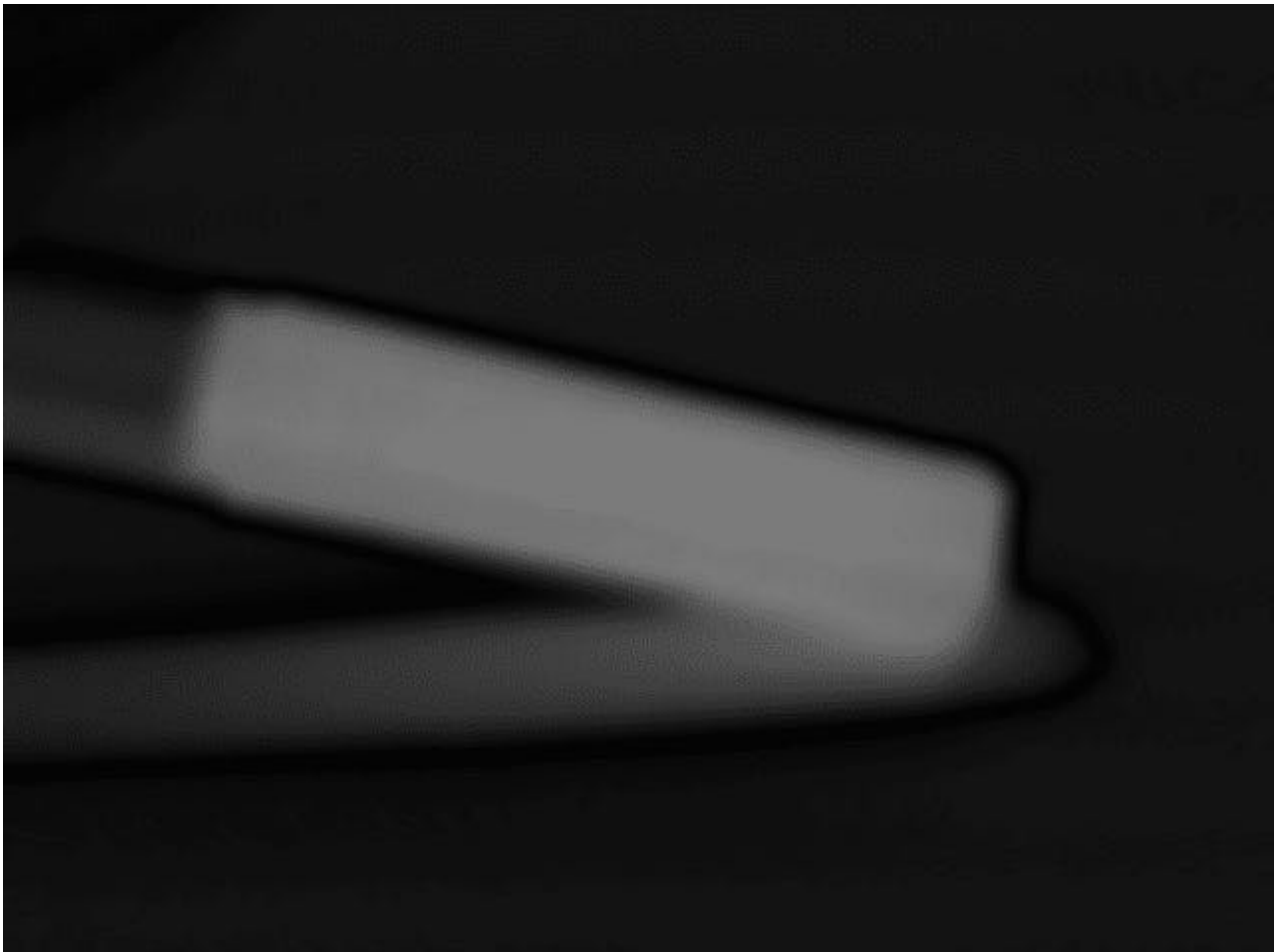


Figure 4 - result of background subtraction and Gaussian filter applying

2. Binarization

In most cases of image processing, binarization is almost a mandatory step after highlight objects / characteristics in a image. Reason: in a binary image, each pixel color can assume two values only: 0x00 (black) or 0xFF (white). This helps a lot the image processing in order to require even less "computing power" to apply image processing

techniques in the next steps. Binarization can be done comparing each pixel color of the gray-scale image to a certain threshold. If the value of the pixel color is greater than threshold, this pixel color will assume white value (0xFF), and if the value of the pixel color is lower than threshold, this pixel color will assume black value (0x00).

Unfortunately, threshold value's choice isn't so easy to make. It depends on environment factors, such as lighting conditions. A wrong choice of a threshold value can ruin all the steps further. So, I strongly recommend you adjust manually a threshold in the project for your case before any further actions. This threshold value must ensure that the moving object shows in binary image.

In my case, after a threshold's adequate choice, results in what you see in figure 5.

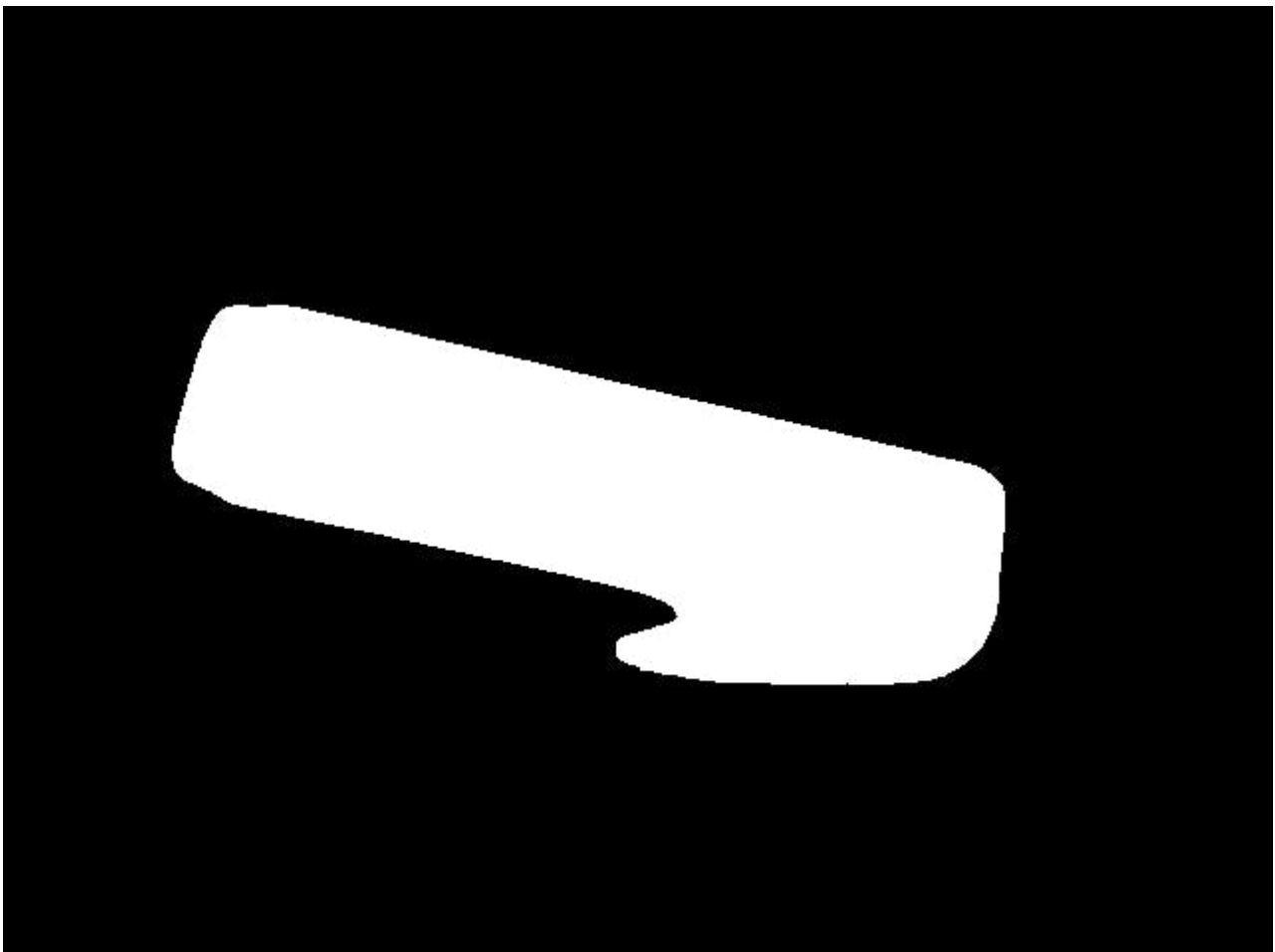


Figure 5 - binary image

3. Dilate

Until now, it was possible to detect moving objects, highlight them and apply binarization, what results in a pretty clear image of moving object (= pretty clear image of the object for image processing purposes). The preparation for object counting is ALMOST done. The "ALMOST" here means that there're some fine adjusts to make before moving on.

At this point, there're real chances of presence of "holes" in the objects (black masses of

pixels into the white highlighted object). These holes can be anything, from particular lighting conditions to some part of the object shape. Once holes can "produce" false objects inside real objects (depending on how big and where they're located), the consequences of holes presence in a image can be catastrophic to objects' counting.

A way to eliminate these holes is using an image processing technic called **Dilate**. Use this and holes go away.

4. The search for the contours (and its centroids)

At this point, we have the highlighted objects, no holes inside it and ready for what's next: the search for the contours (and its centroids). There're resources in OpenCV to detect automatically contours, but the detected countours must be wisely chosen (to pick the real object or objects only). So, the criteria to detect the contours is the area of the object, measured in pixels². If a contour has a higher area than a limit (configured in software),so it must be considered as a real object to be counted. The choice of this area limit/criteria is very important, and a bad choice here means wrong countings. You must try some area value limits values and check what fits better to your usage. Don't worry, these limit isn't sohard to find / adjust.

Once all the objects in the image are picked, the next step is to draw a retangle on it (this retangle must contain an entire detected object inside it). And the center of this rectangle is.... the object centroid!

You are maybe thinking "What's the big deal with this centroid?", right? Here's your answer: doesn't matter how big or how is the shape of the object, its movement is the same of the centroid. In another words: this simple point called centroid represents all the movement of the object. It does makes the counting very simple now, doesn't it?

See the image below (figure 6), where the object's centroid is represented as a black point.

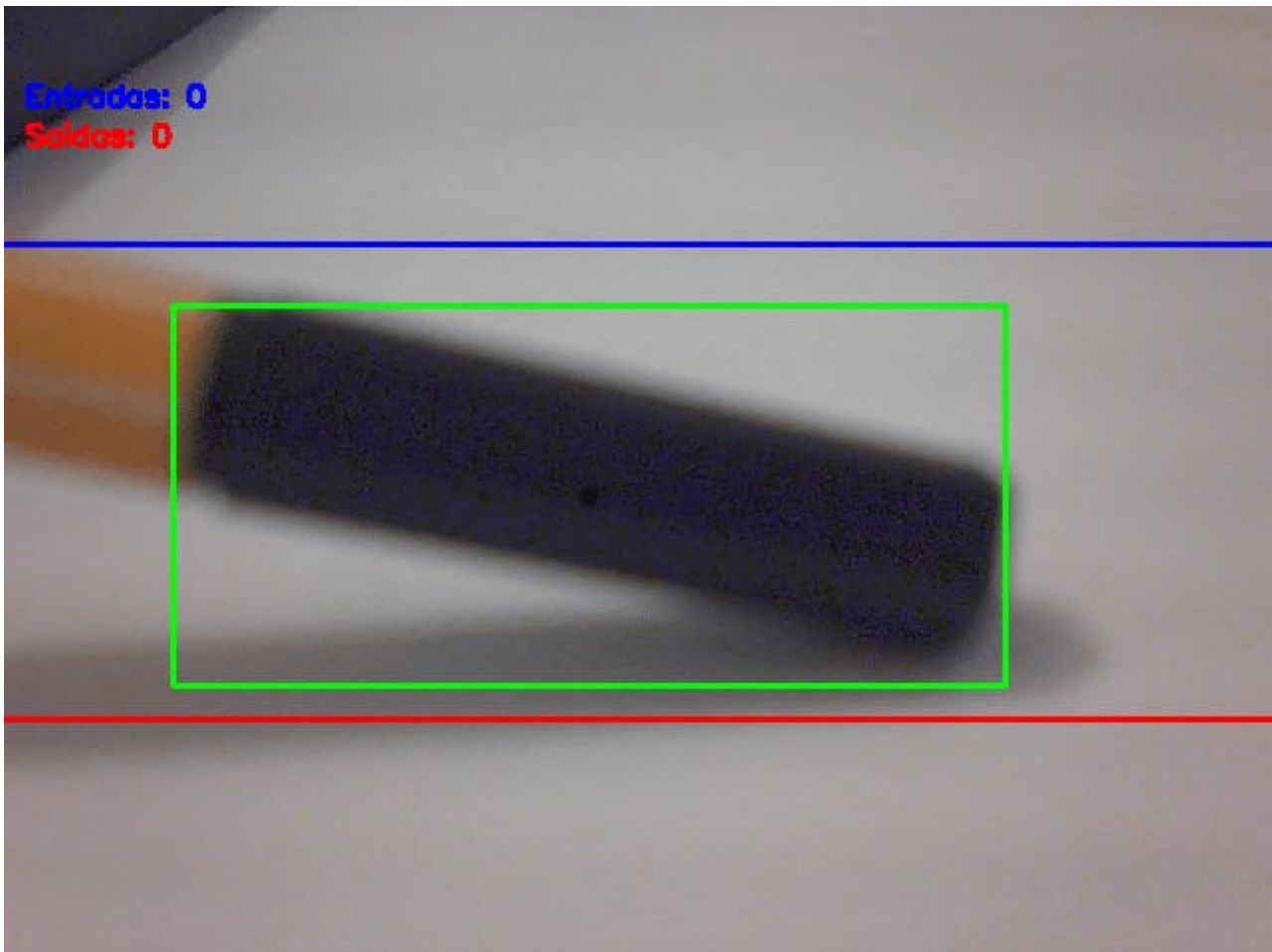


Figure 6 - object and its centroid (black point)

5. Centorid's movement and object counting

The grand finale: compare object's centroid coordinates to entrance and exit lines coordinates and apply the counting algorythm described before. And there'll be counting of moving objects!

Final result

As shown in the very beginning of this post, here is the project in action:

And here goes a very big "thank you" for pyimagesearch website (<http://www.pyimagesearch.com/>). This site really helped me on this project development.

Code

```
import datetime
import math
import cv2
import numpy as np
```

```
#global variables
```

```

width = 0
height = 0
EntranceCounter = 0
ExitCounter = 0
MinCountourArea = 3000 #Adjust ths value according to your usage
BinarizationThreshold = 70 #Adjust ths value according to your usage
OffsetRefLines = 150 #Adjust ths value according to your usage

#Check if an object in entering in monitored zone
def CheckEntranceLineCrossing(y, CoorYEntranceLine, CoorYExitLine):
    AbsDistance = abs(y - CoorYEntranceLine)

    if ((AbsDistance <= 2) and (y < CoorYExitLine)):
        return 1
    else:
        return 0

#Check if an object in exiting from monitored zone
def CheckExitLineCrossing(y, CoorYEntranceLine, CoorYExitLine):
    AbsDistance = abs(y - CoorYExitLine)

    if ((AbsDistance <= 2) and (y > CoorYEntranceLine)):
        return 1
    else:
        return 0

camera = cv2.VideoCapture(0)

#force 640x480 webcam resolution
camera.set(3,640)
camera.set(4,480)

ReferenceFrame = None

#The webcam maybe get some time / captured frames to adapt to ambience lighting. For this
reason, some frames are grabbed and discarted.
for i in range(0,20):
    (grabbed, Frame) = camera.read()

while True:
    (grabbed, Frame) = camera.read()
    height = np.size(Frame,0)
    width = np.size(Frame,1)

    #if cannot grab a frame, this program ends here.
    if not grabbed:
        break

    #gray-scale conversion and Gaussian blur filter applying
    GrayFrame = cv2.cvtColor(Frame, cv2.COLOR_BGR2GRAY)
    GrayFrame = cv2.GaussianBlur(GrayFrame, (21, 21), 0)

    if ReferenceFrame is None:
        ReferenceFrame = GrayFrame

```

```

        continue

#Background subtraction and image binarization
FrameDelta = cv2.absdiff(ReferenceFrame, GrayFrame)
FrameThresh = cv2.threshold(FrameDelta, BinarizationThreshold, 255, cv2.THRESH_BINARY)
[1]

#Dilate image and find all the contours
FrameThresh = cv2.dilate(FrameThresh, None, iterations=2)
_, cnts, _ = cv2.findContours(FrameThresh.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

QtyOfContours = 0

#plot reference lines (entrance and exit lines)
CoorYEntranceLine = (height / 2)-OffsetRefLines
CoorYExitLine = (height / 2)+OffsetRefLines
cv2.line(Frame, (0,CoorYEntranceLine), (width,CoorYEntranceLine), (255, 0, 0), 2)
cv2.line(Frame, (0,CoorYExitLine), (width,CoorYExitLine), (0, 0, 255), 2)

#check all found countours
for c in cnts:
    #if a contour has small area, it'll be ignored
    if cv2.contourArea(c) < MinCountourArea:
        continue

    QtyOfContours = QtyOfContours+1

    #draw an rectangle "around" the object
    (x, y, w, h) = cv2.boundingRect(c)
    cv2.rectangle(Frame, (x, y), (x + w, y + h), (0, 255, 0), 2)

    #find object's centroid
    CoordXCentroid = (x+x+w)/2
    CoordYCentroid = (y+y+h)/2
    ObjectCentroid = (CoordXCentroid,CoordYCentroid)
    cv2.circle(Frame, ObjectCentroid, 1, (0, 0, 0), 5)

if (CheckEntranceLineCrossing(CoordYCentroid,CoorYEntranceLine,CoorYExitLine)):
    EntranceCounter += 1

if (CheckExitLineCrossing(CoordYCentroid,CoorYEntranceLine,CoorYExitLine)):
    ExitCounter += 1

print "Total countours found: "+str(QtyOfContours)

#Write entrance and exit counter values on frame and shows it
cv2.putText(Frame, "Entrances: {}".format(str(EntranceCounter)), (10, 50),
    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (250, 0, 1), 2)
cv2.putText(Frame, "Exits: {}".format(str(ExitCounter)), (10, 70),
    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)
cv2.imshow("Original Frame", Frame)
cv2.waitKey(1);

```

```
# cleanup the camera and close any open windows  
camera.release()  
cv2.destroyAllWindows()
```

Credits

