UNIVERSITY of DEBRECEN

Dr. József Sütő
Department of IT Systems and Networks

Embedded Systems Laboratory

Lab 9: Basic motion detection

Project overview

The goal of this laboratory is to show you, how can you create a basic motion detection and tracking program on your Raspberry Pi (or even on another computer with built-in or USB camera) using computer vision methods. The video source of the algorithm can be both pre-recorded video or live video stream from camera.

The following topics will be covered in this lab:

- Motion detection
- Motion tracking

Technical requirements

The following components are required to complete this lab:

- Raspberry Pi 3 Model B+ (MicroSD card, power supply, keyboard, mouse)
- Pi camera module

OR

- PC
- USB/built-in camera

Background subtraction

Background subtraction is an important operation in many computer vision applications. Primarily, we can use it to count moving object (cars, people, etc.) in a video stream. Actually, there are more different ways to perform background subtraction. For example:

- Gaussian Mixture Model based
- Bayesian method based
- Image subtraction

Independently the type of the used method, all of those try to separate **background** from the **foreground**. Typically, the background of a fixed position camera is static and unchanging over a “short” time interval. If we can model the background, we will be able to detect substantial changes in the video stream. If an object passes in front of the camera it will generate substantial changes that can be detected by an algorithm.
Unfortunately, due to lighting changes, shadowing, reflection, and other environmental factors, the background can look different in various frames. When we choose a method for background and foreground segmentation, we have to take consider its computation requirement. In this laboratory we will develop a basic algorithm which will not be perfect but it will run normally on the Raspberry Pi and gives acceptable result.

1. Basic motion detection program

The Raspberry Pi requires that, you install a few packages before you get started. Perhaps you have installed all of them in the previous laboratory.

1.1. Import necessary libraries:

```
from imutils.video import VideoStream

import datetime

import imutils

import time

import cv2
```

1.2. Initialize threshold values. *T* is the threshold of image binarization while the other one defines the minimum contour area to insignificant contour suppression:

```
T = 50

min_area = 1000
```

1.3. Initialize a variable which will store the background. Here we suppose that, only the background is visible on the first frame:

```
background = None
```

1.4. In this step, we create a reference to the video stream. If you would like to use pre-recorded video, you should provide the path to the video location. If you would like to use the camera, video_path variable should remain *None*:

```
video_path = None
if video_path is None:
    vs = VideoStream().start()
    # warm up the camera
    time.sleep(2)
else:
    vs = cv2.VideoCapture(video_path)
```

1.5. Read images one by one from the video source:

```
while True:
    frame = vs.read()
    frame = frame if video_path is None else frame[1]
    # this variable keeps track the state of motion detection
    state = "No change"

    # if there is no more frame in the video break the loop
    if frame is None:
        break
```

1.6. Resize the frame, convert it to gray scale, and blur (or filter) it:

```
    frame = imutils.resize(frame, width=500)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    #smooting image in 21x21 pixels regions
    gray = cv2.GaussianBlur(gray, (21,21), 0)
```

1.7. If the background is *None*, initialize it:

```
if background is None:
    background = gray
    continue
```

1.8. Compute the absolute difference between the current frame and the background. Actually, it will be the intensity difference between two images. Thereafter, the delta frame goes through a threshold function where less significant changes on the image will be suppressed:

```
delta_frame = cv2.absdiff(background, gray)
threshold = cv2.threshold(delta_frame, T, 255,
cv2.THRESH_BINARY)[1]
```

1.9. Perform a dilation operation on the thresholded image to fill in the holes:

```
threshold = cv2.dilate(threshold, None, iterations=2)
```

1.10. On the thresholded image, we can apply contour detection to find outlines of white region(s) (moving object):

```
cnts = cv2.findContours(threshold.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)
```

1.11. Only those contours will be taken consider which have greater area than a predefined minimum value. If this is met, the program draws a bounding box around the contour and the motion state will be changed.

```
for c in cnts:
    if cv2.contourArea(c) < min_area:
        continue

    (x,y,w,h) = cv2.boundingRect(c)
    cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 2)
    state = "New object"
```

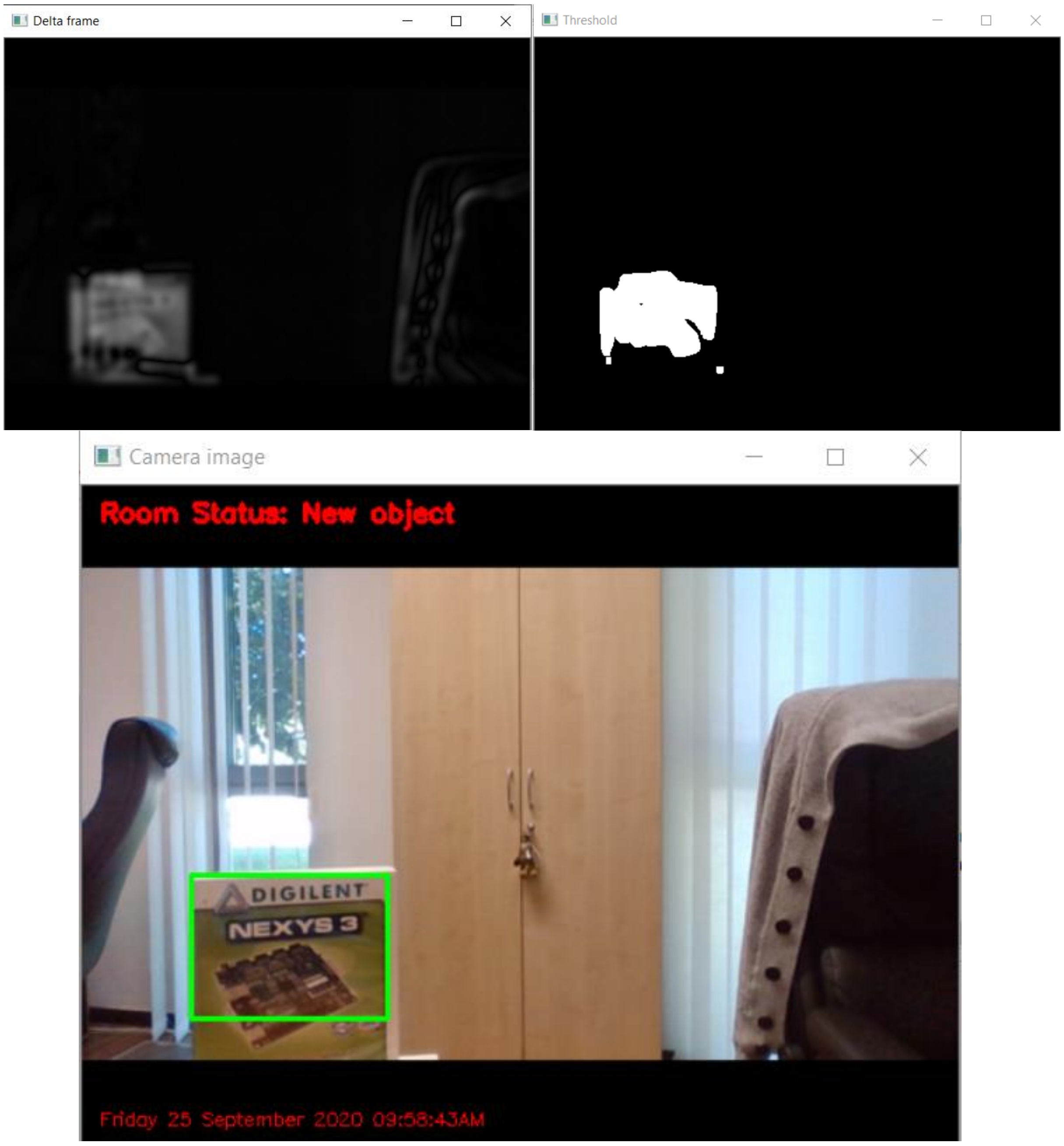
1.12. In the last step, we visualize the result to the user. Output images are extended with the status of the motion detection and a timestamp. In addition, delta and the thresholded images also will be visible. The algorithm can be stopped with the **q key.**

```
        cv2.putText(frame, "Room Status: {}".format(state), (10, 20),
    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)
        cv2.putText(frame, datetime.datetime.now().strftime("%A %d
    %B %Y %I:%M:%S%p"), (10, frame.shape[0] - 10),
    cv2.FONT_HERSHEY_SIMPLEX, 0.35, (0, 0, 255), 1)
        cv2.imshow("Camera image", frame)
        cv2.imshow("Threshold", threshold)
        cv2.imshow("Delta frame", delta_frame)
        key = cv2.waitKey(1) & 0xFF

        if key == ord("q"):
            break

    # shut down the camera and close all open windows
    vs.stop() if video_path is None else vs.release()
    cv2.destroyAllWindows()
```

1.13. Try out the program**. Adjust the threshold and minimum are values in order to achieve the best result.** The outcome of this simple algorithm is far from ideal because small changes in lighting trigger false positive events. Below you can see samples about the outcome of the program.



2. Improved motion detection

In the previous motion detection program, we used the first frame of the video stream as background. This approach will not work if lighting conditions are changed (think of the times of a day) or objects will appear in the field of view permanently. To improve this problem, we will use the weighted average of earlier frames as background image. In this case, the program adapts to the varying background conditions. This method is still not perfect but better than the earlier approach. In this section you should **re-use the previous code** and perform some modifications inside it.

2.1. Change the code from section 1.7 as can be seen below. Now we also use the first frame of the video stream as the initial background but now the background is stored as a float array:

```
# if the average frame is None, initialize it
if background is None:
    background = gray.copy().astype("float")
    continue
```

2.2. Modify the earlier delta frame calculation. Here we convert back the float background array into byte array (grayscale image):

```
delta_frame = cv2.absdiff(gray, cv2.convertScaleAbs(background))
```

2.3. Accumulate the weighted average between the current frame and earlier frames. Here the weight value is 0.5 and the accumulated value will be stored in the background variable. Put this line of code before the visualization code section (1.12):

```
cv2.accumulateWeighted(gray, background, 0.5)
```

3. Tracking colored object

The goal of this exercise is to detect and track a colored object in a video stream using computer vision. In this exercise you will also **re-use some code parts** from section 2.

3.1. Import all libraries from 1.1 and the **numpy** as **np**.

3.2. Define some necessary variables and constants. The *buffsize* constant defines how many earlier center points will be stored to draw the motion track while the *indx* variable keeps tracks the index of the last stored point:

```
video_path = None

buffsize = 64
```

```
indx = 0
```

3.3. The key of this colored object tracking is a “color filter” where the filter will keep only those pixels of the image which fall into a predefined color range. The color range can be defined by its lower and upper boundaries. The adjustment in this sample code assumes green object(s). In addition, here we initialize a *numpy array* which will store previous **(x, y)** coordinates of the object center to visualize the object’s movement path.

```
# lower and upper boundaries of a "green" object in HSV color space

green_range = [(25, 55, 5), (65, 255, 255)]

#initialize the list of tracked points
path = np.zeros((buffsize, 2), dtype='int')
```

3.4. Create a reference to the video stream (same as 1.4)

3.5. Read images one by one from the video stream (same as 1.5 without the state variable)

3.6. Resize the frame, blur (or filter) it, and convert it into the *HSV* (Hue, Saturation, Value) color space (more information about the HSV space: https://en.wikipedia.org/wiki/HSL_and_HSV):

```
frame = imutils.resize(frame, width=500)

blur = cv2.GaussianBlur(frame, (9,9), 0)

hsv = cv2.cvtColor(blur, cv2.COLOR_BGR2HSV)
```

3.7. In this step we create a binary image (mask) using the above defined range. In the mask all pixels that are out of the range will be 0. Moreover, a morphological **opening** (erosion before dilation) operation will be performed on the mask to remove all small noises that may remained in the mask:

```
mask = cv2.inRange(hsv, green_range[0], green_range[1])

mask = cv2.erode(mask, None, iterations=2)

mask = cv2.dilate(mask, None, iterations=2)
```

3.8. Apply contour detection on the mask (same as 1.10).
3.9. Initialize a variable where the center coordinate of the object will be stored. If there is any contour in the mask, we will choose the largest contour and it will be used to calculate the bounding box or enclosing cycle of the object. In this sample code, we suppose that the object is *circular*:

```
if len(cnts) > 0:
    # find the largest contour in the mask
    cnt = max(cnts, key=cv2.contourArea)
    ((x, y), radius) = cv2.minEnclosingCircle(cnt)
```

3.10. To determine the centroid of an object, we use its **moments** (more information about image moments: https://www.learnopencv.com/shape-matching-using-hu-moments-c-python/)

```
    M = cv2.moments(cnt)
    center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))
```

3.11. Draw a cycle around the circular object if its radius is higher than the predefined threshold value. In addition, draw another filled cycle around the center of the object:

```
    if radius > 10:
        cv2.circle(frame, (int(x), int(y)), int(radius), (0, 255, 255), 2)
        cv2.circle(frame, center, 5, (0, 0, 255), -1)
```

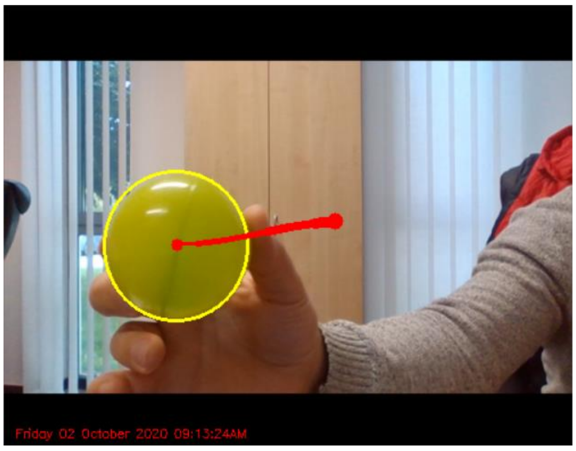
3.12. If the earlier defined numpy array (our point buffer) is full, shift it one element to the left and past the new center point at the end of the array:

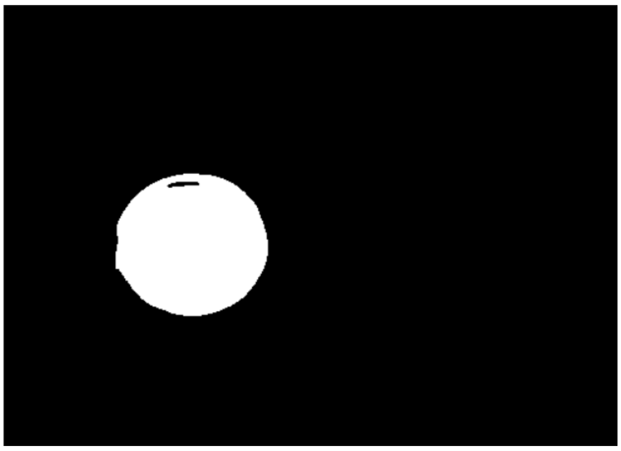
```
        # update the path list
        if indx < buffsize:
            path[indx] = (center[0], center[1])
            indx += 1
        else:
            path[0:indx-1] = path[1:indx]
            path[indx-1] = (center[0], center[1])
```

3.13. Draw the movement track of the object onto the image. The movement track is visualized as a sequence of connected lines where the thickness of lines depends on its endpoint indexes in the array:

```
for i in range(1, len(path)):
    # if either of the tracked points are None, ignore them
    if path[i - 1] is None or path[i] is None:
        continue
    # otherwise, compute thickness and draw lines
    thickness = int(np.sqrt(len(path) / float(i + 1)) * 2.5)
    cv2.line(frame, (path[i-1][0], path[i-1][1]), (path[i][0],
path[i][1]), (0, 0, 255), thickness)
```

3.14. Try out the program**.** To see the final result, you need a **green object**! A sample image can be seen below.





Task to be solved alone:

3.A. Modify the above code to detect red objects!

3.B. If the object disappears from the screen, the object's path is still visible. Eliminate the unnecessary track if the target object is not visible on the screen!

3.C. Change the appearance of the movement track. In the modified code, track thickness is reversed relative to the original version of the program. It means that, the track needs to be thickened toward the center of the object.

References

[1] Rosebrock, A. https://www.pyimagesearch.com/2015/09/14/ball-tracking-with-opencv/. Accessed on 03/01/2020.

[2] Rosebrock, A. https://www.pyimagesearch.com/2015/05/25/basic-motion-detection-and-tracking-with-python-and-opencv/. Accessed on 03/01/2020.

[3] Rosebrock, A. https://www.pyimagesearch.com/2015/06/01/home-surveillance-and-motion-detection-with-the-raspberry-pi-python-and-opencv/. Accessed on 03/01/2020.

This work was supported by the construction EFOP-3.4.3-16-2016-00021. The project was supported by the European Union, co-financed by the European Social Fund.