**Basic motion detection and tracking with Python and OpenCV**

Before starting this tutorial, you should aware of OpenCV Module, how to start video stream, capturing images and what is the use of OpenCV.

Open up a editor, create a new file, name it motion\_detector.py , and let’s get coding:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | # import the necessary packages  from imutils.video import VideoStream  import argparse  import datetime  import imutils  import time  import cv2    # construct the argument parser and parse the arguments  ap = argparse.ArgumentParser()  ap.add\_argument("-v", "--video", help="path to the video file")  ap.add\_argument("-a", "--min-area", type=int, default=500, help="minimum area size")  args = vars(ap.parse\_args())    # if the video argument is None, then we are reading from webcam  if args.get("video", None) is None:  vs = VideoStream(src=0).start()  time.sleep(2.0)    # otherwise, we are reading from a video file  else:  vs = cv2.VideoCapture(args["video"])    # initialize the first frame in the video stream  firstFrame = None |

**lines 2-7** import our necessary packages. All of these should look pretty familiar, except perhaps the imutils  package, which  is a set of convenience functions that I have created to make basic image processing tasks easier. If you do not already have [imutils](https://github.com/jrosebr1/imutils" \t "_blank) installed on your system, you can install it via pip: pip install imutils .

Next up, we’ll parse our [command line arguments](https://www.pyimagesearch.com/2018/03/12/python-argparse-command-line-arguments/) on **Lines 10-13**. We’ll define two switches here. The first, --video , is optional. It simply defines a path to a pre-recorded video file that we can detect motion in. If you do not supply a path to a video file, then OpenCV will utilize your webcam to detect motion.

We’ll also define --min-area , which is the minimum size (in pixels) for a region of an image to be considered actual “motion”. As I’ll discuss later in this tutorial, we’ll often find small regions of an image that have changed substantially, likely due to noise or changes in lighting conditions. In reality, these small regions are not actual motion at all — so we’ll define a minimum size of a region to combat and filter out these false-positives.

**Lines 16-22** handle grabbing a reference to our vs  object. In the case that a video file path is not supplied (**Lines 16-18**), we’ll grab a reference to the webcam and wait for it to warm up. And if a video file *is* supplied, then we’ll create a pointer to it on **Lines 21 and 22**.

Lastly, we’ll end this code snippet by defining a variable called firstFrame .

***Assumption:*** The first frame of our video file will contain no motion and just background — therefore, we can model the background of our video stream using only the first frame of the video.

Obviously we are making a pretty big assumption here. But again, our goal is to run this system on a Raspberry Pi, so we can’t get too complicated. And as you’ll see in the results section of this post, we are able to easily detect motion while tracking a person as they walk around the room.

27 # loop over the frames of the video

28 while True:

29 # grab the current frame and initialize the occupied/unoccupied

30 # text

31 frame = vs.read()

32 frame = frame if args.get("video", None) is None else frame[1]

33 text = "Unoccupied"

34

35 # if the frame could not be grabbed, then we have reached the end

36 # of the video

37 if frame is None:

38 break

39

40 # resize the frame, convert it to grayscale, and blur it

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

42 gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

43 gray = cv2.GaussianBlur(gray, (21, 21), 0)

44

45 # if the first frame is None, initialize it

46 if firstFrame is None:

47 firstFrame = gray

48 continue

So now that we have a reference to our video file/webcam stream, we can start looping over each of the frames on **Line 28**.

A call to vs.read()  on **Line 31**returns a frame that we ensure we are grabbing properly on **Line 32**.

We’ll also define a string named text  and initialize it to indicate that the room we are monitoring is “Unoccupied”. If there is indeed activity in the room, we can update this string.

And in the case that a frame is not successfully read from the video file, we’ll break from the loop on **Lines 37 and 38**.

Now we can start processing our frame and preparing it for motion analysis (**Lines 41-43**). We’ll first resize it down to have a width of 500 pixels — there is no need to process the large, raw images straight from the video stream. We’ll also convert the image to grayscale since color has no bearing on our motion detection algorithm. Finally, we’ll apply Gaussian blurring to smooth our images.

It’s important to understand that even consecutive frames of a video stream will not be identical!

Due to tiny variations in the digital camera sensors, no two frames will be 100% the same — some pixels will most certainly have different intensity values. That said, we need to account for this and apply Gaussian smoothing to average pixel intensities across an 21 x 21 region (**Line 43**). This helps smooth out high frequency noise that could throw our motion detection algorithm off.

As I mentioned above, we need to model the background of our image somehow. Again, we’ll make the assumption that the first frame of the video stream contains no motion and is a good example of what our background looks like. If the firstFrame  is not initialized, we’ll store it for reference and continue on to processing the next frame of the video stream (**Lines 46-48**).

Given this static background image, we’re now ready to actually perform motion detection and tracking:

# compute the absolute difference between the current frame and

50 # compute the absolute difference between the current frame and

51 # first frame

52 frameDelta = cv2.absdiff(firstFrame, gray)

53 thresh = cv2.threshold(frameDelta, 25, 255, cv2.THRESH\_BINARY)[1]

54

55 # dilate the thresholded image to fill in holes, then find contours

56 # on thresholded image

57 thresh = cv2.dilate(thresh, None, iterations=2)

58 cnts = cv2.findContours(thresh.copy(), cv2.RETR\_EXTERNAL,

59 cv2.CHAIN\_APPROX\_SIMPLE)

60 cnts = imutils.grab\_contours(cnts)

61

62 # loop over the contours

63 for c in cnts:

64 # if the contour is too small, ignore it

65 if cv2.contourArea(c) < args["min\_area"]:

66 continue

67

68 # compute the bounding box for the contour, draw it on the frame,

69 # and update the text

70 (x, y, w, h) = cv2.boundingRect(c)

71 cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)

72 text = "Occupied"

Now that we have our background modeled via the firstFrame  variable, we can utilize it to compute the difference between the initial frame and subsequent new frames from the video stream.

Computing the difference between two frames is a simple subtraction, where we take the absolute value of their corresponding pixel intensity differences (**Line 52**):

delta = |background\_model – current\_frame|

We’ll then threshold the frameDelta  on **Line 53** to reveal regions of the image that only have significant changes in pixel intensity values. If the delta is less than 25, we discard the pixel and set it to black (i.e. background). If the delta is greater than 25, we’ll set it to white (i.e. foreground).

Again, note that the background of the image is black, whereas the foreground (and where the motion is taking place) is white.

Given this thresholded image, it’s simple to apply contour detection to to find the outlines of these white regions (**Lines 58-60**).

We start looping over each of the contours on **Line 63**, where we’ll filter the small, irrelevant contours on **Line 65 and 66**.

If the contour area is larger than our supplied --min-area , we’ll draw the bounding box surrounding the foreground and motion region on **Lines 70 and 71**. We’ll also update ourtext  status string to indicate that the room is “Occupied”.

74 # draw the text and timestamp on the frame

75 cv2.putText(frame, "Room Status: {}".format(text), (10, 20),

76 cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 0, 255), 2)

77 cv2.putText(frame, datetime.datetime.now().strftime("%A %d %B %Y %I:%M:%S%p"),

78 (10, frame.shape[0] - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 0.35, (0, 0, 255), 1)

79

80 # show the frame and record if the user presses a key

81 cv2.imshow("Security Feed", frame)

82 cv2.imshow("Thresh", thresh)

83 cv2.imshow("Frame Delta", frameDelta)

84 key = cv2.waitKey(1) & 0xFF

85

86 # if the `q` key is pressed, break from the lop

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

88 break

89

90 # cleanup the camera and close any open windows

91 vs.stop() if args.get("video", None) is None else vs.release()

92 cv2.destroyAllWindows()

The remainder of this example simply wraps everything up. We draw the room status on the image in the top-left corner, followed by a timestamp (to make it feel like “real” security footage) on the bottom-left.

**Lines 81-83** display the results of our work, allowing us to visualize if any motion was detected in our video, along with the frame delta and thresholded image so we can debug our script.

***Note:*** If you download the code to this post and intend to apply it to your own video files, you’ll likely need to tune the values for *cv2.threshold*  and the *--min-area*  argument to obtain the best results for your lighting conditions.

Finally, **Lines 91 and 92** cleanup and release the video stream pointer.