# Biztonsagi kamera:

Security camera with motion detection and tracking.

Tested on ParrotOS/Linux and Windows10

Dependencies:

- Python 3.9.7
- OpenCV-Python
- Numpy
- Matplotlib.Pyplot
- Argparse
- os
- sys

### Installing dependencies

Use `pip install -r requirements.txt`

## Modules:

- `main.py`
  - The software can be started with the `main.py` file, that takes additional arguments.
- `bgsub.py`
  - This module uses background subtraction to show motion.
- `occupation.py`
  - This module helps with detecting motion.
- `track_object.py`
  - Module for motion detection and motion tracking
- `rec.py`
  - Easy to use video recording module with user guide.
- `optical_flow.py`
  - Just an optical flow testing script, for later use.

## User guide:

```
python.exe .\src\main.py --help
usage: main.py [-h] [--input INPUT] [--algo {KNN,MOG2}]

Security Camera with motion detection.(quit with 'q')

optional arguments:
  -h, --help         show this help message and exit
  --input INPUT      path to the video source (webcam is default).
  --algo {KNN,MOG2}  Choose the background subtraction algorythm MOG2 or
```

```
KNN
```

## main.py

This is the user interface of the program. The user can specify the video source input, the algorithm of the background subtraction and the type of the motion detecting.
And the library i have used for this, is `argparse`.

## bgsub.py

This is the brain of the program. The `main.py` module calls `bgsub.py` to start the whole process.
`bgsub.py` takes agruments `bgsub(vsrc, algo)`. `vsrc` is the path to the video file or the camera. `algo` is the backgroundsubtraction algorythm, it can be MOG2 or KNN.

### Using the Backgroundsubtractor classes:

```
backSub = cv.createBackgroundSubtractorMOG2() # for MOG2

backSub = cv.createBackgroundSubtractorKNN() # for KNN

fgMask = backSub.apply(frame, learningRate=-1) # obtain forground mask of
video stream
```

These constructors can take various arguments for fine tune your background subtraction quality.
But they do the job witouth any arguments aswell.
`backSub.apply(frame, learningRate=-1)`
frame: is a frame from the video learningRate: specify the learning rate of background model (0 - 1), -1 is for automatically chosen
learning rate

### VideoCapture

```
capture = cv.VideoCapture(vsrc) # get stream from source
ret, frame = capture.read() # obtain frame from video stream
cv.imshow('VideoFrame', frame) # show the frame obtained from the
videostream
```

### Optical Flow

For motion tracking, i used the Optical Flow algorithm with Lukas Kanade method.
Optical flow is the pattern of apparent motion of image objects
between two consecutive frames caused by the movement of object or camera.
It is 2D vector field where each vector is a displacement vector
showing the movement of points from first frame to second.
Lucas-Kanade method takes a 3x3 patch around the point.
So all the 9 points have the same motion.
So now our problem becomes solving 9 equations with two unknown variables which is over-determined. A better solution is obtained with least square fit method.

To decide which points to track, we use cv.goodFeaturesToTrack().
We take the first frame, detect some Shi-Tomasi corner points in it,
then we iteratively track those points using Lucas-Kanade optical flow.

**Parameter to pass for Shi-Tomasi corner detection and Lucas Kanade optical flow**

```
    # params for ShiTomasi corner detection
    feature_params = dict( maxCorners = 100,
                           qualityLevel = 0.3,
                           minDistance = 7,
                           blockSize = 7 )

    # params for lucas kanade optical flow
    lk_params = dict( winSize = (15,15),
                      maxLevel = 2,
                      criteria = (cv.TERM_CRITERIA_EPS |
cv.TERM_CRITERIA_COUNT, 10, 0.03))
```

**Implementing Optical Flow**

For the algorith, we must pass an old frame and the next frame.
On these frames we must find the points to track.
Then the algorithm compares the points on the old frame
and the next frame. Every point found has a status number,
saved in a vector `st`. If the point on the old frame found
on the next frame, the point's status set to 1, otherwise 0.
With this `st` vector, we can see which points can we draw.
At the end we update the old frame, and the points.

```
            # find corners on first frame
            old_frame = frame
            old_gray = cv.cvtColor(old_frame, cv.COLOR_BGR2GRAY)
            p0 = cv.goodFeaturesToTrack(old_gray, mask = fgMask,
**feature_params)

            # convert frame to gray
            frame_gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
```

```
            # ensuring opencv wont crash, calcOpticalFlow only when there
are points to track
            if p0 is not None:
                # calculate optical flow
                p1, st, err = cv.calcOpticalFlowPyrLK(old_gray, frame_gray,
p0, None, **lk_params)

                # Select good points
                if p1 is not None:
                    good_new = p1[st==1]
                    good_old = p0[st==1]

                # draw vectors
                for i,(new,old) in enumerate(zip(good_new, good_old)):
                    a,b = new.ravel()
                    c,d = old.ravel()
                    mask = cv.line(mask, (int(a),int(b)), (int(c),int(d)),
color[i].tolist(), 4)
                    frame = cv.circle(frame, (int(a),int(b)), 5,
color[i].tolist(), -1)

                # Now update the previous frame and previous points
                old_gray = frame_gray.copy()
                p0 = good_new.reshape(-1,1,2)
```

## occupation.py

The purpose of this module, to make sure if theres any major object moving in the frame,
for example: animals, humans, bicycles, cars, etc...
This way we dont have to appy any contour detecion algorythm, so we can save precious resources.
`is_occupied(frame, threshold)`
Frame: the input image, this should be a foreground mask
Threshold: with this argument, we can adjust the sensitivity of the detection in range 0 - 10000.
the smaller the number, the smaller motion can be detected.

```
hist = cv.calcHist([frame], [0], None, [2], [0,256]) # calculating the
histogram of the binary image

summa = sum(hist[:,0]) # sum of the pixel intensity

if (hist[1]/summa)*10000 > threshold:
    return True
return False
```

In this code example, we calculate the histogram of the forground mask.
And if there is enought white pixels in the mask (this we can adjust with the threshold argument)

the function returns True.