Image Processing

Project Report

Team 10

Hassan Osama BN:16 sec:1

Hussein Youssef BN:17 sec:1

Momen Hassan BN:13 sec:2

Mohammed Magdy BN:22 sec:2

Used algorithms

- Background removal (implemented)
  - ⇨ In the setup phase, The player is instructed to stand still in front of the camera and Raise his arms aside in a specific position (forming a T shape) like the Rio de Janeiro statue. This process takes 5 seconds for the user to be ready.
  - ⇨ A background removal is done by selected frames subtraction.
- Noise removal
  - ⇨ Dilation and erosion filters are applied to eliminate the discontinuities in the arms and remove the scattered random noise in the remaining sections of the image.
- Arm detection using rasterization (implemented and highly optimized)
  - ⇨ We generated vertical rasterization lines to detect the height and width of the rectangle containing each arm for starters. After comparing this rectangle with the rio de janeiro frame, most ares of the background in each arm is removed. After that, We use a threshold to apply vertical dynamic cropping to the image.

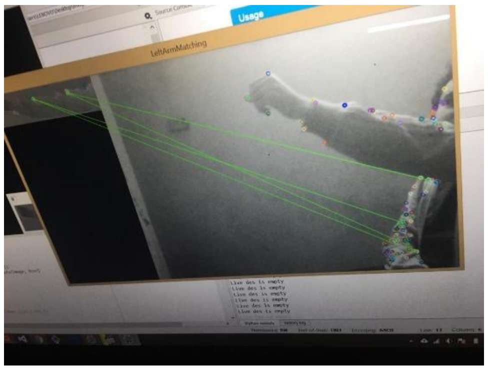
    Lastly, for the feature extraction process to be accurate, We replaced the removed background areas with a calculated average of the border pixels of each arm. Thus, Elimination any unwanted keypoints and assuring that all the features are INSIDE the arm.
  - ⇨ If an error for some reason occurred during this process, The user gets an error message and after 5 seconds the un-detected arm gets the chance to re-calculate.
- ORB feature extraction
  - ⇨ After passing some trials with SIFT,SURF, Fast accompanied with BRISK and BREIF algorithms, We landed on the holy land of ORB algorithm. It’s clearly the fastest and the most customizable feature extracting algorithm we tested. We considered hog at the beginning and already tried it but turns

out its non-scalable property is actually very unscalable. So we didn’t give it much thought.

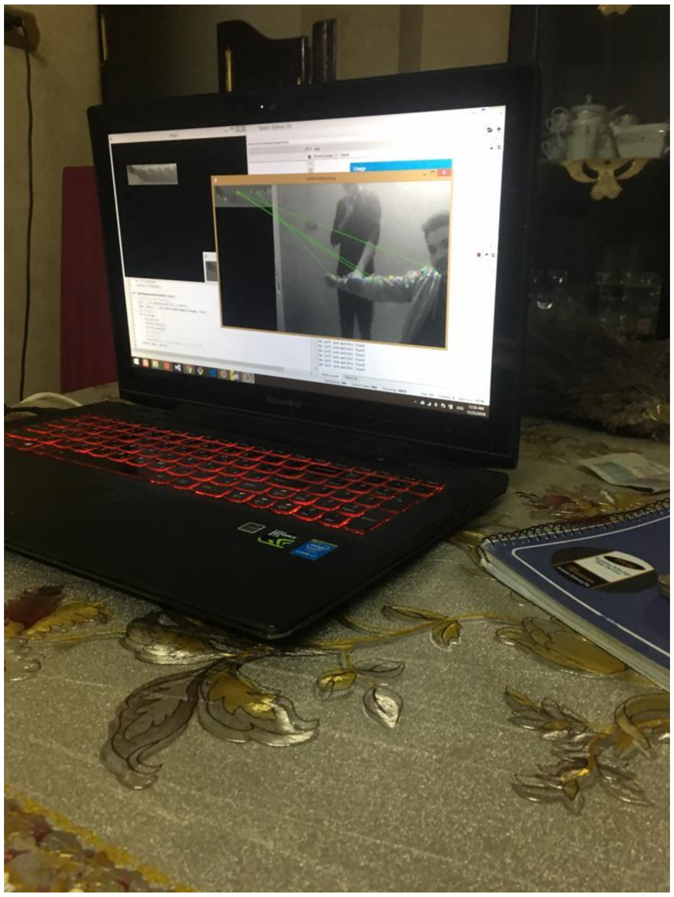
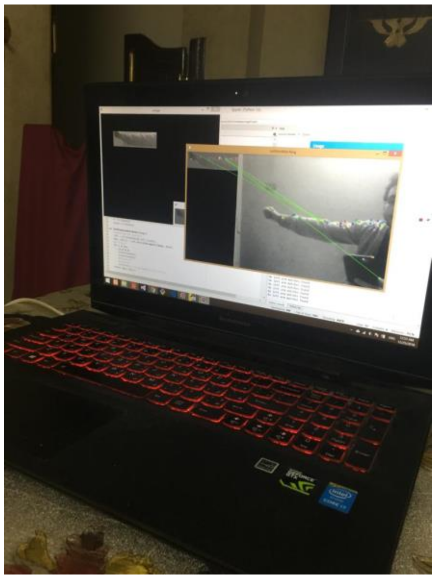
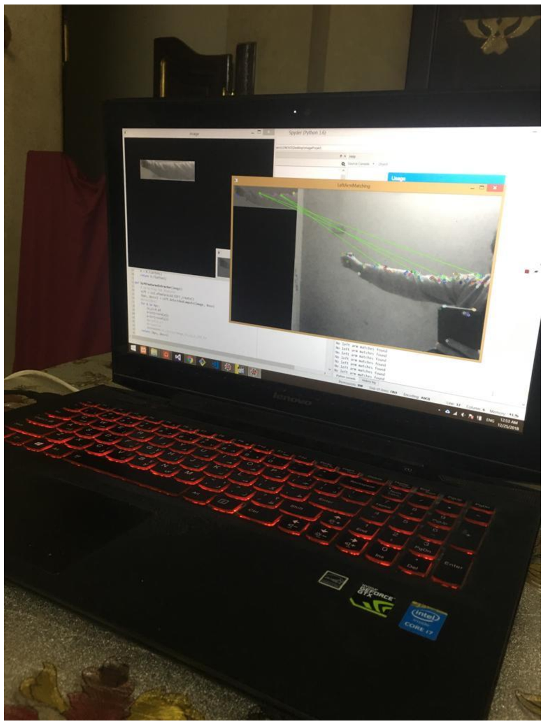
- ⇨ Feature extraction is basically applied to each stored arm ONLY ONCE and applied for each frame.
- ⇨ We were supposed to extract the foreground in each frame for better results but we’re still working on it at the moment.

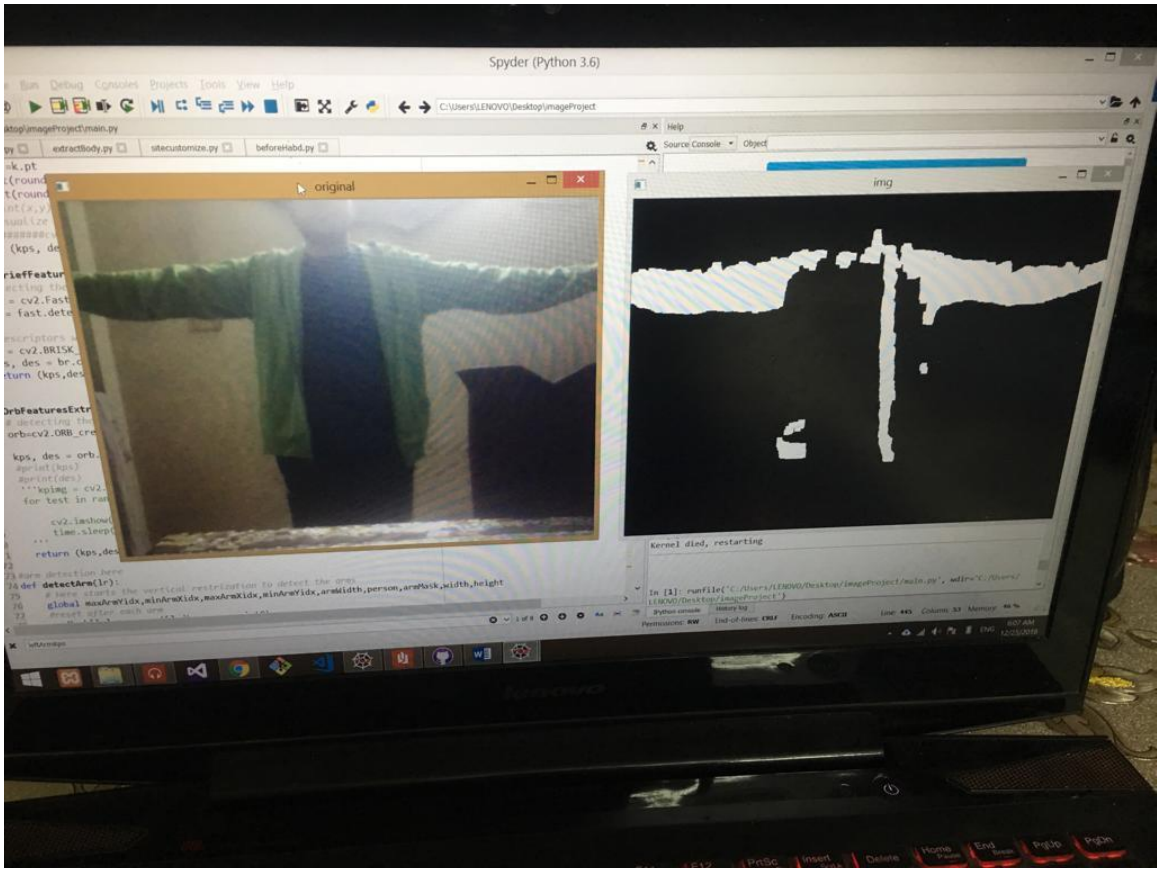
- BF Matching and tracking
  - ⇨ In each iteration of the gameplay, we simply match the descriptors of each arm with the live frame descriptors using brute forcing algorithm. After that we filter those matches to keep the good matches only using a threshold.

- Homography algorithm
  - ⇨ This algorithm is supposed to transform the matches in each frame to use the inlier and the outlier matches to detect the object but after trying this out and visualizing the result with a polygon drawing function, The result was so annoying that we discarded it and created a manual function to calculate the global average of the matches indices and draw it manually for more visualizable result.



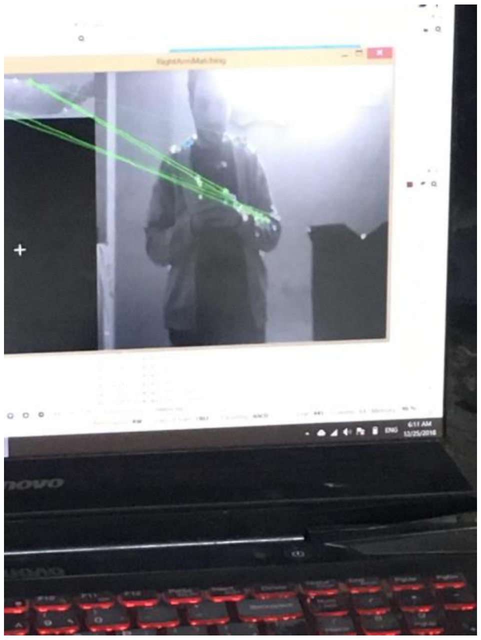
```
Live des is empty
Live des is empty
Live des is empty
Live des is empty
Live des is empty
Live des is empty
```

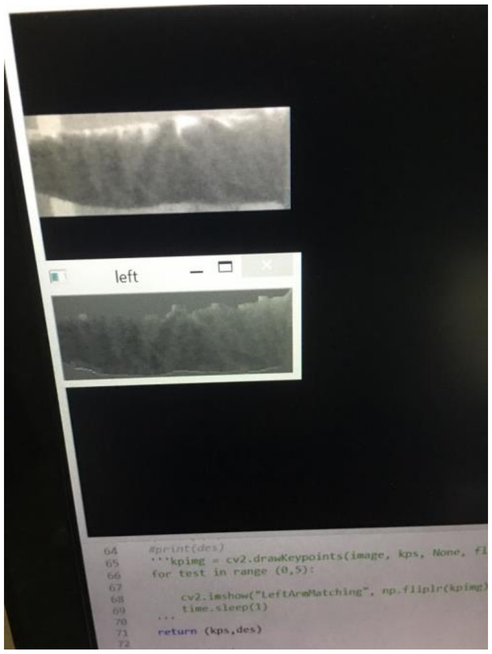






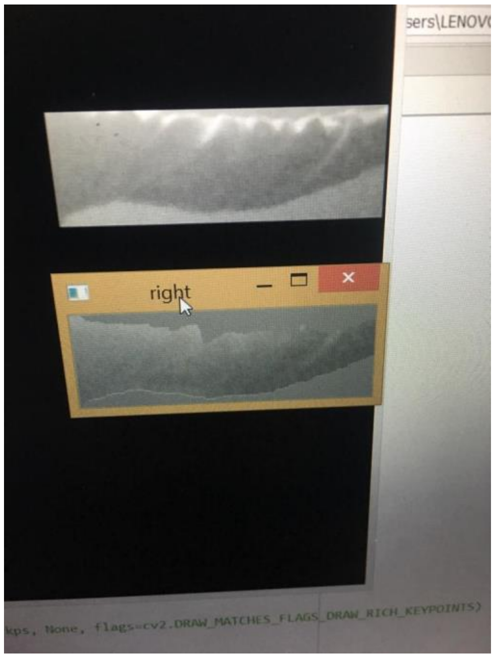
```
 kps, des = orb.
 #print(kps)
 #print(des)
 '''kpimg = cv2.
 for test in ran

     cv2.imshow(
     time.sleep(
 '''
 return (kps,des

#arm detection here
def detectArm(lr):
    # here starts the vertical restriction to detect the arms
    global maxArmYidx,minArmXidx,maxArmXidx,minArmYidx,armWidth,person,armMask,width,height
    #reset after each arm
```

```
Kernel died, restarting

In [1]: runfile('C:/Users/LENOVO/Desktop/imageProject/main.py', wdir='C:/Users/LENOVO/Desktop/imageProject')
```





```
    #print(des)
    '''kpimg = cv2.drawKeypoints(image, kps, None, fl
    for test in range (0,5):

        cv2.imshow("LeftArmMatching", np.fliplr(kpimg)
        time.sleep(1)
    '''
    return (kps,des)
```



```
kps, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```

Experiment results and analysis

Those pictures show the results of the left arm live matching and detection with the green lines after the optimization. When we used SIFT for the feature extraction, it wasn't very fast for the real time application. So we replaced it with ORB and the results were better. The ORB extraction is more controllable for each case needs. We Tried FlannMatcher as well for the same matching cause but it needed different formats for the descriptors for each feature extraction algorithm which weakens the modularity of our code so we preferred BF matching.

Points of strength

- Arms detection is almost perfect when light settings are adjusted
- Fps visualization is high
- Matching algorithms work greatly but unfortunately not very steadily.
- Most of the code is built from scratch without any open source.
- Changing the background complexity and the camera view doesn't affect the output result because of the optimization of the arm extraction.
- The code is module and FOP so there's no dummy pasting for similar repeated operations.
- Almost all the errors we faced during our tests are handled with proper error messages and the program hardly crashes.

Points of weakness

- No great results if the foreground and background are exactly the same color or within a tiny threshold
- The output isn't exactly in the expected format as ideas look pretty promising but their application is very hard.
- Both arms have very close descriptors so if both hands are inside the frame the output is nearly symmetrical around each hand, which is good in a way.

- No segmentation at each frame at this moment but we'll try to complete it before the discussion.

Work division between team members

We worked in small team of 2, A team worked with the detection parts and the other team worked with the matching and visualization sector.

Accuracy, performance

The application runs in real time performance and the accuracy is quite satisfying, for the environment we've been working in at least.

Conclusion and references

- ORB is perfect for real time applications
- Background based detections is very sensitive to the environment conditions and camera focus and those aren't very desirable properties.
- If the foreground and the background are very close in their graylevels, Things get a little edgy. But complex backgrounds aren't much of a problem. Light sources can be devastating though.
- The result could be highly optimized if we used the built-in function cv2::segcut() to extract the foreground in each frame but it's very computationally high as it works iteratively and ruins the real time flow.
- In the Key nearest neighbors matching algorithm, for some reason, some weird reason, the train and query descriptors are reversed in the documentation but we handled it anyway.
- Using Harris algorithm in ORB to modify the keypoints extracted by FAST algorithm is a huge improvement in the result and speed.
- The only open source code we used was the primitive functions in open CV and those are the official documentation links we used

- https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html
- https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_orb/py_orb.html
- https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html