IMAGE PROCESSING

CSE-4019

MOTION TRACKING

PROJECT REPORT

UNDER THE GUIDANCE OF

Prof. SWATHI J.N

BY

ARYAN PATEL (18BCE0897)
KUNWAR ABHISHEK SINGH (18BCE2176)
YASHASWI D RATTAN (18BCE2152)

ABSTRACT

An adjustment in the estimation of speed or vector of an object or objects in the field of view is called motion. Identification of motion can be accomplished by electronic gadgets or mechanical gadgets that interact or measure the adjustments in the given condition. Four motion detectors are being compared to find the most suitable one to be used in an application

INTRODUCTION

Object Tracking
The objective of object tracking is to keep watch on something. Regularly based upon or in a collaboration with object detection and recognition, tracking algorithms are intended to find (and keep a relentless watch on) a moving object (or many moving items) after some time in a video stream.

There's an location history of the object (following dependably handles outlines in relationship to each other) which enables us to know how its position has changed after some time. Also, that implies we have a model of the object's movement. A Kalman channel, an arrangement of scientific conditions, can be utilized to decide the future area of a object. By utilizing a progression of estimations set aside a few minutes, this algorithm gives a way to evaluating past, present and future states.

Object tracking is the process of:

- Taking an initial set of object detections (such as an input set of bounding box coordinates)
- Creating a unique ID for each of the initial detections
- And then tracking each of the objects as they move around frames in a video, maintaining the assignment of unique IDs

Furthermore, object tracking allows us to **apply a unique ID to each tracked object**, making it possible for us to count unique objects in a video.

An ideal object tracking algorithm will:

- Only require the object detection phase once (i.e., when the object is initially detected)
- Will be extremely fast — *much* faster than running the actual object detector itself
- Be able to handle when the tracked object "disappears" or moves outside the boundaries of the video frame
- Be robust to occlusion
- Be able to pick up objects it has "lost" in between frames

We will implement **centroid tracking with OpenCV**, an easy to understand, yet highly effective tracking algorithm. Simple face detection and object tracking with OpenCV

LITERATURE REVIEW

There has been incalculable of work done regarding object tracking from single object to multiple object tracking, either using python or any other programming language with different additional libraries/module like dlib, pandas, etc. Analysing the some of the past works helped us in learning more about object tracking.

[7] Performance Evaluation of Object Tracking Algorithms

This paper manages the non-minor issue of execution assessment of object tracking. The author proposes a rich arrangement of measurements to evaluate diverse parts of the execution of object tracking. The paper utilizes six distinctive video successions that speak to an assortment of difficulties to outline the functional estimation of the proposed measurements by assessing and comparing two object tracking algorithm.

[8] Real-Time Object Tracking and Classification Using a Static Camera

Understanding objects in video data is of particular interest due to its enhanced automation in public security surveillance as well as in traffic control and pedestrian flow analysis. In this paper, a system is presented which is able to detect and classify people and vehicles in different weather conditions using a static camera. The system is capable of correctly tracking multiple objects despite object interactions. Results are then presented by online application of the algorithm.

HARWARE

- Windows-10, i7 processor, 8gb ram
- Webcam/Camera

SOFTWARE

- PyCharm 2018.2.3
- Additional libraries:
    - Opencv
    - Numpy
    - Argparse
    - Imutiles
    - time

METHOD

The object tracking algorithm used is called *centroid tracking* as it relies on the Euclidean distance between (1) *existing* object centroids (i.e., objects the centroid tracker has already seen before) and (2) new object centroids between subsequent frames in a video.

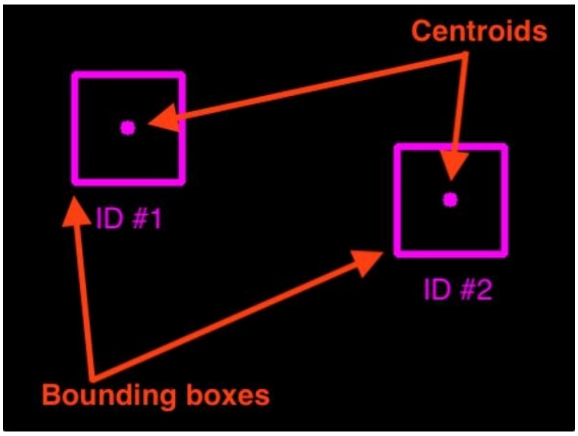
We implanted a Python class to contain this centroid tracking algorithm and then create a Python script to actually run the program and apply it to input videos.

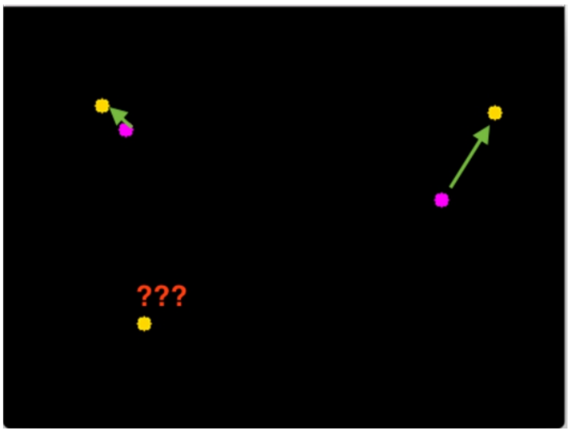
The centroid tracking algorithm accept that we are going in an arrangement of bouncing box (x, y)-coordinates for each distinguished object in each and every casing.

These jumping boxes can be created by a object locator you might want (color thresholding + contour extraction, Haar falls, HOG + Linear SVM, SSDs, Faster R-CNNs, and so forth.), gave that they are figured to each frame in the video. When we have the bouncing box facilitates we should figure the "centroid", or all the more just, the middle (x, y) of the bouncing box.

For each resulting casing in our video stream we apply the previous step of figuring object centroids; be that as it may, rather than relegating another one of a kind ID to each distinguished question (which would nullify the point of object tracking), we first need to decide whether we can connect the new object centroids with the old object centroids. To achieve this procedure, we process the Euclidean separation between each combine of existing item centroids and input object centroids.

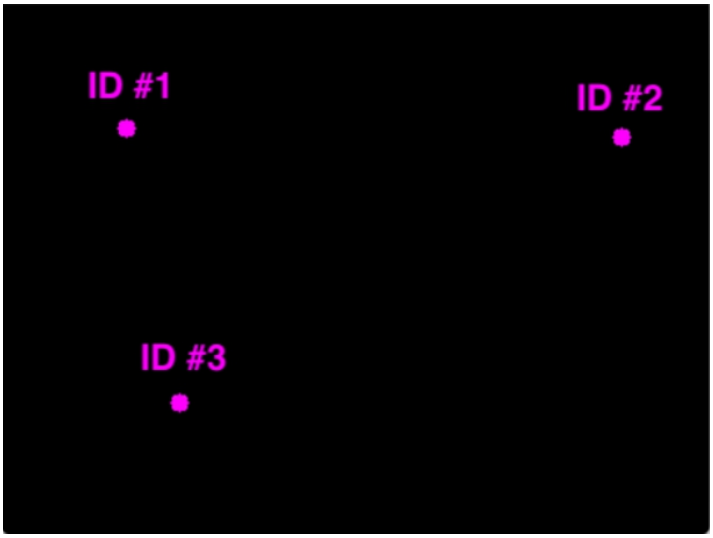
The primary assumption of the centroid tracking algorithm is that a given object will potentially move in between subsequent frames, but the *distance* between the centroids for frames F_t and F_{t+1} will be *smaller* than all other distances between objects. Hence, if we choose to associate centroids with min distances between the frames which then created our object tracker.





In the event that there are more input detections than existing objects being tracked, we have to register the new object. “Registering” simply means that we are adding the new object to our list of tracked objects by:

- Assigning it a new object ID
- Storing the centroid of the bounding box coordinates for that object



To use this object tracker specifically to detect the face we have used two files
.prototxt and .caffemodel which are part of OpenCV deep learning face detector. However, we can also use another form of detection.

CODE

CODE for Centroid Tracker:

```
# import the necessary packages
from scipy.spatial import distance as dist
from collections import OrderedDict
import numpy as np

class CentroidTracker():
    def_init_(self, maxDisappeared=50):
        # initialize the next unique object ID along with two 
ordered
        # dictionaries used to keep track of mapping a given 
object
        # ID to its centroid and number of consecutive frames it 
has
        # been marked as "disappeared", respectively
        self.nextObjectID = 0
        self.objects = OrderedDict()
        self.disappeared = OrderedDict()

        # store the number of maximum consecutive frames a given
        # object is allowed to be marked as "disappeared" until 
we
        # need to deregister the object from tracking
        self.maxDisappeared = maxDisappeared

    def register(self, centroid):
        # when registering an object we use the next available 
object
        # ID to store the centroid
        self.objects[self.nextObjectID] = centroid
        self.disappeared[self.nextObjectID] = 0
        self.nextObjectID += 1

    def deregister(self, objectID):
        # to deregister an object ID we delete the object ID 
from
        # both of our respective dictionaries
        del self.objects[objectID]
        del self.disappeared[objectID]

    def update(self, rects):
        # check to see if the list of input bounding box 
rectangles
        # is empty
        if len(rects) == 0:
            # loop over any existing tracked objects and mark 
them
            # as disappeared
```

```
            for objectID in self.disappeared.keys():
                self.disappeared[objectID] += 1

                # if we have reached a maximum number of
consecutive
                # frames where a given object has been marked as
                # missing, deregister it
                if self.disappeared[objectID] >
self.maxDisappeared:
                    self.deregister(objectID)

            # return early as there are no centroids or tracking
info
            # to update
            return self.objects

        # initialize an array of input centroids for the current
frame
        inputCentroids = np.zeros((len(rects), 2), dtype="int")
        # loop over the bounding box rectangles
        for (i, (startX, startY, endX, endY)) in
enumerate(rects):
            # use the bounding box coordinates to derive the
centroid
            cX = int((startX + endX) / 2.0)
            cY = int((startY + endY) / 2.0)
            inputCentroids[i] = (cX, cY)

        # if we are currently not tracking any objects take the
input
        # centroids and register each of them
        if len(self.objects) == 0:
            for i in range(0, len(inputCentroids)):
                self.register(inputCentroids[i])

        # otherwise, are currently tracking objects so we need
to
        # try to match the input centroids to existing object
        # centroids
        else:
            # grab the set of object IDs and corresponding
centroids
            objectIDs = list(self.objects.keys())
            objectCentroids = list(self.objects.values())

            # compute the distance between each pair of object
            # centroids and input centroids, respectively -- our
            # goal will be to match an input centroid to an
existing
            # object centroid
```

```
        D = dist.cdist(np.array(objectCentroids),
inputCentroids)

        # in order to perform this matching we must (1) find
the
        # smallest value in each row and then (2) sort the
row
        # indexes based on their minimum values so that the
row
        # with the smallest value as at the *front* of the
index
        # list
        rows = D.min(axis=1).argsort()

        # next, we perform a similar process on the columns
by
        # finding the smallest value in each column and then
        # sorting using the previously computed row index
list
        cols = D.argmin(axis=1)[rows]

        # in order to determine if we need to update,
register,
        # or deregister an object we need to keep track of
which
        # of the rows and column indexes we have already
examined
        usedRows = set()
        usedCols = set()

        # loop over the combination of the (row, column)
index
        # tuples
        for (row, col) in zip(rows, cols):
            # if we have already examined either the row or
            # column value before, ignore it
            # val
            if row in usedRows or col in usedCols:
                continue

            # otherwise, grab the object ID for the current
row,
            # set its new centroid, and reset the disappeared
            # counter
            objectID = objectIDs[row]
            self.objects[objectID] = inputCentroids[col]
            self.disappeared[objectID] = 0

            # indicate that we have examined each of the row
and
            # column indexes, respectively
```

```
            usedRows.add(row)
            usedCols.add(col)

         # compute both the row and column index we have NOT
yet
         # examined
         unusedRows = set(range(0,
D.shape[0])).difference(usedRows)
         unusedCols = set(range(0,
D.shape[1])).difference(usedCols)

         # in the event that the number of object centroids is
         # equal or greater than the number of input centroids
         # we need to check and see if some of these objects
have
         # potentially disappeared
         if D.shape[0] >= D.shape[1]:
            # loop over the unused row indexes
            for row in unusedRows:
               # grab the object ID for the corresponding row
               # index and increment the disappeared counter
               objectID = objectIDs[row]
               self.disappeared[objectID] += 1

               # check to see if the number of consecutive
               # frames the object has been marked
"disappeared"
               # for warrants deregistering the object
               if self.disappeared[objectID] >
self.maxDisappeared:
                  self.deregister(objectID)

         # otherwise, if the number of input centroids is
greater
         # than the number of existing object centroids we
need to
         # register each new input centroid as a trackable
object
         else:
            for col in unusedCols:
               self.register(inputCentroids[col])

      # return the set of trackable objects
      return self.objects
```

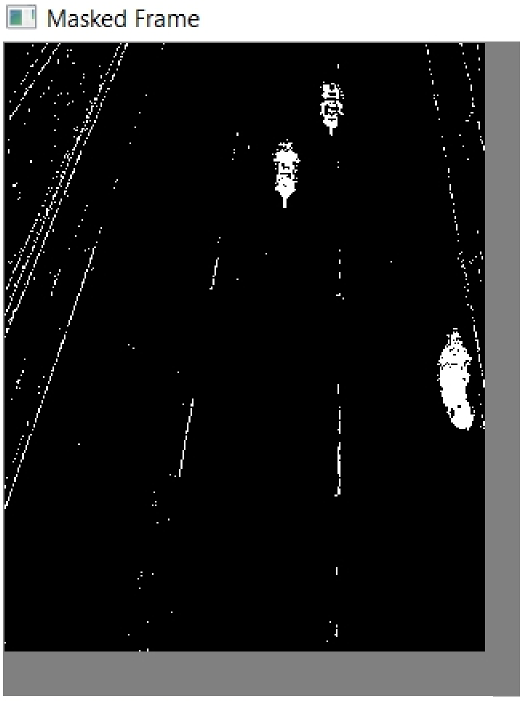
CODE for Motion Detection:

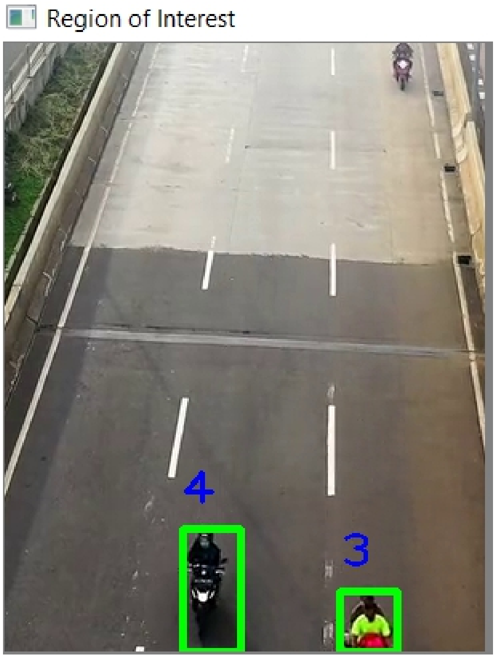
```
# USAGE
# python object_tracker.py --prototxt deploy.prototxt --model
res10_300x300_ssd_iter_140000.caffemodel

# import the necessary packages
from pyimagesearch.centroidtracker import CentroidTracker
from imutils.video import VideoStream
import numpy as np
import argparse
import imutils
import time
import cv2

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-p", "--prototxt", required=True,
   help="path to Caffe 'deploy' prototxt file")
ap.add_argument("-m", "--model", required=True,
   help="path to Caffe pre-trained model")
ap.add_argument("-c", "--confidence", type=float, default=0.5,
   help="minimum probability to filter weak detections")
args = vars(ap.parse_args())

# initialize our centroid tracker and frame dimensions
ct = CentroidTracker()
(H, W) = (None, None)

# load our serialized model from disk
print("[INFO] loading model...")
net = cv2.dnn.readNetFromCaffe(args["prototxt"],
args["model"])

# initialize the video stream and allow the camera sensor to
warmup
print("[INFO] starting video stream...")
vs = VideoStream(src=0).start()
time.sleep(2.0)

# loop over the frames from the video stream
while True:
   # read the next frame from the video stream and resize it
   frame = vs.read()
   frame = imutils.resize(frame, width=400)

   # if the frame dimensions are None, grab them
   if W is None or H is None:
      (H, W) = frame.shape[:2]

   # construct a blob from the frame, pass it through the
```

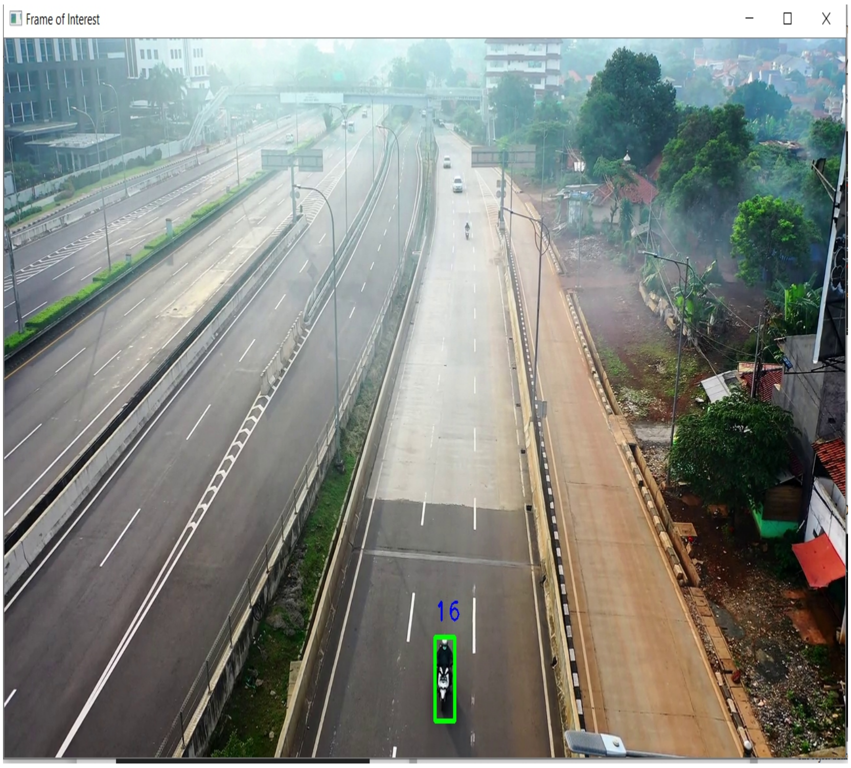
```
network,
   # obtain our output predictions, and initialize the list of
   # bounding box rectangles
   blob = cv2.dnn.blobFromImage(frame, 1.0, (W, H),
      (104.0, 177.0, 123.0))
   net.setInput(blob)
   detections = net.forward()
   rects = []

   # loop over the detections
   for i in range(0, detections.shape[2]):
      if detections[0, 0, i, 2] > args["confidence"]:
         # compute the (x, y)-coordinates of the bounding box
for
         # the object, then update the bounding box rectangles
list
         box = detections[0, 0, i, 3:7] * np.array([W, H, W,
H])
         rects.append(box.astype("int"))
         # draw a bounding box surrounding the object so we
can
         # visualize it
         (startX, startY, endX, endY) = box.astype("int")
         cv2.rectangle(frame, (startX, startY), (endX, endY),
            (0, 255, 0), 2)

   # update our centroid tracker using the computed set of
bounding
   # box rectangles
   objects = ct.update(rects)

   # loop over the tracked objects
   for (objectID, centroid) in objects.items():
      # draw both the ID of the object and the centroid of the
      # object on the output frame
      text = "ID {}".format(objectID)
      cv2.putText(frame, text, (centroid[0] - 10, centroid[1]
- 10),
         cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
      cv2.circle(frame, (centroid[0], centroid[1]), 4, (0,
255, 0), -1)

   # show the output frame
   cv2.imshow("Frame", frame)
   key = cv2.waitKey(1) & 0xFF

   # if the `q` key was pressed, break from the loop
   if key == ord("q"):
      break
```

```
# do a bit of cleanup
cv2.destroyAllWindows()
vs.stop()
```

SAMPLE OUTPUT

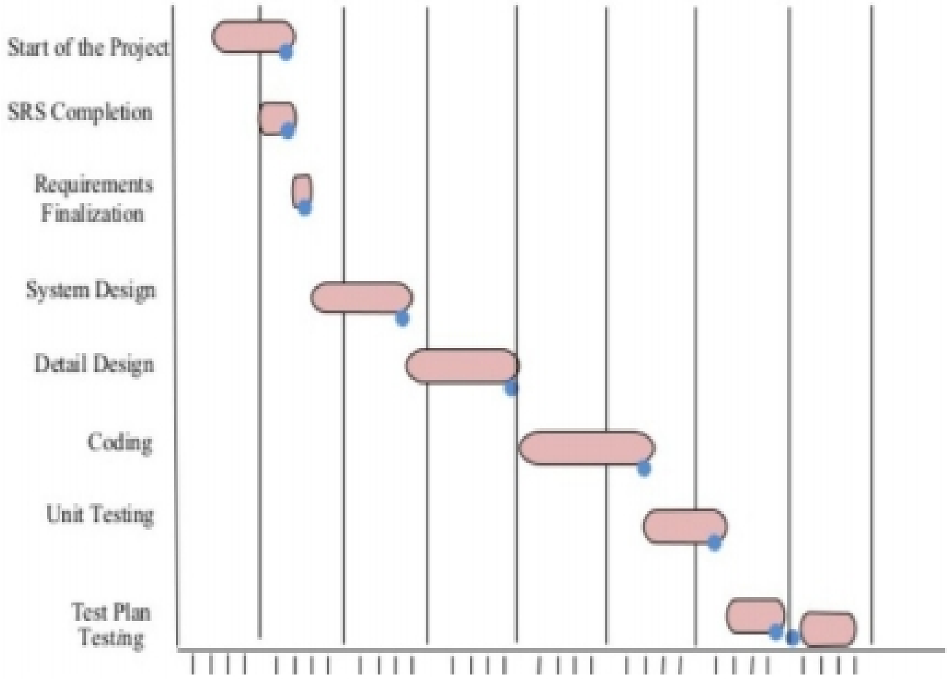






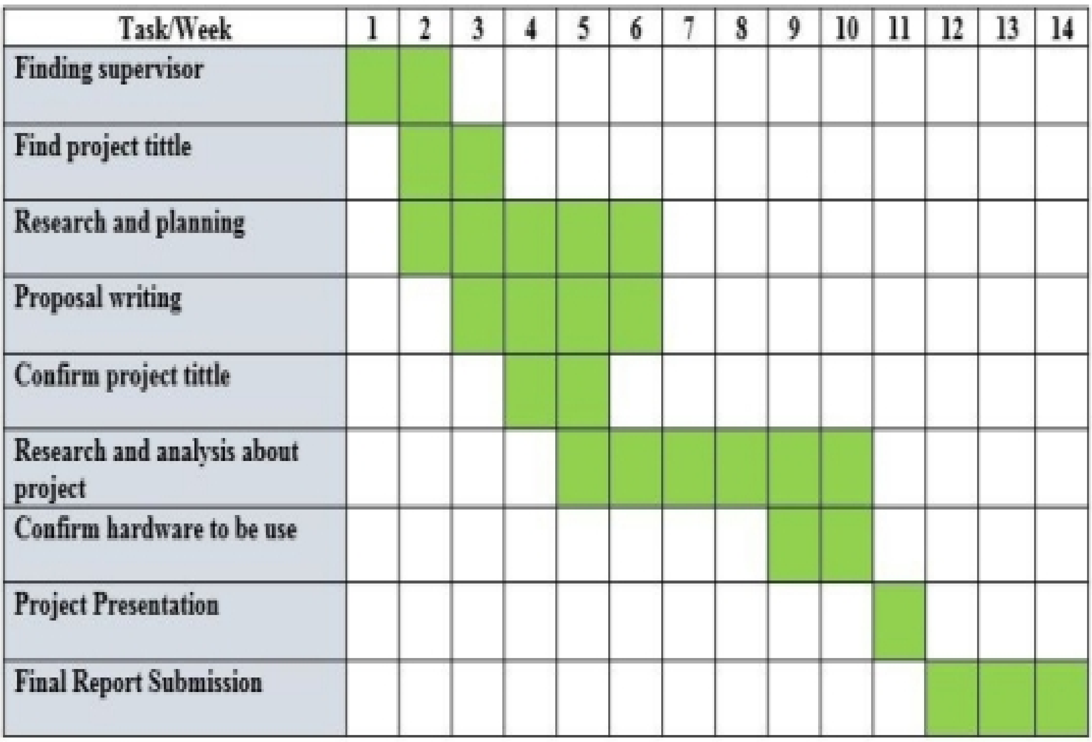
The object tracker can be used to detect more than one face. However, if an object is removed from the frame the tracker will remain till the object has existed outside the field of view of tracker for more than 50 frames, then the object will be deregistered.

WORK TIMELINE



GNATT CHART

[redacted]



REQUIREMENTS

HARDWARE

- Intel Core i3 CPU@ 2.40 Ghz
- RAM: 2 GB

SOFTWARE

- Python
- OS-Windows Microsoft
- Microsoft Visual Studio

RESULT

While our centroid tracker worked great in this example, there are two primary drawbacks of this object tracking algorithm. The first is that it requires that object detection step to be run on every frame of the input video.

The second drawback is related to the underlying assumptions of the centroid tracking algorithm itself — *centroids must lie close together between subsequent frames*.

- This assumption typically holds, but keep in mind we are representing our 3D world with 2D frames — **what happens when an object overlaps with another one?** The answer is that **object ID switching could occur**.
- If two or more objects overlap each other to the point where their centroids intersect and instead have the minimum distance to the other respective object, the algorithm may (unknowingly) swap the object ID.
- However, the problem is more pronounced with centroid tracking as we relying strictly on the Euclidean distances between centroids and no additional metrics, heuristics, or learned patterns.

CONCLUSION

From this project we learned that how to perform object tracking using OpenCV library and Centroid tracking algorithm. The centroid tracking algorithm works by:

- Accepting bounding box coordinates for each object in every frame
- Computing the Euclidean distance between the centroids of the *input* bounding boxes and the centroids of *existing* objects that we already have examined.
- Updating the tracked object centroids to their new centroid locations based on the new centroid with the smallest Euclidean distance.
- And if necessary, marking objects as either "disappeared" or deregistering them completely.

The centroid tracking used has two primary cons:

- We have to run the tracker for every frame of the video.
- Overlapping of objects is not properly handled due to the Euclidean distance and the ids of the objects might end up being swapped.

Despite of its downsides, the tracker is still very efficient with some advantages of its own (1) since we can control the environment of where it is used, there is less worry of objects overlapping and (2) we can use it in real-tim

REFERENCES

[1] https://github.com/soeaver/caffe-model

[2] https://www.pyimagesearch.com/2015/05/25/basic-motion-detection-and- tracking-with-python-and-opencv/

[3] Anderson, F. (1990, September). Real time, video image centroid tracker. In *Acquisition, Tracking, and Pointing IV* (Vol. 1304, p. 82). International Society for Optics and Photonics.

[4] Venkateswarlu, R., Sujata, K. V., & Rao, B. V. (1992, November). Centroid tracker and aimpoint selection. In *Acquisition, Tracking, and Pointing VI* (Vol. 1697, pp. 520-530). International Society for Optics and Photonics.

[5] https://www.pyimagesearch.com/2018/07/30/opencv-object-tracking/

[6] https://github.com/lazyoracle/motion-detection

[7] Yin, F., Makris, D., & Velastin, S. A. (2007, October). Performance evaluation of object tracking algorithms. In *IEEE International Workshop on Performance Evaluation of Tracking and Surveillance, Rio De Janeiro, Brazil* (p. 25).

[8] Johnsen, S., & Tews, A. (2009, May). Real-time object tracking and classification using a static camera. In *Proceedings of IEEE International Conference on Robotics and Automation, workshop on People Detection and Tracking*.