Real-time Ball Tracking and Detection System

ECE-GY 6183 | Digital Signal Processing Laboratory
Fall 2021

Name: Sagar Patel **NETID:** SP5894 **Email:** sp5894@nyu.edu

---

GitHub: https://github.com/Sagar-py/Real-time-Ball-Tracking

Table of Contents

Overview 2
Motivation 2
Libraries Used 4
Understanding the Code 5
Implementation 6
Screenshots 7
Possible Extensions 10
References 10

Overview

I have been a fan of cricket for almost two decades now and ever since I can remember, there has been the concept of the third umpire in the sport. The third umpire uses the technology of the hot spot and slow-motion replays at different angles to gain information and make decisions. Therefore, they use several image and video processing techniques to help make a decision that may sometimes be inaccurate by the two umpires present on the field.

In cricket, there is a rule that you can not intercept the path of the ball towards the wicket without any other means than a bat. Now, should you stop the movement of the balls with your legs and the ball was said to be in a projection to hit the wickets (those three sticks behind the player), you will be considered out? The question arises – even the third umpire, how does he figure out that the ball was in projection to hit the wickets? For this, the sport uses computer vision tools to track the ball, isolate it and create a projection of the ball's trajectory.

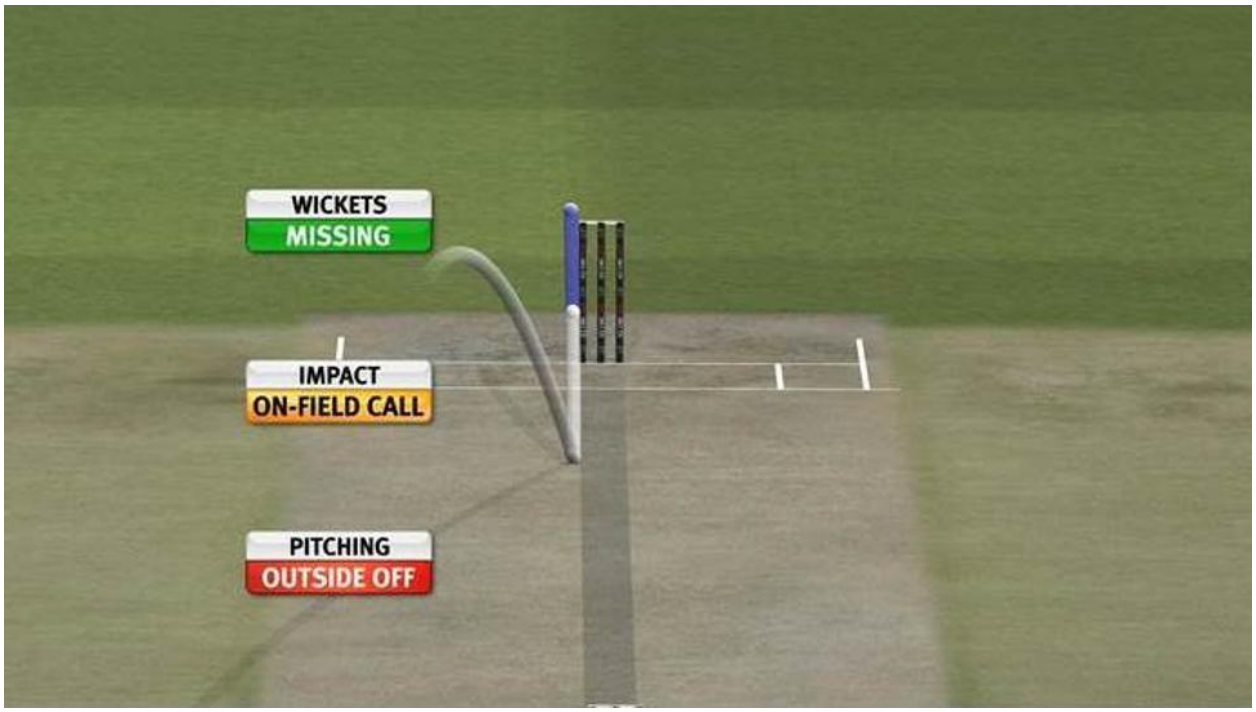

Image source: medium.com

This is where OpenCV comes into the picture. The white path is the track on which the ball since its detection zone has its path covered and the blue part is after the crease (the second horizontal white line below the three sticks).

Motivation

The capability to track moving objects and analyze them plays a crucial role in sports for a few reasons –

1. The number of objects and people moving in every direction with different speeds and sometimes unpredictable paths.

2. The background mixes with players and balls, objects, and people with nearly identical appearances.
3. The camera is constantly panning, zooming, tilting and quite often a replay appears from a radically different angle makes it difficult to focus on the ball for certain viewers.

Image source: tenor.com

As you can see in these particular images, it is hard to keep a track of the ball as the camera is constantly panning and zooming thereby making it difficult to detect the ball let alone track its trajectory.

Another example can be shown in the second image where it is a bit more evidence to track the ball because of its rather large size and the camera angles being placed correctly. However, it would be more efficient if there were a tracker installed which helps you track the movement of the ball.

NOTE: In both examples, the color of the ball is white but for the sake of our experiment, I have used the color ‘green’. Also because that was the only mono-colored ball I had with me for the demonstration.

Image source: edios.ai

Libraries Used

The packages imported in the program are –

```
import numpy as np
import time
import cv2

import argparse
import imutils

from imutils.video import VideoStream
from collections import deque
```

The reason why we install **`imutils`** is that it is a series of convenience functions to make basic image processing functions such as translation, rotation, resizing, skeletonization, displaying Matplotlib images, sorting contours, detecting edges, and much easier with OpenCV and both Python 2.7 and Python 3. (Source: pypi.org)

`deque` is a list-like data structure with superfast appends and pops to maintain a list of past (x, y) coordinates of the ball's center in the real-time stream. Maintaining such a queue allows us to draw the

trail of the ball.

Understanding the Code

In the argument segment, we have introduced two arguments in the form of switches. The first one is the **`--video`** which is the functionality that helps us trail a ball on a pre-loaded video. I often used that argument to test and try different values for the ball's color range, trail length, etc. That said, the video functionality can be used for actual modifications.
The second argument, **`--buffer`** is the maximum size of the list-like data structure which maintains a list of previous (x, y) coordinates of the ball we are tracking. This allows us to draw the trail of the ball. A smaller queue will create a shorter trail and a larger one would create a longer trail since more points will be tracked. The value of our deque is 128.

The variables **`UpperRange`** and **`LowerRange`** specify the RGB range of the color that needs to be tracked. We have taken a range from (62, 255, 255) to (29, 88, 10) which is approximately from cyan to dark/olive green-ish color. The objective is to convert the RGB space into HSV space using the **`range detector`** function in the **`imutils`** library.

From there, we grab access to a pointer **`videoStream`**. If a --video switch was not enabled, then we grab reference to our webcam by using the **`videoStream`** threaded class for efficiency. Otherwise, if a video path was provided, then we open it for reading the file using **`cv2.VideoCapture`**.

A loop is begun that will continue till we press the 'e' key if we are workin with a video switch and control + c if we are working with the webcam. That means, till the end there are no frames found.

A call to the read method is made by our webcam pointer which then returns a tuple. The first entry is an indication of whether the **`frames`** were successfully read and that said, it returns a boolean value. The variable **`frames`** is the video frame itself.
In a situation where we are reading from a video file and the frame is not successfully read, then we know that we are at the end of the video and can break from the **`while`** loop.

In order to preprocess **`frames`** a bit, first we resize the frame to have a width of approximately 600 pixels. Downsizing the **`frames`** allows us to process the frame faster, leading to an increase in Frames per second since we have less image data that needs to be processed. We will then blur the **`frames`** to reduce high frequency noise and allow us to focus on the structural objects inside the **`frames`**, such as the ball. Finally, we will convert the **`frames`** to the HSV color space.

After preprocessing each frame, we then handle the localization of the ball (green, in color) by making a call to the **`inRange`** function of OpenCV. We first supply the lower HSV color the boundaries for the color green, followed by the upper HSV boundaries. The output of that is a **mask**. If we see a white circle, that means we have detected the ball. A series of erosions and dialations remove any small blobs that may be left on the mask.

In order to perform an outline for the ball, we make the function compatible with all versions of OpenCV. We will also initialize the center (x, y) coordinates of the ball to None. A check to ensure that at least one outline was found in the frame is introduced. Now, if at least one outline was detected, we find the largest outline **`LContours`** list. Compute the minimum enclosing circle of the blob, and then compute the (x, y) coordinates. Another check to ensure that the radius of the minimum enclosing circle is sufficiently large. Provided that the **`radius`** passes the test, we then draw two circles – one surrounding the ball and another to indicate the centroid of the ball.

In order to draw the trail, we start looping over each of the **`points`**. If either the current point or the previous point is None, then we ignore the current index to continue looping over the **`points`**. Provided that both the points are valid, we compute the **`thickness`** of the trail and then draw it on the **`frames`**.

The remaining part is just detecting any key presses to close the file, or terminate the code, etc.

Implementation

The implementation for this particular understanding can be broken down into a step-by-step approach –

Step 1: Running an argument passer that helps us know whether I want to use real-time detection and tracking or use the approach on a recorded file. --video is the path to our example video file. I have commented on the line as my primary objective is to focus on real-time tracking. However, I was interested in extending the application to recorded video as well.

Step 2: Read the dimensions and color range of the ball in RGB format (I am taking a range from cyan to dark green).

Step 3: Reading the video frame using **`cv2.VideoCapture()`** and if we want to access the webcam version, then we run **`cv2.VideoStream().`** Note that if we are running a file, it will be running on a loop till there is a manual intervention to quit.

Step 4: Create a bounding box(/circle) around the ball that is green, mask it, and perform a series of dilations to remove any blobs left in the mask.

Step 5: Once we find the bounding box and the centroid, we start moving forward by tracing the path for a span of 4 seconds.

Step 6: To begin executing the code, we can choose what we would like to execute. If we want a live video (real-time) stream, then on the command terminal, we need to execute the following command –

```
python3 rtbt_final.py
```

If we are running a file, make sure that the file is in the same directory as that of the code and that can be executed by –

```
python3 --video rtbt_final.py filename.mp4
```

Notice that there will be a delay of 3 seconds before the webcam or the file open up. The reason why I have done so is that oftentimes, a short lag of execution messes up the memory in the local machine.

Step 7: End the Videostream by pressing **`control + c`** (in Mac) and **`Ctrl + c`** (in Windows) or the key 'e' to end the video loop.

Screenshots

The screenshots with multiple scenarios have been attached below –

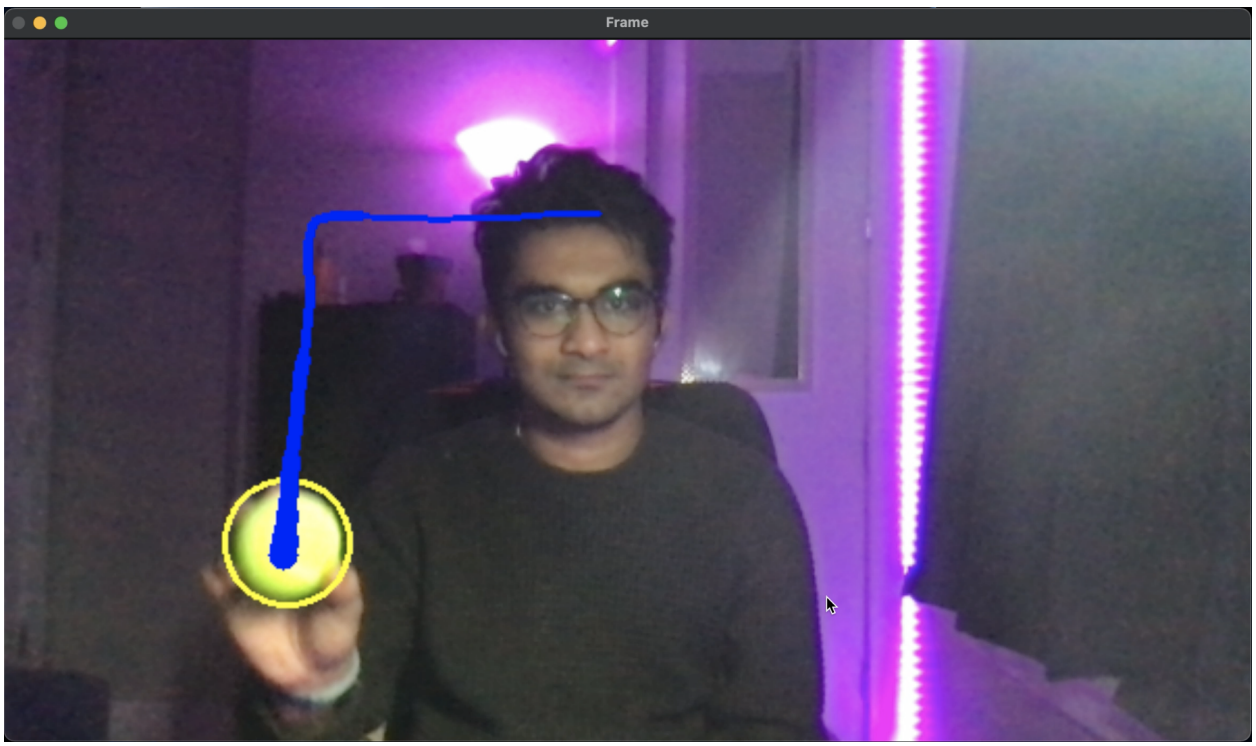

Image 1: Detecting and Tracking one ball - 1

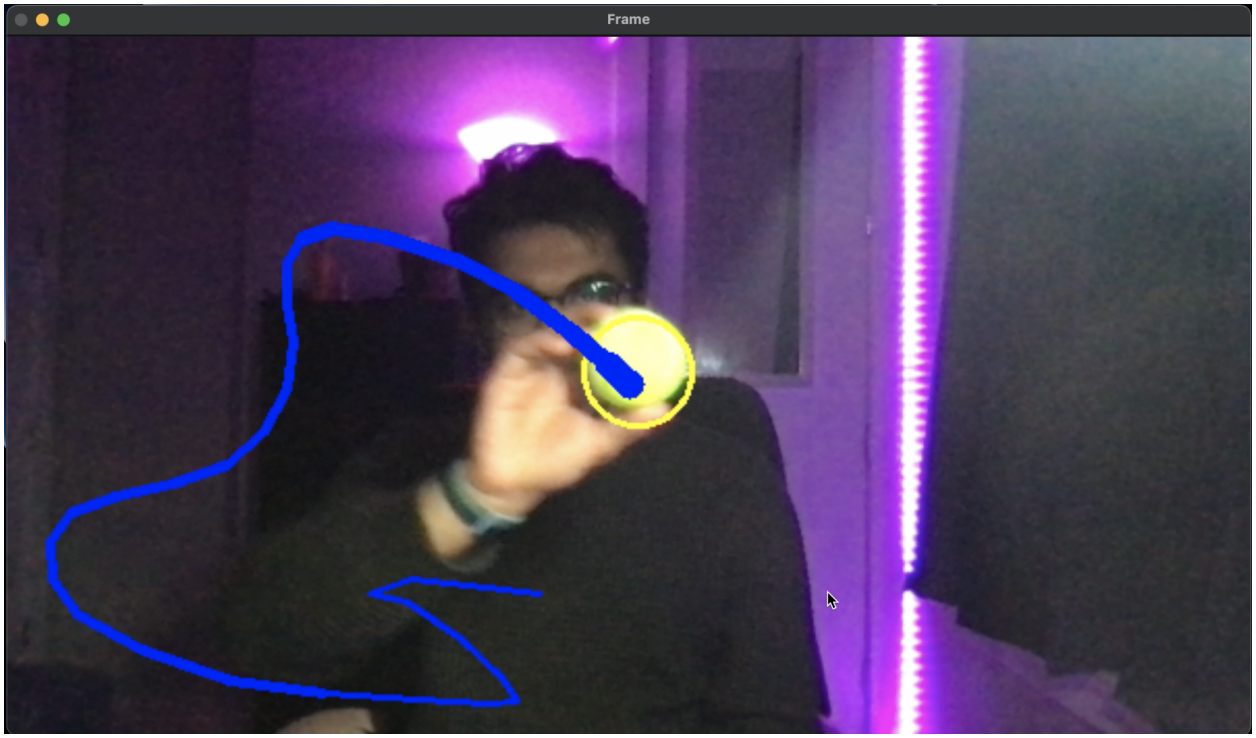

Image 2: Detecting and Tracking one ball - 2

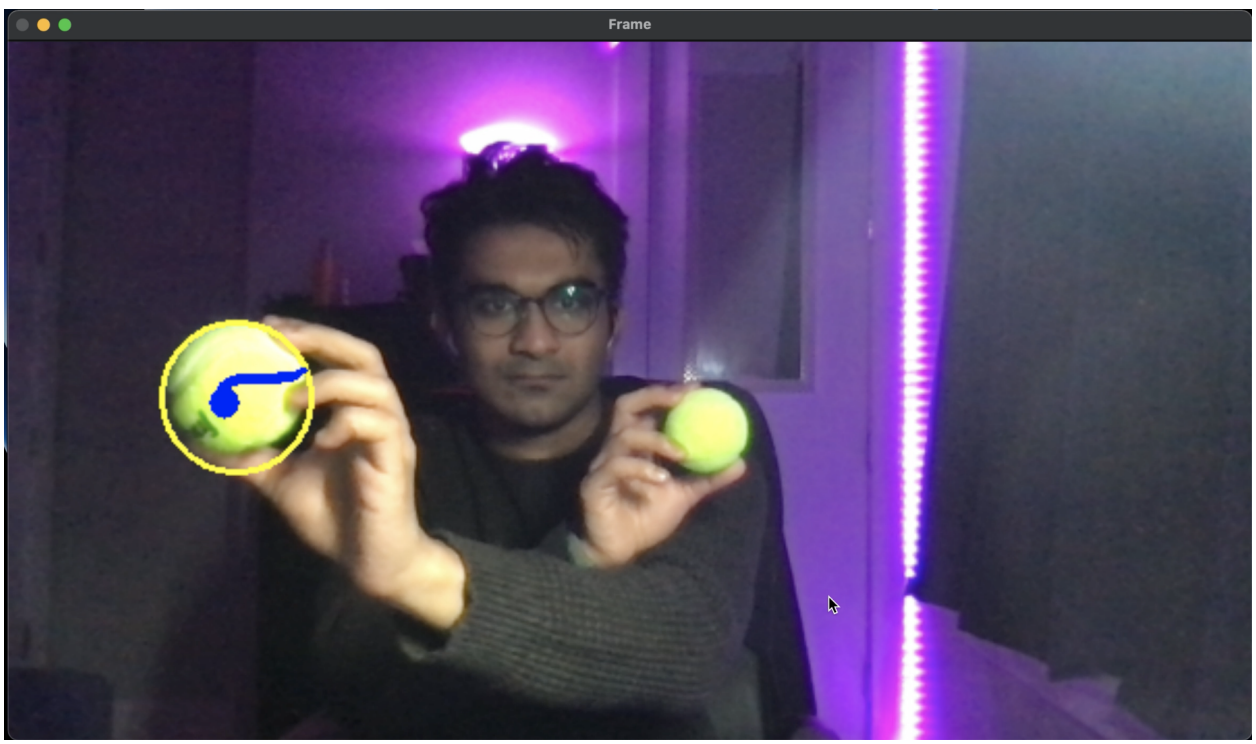

Image 3: Having two balls of the same color. Notice that the ball closer to the webcam is the one that gets detected

Image 4: When two balls are combined. It forms one giant contour because the algorithm is unable to detect it as one circle.

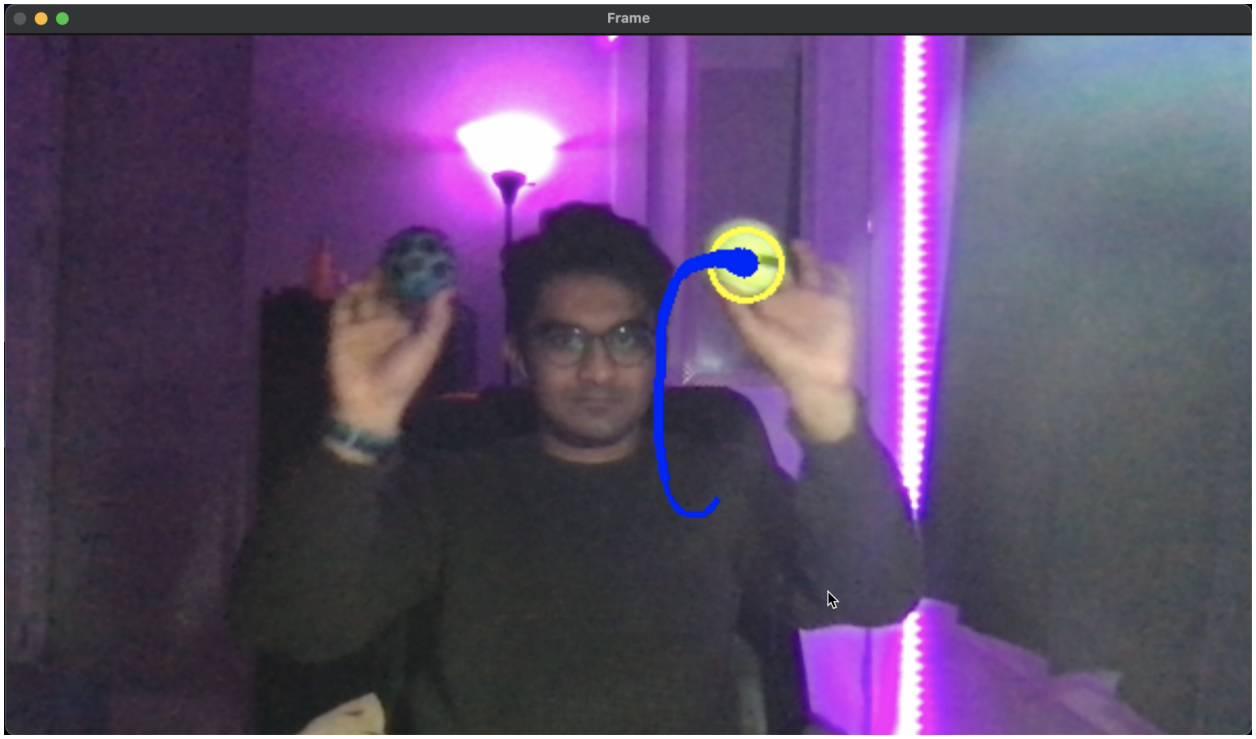

Image 5: A green ball and a blue ball in the frame. Only the green ball gets detected.

Possible Extensions

- Can implement Machine Learning to help with the predictive analysis and determine the path of the ball with previous patterns.
- Detect patterns of multiple balls in a single frame rather than just one. Maybe expand the application to a sport that requires the viewers to focus on more than one ball at a time.

References

[1] OpenCV with Python by Dr. Jha and Dr. Bibhuti Bhusan Biswal (https://amzn.to/32cKI4N)

[2] Object Tracking Using OpenCV by Ramji Balasubramanian (https://bit.ly/3p9TNnS)

[3] Technology for Cricket (https://bit.ly/33ysviG)