OpenCV Ball Tracking Tutorial

Neython Lec Streitz
nlecstreitz@oxy.edu
Occidental College

1 Introduction

As an introduction to computer vision techniques, I attempted a tutorial on using the OpenCV Python library to implement a video-based ball tracking script. OpenCV is a open-source Python library with a variety of tools for image processing and computer vision tasks. The project was first started in 1999 out of Intel, with a first release in 2000. Interestingly, OpenCV takes advantage of another well-known Python library, NumPy. As a result, it is quite simple to integrate other data science libraries such as Matplotlib, Pandas, and SciPy.

There were three main goals for this tutorial. Firstly, I learned how to detect a brightly colored ball using contour detection techniques. Secondly, I learned learn how to track the detected ball's movement in a video, visualizing its path by drawing a contrail based on its previous positions. Lastly, I learned how to generate and export a .csv file of the ball's positions throughout the video, as well as a .svg plot of the ball's theta values versus time.

1.1 Methods

To allow the script to run using either a pre-recorded video or a live webcam feed, I began by handling the parsing of command line arguments. Specifically, I added arguments for an optional video file path as well as a way to change the buffer size for the deque that holds the ball's previous positions. The use of a deque here is important, as it functions as a list-like data structure with rapid appends and pops. The deque holds the last positions of the tracked ball so that we can draw the contrail of the ball smoothly as it moves. A larger buffer size allows for a longer contrail while a smaller buffer size means a shorter contrail.

Once arguments are parsed, we have a few more things to initialize. Firstly, we must specify the color of the ball we want to track. To do so, I set the lower and upper RGB bounds to (29,86,6) and (64, 255, 255) respectively. To track a differently colored ball (that is still distinct from the environment), one simply changes the lower and upper bounds to match. Furthermore, we create an empty deque based on the set buffer size. Lastly, we create a new Pandas DataFrame to hold information on the X and Y positions of the ball as well as the current time in the video. This DataFrame will be used later to generate the time vs theta plot and ball position file.

1.2 Preprocessing

Assuming no pre-recorded video was supplied, we begin ball detection by instantiating the camera. Either way, we also grab the current time at the beginning of our input video. In a loop, we repeatedly grab the current camera frame as well as the change in time since the first time was grabbed. For a pre-recorded video, the end of the video will result in no frame being grabbed, so we break the loop and exit. Otherwise, we being frame preprocessing.

Firstly, we resize the frame to a width of 600px. This is done intentionally, as downsizing the frame leads to an increase in frames per second (FPS) given that there is less image data to process. This increase in FPS is somewhat marginal within a realistic frame size. Next, we blur the frame to reduce any high frequency noise. The focus here is on shapes and not details. Lastly, the frame is converted from BGR to HSV color space. Hue, Saturation, Value (HSV) is more robust than BGR for object detection since it (specifically, hue) is less affected by changes in external lighting. This encompasses frame pre-processing.

1.3 Ball Localization

To detect the ball, we begin by creating a binary mask for the ball color. In our case, this mask detects shapes within a range of the color green. The output of the mask is a black and white frame with blobs within the acceptable green range in white and everything else in black. Multiple iterations of dilations and erosions remove any remaining inconsistencies in the mask.

1.4 Contour Detection

Once the frame has been pre-processed and masked, contour detection takes place. The center location of the ball is kept empty until we verify contour detection success. If a contour is successfully detected, we proceed by only keeping the largest contour found and draw the smallest enclosing circle around it. Only now is the center location of the

ball set. For one last check, the radius of the ball is computed and checked against a somewhat arbitrary value of 10px. This is done to ensure the radius is large enough to realistically be a ball. If valid, we draw a circle around the ball's contour as well as on the center of the ball. These values are saved in our DataFrame and in our deque each time.

1.5 Contrail Drawing

After each successfully contour/ball detection, we loop over our deque to draw the contrail. We take the currently accessed entry and the previous one and draw a line between them. Obviously, if either value is empty, we skip over it. The range of the loop is set to the length of the deque, which is why the initialized buffer size determines the length of the drawn contrail.

1.6 Output Generation

In total, frame pre-processing, ball localization, contour detection, and contrail drawing are done successively and continually until the video ends of a user quits the script. By repeatedly following each step, the ball is detected, tracked, and drawn on screen.

Finally, we generate a .csv file of the ball's position, time, and theta values. To account for camera focal length and shifting of x and y-axis, we apply a -3px shift to the x-axis and a 20px shift to the y-axis. To calculate theta, we take the inverse tangent of each y-axis value multiplied by the corresponding real world pixel length (0.0000762/h). In this case, h represents the camera's focal length.

Why calculate theta and compute it over time? If we record our ball from a top-down or side-on perspective, calculating theta over time gives us an approximation of the angular velocity of the ball. That is, the rotation rate of our detected ball. With that said, this final process completes our OpenCV ball tracking tutorial.

2 Evaluation

In a general sense, the tutorial was a success. A standard Wilson tennis ball is correctly detected and tracked by the final script. Unfortunately, it is difficult to determine the exact accuracy of the ball tracking. Camera focal length and shift was estimated and corrected for using standard measurements and not exact specifications. Furthermore, the ball used was quite yellow and not the deep green used in the tutorial.

Nonetheless, because the goal of this tutorial was to simply track the ball on-screen and not state exact measurements, it was a resounding success. All features were tested including using live webcam feed and pre-recorded video input as well as changing smaller and larger buffer sizes. On the buffer feature, there is some unwanted behavior when the ball is held at a standstill, since the contrail is repeatedly drawn in the ball's last position and as the positions are so close to each other, they essentially "bunch-up" making the contrail look like its not tracking the ball. To remove this bunching behavior, further work might include computing the position distance and only drawing the line if the previous few positions are far enough apart to signify actual ball movement. Yet, this might decrease the smoothness of the contrail during smooth, continuous motion.

Moving the ball at steady and changing paces throughout the screen draws an impressive and smooth contrail. The ball can even be tracked when thrown, but suffers in some performance since catching the ball obscures it with fingers. Additionally, when the ball moves out of frame, the script attempts to detect other, smaller contours until eventually clearing the screen. These two cases cause the only significant issues with performance. As stated before, from the standpoint of tracking and drawing a contour over a ball, following the tutorial was successful.

3 Discussion

In terms of utility, this tutorial showcased the ease and accuracy of contour detection. For my senior project, I hope to use a similar methodology. While I will not be tracking a specified color, I will be contour detecting with QR codes. This means there is an additional step where I will be verifying the inner codification of the QR code. I believe this will increase the robustness of object detection since I will not be relying on an arbitrary contour radius to validate the contour. It is also less likely that multiple visible QR codes will exist within a frame as opposed to various green objects.

What will be much more important for my project than was for this tutorial will be accuracy. The actual detection of the barbell will be a smaller step as the end goal of the script is to track the velocity of the barbell and not just the shape of the barbell as it moves. With that in mind, there are some remaining questions I have in regards to increasing accuracy of object detection. As I mentioned before, I will have to engage in some kind of frame shifting and camera focal length accountability. Possibly, I might need to remove fish-eye distortion from the camera.

Moreso, I am left with questions about factors that influence tracking accuracy. For instance, how does object size affect accuracy? Would opting for a green soccer ball provide more accuracy than a green tennis ball? I am assuming that best practice for my senior project would be to print as large as a QR code as can fit onto the end of a barbell. But it remains to be seen if that actually helps or is neutral towards the goal of the project. The idea here is that hope-

fully increasing the size provides marginal benefits to object detection speed. On a related note, more research needs to be done to determine why exactly video-based velocity trackers offer such poor accuracy. Is the speed at which the object can be detected and its positions saved more influential than the accuracy of those positions? If its the former, I wonder how removing the need to contrail draw might speed object detection since the loop will run faster. Yet, will we see increased computational time to verify the inner codification of the QR code? Not to mention the fact that the contrail eventually will need to be drawn to show bar path. One possibility might be using the tracked positions to draw the contrail after the fact. With that in mind, it might be interesting to compare the differences between tracking a colored contour versus a QR code contour.

Ultimately, the tutorial certainly validated the realizability of a video-based barbell velocity tracker. Furthermore, features like bar path drawing and even more generally, contour detection appear to be quite straightforward to implement. What still remains is questions as to how I might increase the accuracy of the tracking and evaluate that accuracy during development.