Tutorial: Using CamShift to Track Objects in Video

by Adrian Rosebrock

A few days ago I got angry.

And I mean, really, *really* angry.

The type of angry you get when your code won't compile, your project is past due, and your client is emailing you, calling you, texting you, and even trying to Skype with your mother — simply because they want someone to scream at.

Now, I'm not proud of this moment... But I was so mad that I took the Operating Systems book that was sitting my desk and hurled it like a baseball across the room.

I know. Not cool. Definitely not like me. I mean, what did that poor, innocent Operating Systems book

do anyway?

But a few minutes later, after I had calmed down, I thought to myself — you know, this would make an awesome blog post. What if I had a computer vision system that could track my book as it flew across the room? Or at the very least, track the book as it moved across the frames of a video? (It's safer that way).

With this in mind, I picked my book up off the floor (luckily undamaged), walked back to my desk, and promptly exited out of all my client's work. I was inspired — and as I'm sure you can relate, when you're inspired to write code, there's nothing that can stop you.

As I sipped my coffee and opened up vim, a final thought entered my mind before I felt myself become Neo, sinking irrevocably deep into the The Matrix layer of the programming mindset — "Yep, the CamShift algorithm will do the trick".

Using CamShift to Track Objects in Video

Ready to apply the CamShift algorithm?

Awesome. Open up a new file, name it `track.py`, and let's get started.

```
# import the necessary packages
import numpy as np
import argparse
import cv2

# initialize the current frame of the video, along with the list of
# ROI points along with whether or not this is input mode
frame = None
roiPts = []
inputMode = False
```

First, we'll start by importing the packages that we'll need. We'll use NumPy for numerical processing, `argparse` to parse our command line arguments, and `cv2` to bind with the OpenCV library.

Then, on **Lines 8-10** we define three global variables that we'll be using throughout the rest of this script.

The first is `frame`, which is the current frame of the video that we are processing.

The second is `roiPts`, which is the list of points corresponding to the Region of Interest (ROI) in our video.

Finally, we have `inputMode`, which is simply used as a boolean flag, indicating whether or not we are currently selecting the object we want to track in the video.

Speaking of selecting the ROI in a video, let's go ahead and define a function to do that:

```
def selectROI(event, x, y, flags, param):
    # grab the reference to the current frame, list of ROI
    # points and whether or not it is ROI selection mode
    global frame, roiPts, inputMode

    # if we are in ROI selection mode, the mouse was clicked,
    # and we do not already have four points, then update the
    # list of ROI points with the (x, y) location of the click
    # and draw the circle
    if inputMode and event == cv2.EVENT_LBUTTONDOWN and len(roiPts) < 4:
        roiPts.append((x, y))
        cv2.circle(frame, (x, y), 4, (0, 255, 0), 2)
        cv2.imshow("frame", frame)
```

Here we define `selectROI`, which will be used to select our ROI for tracking.

Line 15 simply grabs reference to our current frame, list of ROI points, and input mode indicator.

Then, on **Line 21**, we make a check for three conditions. In order to select the ROI, these three conditions must hold:

1. We are currently in input mode (which allows us to select the ROI of the object we want to track in the video).
2. he left button of the mouse was clicked, as indicated by the `cv2.EVENT_LBUTTONDOWN` constant, giving us the (x, y coordinates of the click.
3. We must have less than four points currently in our list of ROI points. In this example, we'll assume that our ROI can be representing as a bounding box with four points.

Assuming that these conditions hold, we then append the *(x, y)* location of the click to our list of ROI points on **Line 22**, draw a circle representing the location of the mouse click on **Line 23**, and then display the updated frame on **Line 24**.

As you'll see later, using this `selectROI` function we will be able to select the object that we want to track in our video.

Now, let's examine the `main` function where all the magic happens:

```
def main():
    # construct the argument parse and parse the arguments
    ap = argparse.ArgumentParser()
    ap.add_argument("-v", "--video",
        help = "path to the (optional) video file")
    args = vars(ap.parse_args())

    # grab the reference to the current frame, list of ROI
    # points and whether or not it is ROI selection mode
    global frame, roiPts, inputMode

    # if the video path was not supplied, grab the reference to the
    # camera
    if not args.get("video", False):
        camera = cv2.VideoCapture(0)

    # otherwise, load the video
    else:
        camera = cv2.VideoCapture(args["video"])

    # setup the mouse callback
    cv2.namedWindow("frame")
    cv2.setMouseCallback("frame", selectROI)

    # initialize the termination criteria for cam shift, indicating
    # a maximum of ten iterations or movement by a least one pixel
    # along with the bounding box of the ROI
    termination = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1)
    roiBox = None
```

On **Lines 28-31** we parse our command line arguments. We'll need only a single (optional) switch here, `--video`, which is the path to where our video resides on disk.

I say *optional* because we could work this example in one of two ways. Either we could run the video in real-time and perform the detection, or we could use a video that I have already recorded.

In reality, you'll want to apply these techniques to your own video. But for the sake of this example, let's use the video that I have already recorded.

After we have parsed our command line arguments, we grab our reference to our global variables of the current frame, list of ROI points, and input mode indicator on **Line 35**.

Lines 39-44 handle grabbing the reference to either our webcam or video. If the `--video` switch was not supplied, then we are going to use the webcam that is either builtin or attached to our system (**Lines 39 and 40**). If a path to a video was supplied, then we are going to grab a reference to it on **Line 44**.

As I mentioned earlier, we are going to have to select the bounding box of the object we want to

track in our video. In order to do this, we'll need to register an event callback. We do this on **Line 47 and 48** by creating a window named "frame" and indicating that any mouse events applied to the "frame" window should be handled by our `selectROI` function.

Finally, we initialize a tuple containing our termination criteria for the CamShift algorithm on `Line 53`.

The CamShift algorithm is *iterative*, meaning that it seeks to optimize the tracking criterion. In this case, we'll set the termination criterion to perform two checks.

The first check is the epsilon associated with the centroids of our selected ROI and the tracked ROI according to the CamShift algorithm. If the tracked centroid has not changed by at least one pixel, then terminate the CamShift algorithm.

The second check controls the number of iterations of CamShift. Using more iterations will allow CamShift to (ideally) find a closer centroid match between the selected ROI and the tracked ROI; however, this comes at the cost of runtime. If the iterations are set too high, then we will drop below real-time performance, which is substantially less than ideal in most situations. Let's go ahead and use a maximum of 10 iterations so we don't fall into this scenario.

Now that our termination criteria is setup, we can start examining frames of the video:

```
# keep looping over the frames
while True:
    # grab the current frame
    (grabbed, frame) = camera.read()

    # check to see if we have reached the end of the
    # video
    if not grabbed:
        break

    # if the see if the ROI has been computed
    if roiBox is not None:
        # convert the current frame to the HSV color space
        # and perform mean shift
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        backProj = cv2.calcBackProject([hsv], [0], roiHist, [0, 180], 1)

        # apply cam shift to the back projection, convert the
        # points to a bounding box, and then draw them
        (r, roiBox) = cv2.CamShift(backProj, roiBox, termination)
        pts = np.int0(cv2.cv.BoxPoints(r))
        cv2.polylines(frame, [pts], True, (0, 255, 0), 2)
```

Here is where the fun stuff happens.

On **Line 57** we start looping over the frames of the video, grabbing the current frame on **Line 59**.

The call to `camera.read()` returns a tuple with two values. The first, `grabbed`, indicates whether or not reading the frame was a success. The second is the `frame` itself.

On **Line 63** we make a check to see if the frame was successfully grabbed. If it was not, then we make the assumption that we are at the end of the video and we break from the loop.

Line 67 then checks to see if we have selected a bounding box for our object to track yet.

Provided that we have selected a bounding box, the first thing we'll do is convert our image from the RGB color space to the HSV color space (**Line 73**) and compute the histogram back projection on **Line 74**, using only the Hue component of the color space.

Once we have the back projection, we apply the CamShift algorithm on **Line 78** by making a call to `cv2.CamShift`. This function expects three arguments:

1. **`backProj`:** Which is the output of the histogram back projection.
2. **`roiBox`:** The estimated bounding box containing the object that we want to track.
3. **`termination`:** Our termination criterion which we defined on **Line 56**.

The `cv2.CamShift` function then returns two values to us. The first contains the estimated position, size, and orientation of the object we want to track. We then take this estimation and draw a rotated bounding box on **Lines 79 and 80**.

Finally, the second output of the `cv2.CamShift` function is the newly estimated position of the ROI, which will be re-fed into subsequent calls into the `cv2.CamShift` function.

Now, let's see how we can set our input ROI and create the reference histogram used by the back projection.

```
# show the frame and record if the user presses a key
cv2.imshow("frame", frame)
key = cv2.waitKey(1) & 0xFF

# handle if the 'i' key is pressed, then go into ROI
# selection mode
if key == ord("i") and len(roiPts) < 4:
    # indicate that we are in input mode and clone the
    # frame
    inputMode = True
    orig = frame.copy()

    # keep looping until 4 reference ROI points have
    # been selected; press any key to exit ROI selction
    # mode once 4 points have been selected
    while len(roiPts) < 4:
        cv2.imshow("frame", frame)
        cv2.waitKey(0)

    # determine the top-left and bottom-right points
    roiPts = np.array(roiPts)
    s = roiPts.sum(axis = 1)
    tl = roiPts[np.argmin(s)]
    br = roiPts[np.argmax(s)]

    # grab the ROI for the bounding box and convert it
    # to the HSV color space
    roi = orig[tl[1]:br[1], tl[0]:br[0]]
    roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
    #roi = cv2.cvtColor(roi, cv2.COLOR_BGR2LAB)

    # compute a HSV histogram for the ROI and store the
    # bounding box
    roiHist = cv2.calcHist([roi], [0], None, [16], [0, 180])
    roiHist = cv2.normalize(roiHist, roiHist, 0, 255, cv2.NORM_MINMAX)
    roiBox = (tl[0], tl[1], br[0], br[1])

# if the 'q' key is pressed, stop the loop
elif key == ord("q"):
    break
```

Line 80 handles displaying the current frame on screen, while **Line 81** handles if a key is pressed.

If the '`i`' key was pressed and there are less than four points in our list of points corresponding to the ROI, then we drop down into input mode (**Line 85**).

From there, we set `inputMode = True` to indicate that we are selecting our ROI. We also make a clone of the original frame.

Lines 95-97 freezes the frame on display, waiting for the bounding box of 4 points to be selected.

After we have our bounding box, we convert our ROI points to a NumPy array, and grab the top-left and bottom-right points, respectively (**Lines 99-102**).

Given the top-left and bottom-right points, we then grab the ROI from the original frame and convert it from the RGB to the HSV color space on **Lines 107 and 108**.

In order to apply the `cv2.CamShift` function, we need the output of the back projection. And the only way to obtain the back projection is to create a reference histogram of the object we want to track.

We construct this histogram on **Lines 111 and 112** by making a call to `cv2.calcHist` and the normalizing it. Again, we are only using the Hue component of the HSV color space.

It is important to note the 4th parameter of `cv2.calcHist` — the number of bins. In this case, we are using only 16 bins in the histogram. In OpenCV, hue values can fall within the range *[0, 180]*, so tuning the number of bins for your application will certainly be important.

Finally, we construct the ROI bounding box on **Line 113** using the top-left and bottom-right points, respectively.

In order to break out of our frame loop, we'll also monitor if the '`q`' key is pressed on **Line 116**. If the '`q`' key is pressed, we'll break out of the loop.

```
# cleanup the camera and close any open windows
camera.release()
cv2.destroyAllWindows()

if __name__ == "__main__":
    main()
```

The rest of our code is just cleanup. We release the reference to our camera (whether webcam or video) and close all open windows on **Lines 120 and 121**.

Lines 123 and 124 simply execute our script.

Results

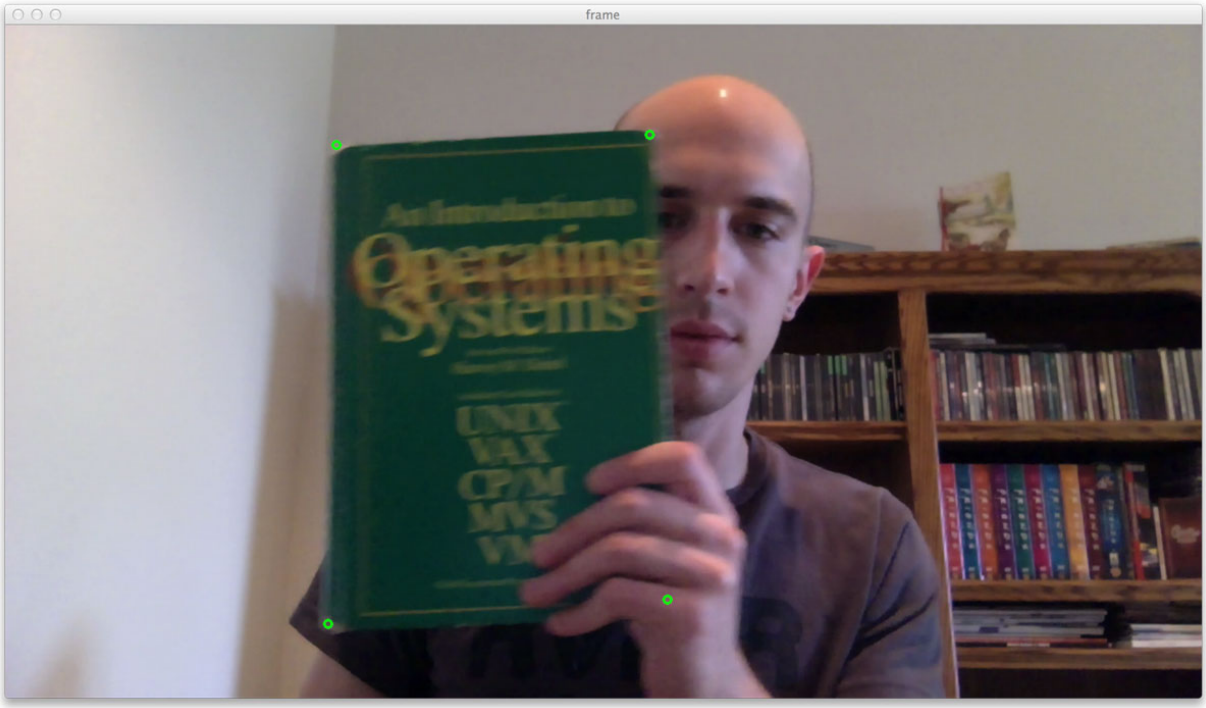
To execute our CamShift example, open up a terminal and issue the following command:

```
$ python track.py --video video/sample.mov
```

Of course, if you wanted to use your own webcam, simply omit the `--video` switch:

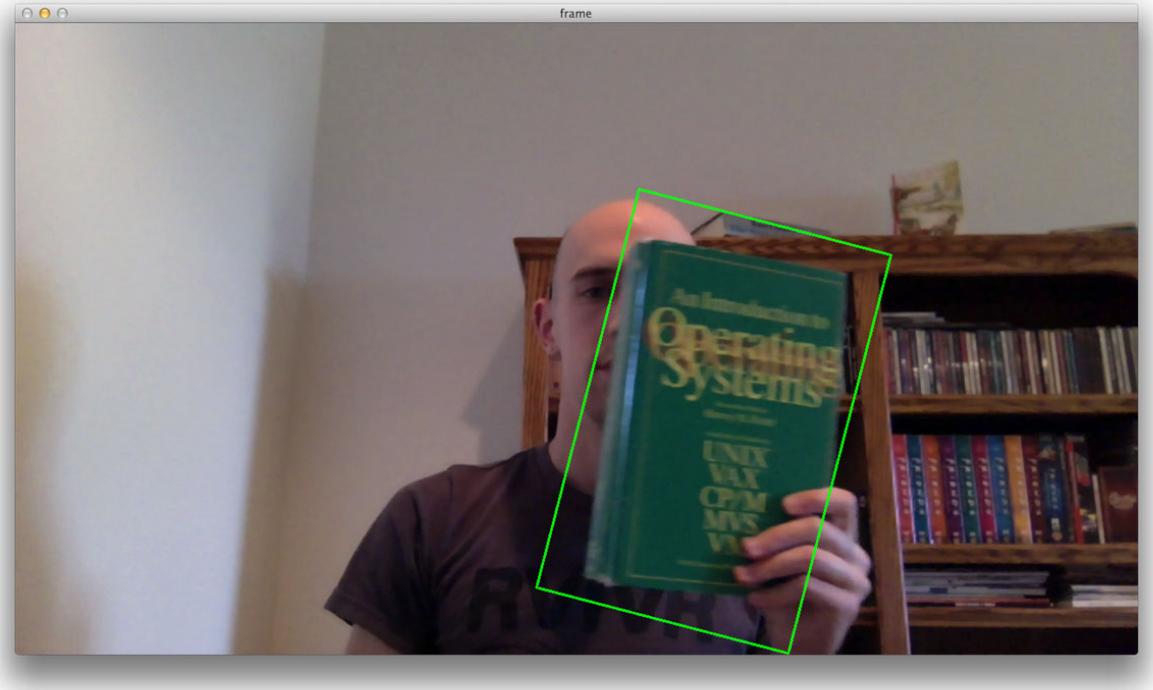
```
$ python track.py
```

Once your video is loaded, press the `i` key and select four points surround the object that you want to track:

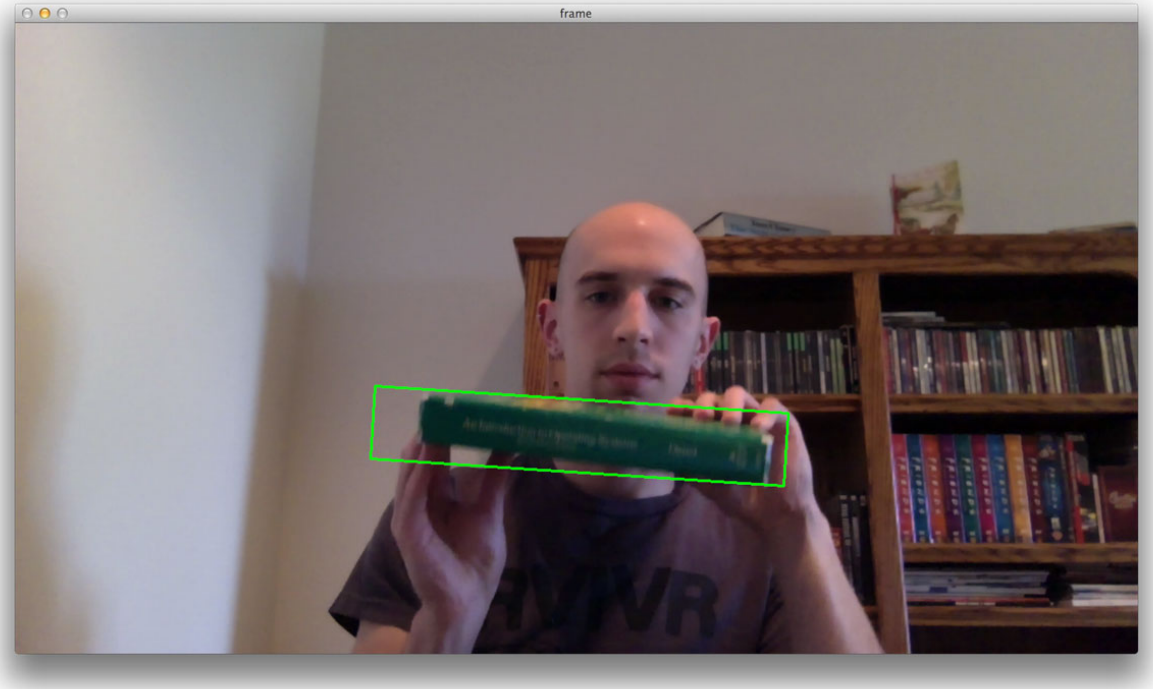


After your four points are selected, press any key to exit input mode.

Now that our script knows what to track and the reference histogram is computed, you should see the object being tracked across the screen in subsequent frames:



Since both the cover of the book and the spine have similar color distributions, you can also track the spine of the book as well.



Limitations

There are some limitations to our approach.

The first is that we are only using the Hue component of the HSV color space. This means that unless the object you are trying to track is not only a single shade, then your results will likely be sub-optimal.

To remedy this, you can simply extend the code to compute a 2D histogram using both the Hue and Saturation components.

As of the writing this blog post, OpenCV currently ***DOES NOT*** support 3D histograms in the back projection calculation and CamShift tracking. Unfortunate, but true.

The second limitation is tuning the number of bins in the color histogram. This will depend on many aspects, including your application along with lighting conditions. You will need to tune this parameter for your own applications.

Finally, if you are looking for a more robust tracking solution that can take into account texture and localized features, then you should look into keypoint detection, local invariant descriptors (ex. DoG and SIFT), and computing homographies between two sets of keypoints and their corresponding features.

Summary

In this blog post I have showed you how to track objects in video using the CamShift.

We implemented our code using the Python programming language, the OpenCV library, and the `cv2.CamShift` function.

While simple, the CamShift approach is quite powerful for object tracking, especially when you consider we are only utilizing color histograms.

However, if you are looking for a more robust approach, using keypoint detection, local invariant descriptors, and computing homographies to match the keypoints will likely obtain better results.

All that said, CamShift is still an excellent starting point for object tracking in video.

Downloads

To download the code to this blog post, along with accompanying example video, just click here.

About Adrian

Hey, my name is Adrian Rosebrock. I'm an entrepreneur and Ph.D who blogs over at PyImageSearch.com. If you liked this blog post, and want to learn more about me, head on over to my blog. And if you want to learn computer vision in a ***single weekend***, then definitely take a look at my book, Practical Python and OpenCV + Case Studies.

This post originally appeared as a guest post on Computer Vision Online.