Swarthmore College
E27: Computer Vision

Final Project: Attempted Implementation of an Extension of AprilTag Tracking

May 5, 2023

Quentin Adolphe
Cole Smith

Background:

For his engineering design project, Cole performed biomechanics research investigating the movement of the thumb-tip throughout 3D space in a computer model. With future work on his project looking to validate this in real life, part of the analysis will involve recording the movement of a thumb and determining joint angles using motion capture techniques. As we learned about motion tracking in the latter half of the course, that made us wonder how computer vision techniques could be leveraged to attempt to streamline the analysis process. Using stereo depth and triangulation, we wondered if we might be able to develop a method to record the video of a limb moving in 3D space and run it through an algorithm to detect points of interest and determine the angle between them over time. With this framework in mind, the goal of our final project was to use triangulation to measure joint angles over time for videos shot on synchronized cameras. We hoped to start with larger angles, such as the elbow, to validate that our code runs successfully and can be applied to any movement with 3 points to track.

To carry out our analysis we specifically sought to easily track 3 distinct points in 3D space. Having first wanted to use template tracking with different colored points, we quickly shifted to AprilTag tracking from the advice of Professor Phillips. Usually seen in applications in robotics, AprilTags, as shown in Figure 1, are unique visual tags that aid in several computer vision properties such as object tracking, creating a visual reference system for robots to know where they are in the field, camera calibration, and more. Sensors and cameras can identify AprilTags via the use of an AprilTag detection software that is importable in Python that easily calculates the position, orientation, and identity of the tag relative to the camera position. The simple, unique structure of these allows the user to detect and process them very quickly.

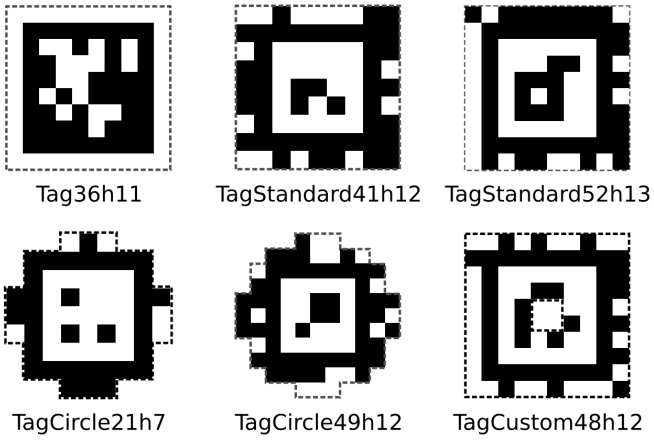

Figure 1: Example AprilTags

Understanding the framework of AprilTags, we wanted to see if we could use them as our 3D points to track. We hoped using an algorithm to follow the AprilTags would help us streamline the process of finding the points to track instead of performing template matching. With that in mind, we set out to create our workflow, calibrating our cameras before recording elbow flexion activities with AprilTags attached and calculating the change in angle over time with the hope of comparing results to a known motion tracking program if we obtained strong results. The following sections follow the process of the development of our program, the results we obtained, and how we could have improved it had we had more time.

Theory:

Object Tracking:

Object tracking is a lucrative but difficult field in computer science. For this project, it was best to implement an existing algorithm rather than starting from scratch or using a less versatile method of object detection such as template matching. Opting to use OpenCV's KCF

tracker to follow objects of interest in real-time came with a few key advantages. KCF tracking is both a very fast and highly accurate tracking method, while only struggling when the object it is following becomes obstructed [1]. The videos used for this project had clear and unobstructed views of the objects of interest, so this fast and accurate algorithm was the ideal choice.

Camera calibration:

Camera calibration is a method that allows us to account for and correct any distortion caused by the cameras by determining the intrinsic and extrinsic properties of the camera and using them to correct any distortion. Pinhole cameras can distort images when they take a picture. This distortion can cause straight lines to appear curved (radial distortion), and the phenomenon worsens the further from the center of the image the line is. Another type of distortion causes some areas in an image to appear closer than expected (tangential distortion) as a result of the image-taking lens not being parallel to the plane of the image being taken [2]. To perform camera calibration you need to extract parameters known as distortion coefficients, as well as the intrinsic and extrinsic which are specific to each camera and include information including the focal length, optical center, and rotational and translational matrices. For our application of triangulation and stereo depth, we are seeking to extract these values and correct the distortion that occurs. Using an image of a chessboard-like grid, we can perform the calibration using the images with OpenCV's calibrateCamera() function [2]. This gives us the parameters we are looking to extract from 3D object points and their associated 2D image points for each frame in a video. These points are associated with the corners of the boxes on the chessboard and allow for easy determination of the 3D and 2D points using a corner detection algorithm. Once we have the distortion parameters and intrinsic values for the cameras, and

extrinsic values of the stereo setup, we can use them in further analysis such as triangulation or removing the distortion from the image.

Triangulation:

Triangulation is a critical component of scene reconstruction as you look to use the projections of a shared point from 2 images to determine the location of that point in 3D space. Using two calibrated cameras in the same scene, the camera matrices (both intrinsic and extrinsic), and the corresponding points, several methods can be used to compute the 3D coordinates of the corresponding point. Common practice involves using an error-minimizing ordinary least squares method to solve for the shared points. Depending on the method employed, this error that is being minimized could be algebraic or geometric. For this project, we used a direct linear transform (DLT) function [3]. This function takes in the 2D points as a single $2 \times n$ vector and associated projection matrices as another input vector to output a $3 \times n$ vector of the computed 3D points using a DLT. This solves the matrix equation $Ax = w$ for unknowns x such that w is minimized with a least squares calculation [4]. For our project, the DLT function outputs the vector x as the calculated 3D points.

Methods and Approach:

Originally, the project was intended to have four main components: (1) recording a video of a limb from two perspectives with two phones, (2) implementing template matching to track three points with one camera, (3) using stereo calibration and triangulation to determine the location of each point in the 3D world, and finally (4) plotting the points in a video format with accompanying joint angles over time. Though this plan was mainly followed, KCF object tracking using OpenCV's legacy algorithms ended up being far more effective at locating objects

of interest than template matching. Template matching has its uses, but it is far too limited and inaccurate for the real-time tracking required for this project.

A critical step in the process of finding 3D locations of objects is to first find their 2D locations in the camera planes. Though AprilTags are extremely versatile and an excellent way of locating objects, their full functionality was not used in this project. Instead, they were used simply as high contrast and easily identifiable cues for OpenCV's KCF tracking algorithm to lock onto. Opting for a tracking algorithm instead of the AprilTags' built-in functionality allows this project to be used for a wider range of objects. Thus, time was sacrificed in implementing a separate algorithm in return for more versatility in object tracking.

The next step was to calibrate the stereo setup. First, both cameras must be calibrated to find their intrinsic parameters. The cameras were set in their stereo positions and began recording. A distinctive clap was performed in front of both cameras to ensure that the videos were synchronized. Then, a chessboard-like grid was waved in front of their view as they recorded. The algorithm implemented uses OpenCV functions to find the locations of the chessboard in each frame for the videos. These chessboards allow both the intrinsic properties of the cameras and the extrinsic properties of the stereo environment to be derived. Finally, after collecting all the 2D points from the objects of interest and the parameters for both of the cameras, the 3D triangulated points can be derived with a DLT.

Results and Discussion:

The implementation of OpenCV's KCF tracking algorithm was successful. Simply highlighting the regions of interest on both videos when prompted allows for nearly any object in a scene to be tracked. As seen in Figure 2, three AprilTags to serve as tracking points for the

KCF algorithm are placed on an arm. Selecting the same regions of interest from both the left and right cameras allowed for 2D location data to be collected for each frame of the videos.

Figure 2 (a): Three points of interest in red being tracked by left camera

Figure 2 (a): Three points of interest in red being tracked by right camera

After successfully implementing the tracking algorithm, the cameras and stereo setup had to be calibrated. The intrinsic matrices computed for the left camera (iPhone 13 Pro) and right camera (iPhone 13 mini) along with extrinsic parameters for the stereo setup can be seen below.

$$Intrinsic\ matrix\ for\ left\ camera\ = \begin{bmatrix} 1661.49, & 0, & 990.275 \\ 0, & 1674.68, & 483.835 \\ 0, & 0, & 1 \end{bmatrix}$$

$$Intrinsic\ matrix\ for\ right\ camera\ = \begin{bmatrix} 1635.05, & 0, & 1010.56 \\ 0, & 1656.91, & 422.097 \\ 0, & 0, & 1 \end{bmatrix}$$

$$Extrinsic\ rotation\ matrix\colon\ R\ = \begin{bmatrix} .224832, & -0.0170742, & 0.974396 \\ 0.0167305, & 0.999858, & -0.0210836 \\ -0.974254, & 0.0167761, & 0.224829 \end{bmatrix}$$

$$Extrinsic\ translation\ vector\colon\ T\ = \begin{bmatrix} -62.7286 \\ 1.74351 \\ 46.1935 \end{bmatrix}$$

Both cameras recorded a 1080 pixel * 1920 pixel video (9:16 aspect ratio). This means the expected offsets u_0 and v_0 for both cameras would be:

$$u_0 = 1920 / 2 = 960$$

$$v_0 = 1080 / 2 = 540$$

This yields a percent difference of 3.10% between expected and calculated values for u_0 in the left camera and a 5.13% difference in the right camera. The calibration also yielded a percent difference of 11.0% between expected and calculated values for v_0 in the left camera, and a 13.6% difference in the right camera. Though the calculated values for u_0 had a relatively small percent difference, the large errors in the v_0 values were a cause for concern. It is difficult to calculate focal lengths for modern cell phones, but the relative consistency between the f_x and f_y values for the two phones was confidence-inspiring. There was only a 1.60% difference between the calculated f_x values and a 1.06% difference between the calculated f_y values. The calibration of each camera was mostly successful, but the large differences previously discussed may be the cause of inaccuracies in computation later in the project. These inaccuracies could have come from the cameras being bumped or slightly moved, or not perfectly synced.

Unfortunately, the 3D triangulation proved to be only marginally successful. As seen in Figure 3, the real-world points (Figure 3a) correspond reasonably well with the points calculated with the triangulation algorithm (Figure 3b). The points of interest are in a straight line in the real world, and come out in a straight line when triangulated.

Figure 3 (a): Three points of interest in red being tracked

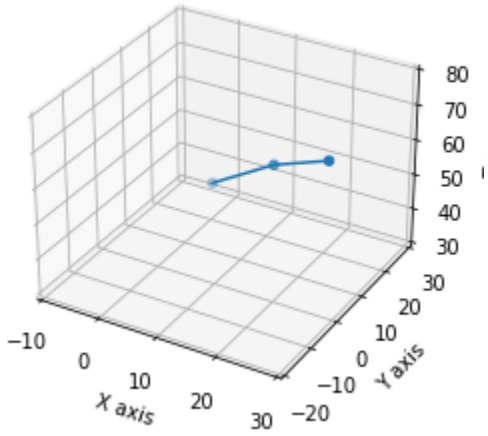

Figure 3 (b): The same three points of interest triangulated

In Figure 4, the real-world points (Figure 4a) do not correspond well with the points calculated with the triangulation algorithm (Figure 4b). The points of interest are at a 90º angle in the real world, but come out only at a slight angle when triangulated. Comparing Figure 3b with Figure 4b, the relative angle between the three points is very similar, but the points are closer

together in Figure 4b. The algorithm does recognize the changing distances between the three points, but it does not do so with high accuracy.

Figure 4 (a): Three points of interest in red being tracked

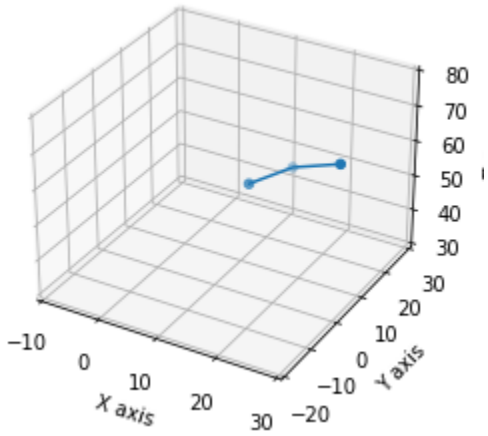

Figure 4 (b): The same three points of interest triangulated unsuccessfully

In an attempt to debug the code, the experiment was repeated with a different dataset and different stereo setup, but inaccuracies still arise as seen in Figure 5. With this dataset, two fixed

points were chosen while the third tracked the moving face. The triangulated outcome had the general layout of the three points correct, but their specific positions were still inaccurate. In Figure 5 (a), there was one point being tracked in front of two fixed points in the background. But as seen in Figure 5 (b), the triangulated point of the face was put behind the two fixed points.

Figure 5 (a): Three points of interest in red being tracked

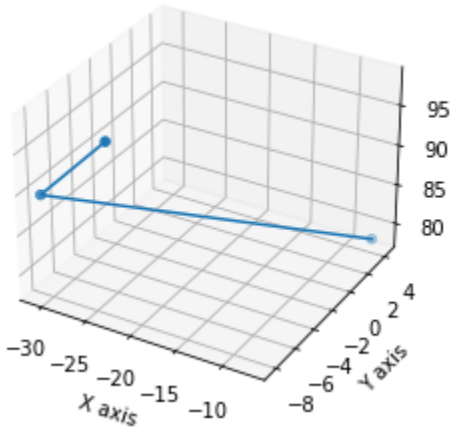

Figure 5 (b): The same three points of interest triangulated unsuccessfully

The algorithm successfully calculated the relative position between the two fixed points but unsuccessfully calculated the position of the face in the foreground. The cause of the incorrect mapping was unclear, but most likely arose from poor camera or stereo calibration. The differences between expected and computed intrinsic parameters were sufficiently large to cause errors in 3D triangulation.

Conclusion:

Although the triangulation of points did not come out as expected, there were many successes throughout this project. It is difficult to ascertain the reasoning behind the failed triangulation, but it can probably be solved through more experimentation with different scenes and stereo setups. Though this 3D location algorithm is not suitable for real-world applications as it stands, with some polishing it may be a useful tool. From utilizing cameras on cars to analyze the world around them, to a simple cell phone application that uses multiple cameras to detect depth in an environment, stereo vision has many useful applications for everyday life. Future work could be done on this model for live-action triangulation and tracking, and making improvements on the triangulation of the AprilTags to achieve its desired goal in real-time. With regard to the original purpose of streamlining joint angle calculations, further, correct implementation of this algorithm could potentially allow researchers to obtain results for motion tracking using AprilTags in real-time, and reduce the need to rely on purchasing expensive cameras or software for analysis of motion capture data. This could make many studies repeatable at a lower cost to the researcher.

References

[1] “Object Tracking using OpenCV (C++/Python) |,” Feb. 13, 2017. https://learnopencv.com/object-tracking-using-opencv-cpp-python/

[2] OpenCV, “OpenCV: Camera Calibration,” docs.opencv.org. https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html

[3] T. Batpurev, “Stereo Camera Calibration and Triangulation with OpenCV and Python,” Temuge’s webpage, Feb. 02, 2021. https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and-triangulation.html (accessed May 06, 2023).

[4] “LOST in Triangulation,” GTSAM, Feb. 04, 2023. https://gtsam.org/2023/02/04/lost-triangulation.html#:~:text=A%20commonly%20used%20method%20for (accessed May 06, 2023).

Appendices

Project Folder:

https://drive.google.com/drive/folders/1DxS4_k7FUn2e3M8ocawICIJVgklwRztY?usp=sharing

Final video of arm scene:

https://drive.google.com/file/d/1jnC2WqPSlvvJOwdXcxDVNSEYBTsiPwS4/view?usp=share_li

nk

Final video of room scene:

https://drive.google.com/file/d/1jhRVoLh8s1WOcW9IvLMCUVJbYO9n7KiT/view?usp=share_l

ink

```
"""
Created on Wed Apr 26 15:01:20 2023

@author: quentinadolphe

Description: User must have a total of 4 videos to effectively use the
script below. 2 videos for calibration and 2 videos for
tracking/triangulation. Calibration information for videos I have used is
hardcoded in. If calibration wants to be done, delete mtx1, mtx2, R and T,
and uncomment the ster_calibrate() function call. It is best to run this in
some IDE like Spyder, so the graphs can be outputted to the console.
Replace current video names with desired videos and code should run fine.

Code taken from other sources cited below with links.
"""

import numpy as np
import cv2
import matplotlib.pyplot as plt
from scipy import linalg
from mpl_toolkits.mplot3d import Axes3D


###
https://www.instructables.com/Object-Tracking-With-Opencv-and-Python-With-J
ust-5/
```

```
def get_points(video):
    cap=cv2.VideoCapture(video)
    vid = []


    success, img = cap.read()
    bboxes = []
    tracked = 3
    points = [[] for i in range(tracked)]

    for i in range(tracked):
        bboxes.append(cv2.selectROI("Multi-tracker {}".format(i+1)
,img,False))

    ###
https://learnopencv.com/multitracker-multiple-object-tracking-using-opencv-
c-python/
    # Create MultiTracker object
    multiTracker = cv2.legacy.MultiTracker_create()

    # Initialize MultiTracker
    for bbox in bboxes:
      multiTracker.add(cv2.legacy.TrackerKCF_create(), img, bbox)


    while cap.isOpened():
      success, frame = cap.read()
      if not success:
        break

      # get updated location of objects in subsequent frames
      success, boxes = multiTracker.update(frame)

      # draw tracked objects
      for i, newbox in enumerate(boxes):
        p1 = (int(newbox[0]), int(newbox[1]))
        p2 = (int(newbox[0] + newbox[2]), int(newbox[1] + newbox[3]))
        cv2.rectangle(frame, p1, p2, (225,225,0), 2, 1)

        #draw centroid
        pc = (int((p1[0] + p2[0])/2),  int((p1[1] + p2[1])/2))
        points[i % tracked].append(pc)
        frame = cv2.circle(frame, pc , 3, (0,0,255), 3)
```

```
      # show frame
      cv2.imshow('MultiTracker', frame)

      # quit on ESC button
      if cv2.waitKey(1) & 0xFF == 27:  # Esc pressed
        break

    cap.release()
    cv2.destroyAllWindows()
    return [np.array(points[0]), np.array(points[1]), np.array(points[2])]


def cam_calibrate(vid1, vid2):
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30,
0.001)

    # prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
    objp = np.zeros((6*7,3), np.float32)
    objp[:,:2] = np.mgrid[0:7,0:6].T.reshape(-1,2)

    # Arrays to store object points and image points from all the images.
    objpoints = [] # 3d point in real world space
    imgpoints_left = [] # 2d points in image plane.
    imgpoints_right = []

    cap = cv2.VideoCapture(vid1)
    dap = cv2.VideoCapture(vid2)

    found = 0
    while(found < 240):  # Here, 10 can be changed to whatever number you
like to choose
        ret, img1 = cap.read() # Capture frame-by-frame

        ret, img2 = dap.read() # Capture frame-by-frame


        gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
        gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

        # Find the chess board corners
        ret1, corners1 = cv2.findChessboardCorners(gray1, (7,6),None)
        ret2, corners2 = cv2.findChessboardCorners(gray2, (7,6),None)
```

```
        # If found, add object points, image points (after refining them)
        if ret1 == True and ret2 == True:
            corners1 = cv2.cornerSubPix(gray1, corners1, (11, 11), (-1,
-1), criteria)
            corners2 = cv2.cornerSubPix(gray2, corners2, (11, 11), (-1,
-1), criteria)

            cv2.drawChessboardCorners(img1, (5,8), corners1, ret1)
            cv2.drawChessboardCorners(img2, (5,8), corners2, ret2)


            objpoints.append(objp)
            imgpoints_left.append(corners1)
            imgpoints_right.append(corners2)
            found += 1

        cv2.imshow('img', img1)
        cv2.imshow('img2', img2)
        k = cv2.waitKey(500)

    # When everything done, release the capture
    h, w, _ = img1.shape
    print(h)
    print(w)
    cap.release()
    cv2.destroyAllWindows()

    ret, mtx1, dist1, rvecs, tvecs = cv2.calibrateCamera(objpoints,
imgpoints_left, gray1.shape[::-1], None, None)
    ret, mtx2, dist2, rvecs, tvecs = cv2.calibrateCamera(objpoints,
imgpoints_right, gray2.shape[::-1], None, None)

    return mtx1, dist1, mtx2, dist2, imgpoints_left, imgpoints_right,
objpoints, h, w

def ster_calibrate(vid1, vid2):
    mtx1, dist1, mtx2, dist2, imgpoints_left, imgpoints_right, objpoints,
height, width = cam_calibrate(vid1, vid2)

    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100,
0.001)
```

```
    stereocalibration_flags = cv2.CALIB_FIX_INTRINSIC
    ret, CM1, dist11, CM2, dist22, R, T, E, F =
cv2.stereoCalibrate(objpoints, imgpoints_left, imgpoints_right, mtx1,
dist1,
                                                                    mtx2,
dist2, (width, height), criteria = criteria, flags =
stereocalibration_flags)

    print(ret)
    return R, T, mtx1, dist1, mtx2, dist2

###
https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibratio
n-and-triangulation.html
def DLT(P1, P2, point1, point2):

    A = [point1[1]*P1[2,:] - P1[1,:],
         P1[0,:] - point1[0]*P1[2,:],
         point2[1]*P2[2,:] - P2[1,:],
         P2[0,:] - point2[0]*P2[2,:]
        ]
    A = np.array(A).reshape((4,4))


    B = A.transpose() @ A
    U, s, Vh = linalg.svd(B, full_matrices = False)

    print('Triangulated point: ')
    print(Vh[3,0:3]/Vh[3,3])
    return Vh[3,0:3]/Vh[3,3]

#%%
'''
### camera and stereo calibration
R, T, mtx1, dist1, mtx2, dist2 = ster_calibrate("leftcalib.mp4",
"rightcalib.mp4")

### calibration is complete so this no longer has to be run'''

#%%

mtx1 = np.array([[1661.49, 0, 990.275],
```

```
                [0, 1674.68, 483.835],
                [0,0,1]])

mtx2 = np.array([[1635.05, 0, 1010.56],
                [0, 1656.91, 422.097],
                [0,0,1]])

R = np.array([[.224832, -0.0170742, 0.974396],
              [0.0167305, 0.999858, -0.0210836],
              [-0.974254, 0.0167761, 0.224829]])

T = np.array([[-62.7286],
              [1.74351],
              [46.1935]])


#%%
#### triangulation

#RT matrix for C1 is identity.
RT1 = np.concatenate([np.eye(3), [[0],[0],[0]]], axis = -1)
P1 = mtx1 @ RT1 #projection matrix for C1

#RT matrix for C2 is the R and T obtained from stereo calibration.
RT2 = np.concatenate([R, T], axis = -1)
P2 = mtx2 @ RT2 #projection matrix for C2

#%%
### Get points from stereo pair
points1 = get_points("lefttags.mp4")
#%%
points2 = get_points("righttags.mp4")

#%%



#i = 480 # delete for loop below and set i to any value to see desired
moment in time

for i in range(0,800,3):

    p3ds = []
```

```
for j in range(len(points1)): #tags
    _p3d = DLT(P1, P2, points1[j][i], points2[j][i])
    p3ds.append(_p3d)
p3ds = np.array(p3ds)
p3ds = np.transpose(p3ds)



fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlim3d(-10, 30)
ax.set_ylim3d(-20, 30)
ax.set_zlim3d(30, 80)
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')


ax.plot(p3ds[0], p3ds[1], p3ds[2])
ax.scatter3D(p3ds[0], p3ds[1], p3ds[2], 'red')
plt.show()
```