Video Augmented Reality

Just as we implement augmented reality on image we can do the same with video using find and warp perspective concept but with some tweaks.For this one we first define a our find and warp function

1) Script:import packages to be used and set initial value for ref_pts which is the location of markers in previous frame,which serves as caching incase of unforeseen condition like lighting conditions, viewpoint, or motion blur in the current where we cannot detect the aruco markers

```
# import the necessary packages
import numpy as np
import cv2
# initialize our cached reference points
CACHED_REF_PTS = None
```

2) Script:So we define our function to take in frame which is the video we will be capturing from webcam.,source,cornerIDs,arucodict, arucoparams and useCache which is by default false.so we grab the cached reference point in the function since we won't be using the function in this file.We then get the spatial dimensions of both the frame and the source.

```
def find_and_warp(frame, source, cornerIDs, arucoDict, arucoParams,
    useCache=False):
    # grab a reference to our cached reference points
    global CACHED_REF_PTS

    # grab the width and height of the frame and source image,
    # respectively
    (imgH, imgW) = frame.shape[:2]
    (srcH, srcW) = source.shape[:2]
```

3) Script:With the Aruco dictionary and params passed we detect the aruco makers in the frame.if the four aruco markers are detected then flatten it to a 1-D array,otherwise we set ids to an empty array.Also initializing our refpts

```
    # detect AruCo markers in the input frame
    (corners, ids, rejected) = cv2.aruco.detectMarkers(
        frame, arucoDict, parameters=arucoParams)

    # if we *did not* find our four ArUco markers, initialize an
    # empty IDs list, otherwise flatten the ID list
    ids = np.array([]) if len(corners) != 4 else ids.flatten()
```

```
    # initialize our list of reference points
    refPts = []
```

4) Script:Loop through our cornerIDs of makers in the order top-left,top-right,bottom-right,bottom-left.We grab the corner maker ID index,if it does not exist the we continue looping otherwise we append the coordinate of the corner with in refPts.

```
    # loop over the IDs of the ArUco markers in top-left, top-right,
    # bottom-right, and bottom-left order
    for i in cornerIDs:
        # grab the index of the corner with the current ID
        j = np.squeeze(np.where(ids == i))

        # if we receive an empty list instead of an integer index,
        # then we could not find the marker with the current ID
        if j.size == 0:
            continue

        # otherwise, append the corner (x, y)-coordinates to our
list
        # of reference points
        corner = np.squeeze(corners[j])
        refPts.append(corner)
```

5) Script: if the four aruco marker are not obtained,then we are left with either using caching or returning to exit the program

```
    # check to see if we failed to find the four ArUco markers
    if len(refPts) != 4:
        # if we are allowed to use cached reference points, fall
        # back on them
        if useCache and CACHED_REF_PTS is not None:
            refPts = CACHED_REF_PTS

        # otherwise, we cannot use the cache and/or there are no
        # previous cached reference points, so return early
        else:
            return None

    # if we are allowed to use cached reference points, then update
```

```
    # the cache with the current set
    if useCache:
        CACHED_REF_PTS = refPts
```

6) Script:Getting to this stage in this defined function means have the 4 corner by detecting or caching,we then do the main perspective warp by first finding the homography matrix using the dstMat and srcMat obtain from refPts and source spatial dimension respectively.

```
    # unpack our ArUco reference points and use the reference points
    # to define the *destination* transform matrix, making sure the
    # points are specified in top-left, top-right, bottom-right, and
    # bottom-left order
    (refPtTL, refPtTR, refPtBR, refPtBL) = refPts
    dstMat = [refPtTL[0], refPtTR[1], refPtBR[2], refPtBL[3]]
    dstMat = np.array(dstMat)

    # define the transform matrix for the *source* image in top-
left,
    # top-right, bottom-right, and bottom-left order
    srcMat = np.array([[0, 0], [srcW, 0], [srcW, srcH], [0, srcH]])

    # compute the homography matrix and then warp the source image
to
    # the destination based on the homography
    (H, _) = cv2.findHomography(srcMat, dstMat)
    warped = cv2.warpPerspective(source, H, (imgW, imgH))
```

7) Script:Here we create a mask for the source image and make it into 3 channels.

```
    # construct a mask for the source image now that the perspective
    # warp has taken place (we'll need this mask to copy the source
    # image into the destination)
    mask = np.zeros((imgH, imgW), dtype="uint8")
    cv2.fillConvexPoly(mask, dstMat.astype("int32"), (255, 255,
255),
        cv2.LINE_AA)

    # this step is optional, but to give the source image a black
    # border surrounding it when applied to the source image, you
    # can apply a dilation operation
```

```
    rect = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
    mask = cv2.dilate(mask, rect, iterations=2)

    # create a three channel version of the mask by stacking it
    # depth-wise, such that we can copy the warped source image
    # into the input image
    maskScaled = mask.copy() / 255.0
    maskScaled = np.dstack([maskScaled] * 3)
```

8) Script:We place the warped Image unto the frame.

```
# copy the warped source image into the input image by
    # (1) multiplying the warped image and masked together,
    # (2) then multiplying the original input image with the
    # mask (giving more weight to the input where there
    # *ARE NOT* masked pixels), and (3) adding the resulting
    # multiplications together
    warpedMultiplied = cv2.multiply(warped.astype("float"),
        maskScaled)
    imageMultiplied = cv2.multiply(frame.astype(float),
        1.0 - maskScaled)
    output = cv2.add(warpedMultiplied, imageMultiplied)
    output = output.astype("uint8")

    # return the output frame to the calling function
    return output
```

In the below we use the find_and_warp function

1) Script: Import all the necessary packages to be used

```
# import the necessary packages
from pyimagesearch.augmented_reality import find_and_warp
from imutils.video import VideoStream
from collections import deque
import argparse
import imutils
import time
import cv2
```

2) Script: Using python opencv_ar_video.py --input videos/jp_trailer_short.mp4 to run the program we need to construct an augment parser.

```
# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--input", type=str, required=True,
    help="path to input video file for augmented reality")
ap.add_argument("-c", "--cache", type=int, default=-1,
    help="whether or not to use reference points cache")
args = vars(ap.parse_args())
```

3) Script:we set the Aruco dictionary and parameter used.We read the video input using cv2.VideoCapture() which gives us file pointers. We then read file pointer into a queue Q which helps to improve I/O latency.Start webcam waiting a few second

```
# load the ArUCo dictionary and grab the ArUCo parameters
print("[INFO] initializing marker detector...")
arucoDict = cv2.aruco.Dictionary_get(cv2.aruco.DICT_ARUCO_ORIGINAL)
arucoParams = cv2.aruco.DetectorParameters_create()

# initialize the video file stream
print("[INFO] accessing video stream...")
vf = cv2.VideoCapture(args["input"])

# initialize a queue to maintain the next frame from the video
stream
Q = deque(maxlen=128)

# we need to have a frame in our queue to start our augmented
reality
# pipeline, so read the next frame from our video file source and
add
# it to our queue
(grabbed, source) = vf.read()
Q.appendleft(source)

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

4) Script:we loop through the queue until it has no frame left in the input video.on each frame we apply our find_and_warp function passing the necessary parameters.

```
# loop over the frames from the video stream
while len(Q) > 0:
    # grab the frame from our video stream and resize it
    frame = vs.read()
    frame = imutils.resize(frame, width=600)

    # attempt to find the ArUCo markers in the frame, and provided
    # they are found, take the current source image and warp it onto
    # input frame using our augmented reality technique
    warped = find_and_warp(
        frame, source,
        cornerIDs=(923, 1001, 241, 1007),
        arucoDict=arucoDict,
        arucoParams=arucoParams,
        useCache=args["cache"] > 0)
```

5) Script:If the 4 corner aruco markers are obtained then set the frame to the output and grab the next frame.we read the next frame into the queue provided the maxlength is not exceeded

```
    # if the warped frame is not None, then we know (1) we found the
    # four ArUCo markers and (2) the perspective warp was
successfully
    # applied
    if warped is not None:
        # set the frame to the output augment reality frame and then
        # grab the next video file frame from our queue
        frame = warped
        source = Q.popleft()

    # for speed/efficiency, we can use a queue to keep the next
video
    # frame queue ready for us -- the trick is to ensure the queue
is
    # always (or nearly full)
    if len(Q) != Q.maxlen:
        # read the next frame from the video file stream
        (grabbed, nextFrame) = vf.read()
```

```
        # if the frame was read (meaning we are not at the end of
the
        # video file stream), add the frame to our queue
        if grabbed:
            Q.append(nextFrame)
```

6) Script:Display the augmented reality frame and listen for key ,if key is q then it will quit else it repeats the loop. The the loop is over we destroy the output window and stop the webcam.

```
    # 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()
```