Panoramic Images from Video

Completed by Lonnie Chien, Suraj Kalidindi

For this project, we created a Panorama picture generator. It takes in a video as input, and converts it into a panorama image using repeated image stitching. The input video is in the form of a "panning shot," as it is called in film. This is a technique where the camera is in a fixed and central location, and camera person rotates their body to capture a video of the scene. We used our project 4 code as a starting point.

Where we got our videos from:

art: https://www.youtube.com/watch?v=V5tGanZR3XU

woman: https://www.youtube.com/watch?v=eBL6vu9NQtw

First, we want to process the video and get the images that we will later on stitch together. To do this, we first crop/resize the video to a window of a consistent size, to make it a bit easier to work with. Then, we break up the video into its individual frames to be extracted as images.

Even a short, 10-second video clip can have hundreds of frames. It is not necessary to use that many images to build the panorama, so instead we only need to keep every 20th, 30th, or Nth frame. How large N should be depends on a few factors: first, the quality of the video in terms of FPS (frames per second). If the video takes many frames per second, most of the frames will appear similar, so more of them can be thrown out. Second, the speed at which the camera is rotated. If the camera moves quickly, we will have to choose frames closer together so we don't lose features in the panorama.

To create a stitch, we used a stitching function that is similar to the one we created in Project 4. The function takes in two images, and a set of pixel coordinates (points) for matching features between the images. The stitcher function finds the homography that transforms the first set of points onto the other, warps the images, and superimposes them.

We used the SIFT detector perform the feature matching. The matching algorithm uses k nearest neighbors to find many matching points, then uses the ratio test to immediately filter out some of the bad matches. The ratio test works by matching keypoints between the first and second images, and finding the best matches for the keypoints. That is, the two nearest neighbors to the feature descriptor. It finds the difference using the ratio of distances, and if the distances are not different enough then the match is thrown out. We then use RANSAC (as it is implemented in Project 4) to further improve our set of matching points.

Matches are found and stitches are made iteratively. We begin by stitching together the first two images. Then, result image and the third image are stitched together. Then that result is stitched with the fourth image, and so on. After all images are stitched, the resulting image is a panorama. Finally, we can improve the result by warping it to even out some of the differing widths and heights that resulted from the stitches.

A challenge we faced in this project was getting quality matches from the feature detector. We started with ORB detector, but later found that SIFT works better. SIFT takes longer than ORB, so we had to downsample the dataset. We also found that indoor videos tended to work better than outdoor videos. This is because even after performing the ratio test and RANSAC on our matches, different trees or foliage just have similar features, which resulted in a lot of false positive matches. Another challenge was that the basic method for combining the warped images in the stitcher was creating dark shades in regions of the images as older parts of the images lost intensity (geometric sequence). The solution was by indexing the warped images with masking.

Our code follows:

```
import numpy as np
import numpy.ma as ma
```

```
import sys, os
import cv2
import matplotlib.pyplot as plt
import random
import time
```

Resize the video and extract the frames as images:

```
def resize_video(input_filename, start=0, end=None):
    """Resize frames `start` to `end` of video `input_filename`

    Resizes to 640x480 (or smaller depending on aspect ratio).
    """
    ################################################
    # get video properties

    if not os.path.exists(input_filename):
        print('error:', input_filename, 'does not exist')
        sys.exit(1)

    nframes, width, height, fps = get_num_frames(input_filename)

    print(input_filename, 'has', nframes,
          f'frames of size {width}x{height} at {fps} fps')

    ################################################
    # compute output size

    frac = 360.0/max(width, height)

    if frac >= 1.0:
        print('max dimension already <= 360, not resizing!')
        sys.exit(1)

    output_size = (int(round(width*frac)), int(round(height*frac)))

    print('will resize to {}x{}'.format(*output_size))

    ################################################
    # deal with start/end indices

    if end is None:
        end = nframes

    if start < 0 or start > nframes or end < 0 or end > nframes or end < start:
        print('invalid frame indices, must have 0 <= STARTFRAME < ENDFRAME <', nframes)
        sys.exit(1)

    if start > 0 or end < nframes:
        print(f'will write frames {start}-{end}')

    ################################################
    # create the output video
    path_prefix, basename = os.path.split(input_filename)
    basename, _ = os.path.splitext(basename)

    fourcc, ext = (cv2.VideoWriter_fourcc('m', 'p', '4', 'v'), 'mp4')
    output_filename = os.path.join(path_prefix, basename + '_resized.' + ext)

    writer = cv2.VideoWriter(output_filename, fourcc, fps, output_size)

    cap = cv2.VideoCapture(input_filename)
    frame_idx = 0

    while True:
        ok, frame = cap.read()
        if not ok or frame is None:
            break
        frame = cv2.resize(frame, output_size, interpolation=cv2.INTER_AREA)
        if frame_idx >= start and frame_idx < end:
            writer.write(frame)
        frame_idx += 1

    print(f'wrote {end-start} frames to {output_filename}')

def get_num_frames(input_filename):
```

```
    cap = cv2.VideoCapture(input_filename)

    # try the fast way
    nframes = cap.get(cv2.CAP_PROP_FRAME_COUNT)
    width = cap.get(cv2.CAP_PROP_FRAME_WIDTH)
    height = cap.get(cv2.CAP_PROP_FRAME_HEIGHT)
    fps = cap.get(cv2.CAP_PROP_FPS)

    if not fps > 1:
        # fps buggy sometimes
        fps = 30.0

    if nframes > 1 and width > 1 and height > 1:
        # it worked
        return int(nframes), int(width), int(height), fps

    # the slow way
    cnt = 0

    while True:
        ok, frame = cap.read()
        if not ok or frame is None:
            break
        height, width = frame.shape[:2]
        cnt += 1

    return cnt, width, height, fps
```

The following function is useful for warping our results at the end:

```
def t_homog(image, i, j, m):

    # Load an image
    orig = image.copy()

    # Get its size
    h, w = orig.shape[:2]
    size = (w, h)

    ################################################################################
    # Now make a neat "keystone" type homography by composing a
    # translation, a simple homography, and the inverse translation.

    # Translate center of image to (0,0)
    Tfwd = np.eye(3)
    Tfwd[0,2] = -0.5 * w
    Tfwd[1,2] = -0.5 * h

    # Get inverse of that
    Tinv = np.linalg.inv(Tfwd)
    # marker
    # Homography that decreases homogeneous "w" coordinate with increasing
    # depth so bottom rows appear "closer" than top rows".
    H = np.eye(3)
    H[i,j] = m

    S = np.eye(3)
    S[0,0] = 0.5
    S[1,1] = 0.5

    # Compose the three transforms together using matrix
    # multiplication.
    #
    # Use @ operator to do matrix multiplication on numpy arrays
    # (remember * gives element-wise product)

    H = S @ Tinv @ H @ Tfwd

    ################################################################################
    # Now translate the final warped image so we can see it all. This uses
    # the same trick from transrot.py, except instead of modifying the
    # homography matrix directly, just composes it with a translation.

    # Get corner points of original image - note this is shaped as an
    # n-by-1-by-2 array, because that's what cv2.perspectiveTransform
    # expects. If you have a more typical n-by-2 array, you can use
```

```
    # numpy's reshape method to get it into the correct shape.
    p = np.array( [ [[0, 0]],
                    [[w, 0]],
                    [[w, h]],
                    [[0, h]] ], dtype='float32' )

    # Map through warp
    pp = cv2.perspectiveTransform(p, H)

    # Get integer bounding box of form (x0, y0, width, height)
    box = cv2.boundingRect(pp)

    # Separate into dimensions and origin
    dims = box[2:4]
    p0 = box[0:2]

    # Create translation transformation to shift image
    Tnice = np.eye(3)
    Tnice[0,2] -= p0[0]
    Tnice[1,2] -= p0[1]

    # Compose them via matrix multiplication
    Hnice = Tnice @ H

    # Show it
    warpedNice = cv2.warpPerspective(orig, Hnice, dims)
    return warpedNice
```

The stitcher function finds the homography that transforms the first set of points onto the other, warps the images, and superimposes them.

```
def stitcher(img1, pts1, img2, pts2):

    """Stitches img1 and img2 into a single image using correspondences
    Inputs:
        img1, img2 - 2d np.array (np.uint8), of the images to stitch together
        pts1, pts2 - 2d np.array (np.float32, size 2 by n), of the points to stitch together. Each
                     row in pts1 corresponds to the same row in pts2
    Returns:
        result - 2d np.array (np.uint8) of warped image (properly cropped).
    """

    pts1 = pts1.reshape((len(pts1),1,2))
    pts2 = pts2.reshape((len(pts2),1,2))

    h,w,c = img1.shape
    h2, w2, c2 = img2.shape

    # find homography
    borderPoints1 = np.array([[0,0],[w,0],[w,h],[0,h]]).reshape((4,1,2))
    borderPoints1 = borderPoints1.astype(np.float32)
    borderPoints2 = np.array([[0,0],[w2,0],[w2,h2],[0,h2]]).reshape((4,1,2))
    borderPoints2 = borderPoints2.astype(np.float32)

    H, mask = cv2.findHomography(pts1,pts2)
    #p_trans = cv2.perspectiveTransform(pts1,H)
    borderPoints1_warped = cv2.perspectiveTransform(borderPoints1,H)

    allpts = np.concatenate((borderPoints1_warped,borderPoints2)).reshape(-1,2)
    allpts = allpts.astype(np.float32)

    # use boundingrect to get x0, y0

    x0,y0,w_br,h_br = cv2.boundingRect(allpts)

    # construct T
    T = np.eye(3,3,dtype='float32')
    T[0,2] -= x0
    T[1,2] -= y0


    # get M matrix M = T@H
    M = T @ H

    # Warp A&M, B&T

    img1_resized = np.pad(img1,[(0,h_br - h),(0,w_br - w),(0,0)])
```

```
    img2_resized = np.pad(img2,[(0,h_br - h2),(0,w_br - w2),(0,0)])

    warpA = cv2.warpPerspective(img1_resized,M,(w_br,h_br))
    warpB = cv2.warpPerspective(img2_resized,T,(w_br,h_br))

    # combine warped images

    m1 = ma.make_mask(warpA)
    m2 = ma.make_mask(warpB)

    m3 = np.logical_and(m1,m2)
    combined_img = np.zeros_like(warpA)

    combined_img[m1] = warpA[m1]
    combined_img[m2] = warpB[m2]
    combined_img[m3] = warpA[m3]//2 + warpB[m3]//2

    #Reversing warp

    return combined_img.astype(np.uint8)
```

RANSAC algorithm:

```
def ransac_homography(pts1, pts2, Nmax=1000, thresh=3):
    """Finds homography between pts1 and pts2 using RANSAC
    Inputs:
        pts1, pts2 - 2d np.array (np.float32, size 2 by n), of the points to stitch together. Each
                     row in pts1 corresponds to the same row in pts2. It may contain outliers.
        Nmax - int, maximum number of iterations (default 1000)
        thresh - float, threshold for accepting inlier (default 3)
    Returns:
        inliers - list of indices to rows of valid points
    """

    largest = [0]
    pts1 = pts1.reshape((len(pts1),1,2))
    pts2 = pts2.reshape((len(pts2),1,2))

    for i in range(Nmax):
        # select 4 random rows in pts1, pts2
        random_points =  random.sample(range(len(pts1)),4)

        # find homography b/t those points

        rand_pts1 = pts1[random_points,:]
        rand_pts2 = pts2[random_points,:]

        rand_pts1 = rand_pts1.reshape((4,1,2))
        rand_pts2 = rand_pts2.reshape((4,1,2))


        H, mask = cv2.findHomography(rand_pts1,rand_pts2)
        # Warp pts1 to pts2_est using this homography (using cv2.perspectiveTransform)
        pts2_est = cv2.perspectiveTransform(pts1,H)
        # Find all rows where pts2_est and pts2 are within thresh distance.
        inliers = []
        pts2 = pts2.reshape((len(pts2),2))
        pts2_est = pts2_est.reshape((len(pts2),2))
        for k in range(len(pts2)):
            #Finding euclidean distance between pts2 and pts2_est

            dist = ((pts2[k][0]-pts2_est[k][0])**2+(pts2[k][1]-pts2_est[k][1])**2)**0.5
            if dist <= thresh:
                inliers.append(k)

        # save largest inlier
        if len(inliers) > len(largest):
            largest = inliers


    return largest
```

The SIFT detector to find matches between two images, perform the ratio test, and return the sets of "good" points.

```
def get_points(img1, img2):
    '''
     gets matching points of two images
    Inputs:
        img1, img2
    Returns:
        pts1_ransac,pts2_ransac - lists of matching points for each image
    '''
    gray1 = cv2.cvtColor(img1, cv2.COLOR_RGB2GRAY)
    gray2 = cv2.cvtColor(img2, cv2.COLOR_RGB2GRAY)

    sift = cv2.SIFT_create()

    corners_1,des1 = sift.detectAndCompute(gray1,None)
    corners_2,des2 = sift.detectAndCompute(gray2,None)

    bf = cv2.BFMatcher()
    matches_all = bf.knnMatch(des1,des2,k=2)


    pts1 = []
    pts2 = []

    #Finding good matches
    matches = []
    for m,n in matches_all:
        if m.distance < 0.75*n.distance:
            matches.append([m])

    for i in range(len(matches)):

        (x1,y1) = corners_1[matches[i][0].queryIdx].pt
        (x2,y2) = corners_2[matches[i][0].trainIdx].pt

        pts1.append([x1,y1])
        pts2.append([x2,y2])


    pts1 = np.array(pts1,dtype=np.float32)
    pts2 = np.array(pts2,dtype=np.float32)

    inliers = ransac_homography(pts1, pts2, Nmax=500, thresh=5)
    pts1_ransac = pts1[inliers,:]
    pts2_ransac = pts2[inliers,:]


    return pts1_ransac, pts2_ransac
```

These are some main functions that make everything work in order:

```
def videoToFrames(video_file,N = 55):
    """Converts Video to a list of frames
    Inputs:
        video_file
        N - number of frames required in video
    Returns:
        images - list of frames
    """
    resize_video(video_file)

    # get frames out of resized video
    vidcap = cv2.VideoCapture(video_file)
    success,image = vidcap.read()
    count = 0

    while success:
        cv2.imwrite("/work/Frames/frame%d.jpg" % count, image)     # save frame as JPEG file
        success,image = vidcap.read()
        count += 1
```

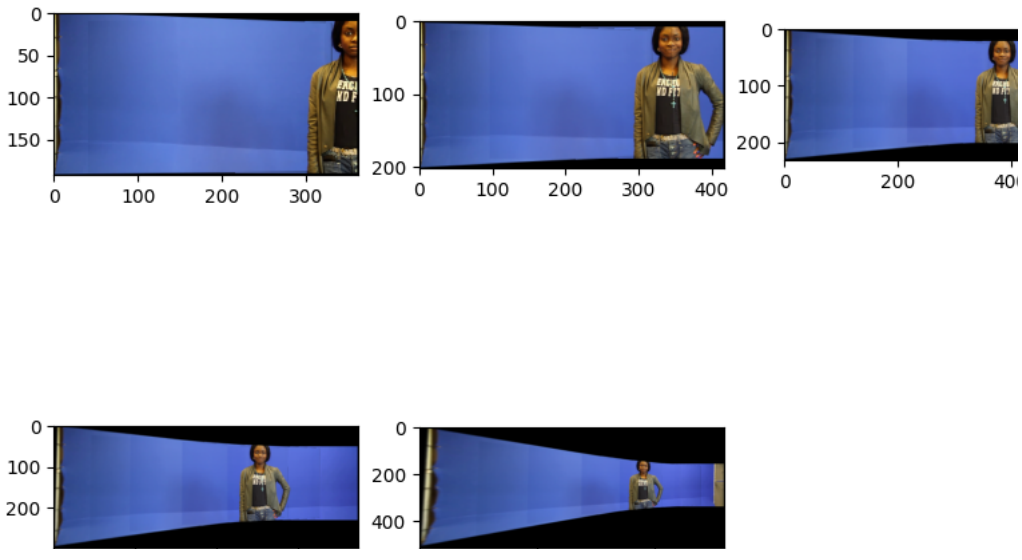
```
    # write each Nth frame into images list
    images = []
    for i in range(count//N):
        img_path = '/work/Frames/frame%d.jpg' % (i*N)
        img_selected = cv2.cvtColor(cv2.imread(img_path),cv2.COLOR_BGR2RGB)
        images.append(img_selected[::4,::4])
    return images


def makePano(images):
    """
    Makes the Panorama image
    Inputs:
        images - list of images to stitch together
    Returns:
        imgNew - Panorama image
    """

    imgNew = images[0]
    fig = plt.figure(figsize = (10,7))
    for i in range(len(images)-1):

        start = time.perf_counter()
        pts1,pts2 = get_points(imgNew,images[i+1])

        imgStitch = stitcher(imgNew,pts1,images[i+1],pts2)
        imgNew = imgStitch

        fig.add_subplot(int((len(images)-1)/3)+int((len(images)-1)%3>0),3,i+1)
        plt.imshow(imgNew)

    return imgNew
```

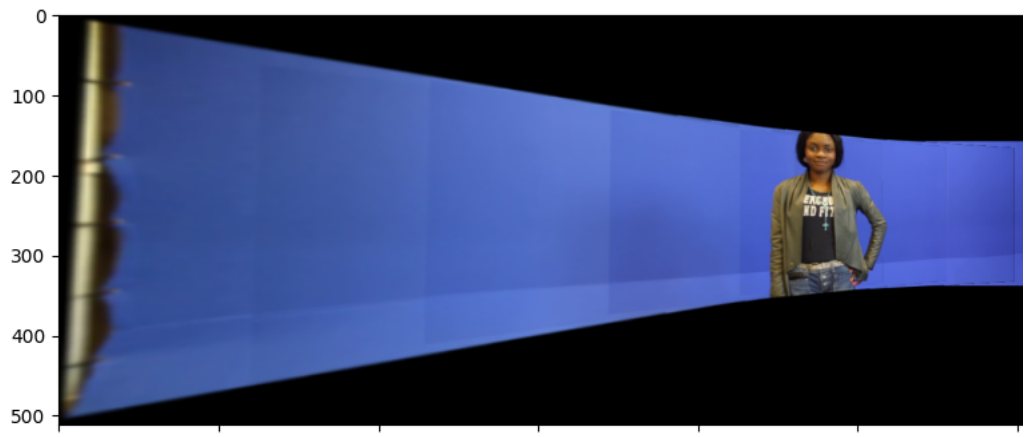
The following code displays the new image after each stitch

```
video_file = "/work/input video/woman.mp4"
images = videoToFrames(video_file,N = 75) # if run out of memory, increase value of N here
finalImage = makePano(images)
plt.figure(figsize = (10,7))
plt.imshow(finalImage)
```

```
/work/input video/woman.mp4 has 458 frames of size 1280x720 at 25.0 fps
will resize to 360x202
wrote 458 frames to /work/input video/woman_resized.mp4
```

```
<matplotlib.image.AxesImage at 0x7f45f89b47d0>
```





We can improve the image by warping it. As we can see from the figure above, we should increase the height more towards the right side of the image.

```
woman_warp = t_homog(finalImage,2,0,-0.00065)
plt.figure(figsize = (10,7))
plt.imshow(woman_warp)
```

```
<matplotlib.image.AxesImage at 0x7f45e9cd9590>
```


```
woman_output = cv2.cvtColor(woman_warp, cv2.COLOR_RGB2BGR)
cv2.imwrite('/work/output panoramas/woman_output.jpg',woman_output)
```

```
True
```

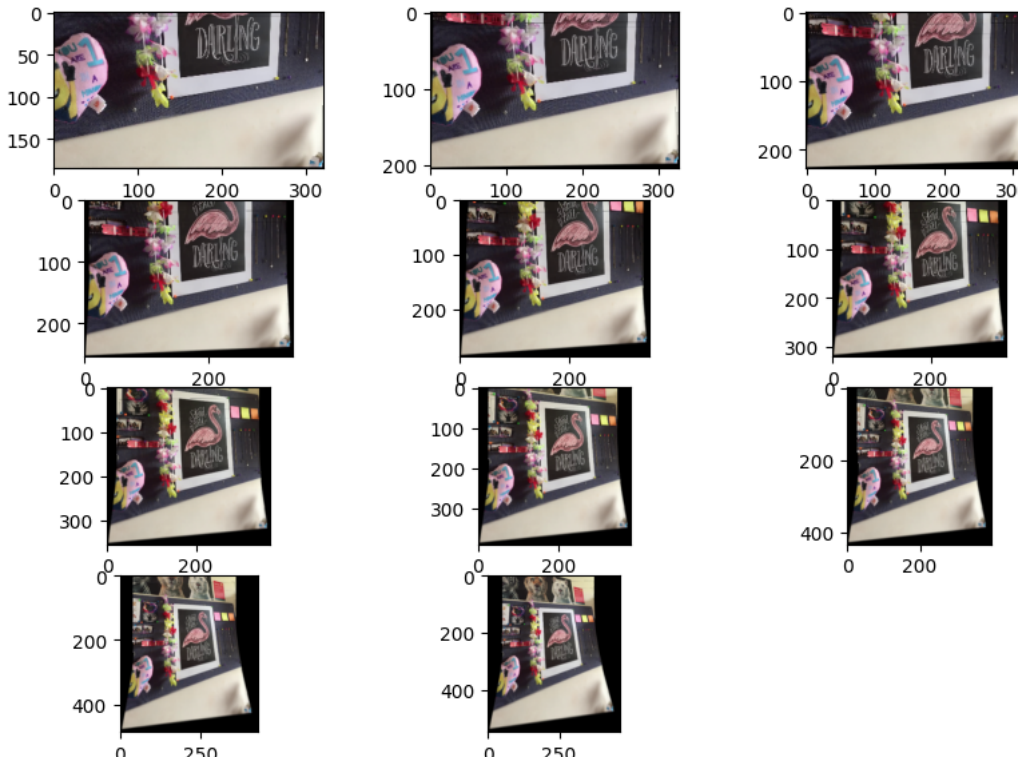
The second video we used is technically a "tilt shot," which is a type of panning shot where the camera moves up and down instead of left and right.

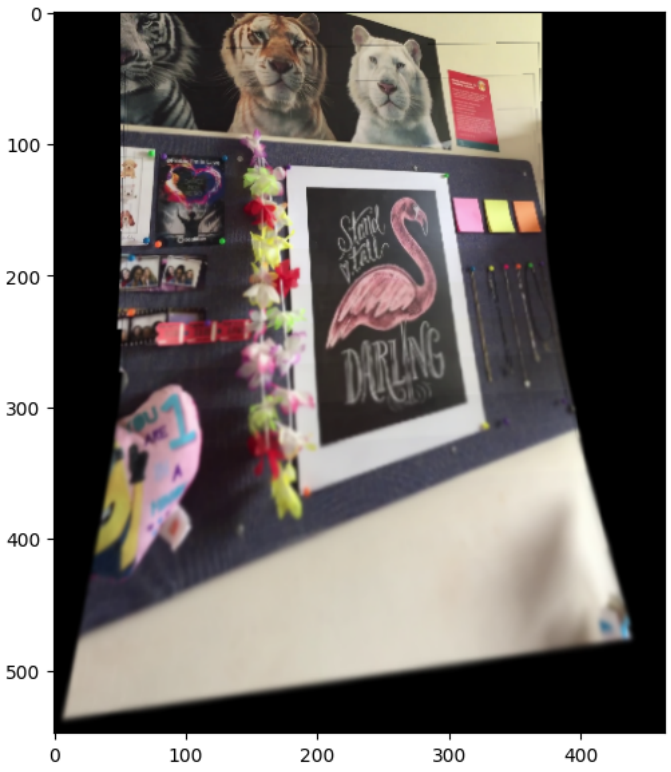
```
video_file = "/work/input video/art.mp4"
images = videoToFrames(video_file,N = 20) # if run out of memory, increase value of N here
finalImage = makePano(images)
```

```
plt.figure(figsize = (10,7))
plt.imshow(finalImage)
```

```
/work/input video/art.mp4 has 253 frames of size 1280x720 at 29.0 fps
will resize to 360x202
wrote 253 frames to /work/input video/art_resized.mp4
```

```
<matplotlib.image.AxesImage at 0x7f45e9b2b0d0>
```

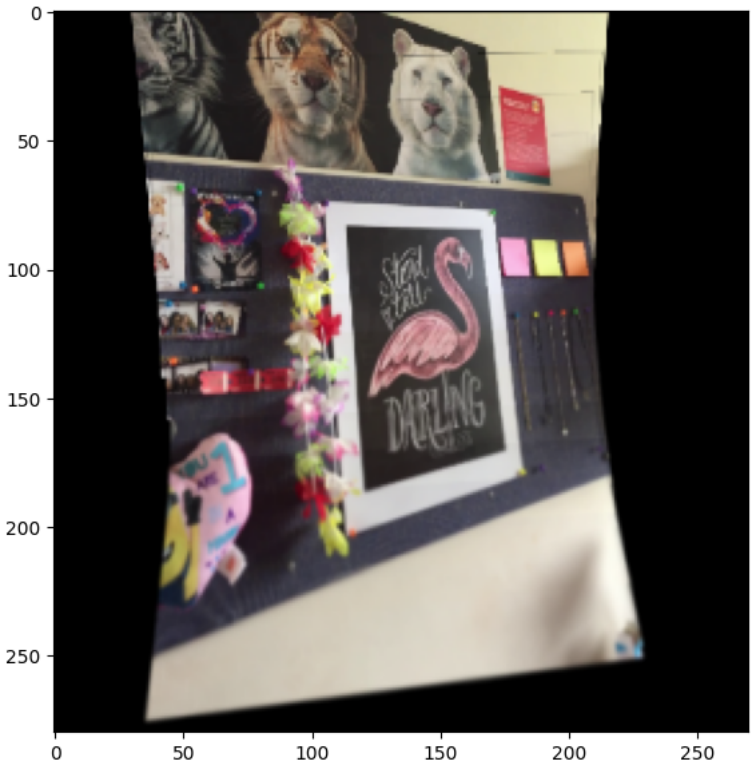




The strange warping of this image is due to the change in depth during the video (video starts from bottom of the wall, which is close, and goes further away to the top-right of the wall). To potentially fix this, we would need to find the corners of the panorama picture without the zero-padded pixels, and use those four points along with the four corner points of the bounding rectangle to create a homography matrix similar to how we did in the stitcher function.

```
art_warp = t_homog(finalImage,2,1,0.0005)
plt.figure(figsize = (10,7))
plt.imshow(art_warp)
```

```
<matplotlib.image.AxesImage at 0x7f45e99a9810>
```



Final image written to output file

```
art_output = cv2.cvtColor(art_warp, cv2.COLOR_RGB2BGR)
cv2.imwrite('/work/output panoramas/art_output.jpg',art_output)
```

```
True
```