Partial video de-animation

Or how to create a dancing glass and
self-playing foosball table.

This project demonstrates one of the ways how to de-animate video partially making certain parts of it dynamic while the rest would be still like a photo.

The Goal of Your Project

Original project scope:

Original scope of the project was to choose good setting for cinemagraph, take a video of it and then de-animate it in an interesting way. The goal was to create smooth final video artifact that would include object of interest animated while keeping its surrounding static.

What motivated you to do this project?

Cinemagraph is becoming very popular way to express one's creativity. And with availability of digital frames nowadays it's even possible to have your own Harry Potter style animated image hanging in your house.

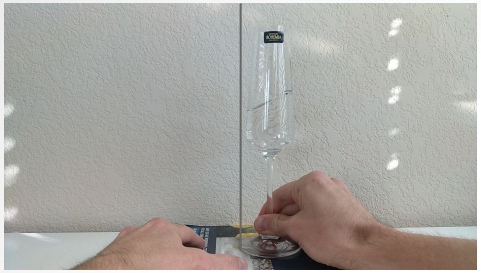
As such I wanted to try and learn how to create my own de-animated videos with effects that normally would be difficult to achieve on a static photo.

Scope Changes

- Did you run into issues that required you to change project scope from your proposal?
  I was able to achieve the minimum requirements described in the scope of this project.
  But the issues I faced is in the initial approach which included feature matching. That approach was computationally expensive and unreliable when processing hundreds of frames. Identified features weren't always the same and matching differ from frame to frame which made producing smooth video almost impossible.
- Give a detailed explanation of what changed
  As a way to fix above mentioned problems I tried the simplified approach from [1] using feature points tracking instead of feature matching. This proved to be much better both in terms of processing speed and reliability.

Showcase

Frame 0



Frame 63

https://youtu.be/gHL4WiJnB40

Frame 0

Frame 63

https://youtu.be/kE1zf_hruRI

Frame 100

Frame 200

https://youtu.be/AGpKgMAi5FA

Frame 100

Frame 200

https://youtu.be/GLGEVXxhoMI

Input

Output

Project Pipeline

1. Pipeline accepts videos as input.
   Manual.
2. Extracts frames from the video and saves them to temp folder for future processing.
   Automatic; Manual Inputs: number of frames to extracts, number of frames to skip in the beginning, frame resize ratio.
3. Detect good feature points. Uses implementation of minimal eigenvalue of gradient matrices for corner detection to find strong corners as good features for tracking. Features should be concentrated on the primary de-animation object (the one that should remain in the same place in the final video).
   Automatic; Manual Inputs: mask or region boundaries of the de-animation object; Used functions: cv2.goodFeaturesToTrack.
4. Track features using Lucas Kanade optical flow tracker with forward-backward validation of good points.
   Automatic; Used functions: cv2.calcOpticalFlowPyrLK
5. Warp frames using found and tracked features points. As the result object of interest should remain in the same place in the warped frame as in the first frame allowing good match between first static frame and all the rest.
   Automatic; Manual Input: warping method (Translation, Affine or Homography) (choose one that works best); Used functions: cv2.getAffineTransform, cv2.findHomography, cv2.warpAffine, cv2.warpPerspective
6. Blend first frame and other frames with mask showing boundaries between static and dynamic regions using alpha-blending.
   Automatic; Manual Input: dynamic mask
7. Save blended images into final video. *Automatic; Manual Input: output video path, fps*

Demonstration: Result Sets

- Experiment 1 (Glass of water)
  - Input: https://youtu.be/gHL4WiJnB40
  - Output: https://youtu.be/kE1zf_hruRI
  - Description: In this experiment I have a glass of water that I move sideways to disturb water. The goal was to make it look like the glass is actually barely moving and that water is moving on its own.
  - Process: To process this video I used label on the glass as a de-animation object since glass itself is transparent and isn't easy to track. Because all tracking points in this case would concentrate on relatively small area of the image homography or affine transformation would likely be overestimated for the rest of the image, so I used simple translation transformation in this case, which worked good considering that the glass was moving mostly in X direction.
    Picture of the right shows tracking points on the glass.

Demonstration: Result Sets (cont'd)

- Experiment 2 (Foosball table)
  - Input: https://youtu.be/AGpKgMAi5FA
  - Output: https://youtu.be/GLGEVXxhoMI
  - Description: In this experiment I took a video of people playing foosball. I had an idea to de-animate the players and make it look like the foosball table is playing on its own while players just hold the handles.
  - Process: To process this video I used table base itself as a de-animation objects which helped to keep it in the same place despite unstable video. After that dynamic mask ensured that table game area showed the game while everything else, including players, remained static. Picture of the right shows tracking points on the table.

Project Development

- For this project I got my inspiration from work on selective video de-animation [1].
- At first I only wanted to take the general idea, simplified pipeline and try to use method and algorithms that I already learned throughout this class and see if I can achieve good results using just that.
- What I tried to implement is to
  - Define de-animation object (the one that should always remain in the same spot)
  - Extract that object from the first frame
  - Detect that same object on other frames using ORB feature matching
  - Warp all the frames using ORB features and homography transformation
  - Blend first frame and all the rest using pre-defined feathered mask and alpha blending
  - Save new frames into a final video
- What has proven to be challenging if not impossible is to detect de-animation object on all the frames and find consistent feature points. First it was computationally expensive when applied to hundreds of frames, second that process was unreliable as ORB detector couldn't find object of interest with enough precision which made video shake if not distorted.
- To fix this problem I again referred to original paper [1] and tried to research method that authors used there. Instead of tracking some object they used that object to find feature points first and then used popular Lucas Kanade method to track those points through the video.
- This method has proven to be much more reliable and faster since it relied on the assumption that points between images moved only to a very small distance, which is a good assumption to make for the video frames.
- But even that method wasn't bulletproof as some points sometimes can be mapped to several other points on the consecutive frames due to local similarity.
- After some research I found about popular method of feature match validation that works well with Lucas Kanader: forward-backward validation [3]. This method after finding feature points on the following frame does the opposite and uses these points to find their location on previous frame. Points that couldn't be mapped to the same location they originated from are deemed unreliable and discarded.

Project Development (cont'd)

- After previously mentioned major changes to the original pipeline I also had to modify my warping method. Some of my videos had a very small region that was good to track making tracking points too close to each other compared to the size of the image. As the result, even the smallest error in points location (fraction of the pixel) caused a major changes on the whole warped image. The tracked object seemed to remain on the same place, but the rest of the frame was distorted to the level that it was impossible to perform good blend later.
- Since tracking bigger object was difficult due to its nature (transparent glass) and quality of the video (frames become blurred at the fast motions) I tried to decrease degrees of freedom at the warping stage trying other transformations: affine and translation.
- Affine transformation showed better results that homography but still suffered from the same problem (small changes in tracking points location lead to big changes in the whole image). But since on the glass video object was moving mostly sideways Translation transformation which incorporated only movements along x and y axis showed decent results in the end. Though slight changes in the object orientations (rotation, perspective change) remain visible on the final artifact.
- But on the other image with much larger tracking object and as the result more spread-out feature points homography warping showed excellent results.
- In the end I was able to achieve good results with simplified pipeline from [1]. In future I would like to try and incorporate other techniques used in that paper, including but not limited to refined warping using dynamic feature points and graph-cut blending.

Computation: Code Functional Description

- Here I'll try to describe primary functions of my pipeline. There are also additional functions (extracting frames, saving frames, creating video), but due to their simplicity they aren't covered in this section.
- First important function is find_features. It's purpose is to find location of the good features for later tracking in a specific region of the image. That region can be defined in two ways: using bounding box defined by the coordinates of its top-left and bottom-right corners and using de-animation mask similar to approach in [1].
- Whatever method is used to define region feature points themselves are discovered using method cv2.goodFeaturesToTrack. This method implements feature detection using minimal eigenvalue of gradient matrices to detect corners in the image which is known to be a good feature points for detection and tracking. One of the main parameters of this method is qualityLevel which defines minimal accepted quality of the image corners that is specified by minimal eigenvalue for that corner. In my case value 0.1 gave me enough good features that were easy to track and at the same time reliable during warping stage.

```
def find_features(img, mask_path=None, bb=None):
    g = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)  # get grayscale image

    if mask_path is not None:
        deanimate_mask = cv2.imread(mask_path)
        deanimate_mask = cv2.cvtColor(deanimate_mask, cv2.COLOR_BGR2GRAY)
        _, deanimate_mask = cv2.threshold(deanimate_mask, 127, 255, cv2.THRESH_BINARY)

        pts = cv2.goodFeaturesToTrack(g,
                                      maxCorners=1000,
                                      qualityLevel=0.1,
                                      minDistance=3,
                                      blockSize=3,
                                      mask=deanimate_mask)
    elif bb is not None:
        img1 = img[bb[0]:bb[2], bb[1]:bb[3]]  # crop the bounding box area
        g = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)  # get grayscale image
        pts = cv2.goodFeaturesToTrack(g,
                                      maxCorners=1000,
                                      qualityLevel=0.1,
                                      minDistance=3,
                                      blockSize=3)

        # pt is for cropped image. add x, y in each point.
        for i in xrange(len(pts)):
            pts[i][0][0] += bb[1]
            pts[i][0][1] += bb[0]
    else:
        raise Exception('deanimate mask or tracking box must be provided')

    return pts
```

Computation: Code Functional Description

- Next function *track* tracks feature points across two consecutive frames.
- First it uses Lucas Kanade algorithm (LK) to calculate optical flow on the whole image. This flow defines direction of the changes in the frame and amount of that change. To calculate this flow LK uses gradient of changes across frames and assumption that those changes are small.
- To overcome limitation of of small change LK can be enhanced using Image pyramids that would allow to find bigger changes in some regions of the image.
- Both of this methods are implemented in cv2.calcOpticalFlowPyrLK method.
- Additionally, as it was discussed earlier, to make sure that we consistently discover the same points I implemented forward-backward validation where I predict points location in the next frame and then reverse the procedure to find the same points again in the first frame. If differences in original location and reverse predicted location is less than one pixel then the feature point is proven to be good. Bad points are later discarded.

```python
def track(img1, img2, p0):
    """
    Uses Lucas Kanade method to track points p0.
    Calculates forward optical flow to get new points location p1.
    Calculates backward optical flow to see which points can still be mapped to p0.

    :param img1: current frame
    :param img2: next frame
    :param p0: points location on current frame
    :return: p1: new location of p0, good: mask of the matching points
    """
    oldg = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    newg = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    p1, st, err = cv2.calcOpticalFlowPyrLK(oldg, newg, p0,
                                           None, **lk_params)
    p0r, st, err = cv2.calcOpticalFlowPyrLK(newg, oldg, p1,
                                            None, **lk_params)
    d = abs(p0 - p0r).reshape(-1, 2).max(-1)
    good = d < 1

    return p1, good
```

Computation: Code Functional Description

- Final two important functions in my pipeline are *warp* and *blend.* Their functionality is very similar to other projects I did in this class.
- Warp function implements 3 types of warp transformation:
  - Translation, that moves image in X and Y direction based on average changes in feature points location;
  - Affine transformation, which allows for more degrees of freedom and requires three best points to calculate transformation matrix;
  - And finally homography which uses RANSAC algorithm to find homography matrix (cv2.findHomography) and later warp the image (cv2.warpPerspective). This method allows to match images accurately no matter the changes (translation, rotation, perspective change, etc.)
    In this method the more feature points we have the better warping we can achieve (considering that all points are more or less accurate), since cv2.findHomography will use all of them to minimize retranslation error to find best possible transformation matrix.
- And finally blend function incorporates very basic alpha blending using pre-defined mask. First frame of the image is multiplied by mask values and added to the rest of the frames which are multiplied by 1-mask value. The results is smooth blend without noticeable borders (considering good matching of the first frame and warped frames)

```python
def warp(pts1, pts2, img, method='h'):
    """
    Warp img using matching points pts1 and pts2 and one of three methods:
    Translation (t), Affine (a) or Homography (h)
    """
    dsize = img.shape[1], img.shape[0]
    if method == 't':
        dx = (pts2[:5] - pts1[:5])[:,0,0].mean()
        dy = (pts2[:5] - pts1[:5])[:,0,1].mean()
        M = np.array([[1,0,dx],
                      [0,1,dy]])
        warped_img = cv2.warpAffine(img, M, dsize)
    elif method == 'a':
        pts1 = pts1[:3,0,:]
        pts2 = pts2[:3,0,:]
        M = cv2.getAffineTransform(pts1,pts2)
        warped_img = cv2.warpAffine(img, M, dsize)
    elif method == 'h':
        H, _ = cv2.findHomography(pts1, pts2, cv2.RANSAC, 5.0)
        warped_img = cv2.warpPerspective(img, H, dsize)
    else:
        raise Exception('not supported warping method')

    return warped_img


def blend(img1, img2, mask):
    """
    Blend two images together using alpha-blending
    """
    blend = np.zeros(img1.shape, dtype=np.uint8)
    if len(img1.shape) == 2:
        blend = img1 * mask + img2 * (1-mask)
    else:
        for ch in range(3):
            blend[:,:,ch] = img1[:,:,ch] * mask + img2[:,:,ch] * (1-mask)
    return blend
```

Resources

[1] Selectively De-Animating Video, *Jiamin Bai, Aseem Agarwala, Maneesh Agrawala, Ravi Ramamoorthi*

[2] Lucas Canade optical flow method: https://en.wikipedia.org/wiki/Lucas%E2%80%93Kanade_method

[3] Lucas Kanade Tracker: https://jayrambhia.com/blog/lucas-kanade-tracker

[4] OpenCV documentation: https://docs.opencv.org/

Appendix: Your Code

Code Language: Python

List of code files:

- *main.py*: executes pipeline
- *pipeline.py*: contains all the methods of the pipeline
- *video_processor.py*: extracts frames from the video and saves frames sequence to the video