ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Corso di Laurea Magistrale in Ingegneria Informatica

Augmented Reality Project

Computer Vision and Image Processing M

Cristian Davide Conte 0001034932

Academic Year: 2022-2023

Contents

Chapter 1

Abstract

The goal of this project is the development of an Augmented Reality System. In particular we're given a video sequence named "Multiple View.avi" and we have to superimpose, as realistically as possible, an augmented reality layer onto it. The video decipts the well-known book "Multiple View Geometry in Computer Vision", by Richard Hartley and Andrew Zissermann as seen by a moving camera which slowly translate from one position to another while also rotating and changing brightness towards the end of the sequence.
We're also given an image containing the first frame of the video, two binary masks (one for the book cover of the reference frame and one for the augmented reality layer) and the layer itself.
Our output should be a video named "Augmented Multiple View.avi" containing the original video after the superimposition of the augmented reality layer.
This paper proposes three alternative solutions to the above described problem and they're all based on finding some salient points inside the video frames, tracking their positions, calculating an homography and superimposing the augmented reality layer by warping it according to the just found homography.

Chapter 2

Images preparation

The first thing we need to do is to prepare the input images before doing any elaboration. In order to do this, we'll use the "OpenCV" and "Numpy" python libraries, here referred to as "cv2" and "np".

```python
import cv2
import numpy as np
```

2.1 Reading the input images

In order to read the images we'll use the "cv2.imread" function which reveals that the reference frame and the augmented reality layer images' colorspaces are "BGR". This indicates that the video sequence will probably also be encoded as "BGR" and that we need a conversion to the "RGB" colorspace. Furthermore the reference frame and the augmented reality layer have different resolution which makes any overlaying between the two impossible.

For the BGR to RGB conversion we'll use the "cv2.cvtColor" function:

```python
# Read the reference frame
reference_frame_bgr = cv2.imread(reference_frame_path)
reference_frame_rgb = cv2.cvtColor(
    reference_frame_bgr,
    cv2.COLOR_BGR2RGB
) #480x640

# ...
```

Next we make sure that the ar layer and the reference frame (and thus the video sequence) have the same width/height:

```python
augmented_layer_rgb = augmented_layer_rgb[
    :reference_frame_rows, #reference_frame_rgb.shape[0]
    :reference_frame_cols  #reference_frame_rgb.shape[1]
] #crop from 480x1525 to 480x640

# Same for the ar layer's binary mask
# augmented_layer_mask = ...
```

2.2 Improving the AR Layer

After having cropped the augmented reality layer, we've noticed that it includes the whole authors' section instead of just the name of the third author and this would make it clearly visible whenever the video brightness changes, so some masking needs to be done. Since the new author's related pixels have a uniform color which is different from the ones of the surrounding area, all we need to do is to select those pixels inside the ar-layer, extract them, use them to filter the bottom part of the corresponding binary mask and apply the new mask to RGB layer.

This is the code for the extraction:

```python
third_author_mask = np.zeros(
    (reference_frame_rows, reference_frame_cols, 3),
    dtype = np.uint8
) #black mask
third_author_color = [201, 255, 255] #rgb

third_author_pixels = np.where(
    (augmented_layer_rgb == third_author_color).all(axis = 2)
)
```

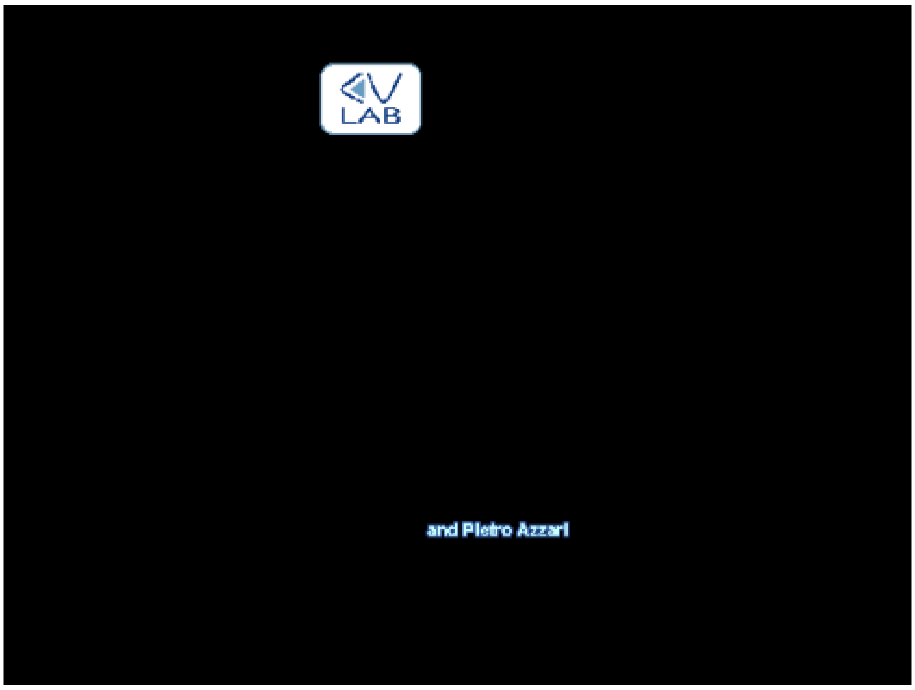
In order to avoid artifacts during the warping of the layers, it's better to slightly "expand" the mask area around the third author. This can be easily achieved by applying a 3x3 dilation kernel to the binary mask since it's an inherently binary image containing just one big blob.

```python
dilation_kernel = np.ones((3, 3), np.uint8)
third_author = cv2.dilate(third_author_mask, dilation_kernel, 1)
```

We can now filter the original ar binary mask and apply it to the augmented layer.

```python
augmented_layer_mask[300:,:] = third_author[300:, :]
augmented_layer_rgb[np.where(augmented_layer_mask == 0)] = 0
```

The augmented layer now looks like this:



Towards the end of the input video sequence, the brightness of the scene changes rapidly and this means that superimposing just the augmented layer above would result in the book rapidly changing colors/brightness but nothing would happen to the augmented layer. One of the requirement of the project is that "one would perceive the overlaid graphics as real items present in the book cover" but because of the last few frames of the video, this cannot be achieved consistently. It is then better to merge the binary mask of the whole book (below referenced to as "object mask") with the modified one of the augmented layer and superimpose this new layer. This makes the brightness changes in the scene almost unnoticeable thus achieving the requested "as realistically as possible" superimposition. Note that this step will not affect any of the solutions/results and the jupyter notebook code will allow to superimpose both layers interchangeably.

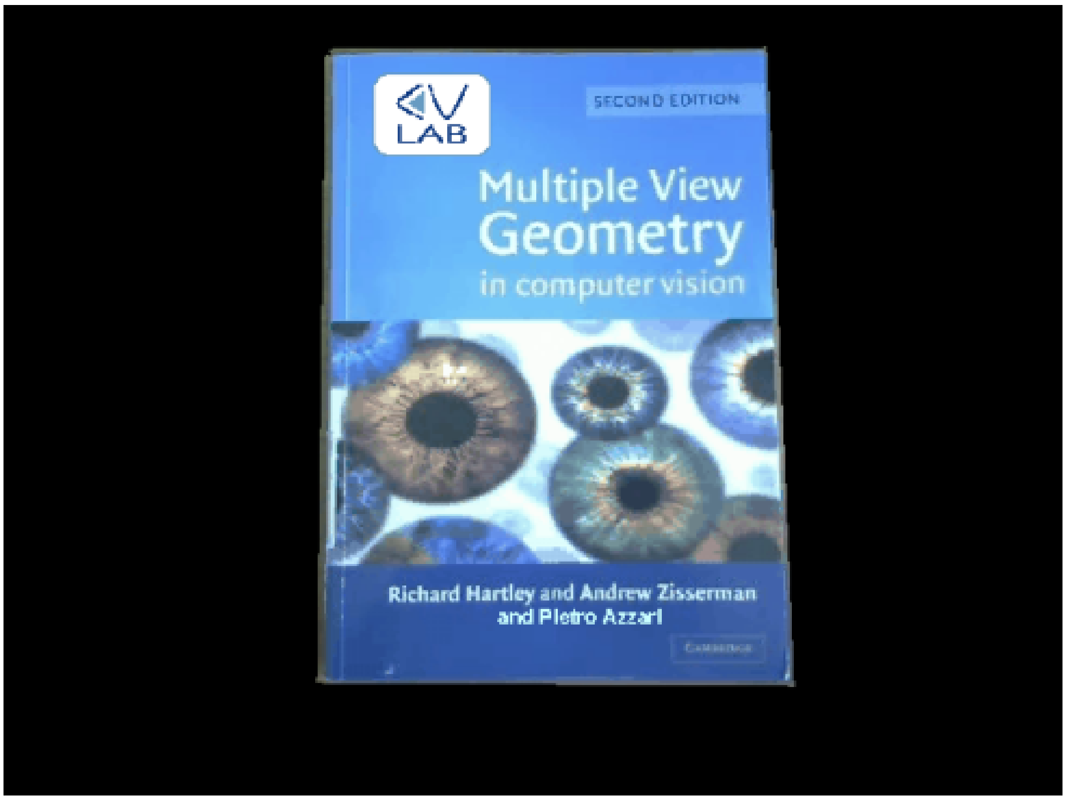
This is the code responsible for the creation of the new augmented reality layer:

```
# Apply the object mask to the reference frame to isolate the book
target_image = cv2.bitwise_and(reference_frame_rgb, object_mask)

augmented_layer_mask_pixels = np.where(augmented_layer_mask == 255)

# Merge the original book cover and the ar-layer into the new ar-layer
target_image_augmented = target_image.copy()
target_image_augmented[augmented_layer_mask_pixels] =
    augmented_layer_rgb[augmented_layer_mask_pixels]
```

The final ar-layer looks like this:



Chapter 3

Proposed solutions

The general idea behind these solutions is to find at least 4 salient points present in the video frames, track their positions and find a homography between the frames and the augmented reality layer and its mask in order to warp the latters accordingly. Once produced, the augmented frames will be stored into arrays for a faster retrieval later on (∼1Mb of total RAM space each).

Since in all the three solutions we need to scrub through the input video sequence, it's convenient to define a function that does that:

```python
def process_video(path, callback):
    # Read the input video via the VideoCapture API
    video = cv2.VideoCapture(path)

    previous_frame = None

    # Play the video
    try:
        while(video.isOpened()):
            frame_read_correctly, current_frame = video.read()

            if not frame_read_correctly:
                break

            if current_frame is not None:
                # Call the augmentation systems of the next sections
                callback(current_frame, previous_frame)
                previous_frame = current_frame

        video.release()
    except KeyboardInterrupt:
        video.release()
```

3.1 Frame to Reference solution

This augmentation system compares keypoints found in a reference frame with the ones found in the frame that's currently being analyzed: in this case the reference frame is the first frame of the video sequence. In order to detect the keypoints and compute their characteristics (descriptors), the DoG + SIFT cv2 implementation is used inside an utility function defined as:

```python
# SIFT detector instance
sift = cv2.xfeatures2d.SIFT_create()

def run_SIFT(image):
    # Detecting keypoints in the image
    keypoints = sift.detect(image)

    # Computing the descriptors for each keypoint
    return sift.compute(image, keypoints) # keypoints, descriptors
```

SIFT is a good choice since it's rotation and scale invariant and robust to intensity changes. Since SIFT only works with grayscale images, every image that will be passed to the "run_SIFT" function will be converted accordingly. Now we can find the keypoints inside the reference frame and compute their descriptors (offline).

```python
# Initial keypoints to keep track of
kp_query, des_query = run_SIFT(reference_frame_grayscale)
```

Once done that, we can proceed to the detection phase of the next frame:

```python
def process_video_F2R(current_frame, _):
    # First frame processing (already done in the offline phase)
    ...
    # Create a grayscale version of the current_frame for SIFT
    current_frame_grayscale = cv2.cvtColor(
        current_frame,
        cv2.COLOR_BGR2GRAY
    )

    # Run SIFT on the current frame
    kp_train, des_train = run_SIFT(current_frame_grayscale)
```

At this point we try to match the keypoints of the reference frame with the just-found ones. This is a multidimensional nearest-neighbours-finding problem which we solve by using the L_2 norm (euclidean distance) between descriptors with the support of a k-d tree in order to speed up the operations.

```python
    # Initializing the matching algorithm
    index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
    search_params = dict(checks = 50)
    flann = cv2.FlannBasedMatcher(index_params, search_params)

    # Matching the descriptors
    matches = flann.knnMatch(des_query, des_train, k = 2)
```

The found nearest neighbours don't necessarily provide valid correspondences as some features in the reference frame may not have a corresponding feature in the current frame and viceversa. In order to solve this problem we use Lowe's ratio with T = 0.7.

```python
    good = [] # list of valid matches
    for m,n in matches:
        if m.distance < 0.7 * n.distance:
            good.append(m)
```

If 4 (minimum) or more points are found an homography between the reference frame and the current frame is estimated. In order to exclude possible outliers from this process we use the ransac method (implemented as "cv2.RANSAC").

```
    # Check if SIFT found the minimum number of matches
    if len(good) < MIN_MATCH_COUNT: # 4 or more
        return

    # Extraction of good reference/current frames' keypoints
    src_pts = #...
    dst_pts = #...

    # Calculation of the homography based on correspondences
    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
```

Since the homography between the augmented reality layer and the reference frame is the identity, we can now directly proceed to warp the augmented reality layer onto the current frame (otherwise we would've had to find the reference frame-to-ar layer correspondence first).

```
    # Warping the augmented layer
    warped_augmented_frame = cv2.warpPerspective(
        target_image_augmented, # or augmented_layer_rgb
        M,
        (reference_frame_cols, reference_frame_rows)
    )

    # Warp a white mask to use as the place to
    # put the augmented layer's pixels onto
    warp_mask = cv2.warpPerspective(
        object_mask, # or augmented_layer_mask
        M,
        (reference_frame_cols, reference_frame_rows)
    )
    warp_mask_white_pixels = np.where(warp_mask == 255)

    # Superimpose the augmented reality layer over the current frame
    final_frame = cv2.cvtColor(current_frame, cv2.COLOR_BGR2RGB)
    final_frame[warp_mask_white_pixels] =
        warped_augmented_frame[warp_mask_white_pixels]
```

Finally we store the augmented frame inside a buffer so that later on it will be easier to retrieve it.

```
    F2R_frames_buffer.append(final_frame)
```

This process is repeated for every frame of the video sequence.

```
F2R_frames_buffer = []
process_video(video_path, process_video_F2R)
```

3.2 Frame to Frame solution

This augmentation system compares keypoints found in the current frame with the ones found in the previously analyzed frame. In order to detect the keypoints and compute their characteristics (descriptors), the DoG + SIFT cv2 implementation is used inside the "run_SIFT" utility function described in the 3.1 chapter. Again, since SIFT only works with grayscale images, every image that will be passed to the "run_SIFT" function will be converted accordingly. In order to warp the augmented layer onto the current frame we need to "keep track" of all the homographies found and then apply them at once to the ar-layer. This is achieved with the help of a "base" homography matrix that will be incrementally "updated" (dot product) with the found homographies.

The mapping between the first frame of the video and the ar-layer is the identity, so our "base" homography matrix should respect this correspondance:

```
# 3x3 identity matrix
homography_matrix = np.identity(3, dtype = np.float32)
```

Since the first frame of the video doesn't have a previous frame to compare it to, we find the first pair of (keypoints, descriptors) inside the reference frame.

```
# Initial keypoints to keep track of
kp_query, des_query = run_SIFT(reference_frame_grayscale)
```

Once done that, we can proceed to the detection phase of the next frame:

```
def process_video_F2F(current_frame, previous_frame):
    # First frame processing (skipped)
    # ...

    # Create a grayscale version of the current_frame for SIFT
    current_frame_grayscale = # ...

    # Run SIFT on the current frame
    kp_train, des_train = run_SIFT(current_frame_grayscale)
```

At this point we try to match the keypoints of the previous frame with the just-found ones. Again, this is a multidimensional nearest-neighbours-finding problem which we solve by using the L_2 norm between descriptors with the support of a k-d tree.

```
    # Initializing the matching algorithm
    index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
    search_params = dict(checks = 50)
    flann = cv2.FlannBasedMatcher(index_params, search_params)

    # Matching the descriptors
    matches = flann.knnMatch(des_query, des_train, k = 2)
```

As in the F2R solution, we use Lowe's ratio with $T = 0.7$.

```
    good = [] # list of valid matches
    for m,n in matches:
        if m.distance < 0.7 * n.distance:
            good.append(m)
```

If 4 or more points are found an homography between the previous frame and the current frame is estimated. In order to exclude possible outliers from this process we use the ransac method.

```
    # Check if SIFT found the minimum number of matches
    if len(good) < MIN_MATCH_COUNT: # 4 or more
        return

    # Extraction of good reference/current frames' keypoints
    src_pts = ...
    dst_pts = ...

    # Calculation of the homography based on correspondences
    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
```

Now we use the homography between the current frame and the previous one to update the total homography matrix.

```
    homography_matrix = np.dot(homography_matrix, M)
```

Since the homography between the augmented reality layer and the reference frame is the identity, we can now directly proceed to warp the augmented reality layer onto the current frame.

```
    # Warping the augmented layer
    warped_augmented_frame = cv2.warpPerspective(
        target_image_augmented, # or augmented_layer_rgb
        homography_matrix,
        (reference_frame_cols, reference_frame_rows)
    )

    # Warp a white mask to use as the place to
    # put the augmented layer's pixels onto
    warp_mask = cv2.warpPerspective(
        object_mask, # or augmented_layer_mask
        homography_matrix,
        (reference_frame_cols, reference_frame_rows)
    )
    warp_mask_white_pixels = np.where(warp_mask == 255)

    # Superimpose the augmented reality layer over the current frame
    final_frame = cv2.cvtColor(current_frame, cv2.COLOR_BGR2RGB)
    final_frame[warp_mask_white_pixels] =
        warped_augmented_frame[warp_mask_white_pixels]
```

Finally we store the augmented frame inside a buffer so that later on it will be easier to retrieve it.

```
    F2F_frames_buffer.append(final_frame)
```

This process is repeated for every frame of the video sequence.

```
F2F_frames_buffer = []
process_video(video_path, process_video_F2F)
```

3.3 Frame to Frame (Refined) solution

This last augmentation system mixes the two previous approaches: it uses successive frames to keep track of small movements of the scene and periodically correct any drift by matching the keypoints of the reference frame with the ones of the current frame. In order to detect the keypoints and compute their descriptors the "run_SIFT" utility function is used. Diffently from the previous F2F approach, for frame-to-frame correspondences the optical flow is tracked with the Lukas-Kanade method instead of finding the keypoints with SIFT. Like in the F2F solution, in order to warp the augmented layer onto the current frame we need to "keep track" of all the homographies found so we use a "base" homography matrix that will be "updated" at every frame.

First we need to set the homography matrix to I:

```python
# 3x3 identity matrix
homography_matrix = np.identity(3, dtype = np.float32)
```

Since the first frame of the video doesn't have a previous frame to compare it to, we use SIFT for the keypoints detection.

```python
# Initial keypoints to keep track of
kp_query, des_query = run_SIFT(reference_frame_grayscale)
old_pts = np.float32([kp.pt for kp in kp_query]).reshape(-1,1,2)
```

Once done that, we can proceed to the optical flow estimation of the next frame.

```python
def process_video_F2FLK(current_frame, previous_frame):
    # First frame processing (skipped)
    # ...

    # Create the grayscale versions of the frames
    previous_frame_grayscale = # ...
    current_frame_grayscale = # ...

    # Use the LK method for 4 frames, and SIFT on the 5th one
    if frame_num % 5 != 0:
        # Calculate optical flow via Lucas-Kanade method
        new_pts, status, err = cv2.calcOpticalFlowPyrLK(
            previous_frame_grayscale,
            current_frame_grayscale,
            old_pts,
            None,
            **lk_params
        )
```

Only points of which the optical flow was tracked successfully should be considered.

```python
        good_new = new_pts[status == 1]
        good_old = old_pts[status == 1]
```

If at least 4 points could be tracked, we estimate an homography between the previous frame and the current frame and update the total homography_matrix like in the F2F solution.

```
        # Check if we found the minimum number of
        # matches for an homography
        if len(good_new) < 4:
            return

        # Calculating homography based on correspondences
        M, mask = cv2.findHomography(
            good_old,
            good_new,
            cv2.RANSAC,
            5.0
        )

        # Update the total homography matrix
        homography_matrix = np.dot(homography_matrix, M)
```

Every N frames, the matrix M is calculated via the "run_SIFT" method of the F2R solution instead of using the optical flow informations:

```
    # Every 5 frames run SIFT
    if frame_num % 5 == 0:
        kp_train, des_train = run_SIFT(current_frame_grayscale)
```

Like in the F2R solution we try to match the keypoints of the reference frame with the just-found ones.

```
        # Initializing the matching algorithm
        index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
        search_params = dict(checks = 50)
        flann = cv2.FlannBasedMatcher(index_params, search_params)

        # Matching the descriptors
        matches = flann.knnMatch(des_query, des_train, k = 2)
```

As in the previous solutions, we use Lowe's ratio with T = 0.7.

```
        good = [] # list of valid matches
        for m,n in matches:
            if m.distance < 0.7 * n.distance:
                good.append(m)
```

If 4 or more points are found an homography between the reference frame and the current frame is estimated.

```python
        # Check if SIFT found the minimum number of matches
        if len(good) < MIN_MATCH_COUNT: # 4 or more
            return

        # Extraction of good reference/current frames' keypoints
        src_pts = #...
        dst_pts = #...
        new_pts = dst_pts

        # Calculation of the homography based on correspondences
        M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
```

At this point two approaches are possible: directly setting the homography_matrix to M or taking an average (perhaps even weighted) between the two and use that value. This solution follows the second way of calculating the homography matrix because it leads to smoother transitions from an augmented layer's position to the next one.

```python
        homography_matrix = np.mean(
            np.array([homography_matrix, M]),
            axis = 0
        )
```

In this solution we need keep track of the position of the points yielded from the optical flow/SIFT calculations too:

```python
    old_pts = new_pts
```

From now on the rest of the solution is exactly as the F2F one.

```python
    # Warping the augmented layer
    warped_augmented_frame = cv2.warpPerspective(
        target_image_augmented, # or augmented_layer_rgb
        homography_matrix,
        (reference_frame_cols, reference_frame_rows)
    )

    # Warp a white mask to use as the place to
    # put the augmented layer's pixels onto
    warp_mask = cv2.warpPerspective(
        object_mask, # or augmented_layer_mask
        homography_matrix,
        (reference_frame_cols, reference_frame_rows)
    )
    warp_mask_white_pixels = np.where(warp_mask == 255)

    # Superimpose the augmented reality layer over the current frame
    final_frame = cv2.cvtColor(current_frame, cv2.COLOR_BGR2RGB)
    final_frame[warp_mask_white_pixels] =
        warped_augmented_frame[warp_mask_white_pixels]

    F2FLK_frames_buffer.append(final_frame)
```

This process is repeated for every frame of the video sequence.

```python
F2FLK_frames_buffer = []
process_video(video_path, process_video_F2FLK)
```

Chapter 4

Saving the augmented videos

Since the requested output data is "a video sequence, referred to here as Augmented Multiple View.avi containing realistic superimposition of the augmented reality layer onto the input video sequence" we now need to save the augmented frames we've produced onto the local storage.

The first step is to create the output directory for our file if it hasn't been created yet:

```
dir = os.path.abspath(...) #output folder path

# Check if the output directory exists, if not: create it
if not os.path.exists(dir) or not os.path.isdir(dir):
    os.makedirs(dir)
```

Now we need a python abstraction of the filesystem in order to create and modify files inside the just-created directory. For this purpose the VideoWriter API is used.

```
filepath = os.path.abspath(...) #dir + file name

video_writer = cv2.VideoWriter(
    filepath,
    cv2.VideoWriter_fourcc(*"DIVX"),                 #MPEG-4 codec
    15,                                              #fps
    (reference_frame_cols, reference_frame_rows) #width x height
)
```

Once everything has been set up, the frames can be stored into the newly created video file.

```
for frame in frames_buffer
    # In the chapter 2.1 we've noticed the video frames' colorspace
    # was BGR, so we perform the needed conversions
    frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    # Save the RGB video frame locally
    video_writer.write(),
```

Finally we release the OS resources.

```
video_writer.release()
```

Chapter 5

Results and Conclusions

After having processed the videos it's clear that, result wise, the original F2F is the worst of three proposed solutions because of the drifting of the ar-layer with respect to the video frames while the F2R approach produced a more consistent positioning of the augmented reality layer even if there's a lot of "shakiness" that the "improved" F2F solution has mitigated. The drift of the F2F solution is probably due to some SIFT related keypoints' detection/matching error which was propagated by the dot product between the homography matrices. Perhaps a lower Lowe's threshold could have improved this behavior at the cost of losing a few more "good" matches. Surely it can't be due to the matching process itself because the k-d-tree is an "exhaustive" search method that does not introduce errors. Even though it's possible, it's unlikely that the error is generated by noise affecting the detection step since SIFT internally uses the Difference of Gaussians detection method which itself alleviates noise related problems by blurring the images with increasingly high values of sigma. It is even possible that, since the DoG is just an approximation of the LoG, there were maxima detection mistakes in the augmenting process.

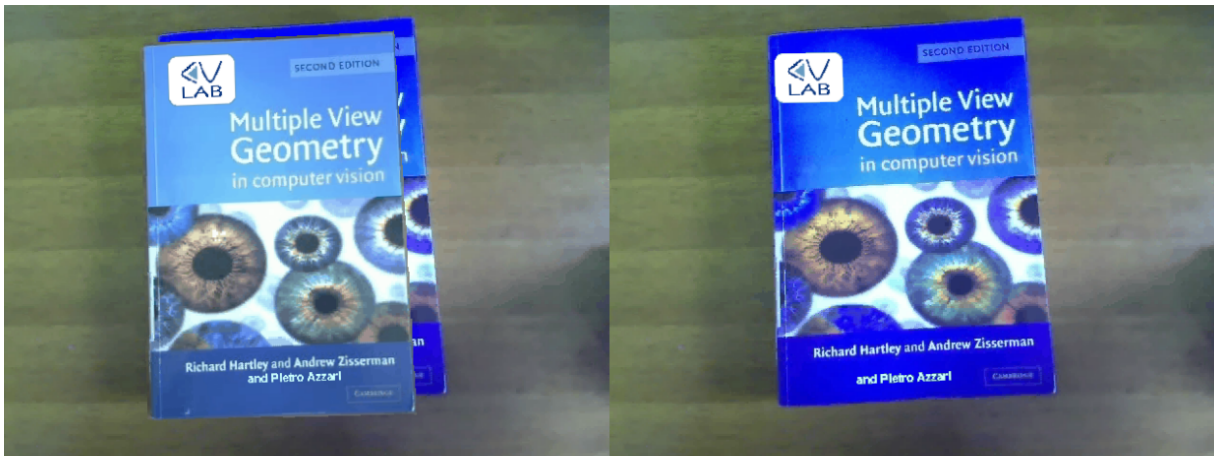


Figure 5.1: Noticeable drift of the ar-layer (both versions) - F2F solution

The F2R solution completly avoids the drift problem by not keeping any "memory" of the previous operations. As a side-effect, sometimes the F2R method may produce two consecutive frames with very different homographies with respect to the reference frame and so the final augmented frames can look quite different from each other even if they're one after another (which explains the "shakiness" of the book cover).

The improved F2F solution has two main advantages compared to the other solutions: speed and consistency. The first one is purely due to the Lukas-Kanade optical flow detection method being computationally less expensive than SIFT, which means that the original F2F solution could have benefited from this different approach too. The second advantage is instead due to the combination of the F2R and F2F techniques, infact without this trick we would have experienced the same drift present in the original F2F solution since we still use the dot product to "update" the homography matrix (and to propagate the error). The drift is (partially) corrected by the mean between the F2R prediction and the F2F prediction which means that the transition between two augmented frames is smoother than the F2R method without the visual inconsistency of the original F2F solution.

This paper, the jupyter notebook code, the original data and all the augmented augmented videos are available at this GitHub repositoy.

Bibliography

[1] https://en.wikipedia.org/wiki/Random_sample_consensus

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

[3] https://en.wikipedia.org/wiki/Optical_flow

[4] https://numpy.org/doc/stable/index.html

[5] https://docs.opencv.org/3.4/dc/de2/classcv_1_1FlannBasedMatcher.html

[6] https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html