Computer Vision and Image Processing M
Augmented reality project

Lorenzo Quaranta

January 10, 2024

Indice

1 Abstract

The goal of this project is to develop an augmented reality system based on a provided video sequence depicting a well-known book "Multiple View Geometry in Computer Vision", by Richard Hartley and Andrew Zissermann, captured by a moving camera.

The system aims to overlay an augmented reality (AR) layer, incorporating a logo in the top-left corner of the book and introducing a third author name below the existing two. The video captures the camera's gradual translation and rotation around the book, culminating in a sudden change in brightness. As per specifications, the AR layer should be superimposed as realistically as possible, therefore colour artefacts should be eliminated. In addition to the initial video, we are supplied with the first frame of the video as reference, along with the layer image and mask.

This paper presents 2 alternative solutions, a Frame to Reference and a Frame to Frame approach. A few variants are also explored in an attempt to enhance the final result. All solutions revolve around identifying local features and calculating a homography in order to warp the existing AR layer for each frame.

2 Images elaboration

The initial phase involves importing essential libraries and initializing variables containing paths to images, input and output, and constants used throughout the Jupyter notebook.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Constants
MIN_MATCH_COUNT = 10
FLANN_INDEX_KDTREE = 1
FLANN_INDEX_LSH = 6
SAVED_VIDEO_FPS = 15
F2R_FC = 10 # frequency for mixed F2R and F2F approach
RANSAC_REPROJ_THRESH = 5.0 # ransac reprojection threshold
LOWE_RATIO = 0.7 # lowe ratio for flann matcher
```

Utility functions for image and video processing are also defined in this phase.

2.1 Reading input images

The function "cv2.imread" is used to read all the images, which encodes color in BGR format, so a conversion is needed. Since a grayscale version is required, each color version is stored in a variable. The first image needed is the reference image, from which we can get the video resolution:

```
reference_frame_bgr = cv2.imread(reference_frame_path)
reference_frame_rgb = cv2.cvtColor(
    reference_frame_bgr,
    cv2.COLOR_BGR2RGB)
reference_frame_grayscale = cv2.cvtColor(
    reference_frame_bgr,
    cv2.COLOR_BGR2GRAY)
reference_frame_rows, reference_frame_cols = #480x640
reference_frame_rgb.shape[0], reference_frame_rgb.shape[1]
```

from this we can crop the augmented layer image and mask, which are provided with a different shape.

```
augmented_layer_rgb = augmented_layer_rgb[
    :reference_frame_rows,
    :reference_frame_cols
] #crop from 480x1525 to 480x640
```

2.2 AR layer improvement

The augmented layer contains the entire authors section, including a portion of the colored book. This could lead to the layer becoming visible during changes in video brightness. To prevent color artifacts during superimposition, a straightforward solution is to extract and retain only the text of the third author, which has a unique colour in the augmented layer image:

```
# Extraction of third author from binary mask
third_author_mask = np.zeros(
    (reference_frame_rows, reference_frame_cols, 3),
    dtype = np.uint8)
third_author_color = [201, 255, 255]
third_author_pixels = np.where(
    (augmented_layer_rgb == third_author_color).all(axis = 2))
```

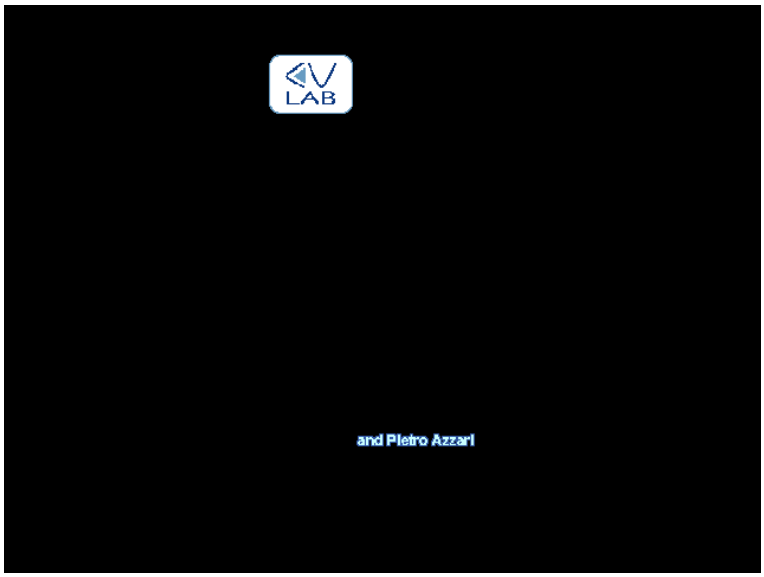
With only this operation the text loses visibility while warping the image. This can be solved applying a dilation on the mask:

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

Finally, the AR layer image is updated:

```
# first 300 rows includes logo, so we don't  update them
augmented_layer_mask[300:,:]=third_author_mask_dilated[300:, :]
augmented_layer_rgb[np.where(augmented_layer_mask == 0)] = 0
```

The augmented layer now appears as follows:



3 Proposed solutions

As outlined in the project specifications, two primary solution are considered to address the problem, a Frame to Reference(F2R) and Frame to Frame(F2F) approach. The core idea is to identify a minimum of four keypoints in each frame using a feature detector, establishing matches between frames, and subsequently calculating a homography. This homography is then applied to warp the augmented reality layer to the correct position. The resulting augmented frames are accumulated in an array, which is later saved as a video output file.

3.1 Frame to Reference approach

This system involves comparing a reference frame, in this case the first one, to each one of the video sequence. Given that the augmented layer image is constructed from the initial frame, the task simplifies to correctly warping this image using the homography calculated between the reference frame and subsequent frames. First of all, we need to initialize a feature detector and then compute keypoints and descriptors for the frames. A utility function is defined using the SIFT cv2 implementation (2 different SIFT initialization method for different opencv versions):

```
# sift = cv2.xfeatures2d.SIFT_create() # OpenCV 3.4.2
sift = cv2.SIFT.create() # OpenCV 4.5.1
def run_SIFT(image_grayscale, mask = None):
    # Detecting keypoints in the image
    keypoints = sift.detect(image_grayscale)
    # Computing the descriptors for each keypoint
    return sift.compute(image_grayscale, keypoints)
```

keypoints and descriptors are then calculated for the reference frame, which will be used throughout the video processing:

```
kp_query, des_query = run_SIFT(reference_frame_grayscale)
```

For the descriptors matching, the FLANN (Fast Library for Approximate Nearest Neighbors) algorithm with kd-tree parameters is employed:

```
# 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)
```

A "process_frame_F2R" function is then defined to handle feature matching and warping for each frame. Matches are filtered using Lowe's ratio test with a threshold (T = 0.7):

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

good = []
```

```
for m,n in matches:
    if m.distance < LOWE_RATIO * n.distance:
        good.append(m)
```

If atleast four good matches are found, the homography is calculated using the RANSAC method to filter out potential outliers:

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

The obtained homography is then used to calculated the warped frame:

```
# Warping the augmented version of the book
warped_augmented_frame = cv2.warpPerspective(
    augmented_layer_rgb,
    M,
    (reference_frame_cols, reference_frame_rows)
)

warp_mask = cv2.warpPerspective(
    augmented_layer_mask,
    M,
    (reference_frame_cols, reference_frame_rows)
)
```

Finally, the warped frame is returned and added to the augmented frame video array.

To process the entire video, each frame is read and the "process_frame_F2R method" is called, adding the resulting augmented frame to the "F2R_frames_buffer":

```
video = cv2.VideoCapture(video_path)
F2R_frames_buffer = []

while(video.isOpened()):
    frame_read_correctly, current_frame = video.read()

    if current_frame is not None:
        augmented_frame = process_frame_F2R(current_frame)
        if augmented_frame is not None:
            F2R_frames_buffer.append(augmented_frame)
```

3.2 Frame to Frame approach

In this case the system compares 2 consecutives frames to calculate the homography. To achieve this, it is necessary to maintain and update the previous homographies. The initial homography matrix is initialized as the identity matrix:

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

Similar to the Frame-to-Reference (F2R) approach, the SIFT method is employed for feature detection and matching. The frame processing function now takes two parameters: the previous frame and the current frame:

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

The same matching and homography calculation process as in the F2R approach is followed. However, this time, the homography matrix is updated by taking the dot product:

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

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

The augmented frame is then returned, and the video processing slightly differs by maintaining track of the previous frame:

```
# processing video
video = cv2.VideoCapture(video_path)
F2F_frames_buffer = []

previous_frame = None
while(video.isOpened()):
    frame_read_correctly, current_frame = video.read()

    if previous_frame is None:
        previous_frame = current_frame

    if current_frame is not None:
        augmented_frame = process_frame_F2F(
            current_frame,
            previous_frame
        )
        if augmented_frame is not None:
            F2F_frames_buffer.append(augmented_frame)
        previous_frame = current_frame
```

3.3 Variant 1: mixed solution

An improvement to consider is the utilization of both F2R and F2F methods at a set frequency, so that the downsides of both methods are mitigated. The "F2R_FC"

constant determines how often the Frame-to-Reference method is used. Modifications in "the process_frame" method are as follows:

```
if (frame_num % F2R_FC) == 0:
    kp_query, des_query = kp_ref, des_ref
else:
    kp_query, des_query = run_SIFT(previous_frame_grayscale)
```

The homography matrix is updated differently for the two cases:

```
# update the homography matrix
if (frame_num % F2R_FC) == 0:
    homography_matrix = np.mean([homography_matrix, M], axis = 0)
else:
    homography_matrix = np.dot(homography_matrix, M)
```

3.4 Variant 2: Different feature detectors

An alternative solution involves using different feature detectors, such as ORB and AKAZE. Utility methods for these detectors are created:

```
orb = cv2.ORB_create()
def run_ORB(image):
    global orb

    # Detect keypoints and compute descriptors
    keypoints, descriptors = orb.detectAndCompute(image, None)

    return keypoints, descriptors
```

```
akaze = cv2.AKAZE_create()

def run_AKAZE(image):
    kp, des = akaze.detectAndCompute(image, None)
    return kp, des
```

The process is similar. In this case a different FLANN algorithm is needed. The FLANN_INDEX_LSH (Locality-Sensitive Hashing) is used for both detectors.

```
# Initializing the matching algorithm
index_params = dict(
    algorithm = FLANN_INDEX_LSH,
    table_number = 6,
    key_size = 12,
    multi_probe_level = 2
)
```

```
search_params = dict(checks = 50)
flann = cv2.FlannBasedMatcher(index_params, search_params)
```

The "process_frame" method remains unchanged, we just call the new methods defined instead of SIFT:

```
# Run ORB on the current frame
kp_train, des_train = run_ORB(current_frame_grayscale)
```

```
# Run AKAZE on the current frame
kp_train, des_train = run_AKAZE(current_frame_grayscale)
```

This solution was tested with a Frame to Reference approach.

4 Results and Conclusion

Between the 2 primary solution explored in this project, the Frame to Reference (F2R) approach clearly gives better result. The Frame to Frame (F2F) approach encounters a significative issue: the augmented layer slowly drifts away from its correct position. This is probably caused by some SIFT related keypoints' detection or matching error which gets propagated by the dot product between homography matrices. Additionally, the F2F method is notably slower due to the repeated computation of SIFT keypoints and descriptors for each pair of frames.

While the F2R method eliminates the drift issue by recalculating the homography independently for each frame, it introduces a new issue – perceptible shakiness in the augmented layer. This instability arises from the lack of memory of previous homographies, leading to variations in the calculated homography even for consecutive and similar frames.

Figure 1: F2R (left) and F2F (right) comparison

The first proposed variant aims to combine the strengths of both methods. By employing the F2F approach and resetting the accumulated homography with the F2R method every "F2R_FC" frames, an attempt is made to limit drifting and mitigate shakiness. While this variant shows some improvement, it doesn't completely eliminate shakiness, especially when the layer has drifted significantly during the F2F phase.

Other attempts to improve the result all involved tweaking parameters and methods. Modifying the RANSAC reprojection threshold, Lowe's ratio test threshold, and experimenting with different feature detectors, such as ORB and AKAZE, were explored. The most promising outcome was observed when utilizing AKAZE as a feature detector in the F2R approach, providing a stable result in a relatively shorter computational time compared to SIFT. With the aid of a utility function, several video comparisons were conducted and saved in a dedicated 'Comparisons' folder within the project directory.

In conclusion, an acceptable solution was identified with a Frame to Reference approach, offering room for fine-tuning through parameter adjustments and alternative detection methods. Further in-depth testing could potentially yield even better results than those presented in this paper.