AUGMENTED REALITY WITH PLANAR HOMOGRAPHIES

ALIA ESSAM 7810
AYA RAGAA 7793

CODE ANALYSIS

```
import cv2
import math
import numpy as np
import matplotlib.pyplot as plt
import os
import tempfile
```

The code imports libraries for image processing, numerical computations, data visualization, and file management.

```
def get_correspondences(frame1, book_img, num_correspondences=50):
    sift = cv2.SIFT_create()
    kp1, des1 = sift.detectAndCompute(book_img, None)
    kp2, des2 = sift.detectAndCompute(frame1, None)

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

    good_matches = []
    for m, n in matches:
        if m.distance < 0.25 * n.distance:
            good_matches.append(m)

    good_matches = good_matches[:num_correspondences]

    # Save the correspondences
    pts1 = np.float32([kp1[m.queryIdx].pt for m in
good_matches]).reshape(-1, 1, 2)
    pts2 = np.float32([kp2[m.trainIdx].pt for m in
good_matches]).reshape(-1, 1, 2)

    return pts1, pts2
```

The function get_correspondences identifies and returns matching keypoints between a reference image (book_img) and a frame (frame1) using SIFT for feature detection and a brute-force matcher. It filters high-quality matches using Lowe's ratio test and limits the results to a specified number of correspondences (num_correspondences). The output is two arrays of matched points from both images.

```
def compute_homography(pts1, pts2):
    A = []
    for i in range(len(pts1)):
        x, y = pts1[i][0], pts1[i][1]
        u, v = pts2[i][0], pts2[i][1]
        A.append([-x, -y, -1, 0, 0, 0, u*x, u*y, u])
        A.append([0, 0, 0, -x, -y, -1, v*x, v*y, v])

    A = np.asarray(A)
    U, S, Vh = np.linalg.svd(A)
    H = Vh[-1, :].reshape(3, 3)
    # Normalize H (optional)
    H /= H[2, 2]
    return H
```

The function compute_homography calculates a homography matrix from point correspondences (pts1, pts2) using Singular Value Decomposition (SVD). It constructs a system of linear equations, solves it via SVD, and returns the 3x3 homography matrix H, which can be used for tasks like image transformation. The matrix is optionally normalized by dividing by its last element.

```
def map_points_with_homography(H, points):
    points_homogeneous = np.hstack((points, np.ones((len(points), 1))))
    mapped_points_homogeneous = np.dot(H, points_homogeneous.T)
    # Normalize homogeneous coordinates
    mapped_points = mapped_points_homogeneous[:2, :] /
mapped_points_homogeneous[2, :].reshape(1, -1)
    return mapped_points.T
```

The function map_points_with_homography applies a homography matrix H to map a set of points. It converts the points to homogeneous coordinates, applies the homography transformation, and then normalizes the result back to Cartesian coordinates. The function returns the transformed points.

```
def RANSAC(pts1, pts2, num_iterations=1000, min_set_size=4,
inlier_threshold=.5, min_inliers=45):
    best_H = None
    max_inliers = 0
    num_correspondences = len(pts1)
    np.random.seed(42)
    for i in range(num_iterations):
        random_indices = np.random.choice(num_correspondences,
size=min_set_size, replace=False)
        sampled_pts1 = pts1[random_indices]
        sampled_pts2 = pts2[random_indices]

        initial_H = compute_homography(sampled_pts1, sampled_pts2)
        transformed_points = map_points_with_homography(initial_H, pts1)

        errors = np.sqrt(np.sum((transformed_points - pts2)**2, axis=1))
        inliers = np.sum(errors < inlier_threshold)

        if inliers > max_inliers:
            max_inliers = inliers
            best_H = initial_H

        if max_inliers > min_inliers:
            break

    return best_H
```

The RANSAC function finds the best homography matrix by iteratively selecting random point subsets, computing the homography, and evaluating inliers based on a distance threshold. It aims to maximize the number of inliers (points that fit the model well) and returns the homography with the most inliers, stopping early if enough inliers are found.

```
def crop_ar_video_frame(video_frame, book_corners):

    # Calculate the target width and height from book corners
    book_width = int(book_corners[1][0] - book_corners[0][0])
    book_height = int(book_corners[3][1] - book_corners[0][1])

    # Resize the AR video frame to exactly match the book region
    resized_frame = cv2.resize(video_frame, (book_width, book_height),
interpolation=cv2.INTER_AREA)

    return resized_frame
```

The crop_ar_video_frame function resizes an AR video frame to fit the region defined by the book corners. It calculates the width and height of the book from the corner points, then resizes the video frame to match this size. The resized frame is returned.

```
def overlay_frames(frame1, frame2, H, book_corners):
    book_coordinates_video = map_points_with_homography(H, book_corners)

    mask = np.zeros_like(frame1, dtype=np.uint8)
    cv2.fillPoly(mask, [np.int32(book_coordinates_video)], (255, 255,
255))

    inverted_mask = cv2.bitwise_not(mask)

    frame1_blacked = cv2.bitwise_and(frame1, inverted_mask)

    overlay_frame = cv2.warpPerspective(frame2, H, (frame1.shape[1],
frame1.shape[0]))

    result = cv2.add(frame1_blacked, overlay_frame)

    return result
```

The overlay_frames function overlays an AR video frame (frame2) onto a book image (frame1) using a homography matrix (H). It maps the book's corners onto the AR video frame, creates a mask to isolate the book region, and then combines the book frame with the transformed AR video frame, resulting in an overlay of the two frames.

```
def process_video(video1_path, video2_path, book_img_path):

    # Open the video files
    video1 = cv2.VideoCapture(video1_path)
    video2 = cv2.VideoCapture(video2_path)

    # Load the book image
    book_img = cv2.imread(book_img_path)

    if book_img is None:
        raise ValueError("Image not found at path: " + book_img_path)

    # Set up the video writer
    width = int(video1.get(3))
```

```
    height = int(video1.get(4))
    fourcc = cv2.VideoWriter_fourcc(*'XVID')

    # Define the output path in the same directory
    output_path = 'output_video_edited.avi'

    output_video = cv2.VideoWriter(output_path, fourcc, 30.0, (width,
height))

    # Define the book corners
    book_corners = np.array([[0, 0],
                             [book_img.shape[1] - 1, 0],
                             [book_img.shape[1] - 1, (book_img.shape[0] -
1)],
                             [0, (book_img.shape[0] - 1)]],
                            dtype=np.float32)

    # Main processing loop
    while True:
        ret1, frame1 = video1.read()
        ret2, frame2 = video2.read()

        if not ret1 or not ret2:
            break

        # Get correspondences for each frame
        pts_book, pts_video = get_correspondences(frame1, book_img)

        # Calculate homography matrix
        H = RANSAC(np.squeeze(pts_book), np.squeeze(pts_video))

        # Crop the video frame centered on the book
        cropped_video_frame = crop_ar_video_frame(frame2, book_corners)

        # Overlay frames and write to the output video
        result_frame = overlay_frames(frame1, cropped_video_frame, H,
book_corners)
        output_video.write(result_frame)

    # Release video captures and writer
    video1.release()
    video2.release()
    output_video.release()

    return output_path
```

The process_video function overlays an AR video onto a book image by processing two input videos (video1_path and video2_path) and a book image (book_img_path). It reads the videos, computes point correspondences, calculates the homography matrix, crops the AR video frame to match the book region, and overlays the frames. The result is written to an output video file (output_video_edited.avi). The function returns the path to the output video.

```
video1_path = "book.mov"
video2_path = "ar_source_edited.mov"
book_img_path = "cv_cover.jpg"

output_path = process_video(video1_path, video2_path, book_img_path)

print(f"Output video saved to: {output_path}")
```

The code calls the process_video function with paths to two video files (book.mov and ar_source_edited.mov) and a book image (cv_cover.jpg). It processes the videos, overlays the AR video on the book image, and saves the result to an output video file. The output file path is printed after processing.