비주얼 오도메트리 중간고사 대체 과제 보고서

120230241 이도훈

1. 과제 문제

   1) 두 이미지를 입력 받아 ORB keypoint와 descriptor를 계산한다.

   2) Hamming distance를 사용해 bruteforce matching을 통해 두 이미지의 keypoint의 일치 여부를 비교한다.

   3) RANSAC 알고리즘을 통해 homography matrix를 구하고 이를 통해 두 이미지의 keypoint가 일치하는 것을 기준으로 두 이미지를 합친다(warp).

   4) 생성된 파노라마 이미지를 출력한다.

2. 코드 리뷰

   1)

```
img1 = cv2.imread('img5_1.jpg')
img2 = cv2.imread('img5_2.jpg')

img1_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
img2_gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
```

   이미지 2개를 입력 받은 뒤 사용에 용이하게 gray scale로 변환한다.

   2)

```
orb = cv2.ORB_create(nfeatures=2000)
keypoints1, descriptors1 = orb.detectAndCompute(img1, None)
keypoints2, descriptors2 = orb.detectAndCompute(img2, None)
```

   ORB keypoint와 descriptor 검출기를 cv2.ORB_create를 통해 생성한다. 이때 검출하는 최대 keypoint 개수는 2000개로 제한한다.

   이후 입력 받은 2개의 이미지에 대해 keypoint와 descriptor를 검출하여 각각의 변수에 저장한다.

3)

```
bf = cv2.BFMatcher_create(cv2.NORM_HAMMING)
matches = bf.knnMatch(descriptors1, descriptors2,k=2)

all_matches = []
for m, n in matches:
    all_matches.append(m)
```

두 keypoint 사이의 hamming distance를 토대로 bruteforce하게 matching을 진행하도록 cv2.BFMatcher_create를 사용한다. 이후 knn 방식을 통해 두 keypoint 사이의 matching을 진행한다. 이를 특정 배열에 저장한다.

4)

```
def draw_matches(img1, keypoints1, img2, keypoints2, matches):
    r, c = img1.shape[:2]
    r1, c1 = img2.shape[:2]

    output_img = np.zeros((max([r, r1]), c+c1, 3), dtype='uint8')
    output_img[:r, :c, :] = np.dstack([img1, img1, img1])
    output_img[:r1, c:c+c1, :] = np.dstack([img2, img2, img2])

    for match in matches:
        img1_idx = match.queryIdx
        img2_idx = match.trainIdx
        (x1, y1) = keypoints1[img1_idx].pt
        (x2, y2) = keypoints2[img2_idx].pt

        cv2.circle(output_img, (int(x1),int(y1)), 4, (0, 255, 255), 1)
        cv2.circle(output_img, (int(x2)+c,int(y2)), 4, (0, 255, 255), 1)

        cv2.line(output_img, (int(x1),int(y1)), (int(x2)+c,int(y2)), (0, 255, 255), 1)

    return output_img
```

```
img3 = draw_matches(img1_gray, keypoints1, img2_gray, keypoints2, all_matches[:50])
cv2.imwrite('img5_key.jpg', img3)
```

임의의 함수 draw_matches를 만들어 keypoint matching이 정상적으로 이루어졌는지 확인 가능하게 만들었다.

Output image를 위한 변수를 선언한 뒤 입력 받은 이미지 2개를 복사한다. 두 이미지의 keypoint들을 원으로 그리고 서로 일치하는 keypoint를 선으로 연결한 뒤 그려져 있는 이미지를 출력한다.

5)

```
good = []
for m, n in matches:
    if m.distance < 0.5 * n.distance:
        good.append(m)
```

두 이미지의 keypoint들을 이용하여 특정 threshold 이하의 "좋은" matching을 찾는다. keypoint들의 거리가 특정 threshold 이하일 경우 특정 변수에 저장한다.

6)

```
MIN_MATCH_COUNT = 10

if len(good) > MIN_MATCH_COUNT:
    src_pts = np.float32([keypoints1[m.queryIdx].pt for m in good])
    dst_pts = np.float32([keypoints2[m.trainIdx].pt for m in good])
    M = findHomographyRANSAC(src_pts, dst_pts, max_dist=4.0)

    if M is not None:
        result = warpImages(img2, img1, M)
        cv2.imwrite('img5_result.jpg', result)
    else:
        print("Not enough inliers to stitch images.")
else:
    print("Not enough good matches to stitch images.")
```

Stitching이 이루어지기 위한 "좋은" matching의 최소 조건을 10개로 정한 뒤 그 이상일 경우에만 stitching을 진행한다.

이후 두 이미지에서 "좋은" matching에 해당하는 keypoint들만 추출하여 배열로 만든다.

RANSAC을 이용하며 homography matrix를 직접 만든 함수로 생성한 뒤 이를 토대로 warping하여 파노라마 이미지를 생성한 뒤 이를 저장한다.

7)

```python
def findHomographyRANSAC(src_pts, dst_pts, max_dist=4.0, max_iterations=2000, confidence=0.99):
    best_H = None
    best_inliers = 0
    src_pts = np.float32(src_pts)
    dst_pts = np.float32(dst_pts)
    num_points = src_pts.shape[0]

    for _ in range(max_iterations):
        random_indices = np.random.choice(num_points, 4, replace=False)
        random_src = src_pts[random_indices]
        random_dst = dst_pts[random_indices]

        A = []
        for i in range(4):
            x, y = random_src[i]
            u, v = random_dst[i]
            A.append([-x, -y, -1, 0, 0, 0, x * u, y * u, u])
            A.append([0, 0, 0, -x, -y, -1, x * v, y * v, v])
        A = np.array(A)
        U, S, Vt = np.linalg.svd(A)
        H = Vt[-1].reshape(3, 3)
        H /= H[2, 2]

        transformed_pts = np.dot(H, np.vstack((src_pts.T, np.ones(num_points))))
        transformed_pts = transformed_pts[:2] / transformed_pts[2]

        errors = np.sqrt(np.sum((transformed_pts - dst_pts.T) ** 2, axis=0))
        inliers = np.sum(errors < max_dist)

        if inliers > best_inliers:
            best_inliers = inliers
            best_H = H

        inlier_ratio = inliers / num_points
        if inlier_ratio >= confidence:
            break

    return best_H
```

최적의 homography와 최대 내부점 개수 저장을 위한 변수를 생성한 뒤 keypoint 배열을 적절한 데이터 타입으로 변환하고, 사용할 keypoint의 총 개수를 계산한다. 이후 4개의 랜덤한 keypoint matching을 선택하고 이를 통해 homography matrix를 계산한다. 이때 svd를 이용하여 homography의 해를 찾는다. 이후 keypoint를 변환하고 거리를 계산하여 오차가 특정 수치보다 작으면 내부점으로 정한다. 이 내부점의 개수가 이전 최대 내부점 개수보다 크면 해당 homography를 최적으로 정한다.

내부점의 비율이 confidence보다 크면 반복문을 정지하고 최적의 H를 반환한다.

8)

```python
def apply_homography(H, points):
    num_points = len(points)
    transformed_points = np.zeros((num_points, 2), dtype=np.float32)

    for i in range(num_points):
        x, y = points[i]
        z = H[2, 0] * x + H[2, 1] * y + H[2, 2]
        x_dst = (H[0, 0] * x + H[0, 1] * y + H[0, 2]) / z
        y_dst = (H[1, 0] * x + H[1, 1] * y + H[1, 2]) / z
        transformed_points[i] = [x_dst, y_dst]

    return transformed_points

def warpPerspectiveCustom(img, H, output_shape):
    rows, cols = img.shape[:2]
    output_img = np.zeros(output_shape, dtype=img.dtype)

    H_inv = np.linalg.inv(H)

    for y in range(output_shape[0]):
        for x in range(output_shape[1]):
            p_dst = np.array([x, y, 1])
            p_src = np.dot(H_inv, p_dst)
            p_src /= p_src[2]

            x_src, y_src = int(p_src[0]), int(p_src[1])

            if 0 <= x_src < cols and 0 <= y_src < rows:
                output_img[y, x] = img[y_src, x_src]

    return output_img
```

Apply_homography 함수는 주어진 좌표들에 homography 행렬을 적용시켜 새로운 좌표계로 변환하는 함수이다. 각 keypoint들에 대해 homography를 적용하고 정규화를 한 뒤 반환한다.

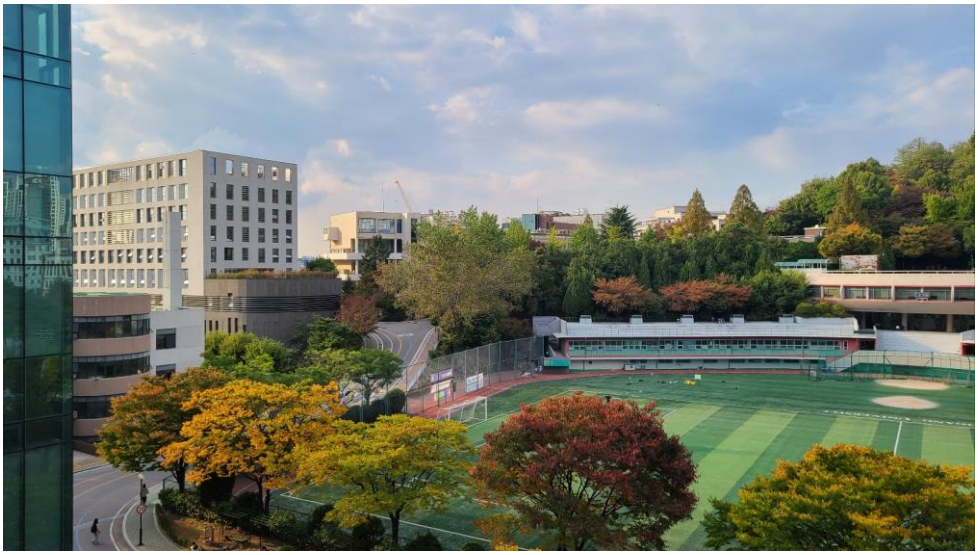
warpCustom 함수는 주어진 이미지에 homography 행렬을 적용시켜 이미지를 변환한다. Homography 행렬의 역행렬을 계산하여 원본 좌표를 계산 후 정규화하고 정수화하여 유효한 범위 내에 있는지 확인한다. 이후 원본 이미지의 해당 픽셀 값을 출력 이미지의 현재 위치에 할당한다. 이후 변환된 이미지를 반환한다.

```python
def warpImages(img1, img2, H):

    rows1, cols1 = img1.shape[:2]
    rows2, cols2 = img2.shape[:2]

    list_of_points_1 = np.float32([[0,0], [0, rows1],[cols1, rows1], [cols1, 0]])
    temp_points = np.float32([[0,0], [0,rows2], [cols2,rows2], [cols2,0]])

    # When we have established a homography we need to warp perspective
    # Change field of view
    list_of_points_2 = apply_homography(H, temp_points)

    list_of_points = np.concatenate((list_of_points_1,list_of_points_2), axis=0)

    [x_min, y_min] = np.int32(list_of_points.min(axis=0).ravel() - 0.5)
    [x_max, y_max] = np.int32(list_of_points.max(axis=0).ravel() + 0.5)

    translation_dist = [-x_min,-y_min]

    H_translation = np.array([[1, 0, translation_dist[0]], [0, 1, translation_dist[1]], [0, 0, 1]])

    output_shape = (y_max - y_min, x_max - x_min, 3)
    output_img = warpPerspectiveCustom(img2, np.dot(H_translation, H), output_shape)
    output_img[translation_dist[1]:rows1+translation_dist[1], translation_dist[0]:cols1+translation_dist[0]] = img1

    return output_img
```

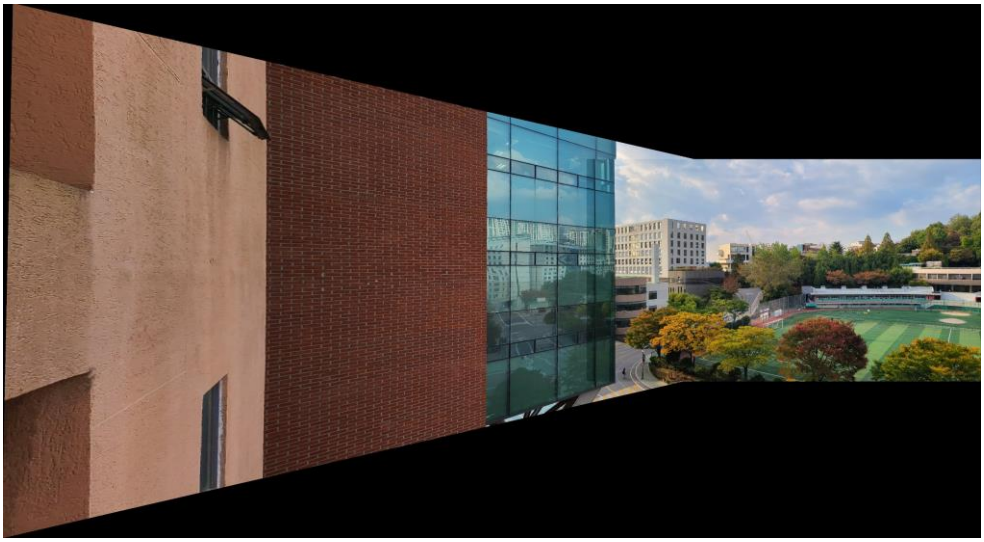
해당 함수는 위 2개의 함수를 사용하여 두 개의 이미지를 변환하여 올바르게 stitch 하는 것이다. Homography를 img2에 적용하여 변환된 좌표를 구하고 img1과 변환된 img2의 모든 모서리 좌표를 결합한다. 이후 출력될 최대 이미지 크기를 구하고 출력 이미지에서의 img1이 시작될 좌표를 구한다. 이후 이동행렬을 정의하여 img2가 올바른 위치로 이동이 가능하게 만든다. 최종 합성될 이미지의 모양을 정의하고 homography를 사용하여 img2를 변환하여 합성 이미지를 생성하고 반환한다.

3. 결과

1-1) 사용한 이미지 2개



1-2) 결과



2-1) 사용한 이미지 2개

2-2) 결과