비주얼 컴퓨팅

<Panorama Image>

담당 교수 : 김동준 교수님

학번 : 2021204010

학과 : 정보융합학부

이름 : 박현준

목차

1. 이미지 병합 과정

본 과제에서는 세 개의 이미지를 병합하여 파노라마를 생성하는 방법을 다룬다. 파노라마 이미지는 서로 겹치는 이미지들을 하나의 연속적인 이미지로 합성하는 기술이다. 이를 위해 이미지를 변환하고 결합하는 과정에서 호모그래피 변환, 백워핑(Back-Warping), 그리고 블렌딩 기법을 사용한다.

1-1. 특징점 검출 및 매칭, 호모그래피 계산

```python
# SIFT 키포인트 탐지 및 매칭
sift = cv.SIFT_create()
kp1, des1 = sift.detectAndCompute(img1, None)
kp2, des2 = sift.detectAndCompute(img2, None)
kp3, des3 = sift.detectAndCompute(img3, None)

# FLANN 매칭
flann = cv.FlannBasedMatcher({"algorithm": 1, "trees": 5}, {"checks": 50})
matches1 = flann.knnMatch(des1, des2, k=2)
matches3 = flann.knnMatch(des3, des2, k=2)

good_matches1 = [m for m, n in matches1 if m.distance / n.distance < 0.7]
src_pts1 = np.float32([kp1[m.queryIdx].pt for m in good_matches1]).reshape(-1, 1, 2)
dst_pts1 = np.float32([kp2[m.trainIdx].pt for m in good_matches1]).reshape(-1, 1, 2)
H12 = cv.findHomography(src_pts1, dst_pts1, cv.RANSAC, 5.0)[0]

good_matches3 = [m for m, n in matches3 if m.distance / n.distance < 0.7]
src_pts3 = np.float32([kp3[m.queryIdx].pt for m in good_matches3]).reshape(-1, 1, 2)
dst_pts3 = np.float32([kp2[m.trainIdx].pt for m in good_matches3]).reshape(-1, 1, 2)
H32 = cv.findHomography(src_pts3, dst_pts3, cv.RANSAC, 5.0)[0]
```

위 코드는 OpenCV의 SIFT 알고리즘을 사용하여 각 이미지에서 KeyPoints와 Descriptors를 추출한다. cv.SIFT_create()객체를 생성한 후, detectAndCompute()메서드를 통해 이미지에서 KeyPoints와 Descriptors를 동시에 검출한다. 추출된 Descriptors를 사용하여 FLANN 매칭 알고리즘을 적용하고, 이미지 간 특징점을 매칭한 후 가장 가까운 두 개의 매칭 결과를 선택한다. 이때, 거리 기준을 설정하여 두 매칭의 거리를 비교하고, 정확한 매칭 쌍만을 선별한다.

특징점 매칭 결과를 바탕으로, RANSAC 알고리즘을 이용한 cv.findHomography()를 통해 두 점 집합 간의 호모그래피 행렬 H를 계산한다. 호모그래피는 두 이미지 간의 기하학적 변환 관계를 정의하며, 이를 통해 한 이미지의 좌표를 다른 이미지의 좌표계로 변환할 수 있다.

H12: 이미지 1에서 이미지 2로의 호모그래피 변환 행렬을 구한다.

H32: 이미지 3에서 이미지 2로의 호모그래피 변환 행렬을 구한다

1-2. 캔버스 크기 및 오프셋 계산

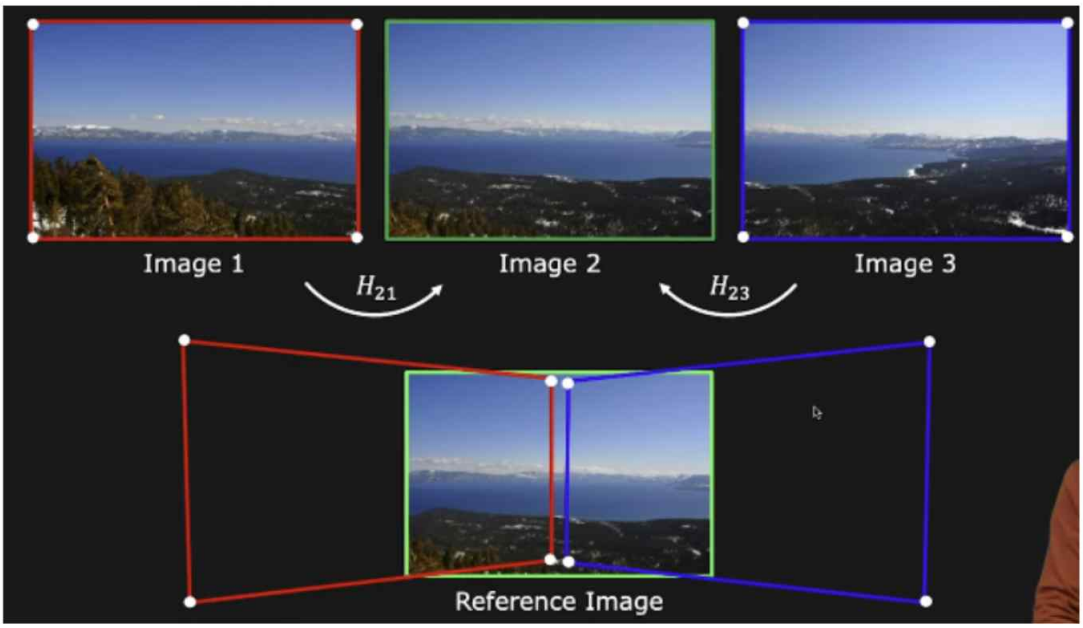
```python
def compute_canvas_size_and_offset(reference_img, img_list, homographies):
    ref_h, ref_w, _ = reference_img.shape
    corners = np.array([[0, 0], [ref_w, 0], [ref_w, ref_h], [0, ref_h]], dtype=np.float32).reshape
    (-1, 1, 2)

    all_corners = [corners]
    for img, H in zip(img_list, homographies):
        h, w = img.shape[:2]
        img_corners = np.array([[0, 0], [w, 0], [w, h], [0, h]], dtype=np.float32).reshape(-1, 1, 2)
        transformed_corners = cv.perspectiveTransform(img_corners, H)
        all_corners.append(transformed_corners)

    all_corners = np.concatenate(all_corners, axis=0)
    x_min, y_min = np.int32(all_corners.min(axis=0).ravel())
    x_max, y_max = np.int32(all_corners.max(axis=0).ravel())

    canvas_width = x_max - x_min
    canvas_height = y_max - y_min
    offset = (-x_min, -y_min)

    return canvas_width, canvas_height, offset
```

이미지들이 하나의 이미지로 합쳐질 때, 각 이미지가 변환된 후 이미지가 잘리지 않도록, 변환된 이미지들의 좌표를 기준으로 캔버스의 크기를 결정하여 모든 이미지가 캔버스 안에 포함되도록 한다. 또한, 각 이미지의 변환 후 위치를 캔버스 내에서 정확히 배치하기 위해 오프셋을 적용한다.

먼저 각 이미지의 코너 좌표들을 계산한 후, 변환된 코너를 cv.perspectiveTransform()을 사용하여 변환한다. 이후, 모든 이미지에서 변환된 코너들을 모은다. 이렇게 모은 코너들의 최솟값과 최대값을 구하여, 캔버스의 width와 height를 결정한다.

- [캔버스를 제대로 계산하지 못해 이미지가 잘렸던 결과]

1-3. Back-Warping

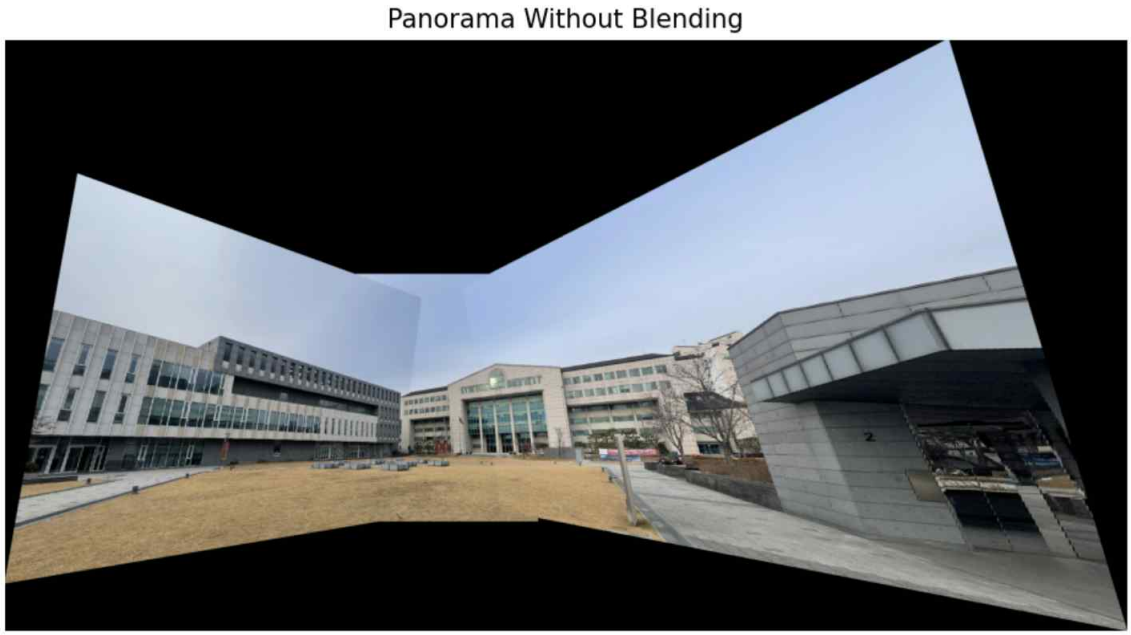


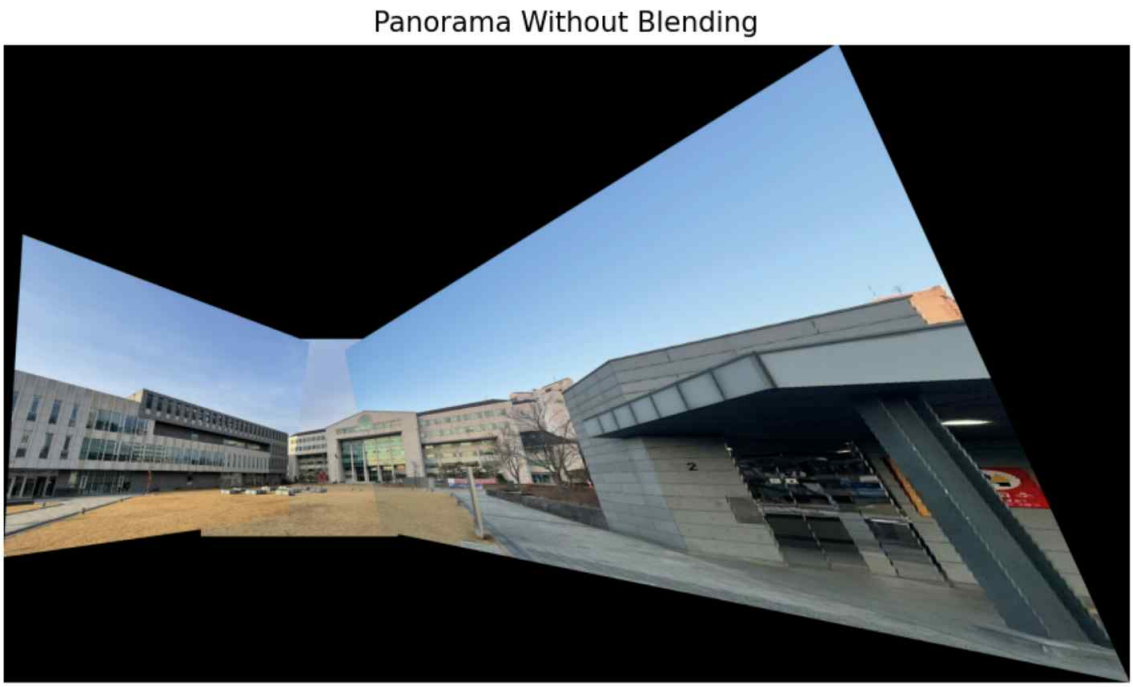

```python
def warp_image(canvas, img, H, offset):
    h, w = canvas.shape[:2]
    H_inv = np.linalg.inv(H)

    # 캔버스 좌표 생성
    y_coords, x_coords = np.indices((h, w))
    coords = np.stack([x_coords.ravel(), y_coords.ravel(), np.ones_like(x_coords).ravel()])
    coords[0] -= offset[0]
    coords[1] -= offset[1]

    # 역 매핑
    mapped_coords = H_inv @ coords
    mapped_coords /= mapped_coords[2]

    mapped_x = mapped_coords[0].reshape(h, w).astype(np.int32)
    mapped_y = mapped_coords[1].reshape(h, w).astype(np.int32)

    # 유효 범위 확인
    valid_idx = (0 <= mapped_x) & (mapped_x < img.shape[1]) & (0 <= mapped_y) & (mapped_y < img.shape
    [0])
    canvas[valid_idx] = img[mapped_y[valid_idx], mapped_x[valid_idx]]
```

이미지 정렬 및 병합 작업을 할 때, 백워핑(Back-Warping) 기법을 사용한다. 백워핑은 이미지를 변환할 때 사용되는 기법으로, 각 픽셀을 변환된 좌표에 올바르게 배치하는 과정이다. 이는 역변환을 통해 이루어지며, 변환된 이미지에서 구멍이 생기거나 이미지가 잘리는 문제를 피할 수 있다. 백워핑 기법은 warp_image함수에서 주로 사용된다. 이 함수는 변환된 이미지의 각 픽셀을 원본 이미지로 역으로 매핑하고, 그 위치에서 값을 가져와 캔버스에 채운다.

H_inv = np.linalg.inv(H)는 변환 행렬H의 역행렬을 구하여 역변환을 적용한다. coords는 변환된 이미지에서 각 픽셀에 대한 역변환된 좌표를 계산하고, mapped_coords는 그 좌표를 통해 원본 이

미지에서 해당 위치를 찾는다. 이후, canvas[valid_idx]는 변환된 좌표가 유효한 범위 내에 있을 경우, 원본 이미지에서 해당 위치의 값을 가져와 캔버스에 기록한다.





[겹치는 부분 블렌딩을 적용하기 전 파노라마 사진 결과]

2. 이미지의 겹치는 부분 블렌딩

겹치는 부분의 블렌딩 과정은 크게 겹치는 영역의 계산, 가중치 마스크 생성, 가중치 마스크 적용, 그리고 최종적은 블렌딩 처리로 나눠진다.

2-1. 겹치는 영역의 마스크 생성

```
def compute_overlap_mask(canvas_shape, img, H, offset):
    h, w = img.shape[:2]
    corners = np.array([[0, 0], [w, 0], [w, h], [0, h]], dtype=np.float32).reshape(-1, 1, 2)
    transformed_corners = cv.perspectiveTransform(corners, H) + offset

    mask = np.zeros(canvas_shape[:2], dtype=np.uint8)
    cv.fillConvexPoly(mask, np.int32(transformed_corners), 255)
    return mask
```

```
# 겹치는 마스크 계산
mask1 = compute_overlap_mask((canvas_h, canvas_w), img1, H12, offset)
mask2 = compute_overlap_mask((canvas_h, canvas_w), img2, np.eye(3), offset)
mask3 = compute_overlap_mask((canvas_h, canvas_w), img3, H32, offset)
```

compute_overlap_mask함수는 이미지가 변환되어 캔버스에 배치될 때, 해당 이미지의 겹치는 영역을 마스크 형태로 생성하는 역할을 한다. 함수는 이미지의 네 모서리를 정의하여, 이를 변환 행렬H를 통해 새로운 위치로 변환한다. 변환된 이미지가 캔버스에서 어느 위치에 놓일지 결정하기 위해 오프셋을 추가하고, 변환된 모서리들을 기준으로 다각형을 생성한다. 마지막으로, 해당 영역을 255로 채운 마스크를 생성하여 이미지의 겹치는 영역을 나타낸다.

```
def compute_overlap_region(mask1, mask2):
    return cv.bitwise_and(mask1, mask2)
```

```
# 겹치는 영역 계산
overlap_mask1_2 = compute_overlap_region(mask1, mask2)
overlap_mask2_3 = compute_overlap_region(mask2, mask3)
```

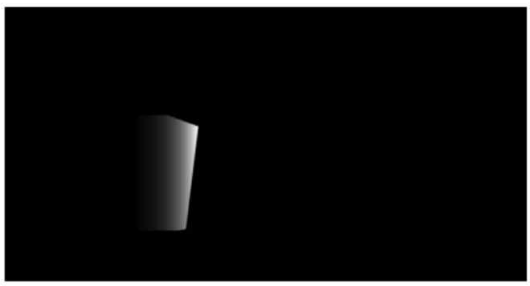
compute_overlap_region함수는 변환된 두 이미지의 마스크를 입력받아, cv.bitwise_and 함수를 사용하여 두 마스크의 교집합을 계산한다. 이 과정에서 겹치는 영역만을 추출한다.

overlap_mask1_2는 img1과 img2가 겹치는 영역을 나타내며, overlap_mask2_3는 img2와 img3의 겹치는 영역을 나타낸다.

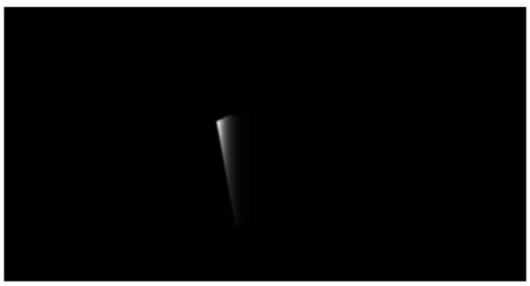
2-2. 마스크에 가중치 적용

```python
def create_stronger_weighted_masks(overlap_mask, direction='horizontal', reverse=False, strength=4.
0, blur_size=(15, 15)):
    h, w = overlap_mask.shape
    gradient = None

    if direction == 'horizontal':
        gradient = np.linspace(0, 1, w, dtype=np.float32).reshape(1, -1)
        if reverse:
            gradient = gradient[:, ::-1]  # Reverse gradient for right-to-left weight
        gradient = np.tile(gradient, (h, 1))
    elif direction == 'vertical':
        gradient = np.linspace(0, 1, h, dtype=np.float32).reshape(-1, 1)
        if reverse:
            gradient = gradient[::-1, :]
        gradient = np.tile(gradient, (1, w))

    gradient = np.power(gradient, strength)
    weighted_mask = (overlap_mask / 255.0) * gradient

    # 가우시안 블러 적용
    weighted_mask = cv.GaussianBlur(weighted_mask, blur_size, 0)
    return weighted_mask
```

Mask 2->1



Mask 3->2



이미지의 겹치는 부분을 자연스럽게 연결하기 위해 가중치 마스크를 생성한다. 이미지가 겹치는 부분은 세로 방향이므로, 가중치 마스크는 수평(horizontal) 방향으로 설정된다. 각 이미지가 겹치는 방향에 따라 reverse 값을 적용하고, strength파라미터를 통해 가중치를 조절한다. 또한, blur_size를 사용하여 가우시안 블러를 적용해 가중치 마스크를 부드럽게 처리한다.

마스크 정규화 및 캔버스 크기로 조정

```
# 마스크 정규화
strong_weighted_mask2_1 = strong_weighted_mask2_1 / np.max(strong_weighted_mask2_1)
strong_weighted_mask3_2 = strong_weighted_mask3_2 / np.max(strong_weighted_mask3_2)


# 가중치 마스크 캔버스 크기로 조정
strong_weighted_mask2_1 = resize_mask_to_canvas(strong_weighted_mask2_1, canvas.shape)
strong_weighted_mask3_2 = resize_mask_to_canvas(strong_weighted_mask3_2, canvas.shape)
```

마스크 값을 0과 1 사이의 범위로 정규화하여, 자연스러운 블렌딩을 위해 가중치가 균일하게 분포되도록 설정하였다. 또한, 캔버스 크기에 맞게 마스크를 조정하여 모든 이미지와 마스크가 동일한 크기의 캔버스에 정확히 맞춰 겹치도록 했다.

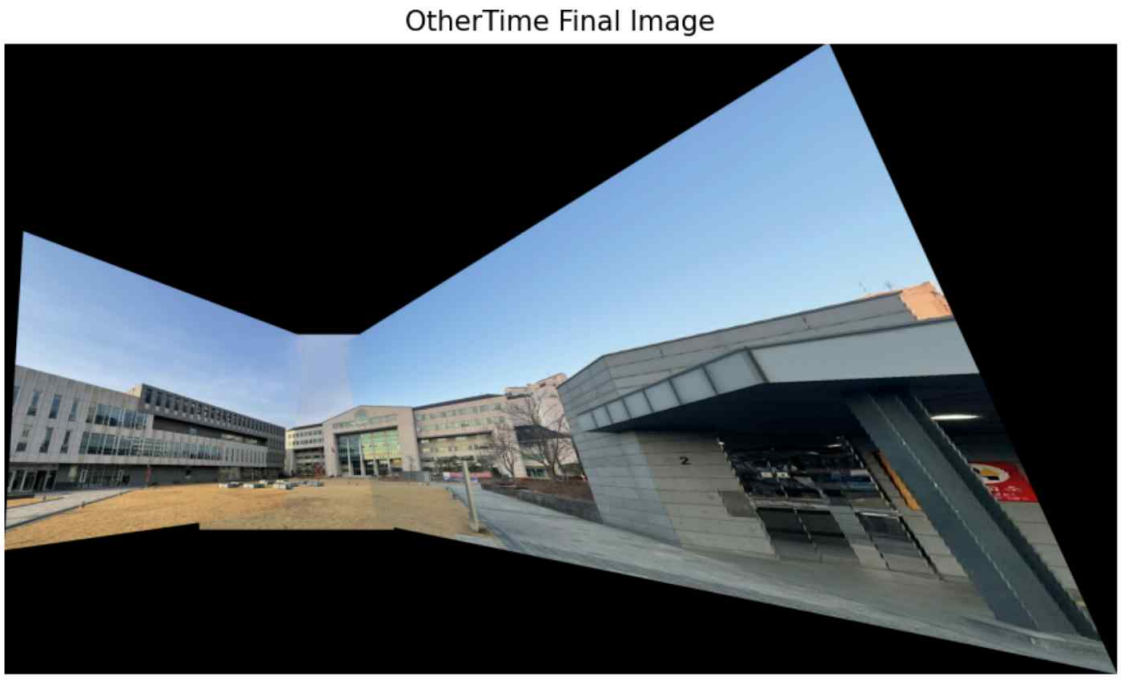
2-3. 마스크 적용과 이미지 블렌딩

```
def apply_blended_mask(canvas, img, mask, H, offset):
    h, w = canvas.shape[:2]
    H_inv = np.linalg.inv(H)

    # 캔버스 좌표 생성
    y_coords, x_coords = np.indices((h, w))
    coords = np.stack([x_coords.ravel(), y_coords.ravel(), np.ones_like(x_coords).ravel()])
    coords[0] -= offset[0]
    coords[1] -= offset[1]

    # 역 매핑
    mapped_coords = H_inv @ coords
    mapped_coords /= mapped_coords[2]

    mapped_x = mapped_coords[0].reshape(h, w).astype(np.int32)
    mapped_y = mapped_coords[1].reshape(h, w).astype(np.int32)

    # 유효 범위 확인
    valid_idx = (0 <= mapped_x) & (mapped_x < img.shape[1]) & (0 <= mapped_y) & (mapped_y < img.shape[0])

    # 마스크 적용하여 블렌딩
    for i in range(3):  # 채널별로 처리
        canvas[..., i][valid_idx] = (
            canvas[..., i][valid_idx] * (1 - mask[valid_idx]) +
            img[mapped_y[valid_idx], mapped_x[valid_idx], i] * mask[valid_idx]
        )
```

apply_blended_mask 함수는 warp_image에서 이미 다룬 역변환 및 좌표 매핑 과정에 이어, 마스크를 활용한 블렌딩과 채널별 처리를 통해 겹치는 부분을 자연스럽게 블렌딩하는 역할을 한다. 마스크는 각 이미지의 겹치는 영역에서 가중치를 조절하여 하나의 이미지를 다른 이미지 위에 점진적으로 덧붙이는 방식으로 블렌딩을 진행한다. 또한, 채널별 블렌딩을 통해 각 RGB 채널에서의 색상 정보를 일관되게 유지하고, 최종적으로 원본 이미지와 변환된 이미지 간의 자연스러운 연결을 만든다.

최종 결과 이미지

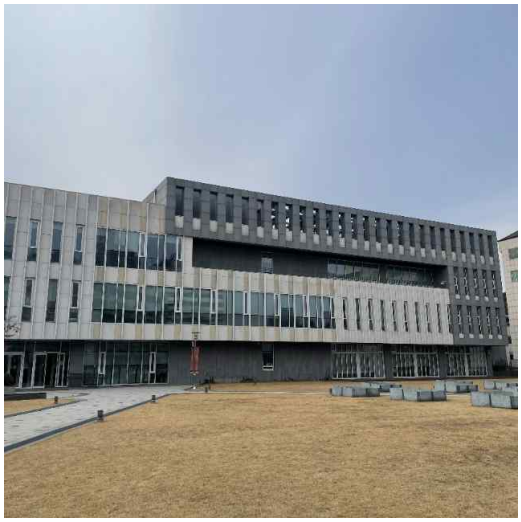
[같은 시간 촬영 버전]



[다른 시간 촬영 버전]

사용한 이미지

같은 시간 촬영 버전 - 15시 촬영



다른 시간 촬영 버전

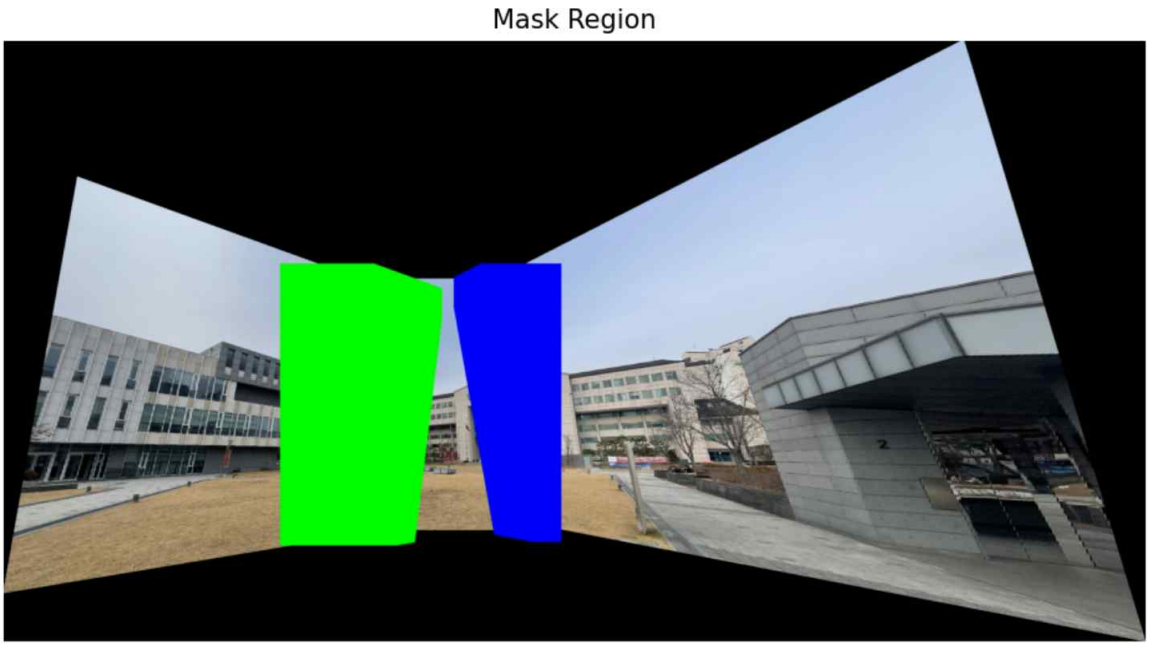
오전 11시 촬영

흐린날 오후 15시 촬영

오후 17시

3. 마무리

이번 과제에서는 SIFT, FLANN 매칭, RANSAC 알고리즘을 사용해 이미지 간의 특징점을 매칭하고, 백워핑과 가중치 마스크를 적용해 겹치는 부분을 자연스럽게 블렌딩했다. 그러나 최종적으로 완벽하게 하나의 이미지로 보이지 않아서 아쉬웠다. 일부 경계에서 왜곡이나 불연속성이 발생했으며, 색상 차이가 클 때 이 문제가 더 두드러졌다. 향후 더 정교한 블렌딩 기법과 보정 작업이 필요할 것으로 보인다.



+ 겹치는 영역을 확실하게 알기 위해 마스크 영역을 색상으로 표현하여 각 겹치는 영역의 정확도를 확인했다. 이를 통해 마스크에 가중치를 주는 과정에서 가중치 마스크의 영역을 확인해가며 블렌딩을 진행했다.