Program Architecture

Initialization and Setup:

- Load necessary libraries (cv2 for OpenCV and numpy for numerical operations).
- Define the list of image file names to be stitched.
- Set the dimensions of the output mosaic image (MOSAIC_WIDTH and MOSAIC_HEIGHT).

Main Function Workflow:

- Feature Detection and Matching: Initialize the feature detector (ORB) and the descriptor matcher (BFMatcher).
- First Image Processing: Read the first image, hardcode the corner points (for development), and define the orthophoto coordinates.
- Homography Calculation: Compute the homography matrix to warp the first image onto the orthophoto.
- Mosaic Initialization: Create the initial mosaic by warping the first image.

Iterative Processing of Remaining Images

For each subsequent image:

- Read the current image.
- Detect and compute features in the current and previous images.
- Match the descriptors and filter good matches.
- Compute the homography matrix between the current and previous images.
- Update the homography matrix to transform the current image onto the mosaic.
- Warp the current image and blend it with the mosaic.
- Update the previous image and homography matrix for the next iteration.

Finalization:

- Save the final mosaic image.
- Prompt the user to close all OpenCV windows.
- Properly release resources and close all windows.

Methods and Techniques Used

Feature Detection (ORB)

ORB (Oriented FAST and Rotated BRIEF) is used to detect and compute keypoints and descriptors in the images. This helps in finding unique points that can be matched between different images.

Feature Matching (BFMatcher)

BFMatcher (Brute-Force Matcher) is used to find the best matches between descriptors of keypoints in different images. K-nearest neighbors matching (knnMatch) with a ratio test is used to filter out poor matches.

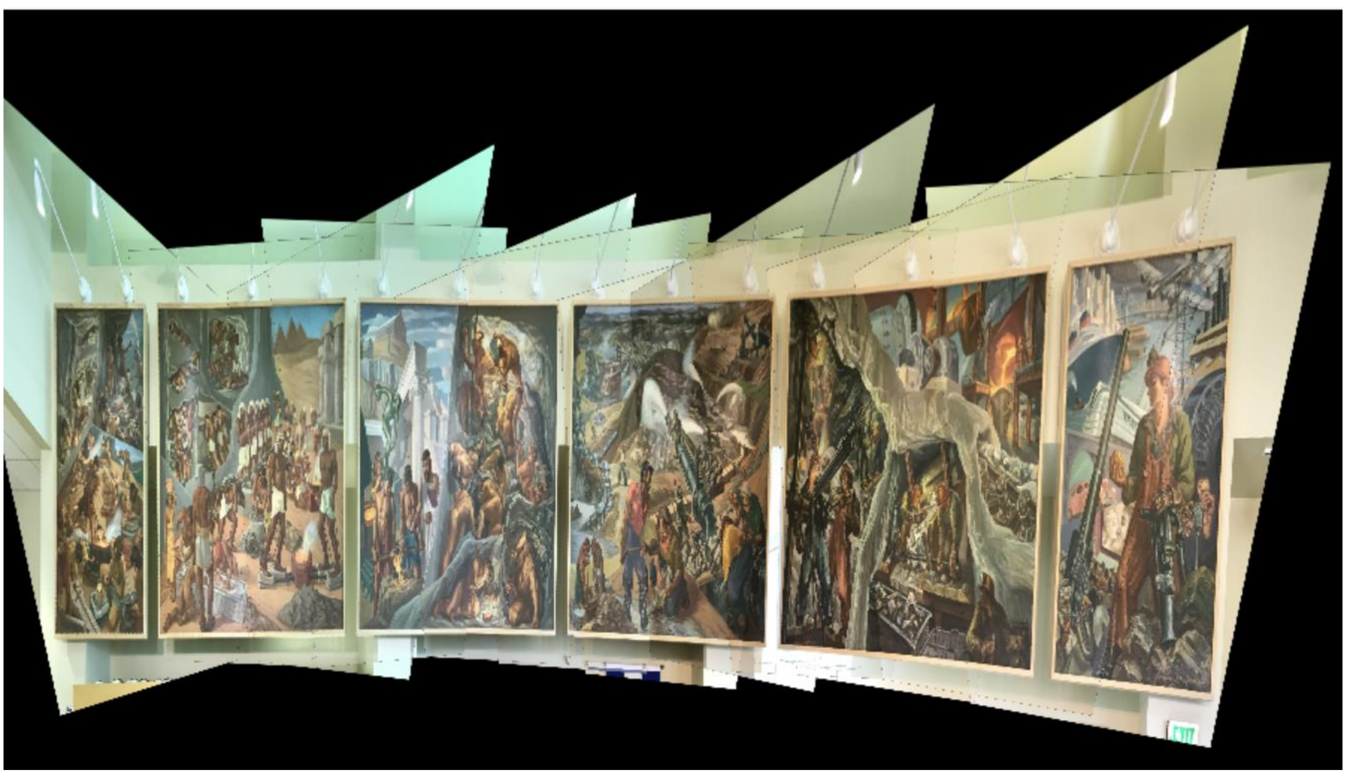
Homography Estimation

Homography is a transformation that maps points from one plane to another. cv2.findHomography with RANSAC (Random Sample Consensus) is used to compute the homography matrix that aligns one image with another. This is essential for warping images to align them in a mosaic.

Image Warping and Blending

cv2.warpPerspective is used to apply the homography matrix to warp images. The fuse_color_images function blends the warped images together, handling overlaps by averaging pixel values where both images contribute.

Result:



Code:

```python
# -*- coding: utf-8 -*-
"""
@author: zezva
"""

import cv2
import numpy as np

fileNames = [
    'mural01.jpg', 'mural02.jpg', 'mural03.jpg',
    'mural04.jpg', 'mural05.jpg', 'mural06.jpg',
    'mural07.jpg', 'mural08.jpg', 'mural09.jpg',
    'mural10.jpg', 'mural11.jpg', 'mural12.jpg',
]

MOSAIC_WIDTH = 7000
MOSAIC_HEIGHT = 1300

def main():
    print("Stitch together images of a planar object")

    # Initialize the ORB feature detector
    detector = cv2.ORB_create(nfeatures=2000, nlevels=8, firstLevel=0,
patchSize=31, edgeThreshold=31)

    # Initialize the BFMatcher
    matcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)

    # Read the first image
    bgr_first = cv2.imread(fileNames[0])

    # Hardcode the corners for the first image
    corners = [(156, 123), (377, 73), (299, 495), (32, 475)]

    print("Ok, here are the corner locations:", corners)

    # Define the actual size of the leftmost mural panel
    leftmost_mural_width_meters = 1.593
    leftmost_mural_height_meters = 2.178

    # Define the scale of the output orthophoto
    pixel_size_cm = 0.5
    pixel_size_m = pixel_size_cm / 100
```

```python
    pixels_per_meter = 1 / pixel_size_m

    # Set the size of the leftmost mural in the output orthophoto
    ortho_width = int(round(leftmost_mural_width_meters * pixels_per_meter))
    ortho_height = int(round(leftmost_mural_height_meters * pixels_per_meter))

    # Set the offset for the first panel in the output image
    offset_x_pixels = 200
    offset_y_pixels = 400

    ortho_corners = np.array([
        [0 + offset_x_pixels, 0 + offset_y_pixels],
        [ortho_width + offset_x_pixels, 0 + offset_y_pixels],
        [ortho_width + offset_x_pixels, ortho_height + offset_y_pixels],
        [0 + offset_x_pixels, ortho_height + offset_y_pixels]
    ])

    # Find the homography that maps the first image to the orthophoto coordinates
    src_pts = np.array(corners)
    H, _ = cv2.findHomography(srcPoints=src_pts, dstPoints=ortho_corners)

    # Warp the first image to create the initial mosaic
    bgr_mosaic = cv2.warpPerspective(bgr_first, H, (MOSAIC_WIDTH, MOSAIC_HEIGHT))

    cv2.namedWindow("Mosaic", cv2.WINDOW_NORMAL)
    cv2.imshow("Mosaic", bgr_mosaic)
    cv2.waitKey(10)

    # Initialize variables for iterative processing
    bgr_previous = bgr_first
    H_prev_mosaic = H

    for image_count in range(1, len(fileNames)):
        bgr_current = cv2.imread(fileNames[image_count])

        # Convert images to grayscale
        gray_previous = cv2.cvtColor(bgr_previous, cv2.COLOR_BGR2GRAY)
        gray_current = cv2.cvtColor(bgr_current, cv2.COLOR_BGR2GRAY)

        # Detect and compute features
        kp_prev, des_prev = detector.detectAndCompute(gray_previous, None)
        kp_curr, des_curr = detector.detectAndCompute(gray_current, None)

        # Match descriptors
        matches = matcher.knnMatch(des_curr, des_prev, k=2)
```

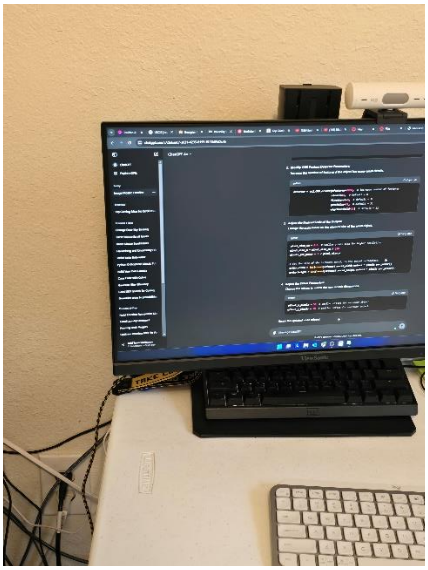
```
        # Apply ratio test to filter good matches
        good_matches = []
        for m, n in matches:
            if m.distance < 0.8 * n.distance:
                good_matches.append(m)

        # Find the homography that maps the current image to the previous image
        src_pts = np.float32([kp_curr[m.queryIdx].pt for m in
good_matches]).reshape(-1, 2)
        dst_pts = np.float32([kp_prev[m.trainIdx].pt for m in
good_matches]).reshape(-1, 2)

        H_current_prev, mask = cv2.findHomography(srcPoints=src_pts,
dstPoints=dst_pts, method=cv2.RANSAC, ransacReprojThreshold=3.0, maxIters=2000)
        num_inliers = sum(mask)
        print("Number of inliers: %d" % num_inliers)

        # Display all matches (optional)
        matchesMask = mask.ravel().tolist()
        bgr_matches = cv2.drawMatches(img1=gray_current, keypoints1=kp_curr,
img2=gray_previous, keypoints2=kp_prev, matches1to2=good_matches,
matchesMask=matchesMask, outImg=None)
        cv2.imshow("matches", bgr_matches)
        cv2.waitKey(10)

        # Combine homographies to get the transformation from the current image
to the mosaic
        H_current_mosaic = H_prev_mosaic @ H_current_prev

        # Warp the current image to align it with the mosaic
        w = bgr_mosaic.shape[1]
        h = bgr_mosaic.shape[0]
        bgr_current_warp = cv2.warpPerspective(bgr_current, H_current_mosaic, (w,
h))

        # Fuse the current warped image with the mosaic
        bgr_mosaic = fuse_color_images(bgr_mosaic, bgr_current_warp)

        cv2.imshow("Mosaic", bgr_mosaic)
        cv2.waitKey(0)

        # Update variables for the next iteration
        bgr_previous = bgr_current
        H_prev_mosaic = H_current_mosaic
```

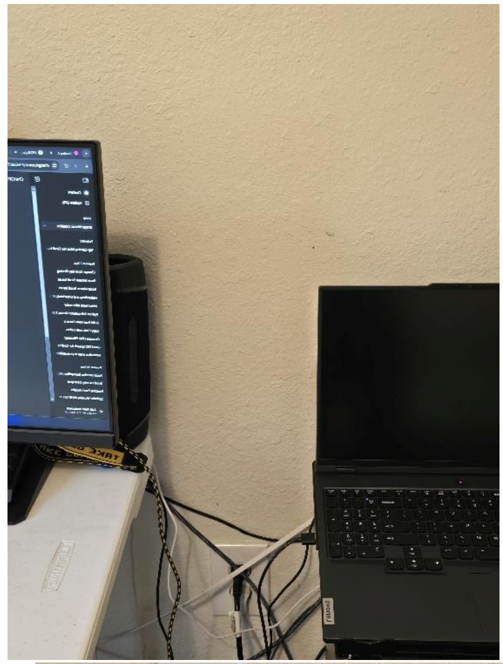
```
    # Save the final mosaic image
    cv2.imwrite("mosaic.jpg", bgr_mosaic)

    print("All done, bye!")

    # Prompt the user to close windows
    while True:
        close = input("Do you want to close all windows? (y/n):
").strip().lower()
        if close == 'y':
            break

    # Properly close all windows and release resources
    cv2.destroyAllWindows()
    cv2.waitKey(1)

def fuse_color_images(A, B):
    assert (A.ndim == 3 and B.ndim == 3)
    assert (A.shape == B.shape)

    # Allocate the result image
    C = np.zeros(A.shape, dtype=np.uint8)

    # Create masks for pixels that are not zero
    A_mask = np.sum(A, axis=2) > 0
    B_mask = np.sum(B, axis=2) > 0

    # Compute regions of overlap
    A_only = A_mask & ~B_mask
    B_only = B_mask & ~A_mask
    A_and_B = A_mask & B_mask

    # Fuse the images based on the masks
    C[A_only] = A[A_only]
    C[B_only] = B[B_only]
    C[A_and_B] = 0.5 * A[A_and_B] + 0.5 * B[A_and_B]

    return C

def get_xy(event, x, y, flags, param):
    if event == cv2.EVENT_LBUTTONDOWN:
        print(x, y)
        param.append((x, y))
```

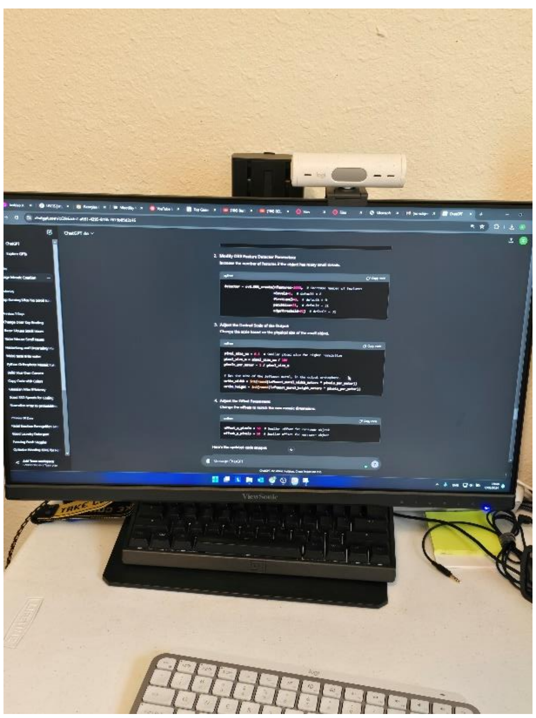
```python
if __name__ == "__main__":
    main()
```

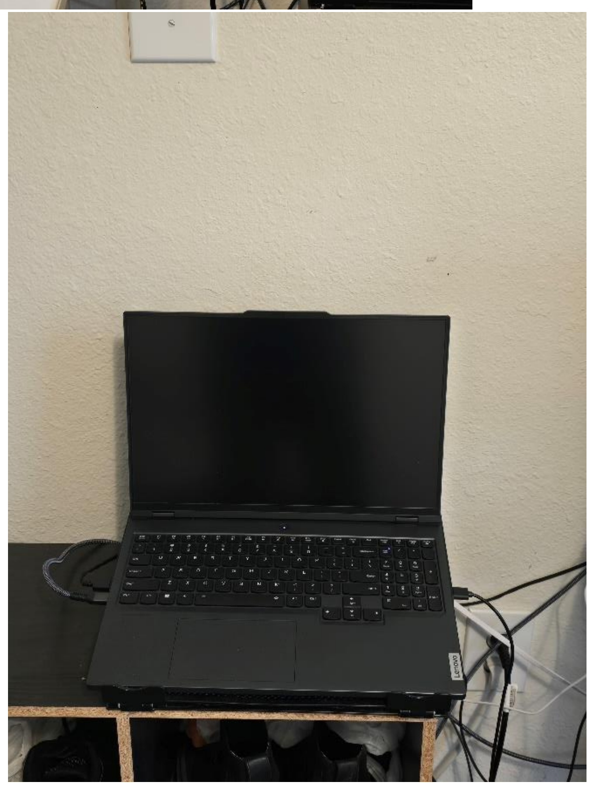
Second Output:

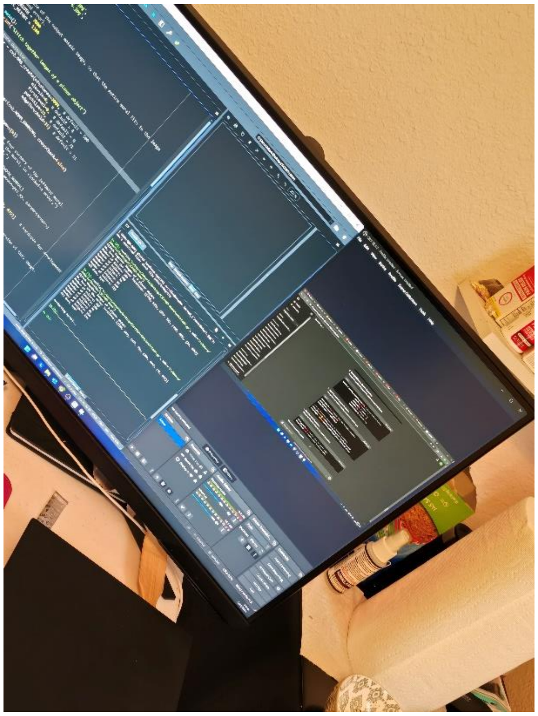
Second input files:

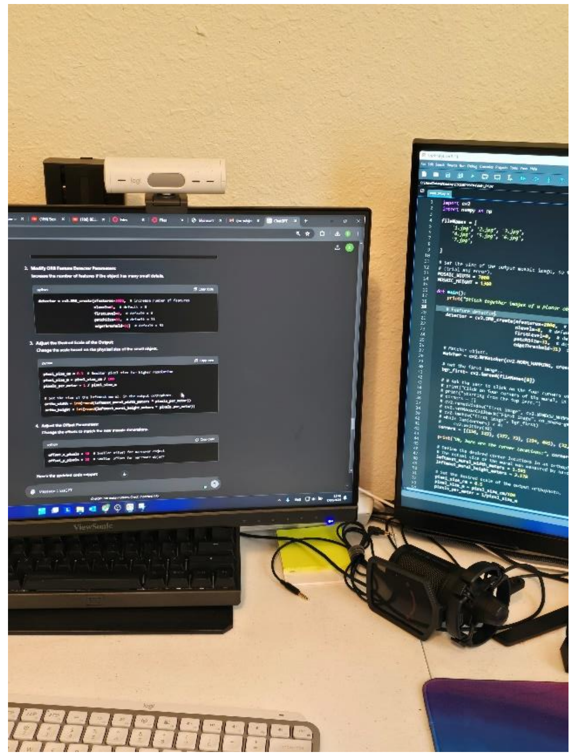


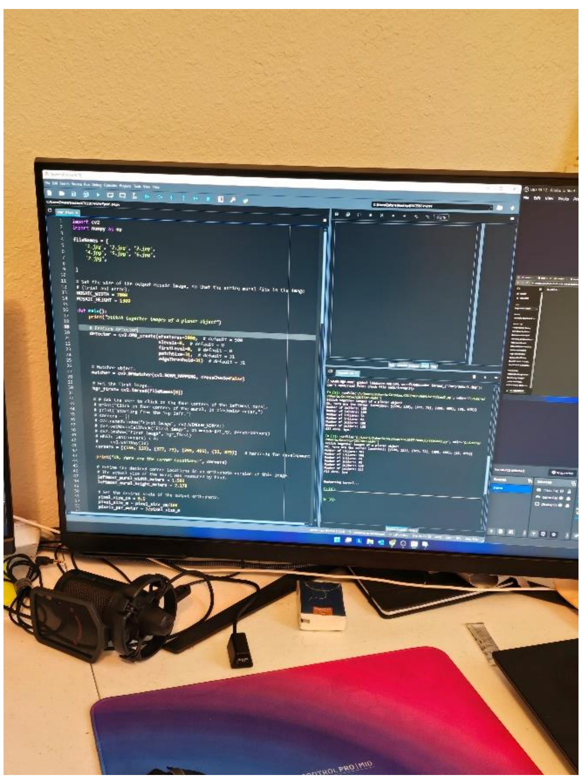




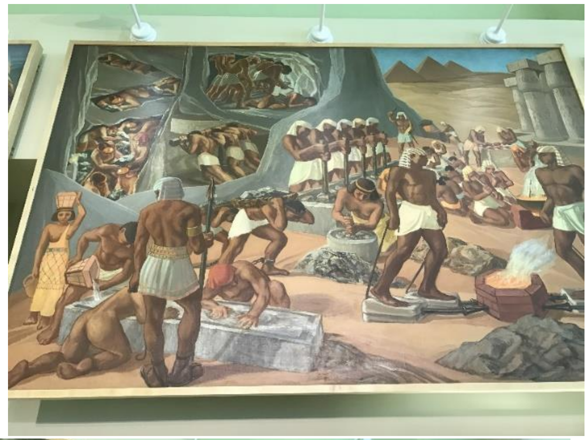


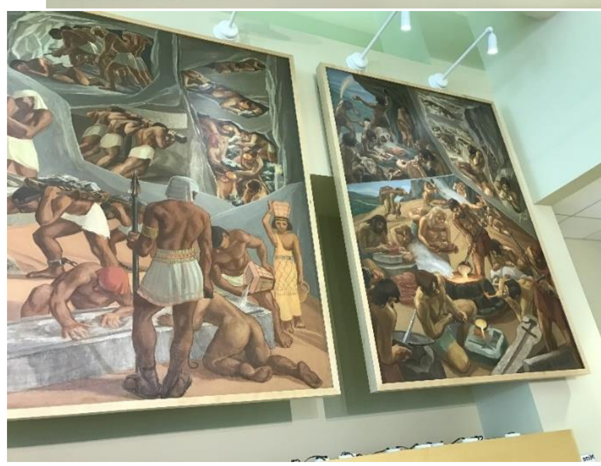






First input images:











EXIT