CSCI 4270 HW3

Aidan Westphal

February 2024

Abstract

This project implements mosaic image generation (autostitching) of images utilizing the Homography matrix, which is generated through SIFT feature detection. This project/paper will explore the individual matching and filtering steps leading up to the generation of a Homography matrix between two images and eventually the mosaic resulting from it. This paper will also explore epipolar lines and multi-image mosaics. Details of my specific algorithms/implementations will be explained and my results will be shown.

Control Flow

The gist of the project is three sequential functions: `matching`, `fundamental`, and `homography`. A directory of images is read and a double loop searches through each pair of images, calling the head of this sequence (matching). If matching succeeds with enough matches, it calls fundamental. If fundamental has enough inliers, it calls homogrphy. Thus, the total output is either `False` if at any point the images didn't match well enough or the homography matrix `H` between the two images. Then, if a homography matrix was returned, the function `mosaic` generates the stitching of the two images and is saved to the out directory.

For the extra credit/grad level, these homography matrices are stored into a variable named `graph`, a linked-list graph structure where each index (image node) contains a list of tuples (each tuple represents an edge, containing a second image index and the homography matrix H between the two). More on this specific algorithm will be mentioned later, but the last part of the code finds the largest connected component of this graph and generates the multi-image mosaic.

There exist several helper functions, such as `read_dir` and `write_dir`. Note that `read_dir` will also downsize any images whos largest dimension is above 1080 pixels, this allows one to quickly generate mosaics from powerful cameras like iPhone. `drawlines` draws epipolar lines (important later).

Matching, Fundamental, and Homography

The code in these sections is fairly straightforward. I used SIFT to extract keypoints from images and used `cv2.FlannBasedMatcher` along with a for loop (through its results) to select good matches and their corresponding points. I had to loop here because it was too difficult to extract information, since the return of `flann.knnMatch` contained `DMatch` types. Note that this part was partially based off a tutorial from OpenCV, found here: `https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html`

As a rough approximation from what was stated in the notes, I chose a ratio of 40/3000 for SIFT matches, i.e. an image with $2000 - 4000$ SIFTs tends to have about $20 - 200$ matches depending on the difficulty. This barrier rarely got triggered unless images were glaringly different.

For calculating the fundamental matrix I used `cv2.findFundamentalMatrix` with RANSAC (threshold of 4 and confidence of .99). Furthermore, I chose a threshold of 30% for inlier matches to move on to homography generation.

For homography generation, I used `cv2.findHomography` with RANSAC. I did not provide a confidence value this time but gave a threshold of 4. Here, I determined that images that are good for mosaic generation need at least 50% of inliers from the Fundamental matrix to reappear in the Homography matrix. This proved to be the largest barrier for images but those which got past always stitched well.

The following table shows each pair of images and the algorithm's decision:

Mosaic Algorithm Decisions					
Image Set	**Image Numbers**	**Match %**	**F %**	**H %**	**Mosaic Created**
drink-machine	0, 1	9.5	49.23	49.33	No
drink-machine	0, 2	9.7	57.58	28.8	No
drink-machine	1, 2	5.3	9.64		No
frear-park	0, 1	16.2	91.6	99.17	Yes
office	0, 1	35.6	88.05	90.45	Yes
office	0, 2	24.1	71.9	65.45	Yes
office	1, 2	21	76.32	65.52	Yes
tree_mrc	0, 1	17.8	85.57	84.56	Yes
tree_mrc	0, 2	5.2	53.11	62.44	Yes
tree_mrc	0, 3	22.4	92.52	87.42	Yes
tree_mrc	1, 2	13.3	70.77	87.59	Yes
tree_mrc	1, 3	9.6	73.22	69.36	Yes
tree_mrc	2, 3	3.3	28.16		No
vcc-entrance	0, 1	5.1	37.12	76.47	Yes
vcc-entrance	0, 2	23.1	83.93	56.63	Yes
vcc-entrance	1, 2	16.3	35.44	71.43	Yes

Mosaic Generation and Blending

After a homography matrix is generated, two images are stitched together. There are two optional parameters to `mosaic`, that being a tuple `origin` and a bool `new_center`. For sake of explanation, we assume these have no impact for now, as they only apply for multi-image mosaics.

As a result of making multi-image mosaics, I structured my mosaic function backward, i.e. image 2 is mapped onto image 1. Thus, I use the inverse of H. The first task is to calculate the bounds and transformation of the mosaic coordinate system. We take the four corners of image 2 and map them onto image 1, and then proceed to find the minimum and maximum value of these corners plus the original corners of image 1. The difference in these values is by definition the width and height of the new mosaic image. We now need to find the translation factor for I2 which is applied on top of the homography. This is simply the minimum row and column values, i.e. the location of the origin in the "mosaic coordinate system."

I then used `cv2.warpPerspective` with the matrix `T @ H` (homography and translation) to map image 2 onto image 1. I then place image 1 into this image starting at (minr,minc) which is either positive or zero. This is the distance between the origin of image 1 and the mosaic coordinate system origin, thus image 1 will be located this distance offset from the corner.

For blending, I averaged the images where they intersected and, everywhere else, simply composed the images onto eachother (summed). This required making a mask (`img_combo`) where the images intersected, accomplished by taking the max along the third dimension (RGB) of each image and multiplying. As such, only if all values are zero will this number be zero. This mask is applied over a blended version of img1 and img2 (50/50 blend) creating `img_blend`. Then, two masked versions of images 1 and 2 (`img_f1` and `img_f2`) are made by copying their corresponding images and being set to zero whenever `img_combo` detected an overlap. The final image is a discrete sum of these three discrete components.

Epipolar Lines

Epipolar lines are generated inside of the `fundamental` function and utilize `cv2.computeCorrespondingEpilines`. This then calls the function `drawlines` which generates an image with the lines drawn over it. Note that generating the corresponding epipolar lines from one set of points refers to the lines on the opposing image, so the inputs to the functions are mismatched. The drawlines function then sifts through all the line coordinates and calculates two points on each line and converts their types to match with the function `cv2.line` which then draws the line on the image. I also draw the keypoints using `cv2.circle`. Note that the specifics of this function, like how to calculate the points, were based on the boundaries of the image (draw the line from one edge of the image to the other image). I did not come up with this myself, but rather based it off of the following tutorial on OpenCV's website: `https://docs.opencv.org/4.x/da/de9/tutorial_py_epipolar_geometry.html`

Multiple Image Mosaics

This was implemented with help of some dynamic-style programming along with some important simplifications in my code (such as how `mosaic` maps image 2 inversely onto image 1. As mentioned before every time a homography matrix is returned for a pair of images it is stored in a graph where an "edge" between the two images is the homography matrix. In practice this is a double list where the list at index i contains a tuple with (j, H) where H is the homography between i and j. As such, the largest image mosaic is the largest connected component, which is found through depth-first search. `largest_cc` runs a full DFS algorithm using `dfs` to explore each connected component. This part of the code is somewhat messy but basically the graph is copied and each initial list (corresponding to the vertices) is turned into a tuple (in my code I used a two-element list) where the second value is `True/False`, representing if that vertex is visited. Thus, we loop through the graph from $i = 0$ to $i = len(graph)$ and we start `dfs` whenever we find that the given vertex has not been visited.

We store connected components in a dictionary. This was the best way I could implement it as we could look up any index (corresponding to the terminal image j) to find its mapping onto the base image i. As such, in the

actual `dfs` algorithm, we update this connected component iteratively, multiplying the homography matrices together in a chain. If we are searching through dfs at a vertex i and find an unvisited j, we define the homography between j and the root of the connected component by taking the homography from the graph and multiplying it onto the one found in the connected component (represents the chain of homographies from the root of the search tree to the current node).

Specifically for dfs, we note that `graph[i][0]` is the list of edges from vertex i and `graph[i][1]` is whether or not vertex i was visited. Thus, when j pulls from `graph[j][0]` it iterates over tuples in the following form: `(k,H)` where k is a new vertex reachable from j and H is the homography between them. We then note that the connected component dict `cc[i]` pulls the homography matrix chain that gets one from the root of the connected component to i. As such, multiplying this by the new H will give us the homography between the root and j.

The value of this dictionary structure comes in the final step, where we loop over all indices in the final connected component. All of these images are reachable from the root image and the value of their dictionary keys is the homography matrix to reach them from the root image. Thus, the final generation is a single loop.

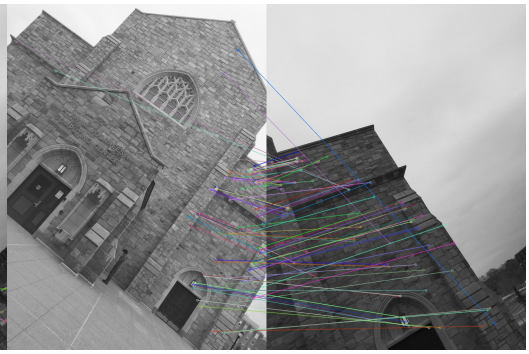
We also note the annoying process of updating the transformations of this multiple-image mosaic through each iteration. Remember that `origin` tuple in `mosaic`? After each stitching, this is updated to the new coordinates of the first image's origin. As such we can guarantee that all latter images are properly transformed onto the original image and not messed up by the changing dimensions of each mosaic. This led to some more complex definitions for the image size, starting point of image 1 (one has to edit the translation matrix), etc. which proved to be difficult to get correct, but I did succeed with some step-by-step debugging.

Results

I will now show and analyze some of my results for each part.

Matching Keypoints

(a) Matching



(b) F Inliers

(c) H Inliers

Figure 1: VCC Images 0 and 1 Matching

Above are some examples of how SIFT features are shrunken after each mapping. Recall that, with RANSAC for the latter two images, the points found are those that survived within a certain threshold from the estimated F and H matrices. As such, the inliers become more and more regular.

(a) Matching

(b) F Inliers

(c) H Inliers

Figure 2: Drink Machine Images 0 and 1 Matching

Here's an example back to those drink machines that didn't generate any mosaics. You can see how fast the points dropped away after the fundamental matrix inliers were added. In this pair, the percentage of homography inliers was just under 50% so no mosaic was formed. However, you can still see that the algorithm was working as intended.

Epipolar Lines

Figure 3: Office Epipolar Lines 1 and 2

For any image, a line can be extended through this camera and said point, known as a backprojection line. Anywhere along this line could the actual three-dimensional coordinate lie. If a second image is taken, the image of this line is known as the epipolar line. Through this construction, the epipolar lines converge at the other camera's pinhole, known as an epipole. Notice above that all lines do pass through the keypoints and converge. It's hard to visualize where exactly they converge in a 3D sense but it checks out.

Figure 4: Drink Machine Epipolar Lines 0 and 1

This also illustrates the previous point. Note how these lines appear more horizontal as the cameras are nearly rectified. With rectified geometry, image planes are coplanar with the base axis (the axis between the two camera pinholes, I forget the exact term). As such, the epipolar lines can never reach the other camera's pinhole and the epipolar lines become parallel, along the rows to infinity. Rectification is a process where images are transformed such that their epipolar lines are horizontal, as such one can simply search along the corresponding row in the other image frames to find matching features.

Single Mosaics

Figure 5: Frear Park

The Frear Park mosaic is one of the most perfect from the given data set. In the table, the homography matrix saw 99.17% inliers. Here you can see the successful sitiching of images, correct boundary sizing, and a decent blending algorithm. However, blending is not super important here as the luminosity of the two images is almost the same, likely part of the reason why H saw so many inliers. You can see a slight artifact from the masks with my blending algorithm in the form of a thin black pixel lining.

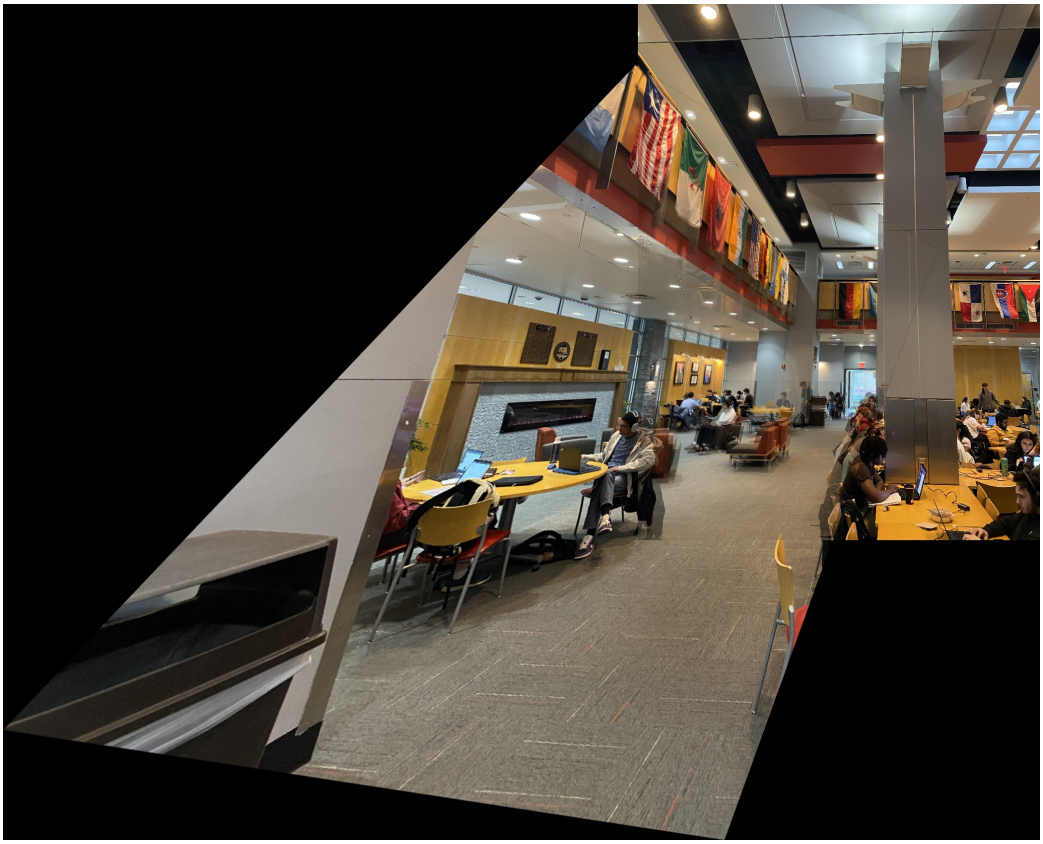
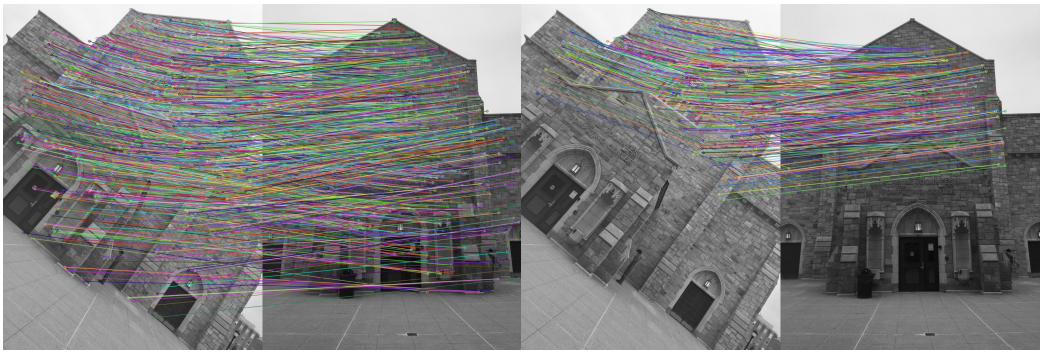


Figure 6: Panera

As seen here, the algorithm still generates strong matches, however some sections appear blurry. This is likely due to motion (from people) messing with the SIFT matches along with the stark difference in orientation of the two images, one being taken towards the nearby garbage can while the other is farther back. This blurriness was a problem but it's hard to fix. Stronger thresholds might fix it, or simply taking better photos.

Figure 7: VCC

As seen here, the top of the VCC aligns perfectly, but the entrance is warped off. There are some major issues with the homography matrix in this image, as it seems that only part of the image properly changed. The source of this mishap can be seen from the matchings below:



(a) F Inliers (b) H Inliers

Figure 8: VCC Images 0 and 2 Matching

As you can see, the homography holds strongly for the top of the VCC but not the front door, where virtually all points disappeared. This image pair had 56.63% homography inliers so it was strong enough to be included, but you can see how this error can result in poor mosaics.

Multiple Image Mosaics

This is where I had a lot of small issues, which I will go into depth on. This also led to me discovering pressing issues in general. For the main assignment I had to generate each pair of mosaics where one had a strong enough H matrix. When I took my own datasets of 7 to 8 images, some of these images would qualify for matching but the warping would be extreme. As such, it would generate massive images which would kill my program (before generating the multiple image mosaic, there may exist some pair of two images which returned an H matrix but the warping led to a massive image, on the magnitude of 30,000 x 30,000 pixels, which would kill the mosaic function). My first solution was to resize any overtly-large image to have a maximum row/column value of 1080 pixels. This worked well, but the above issue was still pressing. As such, for multiple image stereos of my own larger data sets, I had to remove mosaic generation on each new pair of images. Instead, it generates the graph and begins composing at the end. The following are some of my favorite results:

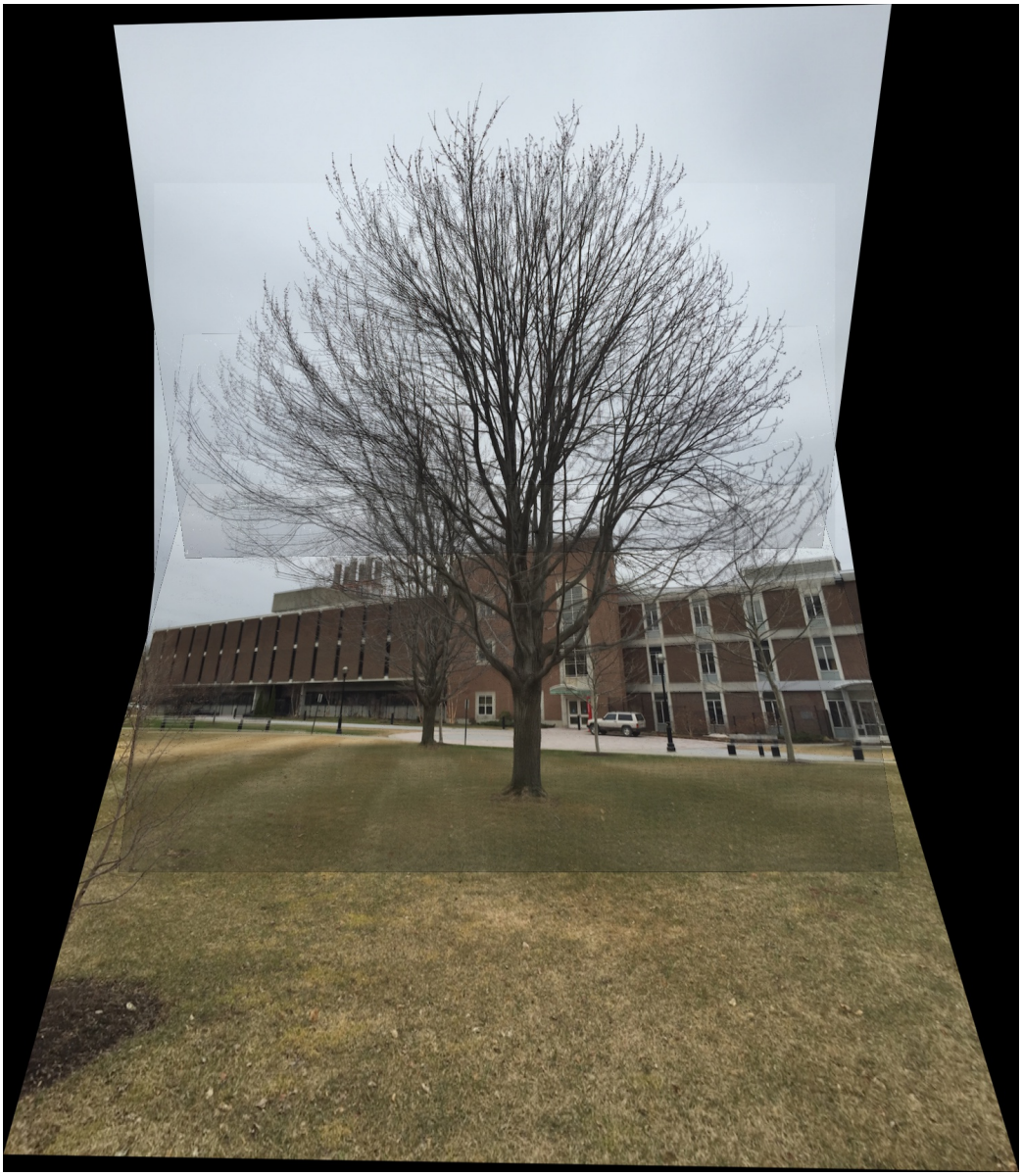


Figure 9: MRC Tree Final Mosaic

This is the full composition of all MRC Tree images. As you can see, the image appears well. It took an EXTREMELY long debugging time for me to get the bounds right for the composition image. In fact, it wasn't until writing this report and giving it one last try that I succeeded. Behold!

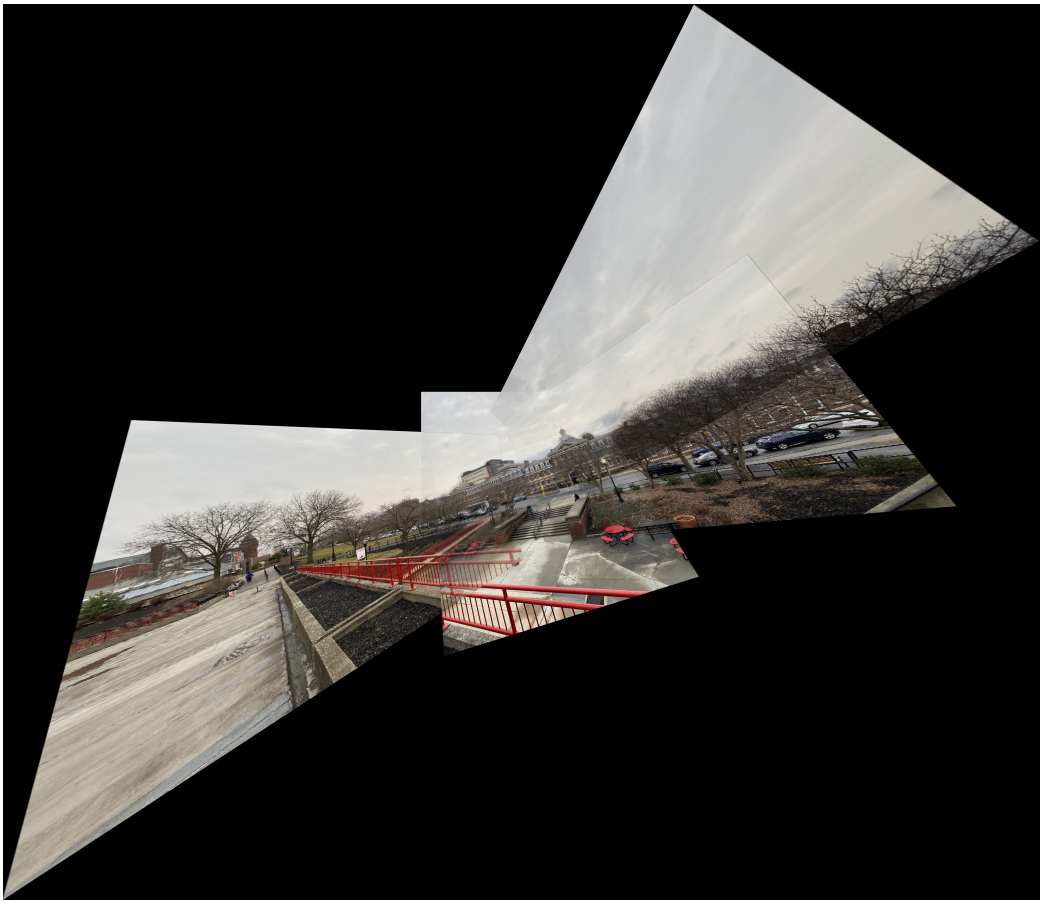


Figure 10: 15th Street (from Union)

This is my favorite result. This was taken outside the Union overlooking 15th street. Once again, you can see the bounding is correct along with the mapping of images.

Figure 11: Panera

This is an example of how to break my algorithm. First off, this mosaic contains EIGHT images! Panera is a bad spot to run this as people keep moving and it messes with SIFT, leading to blur. This image also highlights a REALLY ANNOYING BUG which I couldn't fix. Sometimes the homography matrix will map small snippits of images around the image. This leads to weird black areas and sometimes it will also mess with calculating the boundaries, as say one of the corners of image 2 may map thousands of pixels away and stretch out the image. I don't know why this is happening but if you have any idea please let me know.

Closing Thoughts

I got carried away like mad on this homework because it was super fun making the mosaics, also debugging the aforementioned multiple-image mosaic things had a huge incentive on top of it (seeing cool mosaics). The plus side of this code is its speed. The slowest part of this algorithm is the mosaic generation which doesn't take long. Through the iterations of my code, I have included several optimizations, including the image downsizing which further increased my efficiency. I am satisfied with my thresholds as they always produce great results but in some scenarios, such as the drink machine, it led to no returns. There's also the weird issue with the homography wraparound expressed in the previous image. One thought I had involved taking the determinant of the homography matrix to weed out any that causes crazy changes. When generating multiple image mosaics, I could prioritize a stitching sequence using the safest/best homographies, one idea could be a weighted graph with `1 - percent_inlier` as a weight, then running Dijkstras to find the optimal (best mapping) paths from the root image to each other image. However, this is too much work for a single homework. My code does not account for the quality of homography mappings beyond the simple thresholding, but this is acceptable for my use cases. Furthermore, my code works fast, blends well, and returns quality and correct solutions.