December 13th, 2020

Matan Segal

msegal@smu.edu

CS 5390 – Fall 2020

Project 3 – Augmented Reality using Planar Homography

Introduction:

In this project, we requested to implement an Augmented Reality application using planar homography to create ordinary object "come to life" by replacing the expect object surface with a dynamic animated video projected onto it. First, we will need to match corresponding point from the image of the object by its own and the object in a scene. Once we will have array of matched points, we will be able to compute the homography and wrap the rectangle image and place it in the scene where the object found. Using this perspective, we will be able to create the illusion by generating the video stream where the rectangle we wrap in the scene streams the frames from the video. The next level, in the bonus section, we will be requested to generate the video on the object in a scene, where the scene is a video itself, means the object moves every frame.

Implementation:

After loading the scene image (img_1) and the object image (img_2), I used the ORB feature detector. At the beginning I tried to use SIFT/SURF technique, but since those are patented, I chose the ORB technique.

ORB stands for Oriented FAST (FAST algorithm for corner detector) and Rotated BRIEF (binary robust independent elementary features). This feature detector has two parts[1]:

- **Locator**: where we identify special points on the image – corners, and save their x, y coordinates using the FAST algorithm.
- **Descriptor**: takes the points found in the previous step and keep just the interesting points. The evaluation is based on array of numbers. The ideal situation is where same physical points in two images should have the same descriptor. This process using the BRIEF algorithm.

In my code I create detector object for the locator part of the feature detecting and a descriptor object for the description part. First, I called the *detect()* function for each image to save its keypoints (*keypoints_1* and *keypoints_2* correspondently). Later, I used the *compute()* function to find the

descriptor array of each points (*descriptor_1* and *descriptor_2*). Once I have the description for each point, I called the *match()* function, which find for each point in the first descriptor best matched point in the second descriptor.

After having the vector of matched points (named *matches)* I filtered points with higher distance. I did so by finding the minimum distance of all matched points and set the threshold to be 2.5 times larger than the minimum distance (2.5 work the best for me after testing). Once I saved all the good matches in a vector (named *good_matches*) I print the matched points on the two images on the window "img_goodmatch" (shown in the result section).

Since the function *findHomograpyh()* receives vectors of type *Point* and not *keyPoint* like I used before, I needed to extract the point features from the keypoints. I save those extracted points in the vectors *matched_points1* and *matched_point2*, then I called the *findHomograpth()* function to compute the matrix H and showed the wrap object in the scene perspective in the window "Warped source image" (shown in the result section).

To print the video's frames on the scene image, I will need for each frame to replace the scene pixels with the relevant pixels from the wrap frame. Instead of looping through all the pixels in the images, I will limit the looping to a specific, smaller rectangle. I chose the rectangle by finding the limits of the wrapped objects, using the following steps. First, I set a vector (*corners_source)* which contain the four corners of the object image. Then, I compute the corresponding points in the scene image, by multiplying each point with the matrix H, as described in figure 1 below:

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = H \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

Figure 1 – finding corresponding point in the second image using the H matrix[2]

Finally, I found the highest point in the scene image, as well as the lowest, rightest, and leftist points, and looped just through this rectangle for each frame instead of the whole image. (the process shown in lines 99-116 in the code).

For the last part of the code, I loaded the video I wanted to represent on the object in the scene and looped for each frame of it. first, I checked if it is an empty frame (when the video is done) to break

in such a case. Then, I resized the frame to be the same size of the object image (img_2) and calculate the corresponding homography frame, using the H matrix I found before and the *wrapPerspective()* function. Having the wrapped framed and the scene image, I combined those by looping through the rectangle I found above, and replaced each pixel in the scene image (img_1) with a pixel from the wrap frame of the video, just if the corresponding pixel in the wrap object image was not in black (Scalar(0,0,0)), as shown in lines 146-151 in the code. Eventually I showed the frames in the window "Frame". The loop can be exit by clicking on the Esc button.

Since I tried to do my code as general as possible, by using the ratio between the threshold and the minimum distance between two points and object image size to set the rectangle I am looping through and translate it to the scene, my program was very flexible to work with variety kind of data inputs and it worked pretty well with all the examples given as shown in the result section.

Results:

Below shown the results using the cv_cover.jpg and cv_desk.png images:

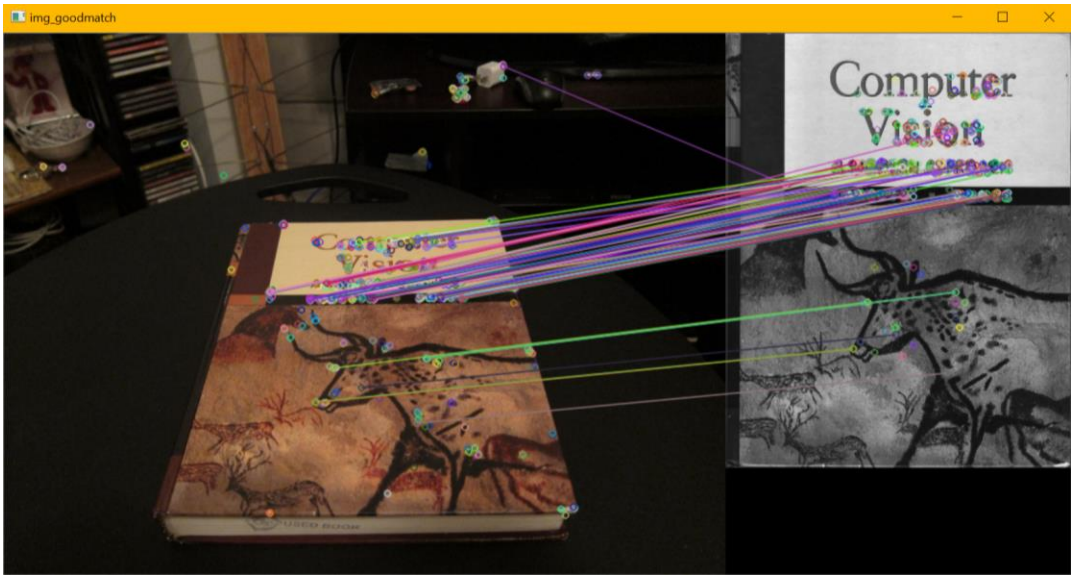

Figure 2 – corresponding points between cv_cover.jpg and cv_desk.png

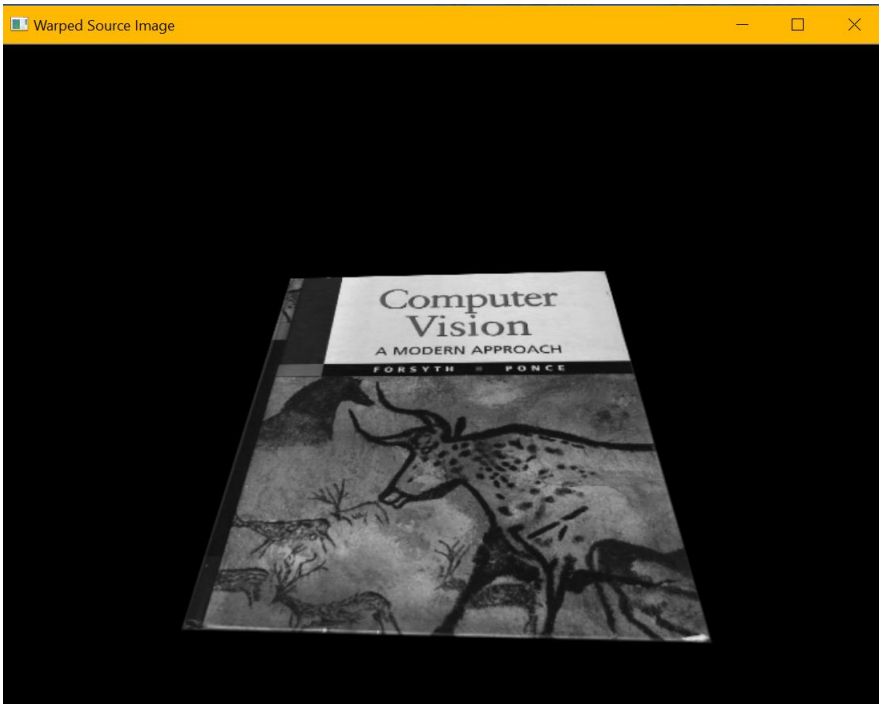

Figure 3 – wrapped object image in the scene

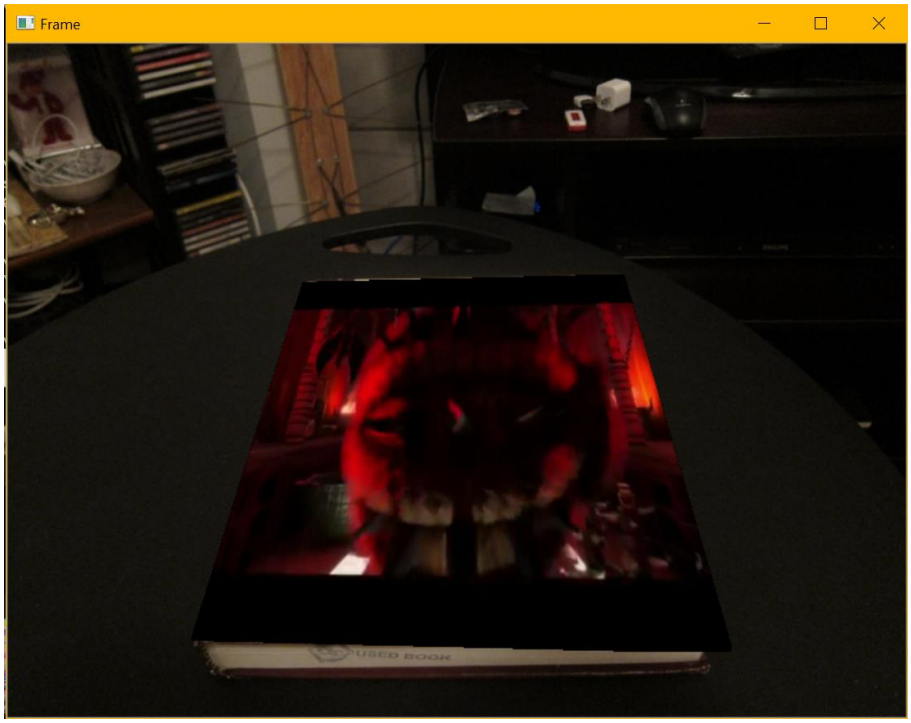

Figure 4 – video running on the cover of the CV book in the scene

Below shown the results using the hp_cover.jpg and hp_desk.png images:

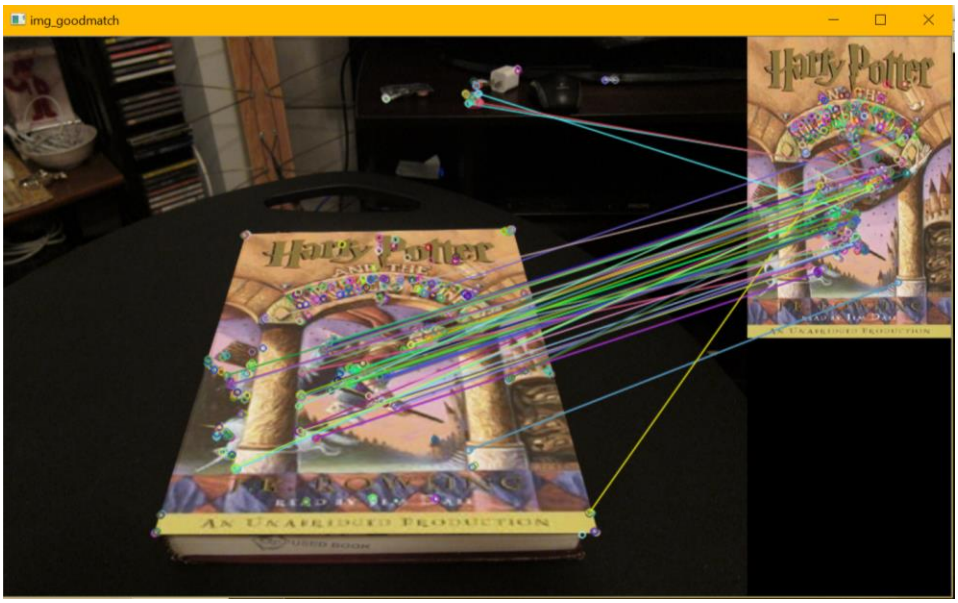

Figure 5 - corresponding points between hp_cover.jpg and hp_desk.png

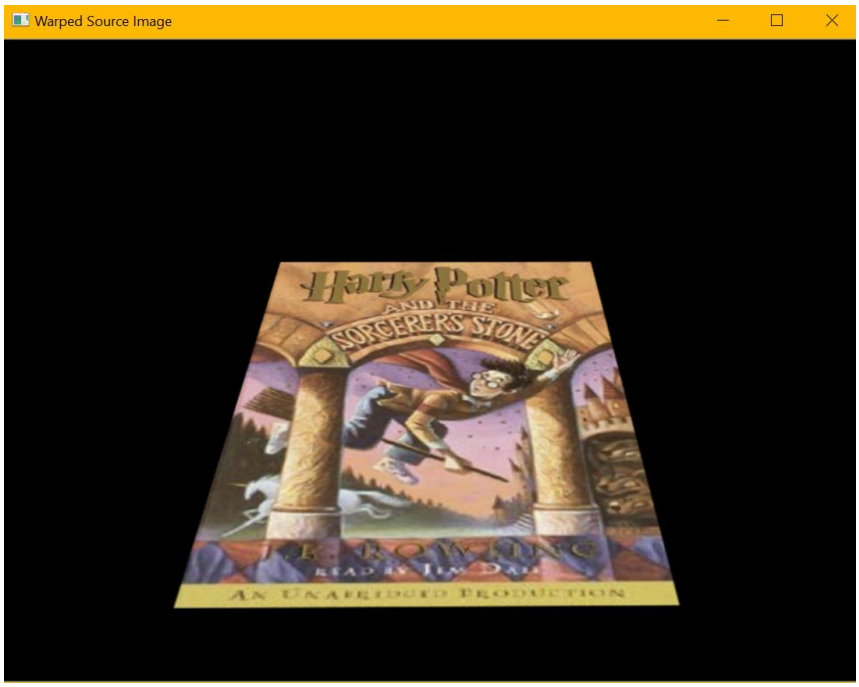

Figure 6 – wrapped object image in the scene

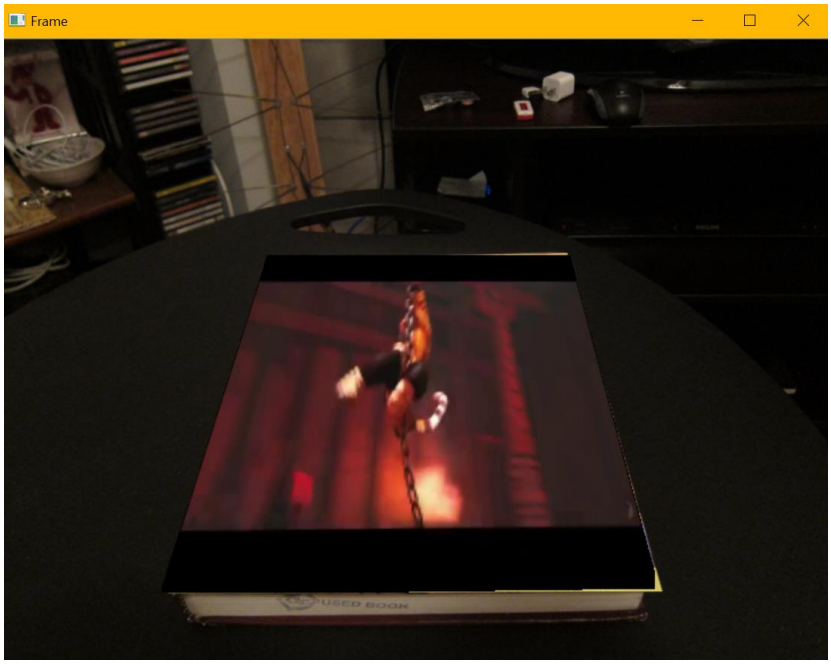

Figure 7 – video running on the cover of the Harry Potter book in the scene

Below shown the results using the box_in_scene.png and box.png images:

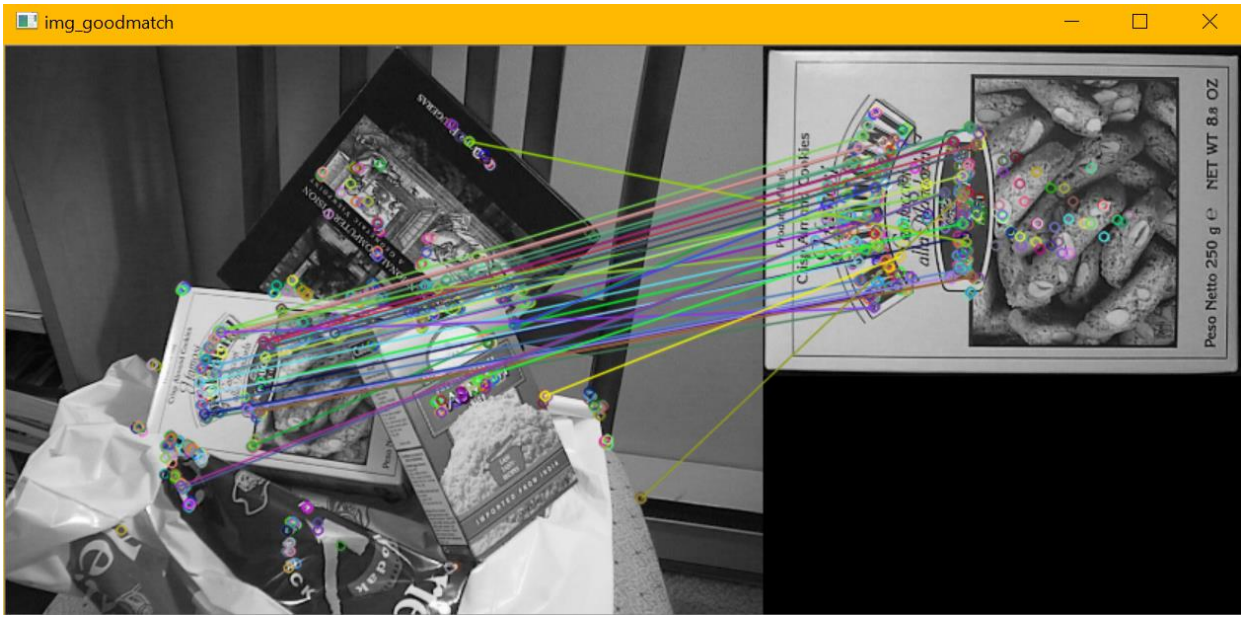

Figure 8 - corresponding points between box_in_scene.png and box.png

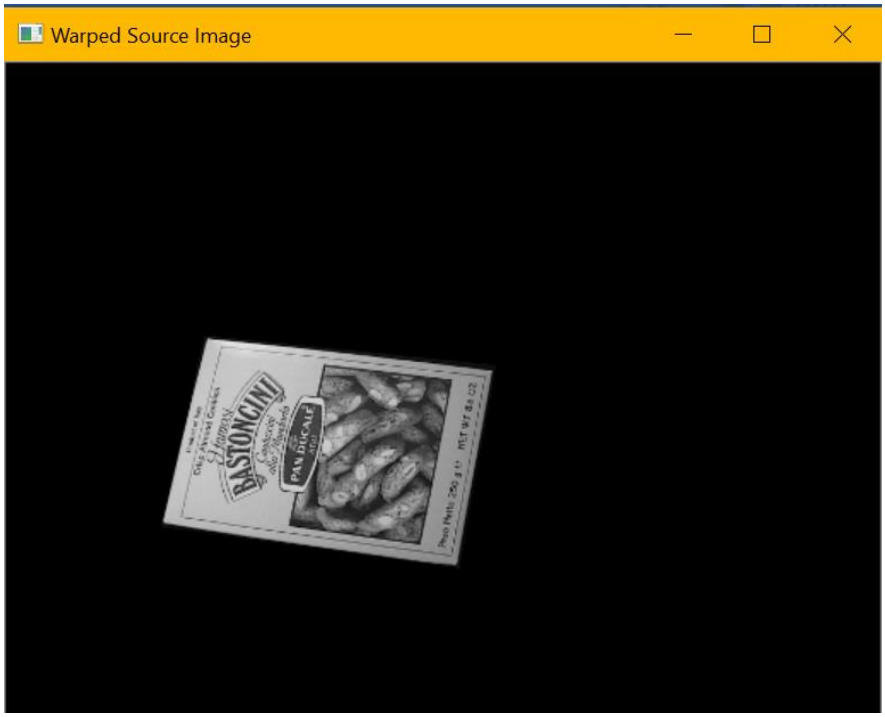

Figure 9 – wrapped object image in the scene

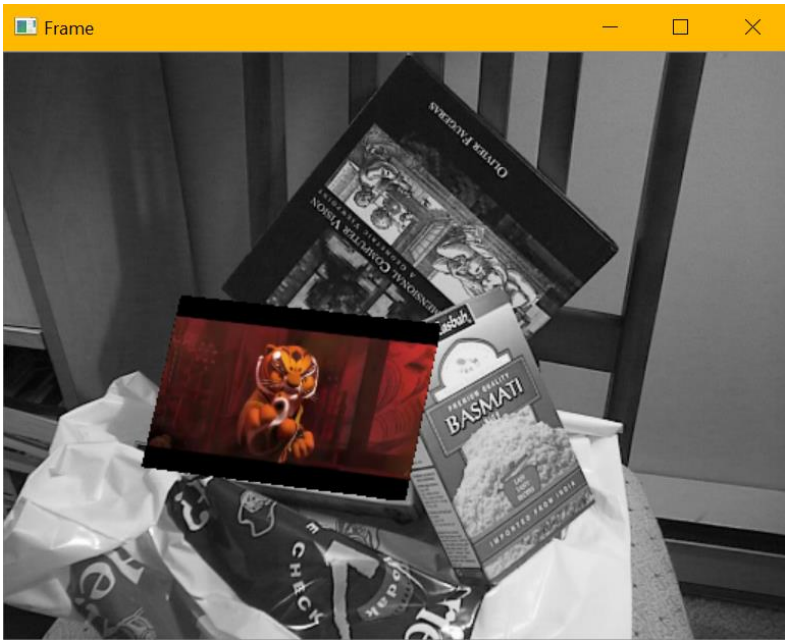

Figure 10 – video running on the cover of the box in the scene

Bonus:

The implementation of the bonus section is shown in the file *project3_Segal_Bonus.*

For the bonus, I used the base structure of the original source code, but since here I need to set the homography matrix every frame, I needed to be more efficient in doing so.

First, I loaded the object image (img_2) to be used each time for calculating the matrix H. Then, I got into the while loop reading each frame from the scene video (named *cover_video)* into img_1. In order to make the program more efficient and make the calculation faster, I change the homography calculation just once every 7 frames. I did so by calculating the homography in different thread while the frames still running on the previous perspective calculated. On the first iteration I am calculating the first homograpyh, then update it just once in 7 frames. I chose 7 frames, because this is the lowest number of frames I could use and still see the video running almost fluently on the cover of the book.

The thread (t1) call each 7 iteration to the function *compute_perspective()* which doing the whole process, as described in the implementation section, for calculating the H matrix. Inside of this function, I set the detector and the descriptor of img_2 (the object image) just on the first time, when the counter is 0 and it need to calculate two descriptors. The rest of the times, it needs to calculate those features for the new frame of the video each time the function is called. Here, I used hard coded threshold (set to 150 after testing) for keeping the good matches but saving time of finding the minimum distance.

Once the thread t1 done its work, it is joined to the main thread (6 frames later). At this point I am resizing the frame and wrapping the perspective using the H matrix I calculated in thread t1. Next, I needed to update the output frame (called *out*) with he wrap frame on the cover of the book.

Instead of calculating the rectangle I need to make the looping within each frame, I used the build-in OpenCV function *forEach()*[3]*.* This function runs given function over all the matrix elements in parallel. The operation passed as argument has to be a function, which in my code is the function *operator()()* of the struct *Operator*. This way, it runs over all the pixels, but in parallel, make it much faster, even when I used the previous method of reduce the loop just over specific rectangle. Once again, the user can exit the loop using the Esc button.

Conclusion and discussion:

I started this project knowing the theory behind homograpthy and perspective of images but with not enough knowledge of how to use it in real life, like needed for this project. This project made me self-researching and learn how to use the build-in OpenCV functions in order to achieve the requirements based on the theoretical knowledge we gain in class. All the references I took information from are mentioned as comments in the code, above the section I used them.

I think in the first part of the project (not the bonus part), I implemented a decent way to handle time efficiency and simplicity of computation using the reduce among of pixels I am looping through. Also, I gain accuracy by saving just the good matches using a generic way which could be used by variety of inputs data.

In the bonus part, I took the computation time efficiency even farther, due to the requirement of making the video running fluently, but also a need for calculating the most accurate homography for each frame. I made a compensation of calculating the H matrix just once in every 7 frames, but I did it in parallel to the running frames, which made it much faster, but with a cost of "losing" the wrap perspective object a bit each time. I got familiar with the most efficient way (in the OpenCV library) running a function on matrix's elements in parallel using the *forEach()* function. All of this taught me much more about the OpenCV library and implementation of tools using the homograpy and the Feature2D section in the library.

References:

[1] https://www.learnopencv.com/image-alignment-feature-based-using-opencv-c-python/
[2] https://math.stackexchange.com/questions/2388259/differences-between-homography-and-transformation-matrix
[3] https://docs.opencv.org/3.4/d3/d63/classcv_1_1Mat.html#a952ef1a85d70a510240cb645a90efc0d