Computer Vision
Task 1-Augmented Reality

Pravin Tulshidas Palve, Matriculation Number-13541620
Manal Ajit Vartak, Matriculation Number-29241614

May 2, 2024

1 Introduction

This task utilizes fundamental concepts in Computer Vision (CV) to demonstrate basic Augmented Reality (AR) applications. We employ AR markers developed by the University of C´ordoba, known as ArUco markers, to overlay a Display Image (DI) [1] onto a wall using an ArUco marker in the Source Image (SI) while matching perspectives and scaling. Later in the report, we will explain the underlying logic of the functions used and showcase final images displaying actual results.

2 Methodology

Choice of programming language was kept as Python considering the availability of state of the art open-source library called OpenCV, which does most of the work for us. Before we delve deep into solution, we will need following libraries `Matplotlib, OpenCV and ArUco` to manipulate, detect, combine, slice, process, convert and plot different data types. Below, we will outline the different components of our solution.

2.1 ArUco Marker Detection

The first problem we dealt with is detection of ArUco markers and extracting information (corner points and id) out of them. This was the easiest part and was done with following functions-

1. `aruco.getPredefinedDictionary()`: This function will import a pre-defined dictionary, in this case we have specified it to get us a 6x6 marker dictionary with 250 markers.

2. `aruco.DetectorParameters()`: It creates a object to store detection parameters.

3. `cv2.aruco.detectMarkers()`: This function is used to detect ArUco markers in an image. It takes input image data, an ArUco dictionary, and previously initiated detection parameters to identify and locate ArUco markers within the image.

The detection mechanism, detailed in the cited article [2], involves image thresholding, contour extraction, perspective projection, image segmentation, binarization, and conditional testing. All this is performed in background and we don't have to worry about it.

2.2 Getting the Transformation Matrix [TM] for Display Image

In this part, we aim to calculate [TM] which will skew and scale the rectangular image to match perspective of ArUco marker in the Source Image. Now we will discuss the functions used along with logic behind.

1. `cv2.getPerspectiveTransform()`: This function takes corner coordinates of DI and ArUco marker (which were generated from previous steps) and calculates the [TM]. Explanation of this matrix calculation can be found in cited lecture notes of Computer Vision [3].

2. `cv2.warpPerspective()`: The actual skewing and scaling is done in this step. Function takes DI, matrix and SI as input arguments and outputs a skewed image with same nature of perspective as of ArUco marker. We will call this 'Transformed Image' (TI) (Refer Fig 1).

2.2.1 Secrete of Scaling

It is important to explain how scaling is done because above method will always map the whole DI on actual size of ArUco marker(which will be too small for us). For this, instead of giving actual corner points of DI as input, we made a small rectangle (shown in red colour) with same aspect ratio inside DI and mapped that to ArUco markers (Refer Fig 1). This way, by changing the rectangle size (x, y) we can change the scaling of augmented DI.

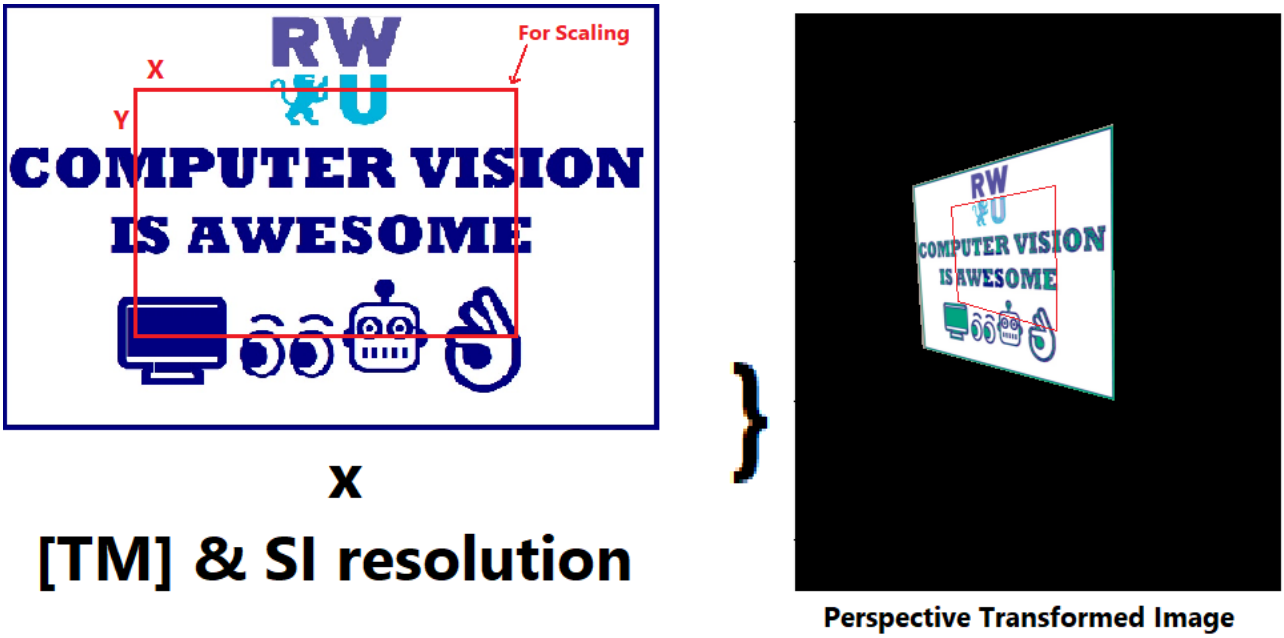

Figure 1: Transformed Image with scaling rectangle

2.3 Masking

We have our Transformed Image (TI). Now, we need to remove the corresponding pixels from SI to create a hollow space, where we can then paste the pixel values from TI. Please refer to Fig 2.

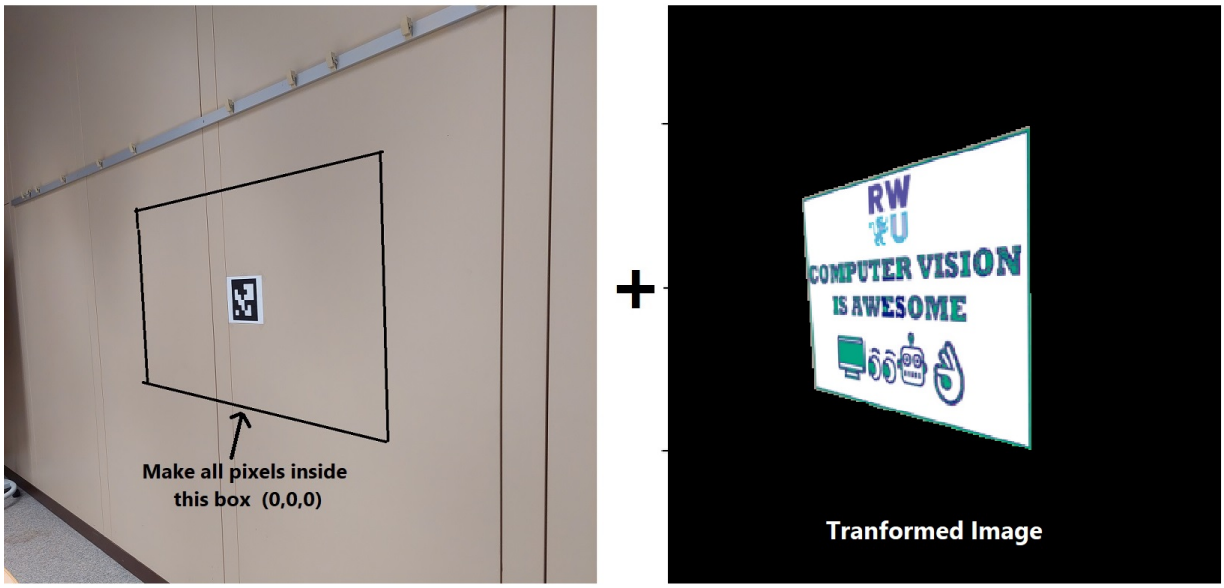

Figure 2: This is what we want to do

1. **Step 1- Binary Thresholding**: In this step, we examine each pixel from the Transformed Image (TI) shown in Figure 2. If a pixel value is not zero, we set it to 255, representing full intensity. This process creates a binary mask where the area corresponding to TI appears white, while the rest of the image remains black, as depicted in Figure 3.

2. **Step 2- Inverting the mask**:Here, we invert every pixel value by subtracting 255, as demonstrated in Figure 3. This inversion is crucial for our next step.

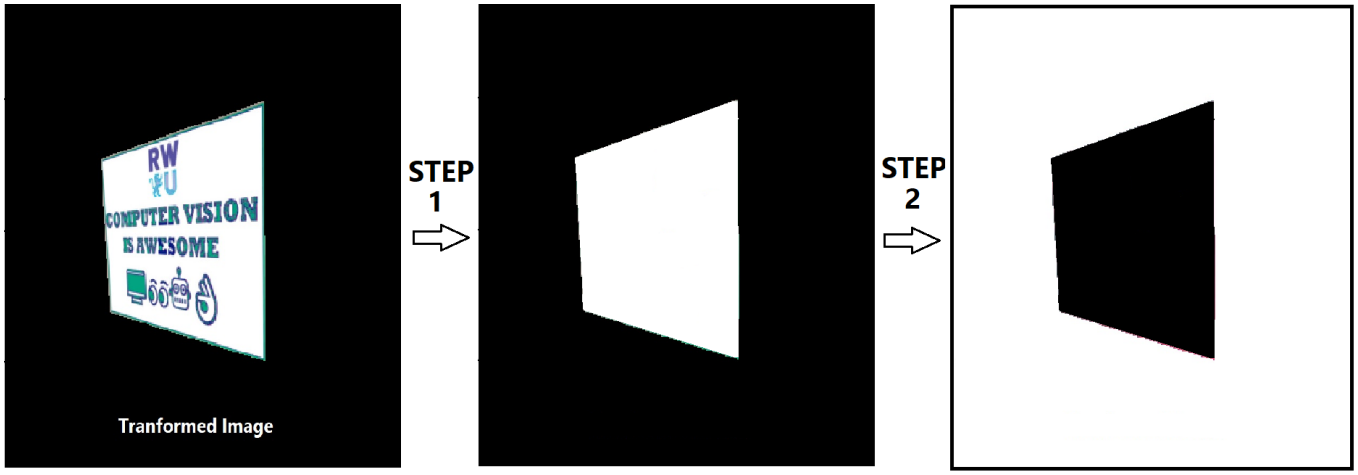

Figure 3: Creating and Inverting Mask

3. **Step 3- Bit-wise AND operation**: This function takes the Source Image (SI) and an inverted mask as input arguments and performs a logical AND operation. This operation retains pixel values from the SI wherever the corresponding mask pixels are white, resulting in a hollowed-out Source Image.

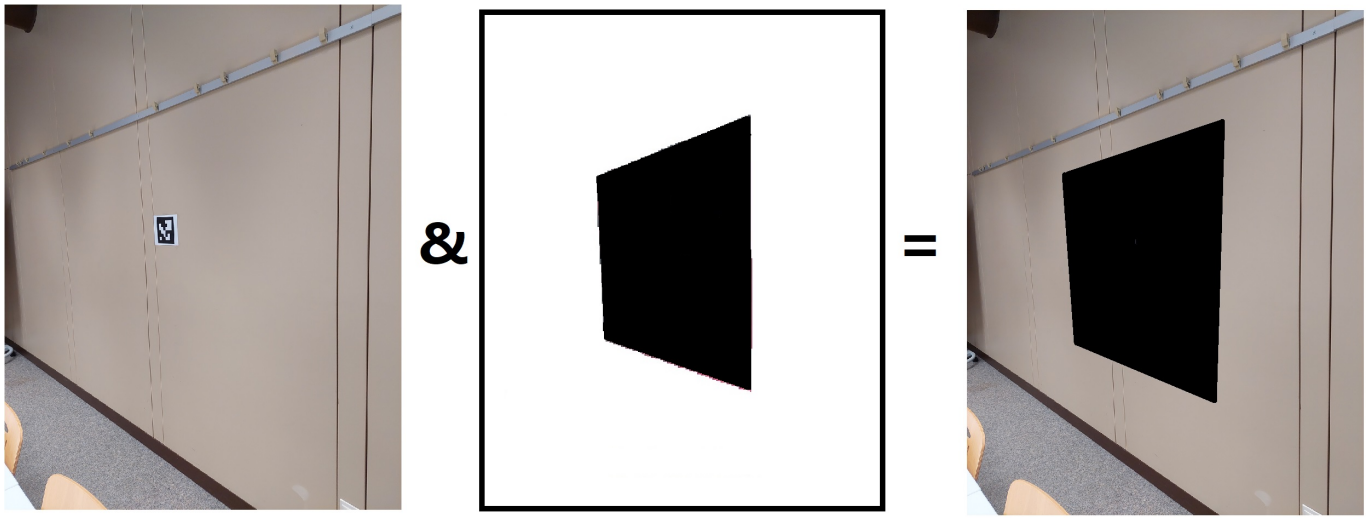

Figure 4: Bit-wise AND operation

2.4 Final Patching

We have got all we need. Now we add Transformed Image (TI) (Fig. 1) and hollowed out Source Image (SI) (Fig. 4) by simply adding corresponding pixel values.

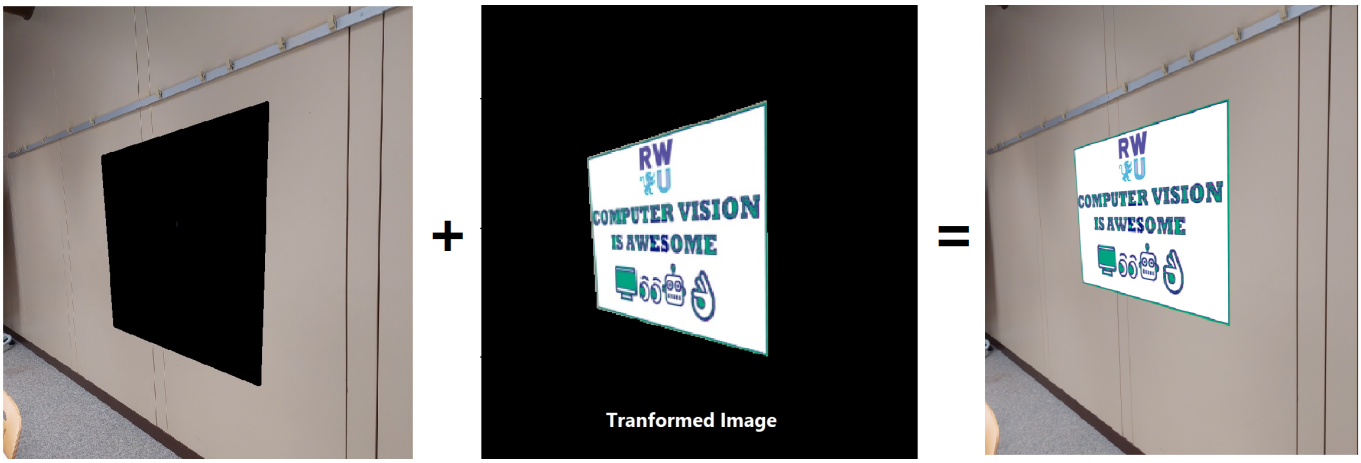

Figure 5: Final Result

3 Results

Now we apply the above-mentioned process to all images. The results are shown below. In cases where no ArUco marker is detected in an image, the corner coordinate array remains empty, leading to an error in Python.

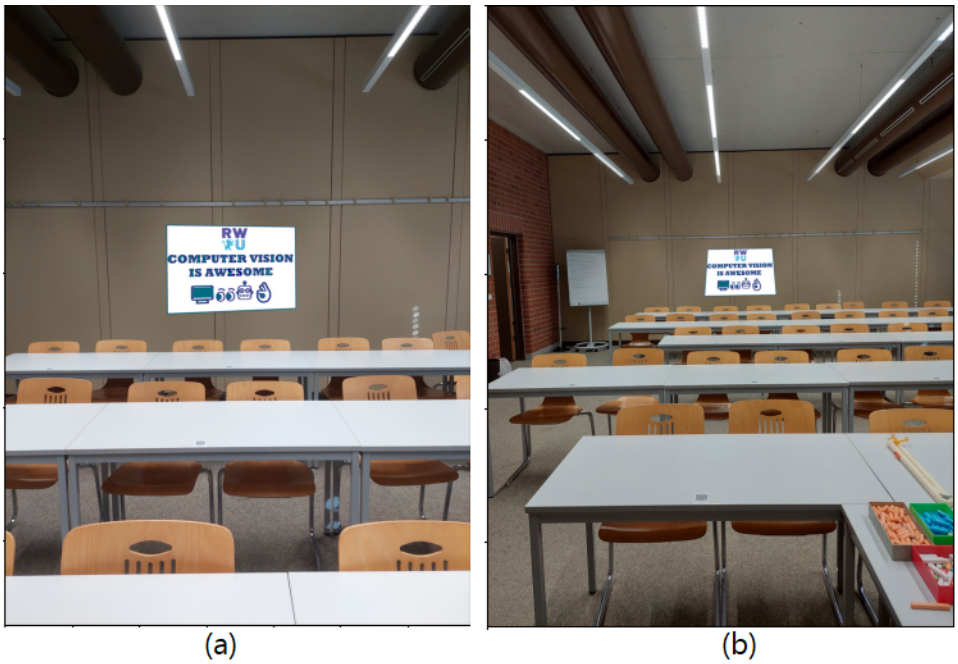

Figure 6

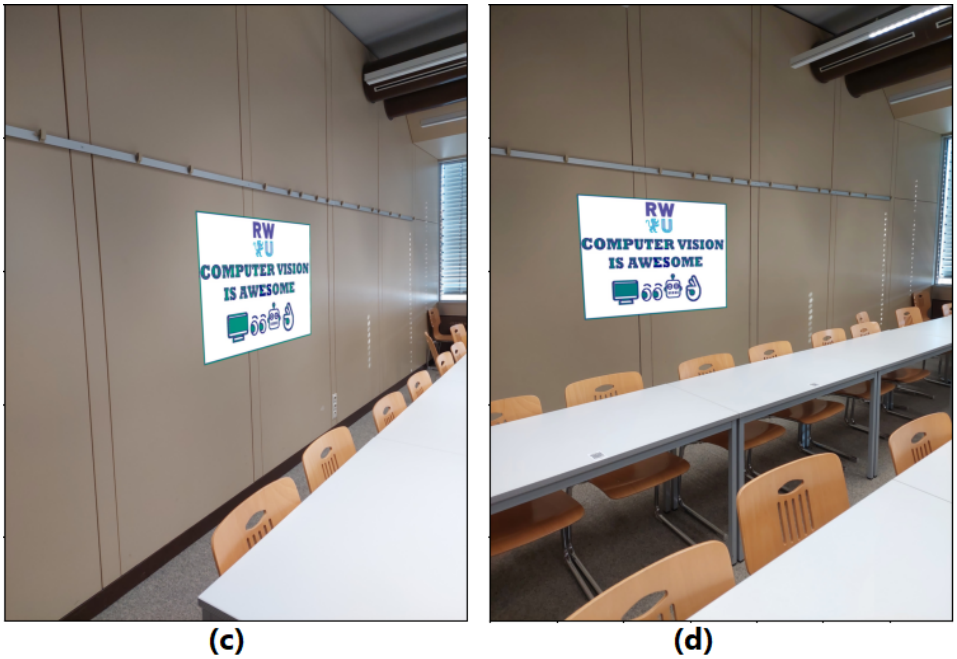

Figure 7

(e) (f)

Figure 8

(g) (h)

Figure 9

(i) (j)

Figure 10

4 Conclusion

On an average, the augmentation works very well. But in case of Source Images where ArUco marker is too small and far from camera, smallest deviation in extracting corner coordinates result in significant perspective error. This can be observed in Figure 6-(b) and Figure 8-(e). Some tiny irregularity is also observed in Figure 9-(g). To ensure that the ArUco markers function reliably, users should ensure that they are large enough to occupy sufficient pixel space in the image. Increasing the image resolution also improves the accuracy of corner coordinate detection for the markers. It is also a good idea to place the ArUco markers in well-lit areas with minimal shadows or obstructions. These things will increase the robustness and accuracy in detection and pose estimations of markers.

5 Further Scope

Any keen-eyed reader would notice that with above explained method, black pixels in the Display Image would also become part of the mask (because of binary thresholding), allowing Source Image pixels to take their place during bit-wise AND operation. In simple terms, area where there are black pixels in Display Image would become transparent in the augmented image, revealing underlying Source Image pixels. This suggests additional opportunities for improvement and/or changes in the methodology we used.

References

[1] Link for Display Image (DI) - https://drive.google.com/file/d/1P533HtUxUtpHFLvbWH5m0BMbJrEUuniD/view?usp=sharing

[2] S. Garrido-Jurado, R. Mu˜noz-Salinas, F.J Madrid-Cuevas, M.J. Mar´ın-Jim´enez Department of Computing and Numerical Analysis. University of C´ordoba. 14071 C´ordoba (Spain)i52gajus,rmsalinas,fjmadrid,mjmarin@uco.es *Automatic generation and detection of highly reliable fiducial markers under occlusion*,

[3] Ravensburg Weingarten University Of Applied Science, Computer Vision- Chapter 4: Geometric Transformation and Simple Model Descriptions, Slide 51