Project Status Report
Visual Inspection Tool

Prepared By – Harsh Shinde
Shardul Nazirkar
Mentored By – Sandeep Joshi

Version 1 -

Comparing images using the Structural Similarity Index (SSIM) with opencv and python:

In order to compute the difference between two images we'll be utilizing the Structural Similarity Index, Image Quality Assessment: From Error Visibility to Structural Similarity. This method is already implemented in the **scikit-image** library for image processing. The trick is to learn how we can determine *exactly where*, in terms of *(x, y)*-coordinate location, the image differences are.

STEPS IMPLEMENTED IN ORDER TO ACHIEVE IMAGE DIFFERENCE:

STEP 1: We import all the necessary packages like compare_ssim from scikit-image, imutils and cv2 (OPEN-CV).

STEP 2: We load the images we want to compare using **cv2.imread** function.

STEP 3: We then converted the loaded images to **gray scale** using the function **cv2.cvtColor.**

STEP 4: Using the compare_**ssim** function from scikit-image, we calculate a function from scikit-image, we calculate a **score** and difference image, **diff**. The score represents the structural similarity index between the two input images. This value can fall into the range *[-1, 1]* with a value of one being a "perfect match". The diff image contains the actual *image differences* between the two input images that we wish to visualize.
The difference image is currently represented as a floating point data type in the range *[0, 1]* so we first convert the array to 8-bit unsigned integers in the range *[0, 255]* before we can further process it using OpenCV.

STEP 5: we threshold our diff image using **cv2.THRESH_BINARY_INV.**
Subsequently we find the contours of threshold image.
The ternary operator the ternary operator **imutils.grab_contours (cnts)**
Simply accommodates the difference.

STEP 6: First, we compute the bounding box around the contour using the **cv2.boundingRect** function. We store relevant (x, y)-coordinates as x and y as well as the width/height of the rectangle as w and h.
Then we use the values to draw a red rectangle on each image with **cv2.rectangle.**
Finally, we show the comparison images with boxes around differences, the difference image, and the thresholded image.
We make a call to **cv2.waitKey** which makes the program wait until a key is pressed (at which point the script will exit).

RESULTS/OUTPUT OF THE IMPLEMENTED STEPS IN ORDER TO ACHIEVE IMAGE DIFFERENCE:

Original Image:

Modified Image:

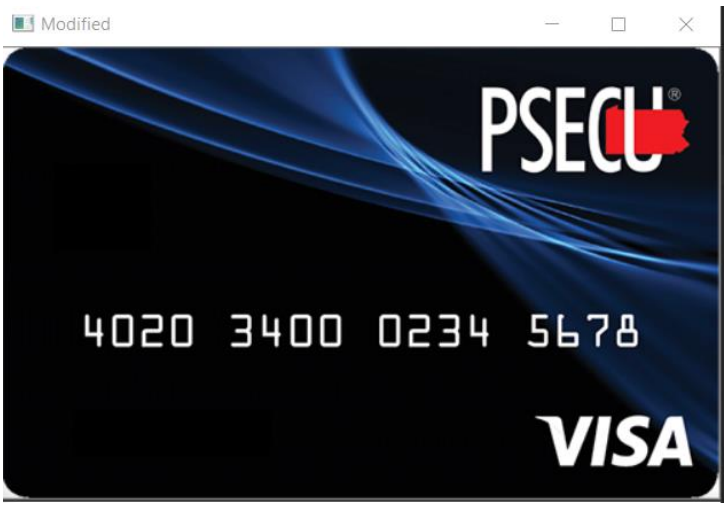


Image difference (Threshold):

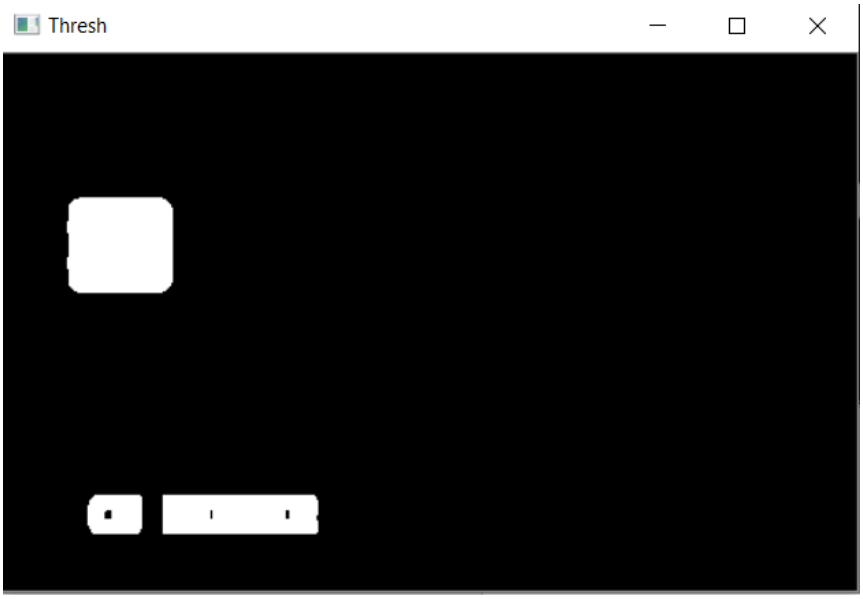
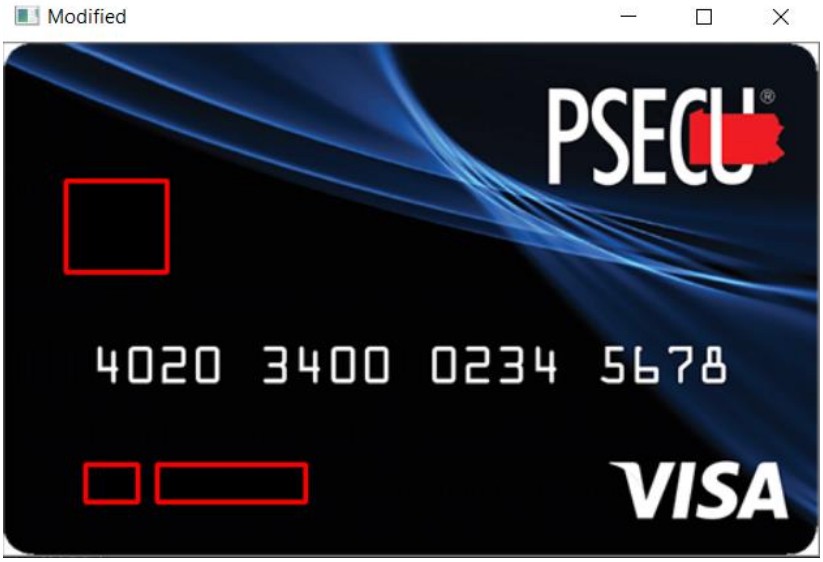
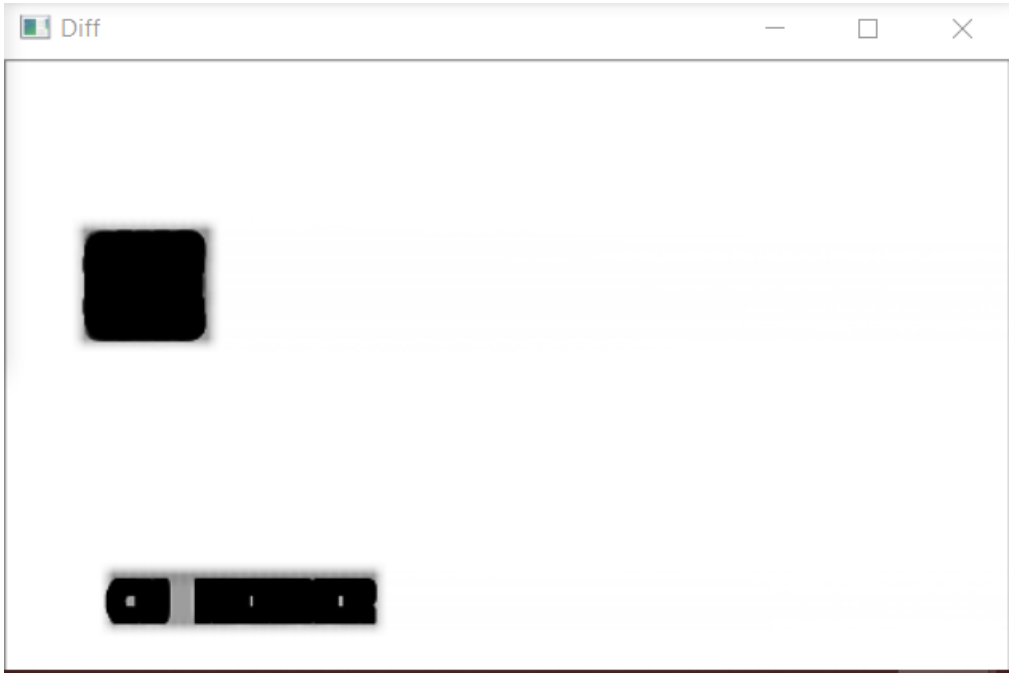


Image difference:





CONSTRAINTS WHILE USING THIS METHOD:

1. Similar pixel dimensions needed.
2. HD quality needed.
3. Edge detection needed.

Eg:

In the example given below edges are detected for accurate results.

4. Images need to be of same format.

Version 2(a) -

Finding biggest contour in the input image and using warp perspective on that contour

To overcome the constraints present in the previous version. We came up with a solution to the problem where the images need to be of same size and same resolution. For this we will use the findContours property and calculate the biggest contour. This contour will be extracted for further use.

STEPS IMPLEMENTED IN ORDER TO ACHIEVE THIS:

STEP 1: Import opencv (cv2) and Numpy library.

STEP 2: Read the input image using cv2.imread().

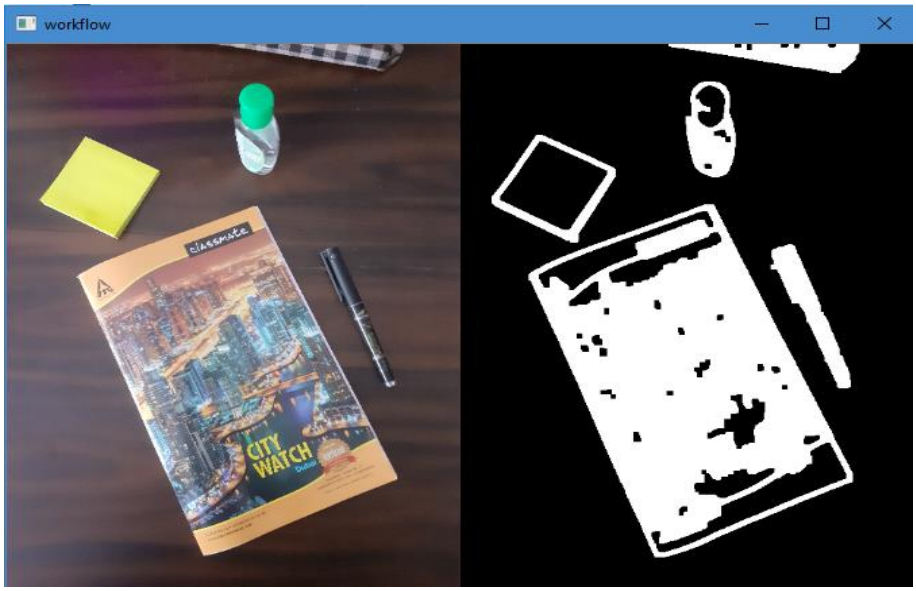
STEP 3: Pass the image through preProcessing() function to get the canny edge image with thresholds.

STEP 4: Pass the gray threshold image through getContours() function to get the biggest contour.

STEP 5: Pass the biggest contour and original image in warpPerspective() function to get the final result image having warped perspective.

RESULTS/OUTPUTS:

INPUT IMAGE AND CANNY EDGED IMAGE



BIGGEST CONTOUR AND WARP PERSPECTIVE(FINAL)

Version 2(b) -

Merging version 1 with version 2(a)

Once we get the warp perspective from above method, we need to compare it with our original master image to find any differences and represent them on our resulting picture.

STEPS IMPLEMENTED IN ORDER TO ACHIEVE THIS:

STEP 1: Merge the two codes.

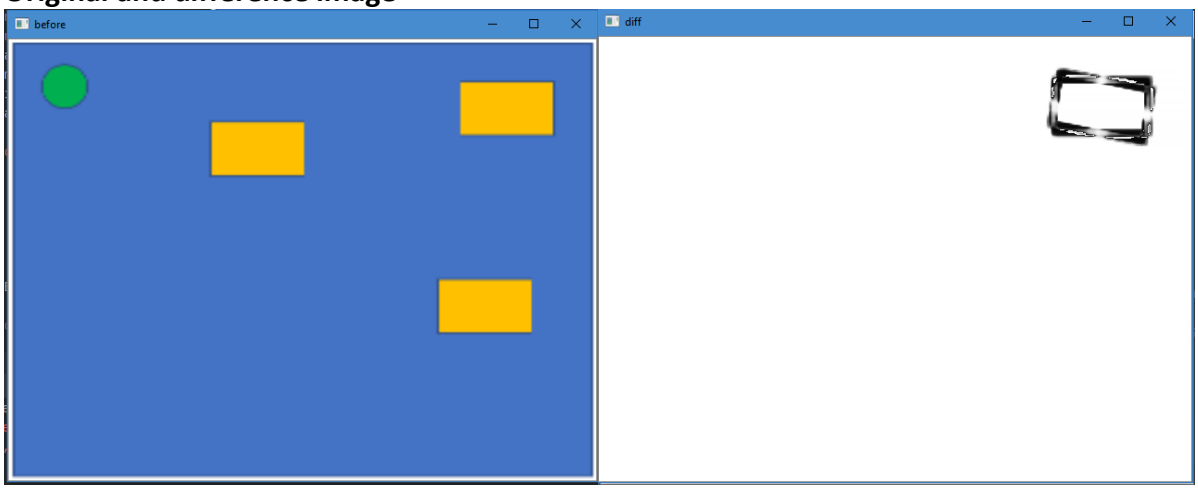
STEP 2: Run the master image and the input image through the version 2(a) process.

STEP 3: Compare the two output images using the code from version 1.

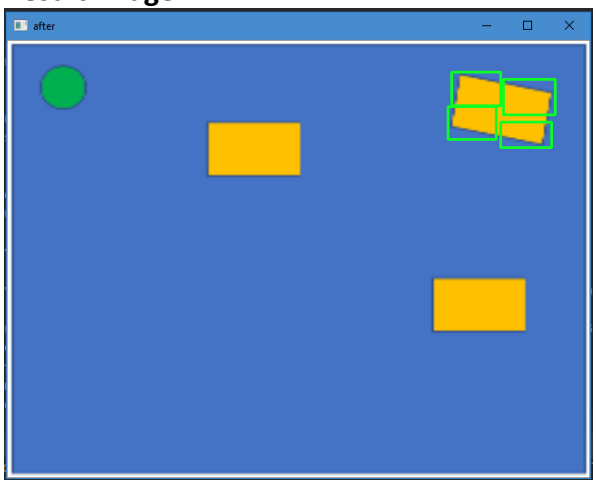
STEP 4: Display the output differences.

RESULTS/OUTPUT :

Original and difference image



Result image



Version 2(c) -

IMPLEMENTING VARIANT BASED SELECTION IN PREVIOUS VERSION

Implementing variant based selection in previous version for ease of use for the user. The user has to enter the serial number(variant number), a path for image selection is automatically generated and the desired results are displayed.

STEPS IMPLEMENTED IN ORDER TO ACHIEVE THIS:

1. Create a file/folder system in the folder where all the images are stored. The name of each folder for each test case is given a specific number.

2. Every folder contains two images. One is the original image and the other is the test image.

3. The two images are passed to the code from version 2(b).

4. The desired outputs are displayed.

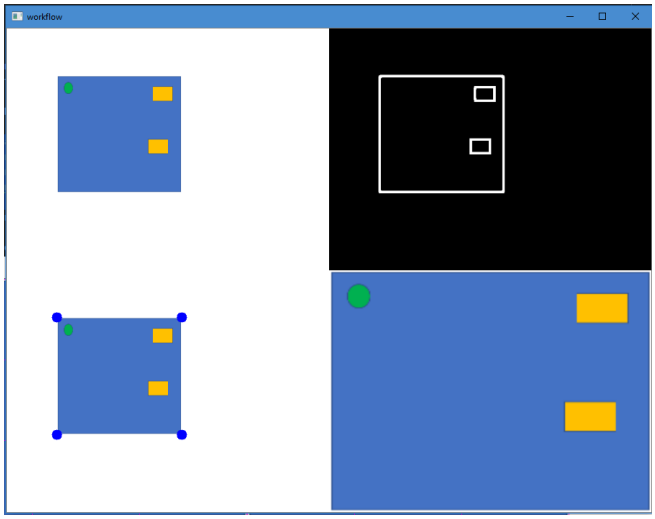
RESULTS/OUTPUT:

Variant selection:

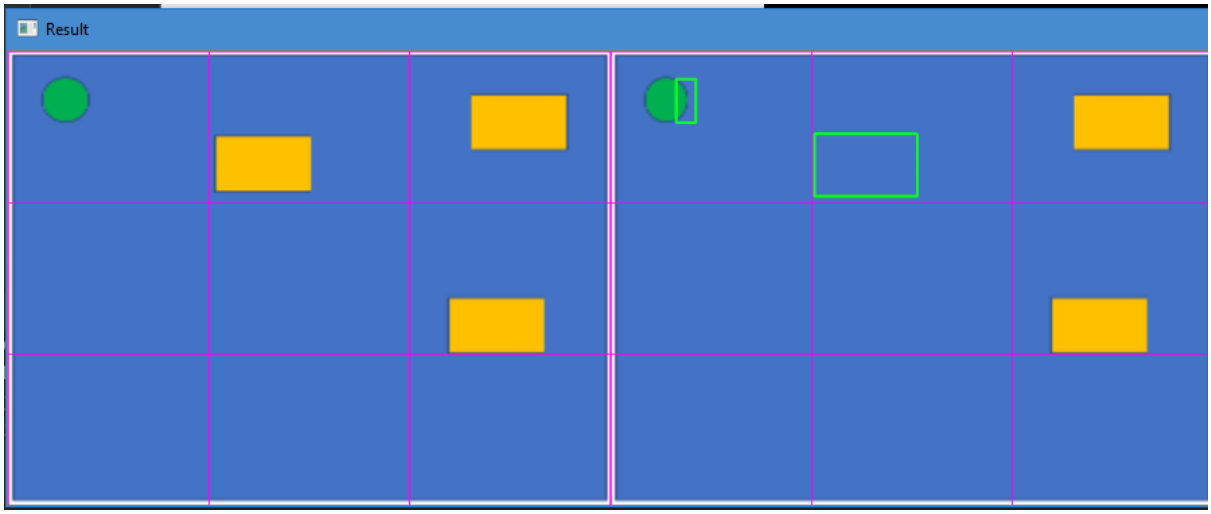
```
"D:\Downloads\Atlas Copco Project\Template Matching\venv\Scripts\python.exe" "D:/Downloads/Atla
Please enter the variant number =1
D:/Downloads/Atlas Copco Project/Visual Inspection Tool/VITool v4.py:118: UserWarning: DEPRECAT
  (score, diff) = compare_ssim(original_gray, test_gray, full=True)

Process finished with exit code -1
```

Workflow:



Result:



CONSTRAINTS in version 2 :

1. Finding contours using canny edge detection is not always efficient and accurate.

2. Images usually have a lot of background noise which makes the object detection process difficult.

   Eg:

   In the above example there is a lot of background noise hence the object detection process was difficult.

3. Works better on HD images.

4. The biggest contour found has to be a square/rectangle to get the warp perspective.

5. Warp perspective works when there is a clear boundary for the square/rectangle contour.

   Eg:

In the above example we are not able to find clear boundary for the given compressor image.

6. The variant selection method works perfectly when all the images are of the same format. It is not perfect when different test cases have different formats of images.

Version 3(a) -

Foreground background separation with semantic segmentation

This method uses semantic segmentation to separate foreground from background, thereby giving us only the object required and subtracting all the unnecessary background present.

STEPS IMPLEMENTED IN ORDER TO ACHIEVE THIS:

1. Import following libraries: torchvision, torch, Numpy, opencv, Matplotlib, pillow.

2. Create an object for the deep labs following class : models.segmentation.deeplabv3_resnet101(pretrained=1).eval()

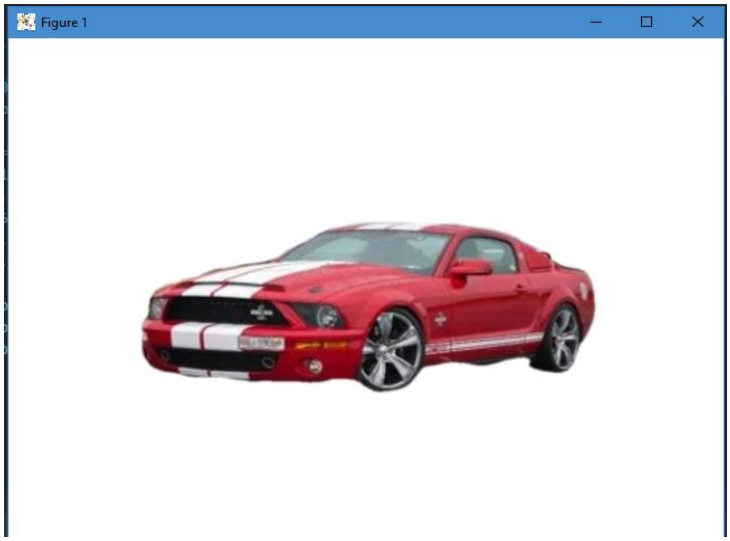
3. Read the image required using opencv commands and pass it to segment function.

4. The segment function using a segmentation and a decoding segmentation map function gives us the required output of only the object required and the background is subtracted.

RESULTS/OUTPUT

Input Image:

Output Image:



CONSTRAINTS:

1. The edges are not smooth and contains unnecessary pixels which affect the further process of visual inspection.

2. This method does not work on all test cases and gives undesired outputs and fails to detect necessary objects from images.

For eg: For the following input image, we get a white screen as a result.

Version 3(b) -

Foreground background separation using GRABCUT method

To achieve the results that we wanted through previous method, we try this new method. In this method a black mask is created over the input image on the unnecessary parts of the image. The black mask filters out these parts and we get the required object minus the background.

STEPS IMPLEMENTED IN ORDER TO ACHIEVE THIS:

STEP 1:We import all the necessary packages like numpy and cv2(OPEN-CV).

STEP 2: We load the images we want to compare using **cv2.imread** function.

STEP 3: We will manually define the coordinates of the face in the image.

STEP 4: We define the bounding box coordinates of the face in the image. These (x, y)-coordinates were determined manually by means of a mouse hovering over pixels in the image and us jotting them down. You can accomplish this with most photo editing software including Photoshop or free alternatives such as GIMP and other apps you find online.

STEP 5: And then we execute GrabCut on the image using **bounding box initialization** on our input.

STEP 6: Before we perform the GrabCut computation, we need two empty arrays for GrabCut to use internally when segmenting the foreground from the background (fgModel and bgModel). We generate both arrays with NumPy's zeros method.

STEP 7: We use GrabCut (timestamps are collected before/after the operation), and the elapsed time is printed. GrabCut returns our populated mask as well as two arrays that we can ignore.

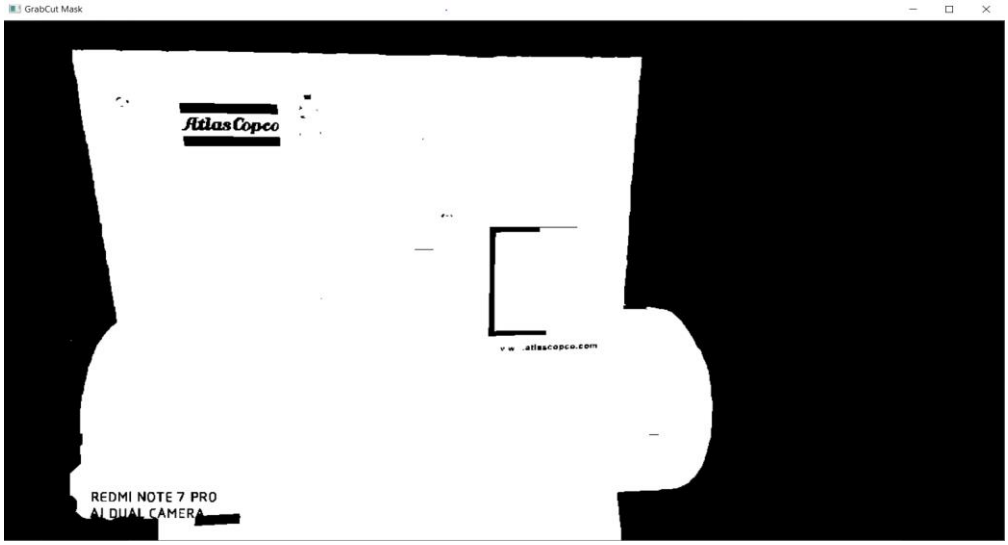
STEP 8: We define possible values in the output GrabCut mask including our definite/probable backgrounds and foregrounds. We then proceed to loop over these values so that we can visualize each. Inside the loop we (1) construct a **mask** for the current value and (2) display it until any key is pressed.

STEP 9: After each of our definite/probable backgrounds and foregrounds have been displayed, our code will begin generating an **output Mask** and an output **image.**

RESULTS / OUTPUT:

Input Image:

Grabcut mask Image:



Grabcut output Image:

Constraints:

1. As long as the algorithm generates a bounding box, you can use it in conjunction with GrabCut.

2. The edges of result are not smooth.

3. Process heavy on the system.

4. In the images displayed above there are parts inside the detected object that also get filtered out due to the grab cut mask which is highly undesirable.

5. Needs a background with very less background noise.

6. Fails to work on some test cases, only partial objects get detected.

For eg: In the below image we can see that our original image is partially detected and the edges are not smooth.

Version 4(a) -

OBJECT DETECTION USING TEMPLATE MATCHING

In this version we will use a new method called template matching to search and detect objects within the input image using pre-defined templates.

STEPS IMPLEMENTED IN ORDER TO ACHIEVE THIS:

1. Read input image and the required templates for template matching.
2. Pass the input image through preProcessing function to get the threshold grayscale image. This helps in reducing the noise to some extent.
3. Pass the threshold image to the template matching function along with the template to be searched.
4. Mark with a rectangle on the image where template is found.
5. Display the output image.

RESULTS / OUTPUTS:

TEMPLATE USED:

Input image:

OUTPUT IMAGE:

Version 4(b) -

DEFINING REGION OF INTEREST ALONG WITH TEMPLATE MATCHING

It is necessary for us to define the region of interest(ROI) for correct working of template matching. Previously, we have seen that defining ROI using Contour method is not very accurate and face lots of issues. Here we use a new method using finding Homography process. In this we use a master template of the ROI and extract exact same area from input image using homography method. Then we can use template matching.

STEPS IMPLEMENTED IN ORDER TO ACHIEVE THIS:

1. Read the input image, master ROI image and the required templates.
2. Pass the input image and master image to preProcessing() function to get the images in desired form.
3. Using Find homography() function we then extract the ROI from input image.
4. We then run template matching from previous version on this image.
5. Display the required result.

RESULTS/OUTPUTS:

Input image:

Master ROI image:

Extracted image from input image:

RESULT:

Version 4(c) -

CORRECT POSITION FOR DETECTED TEMPLATES:

There may come a situation where we would like to check if the templates found are in their correct respective positions.

STEPS IMPLEMENTED IN ORDER TO ACHIEVE THIS:

1. In the previous we make some changes in the template matching function.
2. We add another parameter for an array of correct coordinates for every template.
3. Then we add an if-else condition in which we check if the coordinated found for a template are inside the correct coordinates(which are provided in step 2).
4. If the template is at correct position then a green rectangle is drawn around it saying ‘correct’.
5. If the template is found at wrong position then a red rectangle at its correct position is drawn saying ‘missing’.

Results/Outputs:

The input images are same as previous version.

The new RESULT image:

Constraints:

1. Even if the master ROI image is changed slightly, then a new set of templates are required that work for that particular situation.

2. Correct coordinates are hard coded and have to be changed manually.

3. The code works when a template is present or is at wrong position. The code doesn't work if the template is completely missing. Then we get undesirable outputs.

   Eg: In the example given below the template is displayed at wrong position.