Crouse company

Korosh Roohi's Daily Report

Day 1 – April 4th

First, I read about color spaces in the OpenCV framework. OpenCV has four main color spaces: BGR, HSV, YCrCb, and LAB. This tutorial shows us how to segment colors in different lighting conditions using these color spaces.

In conclusion, The HSV color space works better for segmenting colors in variant lighting conditions because of two properties: having only one parameter for specifying the color and having a compact plot of a single color in different lighting conditions compared to other color spaces.

Then I came to an article [1] about testing vehicle clusters using machine vision algorithms. This article talks about a pipeline of testing the scenario. This article's most essential and relevant part was the software algorithms used for various aspects of visual verifying in a particular cluster.

First, we must check the pointer's position and angular speed. The main algorithm we use here is edge detecting and edge tools (same as bar graph position for fuel), in which we can specify the exact location of the pointer and the zero line. For determining the angular speed of pointers, we need to get the pointer's angle each time we want to measure and use the equation below to calculate angular velocity.

$$\omega = \frac{\theta_2 - \theta_2}{t_2 - t_1}$$

Figure 1 - Angular velocity equation

We must consider the state of being on or off, brightness, and contrast for testing warning signs. We do this test by comparing the greyscale value of on-state with off-state in the region of that sign. Symbols and signs that show on screen must be verified concerning their shape. For this purpose, we train PatMaxTM tools with some templates and check the actual symbols with this tool.

We use blob tools to detect each character and then sort them concerning their sequence in a sentence for text recognition. After that, we recognize each symbol with a pre-trained model called TrainFont.

The discussed approach in another article [2] for measuring needle gauges is the same as the previous one but has some new methods. In this approach, we use cropping, making a binary image, image thinning, and Hough transformation. With Hough transformation, we can specify the needle (or pointer) in the image because it can detect circles. We can calculate the angle between the needle and the selected reference.

This approach gets percentages of white pixels for determining the pixel intensity of signal indicators. We have to obtain a binary image of the pre-located signal indicator for this method

(this approach determines the on or off state of the indicator and can't verify the correct brightness or color)

I learned about the thresholding technique in OpenCV (link) that we can use for separating specific objects with certain attributes from a picture.

At this link, I've seen some good explanations about the pseudocode of a good test for instrument clusters.

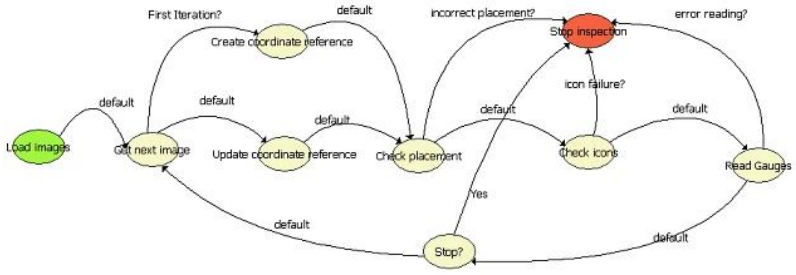

Figure 2 - Pseudocode diagram

Day 2 – April 6th

In an article [3] about instrument cluster diagnosis using computer vision and particularly OpenCV library, I have seen that we can compare two pictures, one as reference (That was from a fine working cluster before) and one from our testing procedure. In this approach, we can use methods like MSE, SSIM, and color channel to compare the matching state between the two pictures.

In this link, I got familiar with an OCR system that uses the tesseract engine provided by Google. In this tutorial, we use the OTSU threshold method, dilating, finding contours method, and cropping system to detect the zone of texts and then recognize them with the tesseract engine.

I have tested the provided code from the previous link with some changes, but I found that this system has a problem detecting the number 7 in seven-segment displays and detecting this as 1.

In this link, We can see a filter that helps us remove noises from an image. However, we can use this approach to detect anomalies in a primary color image for detecting dead pixels. This method depends on the image's resolution that captures a snapshot of our display. This algorithm must know each pixel in the display is illustrated by how many pixels of the image.

I have learned about Hough transform at this link. This link, a document for Hough transform methods in OpenCV, shows how we can extract lines by their properties from an image. Also, we can calculate the angle between the two lines we select.

When our texts are skewed, we can find a bounding box around that and then rotate to deskew that (link). Also, when our text has an angle with our perspective, we can use the Geometric transformation of the OpenCV library. (link, link)

For template matching and finding the location of something special, like warning signs and symbols on clusters, we can use the **matchTemplate** method from the OpenCV library. This method returns the areas of possible matched windows with our template so that we can filter them with a specific threshold. (link)

In some cases, we cannot precisely measure what we want to check; in this situation, we can compare a reference with the sample we are checking. At this link, we can see a **compareHist** method from the OpenCV library that checks if a sample is in our checking base or not. This approach can help us determine whether an object appears on our test or not.

A library called scikit-learn compactly provides machine learning algorithms. In our project, we can use some of the library's algorithms: clustering (for determining some thresholds from a bunch of data), preprocessing, feature extraction, and convolutional filters. (link)

Day 3 – April 9th

One of the major defects on nowadays screens is the Mura defect, which refers to luminance non-uniformity and interrupting the user viewing experience. This defect has many types in the real world and display industry in LCDs and LEDs. (link)

In a mura detection paper [4], two approaches in image processing named SVD and DCT were used to reconstruct the background of our reference image. After that, the differential image extracted can lead us to mura defected in 3 types: point, line, and region.

We can use a linear regression solution for mura detection [5]. In this solution, we use windows for traversing the inspection image. We apply a linear regression equation using a matrix and obtain the hat matrix at every windowed image. After that, we use this matrix to make a fitting image that does not contain the possible mura piece. Then, we use dilation and determine a specific threshold to find the exact pixels that belong to the mura piece. **We can use this approach not only for finding mura zones but on every anomaly that can be appeared on the screen.**

Comparing some machine learning algorithms and four types of LCD screen defects, we can see that BPN algorithms have slightly better accuracy than SVM algorithms. However, SVM has better properties like faster learning speed and a lower possibility of converging in a local minimum regarding efficiency and performance. So, we can propose the SVM algorithm for this type of task if we want to use a machine learning approach [6].

We need data to learn our model for machine learning approaches and algorithms. The best way to create some data for our learning procedure is to create images that present bad situations like crack, dust, non-calibrated color, and mura. Then we can show these images on only one display and get some pictures from that, then learn our model with them.

Day 4 – April 11th

We can categorize display defects into two significant categories: Macro and Micro. Macro consists Mura defect that we explained in previous sections. Micro consists of some little defects the human eye can hardly observe, like Pinholes, fingerprints, particles, and scratches.

To detect these types of little defects, we can use the SVD method and reconstruct an image that preserves the flaws so we can see them [7].

One of our desired tests is that being possible to detect the correct color of the display, which means that we need to capture an image of a monochrome display. For example yellow colored display and confirm that this display is correctly colored and doesn't have any offset in coloring or any other problem. Our purpose of this test is to prevent producing of defective shows in terms of shade. For this, we can define a filter for each testing color, mask our captured image from the screen, and check if that is correct. (link, link)

To provide a good filter for each color, we need to know how each color would represent in our tests. According to the first-day report (link), we must define a threshold for each color. We can obtain this required threshold through some experiments around our desired color in a different situation and displays that show this color correctly.

To implement our code, we must first crop the region of interest of the display so that we only have the pixels, not the display's borders. We can do this with some edge detector algorithms [8].

There is a way that we can find anomalies in displays using statistic modes, like quantiles. This method finds noise and outliers that can be a defect (scratch, dead pixel, particle, and bright pixel) [8].

One of the challenges of using the OCR algorithm is that we are unable to use this algorithm for reading seven-segment characters. However, we can accomplish this objective with this implementation based on the SSOCR algorithm.

In this review, we can see that the Keras-OCR library can perform better for our usage because our environment reads text from images taken from some LCDs.

Day 5 – April 16th

We can detect LCD panel defects by using some outlier detection algorithms in machine learning. We can do this task with other algorithms like CNN, but we have two obstacles to using this approach: hardware constraints and massive data.

Many outliers or anomaly detection algorithms use machine learning and statistical approaches. (link)

We use the Isolation Forest [9] algorithm for our problem because of better accuracy and good performance considering the link mentioned [10].

One of the requirements of our outlier detection is image resizing. We are using 20-megapixel images for our work, so we need to decrease the number of pixels due to the performance and accuracy. OpenCV provides us with the resize method with some helpful interpolation types.

We use INTER_AREA interpolation to decrease the number of pixels in our approach, because we can keep details and focus on them when decreasing the size of the image with this interpolation. (link)

Day 6 – April 20th

I segmented the OCR system into three parts:

First, we need to determine our interest region and extract the sentences or lines from our area. Finally, we need to extract each character from found sentences. After these steps, we need to translate each of these characters into letters and form the correct sentence.

I used this implementation to create the percpectiveFixture method in the project. We can change the perspective of our determined region with this method, but we have to know that we need predefined coordinates, and we can not get these coordinates quickly through the code.

Knowing that the KerasOCR model is very heavy-weighted, we used letsgodigital language for the google tesseract model, which helped us to detect and translate the seven-segment numbers to strings.

The code implemented so far is not suitable for real-world cases and must be configured with the setup of the actual situation. However, selecting an appropriate approach for solving our problem is good. For this purpose, we need to use configurable parameters in implementation to be able to configure settings.

Day 7 – April 27th

We can train a model for our tesseract system if we struggle with our cases' exact format. (link)

There is plenty of option and flags while using the tesseract engine. We can find these options in this link and use them to get better output from this model. Page segmentation modes greatly influence our OCR detection accuracy, and we have to choose them wisely and suitable for our cases. We can read an appropriate tutorial about this subject at this link. Based on this tutorial, it's better to use PSM 10 for single-character modes and PSM 7 for single-line situations in our project.

We can transform Figure 3 to Figure 4 with appropriate preprocessing, improving our accuracy through our OCR procedure.

Figure 3 Sample image before preprocessing

Figure 4 Sample image after preprocessing

After my experiences with different test cases, we have to categorize our texts into three sections: seven-segmented, dot-matrix, and regular. And also, for lighting conditions, we have two modes: light and dark background (if we want to preprocess our image, we need to know our text's background mode).

At this point, we have to sync parameters like: kernel size for dilating and eroding, thresholding method, background lighting, language for tesseract OCR engine, and perspective correcting with our actual situation.

In particular, we can fine-tune our tesseract model to perform better in our scenarios. In this official repository for tesseract, we can find the instructions for fine-tuning the default model.

After some experiments, I've reached out that we can read texts on dot-matrix and seven-segmented displays with suitable preprocessing, which shows that we can accurately read every text on these types of screens with a single method. However, we need some further experiments on our actual screens and provide some accurate data from them to be able for an exact decision about the method for these types of screens.

We can separate preprocessing procedures for each type of screen. For example, we can make a difference in morphological operation parameters for each.

Day 8 – April 30th

In this link, we can see that we can precisely detect colors in images by using HSV color space. We must obtain a range for each RGB color we want to check. After choosing these ranges, we can create a mask to detect zones that are not in that range. We need an actual situation and an isolated space for detecting the ranges and using them for our project.

One of the most useful methods for analyzing colors is drawing the image's histogram. In this method, we can obtain Figure 6 for Figure 5.

Figure 6 Sample image for Histogram

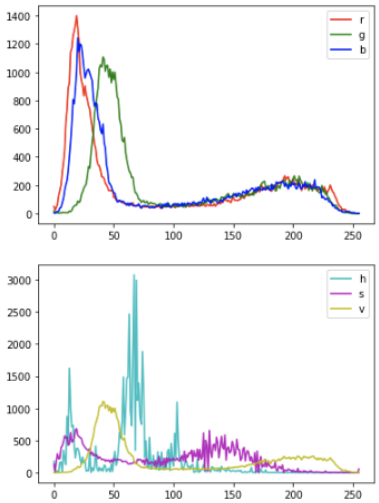

Figure 5 Histograms for Figure 5

As we can see in these two figures, we can learn many things from them and use them to define the ranges for our color verifying procedure.

For example, we have an actual display image from a TFT panel in Figure 7. Its HSV histogram plot is in Figure 8. Our obtained color range based on the plot is (71, 215, 215) to (76, 255, 255).

Figure 7 TFT panel in green color

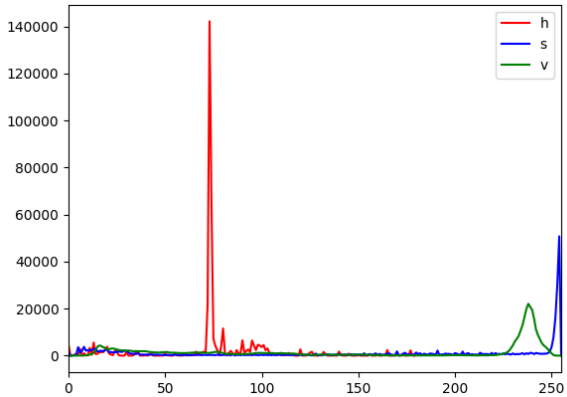

Figure 8 The HSV histogram for Figure 7

As a result of obtained range, we can produce this mask in Figure 9 and determine the percentage of green color in the picture that is 55 percent.

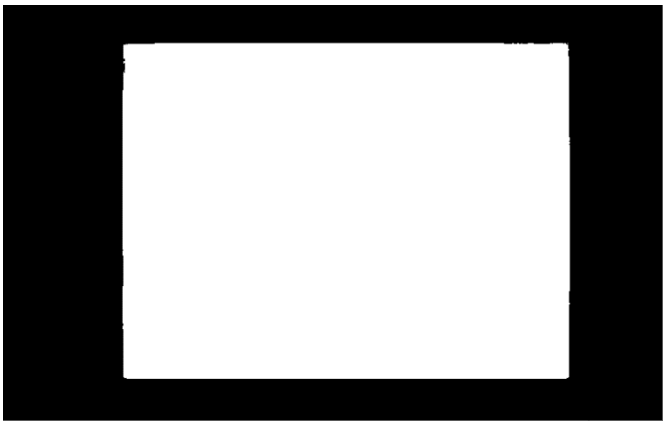

Figure 9 Mask of Figure 7 with the obtained green color range

This link provides a project for training a tesseract model. We need to give a set of paired text and images with our actual situation to fine-tune the default model.

In this thread between tesseract developers and users, we can see that if our character's height is about 30 pixels, we can achieve the highest possible accuracy rate. To accomplish this task, we need to normalize our images with a specific DPI. Also, we need to make a standard for our predefined ROI in test sheets to be able to crop text zones precisely.

I've tested the Super Resolution technique for increasing pixel density by preserving an image's detail. Still, the result was unsuitable for our project because it costs a lot of time for doing this process and can harm the output. In other words, our task is not one of Super Resolution's applications.

Day 9 – May 7th

In this link, we can see that we can use the + operator for combining two languages for our OCR system. We can try fine-tuning and combining two or more languages to get better results.

In this link, we can find a way to strictly fine-tune the eng language trained data. We have to provide our pictures in pairs that contain a picture from our text line and a corresponding text file which is a ground-truth string for the text.

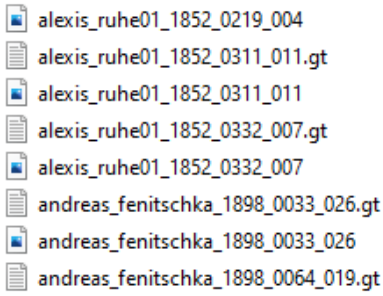

Figure 10 An example of data format for training

I changed the preprocessing procedure for OCR. My main change was in resizing, where I would take one of height or width, and with keeping the aspect ratio, I would resize the image. Also, as I mentioned before, the best height for identifying letters by Tesseract OCR is 30 – 33 pixels, and by using these numbers as our resizing height, we improved our results.

Day 10 – May 14th

Today I want to test the implemented system on some samples I found on the internet. For this test, I have three images that we can see below.

Figure 11 Sample 1

Figure 12 Sample 2

Figure 13 Sample 3

Each of these samples is for a specific type of display. We want to test our OCR system on them and see the result. As I said before, we need our data in our particular situation for better configuration.

In our tests, we still have problems with seven-segment displays, and that is because of our model's lack of samples in this environment. We must fine-tune our model with some seven-segment challenging examples for better results. However, our model is excellently in dot-matrix and standard displays, and the accuracy in these few samples is perfect. Maybe we must do some fine-tuning for some single characters, specifically dot-matrix displays, but our model is in a good state.

In this link, we have a trained model for seven-segment display texts that have good accuracy in our tests if we provide a reasonable and appropriate image for the model.

We can use the + operator in the tesseract framework for making a chain of models we want to use for our OCR system. In our case, I've ordered them by giving the higher priority to the ssd model and then to lets, and at the end, we use the eng model for all texts (*ssd+lets+eng*). I've reached the best accuracy with these settings.

The height of the letters is a critical parameter in the final result. Also, we have to establish a good relationship between the height of a single letter and the applying dilation kernel size to get a better outcome. Over dilating can change the meaning of a letter for our model, and this dilation correlates to that letter's size.

Day 11 – May 25th

Today we set up our camera and captured some valuable images from seven-segment, dot-matrix, and regular screens of automobiles. After that, I prepared and preprocessed the gathered data for the training process for the tesseract system.

According to this link and as I mentioned before, we have to work with the tesstrain module to train new traineddata for the tesseract system. Still, there is no tesstrain version for the Windows OS. Despite installing some modules and packages from Linux on windows, I couldn't run this training procedure on windows. So, we need a virtual box to run this training on ubuntu.

Also, I provide a dataset zip file from today's captures that we can use for forthcoming training and fine-tuning prior models (I had explained the format of datasets before in this report).

Day 12 – May 30th

Today, I worked on running the tesseract training procedure on a Linux virtual machine. First, we must install tesseract on OS with `sudo apt install tesseract-ocr` and `sudo apt install libtesseract-dev`. Also, we have to install GCC compiler for the C program and then install libtool with `sudo apt install libtool`. Then, we must clone the previously mentioned repository of tesstrain into our system. After reaching the project's directory, we can see the help menu by typing the `make` command.

We must put our formatted data (I explained their format and structure in the previous part of this report) in the "data" directory. After that, everything is ready for starting of the training. I recommend the below command for training a model for formatted data.

```
make training MODEL_NAME=name_of_model GROUND_TRUTH_DIR=our_fomatted_data
TESSDATA=.traineddata_file_that_we_want_to_finetune FINETUNE_TYPE=Plus
PSM=6
```

This command is a sample of many possible commands, but we can use this command to start the procedure. After running the training, we must wait for the execution to finish. After that, by running make traineddata, we can have our trained ".traineddata" in two types of best and fast in the corresponding directory in the data folder.

Today, I couldn't transform our formatted file into the Virtual Machine, so I tested this process on ready data from the tesstrain repository just for testing the training system.

References

[1] Huang, Yingping and Mouzakitis, Alexandros and McMurran, Ross and Dhadyalla, Gunwant and Jones, R Peter, "Design validation testing of vehicle instrument cluster using machine vision and hardware-in-the-loop," in *2008 IEEE International Conference on Vehicular Electronics and Safety*, IEEE, 2008, pp. 265--270.

[2] Ren, Tan Wei and Wan Jamaludin, Wan Shahmisufi bin and Lin, Kueh Ying and Mahyuddin, Muhammad Nasiruddin and Rosdi, Bakhtiar Affendi Bin, "Automated Testing of Vehicle Instrument Cluster Based on Computer Vision," in *10th International Conference on Robotics, Vision, Signal Processing and Power Applications*, Springer, 2019, pp. 505--511.

[3] Raj, M Deepan and Kumar, V Sathiesh, "Vision based feature diagnosis for automobile instrument cluster using machine learning," in *2017 Fourth International Conference on Signal Processing, Communication and Networking (ICSCN)*, IEEE, 2017, pp. 1--5.

[4] Yang, Yu-Bin and Li, Ning and Zhang, Yao, "Automatic TFT-LCD Mura detection based on image reconstruction and processing," in *2013 IEEE Third International Conference on Consumer Electronics?` Berlin (ICCE-Berlin)*, IEEE, 2013, pp. 240--244.

[5] Fan, Shu-Kai S and Chuang, Yu-Chiang, "Automatic detection of Mura defect in TFT-LCD based on regression diagnostics," *Pattern recognition letters - Elsevier,* vol. 31, no. 15, pp. 2397--2404, 2010.

[6] Kang, SB and Lee, JH and Song, KY and Pahk, HJ, "Automatic defect classification of TFT-LCD panels using machine learning," in *2009 IEEE International Symposium on Industrial Electronics*, 2009, pp. 2175--2177.

[7] Jiang, Bernard C and Wang, Chien-Chih and Tsai, Du-Ming and Lu, Chi-Jie, "LCD surface defect inspection using machine vision," in *Proceedings of the fifth Asia pacific industrial engineering and management systems conference*, 2004.

[8] Silva, Mateus O and Ferreira, David AO and Ouchi, Kethilen Y and Torres, Gustavo M and Mattos, Edma VC Urtiga and Pereira, Ant{\^o}nio MC and Costa, Luciana R and Cavalcante, Victor LG and Cruz, Caio FS and Silva, Agemilson P and others, "Automated Bright Pixel Detection System on LCD Displays," in *2021 IEEE International Conference on Consumer Electronics (ICCE)*, IEEE, 2021, pp. 1--4.

[9] Liu, Fei Tony and Ting, Kai Ming and Zhou, Zhi-Hua, "Isolation forest," in *2008 eighth ieee international conference on data mining*, IEEE, 2008, pp. 413--422.

[10] Cheng, Zhangyu and Zou, Chengming and Dong, Jianwei, "Outlier Detection Using Isolation Forest and Local Outlier Factor," in *Proceedings of the Conference on Research in Adaptive and Convergent Systems*, Association for Computing Machinery, 2019, p. 161–168.