Separation and Analysis of Surface Level Defects on LCD Screen Samples

Cody Costa   
KLA Masters of Engineering Program  
Department of Electrical Engineering, San Jose State UniversityMilpitas, CA  
cody.costa@sjsu.edu

*Abstract*—This document outlines the process by which surface level defects can be isolated from an LCD image using traditional image processing techniques. In semiconductor manufacturing settings, this process is a critical step in the business model to ensure consistent production quality for the makers of the device. Our analysis and defect recognition are assisted by Python 3 in conjunction with the OpenCV library for its modular and performant code framework, where we explore ways to enhance the base image and tune certain variables to get the best foreground defect separation from the image with binarization methods.

Keywords—foreground, separation, image processing, binarization, python

# Introduction

Semiconductor manufacturing is a lengthy and involved process, one that needs consistent care and monitoring through each step; but alike to all other industries and businesses, production is not without its share of risk and mishaps. Inspection images are routinely captured as the wafer/reticle undergoes each manufacturer's process step (etching, lithography, etc.) to ensure quality is kept to a minimum standard, and my work today uses these image samples as the test subjects for our own defect recognition process validation. Further in the literature, we discuss the image processing techniques used and steps in greater detail, as well as compare the finished product with the specifications set in the methodology.

# METHODOLOGY

## Goal/Specification

Given a sample image, our job is identify, locate, and tally the total number of defects seen in the display.

## Image Processing Procedure

My test image today is a sample slice of an LCD screen with visible surface imperfections:

A brown surface with white spots

AI-generated content may be incorrect.

Defects in this frame can be categorized into 2 distinct subsections: bumps, and craters. We can see the vast majority of imperfections shown are of the bump variety, but the mixed batch means we need to employ slightly different methods/control values for each case. To begin, let’s use some basic image processing techniques to clarify this image and transform it into a state which enhances visibility and makes the foreground separation a bit easier.

First, I converted this color frame into grayscale and used OpenCV’s CLAHE (Contrast Limited Adaptive Histogram Equalization) functionality to create more contrast between the foreground and background. The clip limit defines how much contrast is allowed before clipping occurs. The higher the value, the more enhancement allowed. The tile grid size is the size of each discrete section of the image to apply the CLAHE to. The function will then smoothly combine the results:

|  |
| --- |
| clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8)) |
| gray\_enhanced = clahe.apply(gray) |

This produced the following image:

A grey and white photo

AI-generated content may be incorrect.

Visibility is surely enhanced, but the frame still appears slightly grainy, so next is to apply a Gaussian smoothing filter to reduce some of the noise:

|  |
| --- |
| blurred = cv2.GaussianBlur(gray\_enhanced, (7, 7), 0) |

The (7, 7) snippet describes the filter size, and 0 is the value of sigma. These values are purely experimental and what I found created the best defect recognition results. A 5x5 filter and 9x9 filter were also used with less overall performance. After blurring the photo, I was left with the following image:

A grey surface with white spots

AI-generated content may be incorrect.

We can see the background is more blended, which helps eliminate some of the higher frequency variations in the image that may be falsely treated as defects in the algorithm.

Now that the sample photo has been edited in a more usable fashion, the next techniques described will showcase how each of the crater defects and bump defects get extracted.

Both defect extraction techniques utilize image binarization, another bit of functionality built into the OpenCV library. Binarization is essentially the process of converting a grayscale image into a purely binary image, in which each pixel (normally 0 – 255) can only represent either a 0 or 1, so purely black or white. When doing so, it is important to define a certain binary threshold of interest such that the following equation holds true:

A black background with white text

AI-generated content may be incorrect.

Where T is the defined binary threshold.

Thankfully OpenCV makes binarization very simple with its adaptiveThreshold function, which makes the best possible guess for the binary threshold based on the maximum pixel value (0 – 255), region of interest (called block size) and a value C which represents the value that gets subtracted from the computed local threshold of each pixel. Both the Block Size (N x N segment) and the C value were again experimentally evaluated through trial and error for each bump and crater defect case to yield the best overall foreground extraction:

Bump Defects:

|  |
| --- |
| C = -4 |
| BLOCK\_SIZE = 21 |
| bump\_thresh = cv2.adaptiveThreshold(blurred, 255,cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv2.THRESH\_BINARY, BLOCK\_SIZE, C) |

Crater Defects

|  |
| --- |
| C = 16 |
| BLOCK\_SIZE = 25 |
| crater\_thresh = cv2.adaptiveThreshold(blurred, 255, cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv2.THRESH\_BINARY\_INV, BLOCK\_SIZE, C) |

The value of C for the bump defects is negative because we want to invert the binarization and separate only the lighter values of the image. Contrarily, the crater defect binary image uses a large, positive value to isolate only the very dark craters:

Bump Binary:

White dots in the sky

AI-generated content may be incorrect.

Crater Binary:

A black background with a black square

AI-generated content may be incorrect.

We now have 2 distinct binary images, one for the light bumps, and another for the darkened craters. We can use these images to now trace the contour lines where the binary thresholds occur and concatenate the results into a single image, again using OpenCV:

|  |
| --- |
| crater\_contours, \_ = cv2.findContours(crater\_thresh, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE) |
| bump\_contours, \_ = cv2.findContours(bump\_thresh, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE) |
| contours = bump\_contours + crater\_contours |

With the contour data now mapped, the last step is to plot them onto our sample image. In this case I chose the enhanced, Gaussian blurred test photo to trace the defects upon solely for the added visibility and better ability to visually confirm results. This is achieved with a simple for loop and OpenCV’s drawContour function. After doing so, the final image with mapped defects is produced. Dimensions are expanded to better display contour traces.

A black and white photo of a black and white photo of a black and white photo of a black and white photo of a black and white photo of a black and white photo of a black and

AI-generated content may be incorrect.

# EXPERIMENTAL RESULTS

Recall the specifications outlined in section 2: we are to identify, locate, and count the number of defects detected in the sample image.

The challenge with this work was tuning the sensitivity of the defect detection to match an acceptable capture rate without falsely identifying any otherwise ‘clean’ areas. We discussed how the filter size of the gaussian blur, the adaptive threshold block size and C values all held great influence over the results.

Here are some examples of variations in the input parameters and the results produced. Note I will only be editing and comparing the values of the bump defect parameters. Crater defects had no issues with capture rate:

|  |
| --- |
| Variable Nomenclature: |
| G = gaussian filter size (n x n) |
| C = c value |
| B = block size |

G = (5, 5), C = -3, B = 19, over-capture, false defects

Total defects = 400

A grey and white background

AI-generated content may be incorrect.

G = (5, 5), C = -4, B = 19, over-capture, false defects

Total defects = 167

A grey background with black dots

AI-generated content may be incorrect.

G = (5, 5), C = -5, B = 19, very close, marginal low capture

Total defects = 111

A grey background with black dots

AI-generated content may be incorrect.

G = (5, 5), C = -3, B = 21, over-capture, false defects

Total defects = 444

A grey background with black spots

AI-generated content may be incorrect.

G = (5, 5), C = -4, B = 21, over-capture, false defects

Total defects = 192

A grey background with black dots

AI-generated content may be incorrect.

G = (5, 5), C = -5, B = 21, very close, marginal over-capture

Total defects = 121

A grey background with dots

AI-generated content may be incorrect.

G = (5, 5), C = -3, B = 23, over-capture, false defects

Total defects = 480

A grey background with small dots

AI-generated content may be incorrect.

G = (5, 5), C = -4, B = 23, over-capture, false defects

Total defects = 213

A grey background with black dots

AI-generated content may be incorrect.

G = (5, 5), C = -5, B = 23, very close, marginal over-capture

Total defects = 129

A grey background with black dots

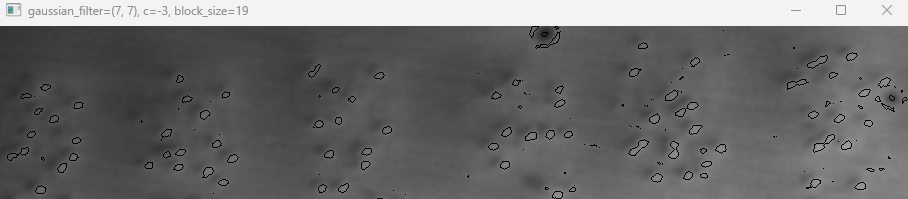
AI-generated content may be incorrect.

The results of the (5 x 5) gaussian filter were rough overall, though a few permutations came very close to accurately capturing all defects on screen, missing just a couple of the darker bump defects, and flagging a few clean regions of the sample image.

Let’s see how the (7 x 7) gaussian mask performs:

G = (7, 7), C = -3, B = 19, over-capture, false defects

Total defects = 159



G = (7, 7), C = -4, B = 19, very close, marginal low capture

Total defects = 102

A grey background with black dots

AI-generated content may be incorrect.

G = (7, 7), C = -5, B = 19, very close, low capture

Total defects = 93

A grey background with black dots

AI-generated content may be incorrect.

G = (7, 7), C = -3, B = 21, over-capture, false defects

Total defects = 179

A grey background with black dots

AI-generated content may be incorrect.

|  |
| --- |
| G = (7, 7), C = -4, B = 21, chosen best result |
| Total defects = 112 |
| A grey background with black dots  AI-generated content may be incorrect. |

G = (7, 7), C = -5, B = 21, low capture rate

Total defects = 95

A grey background with black dots

AI-generated content may be incorrect.

G = (7, 7), C = -3, B = 23, over-capture, false defects

Total defects = 197

A grey background with dots

AI-generated content may be incorrect.

G = (7, 7), C = -4, B = 23, very close, marginal over capture

Total defects = 113

A grey background with black dots

AI-generated content may be incorrect.

G = (7, 7), C = -5, B = 23, low capture rate

Total defects = 100

A grey background with black dots

AI-generated content may be incorrect.

The trend in variable tuning suggests that lower values of C (e.g. -4, -5) avoid the capture of nuisance defects, or very marginal high frequency blips in the sample, but are prone to missing some real defects if the shadow cast by a nearby bump happens to darken the surrounding area, or if the contrast, or size thereof, the defect is not prominent enough to meet the binary threshold.

Contrarily, larger values of C (e.g. -3, -2) tend to have the opposite effect. False defects are routinely identified on clean areas of the sample, though all real defects do get recognized.

Overall, increasing the block size did help with capture rate, even at higher C values, the quantity of false captures did tend to taper off, though was not eliminated entirely. The same effect was seen when increasing the gaussian filter size to (7 x 7).

Evidently, striking a balance with all inputs can be tricky.

# CONCLUSIONS

This experimental analysis has evaluated 18 permutations of foreground extraction parameters in the image processing procedure, each with their own unique results.

My findings highlight that (for the bump defect type) the 7 x 7 gaussian filter size, the 21 x 21 block size, and a C coefficient of -4 provided the most balanced results. This analysis was not 100% accurate, as even in the best case, there are imperfections that are marginally under captured, or over captured in the single digit range.

If we assume that the error margin is in fact single digits (+/- 9 from the capture rate of 112 defects) then we can evaluate the error margin to be:

At an estimated accuracy of about 92%, there is surely still room for improvement in this study. Perhaps some additional pre-processing of the sample image to smooth out any background gradient could help the algorithm detect some of the more marginal, or smaller/dimmer, bumps in the image that were commonly missed.

##### References

1. G. Bradski, “The OpenCV Library,” *Dr. Dobb’s Journal of Software Tools*, 2000.
2. OpenCV Documentation, “cv2.createCLAHE,” OpenCV.org. [Online]. Available: <https://docs.opencv.org/4.x/d6/dc7/group__imgproc__hist.html>
3. OpenCV Documentation, “cv2.adaptiveThreshold,” OpenCV.org. [Online]. Available: <https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html>
4. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018.