Using OpenCV to Create Kanji Calligraphy Database

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

A Thesis Presented To Eastern Washington University Cheney, Washington

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In Partial Fulfillment of the Requirements for the Master of Science in Computer Science

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

By Marco H. Karier

Spring 2018

THESIS OF MARCO KARIER APPROVED BY

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_DATE\_\_\_\_\_\_\_\_\_\_\_

Kosuke Imamura, CHAIR, GRADUATE STUDY COMMITTEE

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_DATE\_\_\_\_\_\_\_\_\_\_\_

Shamima Yasmin, Member, GRADUATE STUDY COMMITTEE

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_DATE\_\_\_\_\_\_\_\_\_\_\_

tbd, Member, GRADUATE STUDY COMMITTEE

Table of Contents

[Abstract 1](#_Toc516941394)

[1. Introduction 1](#_Toc516941395)

[2. Background 3](#_Toc516941396)

[3. Convolution 3](#_Toc516941397)

[4. Gaussian Blur 5](#_Toc516941398)

[5. Sobel Edge Detection 7](#_Toc516941399)

[6. Contour Detection 7](#_Toc516941400)

[7. Methods 11](#_Toc516941401)

[8. Results 12](#_Toc516941402)

[9. Conclusion 21](#_Toc516941403)

[Bibliography 22](#_Toc516941404)

Using OpenCV to Create Kanji Calligraphy Database

by Marco H. Karier

June 20, 2018

# Abstract

The goal of this project was to test the image recognition function of a popular open source program, OpenCV. The research used a particularly challenging set of images based on 7th century Chinese calligraphy to conduct this test. The poor quality of these original images presented a particularly difficult test for this program. The results indicate that the program performed poorly when trying to find matches for sketches of the characters in the database of 207 images. Additional tests indicated that the poor performance could not be entirely attributed to the low quality of the original images nor to the quality of the sketched images. The implication is that alternative methods such as those based on machine learning, may produce better results.

# Introduction

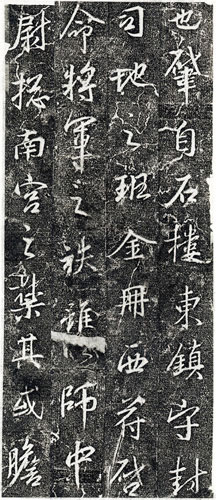
The goal of this project was to evaluate the effectiveness of a popular image recognition software, OpenCV, to identify and match particularly difficult images. For this purpose I created a database of the contours of Chu Suiliang’s 7th century calligraphy to test the accuracy of the *matchShapes()*function in OpenCV. OpenCV is designed for image processing and utilizes more than 2,500 optimized algorithms to analyze images (OpenCV, 2018).

In order to evaluate this function, it was necessary to create a database of images that were capable of being read by the program. Each character had to be digitally smoothed and restored. The images were then processed by OpenCV, creating a database of this particular calligraphy. Finally, each image was input into the program and then compared to all 207 characters in the database.

An exact match, meaning two digitally identical images, is identified by a comparison metric of 0.0 returned by OpenCV matchShapes(). For all other comparisons a low positive value for the metric indicates a higher degree of similarity between characters (Hu, 1962). The precise definition of the comparison metric is explained in Section 6.

An additional purpose of this project was to help historians identify the valuable work of Wang Xizhi. The original work of calligraphy saint, Wang Xizhi, would have been lost forever if it wasn’t for the work of Tang dynasty calligrapher Chu Suiliang. The emperor of the Tang Dynasty requested Chu to carve a stone tablet or *stele* which became known as the “Ji Zi Sheng Jiao Xu” (672 A.D) in the style of Wang Xizhi. To complete his task, Chu spent decades collecting works of Wang Xizhi to make sure that each calligraphy character was done in the same style. Work by Wang can still be found, but the only way to authenticate it is to compare it to Chu’s Chinese stele. The work in this thesis explores the possibility of using contemporary image recognition software to make this comparison efficiently and effectively.

Figure 1: Sample of Chu Suiliang’s Calligraphy



Other methods, such as machine learning, may in the future be able to improve on the matching capabilities documented in this research. However, they will need to demonstrate a better performance than what was found here using, OpenCV. At the time this paper was written OpenCV claims to have 47 thousand users and an estimated 14 million downloads that take advantage of its diverse image processing software.

In the next section I describe details about particular functions in OpenCV that were used in this research. The description focuses on the underlying algorithms that are used by each particular function. Next I describe the steps taken using OpenCV functions to clean the images, find the contours, save the information in a database, and then evaluate the matching function which is named *matchShapes()*. In the confirmation process, I present the results of using each of the images as an input to look for matches in the complete database. However, because I use actual characters from the database there is always one digitally identical image which should produce a perfect match. I also test hand drawn sketches of the characters to see if OpenCV can find the original character. This is essentially what a historian could do with a new character to compare the degree of similarity between the new character and Chu Suiliang’s set of calligraphy.

# Background

OpenCV uses 2,500 algorithms but for my purposes I will focus on a few particularly high-level concepts in order to provide an understanding of how the program processes and compares images. These high-level concepts include convolution, Sobel edge detection, Gaussian blur, and contour detection. These concepts are explained in the following sections.

# 3. Convolution

The basis of many of the OpenCV algorithms is convolution which is best defined as a mathematical operation that uses one function to modify an original function into a third function. OpenCV uses a function defined as a “kernel” to make these modifications. There are different types of convolutions and they are distinguished by different kernels which are represented as fixed size arrays or matrices of numerical coefficients.

OpenCV documentation is a particularly good source for the forms of convolution recommended for use in the program. For example, I used a common 3x3 kernel for two functions, one to blur the image and the other to detect edges (Sobel edge detection). Because these were relatively small images, it was appropriate to use a small (3x3) kernel.

This task is accomplished by sweeping through the image and reducing values to the left of the center and increasing values to the right. For example it is possible to apply the convolution of a 1dimensional array containing [10,10,10,12,17,19,21,21,21] with a 1-dimensional kernel [1,2,1] and for alpha values equal to 0.2 and 0.4. The alpha is used to change the value of the pixels after convolution. Lower values of alpha produce more subtle changes from the original array.

Using 0 array based indexing, the pseudo code for the middle part of the convolution will look like Equation 1.

Equation 1: Representation of Pseudo-Code to Perform a Convolution.

OpenCV will loop through each possible index as represented in Equation 1. Each convolution index calculation includes division by the sum of the kernel elements. This sum is modified for the first and last entry of the array because they lack a match for all three elements in the kernel which causes the program to crash. For this reason the edge values are divided by 3 instead of 4.

To get the base convolution value you subtract the original from the convolution and then multiply by the alpha. This is the basis of many of the OpenCV’s functions that I have implemented.

Table 1 Steps to Perform Convolution



The first column in Table 1 contains the original array values. The second column contains the values after convolution, and the other columns apply different values of alpha to derive a final array.

# 4. Gaussian Blur

A Gaussian blur is a useful technique for blurring sharp edges. This makes edge detection in OpenCV more effective by closing gaps. It is designed to give more weight to the pixels in the center, or anchor point, of the image. The pixels furthest from the anchor point have less weight compared to pixels that are closer. Applying a Gaussian blur removes some of the noise around the Chinese Characters and allows for more accurate Contour detection.

Equation 2 is an example of the 3x3 kernal that I used to apply a Gaussian blur to the original images. Notice that the center value (.18) is larger than the outside values. This ensures that the center point will be more heavily weighed in the process of reweighting the pixel values.

Equation 2: Gaussian Kernel

The Gaussian Kernel is calculated using x and y values (x,y) which are based on the distance from the center. Equation three shows the x and y values for a three by three kernel.

Equation 3: Matrix to Show the (x,y) Values as Distance from the Center

The x and y values are then used to calculate a normal probability distribution using the gaussian formula represented below in Equation 4.

Equation 4: Gaussian Equations as a Function of x and y

In order to calculate the Gaussian value in Equation 4 it is necessary to assume a value for the standard deviation of x and y. OpenCV applies a default calculation based on the kernel size for the width (x) and height (y). This formula is presented below in Equation 5.

Equation 5: Sigma x and Sigma y Calculations (Kaehler & Bradski, 2017)

According to Equation 5, the standard deviation for both x and y in a 3x3 kernel is .95 which was used in equation 4 to calculate the Gaussian values for x and y. Since this is a two-dimensional kernel the Gaussian values for x and y were multiplied to develop the final values for the kernel in Equation 2.

Figure 2 shows a character before it is run through the Gaussian Blur. The spaces in the image make it difficult for OpenCV to detect an edge.

Figure 3 shows the same image but after it has been run through a Gaussian Blur. Notice that the edges are smoother in Figure 3. A blurred image removes the extra noise and improves the effectiveness of OpenCV’s edge detection and Contour Sobel Edge Detection.





Figure 2 Before the Gaussian Blur

Figure 3 After multiple Gaussian Blur

# 5. Sobel Edge Detection

Sobel is a powerful convolution filter that is used to detect edges. Like all convolution operations it has a unique kernel. A 3x3 Scharr kernel for a Sobel edge detection is illustrated in Equation 6 below. In Equation 6, ‘A’ is every x and y coordinate of the 2-dimensional image source.

Equation 6: Sobel 3x3 Kernel (Kaehler & Bradski, 2017)

and

The computer detects an edge by finding the gradient of pixels across the edge, for example a change from dark to light. By applying the Sobel convolution the gradient of an edge becomes more apparent. The pixels that constitute an edge are identified and recorded.

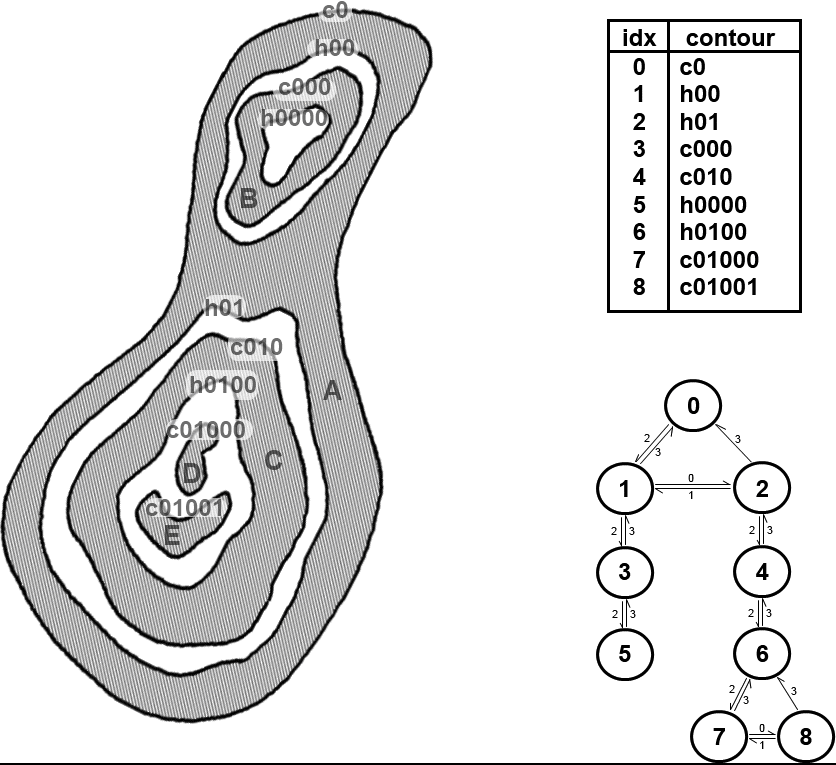
# 6. Contour Detection

A contour is just the outer shape of an image. Contour detection uses Sobel edge detection but also records characteristics of edges such as curves or angles. OpenCV stores a vector of points for each pixel location on an image essentially compiling a list of every edge. OpenCV looks for specific characteristics of edges such as whether the edges form an enclosed shape without any overlap.

OpenCV labels these contours with a specific name depending on whether they are exterior or interior and how deep they reside in the figure. A term, “cX” is used to identify an exterior contour, or outside shape, and “hX” identifies a hole referring to an interior part of the image (Kaehler & Bradski, 2017). The “X” value starts at zero to represent the outside contour and increases place values for contours in the interior of the figure. This hierarchical system provides a standard method for labeling contours in more complex images.

For example, Figure 4 shows how OpenCV labels contours using this system. The outer contour is labeled “c0” and the two holes in that contour are labeled “h00” and “h01”. The contours and holes use 0 based indexing and each digit after “0” shows a new series of holes and contours. This continues for each layer down to “c01001” which is the final contour. The binary tree in the bottom right corner of Figure 4 illustrates the hierarchy used for each hole and contour identified by an index number. The arrows between the nodes in the binary tree is representation of a node in a contour list. The value ‘0’ is the next contour (same level), ‘1’ means previous contour (same level), ‘2’ first child(next level down), and ‘3’ is a parent (next level up).

Figure 4 Image of Layers of contours (Kaehler & Bradski, 2017)



These contours are what OpenCV uses to perform the *matchShapes()*function. Even though OpenCV can use many contour layers, the characters used in this study are defined only by a single outer contour.

To be able to compare two different images, OpenCV uses the significant information about each edge to create a list (Teh & Chin, 1989). Significant information about the contours are called contour moments. Contour moments represent high level characteristics of a contour. The formula that OpenCV uses to calculate contour moments is in Equation 7.

Equation 7: Formula for Image Moments (Kaehler & Bradski, 2017)

In this case, is defined as the contour moment and I(x,y) is the density distribution of pixels in the x,y coordinates. In particular, is “the sum over all the pixels in the object in which the value at x,y is multiplied by (Kaehler & Bradski, 2017)” The moment order for x and y are represented by p and q respectively. These calculations are based on statistical theory where a first order moment represents a mean.

According to the OpenCV tutorial, “if the image is a binary image (i.e., one in which every pixel is either 0 or 1), then is just the area of the nonzero pixels in the image. In the case of a contour, the result is the length of the contour, and in the case of a point set it is just the number of points. (Kaehler & Bradski, 2017)”

The calculated value (), based on specific values of p and q, can be used to compare images and identify degrees of similarity. However, the moments calculated in Equation 7 have some shortcomings when applied to image recognition. For example, they are influenced by both the size and orientation of the images. In order to make these calculations invariant to these factors OpenCV calculates several other types of moments described as central moments and normalized central moments. These enhancements ensure that the moments are invariant to size and orientation.

Finally, OpenCV calculates a series of Hu-moments which are functions of normalized central moments (Hu, 1962). It is ultimately these Hu-moments that are used to compare images. OpenCV employs three methods to compare Hu-moments and calculate a comparison metric for any two images, A and B. These three methods are represented in Table 2. Each of these contour comparisons, I1, I2, and I3, measure the difference between two complex functions (n) where i represents the seven Hu invariant moments. My analysis uses I3 based on initial results.

Table 2 Matching Methods used by cv::matchShapes() (Kaehler & Bradski, 2017)

|  |  |
| --- | --- |
| Value of method | cv::matchShapes() return value |
| cv:: CONTOURS\_MATCH\_I1 |  |
| cv::CONTOURS\_MATCH\_I2 |  |
| cv::CONTOURS\_MATCH\_I3 |  |

OpenCV uses these contour moments to compare the shapes of each contour and determine if it is a similar image. It uses this compare method and returns a number that indicates a degree of similarity. A value of 0 indicates an exact match and is what I expected when using the program to search for a digitally identical image. Users are able to set a maximum level for these numbers when looking for close matches.

# 7. Methods

This research was designed to determine if OpenCV could create a useful database of 7th Century calligraphy and determine if new characters were similar to those in the database. In order to evaluate additional images, one of the requirements was that the program should be able to find the original input image in the database as an exact match. It was also expected to produce a number identifying the degree of similarity to other characters.

The first step was to isolate each calligraphy character into its own image file. I then cleaned them up using different convolution methods described above, including Gaussian and Sobel convolutions to remove noise and restore edges. I grouped significant contours together and removed contours that were clearly noise and not part of the original image. After this was completed, the final contours were saved in the database.

In the example in Figure 5 the character included additional contours that appeared to be noise rather than part of the original image. For this character I removed all but two of the contours, assuming the others represented noise.

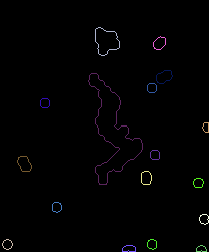


Figure 5: Noisy Image Contour

Finally, I used the *matchShapes()*function to compare each finished image with the entire database. My expectation was that each file would have an exact match with itself indicated by a value of 0.0. Also, low numbers should be returned for similar images and high numbers for dissimilar images.

# 8. Results

Initially I had difficulty getting the *matchShapes()* function to work properly with the calligraphy characters. In order to isolate the problem, I used a data set of characters from the English alphabet. This helped because English letters generally have a single contour with the exception of “i” and “j”. As it turned out, these were the only two letters which OpenCV had difficulty matching. Instead of identifying that “i” and “i” were identical it would return a high number indicating a poor match. This was essentially due to the fact that it was comparing the two initial contours which make up the “i”, a circle and a line, to two other contours, also a circle and a line. The circles matched but they did not match with the lines resulting in a high number indicating a poor match. This required a small adjustment in the program to count only one comparison per contour, i.e. the one with the lowest value or highest match. This successfully resolved the problem.

An important output from *matchShapes()* is the comparison metric which is defined as a number indicating a degree of similarity. It is essentially the value of I3 defined in Table 2. I used this number to create a matrix of results and a subset of that matrix is included in Table 3.

Table 3: A Section of the Result Matrix

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Names | scan1.png | scan2.png | scan3.png | scan4.png |
| scan1.png | 0 | 0.697062 | 0.152959 | 0.372741 |
| scan2.png | 0.697062 | 0 | 0.404698 | 0.243374 |
| scan3.png | 0.152959 | 0.404698 | 0 | 0.431593 |
| scan4.png | 0.372741 | 0.243374 | 0.431593 | 0 |

Each row in Table 3 reports the comparison metric for one image tested against each character in the columns of the database. For example, scan1 is compared to all other images in the first row. This shows a value of 0.0 for itself and a value of 0.697062 when compared to scan2. Alternatively, when scan2 is compared to scan1 in the second row it returns the same value, 0.697062. This is expected because comparison of scan1 to scan2 should produce the same result as comparing scan2 to scan1.

As mentioned earlier, an exact match is indicated by a value of 0.0. Not surprisingly, the program was able to accurately match identical character in the database which results in a value of 0.0 in the diagonal in the matrix. Since the two characters have an identical set of pixels this is not surprising but merely confirms that the program works at this basic level.

Another test was conducted to measure the invariance of the matching routine with respect to orientation and size. I tested the orientation using the English alphabet database by rotating a few letters by 90, 180, and 270 degrees. While matches for the same letters were not exactly 0.0 they were in the range of 10-14 which was effectively zero. Therefore, the comparison was essentially invariant to orientation. The same cannot be said for size. Doubling the size of the characters reduced the ability of the program to identify the original letter. The comparison metric was no longer 0.0 for any of the characters and only 69 percent identified the same letter as the best match. Therefore, the comparison is not entirely invariant to size and I was careful in my analysis to use characters of relatively the same size.

Of more importance is the ability of OpenCV’s *matchShapes()* to find similar and dissimilar characters. To explore this capability, I sorted the matrix to identify the smallest values other than zero to verify that these images were in fact similar. I also looked at the largest values to verify that these images were in fact dissimilar.

Initially, some of the low-value matches did not appear to be similar. I found that this was a problem with incomplete contours which produced unusually low numbers. An example of an incomplete contour is presented in Figure 6. Additional work to improve the contours resolved this problem.

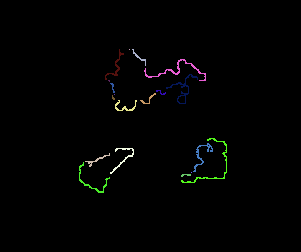


Figure 6: Incomplete Contours

Because there are 207 characters there are 42,642 comparisons in the complete matrix, not counting the comparisons to themselves. Because half of these are duplicates, (i.e. scan1 to scan2 is the same as scan2 to scan1) there are 21,321 unique comparisons. I then found the seven lowest values (other than 0.0) and those are presented in Table 4.

Table 4: The lowest values for the comparison metric (other than a perfect zero match)

|  |  |  |
| --- | --- | --- |
| Scan192 & scan187: 0.0154983 |  |  |
| scan155 & scan189:  0.01878 |  |  |
| Scan189 & scan194:  0.023818 |  |  |
| Scan37 &  Scan 107:  0.030625 |  |  |
| Scan197 &  Scan26  0.032658 |  |  |
| Scan37 &  Scan99:  0.032942 |  |  |
| Scan40 &  Scan20:  0.034481 |  |  |
| Scan8 &  Scan18:  0.036186 |  |  |

The first three matches in Figure 4 (scan192/187, 155/189, and 189/194) are obviously strong because they involve the same characters that were entered as different images from the original stele. They are only slightly different in size and shape because it is impossible to carve stone characters with exact precision. These three matches had the lowest values (other than 0.0) and were all less than .025.

The next four matches in Figure 4 are ranked as the next best matches but are obviously dissimilar. They all have values of 0.03 and higher.

I also sorted the matrix to find the three largest numbers indicating the highest degree of dissimilarity and those characters are presented in Table 5. Visual inspection confirms that the images in Table 5 are not particularly good matches and they have values greater than 12.

Table 5: The most dissimilar comparison

|  |  |  |
| --- | --- | --- |
| scan103 & scan166:  12.7103 |  |  |
| scan103 & scan156:  12.4833 |  |  |
| scan103 & scan170:  12.3566 |  |  |

A key question is how useful OpenCV’s *matchShapes()* is for searching the database for particular characters. Given a new character, can OpenCV find one similar in the database? In order to test this capability, I sketched twenty characters based on the original unprocessed images. Three images are presented in Table 6 for each of the 5 examples: the sketch, the associated database image, and the original image which served as a model for the sketch. Although the sketches are not perfect reproductions they are visually identifiable as similar images. The major differences between the sketch and the original is that the original lacks smooth and continuous lines.

For this analysis, it is useful to rank the comparison metric from low to high with the best match represented by the lowest number. OpenCV’s *matchShapes()* successfully “finds” the similar image when the rank is one. The higher the rank, the lower the performance.

Table 6:Hand Drawn Calligraphy, Database Contour and original Characters.

|  |  |  |
| --- | --- | --- |
| Character #37 Rank = 19, Value = 0.3763 | | |
|  |  | A close up of a logo  Description generated with very high confidence |
| Character # 51 Rank = 19, Value = 0.1188 | | |
| A close up of a logo  Description generated with very high confidence |  | A picture containing gauge, object  Description generated with high confidence |
| Character #18 Rank 60, Value = 0.376 | | |
|  | A close up of a coral  Description generated with high confidence |  |
| Character #39 Rank = 60, Value = 0.367 | | |
|  |  |  |
| Character # 48 Rank = 60, Value = 0.850 | | |
| A close up of a logo  Description generated with very high confidence |  |  |

Based on this test, OpenCV’s *matchShapes()* performed poorly. Not one single pair of characters in Table 6 had a rank of 1. In fact, the best rank for all 20 sketches was for characters #37 and #51 which rank 19 out of 207. In other words, OpenCV’s *matchShapes()* found 18 images with lower values than the original images the sketches were based on. Despite additional efforts modifying the methods for cleaning the images and conducting the comparison, the results did not improve.

This is an important finding and raised another question. Did OpenCV’s *matchShapes()* perform poorly because of the poor quality of the original images? To explore this question I decided to run the same test with the English alphabet based on the “Segoe UI” font. I used OpenCV to create a database of all upper and lower case letters. I again made sure that OpenCV’s *matchShapes()* would identify identical matches with a comparison metric value of 0.0.

Similar to the calligraphy test, I sketched each of the 52 characters and searched for them in the English alphabet database. Five of these sketched and original images are presented in Table 7.

In this case, OpenCV’s *matchShapes()* performed better. 12 of the 52 had a rank of 1 indicating that it found the original letter 23 percent of the time.[[1]](#footnote-1) But this means that even with a perfect set of originals, OpenCV failed to find the original images 77 percent of the time.

Table 7: Sketched and Original English Alphabet

|  |  |
| --- | --- |
| A close up of a logo  Description generated with high confidence | A close up of a sign  Description generated with high confidence |
| A close up of an animal  Description generated with very high confidence | A picture containing clipart  Description generated with high confidence |
| A close up of a logo  Description generated with very high confidence |  |
| A close up of a logo  Description generated with high confidence | A picture containing clipart  Description generated with very high confidence |
| A close up of an animal  Description generated with high confidence |  |

This result led to yet another hypothesis. Perhaps OpenCV’s *matchShapes()* doesn’t work well because of the quality of the sketched characters which were used as the test images in both the Chinese calligraphy and the English alphabet. One way to test this is to print the English alphabet with a simple ink jet printer, scan the printed images, and then search for them in the database. The printed images are much higher quality than the sketched images and should produce better results. And yet the results of this test were also less than perfect. Only 22 of 52 characters had a rank of one which represented 42 percent of the images tested. This result was better than the one using hand drawn characters but still fell short of 100 percent. Therefore, even using high quality images as inputs to search the database did not produce consistent matches. The use of hand drawn letters does not appear to be the only reason for OpenCV’s *matchShapes()’s* poor performance.

# 9. Conclusion

I was successful in my efforts to create a searchable database of a particular 7th Century calligraphy. This required scanning and processing the original characters to create a set of useable individual images. I confirmed this by comparing each of the images to the entire database. Exact matches were confirmed with a value of 0.0, good matches were less than 0.025, and poor matches ranged as high as 12.7.

However, when OpenCV’s *matchShapes()* was tested to find sketched characters in the calligraphy database it performed poorly. None of the sketched images had a rank of one based on the comparison metric produced by OpenCV’s *matchShapes()*. In fact the best result was a rank of 19 out of 207. If the intent is to find and compare similar characters in the database, OpenCV as it currently exists will not be particularly useful.

Additional tests with the English alphabet suggest that some of the poor performance of OpenCV could be attributed to the low quality of the original images. Improving the original images increased the success rate to 23 percent. In addition, increasing the quality of the test images using scanned letters increased the success rate to 42 percent. It appears that improving the original and test images improves the success rate significantly but still misses about 58 percent of the comparisons. This remaining failure rate may be attributable to OpenCV’s performance.

OpenCV uses sophisticated algorithms to compare images based on Hu moments derived from centralized and normalized moments. The results from this work indicate a relatively low level of performance. Alternative image recognition methods need to be developed. My work with OpenCV’s *matchShapes()* shows that it sets a relatively low bar, opening the possibility for future methods to surpass this level of performance.

# Bibliography

Bradski, G., & Kaehler, A. (2008). *Learning OpenCV.* O'Reilly.

Hu, M.-K. (1962, February). Visual Pattern Recognition by Moment Invariants. *IRE Transactions on Information Theory, 8*(2), 179-187.

Kaehler, A., & Bradski, G. (2017). *Learning OpenCV3: Computer Vision in C++ with the OpenCV Library.* O'Reilly Media, Inc.

Puzicha, J., Malik, J., & Belongie, S. (2002, April). Shape Matching and Object Recognition Using Shape Contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 24*(24), 509-588.

Suzuki, S., & Abe, K. (1985). New Fusion Operations for Digitized Binary Images. *IEEE*, 638-651.

team, O. (2018, 05 28). *https://opencv.org/about.html*. Retrieved from https://opencv.org/: https://opencv.org/about.html

Teh, C.-H., & Chin, R. T. (1989, August). On the Detection of Dominant Point on Digital Curves. *IEEE Transactions on Pattern Anaylsis and Machine Intelligence, 11*(8), 859-872.

1. A rank of one is also assigned if the lowest value is associated with a lower case or upper case letter when the two are identical. The same rule was applied for identical letters such as capital “i” and lower case “L”. [↑](#footnote-ref-1)