QR Code Reader Project Report

Wenwen (Mia) Chen

Abstract

In this project, a QR code reader is the target. The reader needs to input the image, scan the image, then detects the position and size of the QR code on the image, reading and decoding the code, and then output the QR code information.

Introduction

This project mainly focuses on detecting the position of the QR code. The result of detector I wrote will be compared with the QR code detector built in cv2 [1]. The decoding is completed by the cv2 QR code detector. Compared with my project outline, the main challenge is when I research for the logic of decoding, I found that the logic of QR Code is too complicated [3] and it has to get data pixel by pixel, which is not too much relative with computer vision. Therefore, I paid more attention on making the detector more accurate. The output of this project shows where the QR code is and draws squares that help to detect the code.

Background

QR codes can be seen everywhere in our life. We can see them on cards, posters, websites, and product packages. Using the QR codes is one of the most interesting ways of digitally connecting consumers to the Internet through mobile phones, because mobile phones have become a necessity for everyone. [4] This is a 2D matrix code, which is designed by considering two points. Compared with 1D bar code, it can store a large amount of data. QR code provides many advantages, such as high data storage capacity, fast scanning, all-round readability, etc. [5]

Methodology

The programming language used for this project is Python with libraries cv2 version 4.5.5.62, random, os and NumPy.

In order to measure the position and size of the QR code, the double squares from the three corners of the code are the targets to be detected. The computer technology used in the QR code detection is contour detection.

The procedure of contour detection is reading the image, gaussian blur, Otsu’ s thresholding, morph closing, then detect the contours of image. [8] This report uses image “7209-v4.png” to show the changes on each step. After reading the image, the detector gets Figure 01.

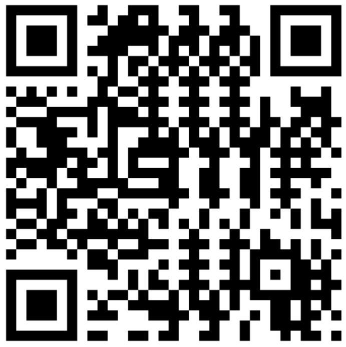


Figure 01. Original image 7209-v4.png

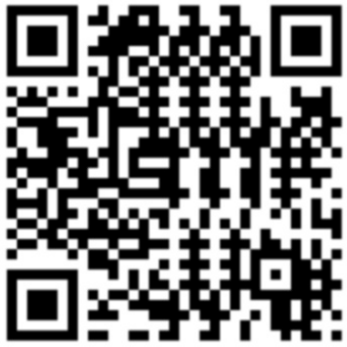


Figure 02. Blurred 7209-v4.png

A gaussian blur smoothens the pixels of the image, making the changes less intense. The purpose of blurring is to reduce noise by removing high-frequency components in signal. [9] The cv2 gaussian blur method is used and outputted Figure 02.

Otsu's threshold converts the gray image to a binary image, which makes the threshold image consist of only two peaks (0 and 255) to enhance contrast and reduce noise. [10] The Otsu's threshold is applied by the cv2 threshold method by inputting cv2.THRESH_OTSU as maximum value to use with the THRESH_BINARY_INV thresholding types. [11] The output image of thresholding shows on Figure 03.

Morph Close shows on Figure 04 is used to close the holes inside the foreground objects or small black spots on it. It has the effect of smoothing the image through the closing operation of adding pixels. [12] The deformation kernel is obtained by CV2 get structuring element method, and then it is applied to the threshold image by cv2 morphologyEx.

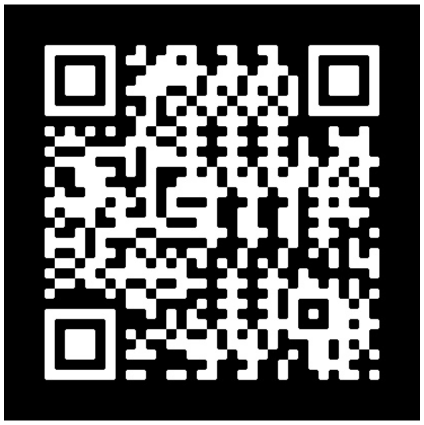


Figure 03. Otsu's threshold

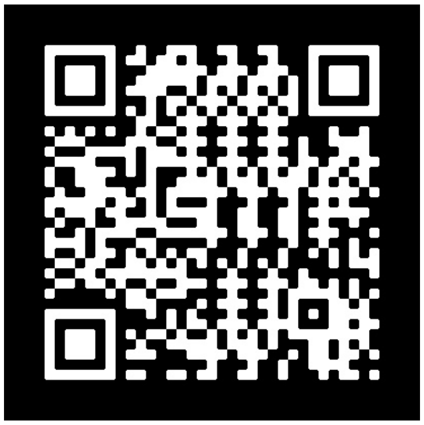


Figure 04. Morph Close

This detector finds the code contours by the cv2 findContours method. There are three parameters in findContours() function. The first one is source image, and our input is the image that has been blurred, Otsu's threshold and Morph closed. Second, the contour retrieval method, and third, the contour approximation method. Enter the RETR_EXTERNAL and CHAIN_APPROX_SIMPLE into these two parameters, which could be referred to the example on OpenCV webpage. [13] It will output contours and hierarchy. The detector collects the contours as Figure 05.

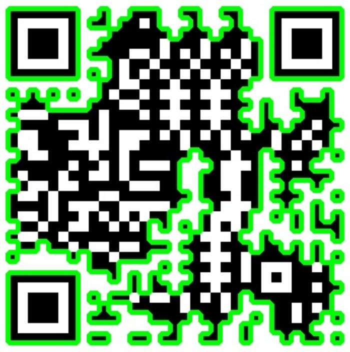


Figure 05. Contours

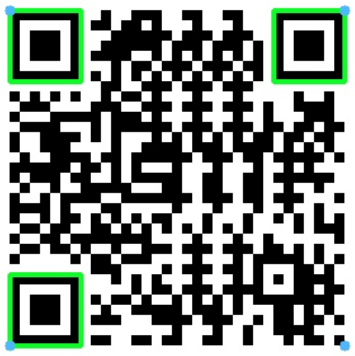


Figure 06. Result

After finding all contours of the QR code, the detector applied some methods to find the features of each counter, such as the arc length, area, and rectangle, by reducing the number of vertices. The top-left point, width, and height of the found rectangle are established by the boundingRect() method of cv2 [14]. With this information, the detector filters them to find the double squares. If the detected polygon is generated by 4 edges, then it's a rectangle. It also filters out small-sized rectangles, because the detector only looks for polygons with an area greater than 1000 pixels. It is determined that a polygon is a square by measuring that the ratio of its height to its width is close to one. Then three squares are detected, and the 4 corners of the QR code can be easily found. The result of this step shows on Figure 06.

These corners are enough for reading code data by cv2 decoding method. However, there are still some errors, such as failing to find a square. This kind of error can be reduced by comparing the X and Y values of each angle and taking the maximum or the minimum value which is depending on the position.

The last step to generate some test case. The testing QR code images are picked randomly from Kaggle QR code dataset [2]. These images are saved in folder "./qr_dataset" . Random number of QR code images are randomly selected from this folder for testing. The number of images is around 10 to 15. The detecting results are compared with the cv2 detectAndDecode() method.

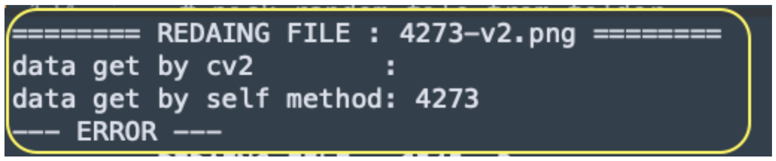
Result

By running 850 test cases, such as the data on Table 01, the success rate of the project detector is 99.06%, and that of the cv2 detector is 99.76%, which are so similar to each other.

Project detector success times	cv2 detector success times
842	848

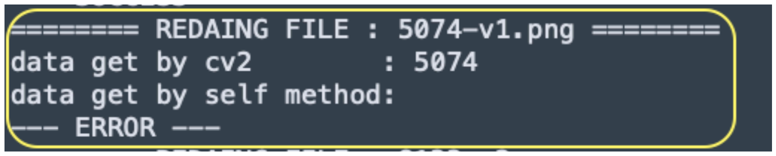
Table 01. Success times in 859 test cases

The image "4273-v 2.png" shows an example in which the project detector succeeds but the cv2 detector fails, and the output information is shown as Figure 07. An example of the failure of the projector detector is reading image "5074-v 1.png", as shown on Figure 08. When both methods are successful, the situation is shown in Figure 09.



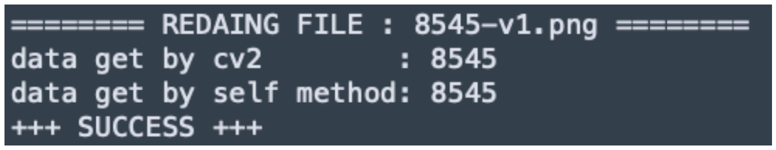
```
======== REDAING FILE : 4273-v2.png ========
data get by cv2         :
data get by self method: 4273
--- ERROR ---
```

Figure 07. CV2 Fail



```
======== REDAING FILE : 5074-v1.png ========
data get by cv2         : 5074
data get by self method:
--- ERROR ---
```

Figure 08. Project Detector Fail



```
======== REDAING FILE : 8545-v1.png ========
data get by cv2         : 8545
data get by self method: 8545
+++ SUCCESS +++
```

Figure 09. Both Success

When reading image "4737-v2.png", the corner error correction is really effective. In Figure 10, the upper left square is not well detected, but the upper left corner of the QR code can still be found by comparing with the position of the upper right square.

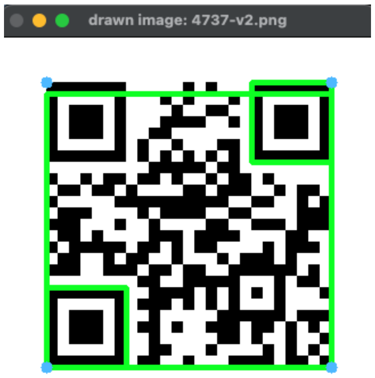


Figure 10. Error Correction

Discussion

The detector of my project could read the QR code with correct shape, and there are no other graphics in the image. By comparing with the CV 2 detector, the accuracy rate is good.

To improve the detector, the detector should read the QR code with a certain degree gradient or a transformative code. Better yet, the detector could find the QR code on the camera by ignoring the background.

Reference

[1] https://docs.opencv.org/4.5.2/de/dc3/classcv_1_1QRCodeDetector.html

[2] https://www.kaggle.com/coledie/qr-codes

[3] https://www.maketecheasier.com/how-qr-codes-work/

[4] https://ieeexplore.ieee.org/abstract/document/5692920

[5] https://ieeexplore.ieee.org/abstract/document/7966807

[6] https://spectrum.ieee.org/top-programming-languages/#toggle-gdpr

[7] https://numpy.org/doc/1.13/index.html

[8] https://stackoverflow.com/questions/60359398/python-detect-a-qr-code-from-an-image-and-crop-using-opencv

[9] https://computergraphics.stackexchange.com/questions/3904/is-it-better-to-blur-the-input-or-output-of-an-edge-detection-shader-for-noise-r

[10] https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html

[11] https://docs.opencv.org/4.x/d7/d1b/group__imgproc__misc.html#gae8a4a146d1ca78c626a53577199e9c57

[12] https://docs.opencv.org/3.4/d9/d61/tutorial_py_morphological_ops.html

[13] https://docs.opencv.org/4.x/d4/d73/tutorial_py_contours_begin.html

[14] https://docs.opencv.org/4.x/dd/d49/tutorial_py_contour_features.html