

QR-Code identifizierung

Ausarbeitung zum Praktikum der Vorlesung Computer Vision

veranstaltet von:

Prof. Dr. Xiaoyi JIANG

xjiang@uni-muenster.de

betreut von:

Claas KÄHLER

claas.kaehler@uni-muenster.de

Teilnehmer:

Armin WOLF

a_wolf28@uni-muenster.de

Christian ESCH

c_esch05@uni-muenster.de

Philipp SCHOFIELD

...@uni-muenster.de

Version vom 22. März 2017

Liste der noch zu erledigenden Punkte

Quellen angeben!	5
verweis	9

KAPITEL 1

Einführung

1.1 Aufgabenstellung

Ziel dieses Praktikums war es eine Software zu konstruieren um QR-Codes in Bildern zu lokalisieren und danach zu extrahieren. Dabei sollten sich die Teams auf QR-Codes, die dem ISO/IEC Standard 18004 entsprechen, beschränken. Unter der Annahme, dass sämtliche QR-Codes auf den Eingabebildern planar waren, sollten perspektivische Transformationen oder ähnliche Verzerrungen entfernt werden. Zusätzlich sollte sich jedes Team darum kümmern ein *Dataset* zur Analyse und späteren Evaluation zu erschaffen.

1.2 Aufbau des QR-Codes

Um das spätere Vorgehen der Lokalisierung des QR-Codes nachvollziehen zu können, wollen wir kurz auf den Aufbau des QR-Codes eingehen. Wie in der Abbildung ?? zu sehen ist, sind im Code die Informationen zur Version, Format und der Fehlerkorrektur enthalten.

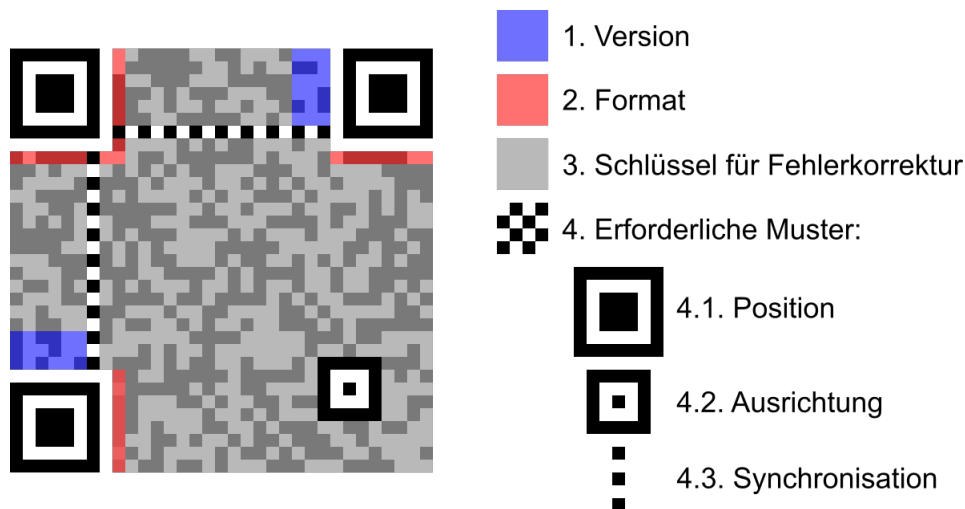


Abbildung 1.1: QR-Code Strukturbeispiel

Des weiteren ist klar zu erkennen, dass der Code in drei von vier Ecken das Muster 4.1 vorweist. Diese dienen zur Bestimmung der Orientierung. Zusätzlich wird das Muster 4.2 (*Finder Pattern*) verwendet um die Ausrichtung noch präziser zu bestimmen. Bei größeren QR-Codes werden weitere Ausrichtungsmuster

Quellen angeben!

(Muster 4.2) eingefügt. Zwischen den *Finder Patterns* befindet sich ein Streifen mit abwechselnd „schwarz - weißen“ Punkten (*Modulen*). Das sind die so genannten Synchronisations Muster oder auch *Timing Patterns*.

Die Größe der QR-Codes ist beschränkt durch die Anzahl der *Module*. Die Anzahl der *Module* liegt zwischen 21×21 und 177×177 . Beispielsweise sei der Code in Abbildung ?? einer der Größe 21×21 . Dann hätte ein *Finder Pattern* die Länge von 7 *Modules*. Die Anzahl ergibt sich aus der einzigartigen 1 : 1 : 3 : 1 : 1 Struktur eines *Finder Patterns*. Diese Fakten werden später ausschlaggebend sein bei der Rasterisierung des QR-Codes.

1.3 Die verwendete Bibliothek OpenCV

*OpenCV*¹ ist eine Bibliothek mit Algorithmen spezialisiert auf „Computer Vision“. Sie wurde für die Programmiersprachen C/C++ geschrieben und steht unter BSD Lizenz. Es gibt mehrere Versionen der Bibliothek, die aktuellste ist die 3.2. Unser Programm setzt die Mindestanforderung auf Version 2.4. Außer *OpenCV* wurde keine weitere Bibliothek verwendet.

¹ WWW-Seite von dem Projekt *OpenCV*: <http://opencv.org/>

KAPITEL 2

Bildvorverarbeitung

Die Bildvorverarbeitung ist ein essenzieller Schritt um die QR-Codes richtig lokalisieren zu können. Das Eingabebild soll in dieser Phase erstmal in ein Graustufenbild umgewandelt werden und danach binarisiert werden. *OpenCV* bietet die Möglichkeit bilder direkt als Graustufenbild einzulesen oder sie in eins umzuwandeln. Das in Graustufen vorhandene Bild kann weiter verarbeitet werden. Die Klasse *ImageBinarization* ist für die binarisierung zuständig.

Listing 2.1.: Der Gesamtablauf der Binarisierung.

```
Mat ImageBinarization::run(Mat image, int thresholdMethod) {  
    this->image = image;  
    computeSmoothing();  
    createHistogram();  
  
    printThresholdMethod(thresholdMethod);  
    switch (thresholdMethod) {  
        case Global:  
            setThresholdValue();  
            computeGlobalThreshold();  
            break;  
  
        case LocalMean:  
            computeLocalThreshold(ADAPTIVE_THRESH_MEAN_C, 11, 0);  
            break;  
  
        case LocalGaussian:  
            computeLocalThreshold(ADAPTIVE_THRESH_GAUSSIAN_C, 11, 0);  
            break;  
    }  
    return binarizedImage;  
}
```

Listing 2.1 zeigt den Ablauf der Binarisierung. In Zeile 4 wird die Methode *computeSmoothing* aufgerufen, die eine Gaußglättung auf dem Bild durchführt, um eventuelles Rauschen zu mindern. Im nächsten Schritt wird ein Farbstufen-Histogramm mithilfe der *computeHistogram* Methode berechnet. Um die best Mögliche Binarisierung zu erreichen, wurden drei Methoden zur Schwellwert Berechnung implementiert:

- globales Schwellwertverfahren,
- Mittelwert basiertes Schwellwertverfahren,
- Gauß'sches Schwellwertverfahren.

Im ersten durchlauf wird das globale Schwellwertverfahren angewandt. Sollte dies keine gültige Ergebnisse liefern (keine drei *Finder Pattern* enthalten), so wird das nächste Verfahren gewählt. Abhängig von der Wahl werden zwei verschiedene *OpenCV* Methoden verwendet. Bei der globalen Schwellwert Berechnung wird auf die *threshold* Methode wie sie im Listing 2.2 steht zurückgegriffen. Der Methode werden das Ein- und Ausgabe Bild, die Schwellwertgrenze, der maximale Farbwert und die Wahl des Algorithmus übergeben.

Listing 2.2.: Binarisierung anhand des globalen Schwellwertverfahrens.

```
void ImageBinarization::computeGlobalThreshold() {
    threshold(blurredImage, binarizedImage, threshold_value, max_BINARY_value, CV_THRESH_OTSU);
}
```

Wir haben uns hier auf den Algorithmus von Otsu¹ geeinigt, da er sehr gute Ergebnisse mit einer geringen Laufzeit liefert.

Bei den lokalen Schwellwertverfahren hingegen wird die Methode *adaptiveThreshold* verwendet. Listing 2.3 veranschaulicht die Methode die für die zwei lokalen Verfahren verwendet wird. Für die jeweiligen Verfahren wird der Parameter *adaptiveMethod* passend übergeben.

Listing 2.3.: Binarisierung anhand eines lokalen Schwellwertverfahrens.

```
void ImageBinarization::computeLocalThreshold(int adaptiveMethod, int blockSize, int C) {
    adaptiveThreshold(blurredImage, binarizedImage, max_BINARY_value, adaptiveMethod,
        THRESH_BINARY, blockSize, C);
}
```

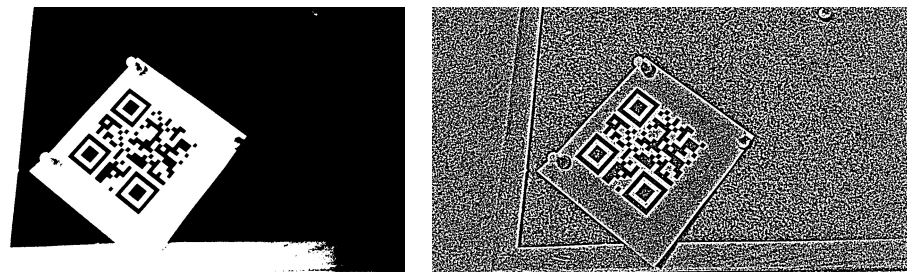


Abbildung 2.1: Das Resultat der Binarisierung mit den jeweiligen Verfahren. Links global und Rechts lokal.

Ausblick: Durch das verändern des Parameters *C* in der *adaptiveThreshold* Methode kann eine ausgeglichene Binarisierung erzielt werden. Diese Konstante wird von jedem Pixel abgezogen um einen Ausgleich zu schaffen. Allerdings können aus leicht Informationen verloren gehen!

¹ Mehr Information zu dem Algorithmus auf der zugehörigen *OpenCV* Seite: http://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html#threshold

KAPITEL 3

Patternidentifikation

Der nächste Schritt auf dem Weg zur Lokalisierung der QR-Codes ist das Identifizieren der *Finder Patterns*. Dafür wird die durch *OpenCV* bereitgestellte `findContours` Methode verwendet. Sie basiert auf dem Algorithmus von Suzuki und wird eingesetzt um die Konturen aus dem Binärbild zu bestimmen.

Listing 3.1.: Aufruf der `findContours` Methode um die Konturen zu bestimmen.

```
/* 1
 * @param image Source, an 8-bit single-channel image. 2
 * @param allContours Detected contours. 3
 * @param hierarchy information about the image topology 4
 * @param CV_RETR_TREE Contour retrieval mode (full hierarchy) 5
 * @param CV_CHAIN_APPROX_NONE Contour approximation method 6
 * (stores absolutely all the contour points. ) 7
 */ 8
findContours(image, allContours, hierarchy, CV_RETR_TREE, CV_CHAIN_APPROX_NONE); 9
```

Listing 3.1 zeigt den Aufruf um alle Konturen des übergebenen Bildes zu erhalten.

3.1 Vorgehensweise des Algorithmus von Suzuki

Das Binärbild wird als ein Raster durch iteriert. Für jeden Pixel $p_{i,j}$ wird geprüft ob er die Bedingungen erfüllt ein Teil einer Kontur sein könnte. Das Paper führt hierzu zwei Formen von Konturen ein *outer border* und *hole border*. Sind beide Bedingungen erfüllt ist der Pixel $p_{i,j}$ ein Anfangspunkt einer neuen Kontur. Diese Kontur muss eindeutig identifizierbar sein daher wird sie mit einer *KonturID* versehen. Großer Vorteil dieses Algorithmus ist die abgespeicherte Hierarchie der Konturen. Um dies zu gewährleisten muss die *parent*-Kontur für die neue Kontur gesetzt werden. Während des Scannens des Gitters wird immer die äußere Kontur zwischen gespeichert, diese ist entweder die *parent*-Kontur oder die Kontur die, die neue Kontur und die *parent*-Kontur teilt. Wenn alle Werte gesetzt sind, wird die Kontur durch sukzessives hinzunehmen von Pixeln erzeugt. Nach jeder Konturerzeugung springt der Algorithmus zurück zum Raster Scan. Der Algorithmus terminiert bei Erreichen der rechten unteren Ecke.

verweis

Das Original Paper beschreibt ausführlich das Vorgehen anhand Beispielen und Pseudocode.

3.2 Filtern der Konturen

Nachdem die Konturen bestimmt wurden, müssen die Konturen gefiltert werden da abhängig vom Bild können eine variable Anzahl an Konturen erkannt werden. Beispielsweise werden bei der Ausführung der

adaptiven Schwellwertoperation sehr viele zu kleine Konturen erkannt. Daher werden Konturen in folgenden Fällen ignoriert:

1. Die Kontur ist zu klein oder zu groß.
2. Die Kontur besitzt kein *parent*-Kontur.
3. Konturen die Nachbarkonturen besitzen.
4. Konturen die eine *child*-Kontur besitzen.

Für die äußeren Konturen also die *parent*-Konturen müssen die Bedingungen 1.-3. natürlich auch gelten. Zusätzlich muss für die äußerste Kontur eine Trapezoide Form gelten.

KAPITEL 4

Code Extrahierung

KAPITEL 5

Normalisierung

KAPITEL 6

Generierung der Datensätze