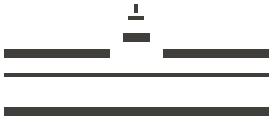


QR-Code Identifizierung

Ausarbeitung zum begleitenden Praktikum zu Computer Vision

Wintersemester 2016/17

Veranstalter:
Prof. Dr. Xiaoyi Jiang
Betreuer:
Claas Kähler

Teilnehmer:
Philipp Schofield
Armin Wolf
Christian Esch

Version vom 17. April 2020

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Aufgabenstellung

Ziel dieses Praktikums war es eine Software zu konstruieren um QR-Codes in Bildern zu lokalisieren und danach als Binärbild 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 sind, 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 einen Teil des Aufbau des QR-Codes eingehen, der für die folgende Arbeit von Bedeutung ist. Jeder QR-Code besteht aus drei *Finder Patterns* (siehe Abbildung 1.1 Muster 4.1) welche in der oberen linken, oberen rechten und unteren linken Ecke angeordnet sind und sowohl zur Lokalisierung und Rotation verwendet werden. Zwischen den Finder Patterns befinden sich die *Timing Patterns* (siehe Muster 4.3), welche aus abwechselnd „schwarz - weißen“ Punkten (*Module*) bestehen.

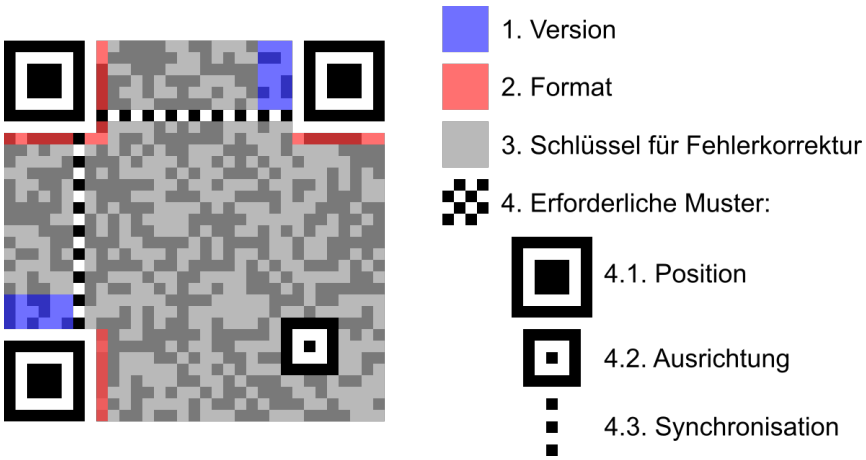


Abb. 1.1: QR-Code Strukturbeispiel (siehe Quelle [**qrcoderef**])

Zusätzlich gibt es noch *Alignment Patterns* (Muster 4.2), Versions- und Formatinformationen, welche hier in Rot und Blau gekennzeichnet sind. Diese werden im Verlauf dieser Arbeit aber nicht verwendet. Die eigentliche kodierte Nachricht ist in Grau dargestellt.

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 1.1 einer der Größe 21×21. Dann hätte ein *Finder Pattern* die Länge von 7 *Modulen*. Die Anzahl ergibt sich aus der einzigartigen $1:1:3:1:1$ Struktur eines *Finder Patterns*. Dieser Aufbau wird später für die Rasterisierung des QR-Codes verwendet.

1.3 Die verwendete Bibliothek OpenCV

OpenCV [**opencv**] 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 Version 3.2. Das implementierte Programm setzt die Mindestanforderung auf Version 2.4. Außer *OpenCV* wird keine weitere Bibliothek verwendet.

Kapitel 2

Bildvorverarbeitung

Die Bildvorverarbeitung stellt ein essenziellen Schritt zur Lokalisierung der QR-Codes dar. Das Eingabebild wird in dieser Phase zuerst in ein Graustufenbild umgewandelt und danach binarisiert. Der Einfachheit halber werden zu große Bilder zuerst noch auf unter 1500 in der größten Dimension verkleinert. *OpenCV* bietet die Möglichkeit Bilder direkt als Graustufenbild einzulesen, welches für Schwellenwertoperationen genutzt wird. Die Klasse `ImageBinarization` ist für die darauf folgende Binarisierung zuständig.

Listing 2.1.: Ablauf der Binarisierung.

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

In Zeile 4 wird die Methode `computeSmoothing` aufgerufen, die eine Gaußglättung auf dem Bild durchführt, um den Einfluss von Rauschen zu mindern. Um mit verschiedenen Belichtungsszenarien umgehen zu können, sind drei Methoden zur Schwellwertberechnung implementiert:

- Globales Schwellwertverfahren
- Mittelwert-Schwellwertverfahren
- Gaußsches-Schwellwertverfahren

Listing 2.2.: Binarisierung anhand des globalen Schwellwertverfahrens.

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

Im ersten Durchlauf wird das globale Schwellwertverfahren angewandt. Bei der Berechnung wird auf die `threshold` Methode, wie sie im Listing 2.2 steht, zurückgegriffen. Diese verwendet den *Otsu* Algorithmus[1], um einen globalen Schwellwert zu berechnen und dann das Bild zu binarisieren. [**opencv**]

Sollte mit dieser Methode im späteren Verlauf nicht mindestens drei Finder Patterns lokalisiert werden, wird das nächste Verfahren ausgeführt.

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);
}
```

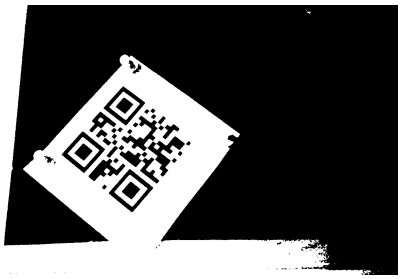
Bei den lokalen Schwellwertverfahren wird die Methode `adaptiveThreshold` verwendet, welche in Listing 2.3 veranschaulicht wird. Im Gegensatz zum globalen Schwellwertverfahren, wird hier für jeden Pixel anhand der Nachbarschaft ein eigener Schwellwert errechnet. Als empirisch gut geeignet hat sich eine Nachbarschaft von 11×11 herausgestellt.

Die Berechnung des Schwellwerts für die Nachbarschaft erfolgt auf zwei unterschiedliche Verfahren:

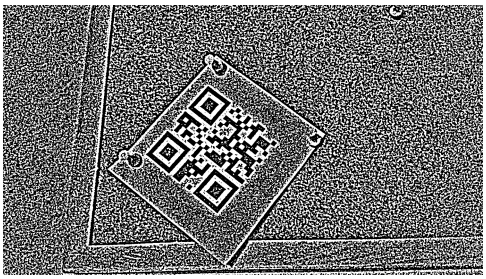
`ADAPTIVE_THRESH_MEAN_C`: Der Mittelwert der Nachbarschaft.

`ADAPTIVE_THRESH_GAUSSIAN_C`: Ein Gauß gewichteter Mittelwert der Nachbarschaft.

Der Parameter `C` wird vom Mittelwert subtrahiert. In der Implementierung wird `C` konstant 0 gewählt.



(a) globales Schwellwertverfahren



(b) lokales Schwellwertverfahren

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

[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 Konturen im Binärbild zu erkennen.

Listing 3.1.: Aufruf der `findContours` Methode

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

3.1 Vorgehensweise des Algorithmus von Suzuki

Iteriere über das Binärbild. Für jeden Pixel $p_{i,j}$ überprüfe ob folgende Bedingungen wahr sind:

outer border Der Farbton des Vorgängers $p_{i,j-1}$ unterscheidet sich vom aktuell betrachteten Pixel $p_{i,j}$.
Bsp.: $p_{i,j} = 1$ und $p_{i,j-1} = 0$ dann ist $p_{i,j}$ ein Startpunkt eines *outer border*.

hole border Sei $x > 1$ ein beliebiger Abstand und es gelte $p_{i,j-x} = \ldots = p_{i,j-1} = 1$ und $p_{i,j} = 0$, dann ist $p_{i,j}$ ein Startpunkt eines *hole border*.
Bsp.: Für $x = 2 : p_{i,j} = 1$ und $p_{i,j-1} = 0$ und $p_{i,j-2} = 0$ dann ist $p_{i,j}$ ein Startpunkt eines *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. Der Vorteil dieses Algorithmus ist, dass so eine Hierarchie der Konturen aufgebaut wird. Dazu muss die *parent*-Kontur für die neue Kontur gesetzt werden. Während der Iteration des Bildes wird immer die äußere Kontur zwischengespeichert. Diese ist entweder eine *parent*-Kontur oder eine Kontur die, die neue Kontur und die *parent*-Kontur teilt. Wenn alle Werte gesetzt sind, wird die Kontur durch sukzessives Hinzunehmen von Pixeln entlang der Kante erzeugt. Das heißt es wird eine Kantenverfolgung durchgeführt. Nach jeder Kontur Erzeugung springt der Algorithmus zurück und führt die Iteration fort. Der Algorithmus terminiert bei Erreichen der rechten unteren Ecke.

Das Original Paper [**journals/cvgip/SuzukiA85**] beschreibt ausführlich das Vorgehen anhand von Beispielen und Pseudocode.

3.2 Filtern der Konturen

Nachdem die Konturen erkannt wurden, werden Sie mithilfe der durch `findContours` erstellten Hierarchie gefiltert. Dabei werden Konturen verworfen, die mindestens eine dieser Bedingung erfüllen:

1. Die Kontur ist zu klein oder zu groß.
2. Die Kontur hat keine *parent*-Kontur.
3. Die Kontur hat Nachbarkonturen.
4. Die Kontur hat eine *child*-Kontur.

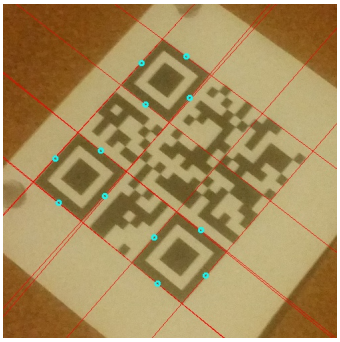
Für jede nicht verworfene Kontur wird ihre zugehörige *parent*-Kontur betrachtet. Für diese muss die Bedingungen 1 und 3 gelten. Zuletzt wird von dieser Kontur noch einmal die nächste *parent*-Kontur betrachtet. Für diese muss wieder Bedingung 1 zutreffen und außerdem muss diese Kontur eine Trapezoide Form aufweisen. Um die Trapezoide Form zu überprüfen wird dazu die Form der Kontur mit der `approxPolyDP` Methode approximiert. Wenn diese aus genau vier Punkten besteht, wird sie als Trapezoid anerkannt.

Falls nicht mindestens drei Finder Patterns lokalisiert wurden, wird das nächste Verfahren für die Bildbinarisierung ausgeführt und die Suche beginnt von vorne. Wurden alle Verfahren durchgeführt ohne das ausreichend Finder Patterns gefunden wurden, wird angenommen das kein QR-Code im Bild zu finden ist und der Prozess wird beendet.

3.3 Kanten Approximation

Nachdem mehrere Kandidaten von Finder Patterns gefunden wurden, wird als nächstes die äußerste Kontur dieser Kandidaten verwendet, um deren Kanten zu approximieren. Zuvor wurden mit *OpenCV* und Suzukis Algorithmus 3.1 alle Konturen `CV_CHAIN_APPROX_NONE` im Bild lokalisiert, weswegen jede Kontur jetzt eine zusammenhängende Kette von Punkten darstellt. Für jede Kontur sollen mithilfe dieser Punktmenge die vier äußeren Kanten des Finder Patterns approximiert werden. Dazu werden die Konturen in vier Segmente aufgeteilt.

(a) Segmentierung



(b) Geraden Approximation

Abb. 3.1: Aufteilen der einzelnen Konturen in Segmente und Approximation der Kanten durch Geraden.

Es müssen zunächst die Schnittpunkte gefunden werden, an denen die Kontur geteilt werden soll. Dazu werden die bereits zuvor berechneten Punkte, die durch Approximation der Trapezoiden Form der äußersten Kontur gefunden wurden, verwendet. Diese liegen jedoch im Regelfall nicht exakt auf den Ecken der Finder

Patterns. Außerdem kann schlechte Bildqualität zu Abrundungen oder verschobenen Ecken führen. Um bessere Kanten Approximationen zu erhalten, werden jeweils 10% aller Punkte am Anfang und Ende jedes Segments verworfen. Diese Segmente, wie sie in Abbildung 3.1a zu sehen sind, werden dann genutzt, um mit der `fitLine`-Methode die Kanten zu approximieren.

Die `fitLine`-Methode beruht auf dem Prinzip von *M-Estimators* und verwendet als Minimierungsfunktion den, in *OpenCV* durch `CV_DIST_FAIR` definierten, Ausdruck. Die approximierten Geraden werden durch einen Stützvektor und Richtungsvektor beschrieben. Eine wichtige Eigenschaft der Stützvektoren ist, dass Sie sich innerhalb der konvexen Hülle aller Punkte befinden, welche zur Approximation verwendet wurden.

Am Ende der Identifikation aller Möglichen Finder Patterns sind somit folgende Informationen bekannt:

- Mindestens drei Finder Patterns
- Eine Kontur pro Finder Pattern
- Vier Segmente pro Finder Pattern
- Vier Kantengeraden pro Finder Pattern

Kapitel 4

Extrahierung

Der nächste Schritt auf dem Weg zum finden eines gültigen QR-Codes besteht darin, alle Möglichen Dreier-Kombinationen von Finder Patterns zu prüfen und festzustellen, ob diese zur Extrahierung eines gültigen QR-Codes führen. Für jede Kombination wird versucht, die vier Eckpunkte des QR-Codes zu berechnen, um aus diesem eine Matrix für eine perspektivische Transformation in eine 2D-Ebene zu berechnen. Dies führt zu $\binom{k}{3}$ möglichen Kombinationen von Finder Patterns, wobei k die Anzahl aller gefundenen Finder Patterns ist.

4.1 Pattern Positionierung

Hat man drei Finder Patterns ausgewählt, muss für die korrekte Extrahierung festgestellt werden, welche Position im QR-Code jedes Finder Pattern einnimmt.

Zuerst werden dazu die Finder Patterns durch Verwendung der Gaußschen Trapezformel im Uhrzeigersinn angeordnet.

Definition 4.1.1 (Gaußsche Trapezformel) *Berechnet den Flächeninhalt eines einfachen Polygons.*

$$A = \frac{1}{2} \sum_{0}^{n-1} (y_i + y_{i+1 \bmod n})(x_i - x_{i+1 \bmod n}) \tag{4.1}$$

Dabei ist n die Anzahl der Punkte in einem Polygon und x_i und y_i die Punktkoordinaten des i-ten Punkts. Abhängig vom Drehsinn der Punkte im Bezug auf das Koordinatensystem ist der Flächeninhalt entweder positiv oder negativ. In einem kartesischen Koordinatensystem entspricht ein positiver Flächeninhalt einem Drehsinn im Uhrzeigersinn und ein negativer gegen den Uhrzeigersinn.

Angewendet wird die Formel auf das Polygon, welches aus dem jeweils ersten Punkt der Kontur jedes Finder Patterns besteht. Ist der Flächeninhalt negativ wird die Reihenfolge geändert, indem das erste und zweite Finder Pattern miteinander vertauscht werden.

Ähnlichkeitsmaß für Geraden

Sind die Finder Patterns in korrekter Reihenfolge, wird als nächstes das obere linke Finder Pattern bestimmt. Zurzeit sind durch die drei Finder Patterns insgesamt 12 approximierte Geraden bekannt. Betrachtet man den Aufbau eines QR-Codes, ist ersichtlich, dass jeweils 4 Geradenpaare die selbe Kante beschreiben. Es gibt insgesamt also nur 8 unterschiedliche Kanten und nur das obere linke Finder Pattern teilt sich jede Kante mit einem anderen Finder Pattern. Die anderen beiden teilen sich je nur 2 Kanten mit dem oberen linken. Stellt man also fest, welche der 12 Geraden die gleichen Kanten im QR-Code approximieren, ergibt sich, welches Finder Pattern das obere linke ist.

Für das effiziente Zuordnen von Geraden zueinander, wird ausgenutzt, dass die Stützvektoren stets innerhalb der konvexen Hülle der Segmente liegen, durch welche sie approximiert wurden. Die Ähnlichkeit von zwei Geraden wird dann definiert, durch die Summe der Entfernungen der Stützvektoren zu den Geraden.

Listing 4.1.: Ähnlichkeitsmaß für zwei Geraden

```
double similarity = pointLineDistance(lineOne.supportVector, lineTwo)
                  + pointLineDistance(lineTwo.supportVector, lineOne);
```

Ein kleinerer Wert bedeutet bei dieser Definition, dass sich zwei Geraden besonders ähnlich sind. In Abbildung 3.1b ist dabei gut zu erkennen, dass wenn es sich um einen QR-Code handelt, dieses Maß ausreichend gut feststellt, welche Geraden die selbe Kante approximieren. Für das obere linke Finder Pattern gilt dann, dass es das Finder Pattern mit den vier kleinsten Ähnlichkeitsmaßen ist.

Da die Finder Patterns zuvor im Uhrzeigersinn angeordnet wurden, wird durch Rotation im Uhrzeigersinn nicht nur das obere linke Finder Pattern an die erste Stelle des Pattern Arrays gebracht, sondern auch direkt das obere rechte Finder Pattern an die zweite Stelle und das untere linke Finder Pattern an die dritte Stelle. Mit Ausnahme für Spiegel verkehrte QR-Codes, kann zuverlässig die korrekte Position aller Finder Patterns im QR-Code bestimmt werden.

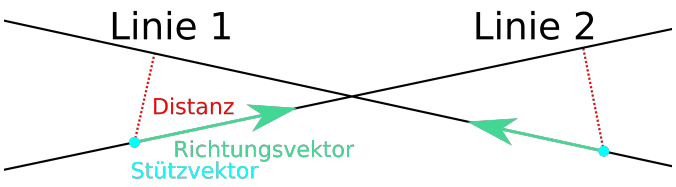


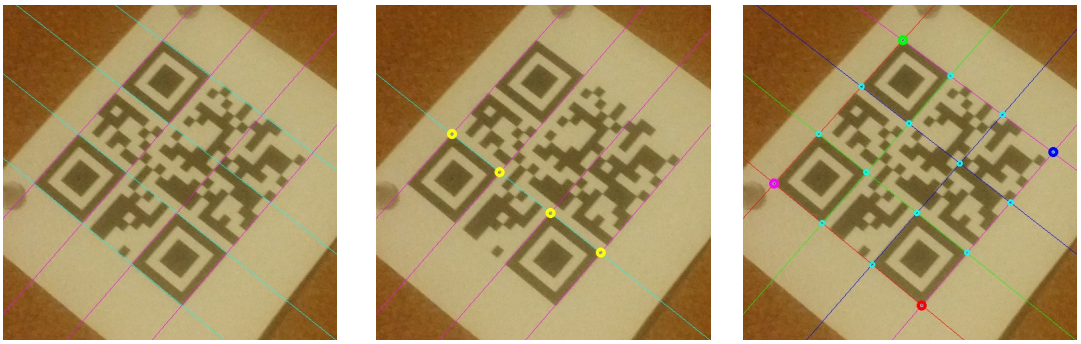
Abb. 4.1: Links: Ähnlichkeitsmaß für zwei Geraden.

4.2 Kanten Zuordnung

Um die Eckpunkte des QR-Codes durch Schnittpunktbildung der äußeren Kanten zu berechnen, muss als nächstes festgestellt werden, welche Geraden die Außenkanten des QR-Codes beschreiben.

Vertikale und Horizontale Geraden

Da bereits berechnet wurde, welche der Geraden die selbe Kante approximieren, kann diese Information benutzt werden um eine qualitativ bessere Approximation für alle Kanten des oberen linken Finder Patterns zu berechnen. Dazu wird die Vereinigung der Segmente der gepaarten Geraden berechnet. Auf der Vereinigung wird erneut der Approximationsprozess ausgeführt. Indem dabei festgestellt wird, ob mit einer Geraden aus dem oberen rechten Finder Pattern oder dem unteren linken Finder Pattern vereinigt wird, ergibt



(a) Zuordnung der Geraden zu horizontalen oder vertikalen Kanten.

(b) Schnittpunkte entlang der Sortier-Achse

(c) Schnittpunkte aller horizontalen und vertikalen Geraden

Abb. 4.2: Kanten Zuordnung und Berechnung der Eckpunkte.

sich im QR-Code Koordinatensystem, ob eine horizontale oder vertikale Kante gefunden wurde. Die nicht verwendeten Kanten am Ende des Prozesses sind trivial zuzuordnen. Falls nicht genau je zwei Geradenpaare vom oberen linken Finder Pattern und oberen rechten Finder Pattern, sowie zwei Geradenpaare vom oberen linken Finder Pattern und unteren linken Finder Pattern vereinigt wurden, ist die Kombination von Finder Patterns kein QR-Code und es kann abgebrochen werden.

War die Vereinigung erfolgreich, gibt es genau vier horizontale und vier vertikale Geraden, wie in Abbildung 4.2a zu sehen.

Sortieren der Geraden

Als nächstes werden die horizontalen Geraden entlang einer beliebigen vertikalen Gerade sortiert, die von nun an als Sortier-Achse bezeichnet wird.

Wie in Abbildung 4.2b gezeigt, werden dazu alle Schnittpunkte zwischen der Sortier-Achse und den horizontalen Geraden berechnet. Dann wird eine Koordinatensystem Transformation ausgeführt, sodass die Sortier-Achse zur neuen X-Achse wird. Somit können die horizontalen Geraden nach der Größe der X-Werte der Schnittpunkte mit der Sortier-Achse sortiert werden. Zuletzt wird festgestellt, ob das erste Element der so sortierten Liste von horizontalen Geraden mit einer Geraden des oberen linken Finder Patterns übereinstimmt. Falls nicht wird die Liste umgedreht.

Für die vertikalen Geraden wird dies analog durchgeführt. Diese Methode stellt sicher, dass unabhängig von perspektivischen Verzerrungen die Sortierreihenfolge stabil bleibt.

Das Ergebnis sind zwei Listen von Geraden, die im QR-Code Koordinatensystem von oben links nach unten rechts sortiert sind. Dementsprechend werden nun trivial die Schnittpunkte der ersten und letzten Geraden berechnet. Diese Bilden die eindeutig identifizierten Eckpunkte des QR-Codes wie in Abbildung 4.2c zu sehen ist. Mit diesen vier Punkten und der *OpenCV* Funktion `getPerspectiveTransform`, sowie `warpPerspective` wird dann eine perspektivische Transformationsmatrix errechnet und der QR-Code extrahiert.

Kapitel 5

Normalisierung

5.1 Rasterisierung

Nach der perspektivischen Transformation, liegt das extrahierte Bild in einem $n \times n$ großem binarisiertem Zustand vor. Nun muss der QR-Code rasterisiert werden.

Man betrachte dazu die untere rechte Ecke des oberen linken Finder Pattern. Es ist bekannt, dass ein Finder Pattern die Dimension 7×7 hat, sodass man die x bzw. y-Koordinate der genannten Ecke durch 7 dividiert. Dies liefert die Zellenlänge z des Rasters, woraus sich die Anzahl der Module m via Division von n mit z berechnen lässt. Im Normalfall gilt jedoch $m \notin \mathbb{Z}$. Es ist jedoch bekannt, dass ein QR-Code nur

$$17 + 4 \cdot version$$

für $version \in \{1, ..., 40\} \subset \mathbb{N}$, Module besitzen kann.

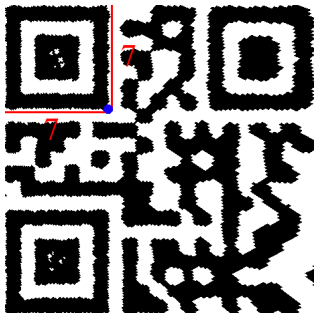


Abb. 5.1: Rasterlänge berechnen

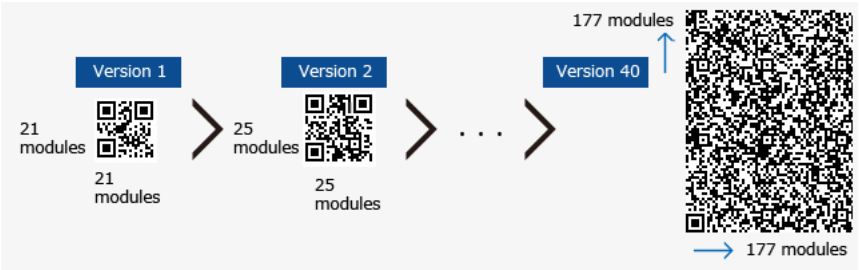


Abb. 5.2: Mögliche Modulgrößen von QR-Codes [**versionimage**].

Die Anzahl der geschätzten Module m wird zur Berechnung der nächsten ganzzahligen Versionsnummer genutzt:

$$version = round((m - 17)/4)$$

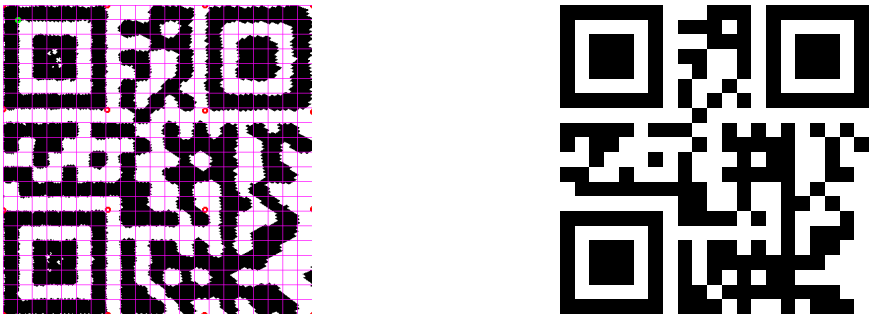
Die zur Versionsnummer gehörenden Anzahl der Module ist dann gegeben durch:

$$modules = 17 + 4 \cdot version$$

Die Zellenlänge des Rasters berechnet sich letztendlich aus Divison von n mit $modules$. Diese Zellenlänge wird für den folgenden Schritt der Normalisierung benötigt.

5.2 Normalisierung

Die Normalisierung wird mithilfe des Rasters durchgeführt, indem *majority votes* pro Zelle den Pixelwert des normalisierten QR-Codes festlegen. Das Ergebnis ist ein $modules \times modules$ großer normalisierter QR-Code.



(a) Extrahiert mit Raster **(b)** Normalisierter QR-Code

Abb. 5.3: Normalisierung des QR-Codes.

5.3 Verifikation

Um fälschlicherweise erkannte QR-Codes zu vermeiden, wird im Anschluss eine Verifikation durchgeführt. Dies geschieht, indem die Pixel des normalisierten Bildes, wo die Finder Patterns und Timing Patterns liegen, mit denen eines Standard QR-Codes der entsprechenden Version verglichen werden.

Abb. 5.4: Finder und Timing Patterns eines Standard QR-Codes der Version 1

Ist die Übereinstimmung des normalisierten Bildes mit einem Standard QR-Code geringer als 85%, so wird, falls möglich, die Normalisierung und die Verifikation mit einer Versionsnummer von ±1 wiederholt. Dadurch wird Rundungsfehlern bei der Rasterisierung vorgebeugt. Sind alle Übereinstimmungen unter 65%, so wird der QR-Code verworfen, andernfalls wird der QR-Code mit der höchsten Übereinstimmung gewählt und gespeichert.

Für den Fall, dass mehr als ein valider QR-Code gefunden wurde, wird am Ende, im Rahmen der Aufgabenstellung, der QR-Code mit der höchsten Übereinstimmung ausgegeben.

Kapitel 6

Evaluation

Um den implementierten Algorithmus zu testen, werden sowohl reale als auch synthetische Datenbanken verwendet. Die generierte synthetische Datenbank stellt eine Vielzahl von verschiedenen Testbildern bereit. Ein Vorteil dieser ist, dass die Parameter der generierten Testbilder manuell eingestellt werden können. Dies ermöglicht eine einfache Analyse, unter welchen Umständen die implementierte QR-Detektion scheitert.

6.1 Ablauf der synthetischen Generierung

Der Generator liest einen Ordner von *Ground Truth* Bildern ein. Diese bestehen aus randlosen QR-Codes mit einem Pixel pro Modul. Auf den eingelesenen Bildern werden der Reihe nach die in Tabelle 6.1 aufgelisteten Operationen ausgeführt, und für jede Operation die Zwischenergebnisse in einem entsprechenden Ordner gespeichert.

`Input`: Menge von *Ground Truth* QR-Codes
`Output`: Datenbank von synthetischen Bildern

1. Randerstellung
2. Skalierung
3. Rotation
4. Perspektivische Transformation
5. Einbettung in Hintergründe
6. Gaußglättung
7. Rauschen

Tabelle 6.1: Operationreihenfolge beim Generieren der synthetischen Datenbank.

Pro Operation kann festgelegt werden, wie viele Dateien generiert und gespeichert werden sollen. Für jede Operation werden diskrete Schrittweiten und Start und Endwerte angegeben. So zum Beispiel:

- Rotation: Diskretisierung in 45° Schritten
- Gaußglättung: Startwert von $n = 3$, Schrittweite von 6, Maximalwert von $n = 27$
- ...

Die diskreten Schrittweiten pro Operation sind gleich verteilt, sodass möglichst alle Variationen abgedeckt werden.

6.2 Testergebnisse

Das Anwenden des implementierten QR-Detektors auf alle Zwischenergebnisse der synthetischen Datenbank liefert folgende Ergebnisse, dabei ist die Qualität durch die relative Übereinstimmung der erkannten QR-Codes mit den *Ground-Truth* Bildern gegeben. Besitzt der ausgegebene QR-Code die falsche Dimension wird er als nicht erkannt gewertet.

	Anzahl	Erkannt	Qualität	Parameter
Skalierung	60	60	98%	s = 6,IL, IN
Rotation	420	420	97.35%	$0 < r < 360$, $k = 45$
Perspektive	1000	935	95.6%	$0 \leq x, y < 0.3$, $k = 0.1$
Einbettung	300	268	96.82	$d = 60\%$
Glättung	200	131	94.4%	$5 \leq n \leq 23$, $k = 6$
Rauschen	200	116	95.24%	ø= 0, $15 \leq \sigma \leq 45$, $k = 15$

s	$\widehat{=}$	Skalierungsfaktor
IL	$\widehat{=}$	lineare Interpolation
IN	$\widehat{=}$	nearest neighbour Interpolation
k	$\widehat{=}$	diskrete Schrittweite
r	$\widehat{=}$	Rotationswinkel
(x, y)	$\widehat{=}$	Position der oberen linken Ecke in Prozent bzgl. Dimension des Bildes
d	$\widehat{=}$	Größe des QR-Codes innerhalb des Hintergrundes entlang der kleinsten Achse
n	$\widehat{=}$	Nachbarschaft
ø	$\widehat{=}$	Mittelwert
σ	$\widehat{=}$	Standardabweichung

Tabelle 6.2: Evaluation der synthetischen Datenbank.

Die Datenbanken von abfotografierten QR-Codes [**databasebrno**] stammen von einer Forschungsgruppe der *Brno University of Technology*. Die Bilder zeigen QR-Codes umschlossen von Text, unter verschiedenen Belichtungsszenarien, in realen Arbeitsumgebungen, in der freien Natur oder vor künstlichen, besonders gekachelten Hintergründen.

Für diese QR-Codes sind *Ground Truth* Bilder vorhanden, jedoch nicht in dem Format, wie sie für automatisierte Qualitätsvergleiche von der Implementierung des QR-Code-Detektors erwartet wird. Deswegen wird anstelle der Qualität nur die Erkennungsrate angegeben. Die Laufzeiten beziehen sich dabei auf die Erkennung von QR-Codes mit anschließender Ausgabe von *Debug* Bildern.

	Anzahl	Erkannt	Erkennungsrate	Laufzeit
dataset1	410	227	55.3%	$\approx$ 100s
dataset2	400	300	75%	$\approx$ 20s

Tabelle 6.3: Evaluation von Foto Datenbanken.

6.3 Auswertung

Bei der synthetischen Datenbank zeigt sich, wie in Tabelle 6.2 zu sehen ist, dass Skalierung, Rotation und Perspektivische Verzerrung keine Nennenswerten Probleme für die Detektion von QR-Codes darstellen. Die einzigen Ausreißer beschränken sich auf QR-Codes mit hoher Versionsnummer oder sehr extremen Perspektiven.

Die Verluste der erkannten QR-Codes von Perspektive zu Einbettung, sind den kumulativen Effekten der erneuten Skalierung zuzuschreiben. Insbesondere Große QR-Codes besitzen nach diesem Vorgang teilweise keine ausreichend großen Finder Patterns mehr. Bei der Glättung und dem Rauschen kommt es durch weitere kumulative Effekte zu noch mehr schwierigen Situationen, sodass selbst das menschliche Auge langsam versagt. Die Einzelnen Komponenten stellen für den implementierten QR-Code-Detektor keine Schwierigkeit dar, jedoch kann ein Zusammenspiel der Komponenten den QR-Code-Detektor an seine Grenzen bringen. Insbesondere das Rauschen und die Glättung hatten starke Effekte auf die Erkennungsrate.

Ist der Detektor aber in der Lage die Finder Patterns zu lokalisieren, so extrahiert er im Durchschnitt mit sehr guter Qualität die gefundenen QR-Codes. Einen Schluss den wir aus diesen Ergebnissen ziehen, ist dass der Detektor am ehesten von einer Verbesserung der Lokalisierung der Finder Patterns profitiert. Dies könnte zum Beispiel durch besser geeignete Schwellenwerte bei der Binarisierung geschehen. Eine erste einfache Verbesserung wäre zum Beispiel die zurzeit mit 0 gewählte Konstante `C` bei der Binarisierung auf einen sinnvolleren Wert zu setzen. Dies würde der derzeitig starken Körnung von adaptiv binarisierten Bildern entgegenwirken und die Konturdetektion verbessern.

Bei der Foto Datenbank zeigt sich, wie in Tabelle 6.3 zu sehen ist, eine grundlegende Schwäche der Lokalisierung von Finder Patterns durch Konturen. Selbst kleine Verletzungen der *Quietzone* durch, wie in diesem Fall, Text oder Schmutz führen direkt zu einem Versagen der Lokalisierung der Finder Patterns. Zudem stellen starke Beleuchtungsunterschiede immer noch ein schwieriges Hindernis dar, trotz adaptiven Schwellwertverfahren für die Binarisierung. Beispielsweise ein kräftiger Schatten, der quer durch den QR-Code verläuft.

Szenarien die komplexe Probleme für klassische auf Hough-Transformation basierende Detektoren bereitstellten, sind jedoch problemlos gelöst worden. Auch mehrere QR-Codes in einem Bild direkt nebeneinander oder über das Bild verteilt stellen kein Problem dar, solange alle Finder Patterns gefunden werden.

Auch hier zeigt sich wieder, wie schon bei der synthetischen Datenbank, dass die Lokalisierung der Finder Patterns das derzeit größte Problem für den implementierten Detektor darstellt.