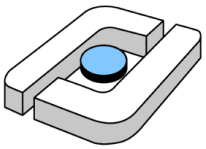

# Hochschule Osnabrück

University of Applied Sciences

## Fakultät

## Ingenieurwissenschaften und Informatik

Schriftliche Projektarbeit zum Thema:

## Entwicklung eines QR-Code Scanners

im Rahmen des Moduls
Bildverarbeitung,
des Studiengangs Informatik-Medieninformatik


| | |
|---|---|
| Autor: | Arpad Horvath |
| Matr.-Nr.: | 1056196 |
| E-Mail: | arpad.horvath@hs-osnab-rueck.de |
| Dozent: | Prof. Dr. Bernhard Lang |

*Inhaltsverzeichnis*

1 Einleitung ..... 1

1.1 Vorstellung des Themas ..... 1

1.2 Ziel der Ausarbeitung ..... 1

1.3 Aufbau der Projektarbeit ..... 1

1.4 Abgrenzung ..... 1

2 Darstellung der Grundlagen ..... 2

2.1 Einführung in die Grundlagen ..... 2

2.2 Geschichte und Anwendung von QR-Codes ..... 2

2.2.1 Ursprung und Entwicklung ..... 2

2.2.2 Anwendungsgebiete ..... 3

2.3 Aufbau und Struktur von QR-Codes ..... 4

2.3.1 QR-Code Versionen ..... 5

2.3.2 Finder-Patterns, Alignment-Patterns und Datenstruktur ..... 5

3 Implementierung des QR-Code-Scanners ..... 7

3.1 Gesamtübersicht der Architektur ..... 7

3.1.1 Vom Eingangsbild zur Dekodierung ..... 7

3.2 Vorverarbeitung des Bildes ..... 8

3.2.1 Graustufenumwandlung ..... 8

3.2.2 Rauschreduktion ..... 8

3.2.3 Binärisierung ..... 9

3.3 Erkennung der Finder-Patterns ..... 10

3.3.1 Labelling ..... 10
3.3.2 Geometrische Merkmale von QR-Codes ..... 11
3.3.3 Algorithmus zur Mustererkennung ..... 13
3.4 Perspektivkorrektur und Dekodierung ..... 16
3.4.1 Rotation ..... 16
3.4.2 Perspektivkorrektur ..... 17
3.4.3 Matrix-Interpretation ..... 22
4 Zusammenfassung und Fazit ..... 27
4.1 Zusammenfassung ..... 27
4.2 Mögliche Optimierungen und Erweiterungen ..... 28
4.3 Fazit 28
4.4 Verwendung von Künstlicher Intelligenz ..... 28
5 Quellen ..... 30

# Abbildungsverzeichnis

Abbildung 1 Aufbau eines QR-Codes .....4

Abbildung 2: Als Quadrate identifizierte Objekte .....12

Abbildung 3 Bestimmung der Eckpunkte .....14

Abbildung 4: Korrekt erkannte Ecken des QR-Codes .....16

Abbildung 5: Ergebnis der Rotation .....17

Abbildung 6 Herausgefilterte Linien (Links) und Bounding Box des QR-Codes (Rechts) .....19

Abbildung 7: Ausschnitt der unteren linken Ecke .....20

Abbildung 8: Samplepunkte und deren Korrektur .....25

Abbildung 9: Isolierter QR-Code .....26

Abbildung 10: Schwammige Binärisierung .....27

## Source-Code Verzeichnis

Snippet 1: Median Blur ..... 9

Snippet 2: Iteratives Verfahren zum Berechnen des Schwellenwertes ..... 10

Snippet 3: Struct zum Speichern von Labellobjekten ..... 11

Snippet 4 Erkennung von quadratischen Mustern ..... 11

Snippet 5: Bestimmung der drei Eckpunkte ..... 13

Snippet 6: Erkennung von L-Förmigen Objekten ..... 15

Snippet 7: Rotation des Bildes ..... 16

Snippet 8: Hough-Transformation ..... 18

Snippet 9: Festlegung der korrigierten Ecken ..... 20

Snippet 10: Berechnung der Homographie-Matrix ..... 21

Snippet 11: Bestimmung einer validen Modulanzahl ..... 23

Snippet 12: Bestimmung des Offsets für den Sampler ..... 24

# 1 Einleitung

## 1.1 Vorstellung des Themas

Im Rahmen des Moduls Bildverarbeitung wurde ein QR-Code-Scanner entwickelt, der die grundlegenden Prinzipien der Bildverarbeitung in der Praxis umsetzt. Der Scanner analysiert Bilder, um QR-Codes zu erkennen und zu lesen.

## 1.2 Ziel der Ausarbeitung

Ziel dieses Projekts ist es, die im Modul Bildverarbeitung erlernten Konzepte praktisch anzuwenden und ein vertieftes Verständnis für die Funktionsweise und Herausforderungen bei der Verarbeitung von QR-Codes zu erlangen. Die Projektarbeit beschreibt die Entwicklung eines QR-Code-Scanners, wobei sowohl die technischen Grundlagen als auch die Implementierung der Bildverarbeitungsmethoden detailliert erläutert werden. Zudem wird erörtert, welche Herausforderungen bei der Erkennung von QR-Codes auftreten und wie diese durch geeignete Bildverarbeitungsverfahren adressiert wurden.

## 1.3 Aufbau der Projektarbeit

Zunächst werden im Grundlagenteil die theoretischen Konzepte der QR-Code-Erkennung erläutert. Im Hauptteil folgt die Beschreibung der Implementierung, wobei die einzelnen Verarbeitungsschritte sowie die verwendeten Algorithmen detailliert erklärt werden. Abschließend fasst das Fazit die wichtigsten Erkenntnisse zusammen und gibt einen Ausblick auf mögliche Erweiterungen.

## 1.4 Abgrenzung

Der Fokus dieser Projektarbeit liegt auf den theoretischen Grundlagen der Bildverarbeitung und deren praktischer Anwendung anhand eines QR-Code-Scanners. Die Bilder werden nicht in Echtzeit verarbeitet, sondern müssen manuell dem Programm als Argumente übergeben werden. Da der Schwerpunkt auf den theoretischen Konzepten liegt, wurde keine Performance-Optimierung vorgenommen. Der erkannte QR-Code wird nicht dekodiert. Der Fokus liegt ausschließlich auf der Erkennung seiner Position. Die QR-Code Matrix wird gelesen, aber nicht weiterverarbeitet. Ziel ist es, die Bildverarbeitungsmethoden zu verstehen und anzuwenden, nicht eine hoch performante Lösung zu entwickeln.

# 2 Darstellung der Grundlagen

## 2.1 Einführung in die Grundlagen

In diesem Kapitel werden die wesentlichen fachlichen Grundlagen erläutert, die der Entwicklung des QR-Code-Scanners zugrunde liegen. Ziel ist es, Wissenslücken der Leser zu schließen und darzulegen, dass diese Arbeit dem aktuellen Stand der Technik und Wissenschaft entspricht.

## 2.2 Geschichte und Anwendung von QR-Codes

### 2.2.1 Ursprung und Entwicklung

Der QR-Code (Quick Response Code) wurde 1994 von der japanischen Firma Denso Wave, einer Tochtergesellschaft der Toyota-Gruppe, entwickelt, um die Effizienz in der Automobilproduktion zu steigern. Grund dafür: herkömmliche eindimensionale Barcodes können nur begrenzte Informationen speichern. Der QR-Code ermöglichte es, große Datenmengen schnell und zuverlässig zu verarbeiten, was die Produktionsprozesse optimierte [1].

Masahiro Hara und sein Team bei Denso Wave entwickelten den QR-Code. Masahiro Hara ließ sich vom japanischen Brettspiel Go inspirieren, bei dem schwarze und weiße Steine auf einem Gitter liegen. Dies führte ihn zur Idee eines zweidimensionalen Codes zur Speicherung großer Informationsmengen [2].

Mit der ISO-Standardisierung im Jahr 2000 (ISO 18004) verbreitete sich der QR-Code international [3]. Anfangs vor allem in der Industrie genutzt, fand der QR-Code mit der Verbreitung von Smartphones und mobilen Anwendungen ab den 2010er Jahren vermehrt Anwendung im Marketing, Ticketing und anderen Bereichen des täglichen Lebens [4].

### 2.2.2 Anwendungsgebiete


*Abbildung 1: Anwendungsbereiche von QR-Codes [5] [6] [7]*

Abbildung 1 zeigt die Verbreitung von QR-Codes und gibt einen Einblick in die Zahlreichen Anwendungsgebiete. Seit ihrer Einführung haben sich QR-Codes in zahlreichen Bereichen etabliert und bieten vielfältige Einsatzmöglichkeiten [8]:

- **WLAN-Zugang**: Durch das Scannen eines QR-Codes erhalten Nutzer einfachen Zugang zu WLAN-Netzwerken, ohne Passwörter manuell eingeben zu müssen.
- **Marketing und Werbung:** Unternehmen nutzen QR-Codes in Anzeigen, Plakaten und Verpackungen, um Kunden direkt zu digitalen Inhalten wie Websites, Videos oder Sonderangeboten zu leiten.
- **Produktverpackungen**: Durch das Scannen von QR-Codes auf Verpackungen können Verbraucher zusätzliche Informationen über Produkte erhalten, wie z. B. Herkunft, Inhaltsstoffe oder Anwendungshinweise.
- **Druckmedien**: In Zeitungen und Magazinen werden QR-Codes verwendet, um Leser zu weiterführenden Artikeln, Videos oder interaktiven Inhalten zu leiten.
- **Zahlungsverkehr**: Im Zahlungsverkehr werden QR-Codes für schnelle Transaktionen genutzt, beispielsweise bei mobilen Bezahlsystemen wie PayPal.

## 2.3 Aufbau und Struktur von QR-Codes

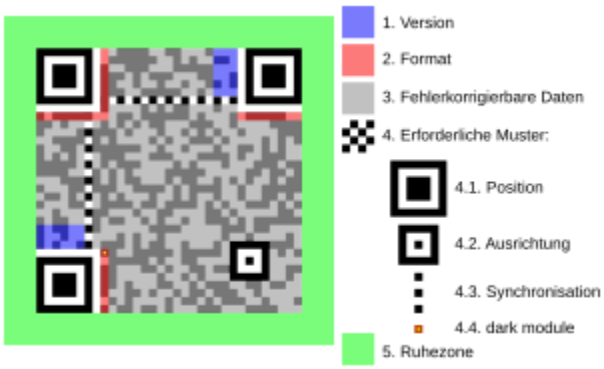


*Abbildung 2 Aufbau eines QR-Codes [1]*

Das quadratische Muster eines QR-Codes bietet eine höhere Datenkapazität im Vergleich zu herkömmlichen Barcodes. Zudem kann er von Bildverarbeitungsprogrammen leichter erkannt werden. Dank einer Fehlerkorrektur von 7 bis 30 % ist der QR-Code widerstandsfähig gegenüber Beschädigungen oder Verschmutzungen [9]. Abbildung 2 zeigt den Aufbau eines QR-Codes:

1. **Version**: Gibt an, welche Version des QR-Codes verwendet wird.
2. **Format**: Speichert Metadaten wie die Fehlerkorrekturstufe und die verwendete Kodierung.
3. **Datenbereich:** enthält die kodierten Informationen sowie redundante Daten zur Fehlerkorrektur.
4. **Erforderliche Muster**:
   - 4.1. **Finder-Patterns**: Drei charakteristische quadratische Muster in den Ecken des Codes, die die Lokalisierung und Ausrichtung des Codes ermöglichen.
   - 4.2. **Alignment-Patterns**: Zusätzliche Muster zur Korrektur perspektivischer Verzerrungen, insbesondere bei größeren QR-Codes.
   - 4.3. **Timing-Patterns**: Alternierende Schwarz-Weiß-Module, die eine präzise Bestimmung der Zellengröße ermöglichen.
5. **Ruhezone**: Ein leerer Bereich um den QR-Code, der eine störungsfreie Erkennung gewährleistet.

### 2.3.1 QR-Code Versionen

Die Version eines QR-Codes bestimmt dessen Größe und Struktur. Sie gibt an, wie viele Module (kleine schwarze und weiße Zellen) im quadratischen Raster des Codes angeordnet sind. Es gibt insgesamt 40 Versionen von QR-Codes, wobei die Versionen 1 bis 40 unterschiedliche Dimensionen und Datenkapazitäten bieten. Jede Version verfügt über eine festgelegte Anzahl an Modulen und unterschiedliche Anzahl an Fehlerkorrektur-Codes, die es ermöglichen, Daten selbst bei Beschädigung des Codes zu rekonstruieren. Je höher die Version, desto mehr Module enthält der QR-Code, was zu einer höheren Datenkapazität führt. Die Anzahl der Module lässt sich mit der folgenden Formel berechnen:

$$\text{Modulanzahl} = 21 + 4 \times (Version - 1)$$

Diese Formel zeigt, dass die Modulanzahl mit jeder Version um vier zunimmt. Die kleinste Version (Version 1) hat demnach eine Größe von 21x21 Modulen und die größtmögliche Version 40 hat eine Größe von 177x177 Modulen.

### 2.3.2 Finder-Patterns, Alignment-Patterns und Datenstruktur

QR-Codes bestehen aus einer präzisen Struktur von Mustern und Symbolen, die zusammen eine effiziente und fehlerresistente Kodierung ermöglichen. Drei zentrale Elemente dieser Struktur sind die Finder-Patterns, Alignment-Patterns und die Datenstruktur, die maßgeblich zur Funktionsweise eines QR-Codes beitragen.

#### *2.3.2.1 Die Finder Patterns*

Die Finder-Patterns sind die drei markanten quadratischen Muster, die sich in den oberen linken, oberen rechten und unteren linken Ecken eines QR-Codes befinden. Sie bestehen aus einem schwarzen 3 × 3 Modul großen Quadrat, das von einem weißen und einem weiteren schwarzen Rahmen umgeben ist, wodurch ihre Gesamtgröße 7 × 7 Module beträgt.

Sie dienen der Lokalisierung des Codes und seiner Orientierung. Die Position und Ausrichtung der Finder-Patterns ermöglichen es Bildverarbeitungsprogrammen, den QR-Code auch bei teilweiser Verzerrung korrekt zu erkennen.

#### *2.3.2.2 Die Alignment-Patterns*

Die Alignment-Patterns ergänzen die Finder-Patterns und dienen dazu, perspektivische Verzerrungen zu korrigieren, die beim Scannen eines QR-Codes auftreten können. Sie

sind besonders wichtig, wenn QR-Codes auf gekrümmten oder geneigten Oberflächen gedruckt werden.

Alignment-Patterns befinden sich nicht an den Ecken, sondern an strategischen Punkten im Code, abhängig von der Version des QR-Codes. Bei größeren Versionen (ab Version 2) gibt es mehrere dieser Muster, die gleichmäßig über den Code verteilt sind. Diese Muster ermöglichen es dem Decoder, Verzerrungen zu korrigieren und die exakte Form des QR-Codes zu bestimmen. Sie tragen zur Stabilität und Genauigkeit der Decodierung bei, indem sie die Geometrie des Codes korrigieren.

#### *2.3.2.3 Datenstruktur und Fehlerkorrektur*

Die Datenstruktur eines QR-Codes ist in einer Matrix aus Modulen organisiert, die verschiedene funktionale Bereiche umfasst (siehe Abschnitt 2.2.3). Diese sorgen für eine stabile Erkennung, zuverlässige Datenkodierung und Fehlerkorrektur. Für die Fehlerkorrektur wird das Reed-Solomon-Verfahren verwendet, das eine Rekonstruktion von bis zu 30 % der beschädigten oder fehlenden Daten erlaubt [10]. Die Stärke der Fehlerkorrektur kann je nach Anwendungsfall angepasst werden, wobei vier Stufen zur Verfügung stehen: Low (L), Medium (M), Quartile (Q) und High (H). Eine höhere Stufe verbessert die Lesbarkeit des Codes in schwierigen Bedingungen, reduziert jedoch die maximale Datenkapazität.

# 3 Implementierung des QR-Code-Scanners

In diesem Kapitel wird der gesamte Prozess der QR-Code-Erkennung und -Extraktion beschrieben. Dabei wird detailliert erläutert, wie die verschiedenen Bildverarbeitungsmethoden und Algorithmen konkret in der Implementierung des QR-Code-Scanners zum Einsatz kommen. Ziel ist es, den gesamten Prozess – vom Einlesen des QR-Codes bis zur erfolgreichen Extraktion des Codes – nachvollziehbar zu machen und die technische Umsetzung im Detail zu erläutern.

## 3.1 Gesamtübersicht der Architektur

Der QR-Code-Scanner folgt einer strukturierten Architektur, die durch eine sequenzielle Verarbeitungspipeline aus mehreren Bildverarbeitungsschritten realisiert wird. Der Pfad zum Eingangsbild wird der Anwendung als Parameter übergeben. Während der Verarbeitung werden sowohl das Endergebnis als auch die Zwischenbilder, die während der einzelnen Verarbeitungsschritte entstehen, im Verzeichnis des Ursprungsbildes gespeichert. Das Endergebnis ist ein ausgeschnittener und korrekt ausgerichteter QR-Code mit einer Auflösung von Modulanzahl × Modulanzahl entsprechend der QR-Code-Version. Beispielsweise hat ein QR-Code der Version 9 eine Größe von 53 × 53 Pixeln.

Zur Unterstützung der Berechnungen wurden eine eigene Vektor Klasse und eine Matrix Klasse aus dem Modul "Computergrafik" verwendet. Diese Klassen ermöglichen es, die Koordinaten im Bild als Vektoren darzustellen und komplexe Operationen effizient durchzuführen. Zusätzlich wurde die Bibliothek Eigen an einer Stelle verwendet, um die Homographie-Gleichung beim Entzerren des Bildes zu lösen.

### 3.1.1 Vom Eingangsbild zur Dekodierung

Auf dem Weg zum finalen Bild durchläuft das Eingangsbild eine Verarbeitungspipeline, die schrittweise die notwendigen Bildverarbeitungsoperationen ausführt. Diese Pipeline lässt sich in die folgenden Hauptschritte unterteilen:

1. **Vorverarbeitung**: Hier wird das Bild für die weitere Analyse optimiert. Dazu gehören Schritte wie Graustufenumwandlung, Rauschreduktion und Binärisierung, um die kommenden Schritte zu erleichtern.

2. **Mustererkennung**: In diesem Schritt werden charakteristische Finder-Patterns identifiziert, die zur Lokalisierung und Ausrichtung des Codes dienen

3. **Transformation**: Nach der Identifikation der relevanten Muster wird das Bild so verzerrungsfrei wie möglich transformiert, um eine exakte, rechteckige Darstellung des QR-Codes zu erhalten.

4. **Einlesen und Dekodierung**: Im letzten Schritt wird die extrahierte Code-Matrix aus dem transformierten Bild ausgelesen. Anstatt die binären Informationen weiter zu dekodieren, wird diese Matrix direkt auf der Konsole ausgegeben. Eine weitere Übersetzung der binären Daten in lesbaren Text oder andere Inhalte erfolgt in dieser Implementierung nicht, da gemäß der in Kapitel 1.4 (Abgrenzung) definierten Einschränkung nur die Ausgabe der Matrix erforderlich ist

## 3.2 Vorverarbeitung des Bildes

Die Vorverarbeitung eines Bildes ist ein entscheidender Schritt, um es für die anschließende QR-Code-Erkennung vorzubereiten. Durch die Anwendung verschiedener Bildverarbeitungstechniken werden unnötige Details entfernt, und die relevanten Merkmale für die QR-Code-Dekodierung hervorgehoben

### 3.2.1 Graustufenumwandlung

Die Graustufenumwandlung entfernt Farbdetails und vereinfacht somit die Bildanalyse. Die Umwandlung erfolgt durch die Funktion *greyscale(Img<RGB_Pixel> rgb)*, die jedes Pixel von einem RGB-Wert in einen einzelnen Grauwert überführt. Dies geschieht durch eine gewichtete Summe der Farbkanäle nach der folgenden Formel:

$$\text{Grauwert} = 0.299 \times R + 0.587 \times G + 0.114 \times B$$

Dieser Grauwert repräsentiert die Helligkeit des Pixels. Da menschliche Augen für grüne Töne empfindlicher sind, wird der Grünanteil stärker gewichtet [11].

### 3.2.2 Rauschreduktion

Störungen und Bildartefakte, die durch Rauschen entstehen, können die spätere Erkennung des QR-Codes erschweren. Um diese zu minimieren, wird ein Mittelwertfilter auf das Bild angewendet. Dieser Filter ersetzt den Wert jedes Pixels durch den Durchschnittswert seiner benachbarten Pixel. Dies hilft, zufällige Bildfehler zu glätten und das Bild klarer zu machen. Snippet 1 zeigt die Implementierung. Hier werden lediglich die unmittelbaren Nachbarn betrachtet. Wie sich ein stärkerer Blur-Effekt auf die Erkennungsrate auswirkt, bleibt noch zu testen.

```
for (int dy = -1; dy <= 1; ++dy) {
    for (int dx = -1; dx <= 1; ++dx) {
        if (x + dx >= 0 && x + dx < width && y + dy >= 0 && y + dy <
height) {
            sum += gray[y + dy][x + dx];
            ++count;
        }
    }
}
blur[y][x] = sum / count;
```

*Snippet 1: Median Blur*

### 3.2.3 Binärisierung

Der letzte Schritt der Vorverarbeitung ist die Binärisierung des Bildes, bei der das Bild in eine reine Schwarz-Weiß-Darstellung umgewandelt wird. Dies ist notwendig, da QR-Codes aus klar abgegrenzten schwarzen und weißen Modulen bestehen.

Die Binärisierung basiert auf der Schwellwertmethode (Thresholding). Jeder Pixel wird anhand eines Schwellwerts T entweder als schwarz (0) oder weiß (1) klassifiziert.

$$I'(\boldsymbol{x},\boldsymbol{y}) = \begin{cases} 1, wenn\ I(x,y) > T \\ 0, wenn\ I(x,y) \leq T \end{cases}$$

Hierbei stellt I(x,y) den Grauwert eines Pixels dar, während I′(x,y)I′(x,y) das binäre Ergebnis ist. Anstelle eines festen Schwellwertes T wird dieser dynamisch ermittelt, um die Helligkeitsverteilung im Bild optimal widerzuspiegeln. Dies geschieht mit einer iterativen Methode, die den optimalen Wert anhand der Helligkeitsverteilung im Bild berechnet (Snippet 2).

```cpp
unsigned int optimal_threshold(const Img<unsigned char> &gray_image) {

    unsigned int T = 128;
    unsigned int prev_T;

    do {
        prev_T = T;
        unsigned int sum1 = 0, sum2 = 0;
        unsigned int count1 = 0, count2 = 0;

        for (unsigned int y = 0; y < CurrentImageHeight; ++y) {
            for (unsigned int x = 0; x < CurrentImageWidth; ++x) {
                unsigned char pixel = gray_image[y][x];
                if (pixel < T) {
                    sum1 += pixel;
                    count1++;
                } else {
                    sum2 += pixel;
                    count2++;
                }
            }
        }

        unsigned int V1 = (count1 == 0) ? 0 : (sum1 / count1);
        unsigned int V2 = (count2 == 0) ? 0 : (sum2 / count2);

        T = (V1 + V2) / 2;
    } while (T != prev_T);

    return T;
}
```

*Snippet 2: Iteratives Verfahren zum Berechnen des Schwellenwertes*

Zunächst wird der Schwellwert T auf 128 gesetzt. Anschließend wird das Bild in zwei Gruppen unterteilt: dunkle Pixel (I[x,y]<T) und helle Pixel (I[x,y]≥T). Für jede Gruppe wird der Mittelwert berechnet, und der neue Schwellwert ergibt sich als

$$T = \frac{V_1 + V_2}{2}$$

Dieser Prozess wird iterativ wiederholt, bis sich der Schwellwert T nicht mehr ändert, und somit ein stabiler Wert erreicht wird.

## 3.3 Erkennung der Finder-Patterns

### 3.3.1 Labelling

Die Erkennung der Finder Patterns basiert auf einem Labelling-Verfahren, das bereits in der Vorlesung und den praktischen Übungen behandelt wurde. In diesem Bericht wird

daher nicht weiter auf die Details dieses Verfahrens eingegangen, sondern nur auf die Anwendung und die daraus resultierenden Daten.

```
struct ObjectProperties {
    Vector touch_point;
    unsigned int label;
    unsigned int x_min, x_max, y_min, y_max;
    unsigned int area;
    Vector center;
    RGB_Pixel color;
};
```

*Snippet 3: Struct zum Speichern von Labellobjekten*

Das Labelling dient dazu, zusammenhängende Regionen im binären Bild zu erkennen und sie zu kennzeichnen. Nach der Anwendung des Labelling-Verfahrens werden die erkannten Objekte extrahiert und in einer speziellen Datenstruktur gespeichert, um weiter verarbeitet zu werden (siehe Snippet 3). Besonders ausschlaggebend sind der Mittelpunkt und die Bounding Box.

### 3.3.2 Geometrische Merkmale von QR-Codes

Die Finder Patterns eines QR-Codes sind quadratisch, was einen wichtigen Hinweis für deren Erkennung gibt. Der erste Schritt besteht darin, alle potenziellen quadratischen Objekte im Bild zu extrahieren und auf ihre Eignung als Finder Pattern zu überprüfen. Um die Erkennung zu erleichtern, müssen quadratische Objekte anhand ihrer geometrischen Merkmale identifiziert werden (Snippet 4).

```
bool IsSquare(const ObjectProperties &obj, const unsigned int MIN_AREA,
const unsigned int MAX_AREA) {
    unsigned int width = obj.x_max - obj.x_min + 1;
    unsigned int height = obj.y_max - obj.y_min + 1;

    // Seitenverhältnis prüfen
    if (abs((int) width - (int) height) > max(width, height) * 0.2) {
        return false;
    }

    if (obj.area < MIN_AREA || obj.area > MAX_AREA) {
        return false;
    }

    return true;
}
```

*Snippet 4 Erkennung von quadratischen Mustern*

Da die Finder Patterns annähernd quadratisch sind, sollten Breite und Höhe möglichst gleich sein. Ein gewisser Toleranzbereich kann zugelassen werden, um leichte Verzerrungen auszugleichen. Die Flächenanalyse dient dazu, überproportional große Elemente zu filtern und auch sehr kleine Objekte auszuschließen, die möglicherweise durch Artefakte oder Rauschen entstanden sind.

Abbildung 3 zeigt eine visuelle Darstellung der Objekte, die in der Analyse von ihrer Bounding Box als quadratisch erkannt wurden und somit als potenzielle Finder Patterns für die weitere Verarbeitung dienen.

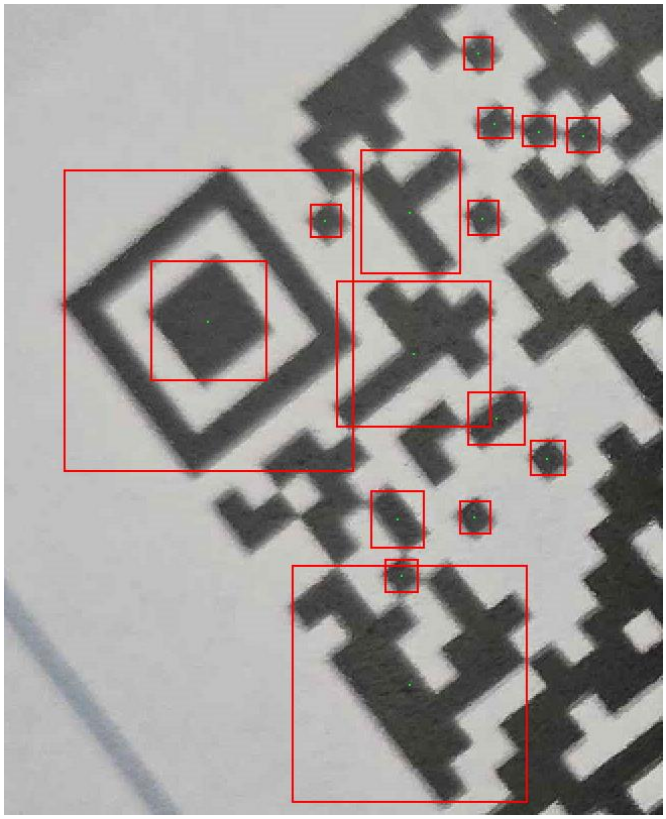

*Abbildung 3: Als Quadrate identifizierte Objekte*

### 3.3.3 Algorithmus zur Mustererkennung

```
void AssignProps(const ObjectProperties &a, const ObjectProperties &b,
const ObjectProperties &c, Vector &topLeft,
                Vector &topRight,
                Vector &bottomLeft) {

    Vector av = a.center;
    Vector bv = b.center;
    Vector cv = c.center;

// Schritt 1: Entfernung zwischen den Punkten  Bestimmen
    float d1 = (bv - av).length(); // AB
    float d2 = (cv - av).length(); // AC
    float d3 = (cv - bv).length(); // BC


    // Schritt 2: Hypotenuse bestimmen
    float max_d = max(d1, max(d2, d3));
    Vector midPoint;

    if (max_d == d1) { // AB ist die Hypotenuse

        midPoint = av + (ab * 0.5f);
        Vector cm = cv - midPoint;
        Matrix mrot;
        mrot.rotationAxis(Vector(0, 0, 1), PI / 2);
        cm = mrot * cm;
        cm = cm + cv; //refrence point

        if ((cm - av).length() > (cm - bv).length()) {
            topLeft = cv;
            topRight = av;
            bottomLeft = bv;
        } else {
            topLeft = cv;
            topRight = bv;
            bottomLeft = av;
        }


    } else if (max_d == d2) { // AC is the hypotenuse
      ...
    } else if (max_d == d3) { // BC is the hypotenuse
      ...
    }
}
```

*Snippet 5: Bestimmung der drei Eckpunkte*

Der Algorithmus zur Mustererkennung für die Finder Patterns besteht aus mehreren Schritten, die auf den zuvor extrahierten und gefilterten Objekten basieren. Im Wesentlichen wird überprüft, ob die erkannten Objekte ein L-förmiges Muster aufweisen, das für

die Finder Patterns charakteristisch ist. In Snippet 5 bestimmt die Funktion *Assign-Props()* zunächst, welcher Punkt am ehesten den drei Eckpunkten des QR-Codes entspricht.

Die Funktion zeigt die Logik zur Bestimmung der Zuordnung der Punkte auf dem durch die drei Ecken gebildeten Dreieck. Die längste Entfernung stellt die Hypotenuse dar, die aufgrund der Diagonalen immer länger ist als die beiden anderen Seiten, die zwischen der oberen linken (OL) und der unteren linken (UL) Ecke sowie der OL und der oberen rechten (OR) Ecke verlaufen. Die Hypotenuse verbindet somit die OR- und UL-Ecken im QR-Code.

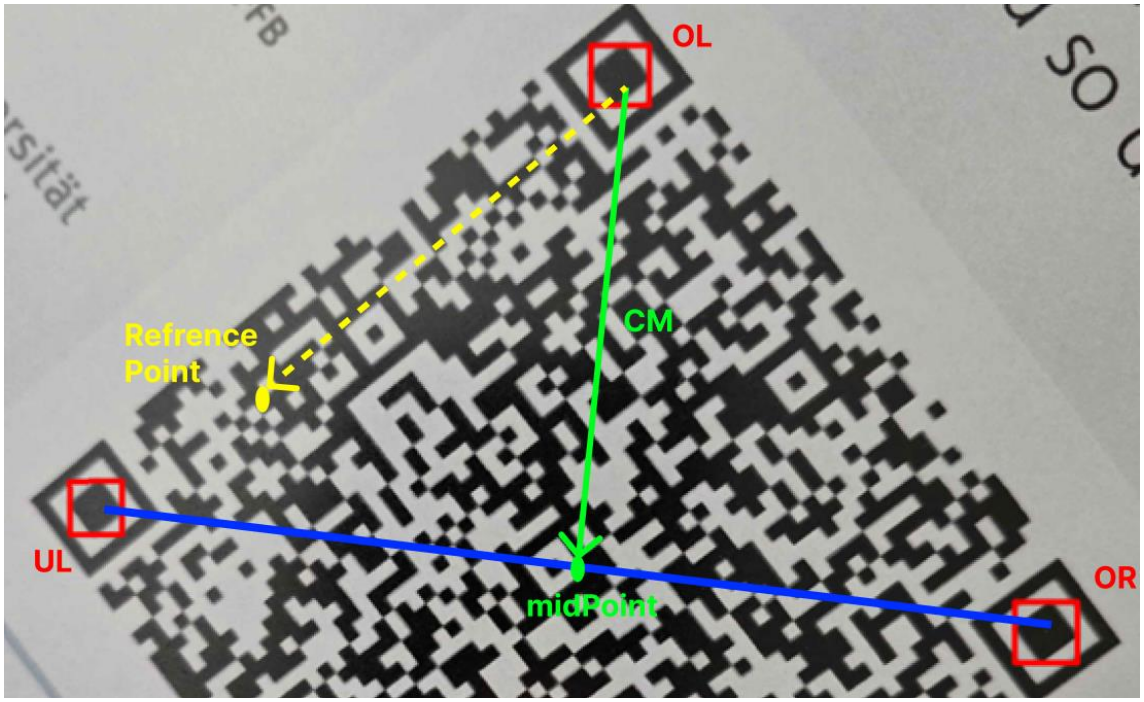


*Abbildung 4 Bestimmung der Eckpunkte*

Anhand dieser Information lässt sich die OL-Ecke automatisch bestimmen, da sie als einzige nicht auf der Hypotenuse liegt. Um OR und Ul zu bestimmen, wird Vektor CM (Von der OL-Ecke zum Mittelpunkt des Quadrats) um PI/2 im Uhrzeigersinn gedreht. Das Ergebnis ist ein Referenzpunkt, der eindeutig näher an der UL-Ecke liegt. Dieses Verfahren wird in Abbildung 4 visualisiert. Durch diese geometrischen Eigenschaften können die Ecken des QR-Codes korrekt identifiziert werden.

```
bool IsLShape(const ObjectProperties &a, const ObjectProperties &b, const
ObjectProperties &c) {


    Vector topLeft;
    Vector topRight;
    Vector bottomLeft;
    AssignProps(a, b, c, topLeft, topRight, bottomLeft);


    float dot = ((topLeft - topRight).normalize()).dot((topLeft -
bottomLeft).normalize());

    if (dot - 0.1 < 0 && dot + 0.1 > 0) {
        if ((topLeft - topRight).length() / (topLeft - bottomLeft).length()
> 0.9 &&
            (topLeft - topRight).length() / (topLeft - bottomLeft).length()
< 1.1) {
            return true;
        }

    }
    return false;

}
```

*Snippet 6: Erkennung von L-Förmigen Objekten*

Im letzten Schritt in Snippet 6 wird überprüft, ob die Verbindungslinien zwischen den Punkten des Dreiecks ein rechtwinkliges Dreieck bilden. Zusätzlich werden die Längenverhältnisse der Ankathete und Gegenkathete überprüft. Dies ist notwendig, um sicherzustellen, dass es sich bei den erkannten Mustern tatsächlich um Finder Patterns des QR-Codes handelt.

Durch die Anwendung dieser Verfahren können die Finder Patterns lokalisiert werden, wie in Abbildung 5 dargestellt. Die drei identifizierten Ecken bilden die Grundlage für die weitere Verarbeitung und Dekodierung des QR-Codes.

*Abbildung 5: Korrekt erkannte Ecken des QR-Codes*

## 3.4 Perspektivkorrektur und Dekodierung

Nachdem die Position des QR-Codes bestimmt wurde, erfolgt als nächster Schritt die Rotation des Bildes, um die Ausrichtung zu korrigieren. Im Anschluss daran wird eine Perspektivkorrektur durchgeführt. Diese Korrektur erleichtert das Sampling im letzten Schritt.

### 3.4.1 Rotation

```
Vector horizontal = Vector(1, 0, 0);
Vector topEdge = topRight - topLeft;
topEdge.normalize();

float angle = acos(horizontal.dot(topEdge)); // Angle with x-axis
if (topEdge.Y < 0) {
    angle = -angle; // Adjust for negative rotation
}
.
.
.
rotVect[0] = 0;
rotVect[1] = 0;
rotVect[2] = angle;
RotMat_from_Rodriguez(R, rotVect);

UndistoreImage(img, intrinsic_d, Binaerbild, intrinsic_d, distCoeffs, rot-
Vect);
```

*Snippet 7: Rotation des Bildes*

Basierend auf den identifizierten Ecken kann der Winkel, wie in Snippet 7 dargestellt, berechnet werden. Dieser Winkel ergibt sich aus dem Skalarprodukt zwischen der Horizontalen X-Achse und der oberen Kante, die die Ecken OL und OR verbindet. Dieser Winkel wird im Rotationsvektor verwendet. Unter Zuhilfenahme neutraler intrinsischer Parameter und der Funktion UndistortImage() aus dem ersten Praktikum wird das Bild entsprechend rotiert, um eine korrekte Ausrichtung des QR-Codes zu gewährleisten. Wie in Abbildung 6 dargestellt, wird das Bild nach der Rotation noch immer durch die Kameraperspektive beeinflusst. Das Quadrat des QR-Codes erscheint in diesem Fall leicht trapezförmig. Diese Verzerrung ist eine natürliche Folge der Kameraperspektive und muss in einem nächsten Schritt durch eine Perspektivkorrektur ausgeglichen werden.

*Abbildung 6: Ergebnis der Rotation*

### 3.4.2 Perspektivkorrektur

Um die Ecken des QR-Codes zu ermitteln, verwenden wir die Kanten der trapezförmigen Bounding Box, die den QR-Code umgibt. Da das Bild binär ist, kann eine neighborhood-based edge detection (kantenerkennende Methode auf Basis von benachbarten Pixeln) eingesetzt werden. Diese Methode erfordert keine zusätzliche Gewichtung wie etwa bei

der Sobel-Kantenerkennung, da die Bilddaten bereits auf Schwarz-Weiß-Pixel reduziert sind.

```
for (unsigned int y = minY; y < maxY; ++y) {
    for (unsigned int x = minX; x < maxX; ++x) {
        if (input[y][x]) {
            for (int theta = 0; theta < theta_dim; ++theta) {
                double radian = theta * M_PI / 180.0;
                double rho = x * cos(radian) + y * sin(radian);
                int rho_index = static_cast<int>(rho + rho_max);

                if (rho_index >= 0 && rho_index <
static_cast<int>(rho_dim)) {
                    hough[rho_index][theta]++;
                }
            }
        }
    }
}
vector<Line> lines = ExtractStrongestLines(hough, rho_max, theta_dim);
vector<Vector> corners = refineQRBounds(lines, qrCodeL);
```

*Snippet 8: Hough-Transformation*

In Snippet 8 wird die Hough-Transformation auf das Kanten Bild angewendet, wobei die Transformation nur auf den Bereich des QR-Codes angewendet wird, der durch die Variablen minX, minY, maxX und maxY festgelegt wird. Dies stellt sicher, dass die Berechnungen auf den Bereich beschränkt werden, in dem der QR-Code vermutet wird, was die Berechnungszeit optimiert.

Die Hough-Transformation funktioniert, indem jedes Kante-Pixel im Bild durch die Parameter theta (Winkel) und rho (Abstand zur Ursprungslinie) im Hough-Raum abgebildet wird. Jeder Wert von rho und theta wird für jedes Kante-Pixel berechnet, und das Ergebnis wird in einem sogenannten Hough-Akkumulator gespeichert. Der Akkumulator zählt „Stimmen“ für jedes mögliche Linienpaar.

Am Ende der Transformation enthält das Hough-Spektrum die Linien, die im Bild am stärksten vertreten sind, d. h. die mit den meisten Stimmen. Diese Linien entsprechen den Kanten im Bild, die am wahrscheinlichsten Teil des QR-Codes sind.

*Abbildung 7 Herausgefilterte Linien (Links) und Bounding Box des QR-Codes (Rechts)*

Die Funktion *refineQRBounds()* filtert die Linien, die die Ecken des QR-Codes definieren. Diese Linien verlaufen überwiegend vertikal oder horizontal und grenzen den QR-Code ein. Durch die Analyse der Abstände zu den vermuteten Ecken kann die nächste Kante auf jeder Seite (links, rechts, oben, unten) des QR-Codes bestimmt werden. Die Schnittpunkte dieser Kanten stellen die genauen Ecken des verformten QR-Codes dar, die nun als Grundlage für die Perspektivkorrektur dienen. Abbildung 7 veranschaulicht diesen Entscheidungsprozess.

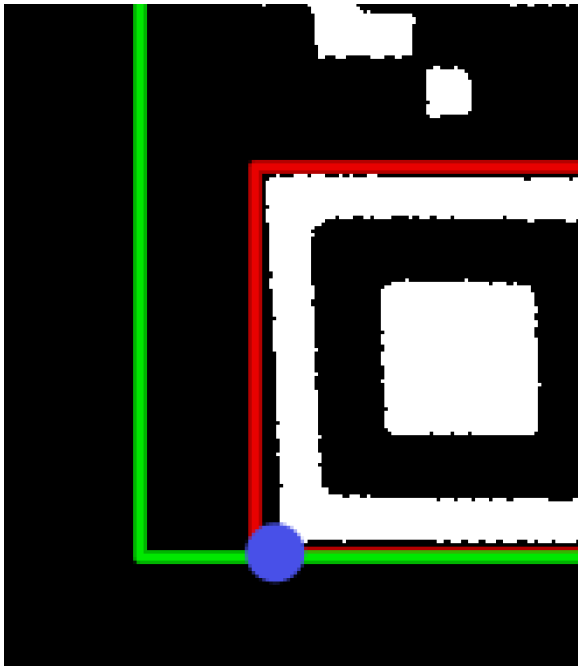

*Abbildung 8: Ausschnitt der unteren linken Ecke*

In der Abbildung 8, die die untere linke Ecke des QR-Codes zeigt, verdeutlicht der blaue Punkt die Position des Eckpunkts des verformten QR-Codes, während die grünen Linien den Verlauf eines perfekten Quadrats darstellen. Dies verdeutlicht die perspektivische Verzerrung des QR-Codes. Um die Perspektive zu korrigieren, werden die X-Koordinaten der oberen linken (OL) und oberen rechten (OR) Punkte mit den Y-Koordinaten der unteren beiden Punkte kombiniert. Dadurch wird ein unverzerrtes Quadrat erzeugt, das die ideale Form des QR-Codes darstellt (Snippet 9).

```
vector<Vector> correctedCorners = {
        topLeft,
        topRight,
        Vector(topLeft.X, bottomLeft.Y, 0),
        Vector(topRight.X, bottomLeft.Y, 0)
};
```

*Snippet 9: Festlegung der korrigierten Ecken*

Mithilfe der tatsächlichen und der korrigierten Punkte können wir eine Homographie-Matrix berechnen, die die verzerrten Punkte auf die idealen (korrigierten) Punkte projiziert. Für diese Matrix gilt:

$$s\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = H \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Die Funktion *computeHomography()* in Snippet 10, berechnet die Homographie-Matrix H. Diese Matrix ermöglicht die Transformation einer Menge von Punkten von einer Ebene auf eine andere und beschreibt eine projektive Transformation zwischen den beiden Ebenen.

```
Eigen::Matrix3d computeHomography(const vector<Vector> &corners, const
vector<Vector> &correctedCorners) {
    Eigen::MatrixXd A(8, 8);
    Eigen::VectorXd b(8);

    for (int i = 0; i < 4; i++) {
        double x = corners[i].X;
        double y = corners[i].Y;
        double xp = correctedCorners[i].X;
        double yp = correctedCorners[i].Y;

        A(2 * i, 0) = x;
        A(2 * i, 1) = y;
        A(2 * i, 2) = 1;
        A(2 * i, 3) = 0;
        A(2 * i, 4) = 0;
        A(2 * i, 5) = 0;
        A(2 * i, 6) = -x * xp;
        A(2 * i, 7) = -y * xp;
        b(2 * i) = xp;

        A(2 * i + 1, 0) = 0;
        A(2 * i + 1, 1) = 0;
        A(2 * i + 1, 2) = 0;
        A(2 * i + 1, 3) = x;
        A(2 * i + 1, 4) = y;
        A(2 * i + 1, 5) = 1;
        A(2 * i + 1, 6) = -x * yp;
        A(2 * i + 1, 7) = -y * yp;
        b(2 * i + 1) = yp;
    }

    Eigen::VectorXd h = A.colPivHouseholderQr().solve(b);

    Eigen::Matrix3d H;
    H << h(0), h(1), h(2),
            h(3), h(4), h(5),
            h(6), h(7), 1.0;

    return H;
}
```

*Snippet 10: Berechnung der Homographie-Matrix*

Die Funktion nimmt 4 Paare von den jeweiligen Eckpunkten. Corners (ursprüngliche Punkte) und correctedCorners (korrigierte Punkte). Jedes Paar besteht aus den Koordinaten eines Punktes in beiden Ebenen. Für jedes Paar von Punkten (x,y) und (x′,y′), das durch die Homographie transformiert werden soll, gibt es zwei lineare Gleichungen [12]:

$$x' = h_{11}x + h_{12}y + h_{13} - h_{31}xx' - h_{32}yx'$$

$$y' = h_{21}x + h_{22}y + h_{23} - h_{31}xy' - h_{32}yy'$$

Diese Gleichungen umgestellt:

$$\begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -xx' & -yx' \\ 0 & 0 & 0 & x & y & 1 & -xy' & -yy' \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

Die Matrix A ist eine 8x8-Matrix, die die Gewichtungen der Variablen h0, h1, ... , h7 enthält. Der Vektor b enthält die korrespondierenden x′- und y′-Koordinaten der korrigierten Punkte.

Der Code befüllt die Matrix A und den Vektor b in einer Schleife, die über die vier Punktpaare iteriert. Für jedes Paar werden die oben genannten Gleichungen in A und b eingetragen. Anschließend wird das Gleichungssystem A*h=b mithilfe der QR-Zerlegung aus der Eigen-Bibliothek gelöst, um den Vektor h zu bestimmen. Die Werte im Vektor h werden Schließlich verwendet, um die 3x3 Homographie Matrix H zu erstellen:

$$H = \begin{pmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & 1 \end{pmatrix}$$

Diese Matrix H kann dann verwendet werden, um beliebige Punkte von der ursprünglichen, verzerrten Ebene auf die korrigierte Ebene zu transformieren.

### 3.4.3 Matrix-Interpretation

Für das Sampling des transformierten Bildes wird die Schrittweite in X- sowie Y-Richtung benötigt. Aus dem transformierten Bild kann nun die Bounding Box des Finder Patterns an jeder Ecke bestimmt werden. Da dieses Muster immer aus 7 Modulen besteht, berechnet sich die Schrittweite wie folgt:

$$schrittweiteX = \frac{BoundingBoxBreite}{7}$$

$$schrittweiteY = \frac{BoundingBoxHoehe}{7}$$

Die Version des QR-Codes und somit auch die Anzahl der Module werden aus dem Durchschnitt der Modulanzahl in X- und Y-Richtung ermittelt:

$$modulAnzahl = \frac{\frac{QrCodeBreite}{schrittweiteX} + \frac{QrCodeHoehe}{schrittweiteY}}{2}$$

Das Ergebnis wird in Snippet 11 auf die nächstgelegene valide Modulanzahl gerundet. Beispielsweise wird eine berechnete Modulanzahl von 51 auf 53 (Version 9) korrigiert. Dies liegt daran, dass sich die Modulanzahl zwischen den QR-Code-Versionen immer um 4 erhöht (siehe Abschnitt 2.2.4).

```
int roundToNearesValidModuleCount(int moduleCount) {

    int closest = 21;
    int minDiff = abs(moduleCount - closest);

    for (int version = 2; version <= 40; ++version) {
        int validModuleCount = 4 * version + 17;
        int diff = abs(moduleCount - validModuleCount);

        if (diff < minDiff) {
            minDiff = diff;
            closest = validModuleCount;
        }
    }

    return closest;
}
```

*Snippet 11: Bestimmung einer validen Modulanzahl*

Das Bild wird mit der entsprechenden Schrittweite in einer doppelten For-Schleife gesampelt, wobei ein Offset zur Mitte des Moduls gesetzt wird. Für das Sampling wurde eine Selbstkorrektur implementiert. Falls die ursprünglich berechnete Modulanzahl zu gering ausfällt, kann sie nachträglich automatisch angepasst werden.

Zusätzlich überprüft der Algorithmus für jedes gesampelte Modul, ob es sich um eine isolierte Zelle handelt, also ein Modul, dessen unmittelbare Nachbarn alle einen anderen Wert haben. Falls dies der Fall ist, wird der Sampler in die Mitte dieses Moduls verschoben (siehe Snippet 12). Hierbei wird die Distanz zu den Nachbarzellen in alle Richtungen berechnet. Anschließend wird ein Offset bestimmt, mit dem sich der Sampler weiterbewegen soll. Dadurch wird verhindert, dass sich die Sample-Koordinaten durch Rundungsfehler langsam aus der tatsächlichen Reihe oder Spalte verschieben, was letztendlich dazu führen würde, dass Module übersprungen werden.

```
.
.
.
dy = iy;
dx = ix;
while (transformed[iy][ix] == transformed[dy][--dx] && dx > bottomLeft.X) {
    distanceLeft++;
}

float newx = (distanceRight - distanceLeft) / 2.0f;
float newy = (distanceTop - distanceBottom) / 2.0f;
return Vector(newx, newy, 0);
```

*Snippet 12: Bestimmung des Offsets für den Sampler*

Zusätzlich zur Korrektur isolierter Zellen wird diese Korrektur auch an den Übergängen zwischen schwarzen und weißen Bereichen angewendet – sowohl in horizontaler als auch in vertikaler Richtung. Abbildung 9 zeigt die Sampling-Punkte, wobei Rote Markierungen die Sampling-Positionen darstellen und Grüne Markierungen isolierte Zellen kennzeichnen. Dieser Korrekturfaktor wird besonders deutlich, da die Sampling-Punkte nicht in einem perfekten Raster angeordnet sind, wie es ohne Korrektur bei einer konstanten Schrittweite zu erwarten wäre.

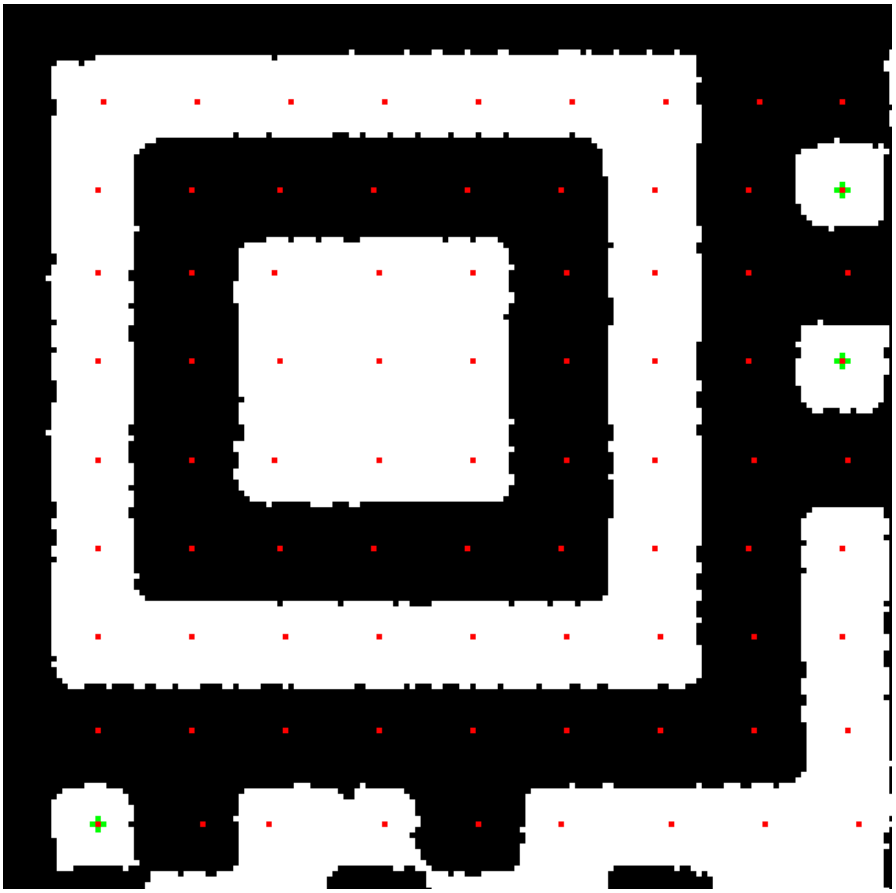

*Abbildung 9: Samplepunkte und deren Korrektur*

Die Sample-Werte werden in ein Bild geschrieben und isolieren somit den QR-Code, wie in Abbildung 10 zu sehen ist. Normalerweise würden diese Werte jedoch direkt die QR-Code-Matrix bilden, die anschließend zum Entziffern der tatsächlichen Werte verwendet wird.

*Abbildung 10: Isolierter QR-Code*

# 4 Zusammenfassung und Fazit

## 4.1 Zusammenfassung

Der entwickelte QR-Code-Scanner kann einige QR-Codes erfolgreich erkennen, allerdings ist die Erfolgsrate noch verbesserungswürdig. Insbesondere gelingt die perspektivische Korrektur gut bei leicht schrägen Winkeln, jedoch bereiten größere Neigungswinkel Probleme. Wenn die Finder Patterns stärker verzerrt und rechteckig erscheinen, werden sie nicht mehr zuverlässig erkannt. Zudem wird nicht für einen Fisheye Effekt korrigiert, wie er auf Runden Oberflächen auftreten könnte.

Ein weiteres Problem zeigt sich bei der Hough-Transformation, die in dieser Iteration empfindlich auf die Wahl des Schwellenwerts für die Linienerkennung reagiert. Die Bestimmung dieses Werts müsste in zukünftigen Iterationen robuster gestaltet werden.

*Abbildung 11: Schwammige Binärisierung*

Die Binärisierung funktioniert gut bei hochaufgelösten Bildern, in denen der QR-Code einen großen Teil des Bildes einnimmt. Bei unscharfen Aufnahmen hingegen kann es passieren, dass helle Bereiche zu dick erscheinen (Abbildung 11). Dies führt insbesondere beim Sampling zu Schwierigkeiten, da Module durch die Verzerrung falsch interpretiert werden können

## 4.2 Mögliche Optimierungen und Erweiterungen

Ein ineffizienter Aspekt des aktuellen Erkennungsprozesses ist die doppelte Erkennung der Finder Pattern. Nach der ersten Rotation wird der Erkennungsprozess erneut ausgeführt, um die Bounding Boxes der rotierten Finder Patterns zu bestimmen. Dieser Schritt könnte in zukünftigen Versionen optimiert werden, indem die bereits erkannten Punkte direkt mit der Rotationsmatrix transformiert werden, anstatt eine erneute Erkennung durchzuführen.

Ein weiterer wichtiger Aspekt ist die Performance-Optimierung. Durch Parallelisierung der Prozesse kann die Verarbeitungsgeschwindigkeit erheblich gesteigert werden, insbesondere bei der Hough-Transformation. Zusätzlich kann ein gezieltes Downsampling des Bildes helfen, Rechenzeit zu sparen, ohne dabei die Erkennungsgenauigkeit wesentlich zu beeinträchtigen.

## 4.3 Fazit

Die Implementierung eines QR-Code-Scanners ist eine hervorragende Übung, um sich mit den grundlegenden Konzepten der Bildverarbeitung vertraut zu machen. Es gibt zahlreiche Ansätze, dieses Problem zu lösen, und selbst in der aktuellen Forschung werden immer wieder neue Methoden entwickelt, um die Erkennung schneller und effektiver zu gestalten. Obwohl die aktuelle Implementierung noch Optimierungspotenzial bietet, stellt sie eine solide Grundlage für weitere Verbesserungen dar. Insgesamt zeigt dieses Projekt, dass selbst eine scheinbar einfache Aufgabe wie das Scannen eines QR-Codes eine Vielzahl an Herausforderungen und Lösungswegen mit sich bringt – und eine wertvolle Möglichkeit darstellt, sich intensiv mit den Kernprozessen der Bildverarbeitung auseinanderzusetzen.

## 4.4 Verwendung von Künstlicher Intelligenz

Der Einsatz von Künstlicher Intelligenz ist in den letzten Jahren in vielen Bereichen stark gestiegen. Unter anderem in der Softwareentwicklung. Werkzeuge wie ChatGPT oder Gemini sind heutzutage nicht mehr aus dem Arbeitsumfeld wegzudenken. Unter korrektem Einsatz können sie Entwickler bei ihrer Arbeit unterstützen und den Entwicklungsprozess beschleunigen und unterstützen. So ach in der Ausarbeitung dieses Projekts wo Tools wie ChatGPT, Google Gemini und GitHub Copilot zum Einsatz kamen.

GitHub Copilot bietet in der IDE live Code Vorschläge. Es wurde als ein autocomplete Feature verwendet, um Methoden oder Variablennamen nicht jedes Mal austippen zu müssen. Es ergänzt die Parameter und einige Methoden Felder automatisch, um die

Tipparbeit zu minimieren. Es wurde auch verwendet bei der Generierung der Kommentare, um Methoden und Parameter automatisch zu beschreiben.

ChatGPT und Gemini wurden genutzt bei der Erarbeitung der Komplexen Algorithmen und Berechnungen. Code, welcher von den KI-Quellen übernommen wurde, wurde an den entsprechenden Stellen gekennzeichnet.

Bei der Erstellung dieses Berichtes wurde KI verwendet, um Sprachliche und grammatikalische Fehler zu korrigieren. Alle Gedanken und Aussagen in den Textabschnitten wurden von mir formuliert, basierend auf dem Wissen aus den Vorlesungen und Quellen. Sie wurden lediglich auf sprachliche und grammatikalische Korrektheit überprüft.

# 5 Quellen

Alle Webseiten zuletzt aufgerufen am 4.3.2025

[1] „https://de.wikipedia.org/wiki/QR-Code,“ [Online].

[2] „https://www.microsoft.com/en-us/microsoft-365-life-hacks/privacy-and-safety/brief-history-qr-codes,“ [Online].

[3] „https://www.qrcode.com/en/about/standards.html,“ [Online].

[4] C. Aktaş, „The Evolution and Emergence of QR Codes,“ *Cambridge Scholars Publishing,* 2017.

[5] „https://www.etsy.com/de/listing/1177469762/wifi-qr-code-scan-to-connect-self,“ [Online].

[6] „https://www.paypal.com/ca/digital-wallet/ways-to-pay/pay-with-qr-code,“ [Online].

[7] „https://www.packagingdigest.com/packaging-design/7-reasons-and-5-ways-to-use-qr-codes-on-your-packs,“ [Online].

[8] „https://de.qrcodechimp.com/qr-code-use-cases/,“ [Online].

[9] „https://www.qrcode-generator.de/qr-code-marketing/qr-codes-basics/,“ [Online].

[10] „https://www.csfieldguide.org.nz/en/chapters/coding-error-control/qr-codes/,“ [Online].

[11] „https://support.ptc.com/help/mathcad/r10.0/en/index.html#page/PTC_Mathcad_Help/example_grayscale_and_color_in_images.html,“ [Online].

[12] *ChatGPT prompt: (in Referenz zu Snippet 10) wie sieht die mathematshe gleichung hierzu aus.*

[1 „https://www.paypal.com/ca/digital-wallet/ways-to-pay/pay-with-qr-code,“ [Online].
3]

## Eidesstattliche Erklärung

Hiermit erkläre ich/ Hiermit erklären wir an Eides statt, dass ich/ wir die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt habe/ haben. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche einzeln kenntlich gemacht. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Osnabrück, den 4.3.25 .....
Ort, Datum Arpad Horvath

## Urheberrechtliche Einwilligungserklärung

Hiermit erkläre ich/ Hiermit erklären wir, dass ich/wir damit einverstanden bin/sind, dass meine/ unsere Arbeit zum Zwecke des Plagiatsschutzes bei der Fa. Ephorus BV bis zu 5 Jahren in einer Datenbank für die Hochschule Osnabrück archiviert werden kann. Diese Einwilligung kann jederzeit widerrufen werden.

..... .....
Ort, Datum Unterschrift

Hinweis: Die urheberrechtliche Einwilligungserklärung ist freiwillig.