

Aufgabe 3: Kreis-Code

Diese Aufgabe ermöglicht unzählige kreative Herangehensweisen. Zum Beispiel eignen sich die abrupten Übergänge von schwarz auf weiß gut für eine Kantenerkennung. Die großen schwarzen Flächen (und der innere weiße Ring) dagegen legen nahe, zunächst einzelne Objekte im Bild zu identifizieren. Es würde den Umfang dieses Dokumentes sprengen, mehrere solcher Ansätze im Detail anzusprechen. Deshalb werden wir uns hier nur auf eine einzelne exemplarische Lösung beschränken. Teilaufgabe 3 entfällt als eigenständiger Abschnitt, da gleich der allgemeine Fall mit Störeffekten behandelt wird.

3.1 Teilaufgabe 1: Mittelpunkte bestimmen

Lösungsidee

Es wird millionenfach nach dem Muster Dunkel-Hell-Dunkel-Hell gesucht, in allen möglichen Größenordnungen, Rotationen und Positionen. Die Punkte, an denen das Muster besonders oft gefunden wurde, werden zu Mittelpunkten deklariert.

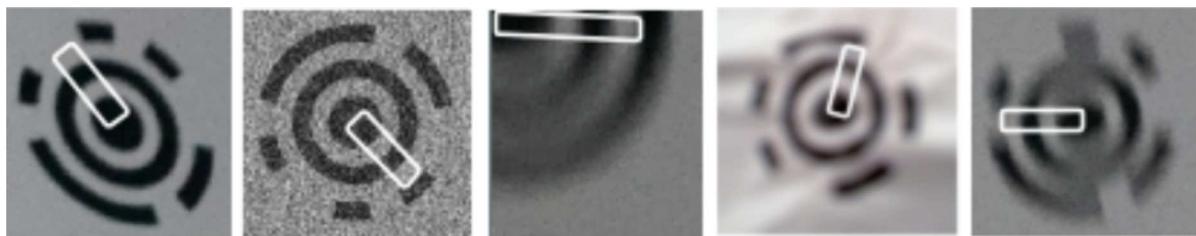


Abbildung 9: Das Suchmuster in Kreiscodes, die verschiedenen Störeffekten ausgesetzt sind.

Umsetzung

Eine Schablone zum Suchen Um das Nachdenken über den Suchvorgang zu vereinfachen, führen wir das Konzept der Schablone ein. Das ist ein Objekt, das man – beliebig rotiert und skaliert – auf einen Ausschnitt des Bildes legen kann, um dann durch einen Test zu erfahren, ob die überdeckten Pixel (hinreichend gut) dem gesuchten Muster entsprechen.



Abbildung 10: Das Muster nach dem gesucht wird, in einer Schablone.

Der Schablonentest Da sehr viele Bildausschnitte auf ihre Ähnlichkeit mit dem Muster hin getestet werden müssen, ist es nützlich, den Test minimalistisch zu halten; zum Beispiel, indem im Schablonenbereich nur 4 Punkte ausgemessen werden. Das ist das Minimum an Punkten, um das gesuchte Muster gerade noch nachzuahmen. Alternativ könnte man auch

etwas mehr Messpunkte benutzt und anders platzieren, um sich stärker auf die Kanten statt auf die Flächen zu konzentrieren.



Abbildung 11: Vier Messpunkte nähern das gesuchte Muster an.

Den Grad der Ähnlichkeit der gemessenen Punkte mit dem Muster könnte durch eine Prozentsangabe beziffert werden. Es reicht aber auch, nur eine Ja/Nein-Antwort zurückzugeben. Ein möglicher Test wäre dann: Ist der Kontrast zwischen den „dunklen“ und den „hellen“ Messpunkten groß genug? Für diesen Test hat sich ein Mindestkontrast von 30 als gute Schwelle erwiesen. Das Helligkeitsspektrum erstreckt sich dabei von 0 (schwarz) bis 255 (weiß).

Einzelne Punkte prüfen Hat man den Schablonentest definiert, bietet es sich an, jeweils für einen Punkt durch Rotation der Schablone überprüfen zu lassen, mit welcher Wahrscheinlichkeit er ein Mittelpunkt ist. Einen Punkt, den wir überprüfen, nennen wir ab nun Ankerpunkt (weil die Schablone an ihm „verankert“ ist).

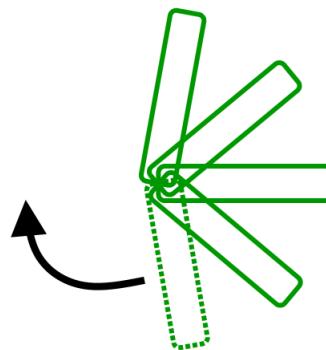


Abbildung 12: Die Trefferquote eines Ankerpunktes wird ermittelt.

Hat man das Testergebnis als Ja/Nein-Rückgabe definiert, ist folgendes Maß naheliegend:

$$\text{Trefferquote} = \frac{\text{Anzahl erfolgreiche Tests}}{\text{Anzahl durchgeführte Tests}}$$

Je höher die Trefferquote, desto wahrscheinlicher handelt es sich beim untersuchten Ankerpunkt um einen Mittelpunkt. Die Trefferquoten werden später die Daten sein, anhand derer Mittelpunkte bestimmt werden.

Länge der Schablone ändern Bei dem Schablonen-Ansatz muss die Länge der Schablone verändert werden, um verschiedene Kreiscodegrößen zu erkennen. Damit alle Größenordnungen identisch behandelt werden (Skaleninvarianz) ist es empfehlenswert, die Längenänderung exponentiell zu gestalten, sodass z.B. jede neue Schablone 15% länger ist als die

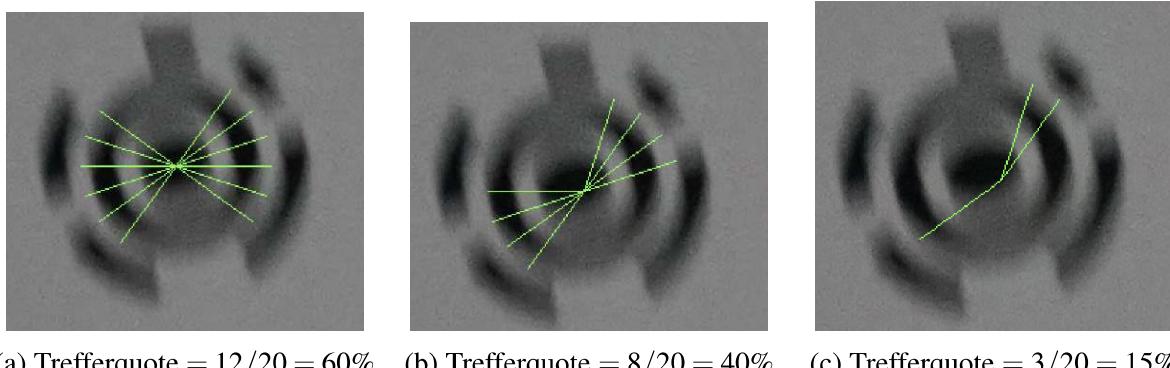


Abbildung 13: Es wurden jeweils 20 Tests durchgeführt. Die grünen Linien markieren jeweils die erfolgreichen Tests.

voherige. Folgendes Vorgehen ist möglich: Es wird eine kleinste Länge (min) und ein Skalierungsfaktor (k) festgelegt. Mit diesen beiden Werten können dann mit steigendem Exponenten n beliebig viele, immer größer werdende Schablonenlängen $L(n)$ erzeugt werden:

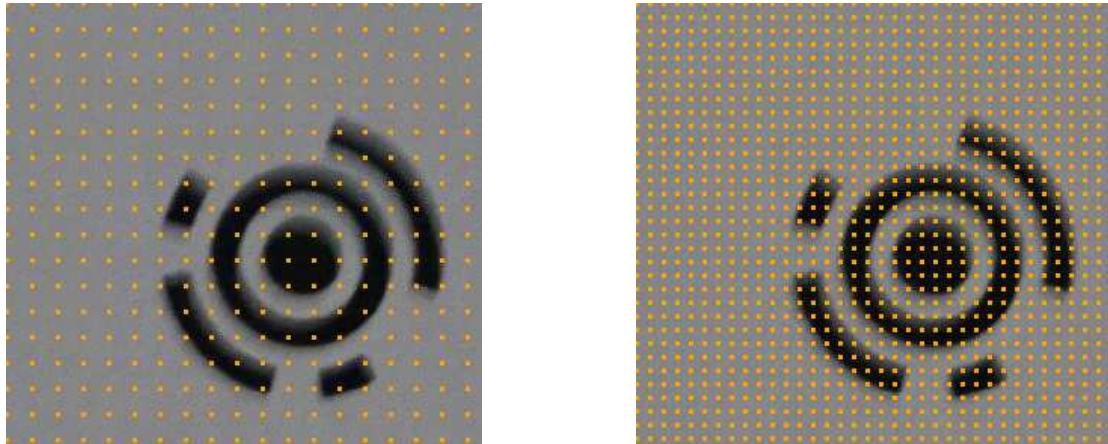
$$L(n) = min \cdot k^n$$

Als kleinste Länge ist 12 Pixel ausreichend klein, als Skalierungsfaktor ist 1,15 ein guter Kompromiss. Als größte Länge kann z.B. der kleinere Wert aus Bildbreite und Bildhöhe gewählt werden.

Ankerpunkte auswählen Es ist zu rechenaufwändig, jedes einzelne Pixel als Ankerpunkt zu betrachten. Eine erste filternde Auswahl könnte deshalb zum Beispiel durch ein Suchraster getroffen werden. Dadurch werden je nach Schablonenlänge 30% bis 99% aller Pixel übersprungen. Damit die Suchgenauigkeit nicht leidet, muss jedoch der Abstand der Ankerpunkte untereinander weiterhin klein genug sein, dass keine Kreiscodes übersprungen werden. Ein guter Kompromiss für den Abstand ist beispielsweise ein Zehntel der aktuellen Schablonenlänge (vgl. Abbildung 14).

Nach Trefferquoten filtern Nachdem die Suche beendet ist, existiert ein großer Datenhaufen, in unserem Fall Trefferquoten. Nicht alle dieser Daten sind nützlich, manche stören sogar, weil sie auf Mittelpunkte an den falschen Orten hinweisen. Diese störenden Ankerpunkte können z.B. durch einen Filter entfernt werden, der alle Ankerpunkte aussortiert, deren Trefferquote zu niedrig ist. Zu streng darf dieser Filter jedoch auch nicht sein, sonst werden Kreiscodes übersehen (vgl. Abbildung 15).

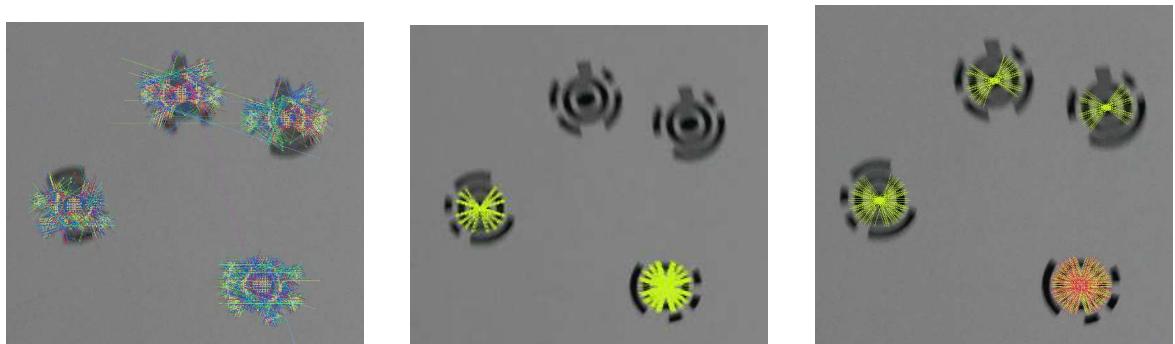
Mittelpunkte auswählen Es könnten einzelne Ankerpunkte als Mittelpunkte deklariert werden, die lokal die höchste Trefferquote haben. Meistens ist es aber robuster, die Informationen mehrerer Ankerpunkte zu akkumulieren. Eine mögliche Vorgehensweise: Es werden gleichfarbige (d.h.: mit der gleichen Schablonenlänge getestete) Ankerpunkte zu Clustern zusammengefasst. Bei diesen Clustern lässt sich dann ein Schwerpunkt berechnen, wobei Ankerpunkte mit höherer Trefferquote stärker gewichtet werden könnten. Falls mehrere Cluster eng



(a) 1/5 Schablonenlänge: Zu großer Abstand zwischen den Ankerpunkten. Der Mittelpunkt wird stark verfehlt.

(b) 1/10 Schablonenlänge: Ausreichend kleiner Abstand zwischen den Ankerpunkten. Der Mittelpunkt wird mindestens einmal sehr gut getroffen.

Abbildung 14: Verschiedene Abstände zwischen den Ankerpunkten



(a) Filter: Trefferquote $\geq 10\%$. Das ist nicht streng genug, es gibt zu viele störende, aus Versehen entstandene Treffer.

(b) Filter: Trefferquote $\geq 70\%$. Das ist zu streng, die beiden unscharfen Kreiscodes werden nicht mehr erkannt.

(c) Filter: Trefferquote $\geq 35\%$. Ein guter Kompromiss

Abbildung 15: Filterung nach Trefferquoten

beieinander liegen, kann z.B. der stärkste unter ihnen ausgewählt und dessen Schwerpunkt als Mittelpunkt deklariert werden. Ein mögliches Maß für die Stärke einer Clusters ist die Summe der Trefferquoten der zugehörigen Ankerpunkte. Wenn bis zu dieser Stelle alles wie geplant ablief, existiert nun für jeden Kreiscode genau ein Cluster.

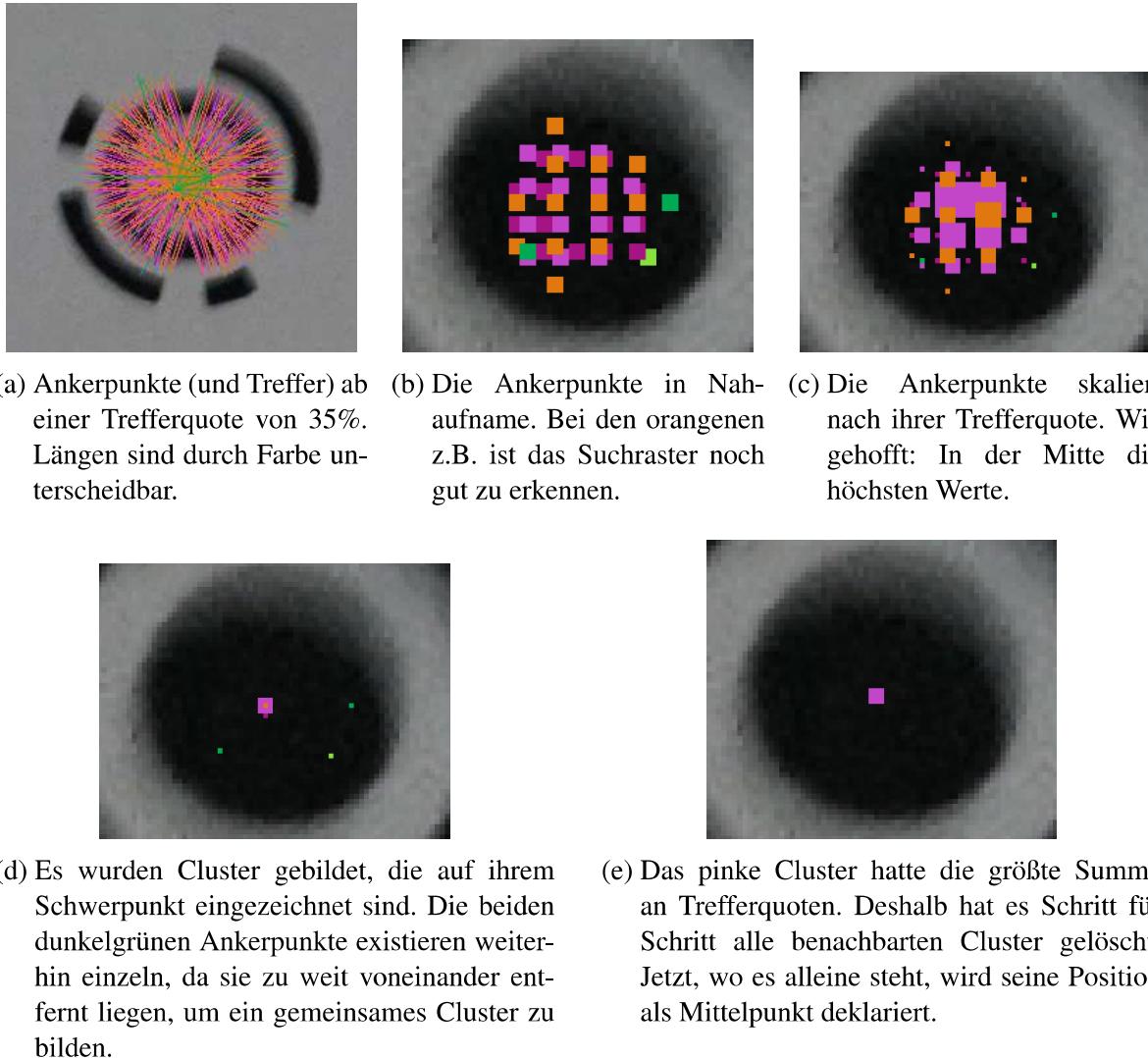


Abbildung 16: Auswahl des Mittelpunkts durch Clustering

3.2 Teilaufgabe 2: Bedeutungen decodieren

Lösungsidee

Mit Wissen der Cluster-Schablonenlänge (Radius) und durch zusätzliche Ellipsennäherungen wird die Form des äußersten Rings ermittelt. Der ausgelesene äußerste Ring wird dann in 16 Blöcke aufgeteilt, die abschließend in eine dunkle und eine helle Gruppe aufgeteilt werden.

Umsetzung

Barcode erzeugen Um die Segmente unabhängig vom eigentlichen Kreiscode zu analysieren, ist es praktisch, den äußersten Ring in eine Art Barcode umzuwandeln. Solch ein Barcode lässt sich einfacher weiterverarbeiten und abstrahiert von irrelevanten Informationen wie Größe oder Position des Kreiscodes.

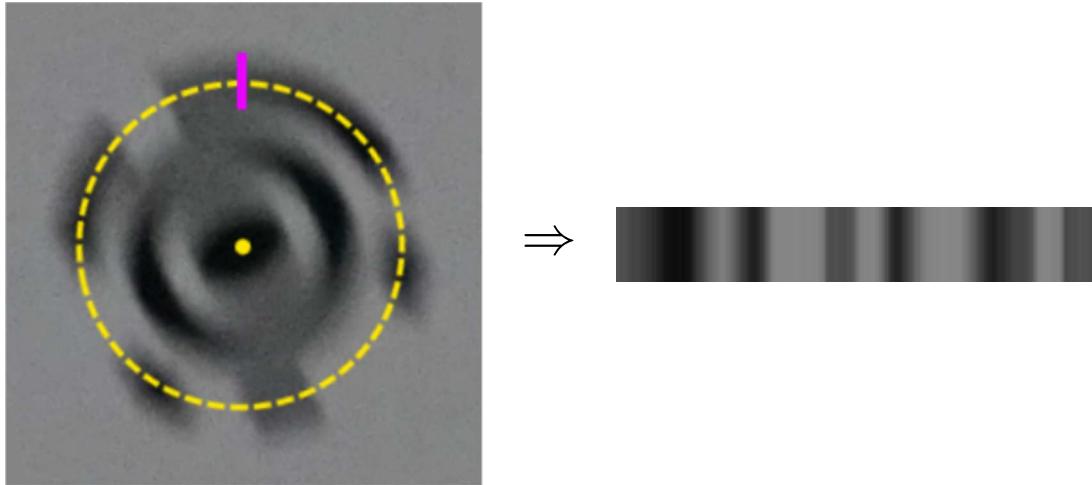


Abbildung 17: Von Kreiscode zu Barcode. Dazu werden, bei 12 Uhr beginnend, einzelne Pixel entlang des äußersten Rings reihum ausgelesen. Zur besseren Ansicht wurde der Barcode in der Höhe gestreckt.

Im einfachsten Fall ist der Kreiscode ein Kreis geblieben. Dann lässt sich der Radius des äußersten Rings sofort berechnen aus der Schablonenlänge des zugehörigen Clusters. In konstanten Winkelabständen können dann Helligkeitswerte entlang des Rings gemessen werden (vgl. Abbildung 17).

Verzerrungen beachten Im Allgemeinen darf nicht davon ausgegangen werden, dass die Kreiscodes im Bild perfekt kreisförmig erscheinen. Stattdessen sind sie oft etwas verzerrt, z.B. weil sie angewinkelt fotografiert wurden oder weil das Papier an ihrer Stelle zerknittert ist. Zumindest die Verzerrung durch Anwinkeln sollte behandelt werden, da sie in den Beispieldaten häufiger auftaucht. Eine gute Näherung ist, die verzerrten Kreiscodes als Ellipsen aufzufassen. Kennt man dann je Kreiscode dessen genauen Ellipsen-Parameter, dann lassen sich die Verzerrungseffekte aufheben.

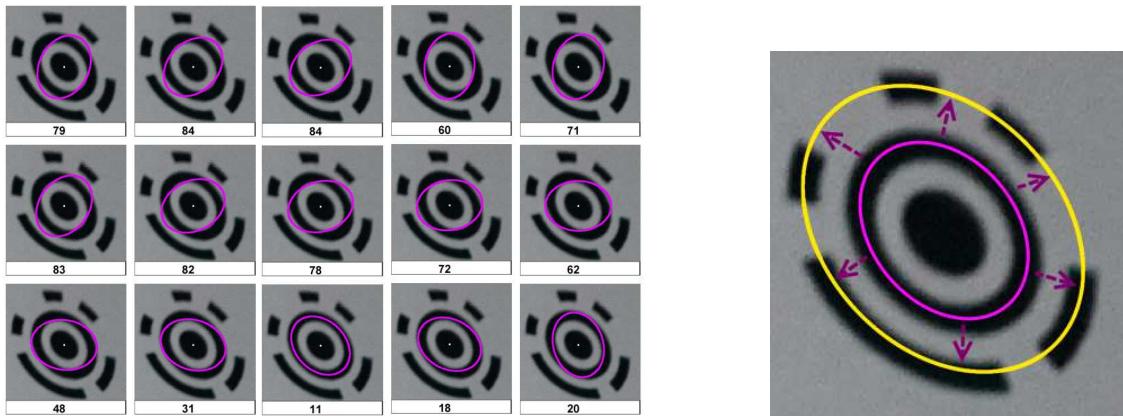
Als Referenz am Kreiscode kann z.B. der mittlere schwarze Ring verwendet werden, der möglichst exakt mit einer Ellipse nachgeahmt werden soll. Es können unterschiedlich geformte und gedrehte Ellipsen ausprobiert und die am genauesten passende ausgewählt werden. Als Maß der Annäherung eignet sich z.B. die durchschnittliche Helligkeit der bedeckten Pixel. Je kleiner dieser Wert, desto dunkler sind die Pixel, auf denen die aktuelle Ellipse liegt. Und je dunkler die Pixel, desto wahrscheinlicher befindet sich die Ellipse im Zentrum des mittleren schwarzen Rings.

Anhand der Schablonenlänge des Clusters kann der Radius r des mittleren schwarzen Rings bestimmt werden. Mögliche Parameterschranken der zu testenden Ellipsen:

Kleine Halbachse b : $0,8r - 1,2r$

Große Halbachse: $1b - 2b$

Ist die am besten passende Ellipse gefunden, kann diese hochskaliert werden. Dadurch erhält man den Verlauf des äußersten Rings.



(a) 15 untersuchte eigene Ellipsen (von insgesamt 648). Die Zahl gibt jeweils die durchschnittliche Helligkeit der bedeckten Pixel an. Unten in der Mitte gibt es eine sehr gute Näherung.

(b) Die gefundene Ellipse wird mit dem Wert $5/3$ vergrößert. Die neue Ellipse (gelb) beschreibt dann den Verlauf des äußersten Rings.

Abbildung 18: Bestimmung der am besten passenden Ellipse

Verzerrte Segmentgrößen beachten Wurde eine Ellipse mit Verzerrung entdeckt, folgt: Die einzelnen Segmente sind auch verzerrt und nehmen unterschiedliche Winkelbreiten ein (statt ursprünglich jeweils $360^\circ / 16 = 22,5^\circ$). Beim Verfahren der Barcodeerzeugung wäre es nun ungeschickt, weiterhin konstante Winkelabstände zum Auslesen zu benutzen. Stattdessen bietet es sich an, bei weit entfernten Segmenten in engeren Abständen zu messen und bei nahen Segmenten in weiteren Abständen. Dadurch wird der Verzerrungseffekt aufgehoben (s. Abbildung 19).

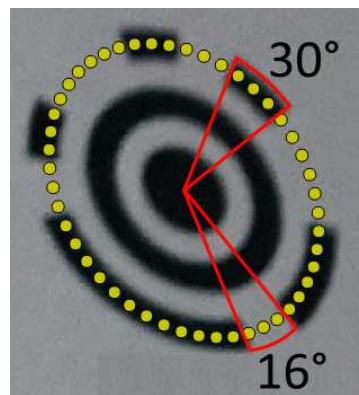


Abbildung 19: Im Umfeld des 16° -Segments wird enger ausgelesen und im Umfeld des 30° -Segments breiter, um die Verzerrung aufzuheben. Hier insgesamt 48 Messwerte, durchschnittlich 3 pro Segment.

Barcode in Blöcke einteilen Ein Barcode enthält keine explizite Information darüber, wo die einzelnen Segmente anfangen bzw. aufhören. Er ist stattdessen nur eine lückenlose Aneinanderreihung von Messwerten (s. Abbildung 20). Bei anderen Verfahren wird man in

ähnlicher Weise auf das Problem stoßen, zwischen verwaschenen Übergängen einzelne Segmente auszumachen.



Abbildung 20: Ein Barcode, bestehend aus 128 Messwerten (im Durchschnitt 8 pro Segment); gelb eingefärbt, um die Ansicht in den nächsten Abbildungen zu erleichtern.

Hier ist es naheliegend, den Barcode in 16 Blöcke zu einzuteilen (die dann jeweils für ein Segment stehen). Bei einer guten Einteilung sollte jeder Block eine möglichst homogene Helligkeit besitzen, also entweder hauptsächlich dunkel oder hauptsächlich hell sein. Ein mögliches Maß: Pro Block wird die Varianz der Helligkeiten berechnet. Je kleiner die Varianz, desto homogener der Block. Aus diesen 16 Blockvarianzen wird dann der Durchschnitt gebildet. Je kleiner dieser Durchschnitt, desto besser die Einteilung (vgl. Abbildung 21). Hier wird vorausgesetzt, dass im Barcode alle Segmente die selbe Breite haben, also aus der selben Anzahl an Messwerten bestehen. Das ist der Fall, wenn der Barcode vorher entzerrt wurde (vgl. Abbildung 19).

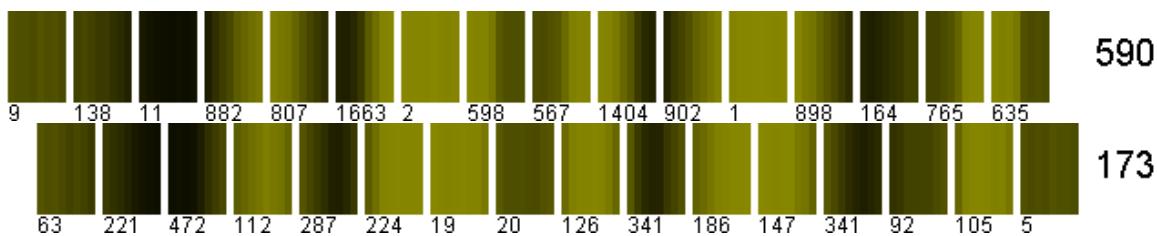


Abbildung 21: Zwei Einteilungen (von insgesamt 8 möglichen). Die untere teilt den Barcode besser auf als die obere, weil ihre Blöcke homogener sind. In diesem Fall ist die untere Einteilung sogar die beste und die obere die schlechteste

Bits zuweisen Abschließend kann jedem Block dessen „Bitwert“ zugewiesen werden: 0 oder 1. Ein Ansatz ist, die Blöcke untereinander in eine helle Gruppe und eine dunkle Gruppe aufzuteilen. Schließlich sollte erwartet werden können, dass jeder Block entweder ziemlich hell oder ziemlich dunkel ist. Ein mögliches Verfahren: Bei jedem Block wird zunächst der Durchschnitt seiner Messwerte gebildet. Nach diesem Durchschnitt werden die Blöcke sortiert und dann die größte paarweise Veränderung ermittelt, quasi der größte Helligkeitssprung. Bei diesem Sprung werden die beiden Gruppen aufgeteilt; vgl. Abbildung 22.

Anmerkung: Die beiden Kreiscodes „00“ (komplett weiß) und „Falsch“ (komplett schwarz) sind mit diesem Verfahren nicht kompatibel und müssen vorher abgefangen werden. Dies geht z.B., indem geprüft wird, ob der hellste Block und der dunkelste Block hinreichend nahe beieinander liegen. Als gut erwiesen hat sich hierfür eine Schwelle von 70.

In der Tabelle nachschlagen Ist die Bitfolge bestimmt, ist das Nachschlagen in der Tabelle kein großer Aufwand mehr. Es gibt 134 Einträge, jeder besteht aus 16 Bits, die auf 16

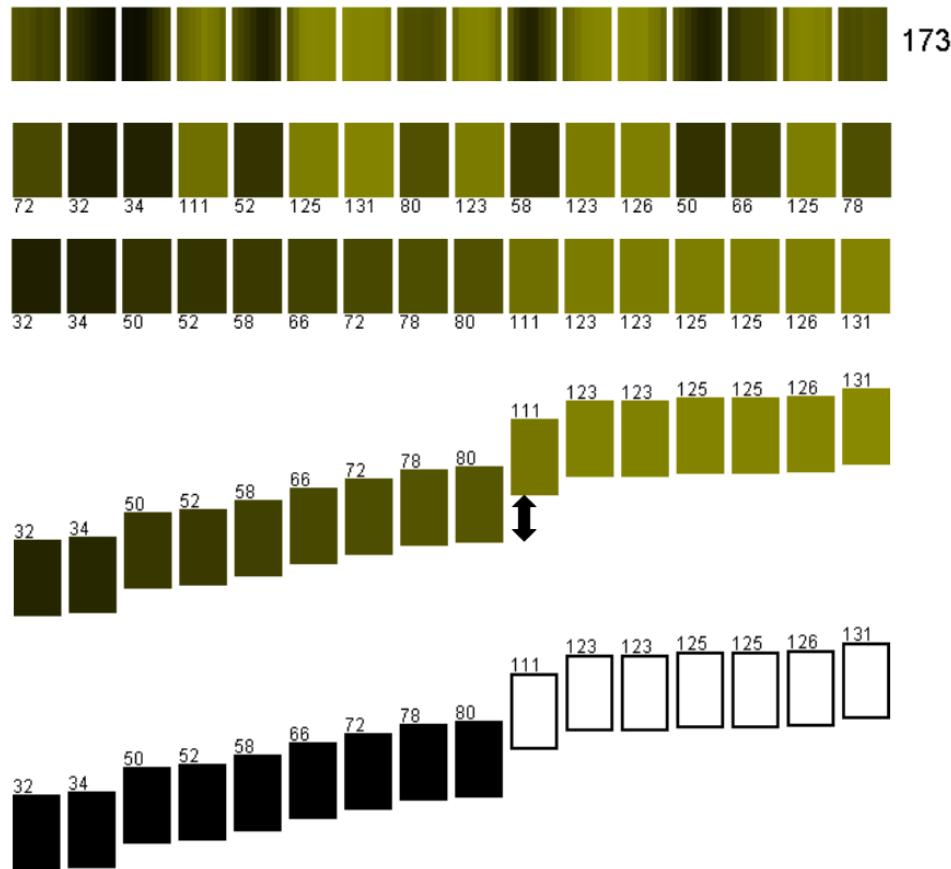


Abbildung 22: Vom Barcode zu einzelnen Bits. Die Werte pro Block stehen für deren Durchschnittshelligkeit. Der Helligkeitssprung von 80 auf 111 ist gut erkennbar, aber nicht so stark ausgeprägt wie in der Theorie erhofft.

verschiedene Arten rotiert sein können. Macht maximal $134 \cdot 16 \cdot 16 = 34,304$ nötige Vergleiche (plus ein bisschen Overhead). Das schafft ein Computer heutzutage im Bruchteil einer Sekunde.

Dieser Ansatz skaliert jedoch sehr schlecht. Wenn man den Anspruch hat, auch auf größere Tabellensysteme vorbereitet zu sein (z.B. 64 Segmente, 50.000 Einträge, ...), dann kann über Beschleunigungen nachgedacht werden. Eine erste wäre beispielsweise, die bitweisen Vergleiche nur auf diejenigen Einträge zu beschränken, welche dieselbe Anzahl an Einsen haben wie die gefundene Folge. Das allein reduziert den Rechenaufwand um 80-99%, verglichen mit dem naiven Ansatz.

Abgeschnittene Kreiscodes am Rand Auch bei einem unvollständigen Kreiscode können bestimmte Bedeutungen schon ausgeschlossen werden. Deshalb bietet es sich an, in diesen Fällen wenigstens eine reduzierte Liste aller noch möglichen Bedeutungen anzugeben. Das sind dann zwar oft noch sehr viele, aber immer noch besser als gar keine Auswahl.

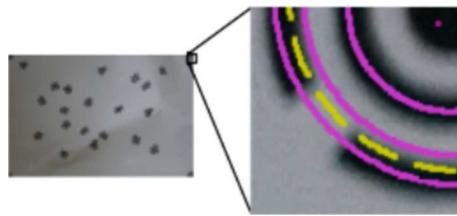


Abbildung 23: Gefundene Bitfolge: 11-,-—,-—,-110. Daraus ergeben sich folgende mögliche Bedeutungen: 02 0e 17 1d 1e ') * . 5 => ? @ A B C E I M N Q R T U Ö ‘ c e g k l m n o r s u z ö ü ß ll ul lr Wahr

3.3 Analyse

Laufzeitanalyse – Berechnung

Üblicherweise gibt man die Laufzeitkomplexität eines Algorithmus in der Landau- bzw. „Groß O“-Notation an. Das geht vor allem dann gut, wenn möglichst wenig Spielraum herrscht beim Eingabe- und Ausgabeformat. Bei dieser Aufgabe aber ist die Eingabe höchst chaotisch. Es gibt keinen vernünftigen Weg, mit nur ein paar Variablen eine Konstellation von über einer Million Pixel zu beschreiben.

Eine andere Art von Analyse ist jedoch möglich: Vorherzusagen, in welcher Größenordnung sich die Rechenzeit (in Sekunden) befindet. Dazu müssen zwei Sachen bekannt sein: Die Anzahl der elementaren Rechenoperationen, die das Programm ausführen wird, und die Taktfrequenz des Prozessors, die angibt, wie viele Operationen dieser pro Sekunde verarbeiten kann. Als elementare Rechenoperationen zählen wir hier Addition, Array lesen, Array schreiben und ähnliche Aufgaben, die sehr viel weniger Aufwand bedeuten als z.B. Fließkommaarithmetik. Bei unserem (unoptimierten) Ansatz verursachen die Schablonentests den größten Rechenaufwand. An ihnen wollen wir deshalb exemplarisch die Größenordnung der benötigten Laufzeit analysieren.

Als erstes wird ermittelt, wie viele Tests es je einzelnen Raster gibt. Da Raster zweidimensional aufgespannt werden, fließt der Rasterabstand $d(n)$ quadriert in die Rechnung ein. Er ist bei uns definiert als Schablonenlänge $L(n)$ durch Rasterauflösung a .

$$\text{Tests im Raster} = \frac{bh}{d(n)^2} \cdot t = \frac{bh}{(L(n)/a)^2} \cdot t = bh \left(\frac{a}{\min \cdot k^n} \right)^2 \cdot t$$

Als zweites wird ermittelt, wie viele verschiedene Längen erzeugt werden; pro Länge gibt es schließlich ein neues Raster zu testen. Die Anzahl kann am größten benötigten Exponenten abgelesen werden, den wir N nennen.

$$\max = \min \cdot k^N \Leftrightarrow N = \frac{\ln(\max/\min)}{\ln(k)}$$

$$N = \frac{\ln(2561/12)}{\ln(1,15)} \approx 38,3742 \approx 38$$

Tabelle 1: Parameter, die man selbst festlegen kann bzw. muss, sind fett gedruckt. Die restlichen Parameter sind durch äußere Umstände vorbestimmt. Anmerkung zu p: Je Schablonenmesspunkt werden in unserer Implementierung 20-30 Operationen ausgeführt (also insges. ca. 100 Operationen pro Test). Für einen genauen Wert müsste man „Operation“ präziser definieren. Für eine solche Analyse reicht es aber, wenn die Größenordnung stimmt.

Parameter		Wert für Beispielrechnung
b	Bildbreite	3895 (Dimension Beispiefotos)
h	Bildhöhe	2561 (Dimension Beispiefotos)
min	Minimallänge	12
max	Maximallänge	2561 (Kleinerer Wert aus Breite, Höhe)
k	Skalierfaktor	1,15
t	Tests pro Ankerpunkt	20
a	Rasterauflösung	10 (Zehntel der Schablonenlänge)
p	Operationen pro Test	100 (geschätzt)
f	Taktfrequenz des Prozessors	2,4 GHz (in Systemsteuerung nachgeschaut)
n	aktueller Exponent	-

Durch Aufsummieren über alle Raster erhält man die Gesamtanzahl der durchzuführenden Tests:

$$\begin{aligned} \text{Anzahl Tests} &= \sum_{n=0}^N \text{Tests im Raster}(n) \\ &= \sum_{n=0}^{38} 3895 \cdot 2561 \cdot \left(\frac{10}{12 \cdot 1,15^n} \right)^2 \cdot 20 \approx 5,68 \cdot 10^8 \end{aligned}$$

Damit ergibt sich die Laufzeit folgendermaßen:

$$\text{Laufzeit} = \frac{p \cdot \text{Anzahl Tests}}{f} = \frac{100 \cdot 5,68 \cdot 10^8}{2,4 \cdot 10^9} = 23,6$$

Die Analyse sagt also eine Laufzeit von 23,6 Sekunden voraus. Das erlaubt folgende Aussage: Sogar wenn man sich beim Parameter p (Operationen pro Test) um eine ganze Größenordnung verschätzt haben sollte, terminiert das Programm in menschlicher Zeit (ca. 4 Minuten). Die Formel für Anzahl Tests eignet sich nun auch, um die Auswirkungen der einzelnen Parameter auf die Laufzeit zu betrachten. Die Dimension des Bildes z.B. wirkt sich fast nur linear auf die Laufzeit aus. N steigt nur logarithmisch mit der Dimension des Bildes, und wenn trotzdem neue Raster entstehen, befinden sich in ihnen exponentiell weniger Ankerpunkte als in den kleineren Rastern. Somit ist die Laufzeitkomplexität der Mittelpunktsfindung mit $\mathbf{O}(n)$ zu charakterisieren, wobei $n = \text{Anzahl Pixel}$.

Falls mehrere echte Laufzeiten gemessen und kommentiert wurden, ist eine solche Analyse bei dieser Aufgabe nicht zwingend nötig. Aber sie ist eine schöne Übung und erlaubt es, die Ausführbarkeit des Algorithmus zu bestätigen, unabhängig von der Ausführumgebung. Und falls sich das Programm z.B. von den Bewertern nicht starten lässt, hilft sie, die Bewerter trotzdem von der behaupteten Laufzeit zu überzeugen.

Laufzeitanalyse – Messung

bitmap.gif	noise50.jpg	camB.jpg
19,8	20,2	19,5
20,3	19,5	19,9

Die oben angegebenen Messwerte (in Sekunden) bestätigen die vorhergesagte Laufzeit. Dass in der Analyse die einzige Information über das Bild dessen Pixelanzahl n war, wird durch das Messergebnis auch widergespiegelt: Alle drei (gleich großen) Bilder brauchen gleich lange, obwohl ihr „Aussehen“ sich stark unterscheidet.

Optimierung

Wenn davon ausgegangen wird, dass sich die Kreiscodes auf einem hellen Hintergrund befinden, lässt sich die Laufzeit stark reduzieren, mit nur minimalem Qualitätsverlust. Denn es können nun alle Ankerpunkte übersprungen werden, die eine durchschnittliche oder überdurchschnittliche Helligkeit haben, da sie (in fast allen Fällen) sowieso nicht als Mittelpunkt in Frage kommen. Durch diese härtere Ankerpunkte-Auswahl verringert sich die Laufzeit der Schablonensuche bei den Fotografien von 20 Sekunden auf 2 Sekunden, bei den S/W-Bildern auf 0,6 Sekunden. Inklusive der restlichen Programmteile (Bild laden: 1,2 Sekunden, Gauß-Weichzeichnen: 0,6 Sekunden, Ellipsen finden: 0,4 Sekunden für 20 Kreiscodes.) kommt das Programm damit bei allen Beispiel-Fotografien auf eine Gesamlaufzeit von gut 4 Sekunden.

Fehlerresistenz

Es kann geschehen, dass sich die gefundene Bitfolge nicht in der Tabelle wiederfinden lässt. Ein gutes Programm kann hartnäckig bleiben und noch einmal prüfen, ob sich nicht durch eine kleine Korrektur eine korrekte Folge erzielen lässt. Eine sehr allgemeine Korrektur wäre z.B., jeweils ein Bit in der gefundenen Folge zu switchen und dann zu schauen, ob die neu generierte Folge sich in der Tabelle wiederfinden lässt.

Graustufen und Bildglättung

Es reicht aus, nur Helligkeitskontraste zu beachten (statt Farbkontraste). Das lässt sich z.B. dadurch rechtfertigen, dass man auch im echten Leben fast nur dunklen Barcodes und QR-Codes begegnet, die sich vor einem hellen Hintergrund befinden.

Um beim Rauschen Ausreißer zu erkennen bzw. zu eliminieren bietet es sich an, eine Form von Bildglättung anzuwenden. Wird dies live beim Auslesen bewerkstelligt, könnte man den Betrachtungsradius um den Pixel herum entsprechend der Kreiscodegröße skalieren. Wird stattdessen das gesamte Bild schon zu Beginn geglättet, sollte darauf geachtet werden, dass die Glättung nicht so stark ist, dass sie kleine Kreiscodes zerstört. Ist sie dann zu schwach für

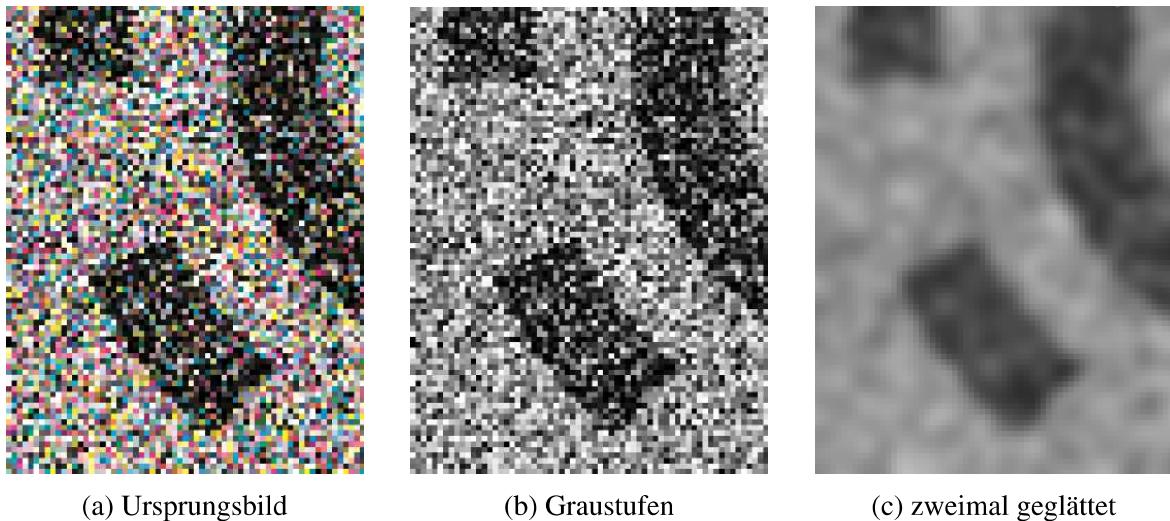


Abbildung 24: Umwandlung in Graustufen und Bildglättung

große Kreiscodes, kann bei Bedarf bei entsprechender Suchgröße ein zweites oder drittes Mal geglättet werden.

Als Glättverfahren ist ein 5×5 Gaussfilter möglich. Dieser kann z.B. einmal zu Beginn angewendet werden und ein zweites Mal bei einer Schablonenlänge von ca. 20 Pixeln. Hier ein 5×5 Filterkern mit (Integer-gerundeter) Gaussverteilung von Sigma=1,4:

$$\frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Fremde Algorithmen und Bibliotheken

Eigene Implementierungen fremder (zitierter) Ideen sind erlaubt. Hier besteht oft schon eine Leistung darin, die Implementierung korrekt hinzubekommen. Wird stattdessen gleich ein ganzer lauffähiger Code kopiert (bzw. aus einer Bibliothek benutzt), sollte trotzdem für die Bewerter klar erkennbar sein, dass die Funktionsweisen des Codes und die Idee dahinter ausreichend verstanden wurden. Das ließe sich z.B. dadurch erreichen, dass man auf die jeweiligen Parameter des kopierten Codes eingeht und erklärt, ob und warum man diese verändert oder warum man sie auf den Standardwerten gelassen hat.

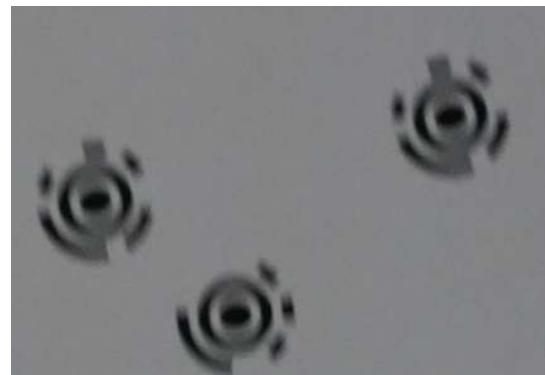
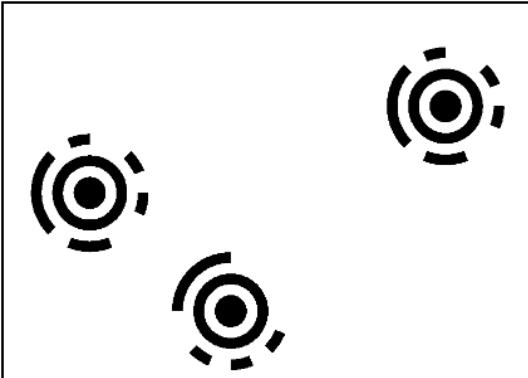
Diese Lösung benutzt z.B. eine fremde Implementierung zum Erzeugen der Ellipsen⁸. Bei dieser ist es besonders praktisch, dass durch die dort gewählte Parameterdarstellung (\sin , \cos) automatisch der von uns gewünschte Effekt eintritt, dass an weit entfernten Bereichen enger ausgelesen werden soll.

⁸http://www.uni-forst.gwdg.de/~wkurth/cb/html/cg_v05a.htm

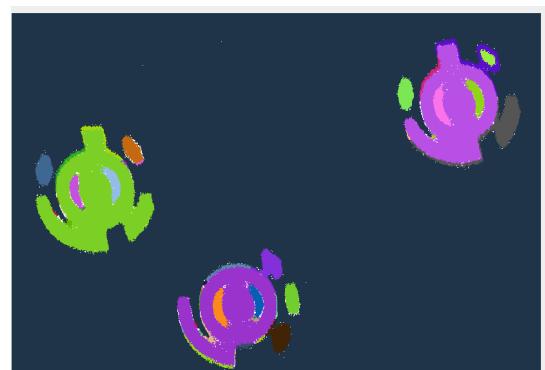
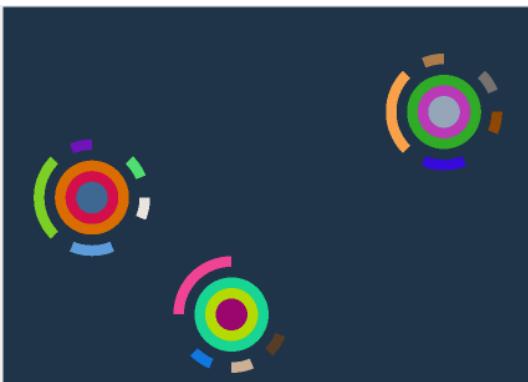
Mögliche Alternativen zur Schablonensuche

Zwei alternative Ansätze wurden zu Beginn erwähnt. Deshalb hier zumindest jeweils zwei Bilder, um ein Gefühl für die Daten zu bekommen, die statt der Trefferquoten entstehen.

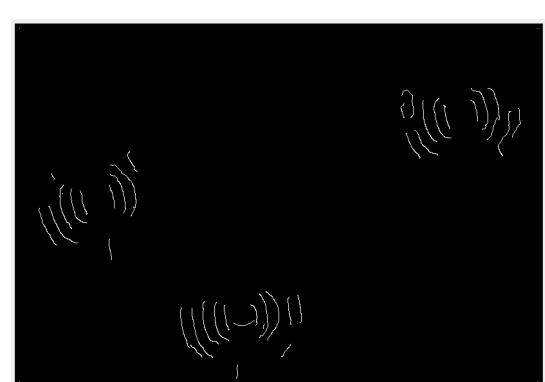
Ausgangsbilder:



Blob-Aufteilung:



Kantenerkennung:



3.4 Bewertungskriterien

- Angesichts der Größe der vorgegebenen Bilder sollte die Laufzeit des Verfahrens begrenzt bleiben. Die meisten sinnvoll anwendbaren Verfahren sind linear in der Anzahl n

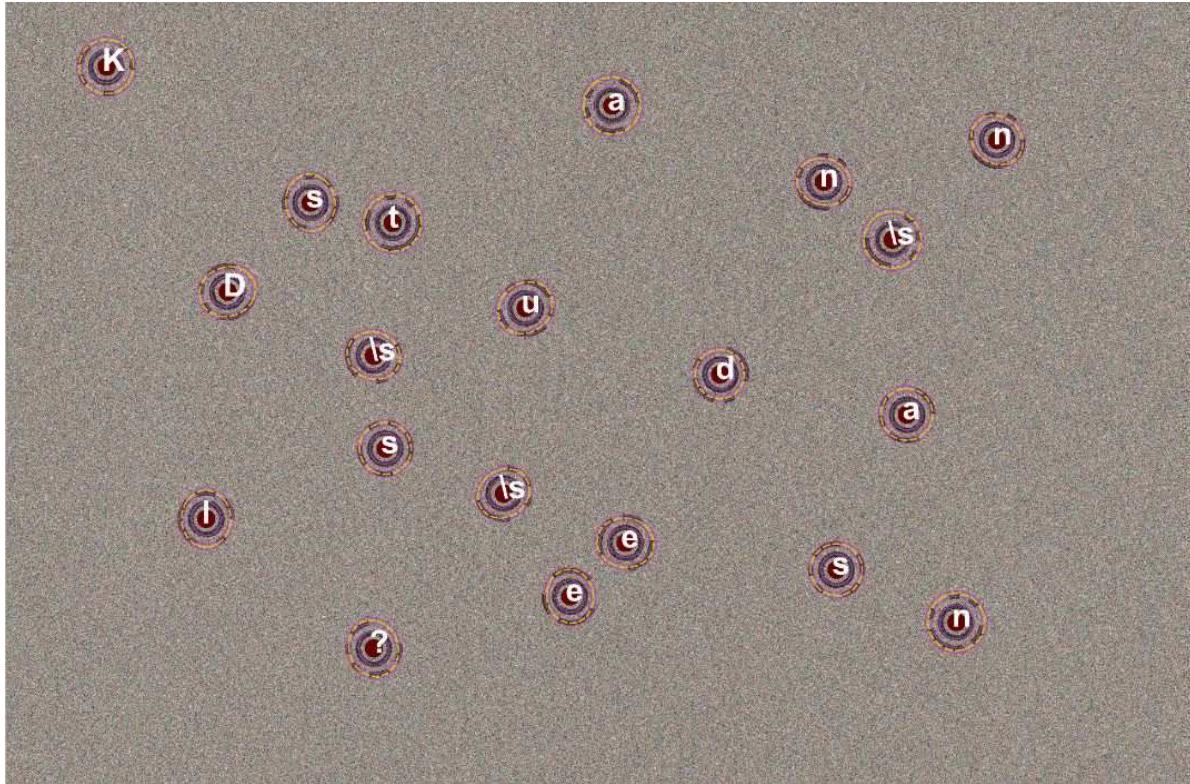
der Bildpunkte. Insgesamt sollte die Laufzeitkomplexität maximal in $\mathbf{O}(n \cdot \ln(n))$ liegen. Fallen bei aufwändigen Analysen große Datenmengen an, kann dem durch Rasterung, Clustern oder Durchschnittsbildungen begegnet werden.

- Die Teilaufgaben 1 und 2, also Mittelpunktbestimmung und Dekodierung für ein Schwarz-Weiß-Bild mit sauber kreisförmigen Codes, sollten erfolgreich bearbeitet worden sein.
- In Teilaufgabe 3 wird es schwieriger. Die Bewertung hängt davon ab, mit welchen Schwierigkeiten die Lösung zurecht kommt. Kritisch sind folgende Aspekte:
 - Verzerrungen: Wer nur (Zitat aus einer Einsendung) „runde Kreise“ erkennt, kommt mit Verzerrungen z.B. bei Aufnahmen von der Seite nicht zurecht.
 - Unschärfe und Rauschen erfordert geeignete Vorverarbeitung durch Filtern und Glätten etc. Es sollte darauf geachtet werden, dass bei anderen Bildern potenziell sehr kleine Kreiscodes nicht kaputt geglättet werden.
 - Die mögliche Rotation der Segmentübergänge sollte berücksichtigt werden.
 - Schwankende Helligkeiten z.B. durch Schatten machen u.a. erforderlich, dass eine Bitzuweisung je Segment relativ zum Umfeld geschieht und nicht anhand einer vorher festgelegten Helligkeitsschwelle.
 - Bonuspunkte werden vergeben, wenn das Programm überdurchschnittlich gut ist und auch mit schwierigen Störeffekten bzw. Testbildern zureckkommt.
- Laufzeitüberlegungen werden auch bei dieser Aufgabe erwartet; hier werden allerdings häufig Schätzungen oder qualitative Bewertungen der Auswirkungen bestimmter Parameter einfließen müssen. Deshalb sollten zusätzlich auch Laufzeitmessungen durchgeführt werden, die die Schätzungen (hoffentlich) bestätigen.
- Auch bei dieser Aufgabe können bekannte Verfahren eingesetzt werden. Ähnlich wie in Aufgabe 1 muss gut begründet werden, dass diese Verfahren funktionieren; es soll auch erkennbar werden, dass die Effekte übernommener Verfahren verstanden wurden.
- Die Wahl der zentralen Parameter und Toleranzgrenzen für die Identifikation von Kreiscode-Zeichen sollte geeignet sein und begründet werden.
- Die vorgegebenen Beispiele sollten bearbeitet sein. Weitere, selbst erstellte Beispiele sind insbesondere dann sinnvoll, wenn damit weitergehende Leistungen, aber auch die Grenzen der Lösung aufgezeigt werden. In der Informatik sollte man sich immer darüber im klaren sein, was ein System kann – und was es nicht kann.
- Es ist nicht nötig, die gefundenen Dekodierungen in die Bilder einzubauen. Aber es genügt auch nicht, einfach den dekodierten Text auszugeben (der in den Beispielen ohnehin immer gleich ist). Die Ausgabe sollte nachvollziehbar machen, wie das Verfahren zur Dekodierung gekommen ist. Zwischenabgaben nach verschiedenen Arbeitsschritten (Filterung, Mittelpunktbestimmung etc.) sind sinnvoll.
- Da die Aufgabe ohnehin Kreativität erfordert, eröffnet sich auch Spielraum für Erweiterungen. Neue Probleme entstehen, wenn die Codes andere Formen annehmen können. Ansprüche an die Bitzuweisung erhöhen sich, wenn eine deutlich größere Code-Tabelle und damit eine deutlich höhere Auflösung der Bit-Codierung angenommen wird.

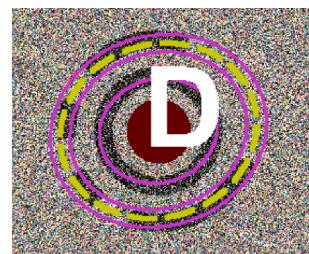
3.5 Beispiele

In allen vorgegebenen Beispielen ist Folgendes kodiert: Kannst Du das lesen?

noise50.jpg



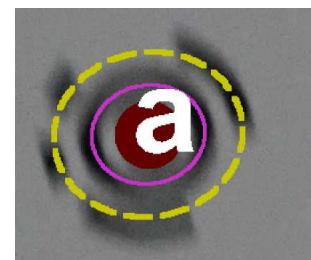
.0001.0010.1100.0101 (x: 328, y: 211): K
.0011.1010.1000.1110 (x: 1981, y: 338): a
.1010.0101.0011.0111 (x: 3240, y: 451): n
.1101.0010.1001.1011 (x: 2673, y: 587): n
.1001.0101.1100.1101 (x: 995, y: 655): s
.1111.1001.1001.0101 (x: 1264, y: 721): t
.0100.1111.0000.0101 (x: 2898, y: 776): \s
.1010.1011.1000.0111 (x: 724, y: 946): D
.1101.1001.0111.0101 (x: 1698, y: 999): u
.0111.1000.0010.1010 (x: 1202, y: 1155): \s
.0010.1110.1000.1111 (x: 2334, y: 1216): d
.0011.1010.1000.1110 (x: 2945, y: 1347): a
.1001.1011.0010.1011 (x: 1240, y: 1458): s
.1111.0000.0101.0100 (x: 1627, y: 1605): \s
.0100.1111.1011.0110 (x: 654, y: 1688): l
.0011.1101.1001.1110 (x: 2025, y: 1766): e
.0110.0101.0111.0011 (x: 2715, y: 1854): s
.0011.1100.0111.1011 (x: 1844, y: 1941): e
.1101.1110.1001.0100 (x: 3107, y: 2025): n
.1011.0000.1101.1010 (x: 1202, y: 2111): ?



cam8.jpg

Die „Viertel-Kreiscodes“ in den Ecken werden erkannt, die (naturgemäß erfolglose) Decodierung wird hier nicht angegeben.

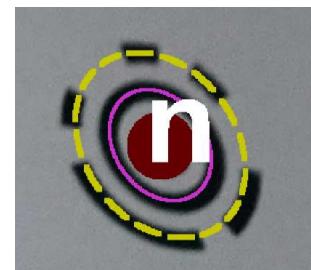
```
.1001.0110.0010.1000 (x: 334, y: 211): K
.1101.0100.0111.0001 (x: 1990, y: 340): a
.1110.1001.0100.1101 (x: 3239, y: 445): n
.1111.0100.1010.0110 (x: 2676, y: 580): n
.0110.1100.1010.1110 (x: 1000, y: 647): s
.1001.1001.0101.1111 (x: 1269, y: 709): t
.1111.0000.0101.0100 (x: 2898, y: 761): \s
.0101.0111.0000.1111 (x: 726, y: 930): D
.0111.0110.0101.1101 (x: 1703, y: 980): u
.1010.0111.1000.0010 (x: 1208, y: 1132): \s
.0010.1110.1000.1111 (x: 2343, y: 1191): d
.0011.1010.1000.1110 (x: 2953, y: 1322): a
.1001.1011.0010.1011 (x: 1250, y: 1437): s
.1001.1110.0000.1010 (x: 1638, y: 1583): \s
.0100.1111.1011.0110 (x: 665, y: 1669): l
.1110.0011.1101.1001 (x: 2038, y: 1748): e
.0110.0101.0111.0011 (x: 2725, y: 1837): s
.0111.1011.0011.1100 (x: 1857, y: 1927): e
.1101.1110.1001.0100 (x: 3115, y: 2010): n
.1011.0101.0110.0001 (x: 1214, y: 2101): ?
```



camB.jpg

Die „Viertel-Kreiscodes“ in den Ecken werden erkannt, die (naturgemäß erfolglose) Decodierung wird hier nicht angegeben.

```
.0001.0010.1100.0101 (x: 321, y: 217): K
.0011.1010.1000.1110 (x: 1983, y: 325): a
.0100.1010.0110.1111 (x: 3265, y: 330): n
.0111.1010.0101.0011 (x: 2688, y: 483): n
.0011.1100.0001.0101 (x: 2922, y: 641): \s
.1100.1101.1001.0101 (x: 984, y: 663): s
.1111.1001.1001.0101 (x: 1253, y: 724): t
.1111.0101.0111.0000 (x: 710, y: 967): D
.0111.0110.0101.1101 (x: 1688, y: 1003): u
.1011.1010.0011.1100 (x: 2334, y: 1150): d
.1010.0111.1000.0010 (x: 1186, y: 1168): \s
.0111.0101.0001.1100 (x: 2967, y: 1209): a
.1001.1011.0010.1011 (x: 1222, y: 1475): s
.1111.0000.0101.0100 (x: 1608, y: 1598): \s
.1000.1111.0110.0111 (x: 2010, y: 1698): e
.0100.1111.1011.0110 (x: 638, y: 1714): l
.0110.0101.0111.0011 (x: 2702, y: 1801): s
.1111.0110.0111.1000 (x: 1820, y: 1894): e
.1101.1110.1001.0100 (x: 3095, y: 1996): n
.1011.0101.0110.0001 (x: 1176, y: 2091): ?
```



Eigenes Beispiel

Dieses Foto ist durch stark unterschiedliche Lichtverhältnisse, Rauschen und Verformungen des Papiers besonders schwierig und zeigt selbst sehr guten Lösungen ihre Grenzen auf.

