Kantendetektion von geometrischen Konturen in der Computergrafik am Beispiel des Kreises mit Python

Dokumentation einer Projektarbeit

Dustin Winkler
FB Automatisierung und Informatik/Hochschule Harz
Wernigerode, Sachsen-Anhalt, Deutschland
u35151@hs-harz.de

ABSTRACT

Die vorliegende Arbeit befasst sich mit der Fragestellung, wie die in der Programmiersprache *Python* bereits implementierten Funktionen (in weiterführenden Bibliotheken) genutzt werden können, um vordefinierte Konturen in digitalen Fotografien zu lokalisieren. Praxisnah wird das Beispiel für die Zählung von Achsen bei KFZ-Fahrzeugen verwendet. Hierbei dient eine Sammlung von Fotografien als Basis für die Analyse der Funktionsfähigkeit. Weiterführend wird in diesem Paper die Arbeitsweise der Kantendetektion beschrieben. Dies erfolgt primär am Beispiel des Canny-Filters.

CCS CONCEPTS

• Computergrafik → Kantendetektion

KEYWORDS

Computergrafik, Canny-Filter, Python, Kontendetektion

ACM Reference format:

Dustin Winkler. 2021. Kantendetektion von geometrischen Grundkonturen in der Computergrafik am Beispiel des Kreises mit Python: Dokumentation einer Projektarbeit. In *Proceedings of ACM Woodstock conference (WOODSTOCK'18). ACM, New York, NY, USA, 2 pages.* https://doi.org/10.1145/1234567890

1 Einleitung

Das menschliche Auge kann als Linse betrachtet werden, welche das Originalbild auf die Netzhaut (Retina) abbildet. Die Netzhaut enthält eine Vielzahl an Fotorezeptoren. Grundlegend erfolgt hierbei eine Unterteilung in Stäbchen und Zapfen. Die Stäbchen registrieren hierbei die Intensität und sind mit einer Anzahl von $130 * 10^6$ pro Auge vorhanden. Hierbei genügen bereits 256 Grauwerte für eine visuell-realistischen Eindruck. Die Zapfen sind in einer deutlich geringeren Anzahl vorhanden. Pro Auge entfallen $6 * 10^6$ Zapfen für die Registierung der Farben ($7 * 10^6$ Farbtöne, welche jeder Zapfentyp für Grün, Rot oder Blau (in absteigender Empflindlichkeit) wahrnehmen kann. Neben diesen Rezeptoren enthält das Auge eine Reihe von essenziellen Filtern, welche durch das Verschalten von Nervenenden zustande kommt. Einer der bedeutensten Filter ist der sogenannte Kantenfilter. Die Wichtigkeit resultiert aus der Art der Wahrnehmung von Objekten, welche vom menschlichen Auge primär über Kanten erfolgt. In Folge dessen sind Bildinhalte bereits erkennbar, wenn nur wenig Konturen sichtbar sind. Zudem steht der subjektive Schärfeeindruck eines Bildes in direktem Zusammenhang mit seiner Kantenstruktur. Somit ist es für den Menschen möglich, ein Bild beinahe vollständig aus Kanten zu rekonstruieren [2].

Abgeleitet aus der Anatomie des Menschen ergibt sich somit eine enorme Bedeutung der Kantenerkennung in der Visualisierung. Grundlage hierbei ist jedoch eine Definition des Begriffes *Kante* für die Computergrafik.

Eine Kanten markiert hierbei die Positionen in einem Bild, an welcher sich die Intensität entlang einer Vorzugsrichtung lokal ändert.

Die Änderung der Intensität kann im wesentlichen durch die erste Ableitung definiert werden.

1.1 mathematische Grundlagen

Betrachtet man Abbildung 1.1.1, so erkennt das menschliche Auge einen Kreis in Kombination mit einer schwarzen Linie.

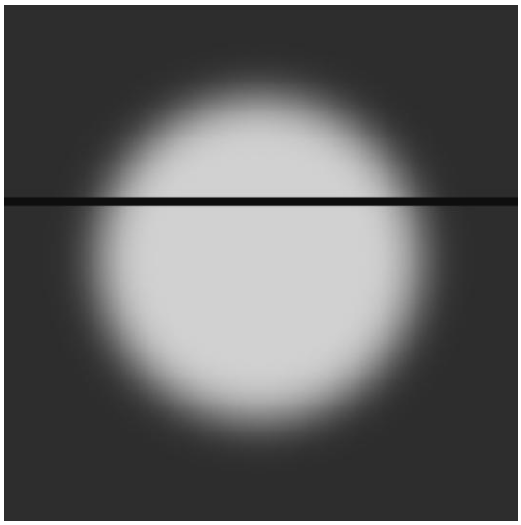

Abbildung 1.1.1: Kreis mit Bildzeile

WOODSTOCK'18, June, 2018, El Paso, Texas USA
 978-1-4503-0000-0/18/06...$15.00
https://doi.org/10.1145/1234567890

Beim Auslesen der entsprechenden Bildzeile in Leserichtung (von links nach rechts) sowie der Translation der resultierenden Informationen in einem Grafen erhielte man die Funktion *f(u)* aus Abbildung 1.1.2. Charakteristisch ist hierbei die Intensität der Helligkeit, welche zu Beginn des Kreises ansteigt und anschließend (innerhalb des Kreises) in einem Hoch verbleibt, bevor beim Verlassen des Kreises erneut das Ausgangsniveau erreicht wird.

Abbildung 1.1.2: Funktion *f(u)* [1]

Bei der Bildung der ersten Ableitung der Funktion *f'(u)* werden alle linearen Verläufe gemindert und auf den Nullwert gesetzt. Die Änderungsbereiche bleiben jedoch weiterhin als Extremwerte erhalten (siehe Abbildung 1.1.3). Hierbei verdeutlicht ein positiver Extremwert den Kontureintritt und ein negativer Extremwert einen Austritt.

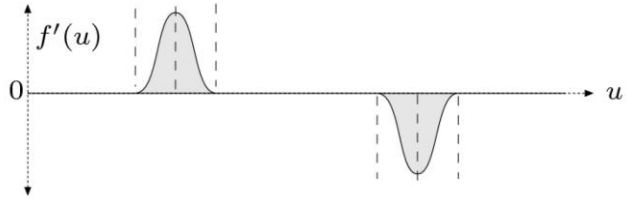

Abbildung 1.1.3: erste Ableitung *f'(u)*[1]

Dieser Effekt kann verstärkt werden, wenn eine Kantenschärfung durchgeführt wird. Dies kann mit dem Laplace Filter geschehen, welcher im Wesentlichen auf der Bildung der zweiten Ableitung *f''(u)* basiert (siehe Abbildung 1.1.4).

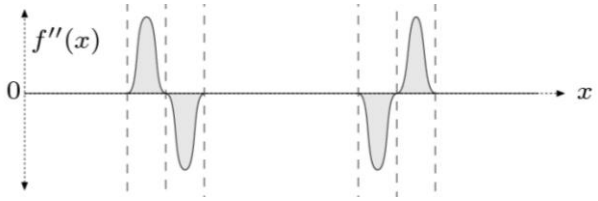

Abbildung 1.1.4: zweite Ableitung *f''(u)* [1]

Erfolgt im Anschluss eine Subtraktion der zweiten Ableitung *f''(u)* vom Originalbild *f(u)*, so wir ein Überschuss erzeugt, Wogleich der Anstieg eine signifikante Steigung erfährt (siehe Abbildung 1.1.5). Dies wird vor allem im Vergleich mit der Ausgangsfunktion (siehe Abbildung 1.1.2) deutlich. Der subjektive Effekt der Schärfesteigerung erfolgt für das menschliche Auge aus den Überschüssen, welche in Abbildung 1.1.5 durch vier Kreise symbolisiert wird.

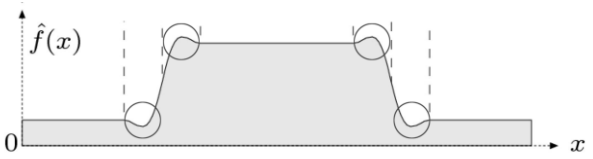

Abbildung 1.1.5: Funktion $\hat{f}(u)$ subtrahiert mit *f''(u)* [1]

Zusätzlich wird die Ableitung *f''(u)* mit einem Gewicht w multipliziert, welche abhängig vom entsprechenden Bild gewählt wird (siehe Abbildung 1.1.6).

$$\hat{f}(x) = f(x) - w * f''(x)$$

Abbildung 1.1.6: Formel für $\hat{f}(u)$

Wie bereits im Abstract formuliert wurde, erfolgt die Bildanalyse mit digitalen Fotografien, wodruch die Bilder in einer Pixelauflösung vorliegen. Aus diesem Grund ist die Bildung einer kontinuierlichen Funktion und somit eines Gradienten unmittelbar aus der Aufnahme nicht möglich.
Einen einfachen Lösungsansatz stellt hierbei eine Differenz der Funktionswerte dar. Wenn für ein beliebigen Pixelwert *(u)* die Ableitung ermittelt werden soll, dann kann dies durch eine Subtraktion der beiden Nachbarpixelwerte *(u-1)* und *(u+1)* sowie anschließenden Division des Produktes durch die Entfernung *(2)* ermittelt werden. Das Ergebnis ist eine Sekante, welche durch die Punkte *(u-1)* und *(u+1)* verläuft. Diese ist eine Näherung zur Steigung, welche durch die Ableitung am Punkt *(u)* bestimmbar ist (siehe Abbildung 1.1.7 und 1.1.8).

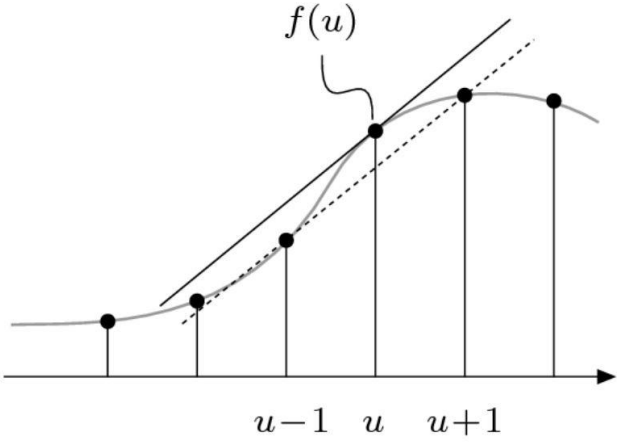

Abbildung 1.1.7: Nachbarpixel von *f(u)* [1]

$$\frac{df(u)}{du} \approx \frac{1}{2}(f(u+1) - f(u-1))$$

Abbildung 1.1.8: Formel zur Steigung der Sekante

Da es sich beim Ausgangsprodukt jedoch nicht um eine Bildzeile handelt, sondern um eine digitale Fotografie, muss die Auslesung in beiden Intensitäten bzw. Richtungen berücksichtigt werden (Richtung *u* und Richtung *v*). Folglich werden auch beide Ableitungen von *u* und *v* benötigt. Dies kann nun im sogenannten Gradientenvektor definiert werden (siehe Abbildung 1.1.9).

$$\nabla I(u,v) = \begin{bmatrix} \partial_u I(u,v) \\ \partial_v I(u,v) \end{bmatrix}$$

Abbildung 1.1.9: Gradientenvektor

Anschließend kann die Ableitung für jeden Punkt in *u* und *v* Richtung geschätzt/berechnet werden (siehe Abbildung 1.1.1). [1][2][3]

2 Kantendetektion

Für diese Aufgabe gibt es eine Mehrzahl an Operatoren, welche auch Filter genannt werden. Jede dieser Filter hat das Ziel, die lokale Ableitung zu schätzen/berechnen (wie in Abschnitt 1.1 mathematische Grundlagen beschrieben wurde). Der Unterschied innerhalb der Verfahren liegt in den Richtungskomponenten.
Wie bereits im Abstract definiert wurde, handelt es sich bei Kanten in der Computergrafik um Regionen, bei welchen es zu einer starken Veränderung der Intensitäten kommt. Die Grundidee der Kantendedektion ist es, überflüssige Informationen aus dem Bild zu entfernen und somit die Datenmenge signifikant zu verringern.

2.1 Prewitt Operator

Beim nach Judith M.S. Prewitt benannter Verfahren wird für jeden Y-Wert in der X-Richtung der selbe Differentenquotient genutzt. Dies wird auf der Y-Richtung gleichermaßen vollzogen, indem der X-Wert den gleichen Differentenquotien erhält. Somit erhalten wir H_x^P und H_y^P mit den folgenden Matrizen:

$$H_x^p = \begin{pmatrix} -1 & \mathbf{0} & 1 \\ -1 & \mathbf{0} & 1 \\ -1 & \mathbf{0} & 1 \end{pmatrix}$$

$$H_y^p = \begin{pmatrix} -1 & -1 & -1 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 1 & 1 & 1 \end{pmatrix}$$

2.2 Sobel Operator

Die Funktionsweise des Sobel Filters im Vergleich zum Prewitt Filter ist grundsätzlich besser einzuordnen. Hierbei wird im Gegensatz zum Prewitt Operator die mittlere Zeile bzw. Die mittlere Spalte doppelt so stark gewichtet.

$$H_x^S = \begin{pmatrix} -1 & 0 & 1 \\ -\mathbf{2} & \mathbf{0} & \mathbf{2} \\ -1 & 0 & 1 \end{pmatrix}$$

$$H_y^S = \begin{pmatrix} -1 & -\mathbf{2} & -1 \\ 0 & \mathbf{0} & 0 \\ 1 & \mathbf{2} & 1 \end{pmatrix}$$

2.2 Kantenstärke und Kantenrichtung

Sowohl für den Prewitt- als auch den Sobel Operator erhält man durch die Ableitung in beide Richtungen *u* und *v* ein neues gefiltertes Bild *(D)* in X- und Y-Richtung (I ist das Originalbild bzw. Ausgangsbild):

$$D_x(u,v) = H_x * I$$

$$D_y(u,v) = H_y * I$$

Weiterführend wird die Kantenstärke E und Kantenrichtung Φ wie folgt bestimmt:

$$E(u,v) = \sqrt{(D_x(u,v))^2 + (D_y(u,v))^2}$$

$$\Phi(u,v) = \arctan\left(\frac{D_y(u,v)}{D_x(u,v)}\right)$$

Mit den gewonnenen Informationen kann anschließend die eigentliche Kantendetektion stattfinden. Hierbei wird aus dem Ausgangsbild I (siehe Abbildung 2.2.1) zwei Bildkopien generiert, welche die Ableitung in der x-Richtung (H_x) und die Ableitung in die y-Richtung (H_y) definieren. Weiterführend erhält man zwei gefilterte Bilder D_x und D_y. Diese werden anschließend zusammengesetzt wodurch sich die Kantenstärke und die Kantenrichtung ergibt [1][3][6].

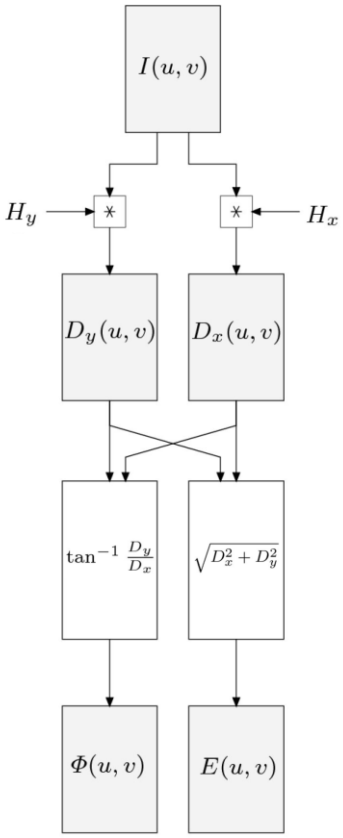

Abbildung 2.1.1: Ablauf der Kantendetektion [1]

2.4 Canny Operator

Ein optimaler Kantenfilter sollte möglichst alle Kanten detektieren, ohne ein Übermaß an Ausschuss zu erzeugen. Zudem sollte eine gute Lokalisation vorhanden sein, wodurch eine minimale Distanz zwischen detektierter und echter Kante vorherrschen sollte. Zuletzt sollte der optimal Kantenfilter eine klare und eindeutige Antwort liefern, indem es nur eine Lösung pro Kante gibt, diese also nicht mehrfach identifiziert werden können.

Diese Anforderungen erfüllt der nach John Francis Canny benannte Filter auch nach heutigen Kentnissen am zuverlässigsten und zählt somit zu den populärsten Kantenfiltern.

In seiner Urform wurde hierbei ein Satz von gerichteten und großen Filtern auf mehreren Auflösungsebenen eingesetzt. Diese Ergebnisse wurden anschließend zu einem gemeinsamen Kantenbild zusammengesetzt.

Heutzutage wird primär auf einer Skalenebene gearbeitet, jedoch einen Glättungsradius σ für den Filterradius verwendet. Je kleiner der Glättungsradius, desto feiner und detaillierter werden die Kanten gefunden. Das Sigma stellt hierbei die Standardabweichung im Gaußfilter dar.

Die Vorgehensweise des Canny Operators kann grundsätzlich in 5 Schritte unterteilt werden.

2.4.1 Schritt 1. Der erste Schritt markiert die Anwendung eines Glättungsfilters. Hierbei kommt typischerweise der Gaußfilter der Breite σ zum Einsatz. Dieser bestimmt die Rauschempfindlichkeit und Breite der zu detektierenden Kanten.

2.4.2 Schritt 2. Im nachfolgenden Schritt findet die Unterdrückung von Nichtmaxima und somit die Findung der Kantenstärken durch die Gradientenbildung statt (dies kann beispielsweise durch den Sobel-Operator geschehen). Als erstes werden hierbei alle lokalen Maxima der Kantenstärke gesucht, welche jedoch senkrecht und nicht entlang der Kante verlaufen. Dazu wird die lokale Gradientenrichtung bestimmt und anschließend werden alle Pixel entfernt, welche nicht den maximalen Gradienten in ihrer Umgebung haben. Die Bestimmung der lokalen Gradientenrichtung kann wie folgt berechnet werden:

$$\Phi(u,v) = \tan^{-1}\left(\frac{D_y(u,v)}{D_x(u,v)}\right)$$

2.4.3 Schritt 3. Da es sich beim Eingangsbild um eine Pixelauflösung handelt, ergeben sich vier Richtungen, in welche die Kanten verläufen. Jedes Pixel besitzt typischerweise 8 Nachbarn (siehe Abbildung 2.4.3.1), folglich kommen nur die Richtungen in 0, 45, 90 oder 135 Grad infrage. Die 4 Richtungen ergeben sich hierbei aus der Symmetrie der 8 Nachbarn.

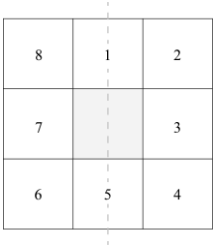

Abbildung 2.3.4.1: Pixelnachbar

2.4.4 Schritt 4. Im vorletzten Schritt werden die nun gefundenen Kanten weiter verfolgt und alle Pixel entfernt, welche nicht zur Kante gehören.

2.4.5 Schritt 5. Der fünfte Schritt dient der Schwellwertbildung. Hierbei soll die Unterbrechung von Linien durch die Nutzung von zwei Schwellwerten θ_1 und θ_2 verhindet werden. Um eine Kante zu beginnen ist hierbei ein Schwellwert θ_1 nötig. Sollte der Schwellwert der Nachbarpixel $\theta_2 < \theta_1$ betragen, so werden diese Pixel der Kante zugerechnet. Hierbei werden die Maxima unterhalb von θ_1 verworfen und die Werte über θ_2 beibehalten. Für alle Zwischenwerte gibt es seine Hysterese, welche besagt, dass das lokale Maximum ebenfalls übernommen wird, wenn mindestens ein Nachbar akzeptiert wurde [2][3][6].

3 Umsetzung

In den bisherigen Abschnitten wurde beschrieben, wie Kanten algorithmisch in digitalen Bildern gefunden werden können. Bei der zu Beginn formulierten Fragestellung bleibt jedoch offen, wie die gefundenen Kanten anschließend als geometrische Kontur definiert werden können und wie dies in Python praxisnah umgesetzt werden kann.

Innerhalb der Programmierumgebung wurde hierbei zuerst das Originalbild geladen und in ein schwarz-weiß Kopie umgewandelt. Um ausschließlich mit Werten im Wertebereich von 1 bis 0 zu arbeiten, wurde anschließend eine Byte-Form Transformation durchgeführt. Hierbei wurden die Bibliotheken von skimage (skimage.color und skimage.util) verwendet. Der Datenstream wurde nachfolgend dem Canny-Filter übergeben. Dieser wurde durch das Laden der skimage.draw Bibliothek bereits importiert (siehe Abbildung 3.1) und funktioniert wie im vorherigen Abschnitt beschrieben. Die Werte für Sigma sowie low_threshold und high_threshold wurden experimentell ermittelt.

```
from scipy.spatial import distance
from skimage.color import rgb2gray
from skimage import data, color, io
from skimage.transform import hough_circle, hough_circle_peaks
from skimage.feature import canny
from skimage.draw import circle_perimeter
from skimage.util import img_as_ubyte
from skimage.transform import rescale, resize, downscale_local_mean

filename = os.path.join('eigenenPfadHierAngeben', 'auto_04.jpg')
car = io.imread(filename)
car_grey = rgb2gray(car)
image = img_as_ubyte(car_grey)
edges = canny(image, sigma=3, low_threshold=0.7, high_threshold=0.8)
```

Abbildung 3.1: Bildübergabe in Python

Nach der Anwendung des Canny-Filters wird anschließend eine Hough-Transformation durchgeführt.

3.1 Hough-Transformation

Das Ziel der Hough-Transformation ist es, geometrische Primitive wie Kreise, Quadrate oder Parabeln zu finden. Die Hough-Transformation ist somit in der Lage, alle Formen zu definieren, welche geometrisch beschrieben werden können. Bei der einfachsten geometrischen Form, einer Geraden, wird nun ein Kantenbild (ein Gradientenbild, welches den Rückgabewert des Canny-Filter entspricht), verwendet. Anschließend wird die Richtung der größten Grauwertveränderung untersucht und ausgehend von diesem Vektor der Betrag berechnet, welcher daraufhin im Kantenbild dargestellt wird. Anschließend soll herausgefunden werden, welche Pixel in einem Bild eine Gerade bilden.

Hierbei baut der Hough-Algorithmus (welcher im vorliegenden Python-Programm durch die Bibliothek von skimage.transform implementiert wurde) zuerst einen Dualraum auf. Dieser Dualraum besteht aus zwei diskreten Seiten, einem Ortsraum auf der linken Seite (siehe Abbildung 3.1.1) und einem Parameterraum auf der rechten Seite.

Der Akkumulator kann hierbei als Speicher definiert werden. Mit jedem Pixel ist außerdem eine Variable verbunden. Der Dualraum kann als 2D Array dargestellt werden, dessen Elemente inkrementiert werden können. Auf der linken Seite (dem sogenannten Ortsraum) wird eine Koordinate (x,y) wie gewohnt im Euklidischen Koordinatensystem interpretiert. Auf der Parameterseite stehen die Achsen für die Steigung m einer Geraden und die Höhe b (siehe Abbildung 3.1.1). Ein Punkt im Parameterraum entspricht somit einer Geraden im Ortsraum. Der Y-Achsenabschnitt verschiebt die Gerade nach oben und unten und die B-Achse entspricht mit der Y-Achse im Ortsraum.

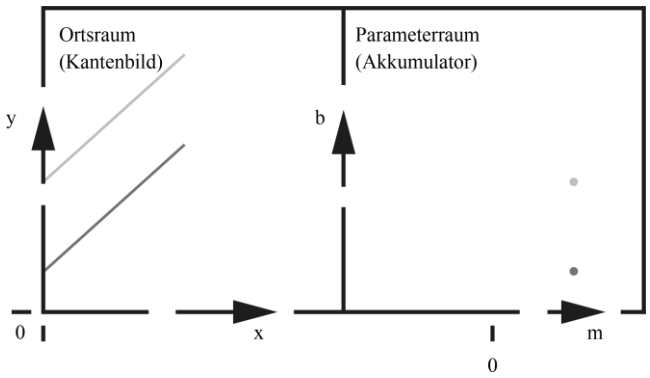

Abbildung 3.1.1: Dualraum

Anmerkung Auf die Problemstellung der Steigung m in Bezug auf Winkel über 45° (von 0° - 45° liegt der Wertebereich der Steigung zwischen 0 und 1 und von 45° - 90° expandiert der Wertebereich von 1 bis ∞) wird in der vorliegenden Arbeit nicht weiter eingegangen. Die Verbesserung der Geradengleichung durch die Arbeit am Einheitskreis ergibt die Umstellung der Parameterform zur Impliziten Form, welche sich zum Testen eines beliebigen Punktes (x,y) besser eignet, da eine Überprüfung ausschließlich auf das Ergebnis 0=0 für die Annahme der Richtigkeit erfolgen muss (siehe Abbildung 3.1.2).

$$y = mx + b$$

$$y = \frac{\sin(\propto)}{\cos(\propto)}x + b$$

$$0 = \sin(\propto)\,x + \cos(\propto)(b - y)$$

Abbildung 3.1.2: Verbesserung der Geradengleichung

Eine Transformation vom Parameterraum in den Ortsraum kann somit als trivial angesehen werden. Die Schwierigkeit besteht nun darin, die binär gesetzten Punkte in den Parameterraum zu übertragen. Anschließend muss überprüft werden, welche Pixel sich im Parameterraum etablieren. Die Lösung des Transformationsproblems eines beliebigen Punktes P vom Ortsraum in einen Parameterraum liegt hierbei in der Untersuchung aller Geraden. Somit ist jede Gerade, welche durch

Punkt P verläuft, ein möglicher Kandidat. Dies hat die Bildung einer Geradenschar zur Folge.
Für jede Gerade der Geradenschar kann nun im Parameterraum das entsprechende Alpha-Wert berechnet werden (also der Winkel der Gerade) und den Y-Achsenabschnitt (also der Punkt, an welchem die Y-Achse geschnitten wird). Das Ergebnis ist eine geschwungene Linie (siehe Abbildung 3.1.3).
Hinweis, die Achsenbezeichnung wurde durch die Umstellung auf die Impliziten Form von m in α geändert

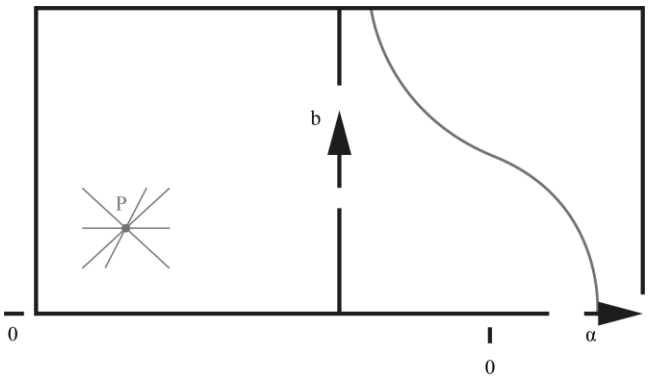

Abbildung 3.1.3: Geradenschar

Die Durchführung mehrere Werte zeigt, dass unterschiedliche Punkte verschiedene Kurven im Parameterraum erzeugen. Oft treffen sich viele geschwungene Linien in einem Punkt P im Parameterraum, was bedeutet, das die Linie, welche P repräsentiert, in vielen Geradenscharen im Ortsraum vorkommt (siehe Abbildung 3.1.4) und somit auch eine Gerade im Bild darstellen könnte.

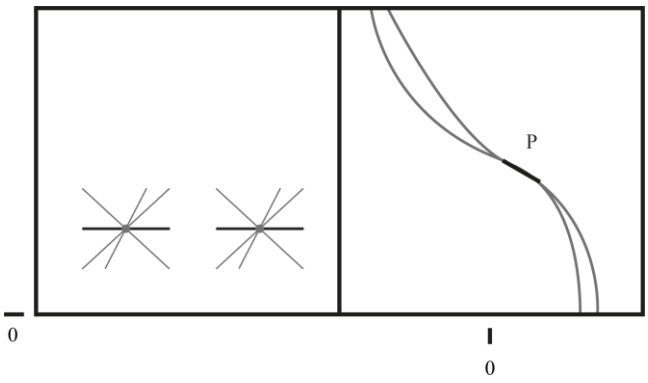

Abbildung 3.1.4: unterschiedliche Punkte

Dies zeigt die grundlegende Funktionsweise der Hough-Transformation. Es befinden sich im Ortsraum (im Ausgangsbild) eine Vielzahl einzelner Punkte, jedoch ist unklar, zu welcher Linie diese Punkte gehören. Aus diesem Grund soll keine Linie ausgeschlossen werden, was dazu führt, das alle Linien, welche durch den Punkt gehen, den entsprechenden Akkumulator im Parameterraum inkrementiert. Wenn dies für alle Punkte vollzogen wurde, so erhält man im Parameterraum eine Kumulierung an einer Stelle, welche somit ein Kandidat für eine Linie im Ausgangsbild definiert. Je mehr Geraden sich also im Parameterraum schneiden, desto mehr Punkte liegen auf der entsprechenden Gerade im Bildraum.
Die bisherigen Berechnungsschritte dienen jedoch ausschließlich der Arbeit mit Geraden. Die Funktionsweise bei der Darstellung eines Kreises funktioniert indessen ähnlich. Allerdings wird in diesem Fall nicht mit zwei Parametern gearbeitet, sondern mit drei (Mittelpunkt [x+y] und Radius).
Bei der Umsetzung in Python wurde für die Hough-Transformation die Methoden hough_circle und hough_cirlce_peaks aus der externen Bibliothek skimage.transform implementiert. Wie in Abbildung 3.1.5 zu sehen ist, wird hierbei zuerst ein Array aufgestellt (Zeile 21), welches die Werte von 50px bis 200px in Zweierschritten beinhaltet.

```
21  hough_radii = np.arange(50, 200, 2)
22  hough_res = hough_circle(edges, hough_radii)
23
24  accums, cx, cy, radii = hough_circle_peaks(hough_res, hough_radii,
25                                             total_num_peaks=10)
```

Abbildung 3.1.5: Hough-Transformation in Python

Anschließend wird die hough_circle Funktion aufgerufen. Dieser wird der bereits gespeicherte Canny-Operator sowie das Array übergeben. Der Canny-Operator enthält hierbei alle Kanten, welche im Bild gefunden wurden. Anschließend sucht die Hough-Transformation in unserem Ausgangsbild nach allen Kreisen für die entsprechenden Werte in unserem Array (vom Durchmesser 50px bis 200px in Zweierschritten).
Mit der hough_circle_peaks Funktion werden anschließend die 10 besten Kreise ausgewählt (die Anzahl der auszuwählenden Kreise kann durch den Parameter total_num_peaks definiert werden). Zusätzlich bekommt die Methode alle gefundenen Kreise und das Array mit den Durchmessern. Die hough_circle_peaks Methode gibt den x- und y-Wert des Mittelpunktes sowie den Radius zurück. Weiterhin erhalten wir einen Wert (accums), welcher die Wahrscheinlichkeit widerspiegelt, mit der die Methode einen Kreis identifiziert hat [3][4][5].

3.2 Auswahl der Konturen

An dieser Stelle des Programmcodes wurden die Werte der 10 wahrscheinlichsten Konturen für einen Kreis definiert. Das Ausgangsproblem beschrieb jedoch die Annahme, Achsen von KFZ-Fahrzeugen zählen zu können. Im aktuellen Verfahren werden alle Kreise im Bild erkannt, wodurch gewisse Fahrzeugeigenschaften (Räder, Felgen, Kotflügel) als Kreise erkannt werden, diese jedoch zu einer Achse gehören. Im vorletzten Schritt muss nun herausgefunden werden, wie die doppelte Zählung verhindert werden kann.
Der Lösungsansatz liegt im Vergleich der Abstände der Mittelpunkte. Hierbei sollen alle Kreise für die spätere Zählung irgnoriert werden, welche innerhalb des Radius eines Ausgangskreises liegen. Für die Umsetzung in Python wurde hierbei zuerst ein Array 'x' generiert (siehe Zeile 37 in Abbildung 3.2.1), welches alle Mittelpunkte der Kreise beinhaltet.

Anschließend wurde die distance.pdist und distance.squareform Funktionen implementiert, welche aus der externen Bibliothek scipy.spatial geladen wurde. Diese Methode erstellt eine 2D Matrix und erfasst die Distanzen zwischen den einzelnen Werten untereinander (und somit zwischen den Mittelpunkten). Der letzte Schritt vor der eigentlichen Schleife lag im Aufbau einer Liste, welche alle Kreise enthält, die im späteren Verlauf nicht gezählt werden sollen. Diese Liste wurde als Set angelegt, um die doppelte Speicherung von Werten zu verhindern (Zeile 40 in Abbildung 3.2.1).
Die for-each-Schleife (von Zeile 41 bis 44 in Abbildung 3.2.1) bearbeitet anschließend alle erkannten Radien. Hierbei dient das 'i' als Vergleichsmittelpunkt. Nun wird überprüft, welche Punkte innerhalb des Radius unseres Vergleichsobjektes liegen (radii_temp). Anschließend findet eine Überprüfung aller nachfolgendem Mittelpunkte in der Entfernungsmatrix (distance_matrix) statt, deren Distanz zwischen den Mittelpunkten liegen und es werden die Elemente bearbeitet, welche eine kleinere Entfernung aufweisen als der Radius des Vergleichsobjektes. Wenn die Entfernung zwischen Vergleichsobjekt und nachfolgedem Mittelpunkt kleiner ist als der Radius des Vergleichsobjektes, wird dieser in die Liste hinzugefügt (Zeile 43 in Abbildung 3.2.1). Circle_out enthält am Ende des Durchlaufes alle Kreise, welche entfernt werden sollen. Um die Anzahl der verbliebenden Kreise zu ermitteln, wird in Zeile 46 in Abbildung 3.2.1 nun die Größe (bzw. Länge) der Listen subtrahiert.

```
37  x = np.array([cx, cy]).transpose()
38  distances = distance.pdist(x)
39  distance_matrix = distance.squareform(distances)
40  circle_out = set()
41  for i, radii_temp in enumerate(radii) :
42      for j in range(i+1, len(radii)) :
43          if distance_matrix [i, j] < radii_temp :
44              circle_out.add(j)
45
46  circle_count = len(radii) - len(circle_out)
```

Abbildung 3.2.1: Auswahl der Kreise

Im letzten Programmabschnitt geht es um die Darstellung der ausgewählten Kreise. In der Praxis wäre die Ausgabe einer einfachen Zahl ausreichen, um die Funktionsweise des Programmes jedoch besser darzulegen wurde für die Darstellung aller 10 Kreise im Bild plädiert. Hierzu wird in einer Schleife (siehe Zeile 63 bis 71 in Abbildung 3.2.2) für jeden Kreis eine Darstellung im Bild durchgeführt. Da der enumerate Operator in Zeile 63 nur einen Wert für die Schleife akzeptiert, erfolgt mit dem 'zip' Befehl die Komprimierung der Werte für den Mittelpunkt und des Radius des entsprechenden Kreises in eine Liste. Anschließend wird in Zeile 64 der Wert für den Mittelpunkt (center_y und center_x) sowie der Radius des zu zeichnenden Kreises gespeichert. Durch den Aufruf der circle_perimeter Funktion aus der skimage.draw Biblitothek ergibt sich nun die Möglichkeit, die entsprechenden Pixelwerte zu speichern, welche für die Darstellung des Kreises umgefärbt werden müssen. Dies wird in den Variablen circy und circx in Zeile 65 in Abbildung 3.2.2 vollzogen. Im vorletzen Abschnitt wird nun der Kreis rot gezeichnet, wenn er nicht ein Teil der Set-Liste der auszuschließenden Kreise ist (Zeile 68 bis 71 in Abbildung 3.2.2).

```
61  fig, ax = plt.subplots(ncols=1, nrows=1, figsize=(10, 4))
62  image = color.gray2rgb(image)
63  for i, kreis in enumerate(zip(cy, cx, radii)) :
64      center_y, center_x, radius = kreis
65      circy, circx = circle_perimeter(center_y, center_x, radius,
66                                      shape=image.shape)
67
68      if i not in circle_out :
69          image[circy, circx] = (220, 20, 20)
70      else :
71          image[circy, circx] = (0,0,0)
72
73
74  ax.set_title('Die Anzahl der Achsen beträgt: ' + str(circle_count))
75  ax.imshow(image)
76  plt.show()
```

Abbildung 3.2.2: Darstellung der Kreise

Im letzten Abschnitt wird nun die Anzahl der Achsen als String oberhalb der Bilddarstellung ausgegeben. Die fertige Ausgabe ist in Abbildung 3.2.3 zu sehen [3].

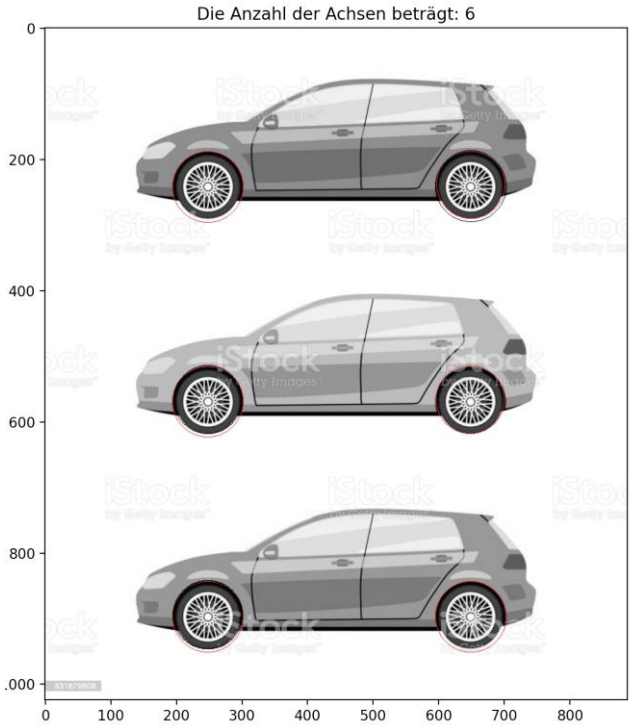

Abbildung 3.2.3: Ergebnisdarstellung

Zusammenfassung

Die Ausgangsidee befasste sich mit der Fragestellung, wie bereits implementierten Funktionen (in weiterführenden Bibliotheken) genutzt werden können, um vordefinierte Konturen in digitalen Fotografien zu lokalisieren. Speziell ging es um die Lokalisierung von Kreisen für die Zählung von Achsen bei KFZ-Fahrzeugen.
Nach Abschluss der Arbeit kann somit geklärt werden, dass durch den Einsatz von Kantendetaktorfiltern wie dem Canny-Operator sowie der Weiterverarbeitung dieser Informationen in der Hough-Transformierung und der anschließenden Auswahl der Ergebnisse die Möglichkeit der Zählung von Achsen besteht. Weiterhin muss jedoch erwähnt werden, dass die Zählung nur für eine ausgewählte Anzahl an Kreisen in einem Bild funktioniert, welche bestimmte Kriterien erfüllen müssen. So ist es beispielsweise nicht möglich, mit den vorliegenden Parametereinstellungen, mehr als 10 Achsen in einem Bild zu ermitteln. Auch das Überschneiden von Konturen ist im aktuellen Algorithmus nicht messbar. Darüber hinaus erweist sich der vorliegende Programmcode jedoch als gute Basis für weiterführende Entwicklungen und Anwendungen.

REFERENCES

[1] M.O.Franz. 2007. Kanten und Konturen. Industrielle Bildverarbeitung. Script zur Vorlesung No. 6. Zugriff vom 28.12.2021 bis 30.12.2021. Link: *http://www-home.fh-konstanz.de/~mfranz/ibv_files/lect05_kanten.pdf*

[2] Prof. Dr. Sibylle Schwarz. 2017. HTWK Leipzig, Fakultät IMN. Digitale Bildverarbeitung. Zugriff vom 28.12.2021 bis 30.12.2021. Link: *https://www.imn.htwk-leipzig.de/~schwarz/lehre/ss17/dbv/dbv17-alles.pdf*

[3] Prof. Dr. Jürgen K. Singer. 2021. Fachbereich Automatisierung und Informatik. Digitale Bildverarbeitung. Zugriff vom 28.09.2021 bis 30.12.2021.

[4] Prof. Dr. Thomas Haenselmann. 2016. Hochschule Mittweide. Bildverarbeitung. Zugriff vom 29.12.2021 bis 30.12.2021. Link: *https://www.cb.hs-mittweida.de/fileadmin/verzeichnisfreigaben/haenselm/dokumente/dbv_2016ss_hough.pdf*

[5] Allam Shehata Hassanein, Sherin Mohammad, Mohamed Sameer and Mohammad Ehad Ragab. A Survey on Hough Transform, Theory, Techniques and Applications. Informatics Department, Electronics Research Institute, El-Dokki, Giza, 12622, Egypt.

[6] Mamta Joshi, Ashutosh Vyas. Comparison of Canny edge detector with Sobel and Prewitt edge detector using different image formats. ISSN: 2278-0181. International Journal of Engineering Research & Technology (IJERT)