Focus peaking - can edge detection improve the performance of current algorithms?

Thomas Schneider
Matrikel-Nr: 60482
Elektro- und Informationstechnik
Hochschule Karlsruhe

CONTENTS

I. Motivation/Einleitung

Wenn es um Bild- und Objekterkennung geht ist das menschliche Auge ungeschlagen. Innerhalb von wenigen Millisekunden erkennt es Kanten, klassifiziert Objekte, ist imstande diese zu benennen und erkennt den Unterschied zwischen einer Zeichnung, einem Bild und der Realität. Es gibt viele Algorithmen zur Kantenerkennung, jedoch ist keiner von ihnen so Leistungsstark und Effizient wie das menschliche Auge. Nicht nur ist der Canny Algorithmus der wohl bekannteste, sondern auch der am meist benutzte.
Der Canny-Algorithmus benutzt mehrere Stufen um in kurzer Zeit zu einem faszinierenden Ergebnis zu gelangen. Während ich meine Seminararbeit im Bereich der Kantenverfolgung geschrieben habe, kam ich nicht umhin auch erste Erfahrungen mit dem Canny-Algorithmus zu sammeln. Präzise und mit durchschnittlich TODO ZEITWERT CANNY NACHSCHLAGE x Sekunden für ein durchschnittliches, mit einer digitalkamera aufgenommenes Bild auch extrem schnell findet dieser Algorithmus alle Kanten des Bildes, markiert diese und gibt das so veränderte Bild zurück.
Neben all der Faszination stellte sich mir schnell die Frage, ob man den Algorithmus nicht durch Kantenverfolgung statt der reinen Kantenerkennung noch schneller machen könne. Mit dieser Frage beschäftige ich mich nachfolgend, werde einen Kantenverfolgungsalgorithmus entwickeln und evaluieren ob und zu was für einem eventuellen Preis man den Canny-Algorithmus so verschnellern kann.
Um den nachfolgenden Gedankengängen besser folgen zu können, schauen wir uns zuerst einmal vereinfacht an, mit welchen Schritten der Canny-Algorithmus zum Ziel kommt. Die genannten Methoden werden im wieteren Verlauf dieses Dokuments mathematisch näher beleuchtet.
Der Canny Algorithmus lässt sich in 5 Schritte unterteilen:

1) Entfernen von Rauschen Schöner FORMULIEREN
2) Suchen der Kanten im Bild
3) Ausdünnen der gefunden Kanten
4) Klassifizierung der Kantenpixel
5) Vervollständigung der Kante

Zuallererst wird vorhandenes Rauschen, welches unter anderem die Grauwerte der einzelnen Bildpixel verfälschen kann, im Bild entfernt. Dies ist ein Schritt, um welchen man auch bei Anwendung einer Kantenverfolgungsmethode nicht umhin kommt.
Anschließend wird für jedes Pixel ein Grauwert und der gradient berechnet. Dies geschieht für jedes einzelne Pixel des Bildes.
Die so gefundenen Kantenpixel wurden schon alle Kanten im Bild erkannt, allerdings sind diese noch sehr grob und unfein. Indem erneut alle Pixel des Bildes durchlaufen und die jeweiligen direkten Nachbarn in gradientenrichtung geprüft und die entsprechenden Grauwerte gegebenenfalls angepasst werden. So werden grobe, unfeine und breite Kanten ausgedünnt und feiner dargestellt.
Durch eine nochmalige Prüfung eines jeden Pixels werden die noch vorhandenen Kantenpixel klassifiziert und als starkes, schwaches oder kein Kantenpixel eingestuft, was das Kantenbild nochmal etwas verfeinert.
Abschließend werden eventuelle Lücken in Kanten durch ein Hystereseverfahren geschlossen, wofür ebenfalls alle Pixel des Bildes durchlaufen werden müssen.
Insgesamt durchlaufen wir das komplette Bild also mehrmals um einzelne Kantenpixel zu finden, diese zu klassifizieren und im Nachhinein die eventuell falsch klassifizierten oder gefundenen wieder zu korrigieren.
Genau hier greift meine Idee der involvierung einer Kantenverfolgung an. Wenn wir bereits im ersten Durchlauf ein Kantenpixel finden, können wir komplette Kante durch verfolgen der Kantenrichtung und Erkennung der anderen Pixel auf dem Weg erhalten. Dies würde die Anzahl der Zugriffe auf ein Pixel reduzieren und damit die Geschwindigkeit des Algorithmus erhöhen.
Gerade auf dem Gebiet des Auto-Focus und der Hochgeschwindigkeitsfotografie ist eine Geschwindigkeitssteigerung um bereits wenige Millisekunden ein enormer Gewinn.

Im Folgenden erläutere ich die einzelnen Methoden des Canny Algorithmus mathematisch genauer, werde meine Ansätze sowie meinen Algorithmus präsentieren und ihn direkt mit dem Canny Algorithmus vergleichen um zu sehen ob und welche Performancesteigerungen es gibt.

II. Begriffsdefinitioen und Erklärungen

A. Was ist eine Kante

Eine Kante zeichnet sich dadurch aus, dass sich der Intensitätswert innerhalb weniger Pixel stark ändert. Hierbei muss allerdings nicht jeder Intensitätswertunterschied auch eine Kante sein, meist bewertet ein Algorithmus ob der Intensitätswertunterschied groß genug ist, als dass es sich um eine Kante handelt.

B. Pixelnachbarn

Unter Nachbarschaft versteht man in der Bildverarbeitung einen fest definierten Bereich um ein Pixel. Die zwei Grundkonzepte der Nachbarschaft sind die Vierer-Nachbarschaft (auch D-Nachbarschaft genannt) und die Achter-Nachbarschaft (*Fig. 1*).

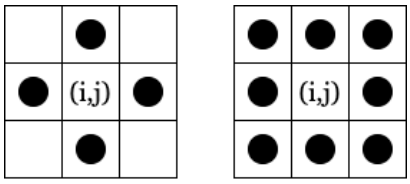

Fig. 1. *Vierer-Nachbarschaft (links) und Achter-Nachbarschaft (rechts)*

Wir benutzen im weiteren Verlauf stets die Achter-Nachbarschaft, da wir auch die diagonalen Nachbarn eines Pixels in die Kantenberechnung mit einbeziehen.

C. Bildrauschen

Bei Bildrauschen handelt es sich um zufällige Schwankungen, mit welchen die Informationen eines Pixels, beispielsweise Farbe oder Helligkeit, überlagert werden. Je mehr Pixel advon betroffen sind, desto höher ist das Rauschen im Bild.

1) Wie entsteht Rauschen in digitalen Bildern:

Bei Digitalkameras kann es aufgrund unterschiedlicher Ursachen zu Rauschen kommen.

a) Quantenrauschen:

Die riesige Lichtmenge, welche auf einen elektronischen Sensor fällt besteht aus vielen Elemtarteilchen, den sogenannten Photonen. Obwohl ein Lichtstrom für das menschliche Auge gleichmäßig aussieht, treffen die Photonen zufällig auf die einzelnen Pixel des Sensors. Wenn man eine große Sensorfläche über eine lange Zeit hinweg observiert wird man eine gleichmäßige Verteilung der Photonen feststellen. Observiert man hingegen einen kleinen Sensor für nur einen kurzen Zeitabschnitt wird man feststellen, dass die einzelnen Pixel des Sensors unterschiedlich viele Photonen empfangen haben. Nimmt man ein stark belichtetes, helles Bild auf, so treffen - da es viel Licht gibt - viele Photonen auf den Sensor. Dies führt dazu, dass man die zufällige Verteilung, das Rauschen, im fertigen Bild weniger stark wahrnimmt.
Bei dunklen Bildern hingegen treffen weniger Photonen auf den Sensor wodurch die zufällige Verteilung auf die Pixel extreme Abweichungen ergeben kann. Das Quantenrauschen ist bei dunkleren Bildern also stärker als bei hellen.

b) Beuteileigenschaften:

Die auf dem Sensor angekommenen Photonen werden von einem Verstärker in Elektronen umgewandelt. Mit kleinen Kondensatoren wird die Ladung auf dem Chip gesammelt und in eine Spannung umgesetzt. Vor jeder neuen Bilderfassung müssen die Kondensatoren also "entleert" werden wobei es zur sogenannten *Reset-Noise* kommt. Das entleeren geschieht mitnichten ideal oder gleichmäßig, wodurch es zu Restspannungen und damit einer zufälligen Schwankung um Ergebnis kommt. Diese Schwankung macht sich im Bild als Rauschen bemerkbar.
Weiterhin muss in vielen Schaltungen ein Ruhestrom fließen, welche zusätzlich für ein Rauschen im Bild sorgt.

c) Thermisches Rauschen:

Eletrkonen bewegen sich bei erhöhter Temperatur mehr. So kann es bereits bei Raumtemperaturen zu Elektronenbewegungen und damit thermischem Rauschen kommen.

d) Quantisierungsrauschen:

Um die analogen Spannungswerte in digitale Daten umzuwandeln wird ein Analog-Digital-Wandler angewendet. Die von einem AD-Wandler erzeugten Signalfehler sind allerdings nicht zufällig. Da bei unserem Fall bereits das Eingssignal zufälligkeiten Aufweist, sind auch diese Signalfehler nichtmehr zuverlässig vorhersagbar. Diese resultierende Schwankung in den Spannungswerten wird *Quantisierungsrauschen* genannt. Die heutige Technik ist so weit entwickelt, dass Quantisierungsrauschen keinen Nennenswerten Beitrag zum Bildrauschen mehr leistet.

2) Methoden zur Rauschunterdrückung:

Es gibt eine Vielzahl von Methoden um das Bildrauschen sowohl bereits vor der Entstehung des Bildes zu mindern als auch im Nachgang durch Bearbeitung des Bildes zu entfernen. In dieser Arbeit beschränken wir uns auf die Methode der örtlichen Faltung mit einer 3×3 Filtermaske, genauer noch auf einen Gauß-Filter mit einer 3×3 Kern.

3) Gauß Filter:

Der Gauß Filter ist ein linearer Filter, welcher in der Bildverarbeitung zur Glättung des Bildes und Verminderung von Rauschen, vor allem weißem, verwendet wird. Feinere Strukturen des Bildes gehen hierbei verloren, wobei gröbere erhalten bleiben. Ein Gaußscher Filterkern der Größe (2k + 1)×(2k + 1) kann mit

$$H_{ij} = \frac{1}{2\pi\sigma^2} * e^{-\frac{(i-(k+1))^2+(j-(k+1))^2}{2\sigma^2}}$$

berechnet werden.

III. Canny-Algorithmus

A. Gauß Kern

Für die Faltung benutzen wir einen oben bereits beschriebenen Gauß Kern. Als Programmiersprache benutzen wir Python und realisieren die Berechnung des Gauß Kerns folgendermaßen:

```
def g_kern(mask, sigm=1.8):
mask = int(mask) // 2
x, y = np.mgrid[-mask:mask + 1,
-mask:mask + 1]
norm = 1 / (2 * np.pi * sigm ** 2)
kern = np.exp(-((x ** 2 + y ** 2) /
(2 * sigm ** 2))) * norm

return kern
```

B. Sobel Operator

Der Sobel Operator besteht aus zwei 3×3 Faltungskernen, wobei ein Kern dem jeweils anderen um 90° gedreht entspricht.

-1	0	+1
-2	0	+2
-1	0	+1
	G_x	

+1	+2	+1
0	0	0
-1	-2	-1
	G_y	

Fig. 2. *Links der Faltungskern für die X- und rechts für die Y-Richtung.*

Für jedes Pixel werden die Komponenten der Matrix aufsummiert um den Grauwert zu erhalten.

In Python realisieren wir den Sobel Filter folgendermaßen:

```
def sobel(img):
  Gx = np.array([[-1, 0, 1], [-2, 0, 2],
  [-1, 0, 1]], np.float32)
  Gy = np.array([[1, 2, 1], [0, 0, 0],
  [-1, -2, -1]], np.float32)

  ablx = ndimage.filters.convolve(img, Gx)
  .astype(float)
  ably = ndimage.filters.convolve(img, Gy)
  .astype(float)

  res = np.hypot(ablx, ably)
  res = res / res.max() * 255
  gradient = np.arctan(ably, ablx)

  return res, gradient
```

C. Non Maximum Suppression

Die Non Maximum Suppression Technik wird angewendet um bereits gefundene Intensitätsmaxima (Kanten) erneut zu prüfen und diese auszudünnen. Hierfür durchläuft der Algorithmus jedes gefundene Kantenpixel und vergleicht, basierend auf dem Gradienten des jeweiligen Pixels, die entsprechenden Nachbarpixel. Sollte eines der Nachbarpixel einen höheren Grauwert als das aktuelle Pixel aufweisen, wird der Grauwert des aktuellen Pixels auf 0 gesetzt.

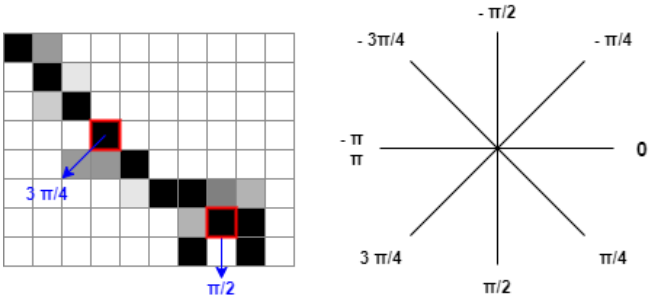

Fig. 3. *Darstellung der Non Maximum Suppression Technik*

Schauen wir uns nun das untere, rot umrahmte Pixel etwas genauer an. Die Richtung der Kante wird hier durch den blauen Pfeil symbolisiert und entspricht einem Winkel von $\frac{\pi}{2}$ (90°).

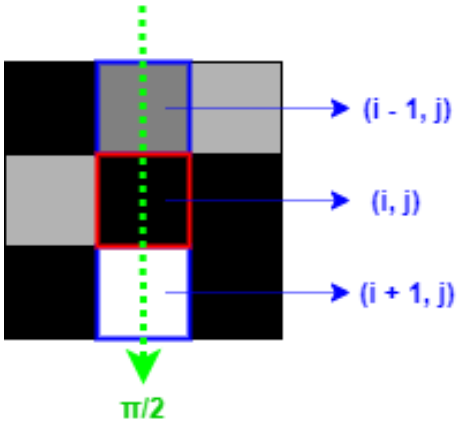

Fig. 4. *Nähere Betrachtung eines einzigen Pixels*

Die Richtung der Kante wird durch den grünen Pfeil dargestellt und verläuft vertikal von oben nach unten. Der Algorithmus prüft nun, ob die Pixel ober- und unterhalb (im Bild blau markiert) des ausgewählten Pixels (im Bild rot) einen höheren Intensitätswert aufweisen. In unserem Beispiel ist der Intensitätswert des unteren Pixels (i + 1, j) höher als der der anderen Beiden (das untere Pixel ist weiß und hat damit einen Intensitätswert von 255). Der Intensitätswert des aktuellen Pixels wird nun auf 0 gesetzt. Hätte keines der beiden anderen Pixel einen höheren Intensitätswert, würde der aktuelle Wert beibehalten werden.

Jedes Pixel hat also 2 Hauptkriterien, die Gradientenrichtung und den Intensitätswert. Die Non Maximum Suppression nutzt diese beiden Merkmale und führt folgende Schritte durch:

- Ein mit Nullen initialiiertes Abbild der Gradientenmatrix erstellen.
- Kantenrichtung anhand der Gradientenmatrix erkennen.
- Prüfen ob Pixel entlang der Kantenrichtung einen höheren Intensitätswert als das aktuelle Pixel haben.
- Das mit dem Non Maximum Suppression Algorithmus bearbeitete Bild zurückgeben.

Den Non-Maximum-Suppression Algorithmus haben wir in Python folgendermaßen implementiert:

```
def non_max_suppression(img, grad):
  y, x = img.shape
  blackscreen = np.zeros(img.shape)
  angle = grad * 180. / np.pi
  angle[angle < 0] += 180

  for i in range(1, y - 1):
    for j in range(1, x - 1):
      if img[i, j] != 0:
        try:
          q = 255
          r = 255

        # angle 0
        if (0 <= angle[i, j] < 22.5) or
        (157.5 <= angle[i, j] <= 180):
          q = img[i, j + 1]
          r = img[i, j - 1]

        # angle 45
```

```
            elif 22.5 <= angle[i, j] < 67.5:
              q = img[i + 1, j - 1]
              r = img[i - 1, j + 1]

            # angle 90
            elif 67.5 <= angle[i, j] < 112.5:
              q = img[i + 1, j]
              r = img[i - 1, j]

            # angle 135
            elif 112.5 <= angle[i, j] < 157.5:
              q = img[i - 1, j - 1]
              r = img[i + 1, j + 1]

          if (img[i, j] >= q) and (img[i, j] >= r):
            blackscreen[i, j] = img[i, j]
          else:
            blackscreen[i, j] = 0

          except IndexError as e:
          pass

          return blackscreen
```

D. Double threshold

Der double threshold Filter unterteilt unser Bild in 3 Arten von Pixeln

1) **Starke Pixel**
   - Pixel, deren Intensitätswert hoch genug ist, dass wir uns sicher sein können, dass sie ein Teil der finalen Kante sind.
2) **Schwache Pixel**
   - Pixel, deren Intensitätswert nicht hoch genug ist um als starkes Pixel eingestuft zu werden, allerdings hoch genug ist um nicht als unsignifikant für die Kantenerkennung zu sein.
3) **Andere Pixel**
   - Alle Pixel, welche keine der anderen beiden Bedingungen erfüllen.

Es wird eine Obergrenze (high threshold) und eine Untergrenze (low threshold) für die Intensitätswerte festgelegt. Ist der Intensitätswert eines Pixels höher als oder gleich der Obergrenze, wird es als starkes Pixel markiert, der Intensitätswert also auf 255 gesetzt. Befindet sich der Intensitätswert des Pixels zwischen Ober- und Untergrenze, wird es als schwaches Pixel markiert. Sollte der Intensitätswert des Pixels kleiner als oder gleich der Untergrenze sein, wird er auf 0 gesetzt.
Die beiden Schwellenwerte werden abhängig vom Eingangsbild berechnet, in unserem Beispiel durch den Faktor 0.05 für die untere und 0.09 für die obere Schwelle. In Python setzen wir die double threshold Funktion folgendermaßen um:

```
        def double_threshold(img,
        unterer_faktor=0.05, oberer_faktor=0.09):
          obere_schwelle = img.max() * oberer_faktor
          untere_schwelle = obere_schwelle *
                            unterer_faktor
          blackscreen = np.zeros(img.shape)

          starkes_pixel = 255
          schwaches_pixel = 25
          keine_kante = 0

          starkes_pixel_i, starkes_pixel_j =
          np.where(img >= obere_schwelle)

          keine_kante_i, keine_kante_j =
          np.where(img <= untere_schwelle)

          schwaches_pixel_i, schwaches_pixel_j =
          np.where((img < obere_schwelle) &
          (img > untere_schwelle))

          blackscreen[starkes_pixel_i,
          starkes_pixel_j] = starkes_pixel

          blackscreen[schwaches_pixel_i,
          schwaches_pixel_j] = schwaches_pixel

          blackscreen[keine_kante_i,
          keine_kante_j] =
          keine_kante

          return blackscreen
```

E. Kantenverfolgung durch Hysterese

Durch eine Hysterese wird festgelegt, ab welcher Kantenstärke ein Pixel zu einer Kante gehört. Mithilfe zweier Schwellwerte $T_1 < T_2$ wird jedes Pixel eines Bildes überprüft. Sobald ein Pixel einen Intensitätswert über T_2 hat, wird diesem Pixel gefolgt und jedes Pixel entlang der so gefundenen Kante, dessen Intensitätswert größer T_1 ist, als Element dieser Kante markiert.

Wir realisieren unsere Hysterese Funktion in Python folgendermaßen:

```
def hysteresis(img, weak, strong=255):
  M, N = img.shape     # c10 * 1
  for i in range(1, M - 1):    # c14 * (M - 2)
    for j in range(1, N - 1):
    # c15 * ((M - 2) * (N - 2))
      if img[i, j] == weak:
      # c31 * ((M - 2) * (N - 2))
        try:
        # c16 * (((M - 2) * (N - 2)) - Z)
          if ((img[i + 1, j - 1] == strong) or
          (img[i + 1, j] == strong) or
          (img[i + 1, j + 1] == strong) or
          (img[i, j - 1] == strong) or
          (img[i, j + 1] == strong) or
          (img[i - 1, j - 1] == strong) or
          (img[i - 1, j] == strong) or
          (img[i - 1, j + 1] == strong)):
          # c32 * (((M - 2) * (N - 2)) - Z)
            img[i, j] = strong
            # c33 * (((M - 2) *
            # (N - 2)) - Z - S)
          else:
            img[i, j] = 0
            # c23 * (((M - 2) *
            # (N - 2)) - Z - T)
        except IndexError as e:
        # c24 * 0
          pass     # c25 * 0
  return img
```

IV. Laufzeiteffizienz eines Algorithmus

Nun wollen wir die Laufzeit des genannten Algorithmus betrachten. Hierzu zählen wir für eine gegebene Eingabe alle Anweisungen x mit einer Zeit t_x, welche von der Art von x abhängig ist. Folgende Annahmen setzen wir voraus:

- Das Ausführen einer Zeile Code benötigt einen konstanten Zeitaufwand
- Verschiedene Zeilen (verschiedene Operationen) benötigen einen unterschiedlichen Zeitaufwand

Der benötigte, konstante Zeitaufwand, welchen eine Zeile z also benötigt, wird mit c_x bezeichnet. Im vorherigen Abschnitt ist in den Kommentaren des Python Codes bereits der jeweilige Zeitaufwand jeder Zeile vermerkt. Im Folgenden werden die Laufzeiten der einzelnen Funktionen sowie im Abschluss des gesamten Canny Algorithmus berechnet.
Zusätzlich betrachten wir - sofern möglich - den jeweils schlechtesten und besten Fall.

A. Gauß Filter

Die Kosten für den Gauß Filter ergibt sich der Laufzeitaufwand hiermit zu:

$$T(n)_{Gauß} = c_1 + c_2 + c_3 + c_4$$

Da wir hier jede Zeile einmal durchlaufen müssen ist der schlechteste Fall gleich dem besten Fall.

B. Sobel Filter

Der Sobel Filter hat ähnlich zum Gauß Filter einen geringen Laufzeitaufwand im Vergleich zum Gesamtaufwand:

$$T(n)_{Sobel} = (c_5 + c_6) * 2 + c_7 + c_8 + c_9$$

Der Sobel Filter ist nur vom jeweiligen Eingangsbild abhängig, jede Zeile muss einmal durchlaufen werden. Je größer das Bild, desto größer auch der Rechenaufwand. Auch hier entspricht der schlechteste Fall gleich dem besten Fall.

C. Non Maximum Supression

Den größten Rechenaufwand beansprucht die Non Maximum Suppression. Hier ergibt sich der benötigte Laufzeitaufwand zu:

$$\begin{aligned} T(n)_{nms} = {} & c_{10} + c_{11} + c_{12} + c_{13} + c_{14} * (M - 2) + \\ & (c_{15} + c_{16} + 2 * c_{17} + c_{18} + 3 * c_{20} + c_{21} + \\ & c_{23}) * ((M - 2) * (N - 2)) + \\ & (2 * c_{19} - 9 * c_{20}) * A0 + \\ & (2 * c_{19} - 6 * c_{20}) * A45 + \\ & (2 * c_{19} - 3 * c_{20}) * A90 + \\ & c_{19} * 2 * A135 + \\ & (c_{22} - c_{23}) * Y \end{aligned}$$

D. Double threshold

Die double Threshold Methode beansprucht folgenden Laufzeitaufwand:

$$\begin{aligned} T(n)_{\text{double threshold}} = {} & c_{10} + c_{11} + 3 * c_{17} + \\ & c_{26} + c_{27} + 2 * c_{28} + c_{29} + 3 * c_{30} \end{aligned}$$

E. Hysterese

Letztendlich noch der Laufzeitaufwand der Hysterese mit:

$$\begin{aligned} T(n)_{hysterese} = {} & c_{10} + c_{14} * (M - 2) + \\ & (c_{15} + c_{16} + c_{23} + c_{31} + c_{32} + c_{33}) * ((M - 2) * \\ & (N - 2)) - \\ & Z * (c_{15} + c_{23} + c_{32} + c_{33}) - \\ & S * c_{33} - T * c_{23} \end{aligned}$$

F. Komplette Laufzeit

Die komplette Laufzeit $T(n)_{Canny}$ des Algorithmus ergibt sich somit zu:

$$
\begin{aligned}
T(n)_{Canny} = \\
&c_1 + c_2 + c_3 + c_4 + c_5 * 2 + c_6 * 2 + c_7 + \\
&c_8 + c_9 + c_{10} * 3 + c_{11} * 2 + c_{12} + c_{13} + \\
&c_{14} * 2 * (M - 2) + c_{15} * (2 * ((M - 2) * (N - 2)) \\
&- Z) + \\
&c_{16} * 2 * ((M - 2) * (N - 2)) + c_{17} * 6 * ((M - 2) \\
&* (N - 2)) + \\
&c_{18} * ((M - 2) * (N - 2)) + \\
&c_{19} * 2 * (A0 + A45 + A90 + A135) + \\
&c_{20} * (-9 * A0 - 6 * A45 - 3 * A90 + 3 * ((M - 2) \\
&* (N - 2))) + \\
&c_{21} * ((M - 2) * (N - 2)) + c_{22} * Y + \\
&c_{23} * (2 * ((M - 2) * (N - 2)) - T - Y - Z) + \\
&c_{26} + c_{27} + c_{28} * 2 + c_{29} + c_{30} * 3 + \\
&c_{31} * ((M - 2) * (N - 2)) + c_{32} * (((M - 2) * (N - \\
&2)) - Z) + \\
&c_{33} * (((M - 2) * (N - 2)) - Z - S)
\end{aligned}
$$

Mit

$$
\begin{aligned}
&A0 + A45 + A90 + A135 = ((M - 2) \\
&* (N - 2)); \\
&Z - T = S; \\
&Z - S = T; \\
&((M - 2) * (N - 2)) \mathrel{\widehat{=}} L;
\end{aligned}
$$

kann man die Gleichung folgendermaßen vereinfachen:

$$
\begin{aligned}
T(n)_{Canny} = \\
&c_1 + c_2 + c_3 + c_4 + c_5 * 2 + c_6 * 2 + c_7 + \\
&c_8 + c_9 + c_{10} * 3 + c_{11} * 2 + c_{12} + c_{13} + \\
&c_{14} * 2 * (M - 2) + c_{15} * (2 * L - Z) + \\
&c_{16} * 2 * L + c_{17} * 6 * L + c_{18} * L + \\
&c_{19} * 2 * L + c_{20} * (-9 * A0 - 6 * A45 - 3 * A90 \\
&+ 3 * L) + \\
&c_{21} * L + c_{22} * Y + c_{23} * (2 * L - Y - S) + \\
&c_{26} + c_{27} + c_{28} * 2 + c_{29} + c_{30} * 3 + \\
&c_{31} * L + c_{32} * L - Z) + c_{33} * (L - T)
\end{aligned}
$$

L hängt hierbei von der Größe des Eingangsbildes ab. Je größer das Eingangsbild, desto größer auch L und desto mehr Laufzeitaufwand ist nötig.

1) Laufzeit im besten Fall:

Im besten Fall (in welchem es trotzdem noch eine Kante im Bild gibt) kann man folgende Annahmen treffen:

$$
\begin{aligned}
&A45 = A90 = A135 = S = 0; \\
&A0 = Y = L; \\
&Z + T = L;
\end{aligned}
$$

$$
\begin{aligned}
T(n)_{Canny} = \\
&c_1 + c_2 + c_3 + c_4 + c_5 * 2 + c_6 * 2 + c_7 + \\
&c_8 + c_9 + c_{10} * 3 + c_{11} * 2 + c_{12} + c_{13} + \\
&c_{14} * 2 * (M - 2) + c_{15} * (2 * L - Z) + \\
&c_{16} * 2 * L + c_{17} * 6 * L + c_{18} * L + \\
&c_{19} * 2 * L + \\
&c_{21} * L + c_{22} * Y + \\
&c_{26} + c_{27} + c_{28} * 2 + c_{29} + c_{30} * 3 + \\
&c_{31} * L + c_{32} * (L - Z) + c_{33} * Z
\end{aligned}
$$

2) Lauftzeit im schlimmsten Fall:

Analog können für den schlimmsten Fall folgende Annahmen getroffen werden:

$$
\begin{aligned}
&A0 = A45 = A90 = Y = Z = T = 0; \\
&S = L;
\end{aligned}
$$

$$
\begin{aligned}
T(n)_{Canny} = \\
&c_1 + c_2 + c_3 + c_4 + c_5 * 2 + c_6 * 2 + c_7 + \\
&c_8 + c_9 + c_{10} * 3 + c_{11} * 2 + c_{12} + c_{13} + \\
&c_{14} * 2 * (M - 2) + c_{15} * 2 * L + c_{16} * 2 * L + \\
&c_{17} * 6 * L + c_{18} * L + c_{19} * 2 * L + \\
&c_{20} * 3 * L + c_{21} * L + c_{23} * L + c_{26} + \\
&c_{27} + c_{28} * 2 + c_{29} + c_{30} * 3 + c_{31} * L + \\
&c_{32} * L + c_{33} * L
\end{aligned}
$$

V. Kantenerkennung mit dem Canny Algorithmus

Im folgenden sind die oben genannten Schritte des Canny Algorithmus einzeln in Python realisiert und auf ein Bild angewendet.

A. Noise Reduction

Kantenerkennung ist sehr anfällig für Rauschen, da die meisten und ausschlaggebendsten mathematischen Operationen auf Ableitungen basieren. Deshalb muss eventuell vorhandenes Rauschen im ersten Schritt entfernt werden. Hierfür wir beim Canny Algorithmus das Bild mithilfe eines Gauß Filters geglättet. Mit einem Gaußschen Kernel (hier 5x5) wird der Intensitätswert an der Stelle (i,j) durch das gewichtete Mittel der ihn umgebenden Werte ersetzt. Der resultierende "blurring" Effekt hängt unmittelbar mit der Wahl der Kerngröße zusammen - je größer der Kern, desto besser ist auch der blurring Effekt. Mit steigender Kerngröße steigt jedoch auch die benötigte Rechenzeit, weshalb man hier nur einen 5x5 Kern nimmt, um bei einem ausreichend guten Ergebnis noch performant zu sein.

Hier auf das Logo der Hochschule Karlsruhe angewendet erkennt man im rechten Bild eine Unschärfe gegenüber dem linken Bild.

Fig. 5. *Links das original und rechts unter Anwendung des Gauß Filters.*

B. Gradient Calculation

In diesem Schritt wird sowohl die Intensität als auch die Richtung der Kanten durch die Berechnung des Gradienten ermittelt. Eine Kante wird durch eine merkliche Änderung der Intensität benachbarter Pixel deutlich. Um eine Kante zu erkennen ist es also am einfachsten, einen Filter anzuwenden, welcher die Änderung der Intensität in horizontaler wie vertikaler Richtung markiert.

Nach Glättung des Bildes werden nun also die Ableitungen in x (horizontaler) und y (vertikaler) Richtung berechnet. Am effizientesten kann man dies durch eine Faltung des Bildes mit einem Sobel Kern berechnen.

Die Intensität und Richtung berechnen sich also zu

$$|G| = \sqrt{I_x^2 + I_y^2}$$

$$\Theta(x,y) = \arctan(\frac{I_y}{I_x})$$

Bereits nach diesem Schritt hat man schon ein ziemlich gutes Ergebnis in welchem das Ursprungsbild durch Kanten hinreichend dargestellt ist.

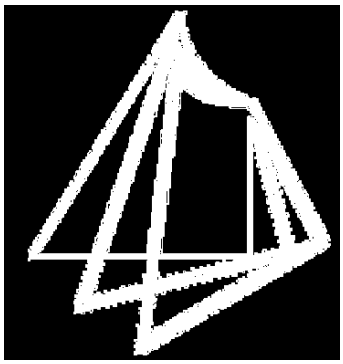

Fig. 6. *Bild nach Anwendung des Sobel Filters*

Man erkennt allerdings, dass die Kanten noch sehr unfein, grob und breit sind. Hier kommt der dritte Schritt ins Spiel, die Non-Maximum Suppression.

C. Non Maximum Suppression

Die momentan noch mehr als 1 Pixel breiten Kanten werden nun mit der sogenannten Non-Maximum Suppression Technik ausgedünnt. Hierbei wird jedes Pixel durchlaufen. Abhängig vom Gradienten, welcher uns die Richtung der Kante angibt, werden die Intensitätswerte der beiden Nachbarpixel des jeweiligen Pixels mit dem Intensitätswert des aktuellen Pixels verglichen. Ist einer der beiden Nachbarwerte größer, so wird der Grauwert des aktuellen Pixels auf Null gesetzt, andernfalls bleibt er unverändert. Nach einem erfolgreichen Durchlauf wurden alle Pixel entlang der Kante mit maximalen Intensitätswerten behalten.

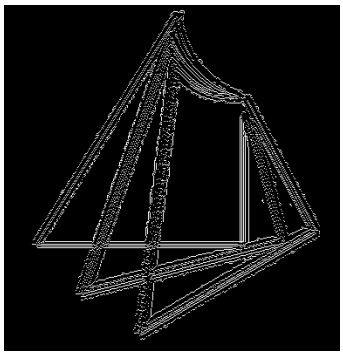

Fig. 7. *Bild nach Anwendung der Non Maximum Suppression*

Man kann eine deutliche Ausdünnung der Kanten erkennen, ebenso ist ersichtlich, dass die Intensitätswerte der gefundenen Pixel Kanten noch stark variieren. Mit den folgenden 2 Schritten versuchen wir das so gut als möglich zu kompensieren und die Intensitätswerte zu vereinheitlichen.

D. Double threshold

Durch die double threshold Funktion markieren wir nun jedes bis jetzt als Kante markiertes Pixel als starkes oder schwaches Pixel und erhalten ein Bild mit nurnoch 3 verschiedenen Intensitätswerten, 255 (starkes Pixel), 25 (schwaches Pixel) und 0 (kein Kantenpixel).

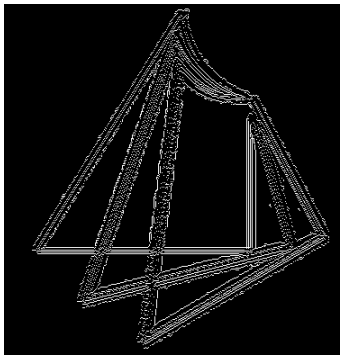

Fig. 8. *Bild mit starken (weiss) und schwachen (grau) Pixeln*

E. Hysteresis

Zuletzt werden noch einmal alle Pixel geprüft und falls einer der direkten Nachbarn ein starkes Pixel ist, wird das aktuelle

Pixel - falls es kein starkes Pixel ist - in ein starkes Pixel umgewandelt, indem der Intensitätswert auf 255 gesetzt wird. Wenn ein Pixel kein starkes Pixel in der Nachbarschaft hat, wird sein Intensitätswert zu 0 gesetzt.

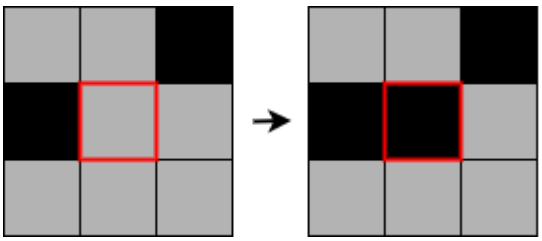

Fig. 9. *Ein Pixel ohne starkes Pixel in der Nachbarschaft*

Hat ein Pixel ein starkes Pixel in der Nachbarschaft, wird sein Intensitätswert auf 255 gesetzt, es selbst zu einem starken Pixel.

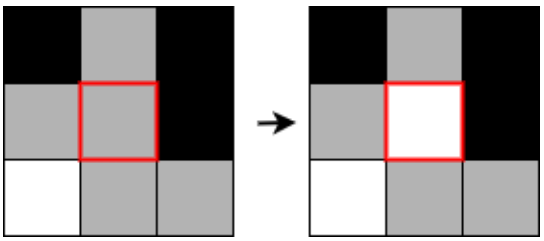

Fig. 10. *Ein Pixel mit einem starken Pixel in der Nachbarschaft*

Das Finale Bild mit den detektierten Kanten sieht nun folgendermaßen aus:

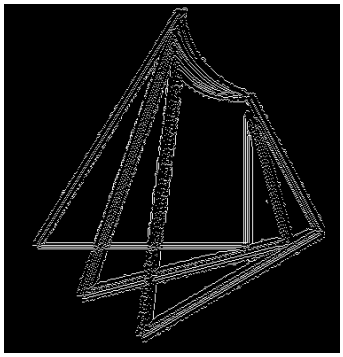

Fig. 11. *Finales Bild*

VI. Effizienzgewinn durch Kantenverfolgung

A. Idee

Sobald eine Kante gefunden wurde ist es nicht mehr notwendig, jedes einzelne Pixel des kompletten Bildes zu überprüfen. Es soll der Kante gefolgt werden und auf dem Weg die entsprechenden Nachbarpixel der Kante zu 0 setzen und als "bearbeitet" markieren. So muss man nicht mehr jedes Pixel des Bildes untersuchen sondern erheblich weniger.

Sobald ein Kantenpixel gefunden wurde, werden - ähnlich zum Canny Algorithmus - die Nachbarn überprüft und der Gradient gebildet. So findet man das nächste Pixel der Kante. Das aktuelle Pixel erhält den Grauwert 255, den 7 übrigen Nachbarn wird der Intensitätswert 0 zugewiesen und der Algorithmus navigiert zum nächsten Pixel.

Dadurch, dass man - sobald eine Kante detektiert wurde - 7 Nachbarpixel auf 0 setzt und diese nicht erneut überprüfen muss ergibt sich ein nicht unerheblicher performance Gewinn mindestens des Faktors 7.

Dadurch, dass ein Bild im Schnitt nur zwischen 5% und 10% aus Kanten beseht, müssen diese zur Realisierung der Idee erst einmal detektiert werden. Hierzu gab es folgende Ansätze:
Man durchläuft beide Diagonalen des Bildes und folgt allen Kanten, welche man dort gefunden hat. Die Wahrscheinlichkeit, dass man durch diese Methode eine Kante findet, ist sehr hoch. Natürlich gibt es auch Bilder, welche keine Kanten auf den Diagonalen haben. Um hier nicht unnötige Laufzeit zu verschwenden, durchläuft man alle horizontalen, beginnend bei der ersten und dann immer im Abstand von 5 Pixeln. So durchläuft man nicht das komplette Bild und sollte definitiv eine Kante finden.

Zur Berechnung der nächsten Richtung diente mir anfangs folgender Code:

```
def direction(pos, img):
  direction = np.zeros((3, 3))
  direction[0][2] = abs(diagonal_top_right
            (pos, img) - diagonal_bottom_left
            (pos, img))
  direction[0][1] = abs(top(pos, img) -
           bottom(pos, img))
  direction[0][0] = abs(diagonal_top_left
          (pos, img) - diagonal_bottom_right
          (pos, img))
  direction[1][0] = abs(left(pos, img) -
           right(pos, img))
  direction[2][0] = value_raising_diagonal
           (pos, img)
  direction[2][1] = value_top_bottom
           (pos, img)
  direction[2][2] = value_falling_diagonal
           (pos, img)
  direction[1][2] = value_left_right
           (pos, img)

  return direction
```

Die Richtung (der Gradient) wurde dabei nach folgendem Algorithmus berechnet:

```
def next_direction_to_move(current_position,
                y, direct):
  if (direct[0] == 0) and (not all
  (k in forbidden_directions for k in
  (1, 2, 3))):  # check for move up
    if (current_position[0] != 0) and
    (1 not in forbidden_directions):
    # check for top border
      moving_direction = 1
    elif (current_position[0] != 0) and
    (2 not in forbidden_directions):
    # check for top border
      moving_direction = 2
    elif (current_position[0] != 0) and
    (3 not in forbidden_directions):
    # check for top border
      moving_direction = 3
    elif (current_position[1] != 0) and
    (4 not in forbidden_directions):
    # move left cause we are at top border
      forbidden_directions.extend((1, 2, 3))
    moving_direction = 4
      elif 7 not in forbidden_directions:
      # move down if left border
    moving_direction = 7
      forbidden_directions.
      extend((1, 2, 3, 4))
    else:
      print("no case for moving_direction
      left")
      moving_direction = 9
  elif (direct[0] == 1) and (not all(k in
  forbidden_directions for k in (4, 5))):
  # check for left  or right
    if (direct[1] == 0) and
    (current_position[1] != 0) and
    (4 not in forbidden_directions):
    # move right if left border
      moving_direction = 4
    elif 5 not in forbidden_directions:
      moving_direction = 5
      forbidden_directions.append(4)
    else:
      print("no case for moving_direction
      left")
      moving_direction = 9
  else:  # move down cause no
  case matches until now
    if (current_position[0] < y) and
    (6 not in forbidden_directions):
    # check for bottom border
      moving_direction = 6
    elif (current_position[0] < y) and
    (7 not in forbidden_directions):
    # check for bottom border
      moving_direction = 7
    elif (current_position[0] < y) and
    (8 not in forbidden_directions):
    # check for bottom border
      moving_direction = 8
    elif (current_position[1] != 0) and
    (4 not in forbidden_directions):
    # move left if bottom border
      moving_direction = 4
      forbidden_directions.
      extend((6, 7, 8))
    elif 5 not in forbidden_directions:
    # move right when also at left border
      moving_direction = 5
      forbidden_directions.
      extend((4, 6, 7, 8))
    else:
      print("no case for moving_direction
      left")
      moving_direction = 9

  return moving_direction
```

Ebenso war der Code zur Bestimmung des nächsten Pixels auf eine sehr triviale Weise vorhanden:

```
def position_of_next_pixel(moving_direction,
            current_position):
  if moving_direction == 1:
    next_pixel_position =
    (current_position[0] - 1,
     current_position[1] - 1)
  elif moving_direction == 2:
    next_pixel_position =
    (current_position[0] - 1,
    current_position[1])
  elif moving_direction == 3:
    next_pixel_position =
    (current_position[0] - 1,
    current_position[1] + 1)
  elif moving_direction == 4:
    next_pixel_position =
    (current_position[0],
    current_position[1] - 1)
  elif moving_direction == 5:
    next_pixel_position =
    (current_position[0],
    current_position[1] + 1)
  elif moving_direction == 6:
    next_pixel_position =
    (current_position[0] + 1,
    current_position[1] - 1)
  elif moving_direction == 7:
    next_pixel_position =
    (current_position[0] + 1,
    current_position[1])
  elif moving_direction == 8:
    next_pixel_position =
    (current_position[0] + 1,
    current_position[1] + 1)
  elif moving_direction == 9:
    print("no_next_pixel_position")
    next_pixel_position =
    "moving in circles"

  return next_pixel_position
```

Hierbei traf ich auf das Problem, dass die Einsparung der Laufzeit durch direktes bearbeiten der Kantennachbarn sehr schnell wieder verloren geht und zwar dadurch, dass man mehrere Matrizen der Größe des Bildes benötigt um die einzelnen Stati wie neuer Grauwert, Richtung und ein *bereits bearbeitet* Flag zu speichern. Diese muss man immer wieder abrufen, also die Matrizen laden, bearbeiten und verändert wieder abspeichern. Gerade bei größeren Bildern oder wenn eine Kante nicht mit dem Durchlaufen der Diagonalen gefunden wird, übersteigt die Laufzeit des geplanten Algorithmus die des Canny Algorithmus schnell.
Ebenso ist mit dem beschriebenen Vorgehen nicht sichergestellt, dass man alle Kanten findet. Um auch Objekte zum Beispiel am Rand zu finden müsste man das komplette Bild erneut mit einem Hysterese Algorithmus bearbeiten.

Trotz anfänglicher, guter Fortschritte hatte der geplante Algorithmus deshalb leider keine Aussicht auf Erfolg und eine Effizienzsteigerung des Canny Algorithmus durch Kantenverfolgung ist mit dieser Methode nicht möglich.

VII. Laufzeitoptimierung des Canny Algorithmus

Ein schwarz-weiß Bild hat einige schwarze Pixel, also Pixel mit dem Intensitätswert 0. Der obige Canny prüft jedes Pixel viele Male. Einige dieser Cases kann man jedoch auslassen, da sie nicht geprüft werden müssen. So sind zum Beispiel Pixel mit einem Intensitätswert von 0 kein Teil einer Kante und man kann die Prüfung dieses Pixels überspringen.

VIII. Literaturverzeichnis

- *http://www9.in.tum.de/seminare/hs.SS06.EAMA/material/01_ausarbeitung.pdf*
- *https://www.kuppelwieser.net/index.php/technik/15-bildverarbeitung/40-canny-algorithmus*
- *http://mi.informatik.uni-siegen.de/teaching/lectures/EI/script/10eiComplexity.pdf*
- *https://edoc.sub.uni-hamburg.de/haw/volltexte/2016/3266/pdf/BA_Tamou.pdf*
- *https://en.wikipedia.org/wiki/Canny_edge_detector*