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*).

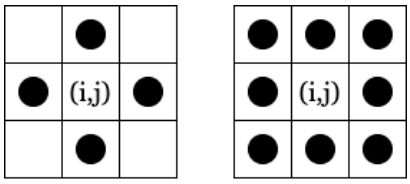

Fig. 1. *Vierer-Nachbarschaft (links) und Achter-Nachbarschaft (rechts)*

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.

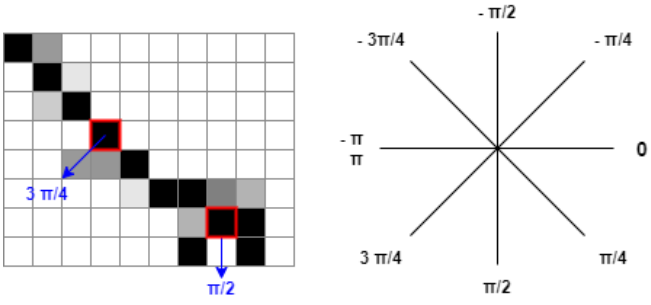

Fig. 3. *Darstellung der Non Maximum Suppression Technik*

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°).

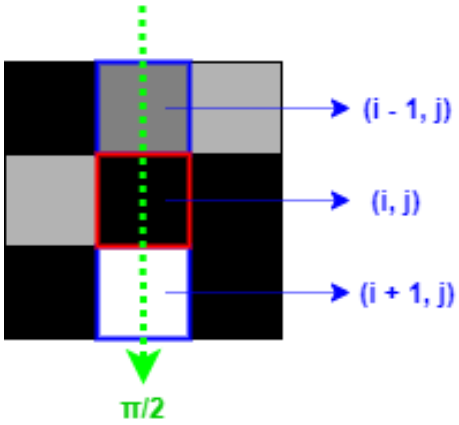

Fig. 4. *Nähere Betrachtung eines einzigen Pixels*

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. Kantenvervollständigung durch Hysterese

TITEL ÄNDERN Durch eine Hysterese wird festgelegt, ab welcher Kantenstärke ein Pixel zu einer Kante gehört. Gemäß unserem obigen Codebeispiel setzen wir die obere Schwelle T_2 auf 255 und die untere Schwelle T_1 auf 25. Anhand dieser zwei Schwellwerte ($T_1 < T_2$) wird jedes Pixel des Bildes überprüft. Falls ein Pixel ein *schwaches Pixel* ist, also einen Intensitätswert von 25 aufweist, werden all seine Nachbarn geprüft. Sollte einer der Nachbarpixel ein starkes Pixel sein (oder, im allgemeinen, einen noch höheren Intensitätswert als T_2 aufweisen), so wird auch das aktuell schwache Pixel als starkes Pixel markiert. Andernfalls wird der Intensitätswert des Pixels auf 0 gesetzt, da es nicht zur Kante gehört. Hierdurch werden alle kanten vervollständigt und alleinstehende, nur eventuell zu einer Kante gehörende, Pixel eliminiert. Die beschriebene Hysteresefunktion realisieren wir durch folgenden Python Code:

```
        def hysterese(img, schwaches_pixel,
        starkes_pixel=255):
          y, x = img.shape
          for i in range(1, y - 1):
            for j in range(1, x - 1):
              if img[i, j] == schwaches_pixel:
                try:
                  if (
                  (img[i + 1, j - 1] ==
                  starkes_pixel)
                  or
                  (img[i + 1, j] == starkes_pixel)
                  or
                  (img[i + 1, j + 1] ==
                  starkes_pixel)
                  or
                  (img[i, j - 1] == starkes_pixel)
                  or
                  (img[i, j + 1] == starkes_pixel)
                  or
                  (img[i - 1, j - 1] ==
                  starkes_pixel)
                  or
                  (img[i - 1, j] == starkes_pixel)
                  or
```

```
                (img[i - 1, j + 1] ==
                starkes_pixel)):
                  img[i, j] = starkes_pixel
                else:
                  img[i, j] = 0
              except IndexError as e:
                pass

        return img
```

IV. Kantenerkennung mit dem Canny Algorithmus

Im folgenden werden die oben genannten Schritte des Canny Algorithmus einzeln auf ein Bild angewendet und näher erläutert.

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 wir hier nur einen 5x5 Kern nehmen, welcher ein ausreichend gutes Ergebnis bei gleichzeitig guter Performance mit sich bringt.

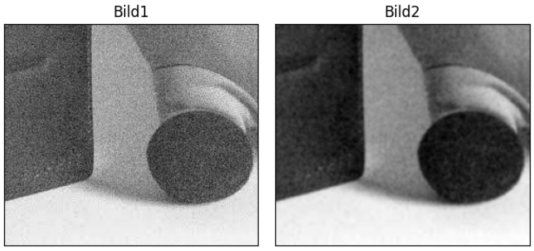

Fig. 5. *Links das Original und rechts unter Anwendung der Noise reduction.*

Das verrauschte Bild wurde Kontrastreicher und Kanten dadurch deutlicher erkennbar.
Im Nachfolgenden werden wir unseren selbst gebauten Canny anhand des Hochshcullogos testen. Hier entfällt der erste Schritt, da das Hochschullogo ein digital erzeugtes Bild ist und somit kein Rauschen beinhaltet.
Hier das Originalbild des Hochschullogos in Graustufen:

Fig. 6. *Schwarzweissbild des Hochschullogos.*

B. Edge and Gradient Calculation

Nun wird berechnet, ob ein Pixel zu einer Kante gehört und gegbenenfalls die Richtung der Kante. Ob ein Pixel einer Kante zugehörig ist wird anhand der Intensiätsänderung bezüglich der Umliegenden Pixel bestimmt. Hierfür werden die Intensitätsänderungen in horizontaler sowie vertikaler Richtungeines jeden Pixels bezüglich seiner Nachbarn berechnet.
Mathematisch realisiert wird dies durch eine Faltung des Bildes mit einem Sobel Kern.
Für die Intensität ergibt sich folgende Formel:

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

Während der Gradient sich wie folgt berechnen lässt:

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

Nach diesem Schritt hat man bereits ein Ergebnis, anhand welchem man die Kanten eines Bildes erkennen kann. Die gefundenen Kanten sind jedoch noch sehr grob und undetailiert.

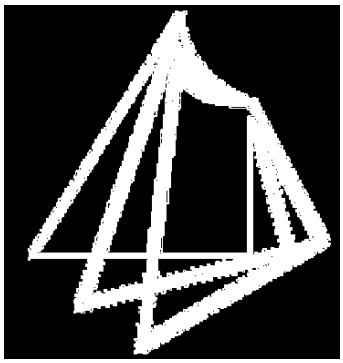

Fig. 7. *Bild nach Anwendung des Sobel Filters*

Das Ergebnis muss also noch weiter verarbetet werden um Kanten auszudünnen, fehlerhaft als Kante erkannte Pixel zu

entfernen und die Darstellung zu verfeinern. Hier setzen wir mit der Non-Maximum-Suppression an.

C. Non Maximum Suppression

Die momentan noch mehr als 1 Pixel breiten Kanten werden nun mit dem Non-Maximum Suppression-Algorithmus ausgedünnt. Hierbei wird das gesamte Bild durchlaufen und jedes Pixel auf seinen Gradienten und Intensitätswert geprüft. Abhängig davon werden die Intensitätswerte der jeweiligen beiden Nchbarpixel mit dem Intensitätswert des aktuellen Pixels verglichen. Hat eines der beiden Nachbarpixel einen höheren Intensitätswert als das aktuelle Pixel, wird der Intensitätswert des aktuellen Pixels auf 0 gestezt. Sollte das aktuelle Pixel einen höheren Intensitätswert als die beiden Nachbarn aufweisen, so bleibt sein Intensitätswert unverändert. Nachdem der Algorithmus das Bild durchlaufen hat, ergibt sich ein neues Bild in welchem nur die Pixel entlang der gefundenen Kanten mit maximalen Intensitätswerten behalten wurden und alle anderen nun den Intensitätswert 0 aufweisen, also nichtmehr zur Kante gehören. Unser Hochschullogo sieht nun folgerndermaßen aus:

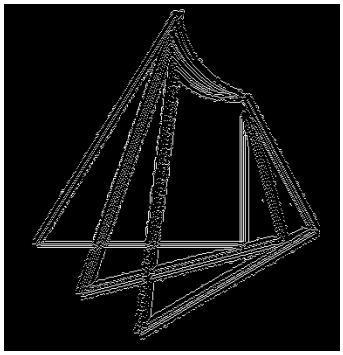

Fig. 8. *Bild nach Anwendung der Non Maximum Suppression*

Verglichen mit dem Bild aus dem vorherigen Schritt kann man erkennen, dass die Kanten deutlich dünner und detailierter wurden. Man kann das Logo nun bereits gut nur anhand der Kanten erkennen. Die Intensitätswerte der nun gefundenen Kanten variieren jedoch noch stark. Viele Bildverarbeitungsprogramme oder auch Algorithmen basieren darauf, dass die gefundenen Kanten einen einzige Intensitätswert aufweisen und verarbeiten das Bild dann anhand des angegebenen Intensitätswertes. Um auch unseren Kanten einen eingeutlichen Intensitätswert zu geben nutzen wir die folgenden zwei Schritte.

D. Double threshold

Der Double Threshold Filter arbeitet mit 2 Schwellenwerten, einem oberen und einem unteren. Die Intensitätswerte eines jeden Pixels werden geprüft und mit den Schwellenwerten verglichen. Je nachdem wo sich der Intensitätswert eines Pixels bezüglich der Schwellenwerte einordnen lässt, wird das Pixel in eine der folgenden Kateforien eingestuft und sein Intensitätswert dahingehend angepasst:

1) starkes Pixel, Intensitätswert wird auf 255 gesetzt.
2) schwaches Pixel, Intensitätswert wird (meist) auf den unteren Schwellenwert gesetzt.
3) nicht der Kante zugehörig, Intensitätswert wird auf 0 gesetzt.

Nach Anwendung des Double Threshold Filters sieht unser Hochschullogo fogendermaßen aus:

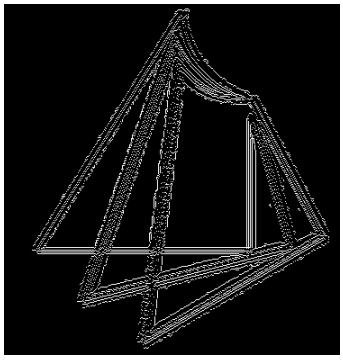

Fig. 9. *Bild mit starken (weiss) und schwachen (grau) Pixeln*

Die starken Kanten sind heller und klarer erkennbar. Ebenso sind leichte, nach der Non Maximum Suppression noch vohandene unfeinheiten oder einzelne Kantenpixel beseitigt worden.

E. Hysteresis

Im letzten Schritt wenden wir noch eine Hysterese auf das gesamte Bild an, um die einzelnen Pixel noch einmal final zu überprüfen und eventuelle Lücken in Kanten zu schließen oder alleinstehende und damit fälschlicherweise erkannte Kantenpixel zu eliminieren.
Jedes Pixel wird auf seine 8 direkten Nachbarn geprüft.
Befindet sich in der direkten Nachbarschaft eines Pixels kein starkes Pixel, so wird der Intensitätswert des Pixels auf 0 gesetzt und es ist damit kein Kantenpixel mehr.

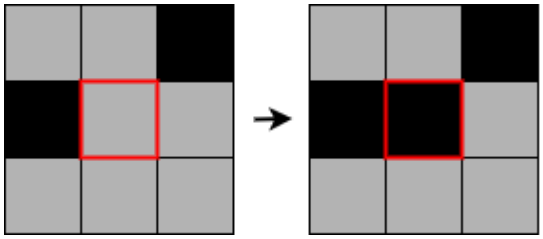

Fig. 10. *Ein Pixel ohne starkes Pixel in der Nachbarschaft*

Befindet sich unter den direkten Nachbarn des Pixels ein starkes Pixel, so wird der Intensitätswert des aktuellen Pixels

auf 255 gesetzt, es wird ebenfalls zu einem starken Pixel.

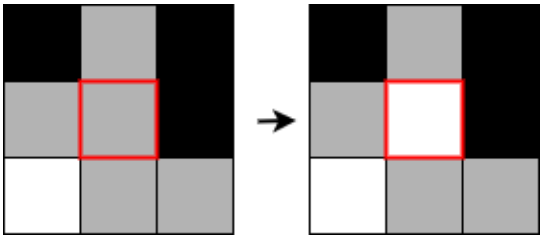

Fig. 11. *Ein Pixel mit einem starken Pixel in der Nachbarschaft*

Nach all diesen Schritten ist dies unser finales Kantenbild:

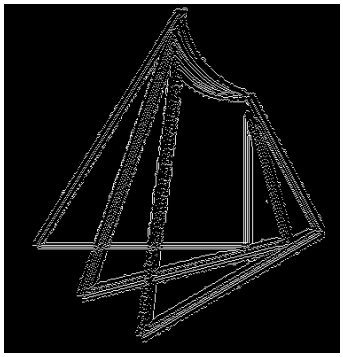

Fig. 12. *Finales Bild*

Es ist frei von alleinstehenden Kantenpixeln und die gefundenen Kanten sind fein und deutlich zu erkennen.

F. Vergeich

Nun wollen wir den eigens Entwickelten Canny Algorithmus mit dem wirklichen Canny Algorithmus vergleichen. Hier das Kantenbild des richtigen Canny Algorithmus und unseres als Vergleich:

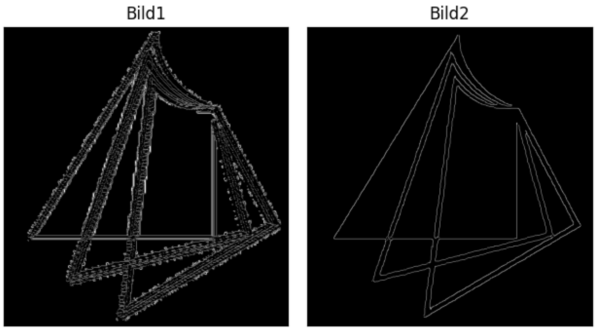

Fig. 13. *Kantedektion mit unserem Algorithmus (links) und mit dem richtigen Canny (rechts)*

Man erkennt den Unterschied sofort, der richtige Canny Algorithmus hat die Kanten noch mehr ausgedünnt und verfeinert. Im Vergleich dazu sind unsere Kanten noch immer sehr dick und unfein.
Werfen wir nun einen Blick auf die Rechenzeit:
Während unser Algorithmus mit 1.221 Sekunden für das Hochschullogo nicht langsam ist, setzt Canny mit 0.026 Sekunden, der knapp 48-fachen Geschwindigkeit, für die Kantenerkennung des Hochschullogos Maßstäbe in ganz anderen Dimensionen. Mit einer einfachen Nachbildung des Canny Algorithmus ist es also nicht getan.

V. Effizienzgewinn durch Kantenverfolgung

A. Idee

Es ist nicht notwendig, das komplette Bild und damit jedes einzelne Pixel mehrmals zu überprüfen und berechnen.
Sobald beim Durchlaufen des Bildes ein Kantenpixel gefunden und die Kantenrichtung bestimmt wurde, kann man mit einem Kantenverfolgungsalgorithmus dieser Kante bis zum Ende folgen und auf dem Weg jedes Pixel der Kante entsprechend berechnen und markieren.
Beginnt man bei dieser Variante allerdings gleich wie beim Canny Algorithmus, das Bild Pixel für Pixel zu untersuchen, müsste man auch hier, sobald ein Kantenpixel gefunden und die Kante verfolgt wurde, das auf das Kantenpixel folgende Pixel prüfen, um sicherzustellen, dass man jede Kante im Bild gefunden hat.
Um also dem Grundgedanken, weniger Pixel zu prüfen und dadurch Rechenzeit einzusparen, nachzukommen, bedarf es einer Methode das Bild abzulaufen, welche von vornherein nicht jedes Pixel untersuchen würde, mithilfe der Kantenverfolgung jedoch trotzdem jedes Kantenpixel findet.
Als Referenzbild verwenden wir das folgende:

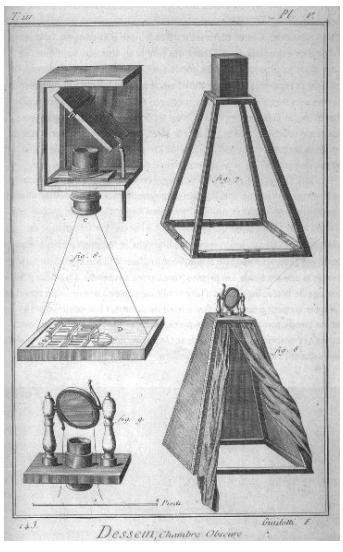

Fig. 14. *Testbild für unseren Kantenverfolgungsalgorithmus*

Dieses Bild enthält Kanten welche nahe beieinander liegen, größere, Grauwertemäßig nicht merklich unterschiedliche,

Flächen, weiter auseinanderliegende Kanten und Kanten unterschiedlicher breite, bis hin zu Linien. Es enthält zudem sowohl runde, als auch eckige oder kurvige Objekte. In diesem Bild sind die meisten Kantenformen enthalten, was es zu einem hervorragenden Testbild für unseren Algorithmus macht.

B. Das Bild durchlaufen

Um sicherzustellen, dass von Beginn an nicht jedes Pixel des Bildes durchlaufen wird, wenden wir eine spezielle Suchmethode an.

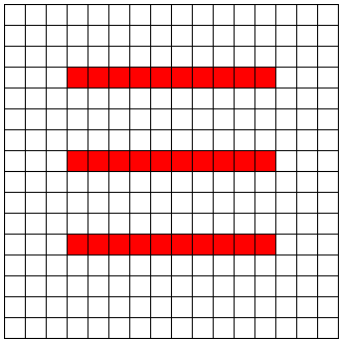

Fig. 15. *Pixelskelett eines Bildes mit untersuchten Pixeln*

Wir durchlaufen das Bild von Links nach Rechts und untersuchen es dabei in horizontalen Linien, welche untereinander sowie vom oberen und den seitlichen Rändern einen Abstand von 3 Pixeln und vom unteren Rand einen Abstand von mindestens 3 Pixeln, maximal jedoch 5 Pixeln, haben.
Grund für diesen Abstand ist unsere Suchmaske. Wir untersuchen immer ein 3×3 Quadrat der jeweils gegenüberliegenden Nachbarn. Für jedes so untersuchte Pixel berechnen wir also für 4 Richtungen, ob es sich um eine Kante handelt. hier unsere 4 Suchmasken:

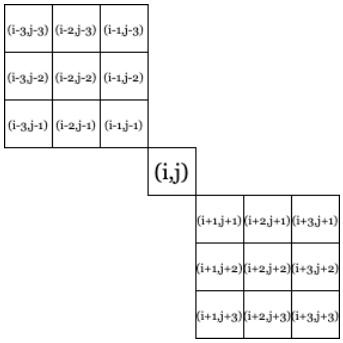

Fig. 16. *Maske zur Untersuchung auf eine Kante in Richtung der 2. Mediane*

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:

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

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

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.

VI. 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*