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 2

II Begriffsdefinitioen und Erklärungen 2
- II-A Was ist eine Kante . . . 2
- II-B Pixelnachbarn . . . 2
- II-C Bildrauschen . . . 3
  - II-C1 Wie entsteht Rauschen in digitalen Bildern . . . 3
  - II-C2 Methoden zur Rauschunterdrückung . . . 3
  - II-C3 Gauß Filter . . . 3

III Canny-Algorithmus 3
- III-A Gauß Kern . . . 3
- III-B Sobel Operator . . . 3
- III-C Non Maximum Suppression . . . 4
- III-D Double threshold . . . 5
- III-E Kantenvervollständigung durch Hysterese . . . 5

IV Kantenerkennung mit dem Canny Algorithmus 6
- IV-A Noise Reduction . . . 6
- IV-B Edge and Gradient Calculation . . . 6
- IV-C Non Maximum Suppression . . . 7
- IV-D Double threshold . . . 7
- IV-E Hysteresis . . . 7
- IV-F Vergeich . . . 8

V Effizienzgewinn durch Kantenverfolgung 8
- V-A Idee . . . 8
- V-B Das Bild durchlaufen . . . 9
- V-C Algorithmus . . . 10
- V-D Vergleich . . . 13
- V-E Performance Steigerung des Algorithmus . . . 13

VI Fazit 14

VII Literaturverzeichnis 14

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 unter einer halben Sekunde Rechenzeit 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 schneller machen 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 weiteren Verlauf dieses Dokuments mathematisch näher beleuchtet.
Der Canny Algorithmus lässt sich in 5 Schritte unterteilen:

1) Rauschuntedrückung
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 umhinkommt.
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 noch mal 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*).

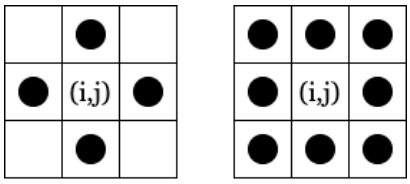

Fig. 1. *Vierer-Nachbarschaft (links) und Achter-Nachbarschaft (rechts)*

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 davon 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 Elementarteilchen, 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:

Elektronen 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 Eingangssignal Zufälligkeiten aufweist, sind auch diese Signalfehler nicht mehr 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 Gaußkern. 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.

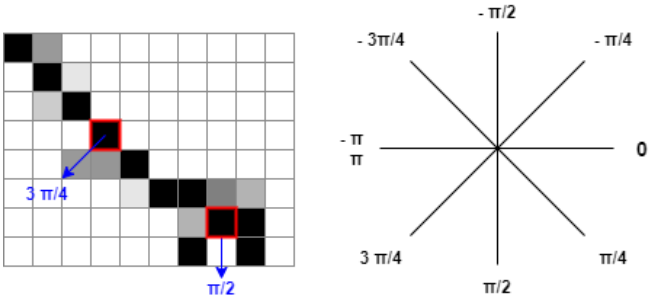

Fig. 3. *Darstellung der Non Maximum Suppression Technik*

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°).

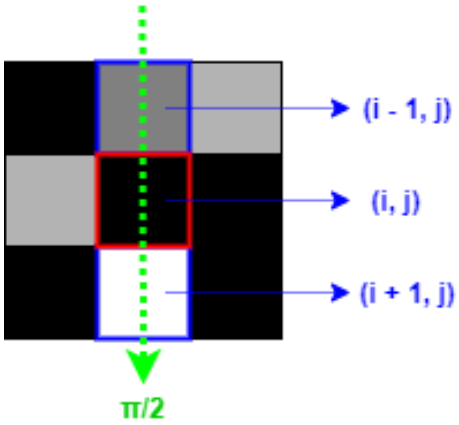

Fig. 4. *Nähere Betrachtung eines einzigen Pixels*

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 initialisiertes 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 insignifikant 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

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
```

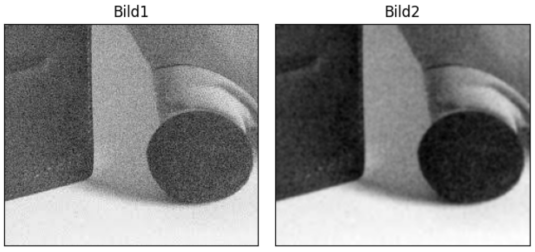

Fig. 5. *Links das Original und rechts unter Anwendung der Noise reduction.*

Das verrauschte Bild wurde Kontrastreicher und Kanten dadurch deutlicher erkennbar.
Im Nachfolgenden werden wir unseren selbst gebauten Canny anhand des Hochschullogos 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.*

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 ausschlaggebenden 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 5×5) 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 5×5 Kern nehmen, welcher ein ausreichend gutes Ergebnis bei gleichzeitig guter Performance mit sich bringt.

B. Edge and Gradient Calculation

Nun wird berechnet, ob ein Pixel zu einer Kante gehört und gegebenenfalls die Richtung der Kante. Ob ein Pixel einer Kante zugehörig ist, wird anhand der Intensitätsänderung bezüglich der umliegenden Pixel bestimmt. Hierfür werden die Intensitätsänderungen in horizontaler sowie vertikaler Richtung eines 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.

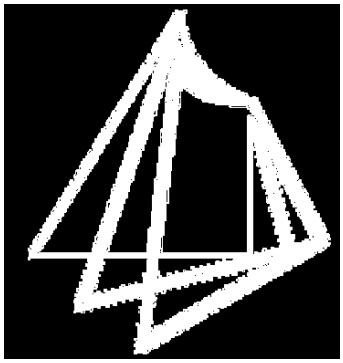

Fig. 7. *Bild nach Anwendung des Sobel Filters*

Das Ergebnis muss also noch weiter verarbeitet 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 Nachbarpixel 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 gesetzt. 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 nicht mehr zur Kante gehören. Unser Hochschullogo sieht nun folgendermaßen aus:

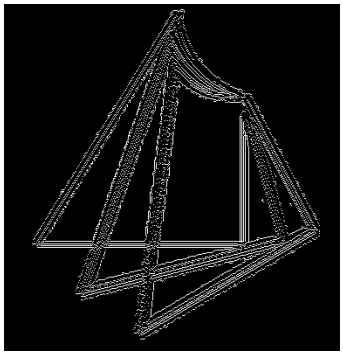

Fig. 8. *Bild nach Anwendung der Non Maximum Suppression*

Verglichen mit dem Bild aus dem vorherigen Schritt kann man erkennen, dass die Kanten deutlich dünner und detaillierter 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 einheitlichen 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 Kategorien 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 folgendermaßen aus:

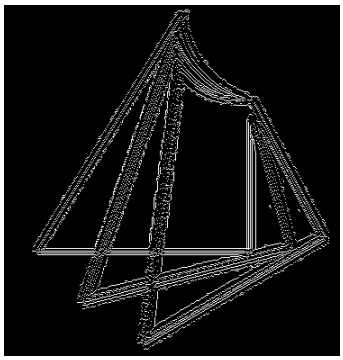

Fig. 9. *Bild mit starken (weiss) und schwachen (grau) Pixeln*

Die starken Kanten sind heller und klarer erkennbar. Ebenso sind leichte, nach der Non Maximum Suppression noch vorhandene 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.

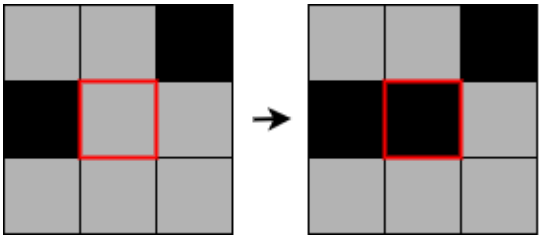

Fig. 10. *Ein Pixel ohne starkes Pixel in der Nachbarschaft*

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.

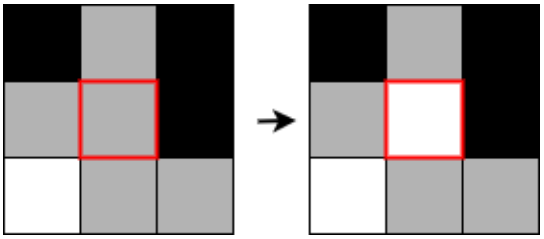

Fig. 11. *Ein Pixel mit einem starken Pixel in der Nachbarschaft*

Nach all diesen Schritten ist dies unser finales Kantenbild:

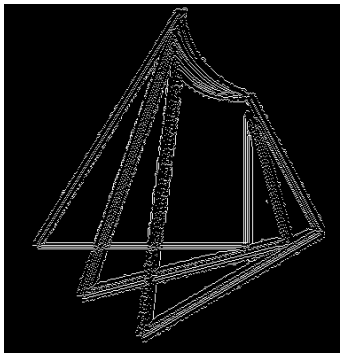

Fig. 12. *Finales Bild*

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:

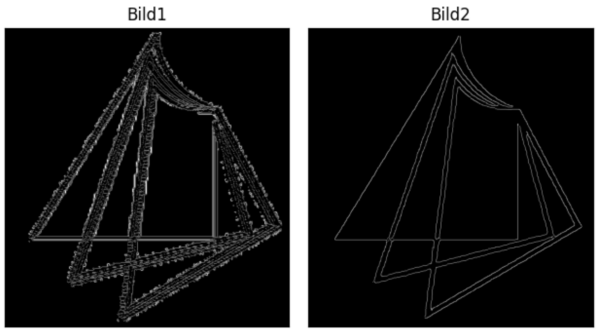

Fig. 13. *Kantedektion mit unserem Algorithmus (links) und mit dem richtigen Canny (rechts)*

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:

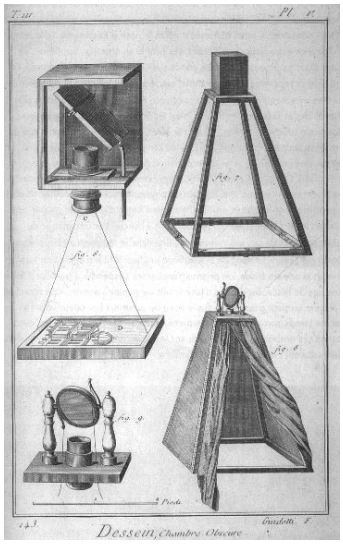

Fig. 14. *Testbild für unseren Kantenverfolgungsalgorithmus*

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.

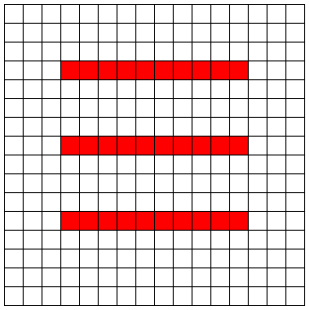

Fig. 15. *Pixelskelett eines Bildes mit untersuchten Pixeln*

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 sind unsere Suchmasken. 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, der blaue Pfeil gibt jeweils die Kantenrichtung an:

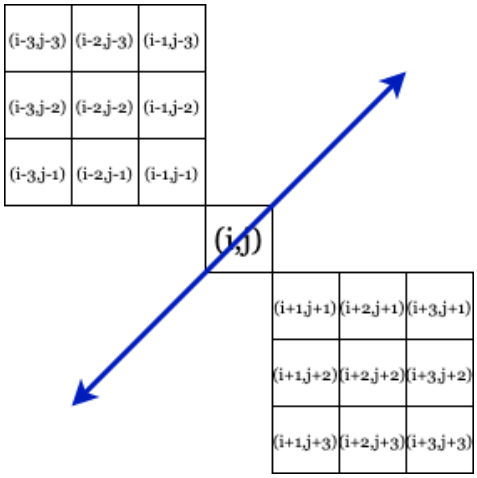

Fig. 16. *Maske zur Untersuchung auf eine Kante in Richtung der 1. Mediane*

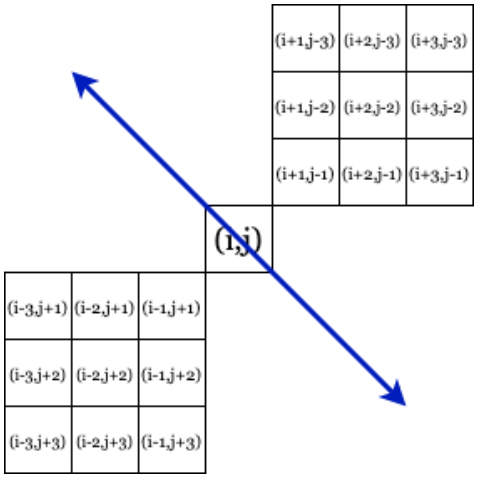

Fig. 17. *Maske zur Untersuchung auf eine Kante in Richtung der 2. Mediane*

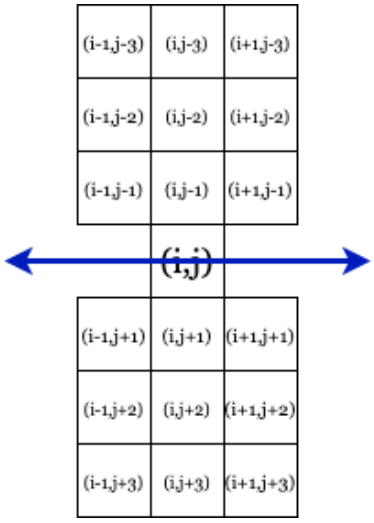

Fig. 18. *Maske zur Untersuchung auf eine Kante in horizontaler Richtung*

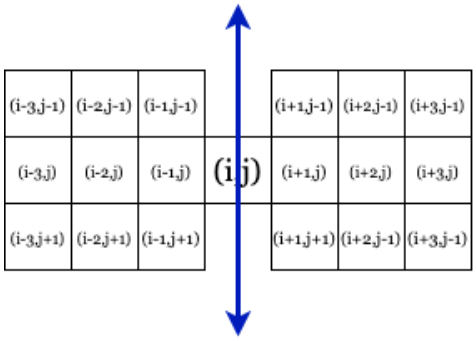

Fig. 19. *Maske zur Untersuchung auf eine Kante in vertikaler Richtung*

Mit dieser Methode das Bild zu durchlaufen gibt es allerdings einige Kanten, welche wir nicht finden können. Sollte eine Kante zwischen 2 roten Suchlinien in horizontaler Richtung verlaufen und unverzweigt sein oder befindet sich eine Kante in weniger als 3 Pixeln Abstand, unverzweigt und parallel zum jeweiligen Rand, so können wir diese nicht finden.
Wir wollen unseren Algorithmus auf mit Digitalkameras aufgenommene Bilder anwenden, dass eine solche Kante im Bild vorkommt, sollte also verschwindend gering sein. Objekte in der Realität sind größer als 3 Pixel. Die Kanten eines Objekts sind verzweigt und schneiden somit unweigerlich eine der roten Linien. Wir können daher sicher sein, dass wir immer mindestens alle Objekte anschneiden und somit die Kante durch Kantenverfolgung auch zwischen horizontal zwischen 2 Linien verlaufend finden werden.
Die Höhe des Bildes in Pixeln sei y, die Breite x. Somit berechnet sich die Anzahl der durchlaufenen Pixel, ohne eine Kante zu verfolgen, auf:

$$\text{Pixel}_{\text{ges}} = \frac{(y-3)}{4} * (x - 6)$$

Im Verhältnis zu allen Pixeln des Bildes, prüfen wir somit mindestens

$$\frac{x*y-3*x-6*y+18}{4*x*y}$$

Was, da der Nenner hier schneller steigt als der Zähler, bei steigender Bildgröße ein immer kleinerer Anteil wird.
Mit der Kantenverfolgung ist keine genaue Vorhersage möglich, da wir nicht im Vorfeld wissen, wie viele Kantenpixel sich im Bild befinden. Mit hoher Sicherheit kann man nur sagen, dass es weniger als alle Bildpixel sein werden.

C. Algorithmus

Wir realisieren auch diesen Algorithmus in Python. Folgende Funktion prüft das komplette Bild auf Kantenpixel und verfolgt gegebenenfalls eine gefundene Kante.

```
def edge_tracker(img, diff):
  start = time.time()
  whitescreen = create_whitescreen(img)
  x_length = img.shape[1]
  y_length = img.shape[0]
  px = 0
  check_diff = diff

  if diff > 3:
    check_diff = 3

  while px <= y_length - 2 * diff - 1:
    px += diff
    for j in range(check_diff,
    x_length - check_diff):
      next_pixel_pos = (px, j)
      if whitescreen[next_pixel_pos] ==
        254:
        dir = check_pixel((px, j),
            img, check_diff)
        whitescreen[next_pixel_pos] = dir

        npparr = []
        dirarr = []

        while (dir != 255)
              and
            (check_diff <
            next_pixel_pos[0] <
            (y_length - check_diff - 1))
              and
            (check_diff + 1 <
            next_pixel_pos[1] <
            (x_length - check_diff - 1)):
          next_pixel_pos =
            set_next_pixel_position
            (next_pixel_pos, dir)
          dir = check_pixel(
              next_pixel_pos,
              img, check_diff)
          npparr.append(next_pixel_pos)
          dirarr.append(dir)

          if len(npparr) > 3:
            for pos in npparr:
              whitescreen[pos] =
```

```
                    dirarr[npparr.index(pos)]
                else:
                  for pos in npparr:
                    whitescreen[pos] = 255

        end = time.time()
        print(end - start)
        return whitescreen
```

Die Funktion erwartet 2 Argumente, zum einen das Bild (*img*), zum anderen den Abstand in Pixeln (*diff*), welchen die Linien auseinanderliegen sollen.
In dieser Funktion aufgerufene Funktionen werde ich im weiteren Verlauf erläutern, schauen wir uns nun zuerst einmal die Funktionsweise des Algorithmus an.
Nachdem die Funktion aufgerufen wurde und 2 passende Argumente bekommen hat, wird ein Timer gestartet, damit wir die Laufzeit des Algorithmus messen können.
Durch *whitescreen* wird ein weißes Bild mit den Maßen des Originalbildes erstellt.

```
def create_whitescreen(image):
  whitescreen = np.full((image.shape), 254)
  return whitescreen
```

Im späteren Verlauf werden den Pixeln, welche nicht zu einer Kante gehören der Wert Grauwert 255 zugewiesen. Um die Pixel von den noch nicht geprüften Pixeln des Whitescreens zu unterscheiden, bekommen die Pixel hier den Wert 254.
Für den Fall, dass *diff* > 3 Pixel gewählt wird, setzen wir eine Ersatzvariable *check_diff*, welche maximal den Wert 3 annehmen kann. Da unsere Suchmasken nur einen Abstand von 3 Pixeln vom Rand benötigen, möchten wir uns mit unseren Suchlinien auch nicht weiter vom Rand entfernen.
Nun beginnen wir das Bild in einer While-Schleife abzulaufen. Die Schleife endet erst, wenn wir uns so weit am unteren Ende des Bildes befinden, dass wir keine weitere Linie mehr zeichnen können, welche einen Abstand von mindestens 3 Pixeln zum unteren Rand hat. Bei jedem Schleifendurchlauf erhöhen wir unsere Laufvariable *px* um *diff*, um den nötigen Abstand zu nächsten Linie zu erhalten.
In einer For-Schleife durchlaufen wir das Bild in x-Richtung. Hierbei setzen wir bereits zu Beginn der Schleife eine variable für die Position des aktuellen Pixels. So können wir in der Schleife leicht die Position wechseln, indem wir der Variable einen neuen Wert zuweisen. Hat ein Pixel im Whitescreen den Grauwert 254, heisst das, dass es noch nicht vom Algorithmus bearbeitet wurde und wir es somit prüfen müssen. Wir nutzen die *check_pixel* Funktion um zu bestimmen ob es sich um ein Kantenpixel handelt und, falls dies der Fall ist, die Richtung der Kante herauszufinden und diese der Variablen *dir* zuzuweisen.

```
def check_pixel(pos, image, diff):
  check_array = []

  check_array.append(
        top_left(pos, image, diff) -
        bottom_right(pos, image, diff))
  check_array.append(
        top(pos, image, diff) -
        bottom(pos, image, diff))
  check_array.append(
        top_right(pos, image, diff) -
        bottom_left(pos, image, diff))
  check_array.append(
        right(pos, image, diff) -
        left(pos, image, diff))

  maxim = max(check_array, key = abs)
  if abs(maxim) > 20:
    direction = check_array.index(maxim)
  else:
    direction = 255
  return direction
```

direction kann den Wert 0, 1, 2, 3 oder 255 annehmen. Die einzelnen Werte stehen räpresentativ für die Richtung der Kante.

- **0:** Die Kante verläuft entlang der 1. Mediane.
- **1:** Die kante verläuft in horizontaler Richtung.
- **2:** Die Kante verläuft entlang der 2. Mediane.
- **3:** Die Kante verläuft in vertikaler Richtung.
- **255:** Das geprüfte Pixel ist kein Kantenpixel.

Die Funktionen *top*, *bottom*, *left*, *right*,*top_left*,*top_right*, *bottom_left* und *bottom_right* berechnen den Mittelwert aus den Intensitätswerten der entsprechenden Nachbarn anhand der oben beschriebenen Suchmasken.

```
def top(pos, img, diff):
  list = [-1,0,1]
  sum_top = 0
  for i in range(1, diff + 1):
    for j in list:
      sum_top += img[pos[0] - i][pos[1] + j]

  return sum_top/(3 * diff)
```

```
def bottom(pos, img, diff):
  list = [-1, 0, 1]
  sum_bottom = 0
  for i in range(1, diff + 1):
    for j in list:
      sum_bottom +=
      img[pos[0] + i][pos[1] + j]

  return sum_bottom/(3 * diff)
```

```
def left(pos, img, diff):
  list = [-1, 0, 1]
  sum_left = 0
  for j in range(1, diff + 1):
    for i in list:
      sum_left +=
      img[pos[0] + i][pos[1] - j]

  return sum_left/(3 * diff)
```

```
def right(pos, img, diff):
  list = [-1, 0, 1]
  sum_right = 0
  for j in range(1, diff + 1):
    for i in list:
      sum_right +=
      img[pos[0] + i][pos[1] + j]

  return sum_right/(3 * diff)
```

```
def top_left(pos, img, diff):
  sum_top_left = 0
  for j in range(0, diff + 1):
    for i in range(0, diff + 1):
      sum_top_left +=
      img[pos[0] - i][pos[1] - j]

  sum_top_left -= img[pos]
  return sum_top_left/((3 * diff + 1) - 1)
```

```
def top_right(pos, img, diff):
  sum_top_right = 0
  for j in range(0, diff + 1):
    for i in range(0, diff + 1):
      sum_top_right +=
      img[pos[0] - i][pos[1] + j]

  sum_top_right -= img[pos]
  return sum_top_right/((3 * diff + 1) - 1)
```

```
def bottom_left(pos, img, diff):
  sum_bot_left = 0
  for j in range(0, diff + 1):
    for i in range(0, diff + 1):
      sum_bot_left +=
      img[pos[0] + i][pos[1] - j]

  sum_bot_left -= img[pos]
  return sum_bot_left/((3 * diff + 1) - 1)
```

```
def bottom_right(pos, img, diff):
  sum_bot_right = 0
  for j in range(0, diff + 1):
    for i in range(0, diff + 1):
      sum_bot_right +=
      img[pos[0] + i][pos[1] + j]

  sum_bot_right -= img[pos]
  return sum_bot_right/((3 * diff + 1) - 1)
```

Die Nachbarn mit der größten Differenz der gemittelten Grauwerte bestimmt die Richtung der Kante. Sollte keine Differenz über einem Schwellenwert liegen, welchen ich hier frei zu 20 gesetzt habe, so handelt es sich nicht um ein Kantenpixel. Der Wert der Kantenrichtung wird nun im Whitescreen im aktuellen Pixel gespeichert.
Nun kommen wir zum Teil der Kantenverfolgung. In einer weiteren While-Schleife überprüfen wir erst, dass *dir* nicht den Wert 255 hat, also ein Kantenpixel ist und die aktuelle Position in Bild weit genug von den Rändern entfernt ist, damit unsere Suchmasken das nächste Pixel noch prüfen können.
Mit *set_next_pixel_position* setzen wir die Position des nächsten Pixels abhängig von der Kantenrichtung.

```
def set_next_pixel_position(next_pixel_pos,
                            dir):
  _px = next_pixel_pos[0]
  _j = next_pixel_pos[1]
  if dir == 0:
    next_pixel_pos = (_px + 1, _j - 1)
  elif dir == 1:
    next_pixel_pos = (_px, _j + 1)
  elif dir == 2:
    next_pixel_pos = (_px + 1, _j + 1)
  elif dir == 3:
    next_pixel_pos = (_px + 1, _j)
  return next_pixel_pos
```

Anschließend überprüfen wir wie oben das nächste Pixel und speichern sowohl die Position als auch den Wert der Kantenrichtung in einem jeweils dafür erstellten Array. Sobald die While-Schleife verlassen wird, entweder weil *dir* den Wert 255 angenommen hat oder weil wir uns zu nah an einem Rand des Bildes befinden, wird die Länge des Arrays der Positionen geprüft. Übersteigt sie den Wert 3, so werden an allen gespeicherten Positionen im Whitescreen die jeweilige Kantenrichtung gesetzt. So werden keine Kanten markiert, deren Länge kleiner 3 Pixel ist. Die Kantenpixel werden jedoch noch immer berechnet, hiermit steigern wir also nur die Qualität des Ergebnisbildes, nicht jedoch die Performance.
Nachdem alle Suchlinien auf diese Weise durchlaufen wurden geben wir die Laufzeit des Algorithmus und das Kantenbild aus.

D. Vergleich

Auch hier ziehen wir den originalen Canny Algorithmus zum Vergleich.

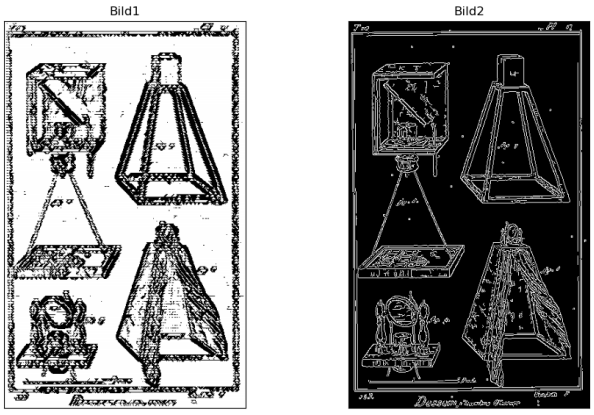

Fig. 20. *Kantedektion mit unserem Algorithmus (links) und mit Canny (rechts)*

Wir erkennen, dass unsere Kanten noch nicht so fein und granular wie die des Canny sind. Grund hierfür ist, dass wir die Kante in voller Breite finden und an keiner Stelle ausdünnen. Statt die Kante auszudünnen wäre der richtige Ansatz, bereits im Vorfeld zu prüfen, ob dieses Kantenpixel noch als Teil der Kante markiert werden soll oder nicht. Mit diesem Vorgehen kann man Rechenzeit einsparen und gleichzeitig die Kanten feiner halten. Trotz allem wurden mit unserem Algorithmus, alleine auf einem Grauwertvergleich basierend, alle Kanten gefunden und verfolgt. Auch die runden und kurvigen Objekte wurden richtig erkannt und markiert.
Werfen wir nun einen Blick auf die Laufzeit. Während Canny das Bild in 0.002 Sekunden bearbeitet, benötigt unser Algorithmus 13.36 Sekunden. Hier muss, wie oben beschrieben, die Performance noch stark erhöht werden.

E. Performance Steigerung des Algorithmus

Unsere Suchmasken berechnen zurzeit 2 3×3 Matrizen für jede der 4 Möglichen Richtungen. Diese Berechnungen nehmen am meisten Zeit in Anspruch. Nun werden wir die Suchmasken Abändern, so, dass sie noch immer die Kantenrichtung anhand der durchschnittlichen Grauwerte der Nachbarpixel berechnen, allerdings mit weniger Rechenaufwand als zuvor. Die neuen Suchmasken sollen nun nur die jeweils 2 nächsten Nachbarpixel des aktuellen Pixels berechnen. Dies sind die 4 neuen, vereinfachten Suchmasken:

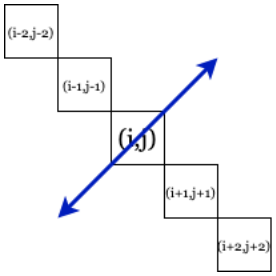

Fig. 21. *Maske zur Untersuchung auf eine Kante in Richtung der 1. Mediane*

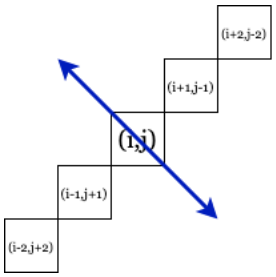

Fig. 22. *Maske zur Untersuchung auf eine Kante in Richtung der 2. Mediane*

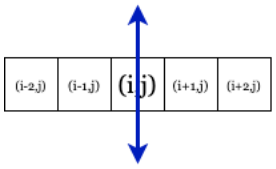

Fig. 23. *Maske zur Untersuchung auf eine Kante in horizontaler Richtung*

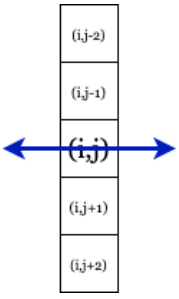

Fig. 24. *Maske zur Untersuchung auf eine Kante in vertikaler Richtung*

Vergleichen wir nun das Ergebnis mit dem Vorherigen Ergebnissen:

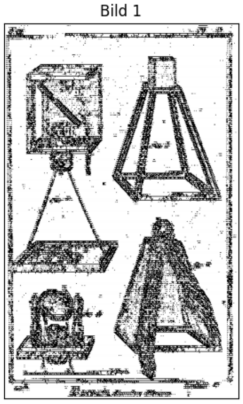

Fig. 25. *Ergebnis der Kantensuche mit den neuen Masken*

Auch mit dieser, durchaus effizienteren Methode wurden alle Kanten erkannt. Die Kanten sind etwas unfeiner als die der vorhergehenden Methode, jedoch ausreichend, um die Objekte zu erkennen. Ebenso wurde Zeit eingespart, mit 9,15 Sekunden ist diese Methode um 4,20 Sekunden schneller als ihr Rechenaufwändigerer Vorgänger.
Im Vergleich zur Kantenerkennung mit dem Canny Algorithmus sind die Kanten allerdings noch immer sehr grob und unfein.

VI. Fazit

Die Idee, Rechenzeit zu sparen indem wir nicht, wie beim Canny Algorithmus jedes Pixel berechnen müssen, sondern durch eine geschickte Abtastung des Bildes erheblich weniger, ist gerechtfertigt. Allerdings ist die Berechnung der Verfolgung einer Kante aufwändiger als das überprüfen eines Pixels auf eine Kante. Ebenso sind die von uns gefundenen Kanten noch sehr rau und müssen geglättet werden, was wiederum erneuten Rechenaufwand mit sich ziehen würde. Der Weg um mehr Rechenzeit zu sparen wäre ein Verfahren zu entwickeln, welches bereits bevor man das Pixel auf ein Kantenpixel prüft, beispielsweise anhand der Nachbarn und deren Werte, Information darüber gibt, ob dieses Pixel überprüft werden muss, oder nicht. Somit würden die Kanten glatter und feiner werden und die benötigte Rechenzeit würde sich noch einmal verringern.
Schlussendlich konnte mein Algorithmus nicht im Ansatz mit der Geschwindigkeit des Canny Algorithmus mithalten. Es werden noch viele Pixel auf eine Kante geprüft, und die dann eventuell gefundene Kante verfolgt, bei welchen es nicht nötig ist diese zu prüfen, da die Kante des Pixel bereits erkannt wurde und wir die Kante nicht in voller Breite benötigen sondern fein halten wollen.
Ich bin mir sicher, dass, mit einer geeigneten Methode zum Entscheiden, ob ein Pixel noch überprüft werden muss oder nicht, noch einmal ein ganzzahliger Faktor an Rechenzeit eingespart werden kann. Davon, dass dieser Algorithmus dann mit der Geschwindigkeit des Canny Algorithmus mithalten kann, bin ich jedoch nicht überzeugt. Der Canny Algorithmus scheint ungeschlagen im Bereich der Kantenerkennung.

VII. 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*
- *https://upload.wikimedia.org/wikipedia/commons/9/9f/Camera_obscu*
- *https://de.linkfang.org/wiki/Nachbarschaft_(Bildverarbeitung)*