# 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

**TODO**

## II. Mathematische Grundlagen

### A. 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. 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.

In Python wird der Gauß Kern folgendermaßen realisiert:

```
def gaussian_kernel(size, sigma=1):
  size = int(size) // 2    # c1 * 1
  x, y = np.mgrid[-size:size + 1,
  -size:size + 1]      # c2 * 1
  normal = 1 /
  (2.0 * np.pi * sigma ** 2)      # c3 * 1
  g = np.exp(-((x ** 2 + y ** 2) /
  (2.0 * sigma ** 2))) * normal      # c4 * 1

  return g
```

### B. Sobel Operator

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

| | | |
|---|---|---|
| -1 | 0 | +1 |
| -2 | 0 | +2 |
| -1 | 0 | +1 |

$G_x$

| | | |
|---|---|---|
| +1 | +2 | +1 |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

$G_y$

Fig. 1. *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_filter(img):
  Gx = np.array([[-1, 0, 1], [-2, 0, 2],
  [-1, 0, 1]], np.float32)      #c5 * 1
  Gy = np.array([[1, 2, 1], [0, 0, 0],
  [-1, -2, -1]], np.float32)      #c5 * 1

  Ix = ndimage.filters.convolve(img, Gx).
  astype(float)      # c6 * 1
  Iy = ndimage.filters.convolve(img, Gy).
  astype(float)      # c6 * 1

  G = np.hypot(Ix, Iy)      # c7 * 1
  G = G / G.max() * 255      # c8 * 1
  theta = np.arctan(Iy, Ix)      # c9 * 1

  return G, theta
```

### 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. Bildlich kann man sich die Technik folgendermaßen vorstellen

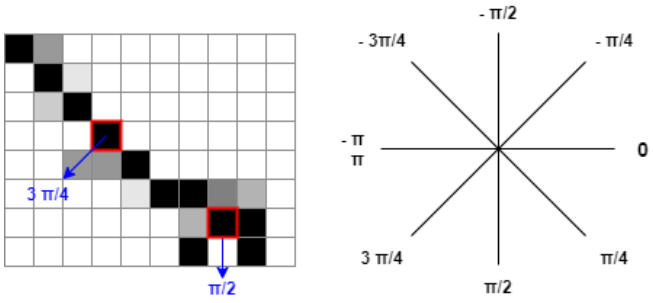


Fig. 2. *Darstellung der Non Maximum Suppression Technik*

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

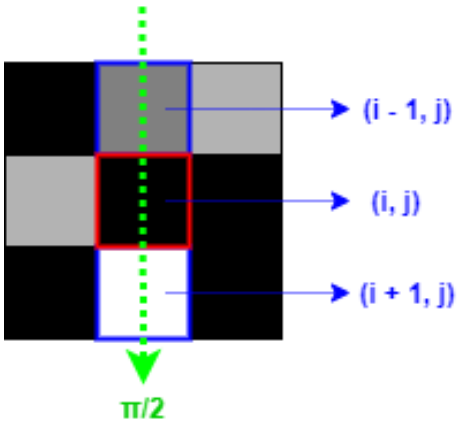


Fig. 3. *Nähere Betrachtung eines einzigen Pixels*

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 (rot im Bild markiert) 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 hat einen Intensitätswert von 255, da es weiß ist). 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:

- Eine Matrix mit nullen der gleichen Größe 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 realisieren wir durch folgenden Python Code:

```
def non_max_suppression(img, D):
  M, N = img.shape     # c10 * 1
  Z = np.zeros((M, N))     # c11 * 1
  angle = D * 180. / np.pi     # c12 * 1
  angle[angle < 0] += 180      # c13 * 1

  for i in range(1, M - 1):    # c14 * (M - 2)
    for j in range(1, N - 1):
    # c15 * ((M - 2) * (N - 2))
      try:     # c16 * ((M - 2) * (N - 2))
        q = 255     # c17 * ((M - 2) * (N - 2))
        r = 255     # c17 * ((M - 2) * (N - 2))

        # angle 0
        if (0 <= angle[i, j] < 22.5) or
        (157.5 <= angle[i, j] <= 180):
        # c18 * ((M - 2) * (N - 2))
          q = img[i, j + 1]    # c19 * A0
          r = img[i, j - 1]    # c19 * A0
        # angle 45
        elif 22.5 <= angle[i, j] < 67.5:
        # c20 * (((M - 2) * (N - 2)) - A0)
          q = img[i + 1, j - 1]    # c19 * A45
          r = img[i - 1, j + 1]    # c19 * A45
        # angle 90
        elif 67.5 <= angle[i, j] < 112.5:
        # c20 * (((M - 2) * (N - 2)) -
        # A0 - A45)
          q = img[i + 1, j]    # c19 * A90
          r = img[i - 1, j]    # c19 * A90
        # angle 135
        elif 112.5 <= angle[i, j] < 157.5:
        # c20 * (((M - 2) * (N - 2)) -
        # A0 - A45 - A90)
          q = img[i - 1, j - 1]    # c19 * A135
          r = img[i + 1, j + 1]    # c19 * A135

        if (img[i, j] >= q) and
        (img[i, j] >= r):
        # c21 * ((M - 2) * (N - 2))
          Z[i, j] = img[i, j]      # c22 * Y
        else:
          Z[i, j] = 0      # c23 * (((M - 2) *
          # (N - 2)) - Y)

      except IndexError as e:      # c24 * 0
        pass     # c24 * 0
  return Z
```

### D. Double threshold

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

- **Starke Pixel**
  - Pixel, deren Intensitätswert hoch genug ist, dass wir uns sicher sein können, dass sie ein Teil der finalen Kante sind.
- **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.
- **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 die 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, lowRatio=0.05,
           highRatio=0.09):
  highThreshold = img.max() * highRatio
  # c26 * 1
  lowThreshold = highThreshold * lowRatio
  # c27 * 1

  M, N = img.shape      # c10 * 1
  res = np.zeros((M, N))  # c11 * 1

  strong = 255  # c17 * 1
  weak = 25     # c17 * 1
  zero = 0      # c17 * 1

  strong_i, strong_j = np.
  where(img >= highThreshold)      # c28 * 1
  zeros_i, zeros_j = np.
  where(img < lowThreshold)      # c28 * 1
  weak_i, weak_j = np.where(
  (img <= highThreshold) &
  (img >= lowThreshold))   # c29 * 1

  res[strong_i, strong_j] = strong
  # c30 * 1
  res[weak_i, weak_j] = weak       # c30 * 1
  res[zeros_i, zeros_j] = zero     # c30 * 1

  return res
```

### 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
```

## III. 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:

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

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:

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

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:

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

### 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:

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

### F. Komplette Laufzeit

Für die komplette Laufzeit $T(n)_{\text{Canny}}$ 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:

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

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

## IV. 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.

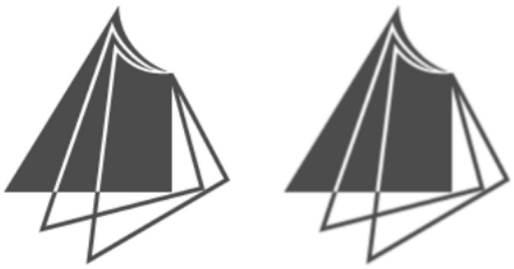

Fig. 4. *Links das original und rechts unter Anwendung des Gauß Filters.*

### *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.

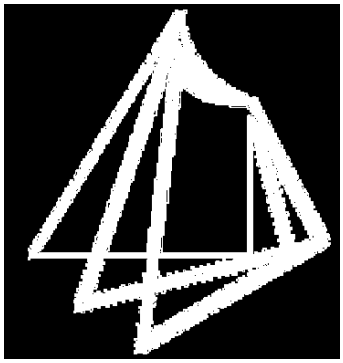

Fig. 5. *Bild nach Anwendung des Sobel Filters*

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.

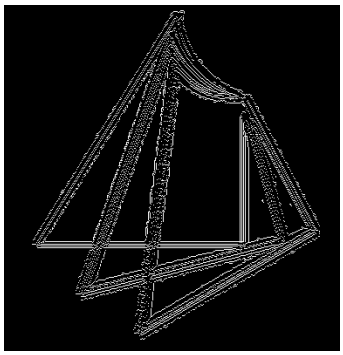

Fig. 6. *Bild nach Anwendung der Non Maximum Suppression*

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).

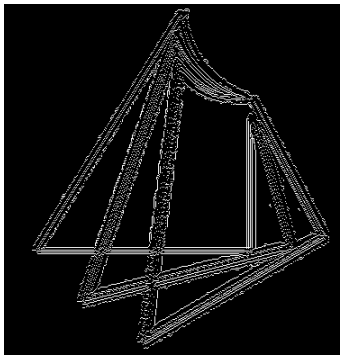

Fig. 7. *Bild mit starken (weiss) und schwachen (grau) Pixeln*

### 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.

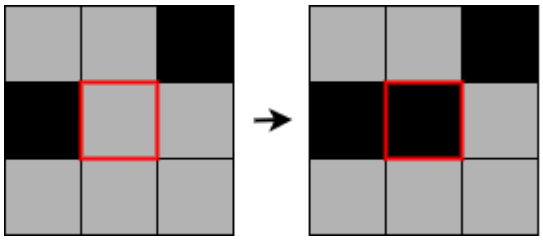

Fig. 8. *Ein Pixel ohne starkes Pixel in der Nachbarschaft*

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

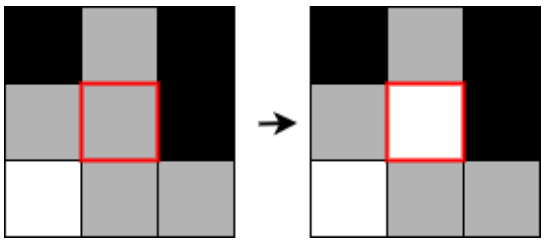

Fig. 9. *Ein Pixel mit einem starken Pixel in der Nachbarschaft*

## V. Effizienzgewinn durch Kantenverfolgung

### A. Idee

Sobald eine Kante gefunden wurde ist es nicht mehr notwendig, jedes einzelne Pixel des kompletten Bildes zu überprüfen. Man kann die Kante entlang laufen und so nur die relevanten Bereiche des Bildes untersuchen, was den Rechenaufwand enorm verringert. Es wurde schnell klar, dass Kantenverfolgung hier nicht zielführend ist, da es sich sehr sehr schnell im Kreis drehen kann. <- Ideen und Rechnungen zeigen.

**TODO**