PERFORMANCEVERGLEICH KONKURRIERENDER IMPLEMENTIERUNGEN DER BILDKORREKTUR VON AUFNAHMEN MIT EINEM FISCHAUGENOBJEKTIV AUF EINER ZYNQ-PLATTFORM

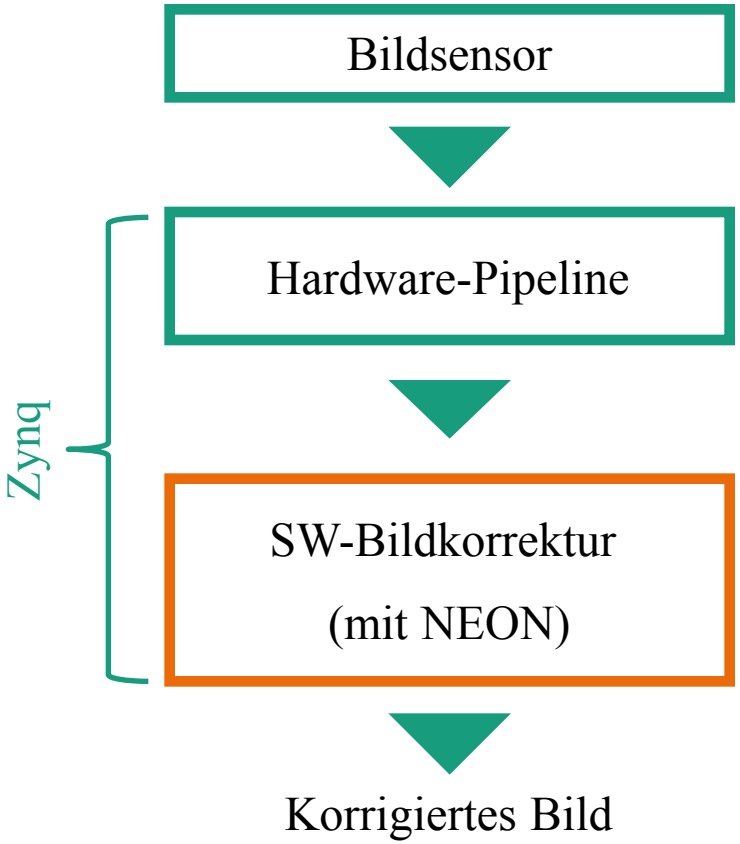
Zwischenpräsentation zur Studienarbeit

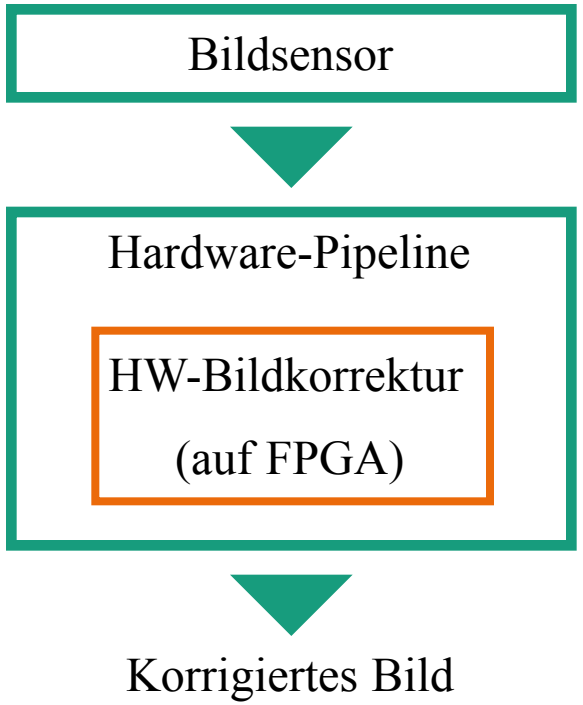
Christian Skubich

Gliederung

- Aufgabenstellung
- Bildkorrektur mit Lensfun
  - Ziel der Bildkorrektur
  - Phasen der Bildkorrektur
    - Korrektur der Projektion
    - Korrektur von Verzeichnungen
    - Interpolation
  - Algorithmus
  - Umsetzung in Software/Hardware
- Stand der Arbeit

Aufgabe: Vergleich von Hardware und SW-Implementierung einer Bildkorrektur auf Zynq-Plattform





Ziel der Bildkorrektur: Linientreue Darstellung von Fisheye Bildern

- Warum sind Linien nicht gerade abgebildet?
  - Projektion des Objektivs
  - Objektiv-Fehler

Bild-Quelle: peña [1]

1. Schritt der Bildkorrektur: Korrektur der Projektion

- Teile vom Bild gehen verloren
- Krümmung der Linien weitestgehend behoben

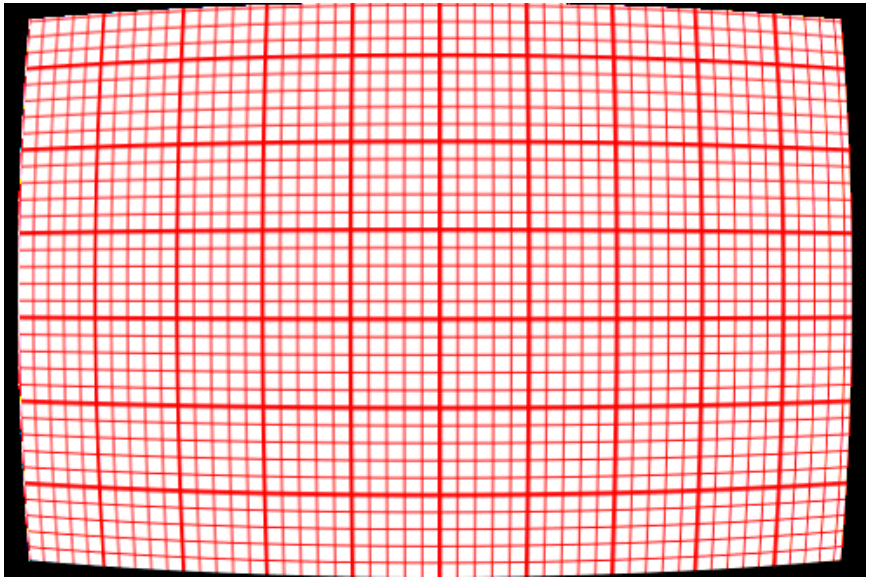
Bild-Quelle: peña [1]

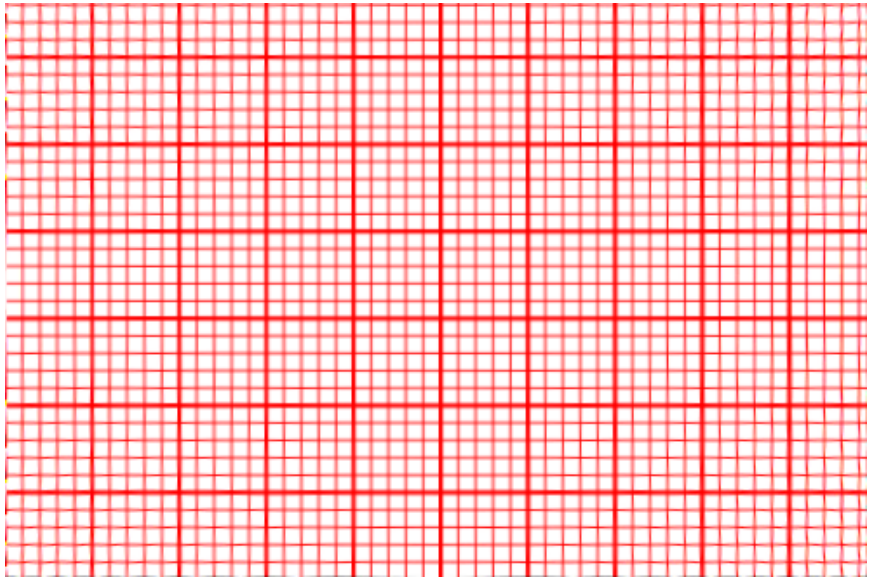
2. Schritt der Bildkorrektur: Objektivfehler

- Verbleibende Verzerrungen vor allem an den Rändern/Kanten

Bild-Quelle: peña [1]

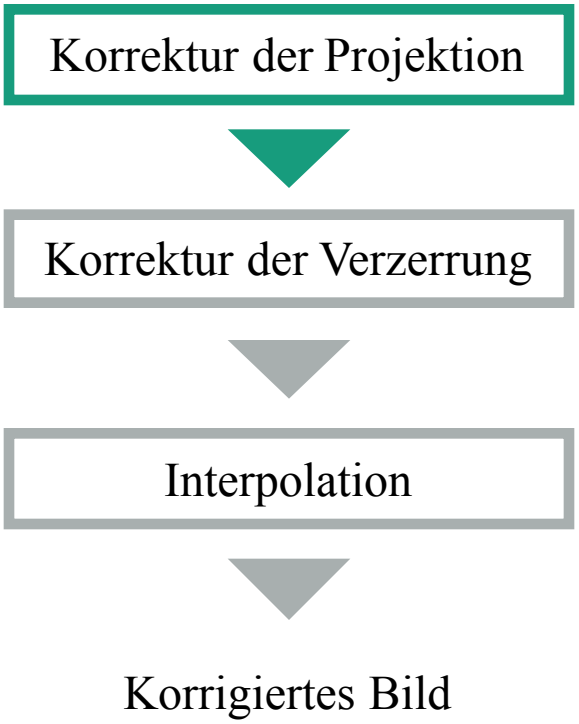
2. Schritt der Bildkorrektur: Objektivfehler





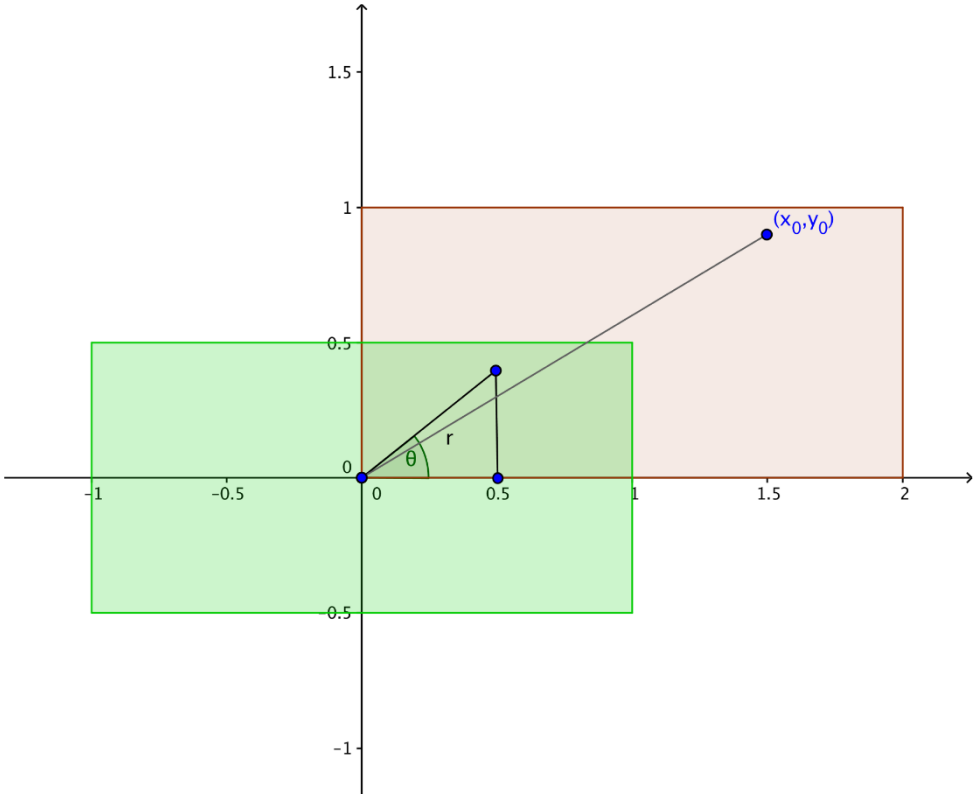
- Simulation der Objektivfehler an Gitter-Bild

1. Phase der Bildkorrektur: Korrektur der Projektion



Koordinatensystem bei Bildkorrektur

- Polarkoordinaten (r, θ)
- Korrektur des Radius:
  - $r' = f(r)$
  - Winkel bleibt erhalten



Wichtige Projektionen

Rectilinear

(linientreu)

$r = f \cdot tan(\theta)$

Equidistant

$r = f \cdot \theta$

Stereographic

(winkeltreu)

$r = 2 \cdot f \cdot tan(\frac{\theta}{2})$

Equisolid

(flächentreu)

$r = 2 \cdot f \cdot sin(\frac{\theta}{2})$

f: Brennweite

θ: Winkel zw. Optischer Achse und Licht

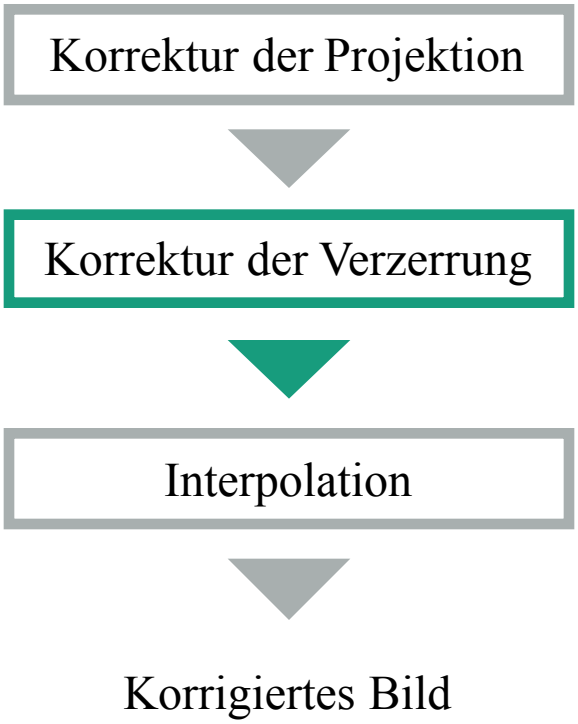
r: Abstand zwischen Pixel auf Sensor und Sensor Mitte

Änderung der Projektion

- Umrechnen möglich, z.B. Equidistant zu Rectilinear:
  - Kartesische Koordinaten werden mit Verhältnis skaliert
  - $x' = \frac{r_{equidistant}}{r_{rec}} \cdot x = \frac{f \cdot tan^{-1}(r_{rec}/f)}{r_{rec}} \cdot x$

Quell-Projektion	Ziel-Projektion	Umrechnungsfaktor
Equidistant	Rectilinear	$\frac{f \cdot tan^{-1}(r_{rec}/f)}{r_{rec}}$
Equisolid	Rectilinear	$\frac{2f \cdot sin(\frac{tan^{-1}\left(\frac{r_{rec}}{f}\right)}{2})}{r_{rec}}$
Stereographic	Rectilinear	$\frac{2f \cdot tan(\frac{tan^{-1}\left(\frac{r_{rec}}{f}\right)}{2})}{r_{rec}} = \frac{2}{1+\sqrt{1^2+\left(\frac{r_{rec}}{f}\right)^2}}$

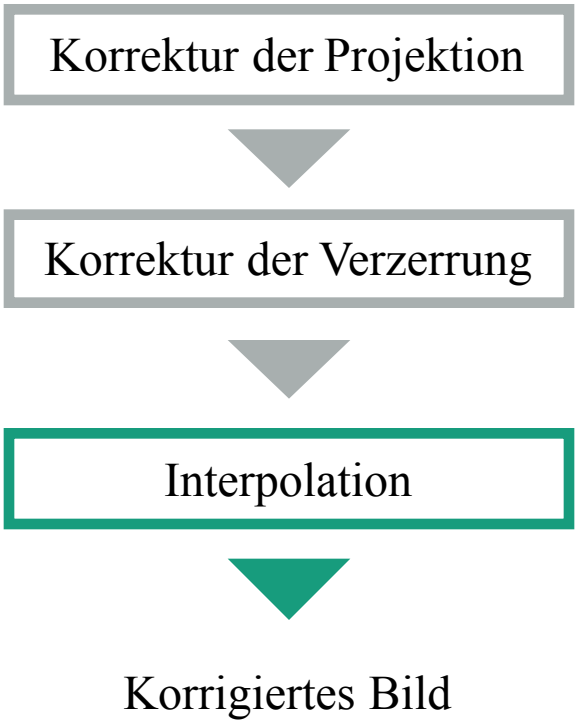
2. Phase der Bildkorrektur: Korrektur der Verzerrung



Modelle zur Verzerrungskorrektur

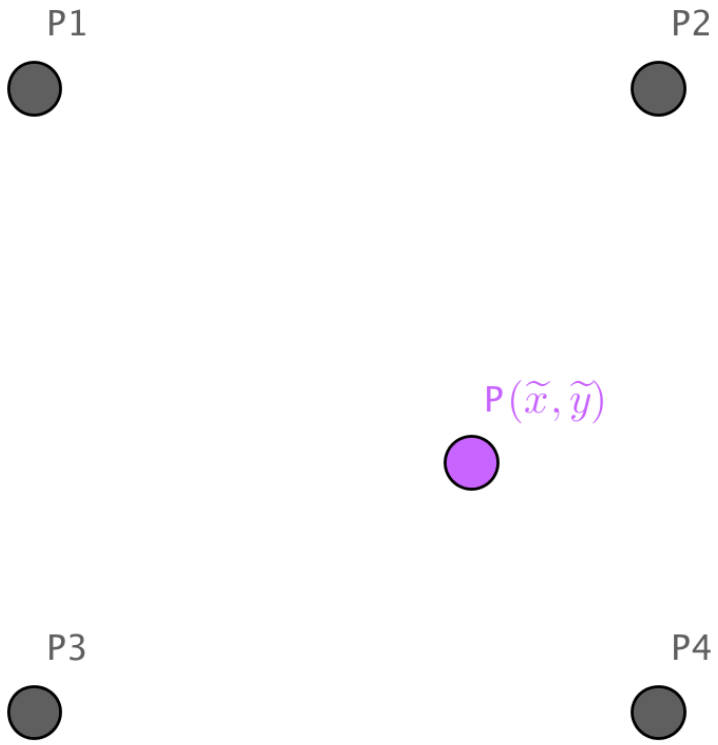
- Unterschied zwischen theoretischer, perfekter Projektion und tatsächlicher Abbildung
- Näherungspolynome für Korrektur:
  - **PTLens**
    - $\boldsymbol{M(r) = r'/r = a \cdot r^3 + b \cdot r^2 + c \cdot r + (1 - a - b - c)}$
  - **Poly3**
    - $\boldsymbol{M(r) = r'/r = b \cdot r^2 + (1 - b)}$
  - **Poly5**
    - $\boldsymbol{M(r) = r'/r = k_2 \cdot r^4 + k_2 \cdot r^2 + 1}$
    - Vorteil: Verzicht auf Wurzelziehen für r $= \sqrt{x^2 + y^2}$

3. Phase der Bildkorrektur: Interpolation



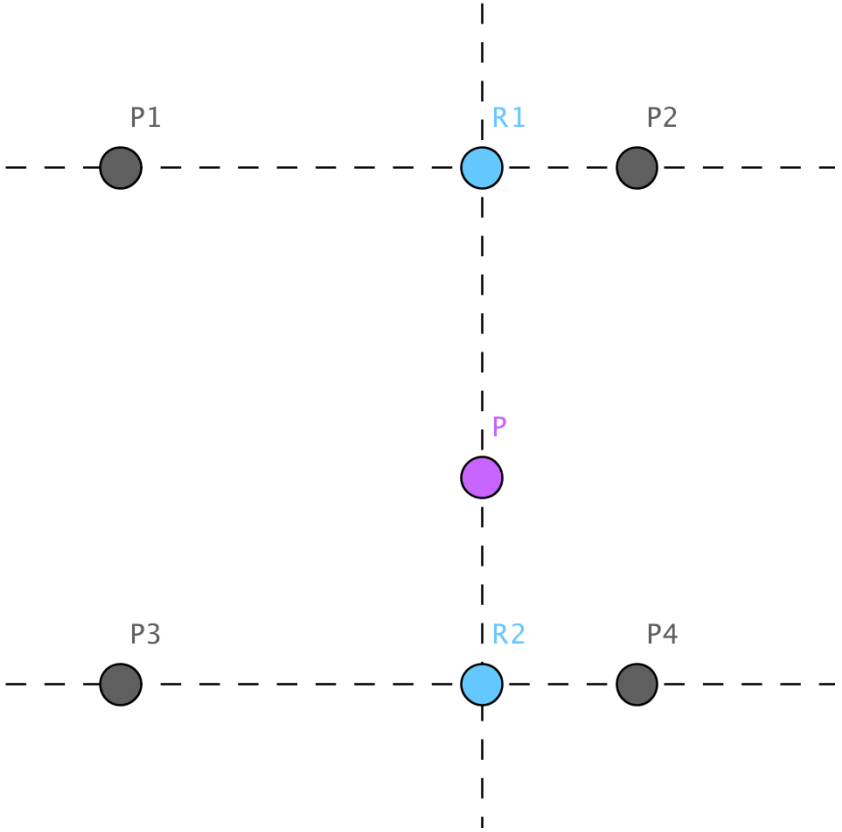
Interpolation

- Algorithmus liefert für jedes Pixel (x,y) Quell-Koordinaten $(\tilde{x}, \tilde{y})$
- $(\tilde{x}, \tilde{y})$: im Allg. nicht ganzzahlig
  - Interpolation
- Einfachste Variante: Nearest Neighbour

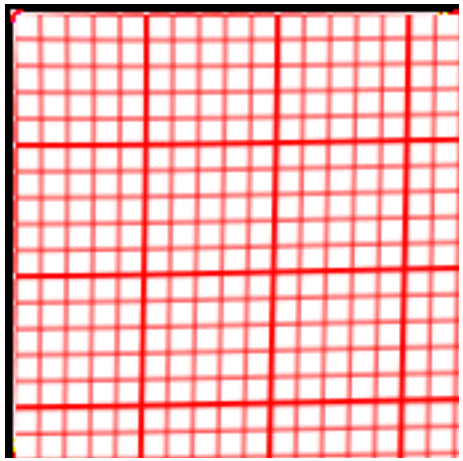


Bilineare Interpolation

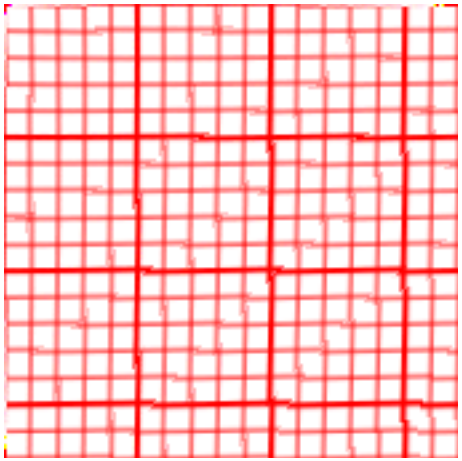
- 3 lineare Näherungen:
  - P1, P2 ⟶ R1
  - P3, P4 ⟶ R2
  - R1, R2 ⟶ P
- Guter Kompromiss zwischen Bildqualität und Rechenaufwand



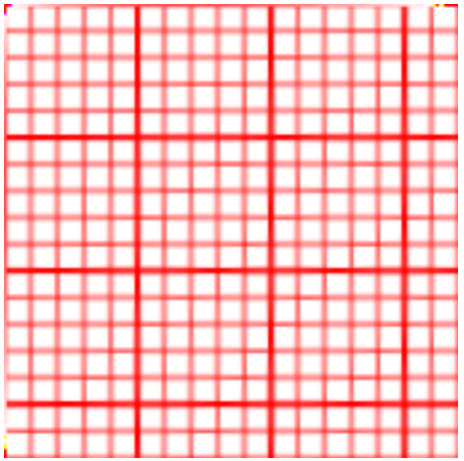
Nearest Neighbour vs Bilineare Interpolation



Verzerrtes Bild



Korrektur + Nearest Neighbour Interpolation



Korrektur + Bilineare Interpolation

Der Code für die Bildkorrektur

```
for y in range(row):
    for x in range(col):

      x_scaled = (x-col/2)*norm_scale
      y_scaled = (y-row/2)*norm_scale

      r = sqrt (x_scaled * x_scaled + y_scaled * y_scaled) * inv_dist;

      if(r == 0.0):
        theta = 1.0 # avoid division by zero
      else:
        theta = atan(r)/r

      x_scaled = x_scaled * theta
      y_scaled = y_scaled * theta

      r2 = x_scaled * x_scaled + y_scaled * y_scaled
      r = sqrt(r2)
      m = a *r*r2 + b * r2 + c * r + d

      new_x = x_scaled * m
      new_y = y_scaled * m

      new_x /= norm_scale
      new_y /= norm_scale

      new_x += col/2
      new_y += row/2

      dest_array[y,x] = bilinear_interpolation(pixel_arr, new_x, new_y)
```

Skalierung

Änderung der Projektion

Korrektur der Verzerrung

Skalierung

Optimierungsansatz: Vorausberechnung der Koeffizienten

- Projektions-Korrektur, Verzerrungs-Korrektur und Interpolation lassen sich zusammenfassen (O: Output-Bild, I: Input-Bild):

$$O_{x,y} = I_{x',y'} \cdot a_0 + I_{x',y'+1} \cdot a_1 + I_{x'+1,y'} \cdot a_2 + I_{x'+1,y'+1} \cdot a_3$$

- Für jedes Pixel: einmalig Berechnen und Abspeichern von:
  - Neue Basiskoordinaten x', y'
  - Alternativ: Abweichung speichern: $\Delta x = x - x'$
  - Koeffizienten a_0, a_1, a_2, a_3
- Vorteil: schnell, Linsen-Modell irrelevant
- Nachteil: Speicherbedarf

Speicherbedarf für Koeffizienten

- Software: kein Problem
  - Speicherbedarf: 6 x 4 Byte pro Pixel (Single Precision FP)
  - 24 MByte/MPixel
  - Daten passen in Hauptspeicher
  - Auch auf FPGA realisierbar?

Ansatz für Hardware

- Begrenzter Speicher erfordert anderen Ansatz
- Variante 1: Implementierung ohne Vorausberechnung
  - $\times,\ tan^{-1}, sin^{-1}$ und $\sqrt{\ }$ in HW
  - CORDIC oder Tabellen für Winkelfunktionen notwendig

Ansatz für Hardware

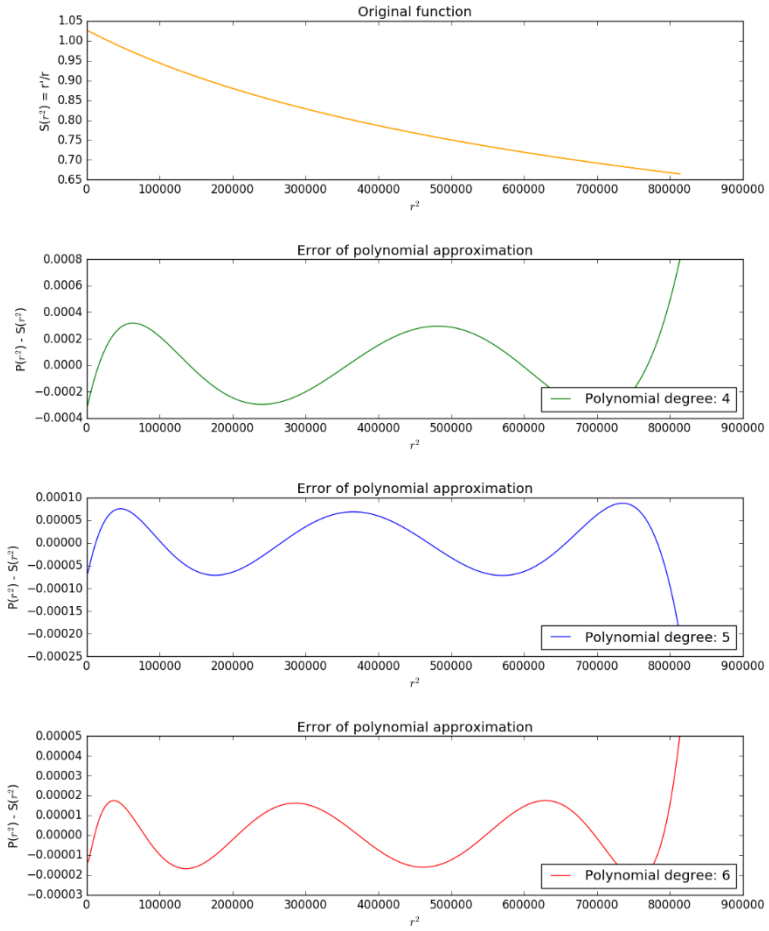
- Variante 2: Vorausberechnung des Skalierungsfaktors
  - Algorithmus skaliert letztendlich nur den Radius jedes Punktes
  - Durch Zusammenfassen der Berechnungsschritte ergibt sich z.B.:
    - $S(r) = \frac{r'}{r} = \left(\frac{\operatorname{atan}(\tilde{r})}{\tilde{r}}\right)^3 a + \left(\frac{\operatorname{atan}(\tilde{r})}{\tilde{r}}\right)^2 b + \left(\frac{\operatorname{atan}(\tilde{r})}{\tilde{r}}\right) c + d$

      mit normiertem Radius $\tilde{r} = r \cdot \frac{norm_{\downarrow}scale}{f}$
  - Mit Skalierungsfaktor können Quellkoordinaten bestimmt werden:

    $(x', y') = (x\frac{r'}{r}, y\frac{r'}{r})$
  - Anschließend: Interpolation
  - $S(r)$ kann nicht für jeden relevanten Radius r abgespeichert werden

    ⇒ Näherung durch Polynom oder Tabelle

Näherung von $S(r^2)$ durch Polynome



- Curve Fitting mit Python (Methode der kleinsten Fehlerquadrate)
- Vergleich zu Tabelle + linearer Interpolation:
  - kein Block-RAM notwendig
  - dafür größerer DSP-Bedarf
- Offene Fragen
  - Welcher Fehler ist klein genug, damit Unterschied nicht auffällt?
  - Wieviel Bit Genauigkeit bei Berechnungen?

Stand der Arbeit

- Erledigt
  - Literaturrecherche zu NEON, Bildkorrektur
  - Algorithmus aus Lensfun Quellcode rekonstruieren
  - Projektion + Verzerrungskorrektur + Interpolation in C implementiert
  - Ansatz für SW-Optimierung: Koeffizienten vorausberechnen
  - Ansatz für HW-Implementierung: Polynom
- TODO
  - C-Implementierung für NEON anpassen
  - VHDL-Implementierung
  - Integration in Pipeline
  - Dokumentation fertigstellen

Quellen

- Fisheye-Bild
  - peña [1]: Carlos F. Peña, CC BY-NC-SA 2,0, https://www.flickr.com/photos/cfpg/6643416977/ (Canon EOS 7D, Sigma Fisheye 2.8 10 mm)
- Literatur
  - Dokumentation zu Lensfun (http://lensfun.sourceforge.net/manual/corrections.html)