

# HSD

HOCHSCHULE DÜSSELDORF

**HAUSARBEIT IM FACH  
TECHNISCHES RAYTRACING  
BEI PROF. ALEXANDER BRAUN**

**Erweiterung des PBRT um  
optische Verzerrung**

Jan Kiene, Philipp Weber und Christian Wittpahl

20. Februar 2019

# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>2</b>
<b>2. Theorie</b>	<b>2</b>
2.1 Was ist ein Raytracer? . . . . .	2
2.2 Verzerrung . . . . .	3
2.3 Lochkameramodell . . . . .	3
2.4 Integration der Verzerrung in den Raytracer . . . . .	4
2.5 Verzerrungsmodelle . . . . .	4
<b>3. Implementierung</b>	<b>7</b>
3.1 Verzerrungsmodelle in der Lensfun-Datenbank . . . . .	7
3.1.1 Koordinatensystem und Normalisierung . . . . .	7
3.1.2 Modelldefinitionen . . . . .	9
3.2 Implementierung . . . . .	9
3.2.1 Least Squares . . . . .	10
3.2.2 Anwenden des Modells . . . . .	11
3.2.3 Umsetzung in C++ . . . . .	11
3.3 Interface in der Szenenbeschreibung . . . . .	13
<b>4. Verifizierung und Ergebnisse</b>	<b>14</b>
4.1 Planung . . . . .	14
4.2 Implementierung in Matlab . . . . .	14
4.3 Region-Of-Interest . . . . .	15
4.4 Vergleich . . . . .	16
<b>5. Ergebnisse, Diskussion und Ausblick</b>	<b>18</b>

## 1. Einleitung

Im Rahmen der vorliegenden Arbeit wurde ein Modell in den Raytracer pbtrt integriert, welches die Verzerrungen durch reale Objektive simuliert. Liegen gemessenen Verzerrungsparameter einer Optik vor, können diese vor dem Rendervorgang übergeben werden und es wird eine physikalisch korrekte Berechnung dieser Verzerrung durchgeführt.

Alle Ergebnisse und die genutzten Test-Szenen finden sich in folgendem Git-Repository:  
[7]

## 2. Theorie

### 2.1 Was ist ein Raytracer?

Dazu muss zu Erst der Begriff *Rendering* erklärt werden. Dieser beschreibt ganz Allgemein den Prozess, der aus der Beschreibung einer dreidimensionalen Szene ein zweidimensionales Bild erzeugt. Dazu sind zahlreiche Algorithmen notwendig, welche beispielsweise Geometrien modellieren, Objekte animieren oder Texturen erzeugen und die Ergebnisse hiervon an den Render-Prozess weitergeben, um diese sichtbar zu machen.

Um eine dreidimensionale Szene in ein Bild abzubilden, kann sich des Raytracing Algorithmus bedient werden. Wie sich bereits aus dem Begriff ableiten lässt, werden dabei die Pfade einzelner „Lichtstrahlen“ verfolgt, bis diese auf eine Oberflächen treffen und es beispielsweise zu einer Reflexion kommt. In diesem Fall würden nun die Richtungen und Intensitäten der reflektierten Strahlen bestimmt und diese weiter verfolgt.

Da dabei nur diejenigen Strahlen von Interesse sind, welche auch tatsächlich auf den virtuellen Bildsensor treffen und für den Helligkeits- und Farbwert eines Pixels im Bild einen Beitrag leisten, verfolgen Raytracer die Strahlen in umgekehrter Richtung. Es werden also Strahlen am Bildsensor erzeugt und in die Szene geschickt, bis sie mit einer Oberfläche interagieren. Auf diese Weise werden alle Lichtquellen identifiziert, die für die Berechnung des Helligkeits- und Farbwertes des gerade betrachteten Pixels relevant sind [12].

Der in dieser Arbeit genutzte Raytracer ist *pbtrt*[1]. Dieser hat den Anspruch das Rendering *physically based*, also möglichst physikalisch korrekt, durchzuführen. Aus diesem Grund sind hier viele fundamentale Gleichungen der Optik implementiert und Phänomene werden somit korrekt berechnet. Im Gegensatz dazu stehen Raytracer, welche für Animationsfilme oder Computerspiele genutzt werden. Hier soll das Endergebnis möglichst *gut* aussehen, ohne unbedingt physikalisch korrekt zu sein.

Trotz des Anspruchs von pbtrt, möglichst physikalisch korrekte Resultate zu erzeugen, sind manche optische Phänomene noch nicht implementiert. So besteht im Moment noch keine Möglichkeit die optischen Verzerrungen nachzubilden, welche von allen realen Optiken verursachte werden.

In der vorliegenden Arbeit soll ein neues Kameramodell in den frei verfügbaren Quellcode von pbtrt eingefügt werden, welches gemessenen Verzerrungen von Optiken berechnet.

## 2.2 Verzerrung

Die Verzerrungen, welche von realen Optiken verursacht werden, führen dazu, dass in der Realität ungekrümmte Linien in der zweidimensionalen Abbildung gekrümmt erscheinen. Dies geschieht dadurch, dass Punkte im aufgenommenen Bild verschoben abgebildet werden und zwar umso stärker, je weiter sie von der optischen Achse entfernt sind. Besonders gut lässt sich dieser Effekt mit Weitwinkelobjektiven an Hausfassaden beobachten.

Genauer wird dabei zwischen Kissen- und Tonnenverzeichnung unterschieden. In Abbildung 1 und 2 wird deutlich, dass bei der Kissenverzeichnung (auch positive Verzeichnung) die Bildpunkte von der optischen Achse weg verschoben werden, während der Abstand der Bildpunkte zur optischen Achse bei der Tonnenverzeichnung (auch negative Verzeichnung) kleiner wird. Durch die bereits erwähnte Zunahme dieses Effekts mit steigendem Radius zur optischen Achse, erfahren in der Realität gerade Linien eine Krümmung [13].

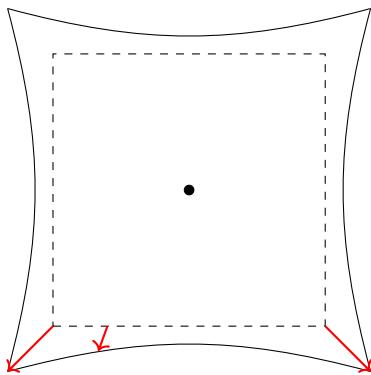


Abbildung 1: Kissenverzeichnung

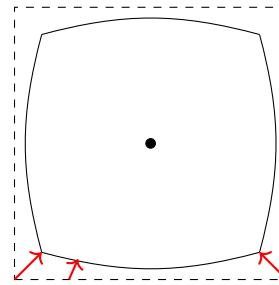


Abbildung 2: Tonnenverzeichnung

## 2.3 Lochkameramodell

Das einfachste Kameramodell, welches beim Rendern von Bildern genutzt werden kann, ist das Lochkameramodell, welches in Abbildung 3 dargestellt ist.

Alle Lichtstrahlen, welche auf den Sensor treffen, müssen die Blende durch das infinitesimal kleine Loch passieren. Durch Größe des Sensors und dessen Abstand zur Blende wird das Sichtfeld (auch FOV, Field of view) festgelegt.

Durch die Anwendung der Ähnlichkeitssätze für Dreiecke, kann aus den Koordinaten des Gegenstands (rechts der Lochblende) auf die Bildkoordinaten auf dem Sensor (links der Lochblende) geschlossen werden, wie in Abbildung 3 für die x-Koordinate dargestellt ist [12]:

$$x_B = \frac{x_G \cdot b}{g} \quad (2.1)$$

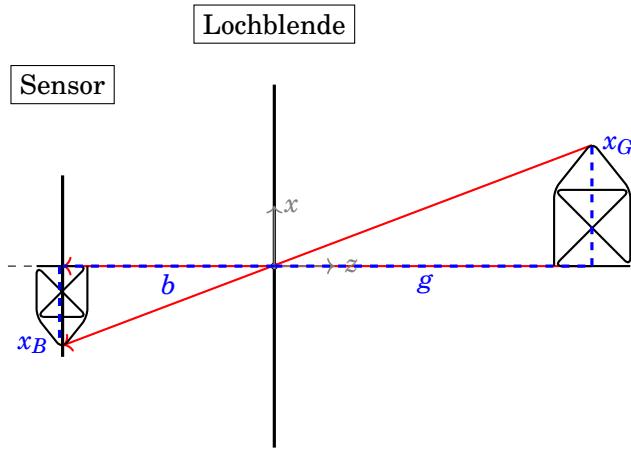


Abbildung 3: Lochkamera-Modell im zweidimensionalen Fall

Wie bereits im Kapitel 2.1 erwähnt, treffen bei der Anwendung des Kameramodells im Raytracer jedoch keine Lichtstrahlen von außen in die Kamera. Vielmehr werden Lichtstrahlen am Sensor erzeugt und nach außen in die Szene geschickt, um nur diejenigen Strahlen zu berechnen, welche tatsächlich einen Beitrag für das zu rendernde Bild leisten.

## 2.4 Integration der Verzerrung in den Raytracer

Die Auswirkungen einer Linsenverzerrung sind in Abbildung 4 dargestellt. Der Lichtstrahl bewegt sich nicht mehr ungehindert in seiner Ausbreitungsrichtung weiter und trifft an einem Punkt  $p_1$  auf den Kamerasensor (siehe blauer Strahl), sondern wird an der Öffnung der Lochblende in einem bestimmten Maße abgeknickt und erreicht den Sensor am Punkt  $p_2$  (roter Strahl). Die Funktion, die die normalisierten Pixelkoordinaten des unverzerrten Punktes  $p_1$  auf die des verzerrten Punktes  $p_2$  abbildet, bezeichnen wir mit  $f$ .

Ein Raytracer, der eine solche Linsenverzerrung berücksichtigen soll, muss also bei der Erzeugung der Strahlen für einen gegebenen Punkt von dem Lochkamera-Modell aus Abschnitt 2.3 abweichen.

Allerdings gibt es für jeden Punkt  $p$  auf dem Sensor einen Punkt  $p'$ , deren zugeordneter Strahl für  $z \geq 0$  *nach dem Lochkamera-Modell* dem Strahl entspricht, der *nach dem Verzerrungsmodell*  $p$  zuzuordnen ist. In Abbildung 4 ist  $p = p_2$  und  $p' = p_1$ . Da  $f(p_1) = p_2$  ist, kann damit  $p'$  durch Anwenden der inversen Verzerrungsfunktion  $g = f^{-1}$  mit  $p' = g(p)$  gefunden werden. Der Strahl für  $p$  kann dann einfach nach dem Lochkamera-Modell berechnet werden, indem  $p = p'$  gesetzt wird.

## 2.5 Verzerrungsmodelle

In der Praxis werden häufig radialsymmetrische Modelle genutzt, um die in Kapitel 2.4 beschriebene Verzerrung zu beschreiben. Da auch der Großteil der im Internet frei verfügbaren Messdaten von realen Objektiven auf solchen radialsymmetrischen Modellen

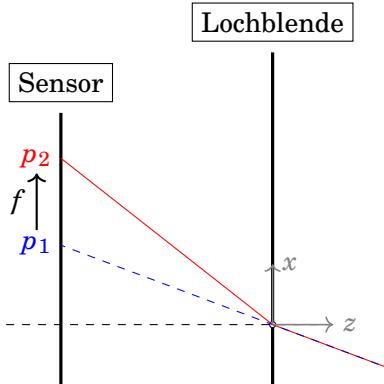


Abbildung 4: Vereinfachtes Verzerrungsmodell in 2D. Der rote von rechts einfallende Lichtstrahl wird durch die Linse abgelenkt. In blau ist der Weg dargestellt, den der Strahl nach dem Lochkamera-Modell nehmen würde. Die gestrichelte Linie zeigt die optische Achse. Die Abbildung zwischen Lochkamera-Modell und Modell mit Verzerrung ist mit  $f$  bezeichnet.

basiert, werden in dieser Arbeit nur solche berücksichtigt.

Aus der Radialsymmetrie folgt, dass die Funktion  $f(p_1)$  lediglich von einem Radius  $r$  abhängt. Zuerst muss dafür ein Verzerrungszentrum  $(x_c, y_c)$  festgelegt werden, welches dem Bildmittelpunkt entspricht, wenn dessen genaue Lage nicht messtechnisch bestimmt wurde. Nach Festlegung des Verzerrungszentrums kann der Radius  $r$  folgendermaßen bestimmt werden:

$$r = \sqrt{(x - x_c)^2 + (y - y_c)^2} \quad (2.2)$$

Die Transformation zwischen zwei Radien ( $r_1$ ) und ( $r_2$ ), kann nun durch Polynome der folgenden Form erfolgen [10]:

$$r_2 = f(r_1) = r_1(k_0 + k_1 r_1 + k_2 r_1^2 + \dots) \quad (2.3)$$

Eine solche Transformation ist in Abbildung 5 dargestellt und resultiert hier in einer kissenförmigen Verzeichnung. Die Koeffizienten  $k_n$  sind in diesem Fall positiv.

Die später im Kapitel 3.1.2 vorgestellten Polynome sind teilweise dahingehend optimiert, dass sich der maximal mögliche unverzerrte Radius durch das Verzerrungsmodell nicht verändert. So wird eine unnötige Skalierung des gesamten Bildes vermieden [9].

Generell kann mit jedem Modell sowohl verzerrt, als auch entzerrt werden. Soll jedoch mit einem Verzerrungsmodell entzerrt werden, müssen die durch Messungen bestimmten Koeffizienten neu erstellt oder eine Umkehrfunktion gebildet werden. Die Koeffizienten legen also damit fest, in welche „Richtung“ ein Modell funktioniert. Da, wie in Kapitel 3.2 beschrieben, keine analytische Umkehrfunktion gefunden werden kann, sind numerische Methoden nötig, um eine solche Invertierung einer Ver- oder Entzerrung zu erreichen.

Aus den Überlegungen am Ende des Kapitels 2.4 folgt, dass eine real gemessene Verzerrungsfunktion beim Einsatz im Raytracer invertiert werden muss, um hier die

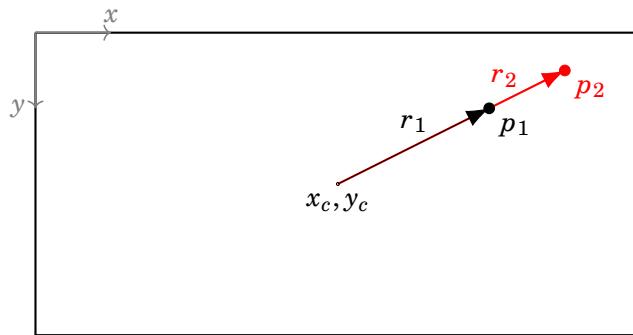


Abbildung 5: Exemplarische Verschiebung des Punktes  $p_1$  auf Punkt  $p_2$  durch ein radiales Verzerrungsmodell

korrekte Verzerrung zu berechnen. Ohne Invertierung würde sich die Art der Verzerrung (kissen-/tonnenförmig) umkehren. Die gemessene tonnenförmige Verzeichnung eines realen Objektivs würde also im Raytracer fälschlicherweise zu einer kissenförmigen Verzerrungen führen und umgekehrt.

Mit radialen Modellen nach Gleichung 2.3 lässt sich die Verzeichnung von Objektiven, wie sie beispielsweise in der Fotografie genutzt werden, auch schon mit sehr wenigen Koeffizienten mit einer ausreichenden Genauigkeit korrigieren. Um eine höhere Genauigkeit zu erreichen, kann eine tangentiale Komponente hinzugefügt werden, da reale Verzerrungen üblicherweise nicht völlig radialsymmetrisch sind.

### 3. Implementierung

Zur Simulation einer zuvor gemessenen Linsenverzerrung wurde in pbrt eine neue Kameraklasse `DistortionCamera` hinzugefügt. Die vorliegende Implementierung unterstützt drei rein radiale Verzerrungsmodelle, die von der Datenbank des Lensfun-Projektes[2] (Version 0.3.2<sup>1</sup>) unterstützt werden. Lensfun ist eine Open-Source Softwarebibliothek, die eine Datenbank vermessener Kameras und Linsen sowie Funktionen zur Korrektur von Linsenverzerrungen, chromatischen Aberrationen und Vignettierung bereitstellt.

#### 3.1 Verzerrungsmodelle in der Lensfun-Datenbank

Die in Lensfun implementierten Modelle bilden rein *radiale* Linsenverzerrungen ab, wie in Abschnitt 2.5 beschrieben. Dazu wird eine Funktion  $f : [0, 1] \rightarrow R$  definiert<sup>2</sup>, die den unverzerrten Radius auf den verzerrten Radius abbildet. Dabei wird in einem normalisierten Koordinatensystem gearbeitet.

##### 3.1.1 Koordinatensystem und Normalisierung

Der Radius wird definiert als der normierte Abstand vom Bildmittelpunkt. Dieser wird standardmäßig als  $p_m = (\frac{x_{res}}{2}, \frac{y_{res}}{2})$  angenommen, wobei  $x_{res}$  und  $y_{res}$  die Bildauflösung in x- bzw. y-Richtung bezeichnen. In der Datenbank kann auch eine gemessene Abweichung von diesem „idealen“ Bildmittelpunkt angegeben werden. Wir behandeln zunächst den Fall, dass keine Abweichung gegeben ist, bzw. die Abweichungen in x- und y-Richtung null betragen. Die Normierung erfolgt so, dass die halbe Bilddiagonale  $r_e$  die Länge eins hat. Da diese den größten Radius im Bild darstellt, ist dadurch immer  $r \in [0, 1]$  (siehe Abbildung 6).

Die Transformation von Pixelkoordinaten zu normalisierten Koordinaten besteht einfach in einer Division durch  $r_e$ . Dies lässt sich einfach nachvollziehen, indem der Radius (in Pixelkoordinaten) für einen der Eckpunkte berechnet wird:

$$r_e = \sqrt{(x_{eck} - x_{mitte})^2 + (y_{eck} - y_{mitte})^2} = \sqrt{\bar{x}^2 + \bar{y}^2} \quad (3.4)$$

Skaliert man nun alle Koordinaten mit einem Faktor  $k$ , so ist der skalierte Eckenradius

$$r_s = \sqrt{(k\bar{x})^2 + (k\bar{y})^2} = \sqrt{k^2(\bar{x}^2 + \bar{y}^2)} = kr_e. \quad (3.5)$$

Um  $r_s = 1$  zu erfüllen, muss dann  $k = \frac{1}{r_e}$  gelten.

Im Fall, dass eine Abweichung der optischen Achse vom Bildmittelpunkt ungleich null vorliegt, stellt sich für die Normalisierung das Problem, dass nun der maximal mögliche Radius nicht mehr durch die halbe Bilddiagonale gegeben ist, sondern durch den Abstand von  $p_m$  zur am weitesten entfernten Ecke. Mit welcher Ecke dann  $r_e$  berechnet werden

---

<sup>1</sup>Zum Zeitpunkt der Abgabe war diese die aktuellste stabile Version von Lensfun

<sup>2</sup>Der tatsächliche Wertebereich der Funktionen hängt von ihren Parametern ab (siehe 3.1.2). Diese werden durch Messung an realen Kameras ermittelt. Typische Werte sorgen dafür, dass der Wertebereich zwischen 0 und 1 liegt, was im Kontext eines normierten Radius sinnvoll ist (siehe 3.1.1).

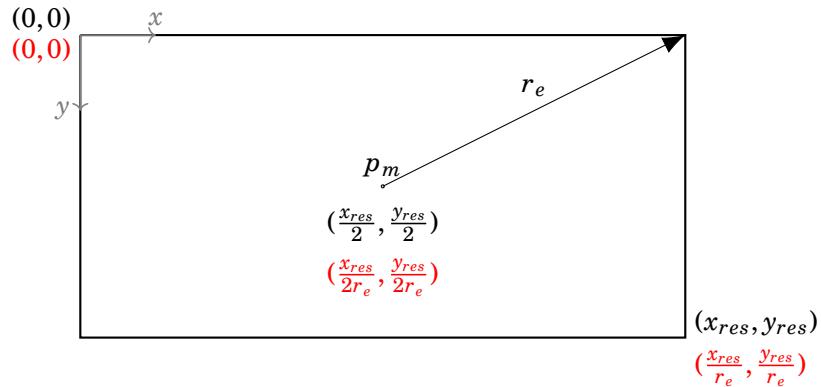


Abbildung 6: Verwendete Normalisierung der Bildkoordinaten nach [11, 3]. Das Koordinatensystem wurde in Übereinstimmung mit [12, S. 359] gewählt. Pixelkoordinaten in schwarz, normalisierte Koordinaten in rot.

muss, hängt von den Vorzeichen der Abweichungen  $x_a$  und  $y_a$  ab. Anschließend erfolgt die Normalisierung wie oben beschrieben. Die gemessenen Werte für  $x_a$  und  $y_a$  werden in der Lensfun-Datenbank ebenfalls normalisiert gespeichert, wobei die Skalierung dadurch definiert wird, dass die kleinere Bilddimension gleich 2 gesetzt wird. Die entsprechenden Pixelwerte können also durch

$$\begin{pmatrix} x_{a,pixel} \\ y_{a,pixel} \end{pmatrix} = \frac{\min\{x_{res}, y_{res}\}}{2} \cdot \begin{pmatrix} x_{a,norm} \\ y_{a,norm} \end{pmatrix} \quad (3.6)$$

berechnet werden.

### 3.1.2 Modelldefinitionen

Im Folgenden steht  $r_u$  immer für den normierten unverzerrten Bildradius, der durch eine Verzerrungsfunktion  $f$  auf den verzerrten Radius  $r_d$  abgebildet wird. Diese Bezeichnungen und die Definitionen der Verzerrungsfunktionen sind der Dokumentation von Lensfun entnommen [3].

**Poly3:** Das Poly3-Modell besteht aus einem Polynom dritter Ordnung mit einem Parameter  $k_1$ . Es gilt unabhängig von  $k_1$  immer  $f(0) = 0$  und  $f(1) = 1$ . Dadurch, dass nur ein Parameter vorhanden ist, eignet sich das Modell vornehmlich zur Modellierung weniger komplexer Verzerrungen.

$$f_{poly3}(r_u) = r_u \cdot (1 - k_1 + k_1 r_u^2) \quad (3.7)$$

**Poly5:** Dieses Modell verwendet ein Polynom fünfter Ordnung und kann dadurch komplexere Verzerrungen nachbilden als das Poly3-Modell. Es wird dafür ein weiterer Parameter benötigt. Anders als beim Poly3-Modell ist hier nicht  $f(1) = 1$  garantiert.

$$f_{poly5}(r_u) = r_u \cdot (1 + k_1 r_u^2 + k_2 r_u^4) \quad (3.8)$$

**PTLens:** Dieses Modell wurde aus der PTLens-Datenbank, auf der das Lensfun-Projekt aufbaut, übernommen. Das Modell besteht aus einem Polynom vierter Ordnung, mit drei Parametern  $a, b, c$  und ist so aufgebaut, dass wieder die Normierung  $f(1) = 1$  gilt.

$$f_{ptlens}(r_u) = r_u \cdot (ar_u^3 + br_u^2 + cr_u + 1 - a - b - c) \quad (3.9)$$

## 3.2 Implementierung

Die Klasse `DistortionCamera` implementiert die Linsenverzeichnung. Dabei werden die Richtungen der ausgesandten Strahlen relativ zum „Referenzmodell Lochkamera“ modifiziert, sodass die Verzeichnung durch die unperfekte Linse im gerenderten Bild sichtbar wird (vgl. Abschnitt 2.4). Der Raytracer ruft zum Erzeugen der Strahlen die Methode `GenerateRay` auf. Dies passiert für jeden Pixel des zu erzeugenden Bildes mindestens einmal<sup>3</sup>. Die Berechnung der Strahlrichtung in Abhängigkeit der betrachteten Pixelkoordinaten erfolgt also in `GenerateRay`.

Damit ergibt sich für die Implementierung die Anforderung, dass bei einer vorgegebenen Verzerrungsfunktion  $f$  vor Beginn des Renderings die inverse Funktion bestimmt werden muss. Da für die in der Lensfun-Datenbank verwendeten Modelle keine einfache analytische Inverse gefunden werden kann, wird ein numerischer Ansatz verfolgt. Dazu werden zunächst Werte von  $f$  für eine Reihe von  $r_i$  im erlaubten Bereich bestimmt. Die Inverse wird dann als Polynom mittels des Least Squares Verfahrens angenähert.

---

<sup>3</sup>Der verwendete Sampler gibt vor, wie häufig und auf welche Art und Weise für ein Pixel Strahlen generiert werden.

### 3.2.1 Least Squares

Die Methode der kleinsten Quadrate (engl.: *Least Squares*, kurz LS) stellt ein Verfahren, zu einer gegebenen Menge an Punkten eine Funktion zu finden, die eine mögliche unbekannte Gesetzmäßigkeit hinter der Lage der Punkte annähert [14].

Sei  $P = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$  die Menge der bekannten Punkte mit  $x_i, y_i \in R$ . Man nimmt nun an, dass zwischen  $x$  und  $y$  ein funktionaler Zusammenhang bestehe, der durch  $m$  reelle Parameter  $a_j$  bestimmt wird. Damit lässt sich  $y_i = f(x_i; \vec{a})$  schreiben, wobei  $\vec{a} \in R^m$  die Parameter zusammenfasst. LS versucht nun, den unbekannten Parametervektor so zu ermitteln, dass die Summe der quadratischen Abweichungen  $S^2$  minimiert wird:

$$\vec{a} \in \underset{\vec{a} \in R^M}{\operatorname{argmin}} \{S^2\}, \quad S^2 = \sum_{i=1}^N (y_i - f(x_i, \vec{a}))^2 \quad (3.10)$$

Setzt man für  $f$  ein Polynom vom Grad  $k$  an, so ist  $m = k + 1$  und

$$f(x_i, \vec{a}) = a_0 + a_1 x_i + \dots + a_k x_i^k = \sum_{j=0}^k a_j x_i^j \quad (3.11)$$

Für das Minimum müssen die partiellen Ableitungen nach allen Parametern null sein.

$$\frac{\partial S^2}{\partial a_l} = -2 \sum_{i=1}^n (y_i - a_0 + a_1 + \dots + a_k x^k) x^l = 0, \quad 0 \leq l \leq m \quad (3.12)$$

Es muss also ein Gleichungssystem gelöst werden, dass sich wie folgt darstellen lässt:

$$X^T X \cdot \vec{a} = X^T \cdot \vec{y} \quad (3.13)$$

mit

$$X = \begin{bmatrix} 1 & x_1 & \dots & x_1^k \\ 1 & x_2^k & \dots & x_2^k \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \dots & x_n^k \end{bmatrix} \text{ und } \vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad (3.14)$$

Sind alle  $x_i$  paarweise voneinander verschieden, so ist  $X^T X$  regulär. Damit ist das Problem unter dieser Bedingung immer eindeutig lösbar. Die Matrixgleichung 3.13 kann somit numerisch oder direkt durch Finden von  $(X^T X)^{-1}$  gelöst werden [15]. In der Implementierung von DistortionCamera geschieht dies mittels LR-Zerlegung (beschrieben zum Beispiel in [8]).

Ist also eine Funktion  $f(x)$  gegeben, so kann durch Auswahl von  $n$  paarweise unterschiedlichen Werten  $x_i$  im Bereich  $[0, 1]$  die Menge  $P = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ ,  $y_i = f(x_i)$  erzeugt werden. Um nun die inverse Funktion  $g(x) = f^{-1}(x)$  per LS anzunähern, tauscht man in den Paaren in  $P$  jeweils  $x_i$  und  $y_i$  und führt dann das LS Verfahren wie oben beschrieben durch. Dies entspricht dem Optimierungsproblem

$$\vec{a} \in \underset{\vec{a} \in R^m}{\operatorname{argmin}} \{S^2\}, \quad S^2 = \sum_{i=1}^n (x_i - f(y_i, \vec{a}))^2 \quad . \quad (3.15)$$

Im Vergleich zu Gleichung 3.10 sind also die Rollen von  $x_i$  und  $y_i$  getauscht.

### 3.2.2 Anwenden des Modells

Sind die Parameter für  $g$  bestimmt, kann damit  $p_1$  aus  $p_2$  bestimmt werden (vgl. Abbildung 4). Da die implementierten Modelle rein radial arbeiten, muss zunächst zu  $p_2 = (x_2, y_2)$  der Radius  $r_2$  bestimmt werden (die Koordinaten sind hierbei und im Folgenden die nach 3.1.1 normalisierten Koordinaten):

$$r_2 = \sqrt{(x_2 - x_{mitte})^2 + (y_2 - y_{mitte})^2} \quad (3.16)$$

Der Radius  $r_1$  des Punktes  $p_1$  wird dann mit

$$r_1 = g(r_2) \quad (3.17)$$

bestimmt. Zum Berechnen der Koordinaten von  $p_1$  ist es hilfreich, einen radialen Einheitsvektor  $\vec{e}_r$ , der vom Bildmittelpunkt zum betrachteten Punkt  $p_2$  zeigt, zu definieren:

$$\vec{e}_r = \begin{pmatrix} x_2 - x_{mitte} \\ y_2 - y_{mitte} \end{pmatrix} \cdot \frac{1}{r_2} \quad (3.18)$$

Damit ergeben sich die Koordinaten von  $p_1$  zu

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_{mitte} \\ y_{mitte} \end{pmatrix} + \vec{e}_r \cdot r_1 = \frac{r_1}{r_2} \cdot \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} x_{mitte} \\ y_{mitte} \end{pmatrix} \cdot \frac{r_1}{r_2} \quad (3.19)$$

### 3.2.3 Umsetzung in C++

Die Implementierung von DistortionCamera ist in den Dateien `distortion.h` und `distortion.cpp` enthalten. DistortionCamera erbt dabei von ProjectiveCamera (siehe Abbildung 7).

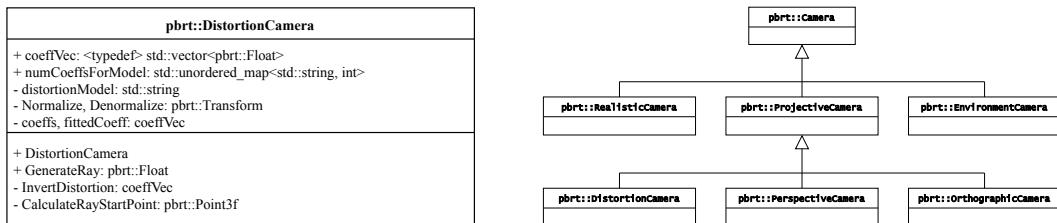


Abbildung 7: UML-Darstellung von `DistortionCamera` (links) und Einordnung in die Klassenstruktur der vorhandenen Kameraklassen (rechts).

Im Konstruktor der Klasse wird zunächst eine Validierung des übergebenen Verzerrungsmodells durchgeführt. Ist kein Modell in der zu rendernden Szene definiert, so wird die inverse Verzerrungsfunktion zu  $f(r) = r$  gesetzt. Das Rendering entspricht dann dem einer PerspectiveCamera. Anschließend wird überprüft, ob das zu übergebende Modell implementiert ist und die korrekte Anzahl an Koeffizienten übergeben wurde. Dies geschieht anhand der Variablen `numCoeffsForModel`. Sind die Eingabeparameter korrekt, wird per LS die inverse Verzerrungsfunktion angenähert.

Außerdem wird im Konstruktor die Transformation `Normalize` definiert (siehe 3.1.1), sowie die dazugehörige inverse Transformation `Denormalize`.

Listing 1: Transformation zur Normalisierung der Pixelkoordinaten

```
/* ----- Define transformations for ray generation -----*/
Float xRes = film->fullResolution.x;
Float yRes = film->fullResolution.y;
Float cornerRadius = sqrt(pow(xRes/2, 2) + pow(yRes/2, 2)) / 2.;
NormalizeToCornerRadius = Scale(1. / cornerRadius, 1. / cornerRadius, 1.);
Denormalize = Inverse(NormalizeToCornerRadius);
```

Zum Bestimmen der inversen Verzerrungsfunktion ist in `distortion.h` die Funktion `fitPolyCoeffs` implementiert. Diese setzt das Least Squares Verfahren wie in Abschnitt 3.2.1 beschrieben um.

Für die Matrizenoperationen wird auf Funktionen aus der Boost-Bibliothek zurückgegriffen. Boost ist eine Sammlung von Open Source C++-Bibliotheken [4]. In diesem Fall wird die Unterbibliothek `Boost::numeric::uBLAS` verwendet [5]. BLAS steht dabei für „Basic Linear Algebra Subprograms“, was Inhalt und Zielsetzung der Bibliothek gut zusammenfasst. Zusätzlich ist eine Funktion `evalPolynomial` vorhanden, die für einen gegebenen Satz an  $m$  Polynomkoeffizienten  $a_i$  und einen Wert  $x \in R$  die Funktion  $y = a_0 + a_1x + a_2x^2 + \dots + a_{m-1}x^{m-1}$  berechnet. In `InvertDistortion` wird die übergebene Verzerrungsfunktion abgetastet und anschließend der Koeffizientenvektor der inversen Funktion genähert.

Listing 2: „Abtasten“ des Verzerrungsmodells und Invertierung

```
// fill sample vector according to the model used
for (int i = 0; i < sampleSize; i++) {
    x[i] = i / scale;
    y[i] = (*modelFunc)(x[i], coeffs);
}
coeffVec polyCoeffs = fitPolyCoeffs(y, x, polyDegree);
```

Zur Erzeugung des Strahls für ein vom Sampler übergebenes `CameraSample` wird dann `CalculateRayStartpoint` in `GenerateRay` aufgerufen. `CalculateRayStartpoint` berechnet den Ausgangspunkt des Strahls für den im aktuellen `CameraSample` übergebenen Punkt auf dem Bildsensor, wie in 3.2.2 beschrieben. Die Funktion `GenerateRay` unterscheidet sich von ihrem Äquivalent in `PerspectiveCamera` nur durch die folgende Zeile:

Listing 3: Anpassung in der Methode `GenerateRay`

```
Point3f pCamera = RasterToCamera(CalculateRayStartpoint(sample));
```

Neben dieser eigentlichen Implementierung der neuen Kameraklasse waren noch geringe Anpassungen nötig, um die Parameter für `DistortionCamera` aus einer `.pbkt`-Datei zu parsen. Diese betrafen die Funktion `MakeCamera` in `/src/core/api.cpp`. Außerdem musste die neue Abhängigkeit von der Boost-Bibliothek in `CMakeLists.txt` eingepflegt werden.

Soll in Zukunft ein neues Verzerrungsmodell in die Klasse eingebaut werden, so sind folgende Punkte zu erledigen:

- Mögliche neue Parameter müssen in der Funktion MakeCamera aus der Szenenbeschreibung geparsed werden und im Konstruktor von DistortionCamera hinzugefügt werden.
- Der Name des neuen Modells und die benötigte Anzahl von Koeffizienten müssen in der Variablen numCoeffsForModel hinzugefügt werden.
- Eine Funktion, die die Modellberechnung durchführt, muss implementiert werden. Die bisherigen Modelfunktionen sind als inline Funktionen in distortion.h implementiert worden.
- Schließlich muss die Modelfunktion in InvertDistortion ausgewählt werden, wenn das neue Modell übergeben wird.

### 3.3 Interface in der Szenenbeschreibung

Die DistortionCamera-Klasse übernimmt alle Argumente von der PerspectiveCamera. Diese sind in [6] bereits dokumentiert. Zusätzlich können folgende Argumente übergeben werden:

Datentyp	Name	Default	Beschreibung
String	model	NO_MODEL	Name des Verzerrungsmodells. Momentan sind verfügbar: „poly3lensfun“, „poly5lensfun“, „ptlens“.
Float	coefficients	[0, 1]	Koeffizientenvektor für das gegebene Verzerrungsmodell.
Float	centerx	0.0	Verschiebung des Bildmittelpunkts in x-Richtung
Float	centery	0.0	Verschiebung des Bildmittelpunkts in y-Richtung

## 4. Verifizierung und Ergebnisse

### 4.1 Planung

Zur Verifizierung der in C++ implementierten Verzerrung, wurde ein Auswertungsskript in MATLAB erstellt. Die Grundidee der Verifizierung ist, dass für einfache Szenen die Verzerrung des Bildes durch Änderung des verfolgten Lichtstrahls, ungefähr äquivalent mit der Interpolation eines unverzerrten Bildes, gerendert durch die PerspectiveCamera des pbrt, sein sollte. Dies wird auf zwei Arten überprüft:

1. Das durch DistortionCamera gerenderte Bild, wird mit dem gegebenen Polynom **entzerrt** und mit dem von PerspectiveCamera gerenderten Bild verglichen
2. Das durch PerspectiveCamera gerenderte Bild wird mit dem gegebenen Polynom **verzerrt** und mit dem von DistortionCamera gerenderten Bild verglichen

Als Grundlage des Tests wird das in Abbildung 8 dargestellte Punkteraster angenommen.

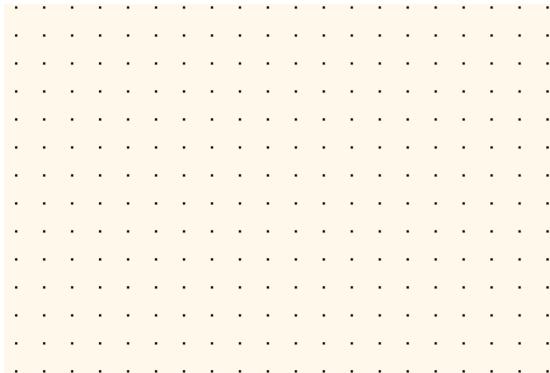


Abbildung 8: Testbild: Punkteraster, gerendert mit pbtrt PerspectiveCamera

Es wird zum Einen die Differenz zwischen den Ergebnissen von pbtrt und Matlab optisch auf systematische Fehler überprüft, zum Anderen wird die Peak-Signal-to-Noise-Ratio (kurz **PSNR**) als quantitative Fehlermetrik genutzt. Diese ergibt sich aus der maximalen Intensität des Bildes  $I_{\max}$  und dem Mean-Square-Error MSE:

$$PSNR = 10 \cdot \log_{10}(I_{\max}^2 / MSE) \quad (4.20)$$

Da die Verzerrung und Entzerrung keine chromatischen Aberrationen mit einbezieht, wird jeder Farbwert mit der gleichen Verzerrung berechnet. Aus diesem Grund werden zum Vergleich lediglich Grauwerte gezeigt.

### 4.2 Implementierung in Matlab

Die Ver- und Entzerrung in Matlab erfolgt durch eine Funktion namens `distortImage`. Diese nimmt als Eingangswerte ein Eingangsbild und ein Array von Polynomkoeffizienten

sowie mehrere optionale Parameter entgegen. Die optionalen Übergabeparameter erlauben ein Verschieben des Bildmittelpunktes und die Auswahl zwischen Ver- und Entzerrung des Bildes mit den gegebenen Koeffizienten.

Als erster Schritt werden jedem Bildpunkt die normierten X und Y Koordinaten zugewiesen und in ein Format gebracht, welches der Interpolant entgegen nimmt:

```

1 %% Bestimmung der normierten X und Y-Koordinaten mit Center-Offset
2 xRange = scale*((-(Lx-1)/2:(Lx-1)/2)-cx);
3 yRange = scale*((-(Ly-1)/2:(Ly-1)/2)-cy);
4 [X,Y] = meshgrid(yRange,xRange);
```

Anschließend werden für jeden Bildpunkt neue Koordinaten nach Verzerrung durch das gegebene Polynom bestimmt:

```

1 %% Berechne das Verzerrungspolynom und verzerrte XY Koordinaten
2 function [distortedX,distortedY] = distortXY(X,Y,polyCoefficients)
3 R = sqrt(X.^2+Y.^2);
4 R_factor = zeros(size(R));
5 for index = 1:numel(polyCoefficients)
6 exponent = index-1;
7 R_factor = R_factor+polyCoefficients(index)*R.^exponent;
8 end
9 distortedX = X.*R_factor;
10 distortedY = Y.*R_factor;
11 end
```

Je nach Ver- oder Entzerrung wird von den normierten in die verzerrten Koordinaten interpoliert oder umgekehrt.

```

1 %% Berechnung der X,Y Position nach Ver/Entzerrung
2 if strcmp(mode,'distort')
3 img_vector = reshape(img,pixelCount,1);
4 P = cat(3,distortedX,distortedY);
5 %%
6 P = reshape(P,pixelCount,2);
7 interpolant = scatteredInterpolant(P,img_vector,'linear','none');
8 imgOut = interpolant(X,Y);
9 elseif strcmp(mode,'undistort')
10 imgOut = interpn(X,Y,img,distortedX,distortedY,'linear',NaN);
11 end
```

### 4.3 Region-Of-Interest

Die Interpolatoren sind so konfiguriert, dass die Bildbereiche außerhalb des gültigen Wertebereiches einen NaN Wert erhalten. Alle Werte, die erfolgreich interpoliert werden, bilden die Region-Of-Interest (kurz **ROI**) für einen Vergleich der Bilder.

```

1 %% Bestimmung der Region of Interest für Entzerrung
2 roi = ~isnan(imgOut);

```

Der für die Verzerrung ScatteredInterpolant interpoliert zwischen gegebenen Bildpunkten auf Basis einer Triangulation der Punkte. Die gesamte Basis interpolierbarer Punkte ist dabei die konvexe Hülle der Gesamtpunkte (vgl. Abbildung 11). Einige dieser Punkte wären bei einer unverzerrten Kamera außerhalb des Bildes, daher kann die Interpolation in diesem Bereich keine sinnvollen Werte liefern. Im Folgenden wird der fehlerhafte Bereich dennoch als Teil der ROI beachtet, da die Bestimmung einer passenden konkaven Hülle den Umfang dieser Arbeit überschreiten würde.



Abbildung 9: Punktfolge von Bildpunkten



Abbildung 10: Punktfolge verzerrter Bildpunkte

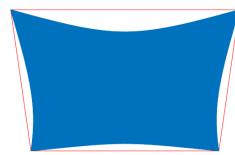


Abbildung 11: Konvexe Hülle der verzerrten Bildpunkte

#### 4.4 Vergleich

Die Basis des folgenden Vergleiches bildet eine Reihe von gerenderten Verzerrungen:

**Verzerrungsmodell** Poly3-Modell

**Bildmittelpunkt-Offset**  $X: 0, Y: 0.5$

**Koeffizient  $k_1$**  Von  $k_1 = 0.0125$  wiederholt verdoppelt bis  $k_1 = 0.4$ .

Sowohl das Testbild aus Abbildung 8, die gerenderten Testbilder und die daraus in Matlab entstandenen Bilder sind im dieser Arbeit zugehörigen Git-Repository [7] zu finden. Beispiele sind in Abbildung 12 zu sehen.

Abbildung 13 zeigt die Wertedifferenz des perspektivisch gerenderten Bildes und der Entzerrung. Auf diesem Bild, wie auch den restlichen Testbildern ist kein systematischer Unterschied bei der Positionierung der Punkte zu erkennen.

Es lässt sich vermuten, dass die Fehler an den Kanten lediglich durch die Interpolation entstehen.

In Abbildung 14a ist im oberen Bildrand zu erkennen, wie Punkte welche durch den pbmt abgebildet werden, durch Matlab nicht korrekt interpoliert werden können. Hier trifft die in Abschnitt 4.3 beschriebene fehlerhafte Interpolation innerhalb der konvexen Hülle ein.

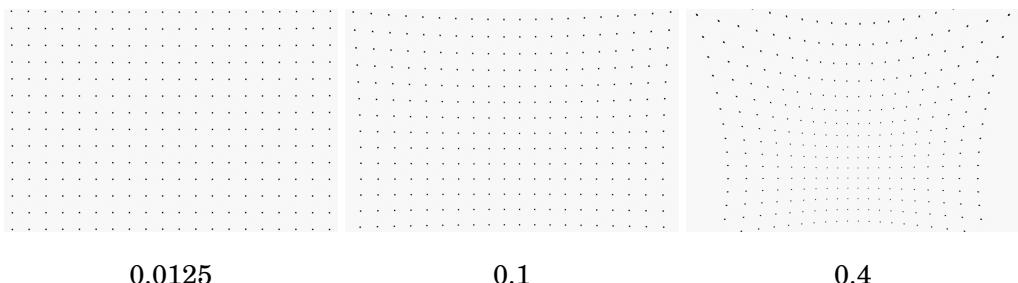


Abbildung 12: Render mit verschiedenen  $k_1$  Werten

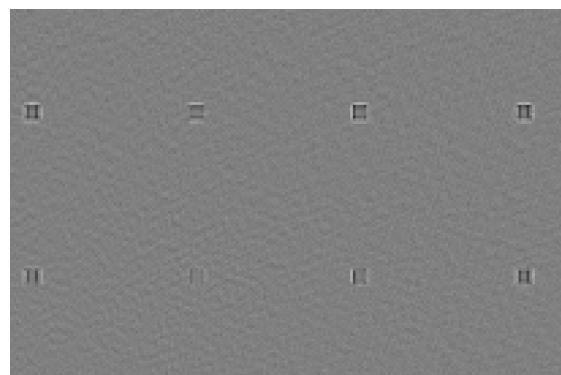
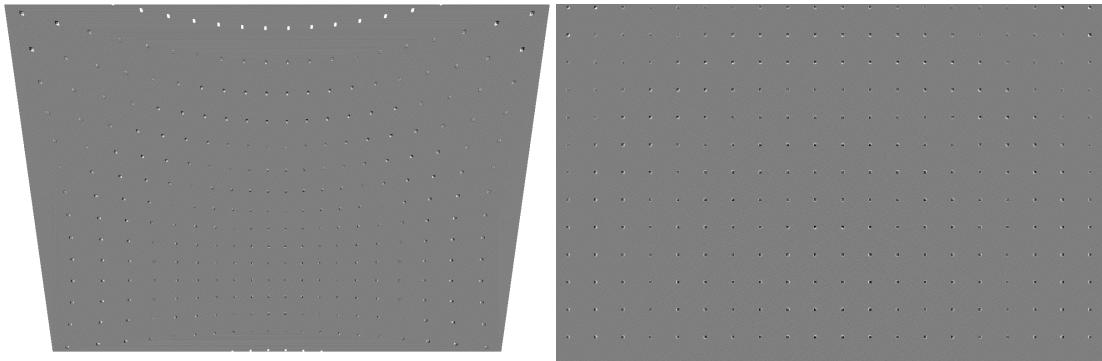


Abbildung 13: Wertedifferenz zwischen perspektivischem Render und Entzerrung (vergrößert)

Tabelle 1 zeigt den Einfluss der stärke der Verzerrung, auf die Ähnlichkeit der Ergebnisse von Matlab und pbtrt. Grundsätzlich ergibt sich aus einer stärkeren Verzerrung ein größerer Interpolationsfehler.

Die PSNR-Werte bei Tests mit anderen Verzerrungsmodellen sind im gleichen Wertebereich (ca. 37 – 40 dB bei geringer Verzerrung). Bei der Verschiebung des perspektivischen Bildes um einen halben Pixel ergibt sich beim Vergleich mit der unverschobenen Version eine PSNR von 32.22 dB.

Insgesamt kann daher davon ausgegangen werden, dass das entwickelte pbtrt Modell die Verzerrung korrekt anwendet.



(a) Differenz Matlab- und pbrt-Verzerrung

(b) Differenz Matlab-Entzerrung und Test-Bild

Abbildung 14: Wertedifferenzen von Ent-/Verzerrung mit  $k_1 = 0.4$  Mittleres Grau = Kein Fehler, Schwarz/Weiß: Vorzeichenbehafteter Fehler

$k_1$	Entzerrung		Verzerrung	
	PSNR/dB	$ e_{MAX} $	PSNR/dB	$ e_{MAX} $
0.0125	40.02	0.2366	39.90	0.2701
0.025	40.00	0.2488	39.93	0.2588
0.05	39.72	0.2675	39.88	0.2918
0.1	39.59	0.2667	40.08	0.2907
0.2	38.86	0.3357	38.87	0.9769 *
0.4	35.42	0.5514	33.16	0.9865 *

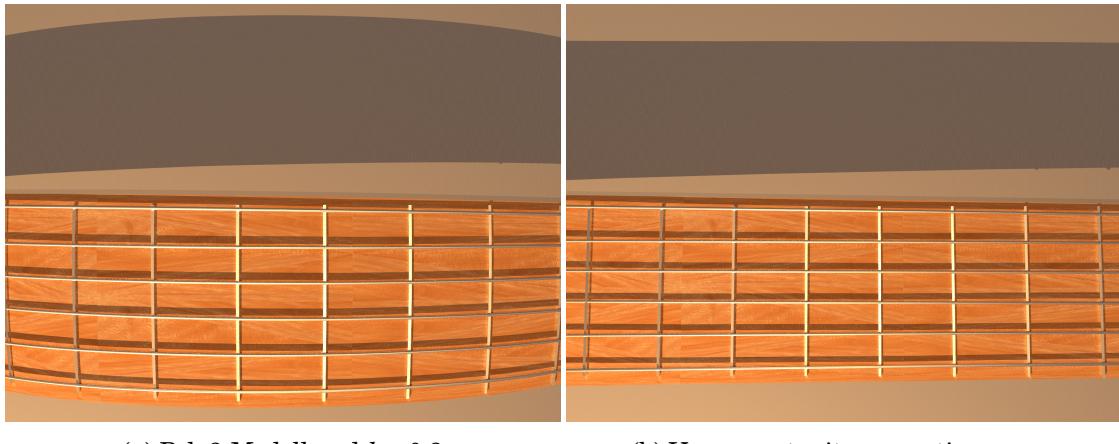
Tabelle 1: Auswertung der Reihe von Bildern mit PSNR und maximalem Absolutfehler (höchster Bildwert ist 1). Mit \* markierte Werte stammen von der ungenauen ROI (Abschnitt 4.3)

## 5. Ergebnisse, Diskussion und Ausblick

In der durchgeföhrten Arbeit hat sich gezeigt, dass mit einer relativ kleinen Modifikation eine Implementierung von gemessenen Verzeichnungen realer Objektive in den vorhandenen Source Code von *pbrt* implementiert werden konnte. Die einzige Schwierigkeit bestand darin, eine Invertierung der gemessenen Verzerrung zu bilden, welche numerisch errechnet werden musste.

Anschließend konnten die damit durchgeföhrten Verzerrungen, wie sie exemplarisch in Abbildung 15 an einem gerenderten Gitarren-Griffbrett zu sehen sind, mit einer bereits vorhandenen Matlab Implementierung verifiziert werden, wobei sich die geringen Unterschiede der beiden Ergebnisse auf Ungenauigkeiten aufgrund von Interpolation zurückführen lassen.

Mögliche Erweiterungen der hier durchgeföhrten Implementierung könnten eine Verzerrung nach einem nicht radialsymmetrischen Modell umfassen. Auch wenn sich die



(a) Poly3-Modell und  $k = 0.2$

(b) Unverzerrt mit perspektiv camera

Abbildung 15: Gerendertes Gitarrengriffbrett

meisten Verzeichnungen von Objektiven mit den hier genutzten Modellen gut abbilden lassen, liegt in der Realität nicht unbedingt eine komplett radialsymmetrische Verzerrung vor. So kann eine größere Genauigkeit durch ein getrenntes Abbilden der Verzerrung für x- und y-Richtung oder Hinzufügen einer Tangentialkomponente zur Radialverzerrung erreicht werden. Auch das momentan in der Alpha Version von Lensfun integrierte Verzerrungsmodell nach Adobe Vorgaben kann integriert werden, wenn dessen Implementierung ausreichend getestet ist.

Als weiteren Ausblick kann außerdem die Implementierung von Randabdunklungseffekten (Vignettierung) modelliert werden, um eine weitere in der Realität auftretende Eigenschaft von Optiken zu berücksichtigen.

## Literatur

- [1] <https://www.pbrt.org/>.
- [2] <http://lensfun.sourceforge.net/>.
- [3] [http://lensfun.sourceforge.net/manual/v0.3.2/group\\_\\_Lens.html#gaa505e04666a189274ba66316697e308e](http://lensfun.sourceforge.net/manual/v0.3.2/group__Lens.html#gaa505e04666a189274ba66316697e308e).
- [4] <https://www.boost.org>.
- [5] [https://www.boost.org/doc/libs/1\\_69\\_0/libs/numeric/ublas/doc/index.html](https://www.boost.org/doc/libs/1_69_0/libs/numeric/ublas/doc/index.html).
- [6] <https://www.pbrt.org/fileformat-v3.html>.
- [7] <https://github.com/Phenylalaninquelle/distortionCamera>.

- [8] I.N. Bronstein, K.A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Europa-Lehrmittel, Haan-Gruiten, 2018.
- [9] Frank M. Candocia. *A Scale-Preserving Lens Distortion Model and its Application to Image Registration*. Florida International University, May 2006.
- [10] Zhongwei Tang et al. A precision analysis of camera distortion models. *IEEE Transactions on image processing*, 26, no.6, 2017.
- [11] Imatest LLC. Sfrplus distortion and field of view measurement. <http://www.imatest.com/docs/sfrplus-distortion-measurements/#radial>. Aufgerufen am 31. 01. 2019.
- [12] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory To Implementation*. 2018.
- [13] W.J. Smith. *Modern Optical Engineering: The Design of Optical Systems*. McGraw-Hill series on optical and electro-optical engineering. McGraw Hill, 2000.
- [14] Eric E. Weisstein. Least squares fitting. <http://mathworld.wolfram.com/LeastSquaresFitting.html>. Aufgerufen am 29. 01. 2019.
- [15] Eric E. Weisstein. Least squares fitting – polynomial. <http://mathworld.wolfram.com/LeastSquaresFittingPolynomial.html>. Aufgerufen am 29. 01. 2019.