Aufgabenstellung Studienarbeit

TECHNISCHE UNIVERSITÄT DARMSTADT

„Beugungsbasierte Optiksimulation unter C++“

Bearbeiter: Jens Ritzert

Prof. Dr.-Ing. C. Tropea
Fachgebiet Strömungslehre und Aerodynamik
Fachbereich 16 • Maschinenbau

Petersenstraße 30
64287 Darmstadt
Telefon (06151) 16-2854 Sekretariat
Telefax (06151) 16-4754
ctropea@sla.tu-darmstadt.de

Betreuer:
Dipl.-Ing. Nils Damaschke
Tel.: (06151) 166558
e-mail: ndamasch@sla.tu-darmstadt.de

Beginn der Arbeit:
30.10.2000
Abgabetermin:
24.04.2001

Problemstellung:
Zur berührungslosen Geschwindigkeits- und Partikelgrößenmessung werden in der Strömungsmeßtechnik optische Verfahren wie die Phasen Doppler oder Global Phase Doppler Technik eingesetzt. Bei der Global Phase Doppler Technik werden zur Größenbestimmung defokussierte Abbildungen von mit kohärentem Licht beleuchteten Partikeln genutzt. Für die Weiterentwicklung dieser Meßtechnik muß zusätzlich zu den Streueigenschaften der Partikel die optische Aufnahmeeinheit, z.B. eine CCD-Kamera, simuliert werden.

Aufgabe:
Im Rahmen der Studienarbeit soll ein Programm erstellt werden, welches die Lichtausbreitung innerhalb der Aufnahmeeinheit mit Hilfe der Beugungstheorie simuliert. Die Programmierung erfolgt unter der objektorientierten Sprache C++.
Für die Bearbeitung der Aufgabe sollen zunächst die aus der Literatur bekannten grundsätzlichen Beziehungen der skalaren Beugungstheorie zusammengestellt werden. Dies beinhaltet die Beugungstheorie für Aperturen und Linsen sowie unterschiedliche Näherungen zur Berechnung der Beugung.
Anhand dieser Zusammenhänge soll ein objektorientierter Klassenentwurf für die Simulation der Aufnahmeeinheit erstellt werden. Der Klassenentwurf ist dabei so zu gestalten, daß sich die Struktur des Programms in ein bestehendes Paket zur Berechnung der Lichtstreuung an kleinen Teilchen einfügt.
Die Implementierung der Beugungstheorie innerhalb dieses Klassenentwurfs erfolgt unter MS Visual C++. Implementiert werden sollen im Rahmen der Studienarbeit Aperturen, Konvex-, Konkav- und Zylinderlinsen sowie Detektoreinheiten. Ziel der Arbeit sind erste beugungsbasierte Simulation einer Kameraoptik.

Inhaltsverzeichnis

1 Einleitung und Motivation 1

2 Beugung 6

2.1 Grundlagen der Lichtausbreitung 6

2.2 Beugungstheorie 8

2.3 Möglichkeiten der Beugungsberechnung 12

3 Linsen 23

3.1 Konvex- und Konkavlinsen 24

3.2 Zylinderlinse 28

4 Programmentwurf 29

4.1 Objektorientierte Analyse, Design und Klassenentwurf 31

4.2 Gültigkeitskonzept 55

4.3 Programmablauf 58

5 Anwendungsfälle und Leistungvermögen 63

6 Ausblick 77

Anhang

A Fouriertransformation

B Quellcode

Bezeichnungen

Literaturverzeichnis

1 Einleitung und Motivation

Die Messtechnik dient zur experimentellen Bestimmung von quantitativ erfassbaren Messgrößen in technischen Systemen [DUB97]. In den vergangenen Jahren sind die Anforderungen an die Messtechnik stark gestiegen. Hierzu zählen nicht nur die Genauigkeit der Messgrößen, sondern auch die Unabhängigkeit der ermittelten Messgrößen von Störungen aus der direkten Umgebung. Um diesen Anforderungen gerecht zu werden, wurden neue Messverfahren entwickelt.

Ein neues Messverfahren ist die Global Phase Doppler (GPD) Messtechnik. Sie wird entwickelt, um die 3D-Position, 3D-Geschwindigkeit und die Größe von kleinen Partikeln zu ermitteln. Das Messverfahren soll im weiteren kurz erläutert werden. Für genauere Ausführungen ist [ALB01] zu empfehlen.
Bei diesem Verfahren sind im Raum ein Laser, mehrere Partikel und eine Kamera so angeordnet, dass das vom Laser ausgehende Licht von den Partikeln gestreut wird und von dort in das Objektiv der Kamera gelangt. Der Strahlengang gemäß der geometrischen Optik kann anhand der Abbildung 1.1 nachvollzogen werden.

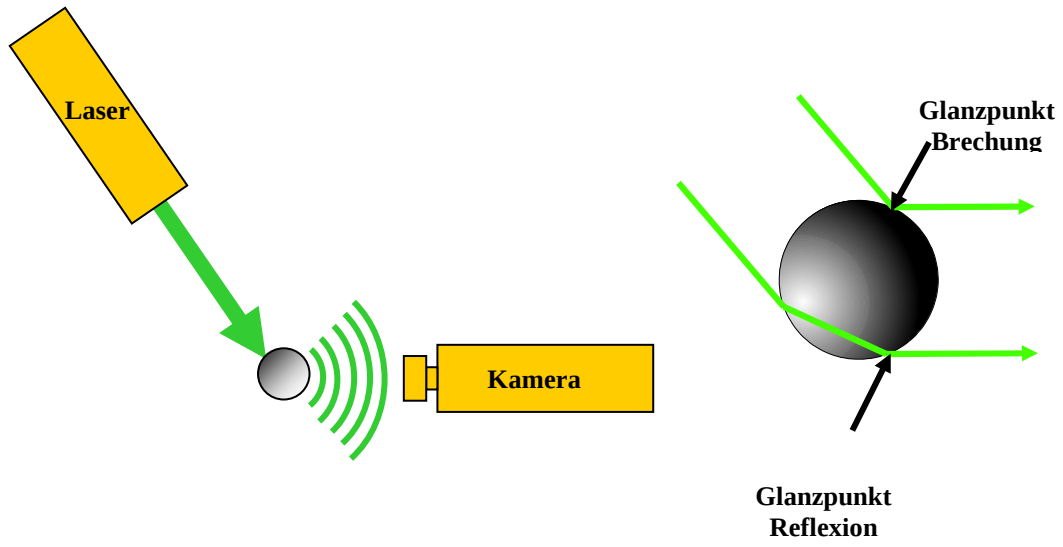


Abbildung 1.1 Versuchsaufbau

Das mit der Kamera fokussierte Bild eines Partikels ist in Abbildung 1.2 dargestellt. Man erkennt die beiden Glanzpunkte aus der Brechung und der Reflexion. Zusätzlich ist noch der Glanzpunkt, der durch die Brechung vierter Ordnung entsteht, zu erkennen.

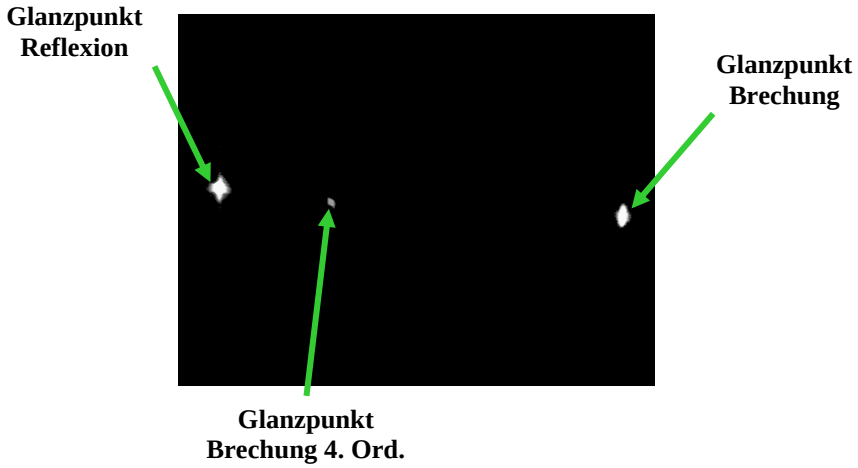


Abbildung 1.2 Kamerabild

Die Anzahl der Glanzpunkte der einzelnen Streulichtordnungen und die Lage auf der Partikeloberfläche sind winkelabhängig. Zu beachten ist, dass die Intensität der Glanzpunkte mit zunehmender Streulichtordnung abnimmt und das die Größe der Punkte durch die geometrische Abmessung der Apertur der Kamera bestimmt ist. Wird der Aufbau der Kamera vereinfacht als eine Aneinanderreihung einer Linse und einem CCD-Chip beschrieben, so ergibt sich der Versuchsaufbau gemäß Abbildung 1.3.

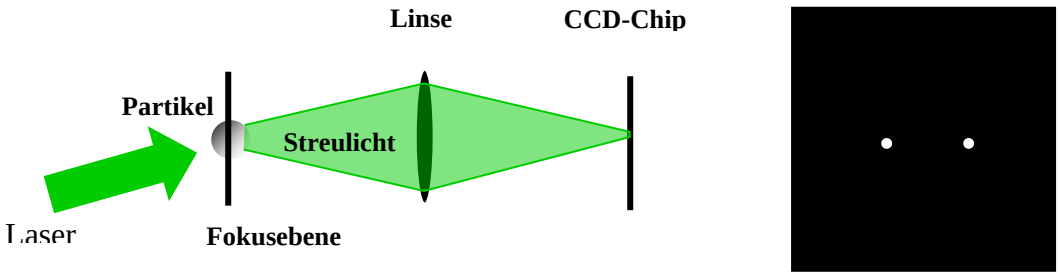


Abbildung 1.3 Bildaufnahme

Der Partikel befindet sich hierbei in der Fokusebene der Linse und die Glanzpunkte erscheinen als sehr kleine Flächen. Die Glanzpunkte verschwimmen jedoch, sobald das Bild defokusiert wird. Dieses Vorgehen ist analog zur defokusierten Einstellung einer Fotokamera. Die Defokussierung kann beim GPD entweder durch die absichtliche Verschiebung des CCD-Chips oder durch die veränderte Position des Partikels erreicht werden. Ab einer bestimmten Defokussierung überschneiden sich

die Abbildungen beider Glanzpunkte. In der sich überschneidenden Fläche interferiert das Licht der beiden Glanzpunkte, da es von ein und derselben einfallenden Welle stammt. Der resultierende Streifenabstand ist analog zum Doppelspaltversuch (Young´ scher Versuch) ein Maß für den Abstand der Glanzpunkte auf der Oberfläche des Partikels. Dieser Streifenabstand ist invers zum Abstand der Glanzpunkte. Da der Abstand der Glanzpunkte aber linear mit der Partikelgröße anwächst, ist die Interferenzstreifenanzahl innerhalb der defokussierten Abbildung ein Maß für die Partikelgröße.
Mit einer zusätzlichen Verringerung der örtlichen Auflösung der Kamera rücken die Zentren der defokussierten Glanzpunkte weiter zusammen. Im Grenzfall werden beide Mittelpunkte auf einer Stelle abgebildet. Der Übergang zwischen den verschiedenen Einstellungen ist in Abbildung 1.4 veranschaulicht.

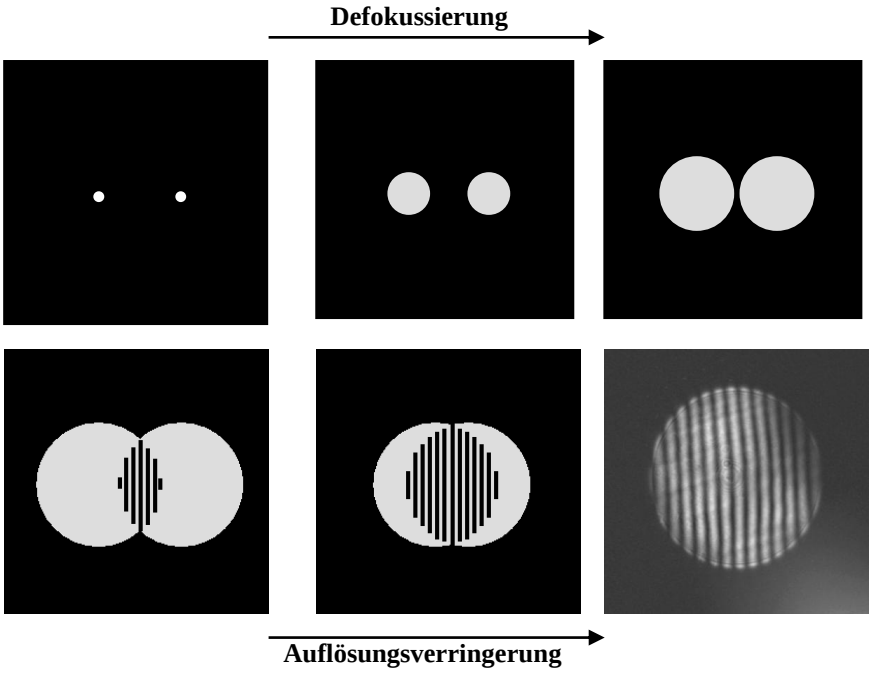


Abbildung 1.4 Defokussierung eines Partikels

Anhand einer solchen Abbildung lassen sich folgende Informationen über den Partikel gewinnen (vgl. Abbildung 1.5):

1. Position in der *x*-*y* Ebene (x_1, y_1)
2. Abstand von der Fokusebene (d_1)
3. Partikelgröße (N_1, Streifenanzahl im Partikel)

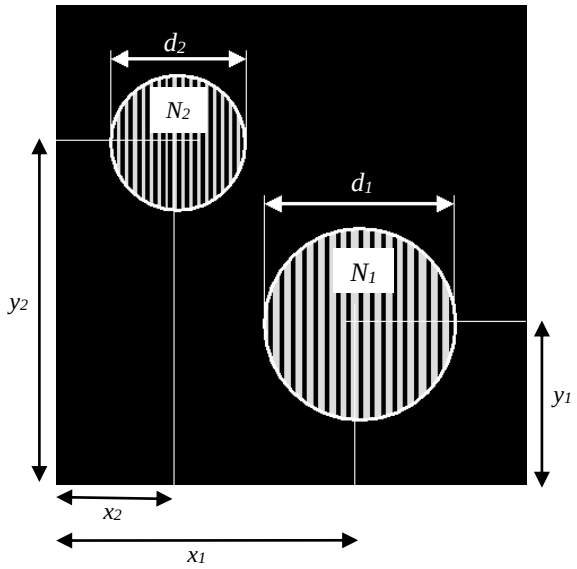


Abbildung 1.5 Informationen in einer doppelt belichteten Aufnahme

Wird ein derartiges Bild nicht nur zu einer Zeit t, sondern mit Hilfe einer Doppelbelichtung (Abbildung 1.5) auch zu einem Zeitpunkt t+Δt aufgenommen, erhält man eine weitere Abbildung aus der ebenfalls die oben beschriebenen Informationen gewonnen werden können (x_2, y_2, d_2, N_2). Mit Hilfe der Differenzen der Raumpositionen und der Zeit können die Geschwindigkeitskomponenten in allen drei Raumrichtungen errechnet werden. Der doppelt erhaltene Wert für die Partikelgröße kann für eine zusätzliche Kontrolle der Messwertverarbeitung verwendet werden Dies ist besonders hilfreich wenn sich nicht nur ein Partikel, sondern wie in Abbildung 1.6 sichtbar ist, 20, 30 oder mehr Partikel auf einer Aufnahme befinden. Hier kann mit Hilfe der Partikelgröße eine „bessere“ paarweise Zuordnung erfolgen.

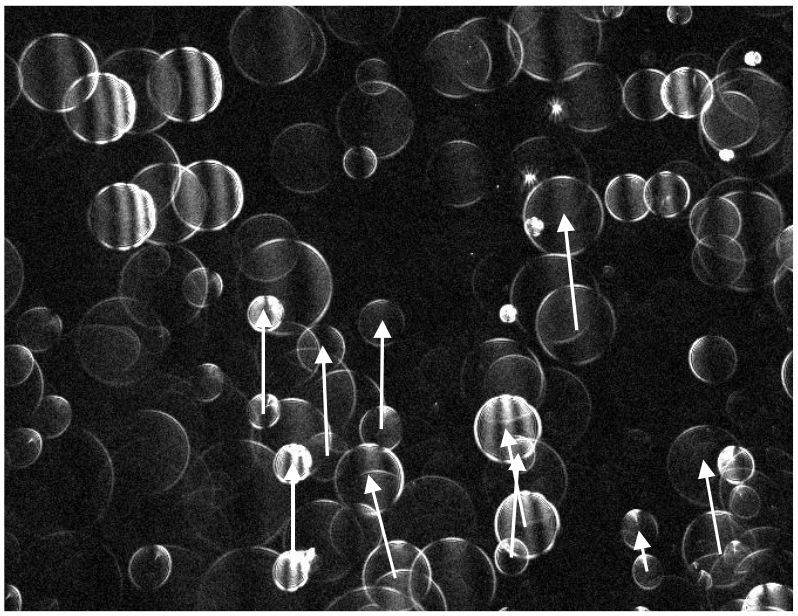


Abbildung 1.6 Bild mit Messwertauswertung

Es ist zu beachten, dass die verschiedenen physikalischen und mathematischen Beziehungen wie z.B. zwischen dem Glanzpunktabstand und der Partikelgröße genau genommen nichtlinear sind. Um diese Nichtlinearitäten zu überprüfen, sind Simulationen notwendig.

Eine dieser Simulationen betrifft die Kameraoptik. Sie berrechnet die Entstehung des Bildes von Beginn des ersten optischen Elements bis zum CCD-Chip. Dabei können sich innerhalb der Kamera verschiedene optische Elemente befinden. Beispiele hierfür sind die Rechteck-, Kreisapertur, Konvex-, Konkav- oder auch die Zylinderlinsen. Für die Simulation ist zunächst eine mathematische und physikalische Herleitung der beschreibenden Gleichungen notwendig. Nur aus dieser Betrachtung heraus können die durchgeführten Vereinfachungen und Gültigkeitseinschränkungen nachvollzogen und bewertet werden. Diese Theorien werden in den Kapiteln 2 und 3 behandelt.
Anschließend erfolgt die Umsetzung der Theorien in einen Programmcode. Hierfür wird zunächst eine objektorientierte Analyse und ein Klassenentwurf durchgeführt, bevor mit der eigentlichen Implementierung begonnen wird. Kapitel 4 beinhaltet dementsprechend die Beschreibung des Softwareentwurfs und die vollständige Programmdokumentation.
Danach werden in Kapitel 5 erste Berechnungsergebnisse vorgestellt und das Leistungsvermögen der verschiedenen physikalischen Theorien behandelt. Abschließend wird in Kapitel 6 ein Ausblick.

2 Beugung

Die Kamera beinhaltet, wie zuvor beschrieben, mehrere Elemente die in der Lage sind, die Lichtausbreitung auf unterschiedliche Art und Weise zu beeinflussen. Diese Elemente lassen sich zunächst grob in drei Klassen untergliedern:

- Apertur
- Linse
- Freier Zwischenraum

Die Eigenschaft einer Apertur ist, einen bestimmten Teil des ankommenden Lichts auszublenden und nur den Rest ungehindert passieren zu lassen. Die Linse hingegen besitzt die Fähigkeit, die Phasenlage des Lichtes zu ändern. Die dritte Klasse, der freie Zwischenraum, beschreibt die ungehinderte räumliche Ausbreitung des Lichts.

Ziel dieses Kapitels ist die mathematische Beschreibung einer Apertur und des freien Zwischenraums. Voraussetzung hierfür ist, dass das Licht als elektromagnetische Welle betrachtet werden kann.

2.1 Grundlagen der Lichtausbreitung

Licht ist eine elektromagnetische Welle, von denen das Auge nur den Wellenlängenbereich von etwa 400 bis 800nm wahrnimmt. Licht ist aufgrund seiner Eigenschaften sowohl in der Lage Energie von einem Ort A zu einem Ort B zu transportieren als auch, dass die Energie der elektromagnetischen Welle in eine andere Energieform (Bsp. Elektrische Energie) umgewandelt werden kann.

Zur Berechnung der Ausbreitung des Lichtes gibt es zwei Möglichkeiten:

- Strahlen- oder geometrische Optik
- Wellenoptik

Die erste Variante vernachlässigt die geometrische Breite eines Lichtbündels und bildet dieses auf einen Strahl ab. Die physikalische Vorraussetzung für diese Abstraktion ist, dass die betrachteten geometrischen Strukturen viel größer als die Wellenlänge sind. Die Ausbreitung des Lichtes gemäß der geometrischen Optik kann mit einfachen geometrischen Beziehungen berechnet werden. Linsen, Spiegel, Prismen und andere optische Elemente können so ausgelegt werden. Die Strahlenoptik ist jedoch nicht in der Lage Interferenz- und Beugungserscheinungen, wie sie bei defokussierten Abbildungen vorkommen, zu erklären.
Diese Erscheinungen werden von der Wellenoptik vollständig mathematisch erfasst und beschrieben. Auf sie soll im weiteren näher eingegangen werden.

Wird die allgemeine Wellengleichung in vektorieller Form für das elektrische Feld aufgestellt, so lässt sie sich gemäß den Gleichungen (2.1) bis (2.3) aufschreiben.

$$\nabla^2 \underline{\mathbf{E}} - \frac{n^2}{c_0^2} \frac{\partial^2 \underline{\mathbf{E}}}{\partial t^2} = 0 \tag{2.1}$$

$$c_0 = \frac{1}{\sqrt{\mu_0 \varepsilon_0}} = nc \tag{2.2}$$

$$c = \frac{1}{\sqrt{\varepsilon_0 \varepsilon_r \mu_0 \mu_r}} = \frac{c_0}{\sqrt{\varepsilon_r \mu_r}} \tag{2.3}$$

Im Falle des kartesischen Koordinatensystems besitzt der Vektor $\underline{\mathbf{E}}$ die Komponenten $\underline{\mathbf{E}}_x$, $\underline{\mathbf{E}}_y$.
Nähere Erläuterungen bezüglich des Gleichungssystems und der Abhängigkeit des elektrischen Feldes vom magnetischen Feld sind [HEC98], [GOO68] und [PÉR??] zu entnehmen.

Zum Finden von mathematischen Lösungen der Gleichung (2.1) müssen Einschränkungen getroffen werden. Das Ausbreitungsmedium der Welle wird deswegen auf ein lineares, isotropes, homogenes und unmagnetisches Medium begrenzt.
„Linear“ bedeutet, dass verschiedene Lösungen der Wellengleichung superponiert werden können. Das Medium ist „isotrop“ wenn die Eigenschaften unabhängig von der Polarisationsrichtung der Welle sind. Als „homogen“ wird es bezeichnet, sobald die Dielektrizitätszahl ε und die Permeabilität μ konstant sind. Schließlich bezeichnet der Begriff des „nichtmagnetischen“ Mediums ein Material dessen magnetische Permeabilität identisch der Permeabilität μ_0 im Vakuum ist.

Wird der konstante Term vor der Zeitableitung durch die Wellenzahl k ersetzt, spricht man von der Helmholtz-Gleichung.

$$\left(\nabla^2 + k^2\right) \underline{\mathbf{E}} = 0 \tag{2.4}$$

$$k = 2\pi\, n \frac{f}{c_0} = \frac{2\pi}{\lambda} \tag{2.5}$$

Sie beschreibt die Ausbreitung einer Welle und deren Störung in dem oben näher spezifizierten Medium und bildet somit die Grundlage weiterer Betrachtungen.

2.2 Beugungstheorie

Für die weitere Berechnung der Störung nach der Helmholtz-Gleichung ist die Anwendung eines Hilfssatzes notwendig. Dieser Hilfssatz soll hier bereits vorweggenommen werden. Es ist das Green´sche Theorem.

> *U(P)* und *G(P)* sind zwei komplexe Funktionen in Abhängigkeit vom Ort P. S stellt zusätzlich eine geschlossene Oberfläche um das Volumen V dar. Sind *U*, *G* und deren ersten beiden Ableitungen innerhalb und auf S stetig und partiell differenzierbar, dann gilt:
>
> $$\iiint_V \left(U\nabla^2 G - G\nabla^2 U\right)dv = \iint_S \left(U\frac{\partial G}{\partial n} - G\frac{\partial U}{\partial n}\right)ds \tag{2.6}$$
>
> Die partielle Ableitung in Normalenrichtung ist dabei die Ableitung nach dem nach außen gerichteten Normalenvektor n auf jedem Punkt der Oberfläche S.

Die Funktion *U* stellt hier das elektrische Feld dar, während die Funktion *G* eine einfache Hilfsfunktion repräsentiert. Streng genommen ist *G* die Green´sche Funktion. Für nähere mathematische und physikalische Erläuterungen ist [GOO68] zu empfehlen.
Gleichung (2.6) ist nur mit speziellen Annahmen bezüglich der Funktionen *U*, *G* und deren Randbedingungen lösbar. Stellt man an die Hilfsfunktion *G* auch die Forderung, dass sie die Helmholtz-Gleichung erfüllen soll, so gelten für beide Funktionen folgende Formeln:

$$\nabla^2 \underline{\mathbf{G}} = -k^2 \underline{\mathbf{G}} \tag{2.7}$$

$$\nabla^2 \underline{\mathbf{E}} = -k^2 \underline{\mathbf{E}} \tag{2.8}$$

Werden diese Gleichungen in das Green´sche Theorem eingesetzt, erhält man für das Volumenintegral den Ausdruck

$$\iiint_V \left(\underline{\mathbf{E}}\nabla^2 \underline{\mathbf{G}} - \underline{\mathbf{G}}\nabla^2 \underline{\mathbf{E}}\right)dv = \iiint_V \left(-\underline{\mathbf{E}}\,k^2 \underline{\mathbf{G}} + \underline{\mathbf{G}}\,k^2 \underline{\mathbf{E}}\right)dv \equiv 0 \tag{2.9}$$

Da das Volumenintegral unabhängig von den gewählten Integrationsgrenzen den Wert 0 annimmt, muss im folgenden nur noch das Oberflächenintegral aus Gleichung (2.6) berechnet werden.
Hiefür wird zunächst ein Volumen V mit der Oberfläche S betrachtet (Abbildung 2.1). Innerhalb dieses Volumens liegt ein Punkt P_0, an dessen Position der Einfluss der Störungen berechnet werden soll, die sich von der Oberfläche S in das Innere ausbreiten. Diese resultierende Störung am Punkt P_0 kann mit den gegebenen Formeln nicht explizit berechnet werden.

Damit der Punkt P_0 mit den beschriebenen Gleichungen einer mathematischen Betrachtung zugänglich gemacht werden kann, wird ein infinitesimales Volumen V_ε einer Kugel mit der Oberfläche S_ε um ihn gelegt. Berechnet man auf dieser Fläche den Wert des Oberflächenintegrals für den Grenzwertübergang $\varepsilon \to 0$ erhält man schließlich denselben Wert wie für P_0 selber.
Gleichzeitig lässt sich auch ein Volumen $V'=V-V_\varepsilon$ definieren. Die zugehörige Oberfläche ergibt sich zu $S'=S+S_\varepsilon$. Eine anschauliche Betrachtung ist Abbildung 2.1 zu entnehmen.

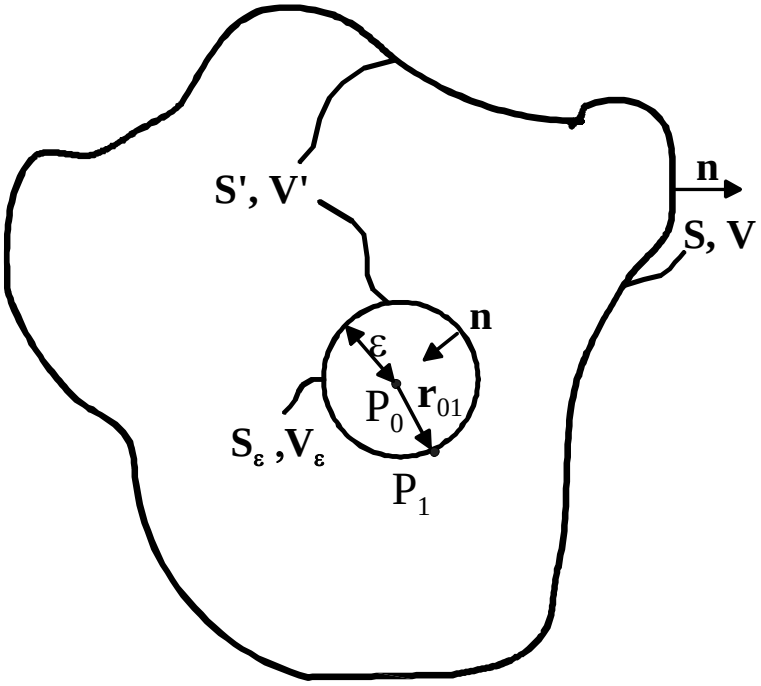


Abbildung 2.1 Darstellung der verschiedenen Volumina und Oberflächen

Bezieht man diese Anordnung in die weiter oben durchgeführte mathematische Untersuchung mit ein, erhält man für das Volumenintegral V' erneut den Wert 0 (vgl. Gleichung (2.9)). Das Oberflächenintegral besitzt dementsprechend denselben Wert.

$$\iint_{S'} \left(\underline{\underline{\mathbf{E}}} \frac{\partial \underline{\underline{G}}}{\partial n} - \underline{\underline{\mathbf{G}}} \frac{\partial \underline{\underline{\mathbf{E}}}}{\partial n} \right) ds = 0 \tag{2.10}$$

Teilt man dieses Oberflächenintegral auf die Teilflächen auf, so ergibt sich Gleichung (2.11).

$$\iint_{S} \left(\underline{\underline{\mathbf{E}}} \frac{\partial \underline{\underline{\mathbf{G}}}}{\partial n} - \underline{\underline{\mathbf{G}}} \frac{\partial \underline{\underline{\mathbf{E}}}}{\partial n} \right) ds + \iint_{S_\varepsilon} \left(\underline{\underline{\mathbf{E}}} \frac{\partial \underline{\underline{\mathbf{G}}}}{\partial n} - \underline{\underline{\mathbf{G}}} \frac{\partial \underline{\underline{\mathbf{E}}}}{\partial n} \right) ds = 0 \tag{2.11}$$

Durch eine geeignete Wahl der Hilfsfunktionen **G** lassen sich diese Gleichungen lösen. Legt man einen Punkt P_1 auf die Oberfläche S_ε und verwendet für die Hilfsfunktionen den Ansatz einer sich ausbreitenden Kugelwelle, dann erhält man für **G**

$$\underline{G}_i(P_1) = \frac{\exp(\mathrm{j}kr_{01})}{r_{01}} \qquad \text{i=1,2} \qquad (2.12)$$

Dies ist identisch mit der Annahme, dass die gesamte Energie von der Oberfläche S_ε auf dem Punkt P_1 konzentriert sei.
Entsprechend gilt für die Ableitung in Richtung der Flächennormalen

$$\frac{\partial \underline{G}_i(P_1)}{\partial n} = (\mathbf{n} \otimes \mathbf{r}_{01})\left(\mathrm{j}k - \frac{1}{r_{01}}\right)\frac{\exp(\mathrm{j}kr_{01})}{r_{01}} \qquad \text{i=1,2} \qquad (2.13)$$

Dabei ist der Operator $\otimes$ wie folgt definiert:

$$\mathbf{n} \otimes \mathbf{r}_{01} = \frac{\mathbf{n} \cdot \mathbf{r}_{01}}{|\mathbf{n}||\mathbf{r}_{01}|} \qquad (2.14)$$

Der Vektor $\mathbf{r}_{01}$ zeigt dabei von Punkt P_0 zum Punkt P_1.

Als nächstes müssen die Integrale aus (2.11) berechnet werden. Für den Punkt P_1 auf S_ε können (2.12) und(2.13) mit $\mathbf{n}\otimes\mathbf{r}_{01}$= -1 vereinfacht werden zu

$$\underline{G}_i(P_1) = \frac{\exp(\mathrm{j}k\varepsilon)}{\varepsilon} \qquad \text{i=1,2} \qquad (2.15)$$

$$\frac{\partial \underline{G}_i(P_1)}{\partial n} = \frac{\exp(\mathrm{j}k\varepsilon)}{\varepsilon}\left(\frac{1}{\varepsilon} - \mathrm{j}k\right) \qquad \text{i=1,2} \qquad (2.16)$$

Werden diese beiden Gleichungen in das Oberflächenintegral für S_ε von Gleichung (2.11) eingesetzt und der Grenzwertübergang $\varepsilon \to 0$ durchgeführt ergibt sich

$$\lim_{\varepsilon\to 0} \iint_{S_\varepsilon}\left(\underline{E}_i \frac{\partial \underline{G}_i}{\partial n} - \underline{G}_i \frac{\partial \underline{E}_i}{\partial n}\right)ds =$$

$$\lim_{\varepsilon\to 0} 4\pi\varepsilon^2\left(\underline{E}_i(P_0)\frac{\exp(\mathrm{j}k\varepsilon)}{\varepsilon}\left(\frac{1}{\varepsilon} - \mathrm{j}k\right) - \frac{\partial \underline{E}_i(P_0)}{\partial n}\frac{\exp(\mathrm{j}k\varepsilon)}{\varepsilon}\right) = 4\pi\underline{E}_i(P_o)$$

$$\text{i=1,2} \qquad (2.17)$$

Mit Hilfe dieser Vereinfachung lässt sich Gleichung (2.11) auch wie folgt schreiben:

$$\iint_S\left(\underline{\mathbf{E}}\frac{\partial \underline{\mathbf{G}}}{\partial n} - \underline{\mathbf{G}}\frac{\partial \underline{\mathbf{E}}}{\partial n}\right)ds + 4\pi\underline{\mathbf{E}}(P_0) = 0 \qquad (2.18)$$

Diese Form der Darstellung wird im allgemeinen als Integralgleichung von Helmholtz und Kirchhoff bezeichnet. Mit einer einfachen Äquivalenzumformung ergibt sich

$$\underline{\mathbf{E}}(P_0) = \frac{1}{4\pi} \iint_S \left(\underline{\mathbf{G}} \frac{\partial \underline{\mathbf{E}}}{\partial n} - \underline{\mathbf{E}} \frac{\partial \underline{\mathbf{G}}}{\partial n} \right) ds \tag{2.19}$$

Für eine nähere Betrachtung des Einsatzgebietes der Gleichung (2.19) wird die Oberfläche S in drei Teilflächen S_1, S_2 und Σ aufgeteilt. Die räumliche Anordnung der Teilflächen kann Abbildung 2.2 entnommen werden.

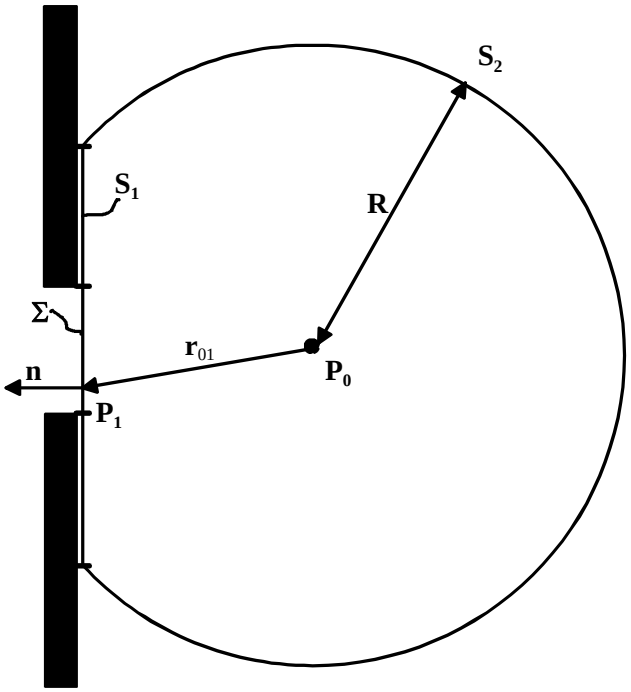


Abbildung 2.2 Apertur

Im Bereich der Kamera kann man diese Anordnung so nutzen, als ob das von links eingestrahlte bekannte elektrische Feld **E** auf den Schirm trifft und von dort aus im Punkt P_0 ein elektrisches Feld der Größe **E**(P_0) erzeugt. Um die Größe **E**(P_0) zu berechnen, müssen die drei Oberflächenintegrale S_1, S_2 und Σ bestimmt werden.

Für das Integral über S_1 ist festzuhalten, dass diese Fläche, weil sie im geometrischen Schatten des Schirms liegt, keinen Beitrag liefert.
Das Integral über S_2 kann bei dieser Berechnung identisch null gesetzt werden, solange durch S_2 nur Wellen von Innen nach Außen hindurch treten und nicht umgekehrt. Diese Bedingung wird in der Literatur als die „Sommerfeld radiation condition“ bezeichnet.
Die Integration braucht daher nur über die Fläche Σ zu erfolgen, die den Bereich der durchlässigen Apertur kennzeichnet. Auf dieser Fläche wird angenommen, das sich

durch die Anwesenheit der Apertur das Feld und seine Ableitung in Normalenrichtung nicht ändern.
Diese beiden Vereinfachungen sind gültig, sofern die Abmessungen der Apertur größer sind als die Wellenlänge des elektrischen Feldes. Die Gleichung (2.19) vereinfacht sich damit weiter zu

$$\underline{\mathbf{E}}(P_0) = \frac{1}{4\pi}\iint_{\Sigma}\left(\underline{\mathbf{G}}\frac{\partial \underline{\mathbf{E}}}{\partial n} - \underline{\mathbf{E}}\frac{\partial \underline{\mathbf{G}}}{\partial n}\right)ds \tag{2.20}$$

Diese Gleichung ist Ausgangspunkt weiterer Betrachtungen, die im wesentlichen nur unterschiedliche Ansätze bezüglich der Hilfsfunktion $\underline{\mathbf{G}}$ beinhalten.

2.3 Möglichkeiten der Beugungsberechnungen

Ziel dieses Kapitels ist es unterschiedliche Ansätze für die Hilfsfunktion $\underline{\boldsymbol{G}}$ und die daraus resultierenden Lösungen vorzustellen. Diese Ansätze werden an dieser Stelle nur mit ihren Deutungen und Einschränkungen erläutert, ohne exakt auf die mathematische Herleitung näher einzugehen.

Rayleigh und Sommerfeld stellten für dieses Problem zwei Theorien auf, die beide dieselbe Basis besitzen. Ihre Ansätze für die Funktion $\underline{\mathbf{G}}$ beinhalten entweder die Annahme, dass die Funktion $\underline{\mathbf{G}}$ auf Σ selber verschwindet oder aber deren Ableitung in Normalenrichtung identisch null ist. Eine solche Funktion würde einen Summanden unter dem Integral eliminieren und die analytische Behandlung weiter vereinfachen.
Sie nehmen an, dass die Hilfsfunktion $\underline{\boldsymbol{G}}_i$ nicht nur durch den Punkt P_0, sondern auch durch einen Punkt $\tilde{P}_0$ bestimmt wird. Die Lage des neuen Punktes $\tilde{P}_0$ ist genau die an dem Schirm gespiegelte Position des Punktes P_0 (Abbildung 2.3).

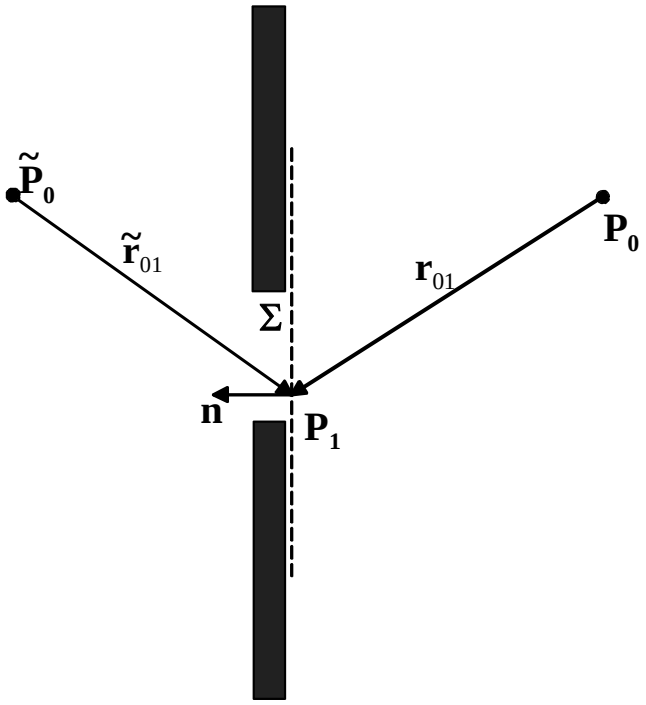


Abbildung 2.3 Darstellung der berechneten Geometrie

Wenn beide Punkte Ausgang einer Störung mit der gleichen Wellenlänge sind und mit 180° Phasenunterschied schwingen, ist die Green´sche Funktion bestimmt durch

$$\underline{G}_i^-(P_1) = \frac{\exp(\mathrm{j}kr_{01})}{r_{01}} - \frac{\exp(\mathrm{j}k\tilde{r}_{01})}{\tilde{r}_{01}} \qquad \text{i=1,2} \qquad (2.21)$$

Die Ableitung in Normalenrichtung ergibt sich unter Beachtung von Symmetriebeziehungen zu

$$\frac{\partial \underline{G}_i^-(P_1)}{\partial n} = 2(\mathbf{n} \otimes \mathbf{r}_{01})\left(\mathrm{j}k - \frac{1}{r_{01}}\right)\frac{\exp(\mathrm{j}kr_{01})}{r_{01}} \qquad \text{i=1,2} \qquad (2.22)$$

Mit der Einschränkung $r_{01} \gg \lambda$ ergibt sich

$$\frac{\partial \underline{G}_i^-(P_1)}{\partial n} = 2\,\mathrm{jk}(\mathbf{n} \otimes \mathbf{r}_{01})\frac{\exp(\mathrm{j}kr_{01})}{r_{01}} \qquad \text{i=1,2} \qquad (2.23)$$

In diesem Fall verschwindet die Funktion selber auf der Fläche Σ. Gleichung (2.20) vereinfacht sich damit zu

$$\begin{aligned}\underline{\mathbf{E}}(P_0) &= -\frac{1}{4\pi}\iint_\Sigma \underline{\mathbf{E}}\frac{\partial \underline{\mathbf{G}}_i^-}{\partial n}ds = -\frac{2\,\mathrm{j}\frac{2\pi}{\lambda}}{4\pi}\iint_\Sigma \underline{\mathbf{E}}\frac{\exp(\mathrm{j}kr_{01})}{r_{01}}(\mathbf{n} \otimes \mathbf{r}_{01})ds \\ &= \frac{1}{\mathrm{j}\lambda}\iint_\Sigma \underline{\mathbf{E}}\frac{\exp(\mathrm{j}kr_{01})}{r_{01}}(\mathbf{n} \otimes \mathbf{r}_{01})ds \qquad (2.24)\end{aligned}$$

Diese Lösung wird als erste Rayleigh-Sommerfeld-Gleichung bezeichnet. Eine zweite Rayleigh-Sommerfeld-Gleichung ergibt sich indem angenommen wird, dass die beiden Punktquellen nicht phasenverschoben, sondern in Phase schwingen. Bei dieser Annahme ist die Hilfsfunktion durch

$$\underline{G}_i^+(P_1) = \frac{\exp(\mathrm{j}kr_{01})}{r_{01}} + \frac{\exp(\mathrm{j}k\tilde{r}_{01})}{\tilde{r}_{01}} \qquad \text{i=1,2} \quad (2.25)$$

gegeben. Bei diesem Ansatz sind die Ableitungen in Normalenrichtung null. Wird auch diese Gleichung in (2.20) eingesetzt, erhält man

$$\underline{E}_i(P_0) = \frac{1}{4\pi}\iint_\Sigma \underline{G}_i^+ \frac{\partial \underline{E}_i}{\partial n}\,ds = \frac{1}{4\pi}\iint_\Sigma 2\frac{\exp(\mathrm{j}kr_{01})}{r_{01}}\frac{\partial \underline{E}_i}{\partial n}\,ds \qquad \text{i=1,2} \quad (2.26)$$

Als nächstes muss bei dieser Theorie noch eine Aussage über die Ableitung des einfallenden elektrischen Feldes erfolgen. Dabei geht man davon aus, dass auch dieses elektrische Feld nur durch Störungen einer Punktquelle P_2 entsteht. Dieser Punkt befindet sich wie Abbildung 2.4 zeigt links vom Schirm.

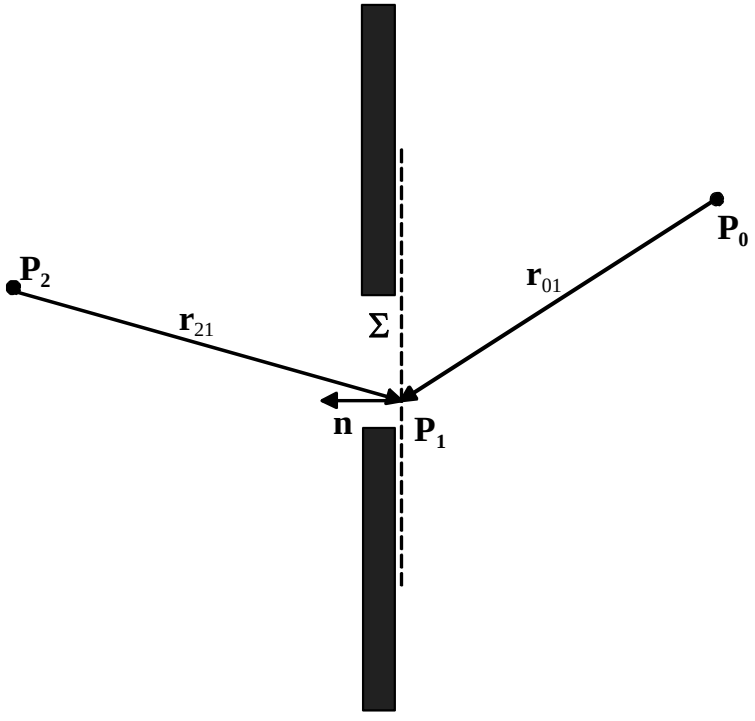


Abbildung 2.4 Darstellung der berechneten Geometrie

Die von diesem Punkt P_2 verursachten Störungen in Punkt P_1 ergeben sich bei Annahme einer Kugelwelle zu

$$\underline{E}_i(P_1) = \frac{\exp(\mathrm{j}kr_{21})}{r_{21}} \qquad \text{i=1,2} \quad (2.27)$$

Die Ableitung in Normalrichtung beträgt mit der Vereinfachung $r_{21} \gg \lambda$

$$\frac{\partial \underline{E}_i(P_1)}{\partial n} = (\mathbf{n} \otimes \mathbf{r}_{21})\left(\mathrm{j}k - \frac{1}{r_{21}} \right)\frac{\exp(\mathrm{j}kr_{21})}{r_{21}} \approx \mathrm{j}k(\mathbf{n} \otimes \mathbf{r}_{21})\underline{E}_i(P_1) \qquad \text{i=1,2} \qquad (2.28)$$

Eingesetzt in (2.26) erhält man die zweite Rayleigh-Sommerfeld-Gleichung.

$$\begin{aligned} \underline{E}_i(P_0) &= \frac{\mathrm{j}k}{2\pi}\iint_{\Sigma} \frac{\exp(\mathrm{j}kr_{01})}{r_{01}}\underline{E}_i(P_1)(\mathbf{n} \otimes \mathbf{r}_{21})ds \\ &= \frac{1}{\mathrm{j}\lambda}\iint_{\Sigma} \frac{\exp(\mathrm{j}kr_{01})}{r_{01}}\underline{E}_i(P_1)(-1)(\mathbf{n} \otimes \mathbf{r}_{21})ds \qquad \text{i=1,2} \qquad (2.29) \end{aligned}$$

Eine zweite Möglichkeit, die Gleichung (2.20) zu lösen, veröffentlichte Kirchhoff [GOO68]. Zwischen seinem Ansatz für die Hilfsfunktion $\underline{\boldsymbol{G}}$ und der Ansätze von Rayleigh und Sommerfeld existieren folgende Beziehungen:

$$\underline{\mathbf{G}}^{K} = \tfrac{1}{2}\underline{\mathbf{G}}^{+} \qquad (2.30)$$

$$\frac{\partial \underline{\mathbf{G}}^{K}}{\partial n} = \frac{1}{2}\frac{\partial \underline{\mathbf{G}}^{-}}{\partial n} \qquad (2.31)$$

Werden diese Gleichungen in (2.20) eingesetzt und dieselben Einschränkungen wie bei Rayleigh und Sommerfeld durchgeführt, ergibt sich die Kirchhoff´sche Lösung.

$$\underline{E}_i(P_0) = \frac{1}{\mathrm{j}\lambda}\iint_{\Sigma} \frac{\exp(\mathrm{j}kr_{01})}{r_{01}}\underline{E}_i(P_1)\tfrac{1}{2}\big((\mathbf{n} \otimes \mathbf{r}_{01}) - (\mathbf{n} \otimes \mathbf{r}_{21})\big)ds \qquad \text{i=1,2} \qquad (2.32)$$

Fasst man alle drei Lösungen in der Gleichung (2.33) zusammen, erhält man eine Gleichung, die sich nur in einem gemeinsamen „Neigungsfaktor“ ψ unterscheiden.

$$\underline{\mathbf{E}}(P_0) = \frac{1}{\mathrm{j}\lambda}\iint_{\Sigma} \frac{\exp(\mathrm{j}kr_{01})}{r_{01}}\underline{\mathbf{E}}(P_1)\,\psi ds \qquad (2.33)$$

$$\text{mit} \quad \psi = \begin{cases} \tfrac{1}{2}\big((\mathbf{n} \otimes \mathbf{r}_{01}) - (\mathbf{n} \otimes \mathbf{r}_{21})\big) & \text{Kirchhoff – Theorie} \\ \mathbf{n} \otimes \mathbf{r}_{01} & \text{Erste Rayleigh – Sommerfeld – Gleichung} \\ \mathbf{n} \otimes \mathbf{r}_{21} & \text{Zweite Rayleigh – Sommerfeld – Gleichung} \end{cases}$$

Betrachtet man die jeweils durchgeführten Vereinfachungen, dann ist zusätzlich zu erwähnen, dass die Kirchhoff´sche Theorie auch für nicht planare Beugungsschirme verwendet werden kann. Des weiteren sind die Gleichungen im Fernfeld alle mathematisch identisch, sofern die geometrischen Abmessungen der Apertur größer als die Wellenlänge sind.
Für den in der Kamera vorliegenden Spezialfall der eben einfallenden Welle ($\mathbf{n}\otimes\mathbf{r}_{21}= -1$) lassen sich die Terme weiter umformen.

$$\underline{\mathbf{E}}(P_0) = \frac{1}{\mathrm{j}\lambda}\iint_{\Sigma}\underline{\mathbf{E}}(P_1)\frac{\exp(\mathrm{j}kr_{01})}{r_{01}}\psi\,ds \tag{2.34}$$

$$\text{mit}\quad \psi = \begin{cases} \frac{1}{2}\left(1+(\mathbf{n}\otimes\mathbf{r}_{01})\right) \\ \mathbf{n}\otimes\mathbf{r}_{01} \\ 1 \end{cases}$$

Um diese Formel physikalisch einfacher zu deuten, ist es sinnvoll sich an dieser Stelle die Verhältnisse in der Kamera, wie in Abbildung 2.5 und 2.6 dargestellt, zu betrachten.
Auf eine Apertur mit der offenen Fläche Σ und dem Koordinatensystem *x,y,z* trifft das bekannte elektrische Feld $\underline{\mathbf{E}}_\Sigma(x,y,z=0)$, welches oben noch als $\underline{\mathbf{E}}(P_1)$ bezeichnet wurde. Des weiteren wird nun die elektrische Feldstärke $\underline{\mathbf{E}}(\xi,\eta,z_0)$ in einem Punkt P_0 berechnet, der sich in einer im Abstand z_0 aufgespannten Ebene befindet. Die Position dieses Punktes wird in einem ξ,η,z – Koordinatensystem angegeben.

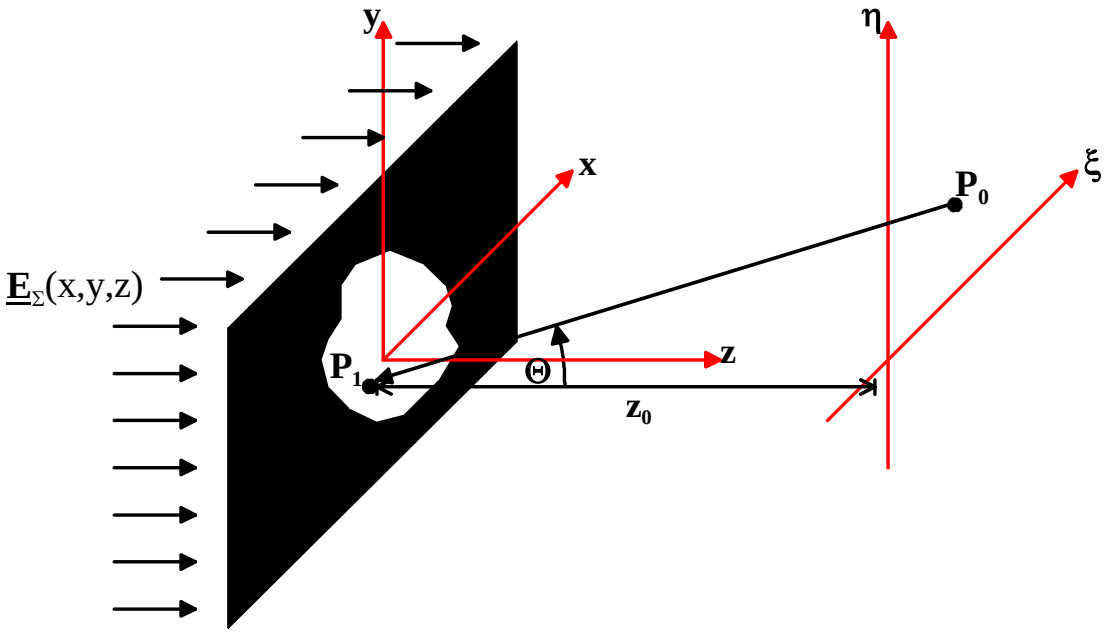


Abbildung 2.5 Darstellung einer Apertur

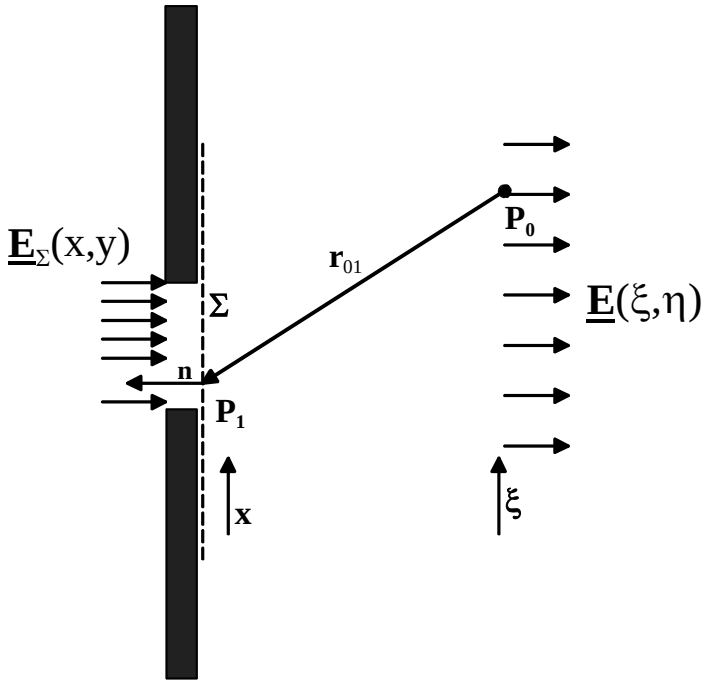


Abbildung 2.6 Abbildung des Radiusvektors

Schreibt man den Radiusvektor, dessen Länge und den Winkel zwischen **n** und $\mathbf{r}_{01}$ ausführlich auf, erhält man

$$\mathbf{r}_{01} = \begin{pmatrix} x-\xi \\ y-\eta \\ -z_0 \end{pmatrix} \tag{2.35}$$

$$r_{01} = \sqrt{(-z_0)^2 + (x-\xi)^2 + (y-\eta)^2} = \sqrt{z_0^2 + (\xi-x)^2 + (\eta-y)^2} \tag{2.36}$$

$$\mathbf{n} \otimes \mathbf{r}_{01} = \frac{\mathbf{r}_{01}\mathbf{n}}{|\mathbf{r}_{01}||\mathbf{n}|} = \frac{\begin{pmatrix} x-\xi \\ y-\eta \\ -z_0 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}}{\sqrt{z_0^2 + (\xi-x)^2 + (\eta-y)^2}} = \frac{z_0}{\sqrt{z_0^2 + (\xi-x)^2 + (\eta-y)^2}} \tag{2.37}$$

Hiermit lässt sich Gleichung (2.34) umformen zu

$$\underline{\mathbf{E}}(\xi,\eta,z_0)=\frac{1}{\mathrm{j}\lambda}\iint_{\Sigma}\underline{\mathbf{E}}_{\Sigma}(x,y,z=0)\underline{h}(\xi-x,\eta-y,z_0)\,dxdy \tag{2.38}$$

mit

$$\underline{h}(\xi-x,\eta-y,z_0)=\begin{cases}\frac{1}{2}\left(1+\frac{z_0}{\sqrt{z_0^2+(\xi-x)^2+(\eta-y)^2}}\right)\frac{\exp\left(\mathrm{j}k\sqrt{z_0^2+(\xi-x)^2+(\eta-y)^2}\right)}{\sqrt{z_0^2+(\xi-x)^2+(\eta-y)^2}} \\ \frac{z_0}{\sqrt{z_0^2+(\xi-x)^2+(\eta-y)^2}}\;\;\frac{\exp\left(\mathrm{j}k\sqrt{z_0^2+(\xi-x)^2+(\eta-y)^2}\right)}{\sqrt{z_0^2+(\xi-x)^2+(\eta-y)^2}} \\ 1\qquad\qquad\frac{\exp\left(\mathrm{j}k\sqrt{z_0^2+(\xi-x)^2+(\eta-y)^2}\right)}{\sqrt{z_0^2+(\xi-x)^2+(\eta-y)^2}}\end{cases}$$

Es ist besonders zu beachten, dass sich alle drei Theorien nur in der Funktion $\underline{h}(\xi\text{-}x,\eta\text{-}y,z_0)$ unterscheiden. Diese Eigenschaft kann bei der Implementierung besonders berücksichtigt werden.
Für die weitere Berechnung der Gleichung (2.38) bestehen zwei Möglichkeiten. Entweder wird die Gleichung exakt berechnet oder es wird eine zusätzliche Approximation durchgeführt.

2.3.1 Exakte Berechnung

Zur exakten Berechnung der Gleichung (2.38) versucht man die besondere Art des Integrals zu nutzen. Dazu dient die Fouriertransformation. Ihre Eigenschaften und Rechenregeln sind im Anhang aufgeführt. Wendet man diese Transformation und die dazugehörige Inverse auf die rechte Seite der Gleichung an, dann ist

$$\underline{\mathbf{E}}(\xi,\eta,z_0)=\mathfrak{I}^{-1}\left\{\mathfrak{I}\left\{\frac{1}{\mathrm{j}\lambda}\iint_{\Sigma}\underline{\mathbf{E}}_{\Sigma}(x,y,z=0)\underline{h}(\xi-x,\eta-y,z_0)\,dxdy\right\}\right\} \tag{2.39}$$

Wird die Eigenschaft der Linearität und der Faltung genutzt, ergibt sich

$$\underline{\mathbf{E}}(\xi,\eta,z_0)=\mathfrak{I}^{-1}\left\{\;\mathfrak{I}\left\{\underline{\mathbf{E}}_{\Sigma}(\xi,\eta,z=0)\right\}\mathfrak{I}\left\{\tfrac{1}{\mathrm{j}\lambda}\,\underline{h}(\xi,\eta,z_0)\right\}\right\} \tag{2.40}$$

$$\text{mit}\quad \underline{h}(\xi-x,\eta-y,z_0)=\begin{cases}\frac{1}{2}\left(1+\frac{z_0}{\sqrt{z_0^2+\xi^2+\eta^2}}\right) & \frac{\exp\left(\mathrm{j}k\sqrt{z_0^2+\xi^2+\eta^2}\right)}{\sqrt{z_0^2+\xi^2+\eta^2}}\\ \frac{z_0}{\sqrt{z_0^2+\xi^2+\eta^2}} & \frac{\exp\left(\mathrm{j}k\sqrt{z_0^2+\xi^2+\eta^2}\right)}{\sqrt{z_0^2+\xi^2+\eta^2}}\\ 1 & \frac{\exp\left(\mathrm{j}k\sqrt{z_0^2+\xi^2+\eta^2}\right)}{\sqrt{z_0^2+\xi^2+\eta^2}}\end{cases}$$

Dabei stellt $\Im\left\{\frac{1}{\mathrm{j}\lambda}\underline{\mathrm{h}}(\xi,\eta,z_0)\right\}$ den Frequenzgang der betrachteten Apertur dar.
Um eine möglichst schnelle Berechnung der drei Theorien zu gewährleisten, ist noch die Anzahl der Multiplikationen auf ein Minimum zu reduzieren. Damit erhält man endgültig

$$\underline{\mathbf{E}}(\xi,\eta,z_0)=\Im^{-1}\left\{\ \Im\{\underline{\mathbf{E}}_\Sigma(\xi,\eta,z=0)\}\,\Im\left\{\frac{1}{\mathrm{j}\lambda}\underline{h}(\xi,\eta,z_0)\right\}\right\} \tag{2.41}$$

$$\text{mit}\quad \underline{h}(\xi-x,\eta-y,z_0)=\begin{cases}\frac{1}{2}\frac{rs+z_0}{s} & \exp(\mathrm{j}krs)\\ \frac{z_0}{s} & \exp(\mathrm{j}krs)\\ \frac{1}{rs} & \exp(\mathrm{j}krs)\end{cases}$$

$$\text{und}\quad s=z_0^2+\xi^2+\eta^2$$

$$\text{und}\quad rs=\sqrt{s}$$

2.3.2 Approximationen

Ausgangspunkt für die Approximationen ist die Gleichung (2.40). Insgesamt werden für die Approximationen zwei unterschiedliche Näherungen durchgeführt. Zuerst betrachtet man den Term im Exponenten. Er oszilliert sehr stark und ist für die Phase des elektrischen Feldes verantwortlich. Hierfür lässt sich ein Wurzelterm $\sqrt{1+x}$ durch eine Reihenentwicklung der Form

$$\sqrt{1+x}\approx 1+\frac{1}{2}x-\frac{1}{8}x^2+\ldots \tag{2.42}$$

darstellen. Wendet man diese Beziehung mit einem Abbruchfehler 2. Ordnung auf die Exponentialfunktion an, erhält man die Näherung für den Exponenten.

$$\sqrt{z_0^{\,2}+\xi^2+\eta^2}=z_0\sqrt{1+\left(\frac{\xi}{z_0}\right)^2+\left(\frac{\eta}{z_0}\right)^2}\approx z_0+\frac{1}{2z_0}\xi^2+\frac{1}{2z_0}\eta^2 \tag{2.43}$$

Der Nenner ist das Ziel der bereits erwähnten zweiten Näherung. Diese Approximation ist als unkritisch anzusehen, da sie nur die Amplitude des elektrischen Feldes betrifft. Dementsprechend ist eine einfache Näherung der Art

$$\sqrt{z_0{}^2 + \xi^2 + \eta^2} \approx |z_0| = z_0 \tag{2.44}$$

ausreichend. Sie schränkt die Gültigkeit der drei Theorien jedoch auf den achsnahen Bereich ein. Die Gleichungen (2.40) sind dann

$$\underline{\mathrm{E}}(\xi,\eta,z_0) = \Im^{-1}\left\{ \Im\{\underline{\mathrm{E}}_\Sigma(\xi,\eta,z=0)\}\,\Im\left\{\tfrac{1}{\mathrm{j}\lambda}\,\underline{\mathrm{h}}(\xi,\eta,z_0)\right\}\right\} \tag{2.45}$$

$$\text{mit}\quad \underline{\mathrm{h}}(\xi,\eta,\mathrm{z}_0) = \begin{cases} \dfrac{1}{z_0}\exp\left(\mathrm{j}k\left(z_0 + \dfrac{1}{2z_0}\xi^2 + \dfrac{1}{2z_0}\eta^2\right)\right) \\ \dfrac{1}{z_0}\exp\left(\mathrm{j}k\left(z_0 + \dfrac{1}{2z_0}\xi^2 + \dfrac{1}{2z_0}\eta^2\right)\right) \\ \dfrac{1}{z_0}\exp\left(\mathrm{j}k\left(z_0 + \dfrac{1}{2z_0}\xi^2 + \dfrac{1}{2z_0}\eta^2\right)\right) \end{cases}$$

Es ist erkennbar, dass sich die drei unterschiedlichen Lösungen im Falle der Approximation zu einer einzigen vereinigen.
Bei einer direkten Berechnung dieser Näherung, lässt sich auch hier die Anzahl der Multiplikationen für die spätere Implementierung reduzieren. Dann gilt

$$\underline{\mathrm{E}}(\xi,\eta,z_0) = \Im^{-1}\left\{ \Im\{\underline{\mathrm{E}}_\Sigma(\xi,\eta,z=0)\}\,\Im\left\{\tfrac{1}{\mathrm{j}\lambda}\,\underline{\mathrm{h}}(\xi,\eta,z_0)\right\}\right\} \tag{2.46}$$

$$\text{mit}\quad \underline{\mathrm{h}}(\xi,\eta,z_0) = \frac{1}{z_0}\exp\left(\frac{\mathrm{j}k}{2z_0}\left(2z_0^2 + \xi^2 + \eta^2\right)\right)$$

Sollte sich ihre Rechengenauigkeit als ausreichend erweisen, so lässt sich die Rechengeschwindigkeit durch eine analytische Berechnung der Fouriertransformierten $\Im\left\{\frac{1}{\mathrm{j}\lambda}\,\underline{h}(\xi,\eta,z_0)\right\}$ weiter beschleunigen. Für diesen Fall ergibt sich

$$\begin{aligned}
\Im\left\{\tfrac{1}{\mathrm{j}\lambda}\,\underline{h}(\xi,\eta,z_0)\right\} &= \Im\left\{\frac{1}{\mathrm{j}\lambda}\frac{1}{z_0}\exp\left(\mathrm{j}k\left(z_0 + \frac{1}{2z_0}\xi^2 + \frac{1}{2z_0}\eta^2\right)\right)\right\} \\
&= \frac{1}{\mathrm{j}\lambda}\Im\left\{\frac{1}{z_0}\exp(\mathrm{j}kz_0)\exp\left(\mathrm{j}\pi\left(\sqrt{\frac{1}{z_0\lambda}}^{\,2}\xi^2 + \sqrt{\frac{1}{z_0\lambda}}^{\,2}\eta^2\right)\right)\right\} \\
&= \frac{1}{\mathrm{j}\lambda}\frac{1}{z_0}\exp(\mathrm{j}kz_0)\Im\left\{\exp\left(\mathrm{j}\pi\left(\sqrt{\frac{1}{z_0\lambda}}^{\,2}\xi^2 + \sqrt{\frac{1}{z_0\lambda}}^{\,2}\eta^2\right)\right)\right\}
\end{aligned}$$

$$= \frac{1}{\mathrm{j}\lambda}\frac{1}{z_0}\exp(\mathrm{j}kz_0)\frac{\mathrm{j}}{\left|\sqrt{\frac{1}{z_0\lambda}}\sqrt{\frac{1}{z_0\lambda}}\right|}\exp\left(-\mathrm{j}\pi\left(\frac{f_\xi^2}{\sqrt{\frac{1}{z_0\lambda}}}+\frac{f_\eta^2}{\sqrt{\frac{1}{z_0\lambda}}}\right)\right)$$

$$= \frac{1}{\mathrm{j}\lambda}\,\mathrm{j}\lambda\exp\left(\mathrm{j}\left(kz_0 - z_0\pi\lambda\left(f_\xi^2 + f_\eta^2\right)\right)\right)$$

$$= \exp\left(\mathrm{j}\left(kz_0 - z_0\pi\lambda\left(f_\xi^2 + f_\eta^2\right)\right)\right) \qquad (2.47)$$

2.3.3 Abbildungsverschiebung

Bisher wurden die diskreten Felder $\underline{\mathbf{E}}_\Sigma(\xi,\eta,z_0{=}0)$ und $\underline{\mathbf{E}}(\xi,\eta,z_0)$ als eine Anzahl von Punkten P_0 und P_1 betrachtet (Gl. 2.34). Die genaue Lage dieser Punkte in der *x*-*y* - und ξ-η - Ebene wurde nicht berücksichtigt. Da sich die Simulation aber auf den Bereich innerhalb der Kamera beschränkt, müssen nur diskrete Werte in diesem Bereich berechnet werden. Der Einfluss eines von außen einstrahlenden Störfeldes kann vernachlässigt werden (Gl. 2.19/20). Die elektrischen Felder besitzen daher, wie in Abbildung 2.7 links erkennbar ist, eine räumliche Ausdehnung in der sich alle Punkte befinden. Die Punkte des berechneten Feldes $\underline{\mathbf{E}}(\xi,\eta,z_0)$ haben aufgrund der Anwendung der **F**ast **F**ourier **T**ransformation (FFT) die Eigenschaft, dass die Punkte P_0 nur durch eine parallel Verschiebung zur z-Achse aus den Punkten P1 entstehen (Gl. 2.41/46).

Mit den bisher aufgezeigten Formeln kann daher ein eingestrahltes und räumlich begrenztes elektrisches Feld nur parallel zur *z*-Achse verschoben und mit der Übertragungsfunktion $\Im\left\{\frac{1}{\mathrm{j}\lambda}\underline{h}(\xi,\eta,z_0)\right\}$ beeinflusst werden. Für eine bestimmte Variante der GPD-Versuchsanordnung ist es jedoch nützlich das zu berechnende Feld $\underline{\mathbf{E}}(\xi,\eta,z_0)$ auch in x- und y-Richtung zu verschieben. Diese Anordnung kann Abbildung 2.7 entnommen werden.

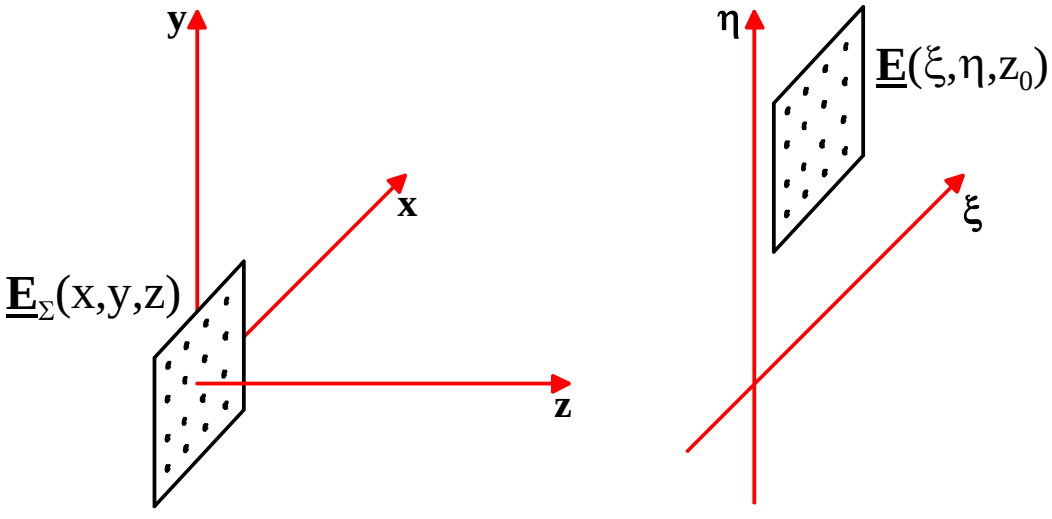


Abbildung 2.7 Verschiebung eines räumlich begrenzten Feldes

Für die Berechnung dieses Feldes muss eine Koordinatentransformation vom ξ-η- in ein ξ_S-η_S - Koordinatensystem durchgeführt werden (Abbildung 2.8).

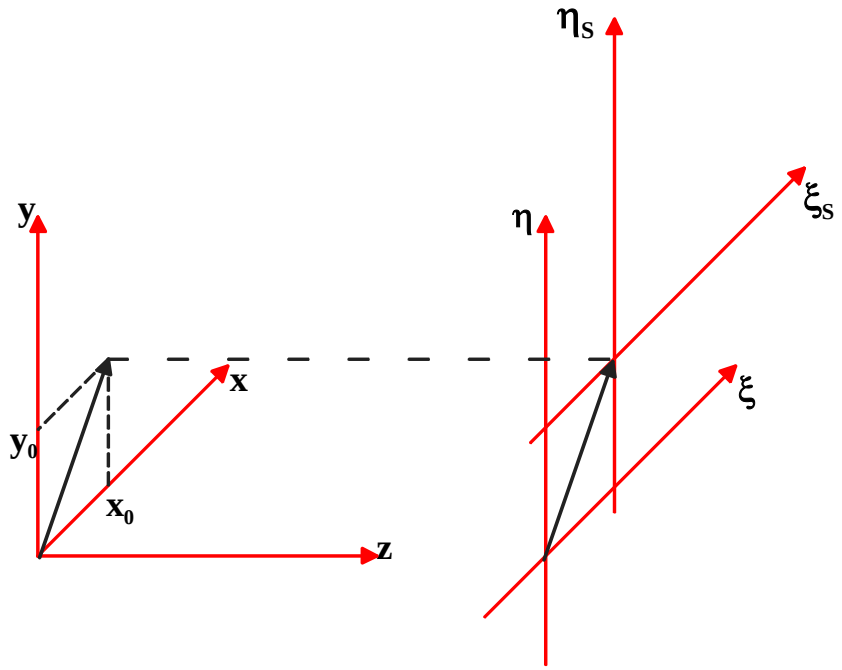


Abbildung 2.8 Koordinatentransformation

Daher gelten folgende Transformationsbeziehungen (s=shift):

$$\left.\begin{aligned}\xi_S &= \xi - x_0\\ \eta_S &= \eta - y_0\end{aligned}\right\} \tag{2.48}$$

Stellt man nun analog zu Gleichung (2.45) die Berechnung für das verschobene Feld auf, erhält man

$$\begin{aligned}\underline{\mathbf{E}}(\xi_S,\eta_S,z_0) &= \underline{\mathbf{E}}(\xi - x_0,\eta - y_0,z_0)\\ &= \Im^{-1}\left\{\ \Im\{\underline{\mathbf{E}}_\Sigma(\xi,\eta,z=0)\}\,\Im\left\{\tfrac{1}{\mathrm{j}\lambda}\,\underline{h}(\xi,\eta,z_0)\right\}\right\}\\ &= \Im^{-1}\left\{\ \Im\{\underline{\mathbf{E}}_\Sigma(\xi,\eta,z=0)\}\,\Im\left\{\tfrac{1}{\mathrm{j}\lambda}\,\underline{h}(\xi_S+x_0,\eta_S+y_0,z_0)\right\}\right\}\end{aligned} \tag{2.49}$$

Wird noch die Linearität und die Verschiebungseigenschaft der Fouriertransformation verwendet, dann ist

$$\underline{\mathbf{E}}(\xi_S,\eta_S,z_0) = \Im^{-1}\left\{\Im\{\underline{\mathbf{E}}_\Sigma(\xi,\eta,z=0)\}\Im\left\{\tfrac{1}{\mathrm{j}\lambda}\,\underline{h}(\xi,\eta,z_0)\right\}\exp\left(\mathrm{j}2\pi\left(x_0 f_\xi + y_0 f_\eta\right)\right)\right\} \tag{2.50}$$

Eine Verschiebung des elektrischen Feldes im Ortsbereich ist infolgedessen identisch mit einer Multiplikation im Frequenzbereich.

3 Linsen

Eine Linse ist ein optische Element, welches die Fähigkeit besitzt einen Teil eines elektrischen Feldes in einer bestimmten Art und Weise zu verändern. Die räumliche Anordnung kann Abbildung 3.1 entnommen werden.

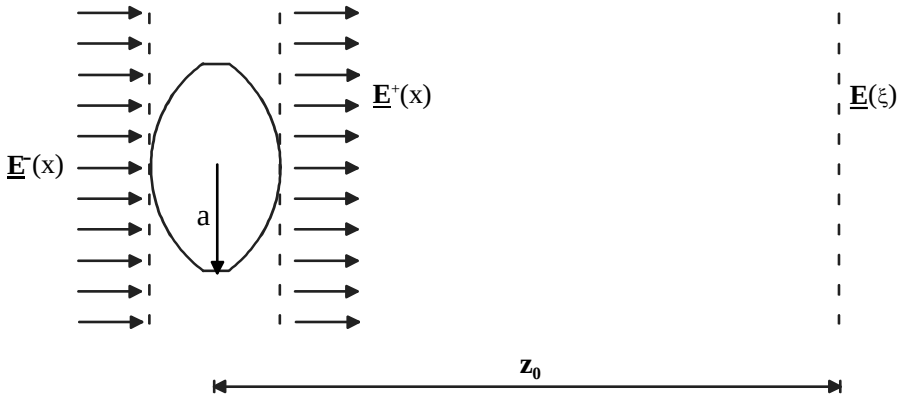


Abbildung 3.1 Darstellung einer Linse und der elektrischen Felder

Geht man davon aus, dass das einfallende Feld durch $\underline{\mathbf{E}}^-$ und das beeinflusste Feld durch $\underline{\mathbf{E}}^+$ repräsentiert werden, dann ist der Quotient aus beiden die sogenannte Transmissionsfunktion $\underline{t}(x)$ der Linse. Das Vorgehen in der Linsenebene lässt sich dann mathematisch vollständig durch

$$\underline{\mathbf{E}}^+(x) = \begin{cases} \underline{t}(x)\,\underline{\mathbf{E}}^-(x) & |x| < a \\ \underline{\mathbf{E}}^-(x) & |x| > a \end{cases} \tag{3.1}$$

beschreiben. Im Bereich $|x|<a$ wird das Feld durch die Transmissionsfunktion verändert, während im Außenbereich der Linse ($|x|>a$) das elektrische Feld ungehindert durch die Linsenebene gelangt. Nach [PAP67] ist der Einfluss des Feldes $\underline{\mathbf{E}}^+$ im Außenbereich $|x|>a$ auf den achsnahen Bereich vernachlässigbar. Die Gleichung (3.1) vereinfacht sich damit zu

$$\underline{\mathbf{E}}^+(x) = \begin{cases} \underline{t}(x)\,\underline{\mathbf{E}}^-(x) & |x| < a \\ 0 & |x| > a \end{cases} \tag{3.2}$$

Diese Vereinfachung ist identisch mit der Annahme, dass eine Linse in einer Apertur eingefasst ist.

Der gesamte Vorgang der Ausbreitung eines elektrischen Feldes durch eine Linse kann somit in drei Abschnitte gegliedert werden:

1. Abschneiden des außen liegenden Feldes $\left(\underline{\mathbf{E}}^{+}(x) = 0 \quad \wedge \quad |x| > a\right)$
2. Veränderung des elektrischen Feldes durch die Transmissionsfunktion $\underline{t}(x)$
3. Ausbreitung des veränderten elektrischen Feldes gemäß den Gleichungen aus Kapitel 2

Im weiteren müssen nur noch die Transmissionsfunktionen für die zu implementierenden Konvex-, Konkav- und Zylinderlinsen beschrieben werden. Es soll an dieser Stelle jedoch nicht unerwähnt bleiben, dass auch Transmissionsfunktionen für weitere Linsenarten und optische Elemente (Bsp. Polarisator,...) existieren.

3.1 Konvex- und Konkavlinsen

Erläutert wird zunächst der Fall einer zweidimensionalen Konvexlinse. Die Konkavlinse wird im Anschluss daran kurz vorgestellt.

Um die Transmissionsfunktion einer Konvexlinse (Abbildung 3.2) zu berechnen, muss das Material und die Form der Linse berücksichtigt werden. Nur diese Einflussgrößen sind in der Lage das elektrische Feld zu verändern.
Die entscheidenden Parameter sind dabei der Brechungsindex *n* und der Dickenverlauf *d(x,y)*.
Näherungsweise wird angenommen, dass diese beiden Faktoren nur die Phasenlage und nicht die Amplitude der einfallenden Welle beeinflussen. Diese Annahmen spezifizieren aber eine „dünne“ Linse. Daher wird der Gültigkeitsbereich der folgenden Transmissionsfunktionen auf diesen Bereich eingeschränkt.

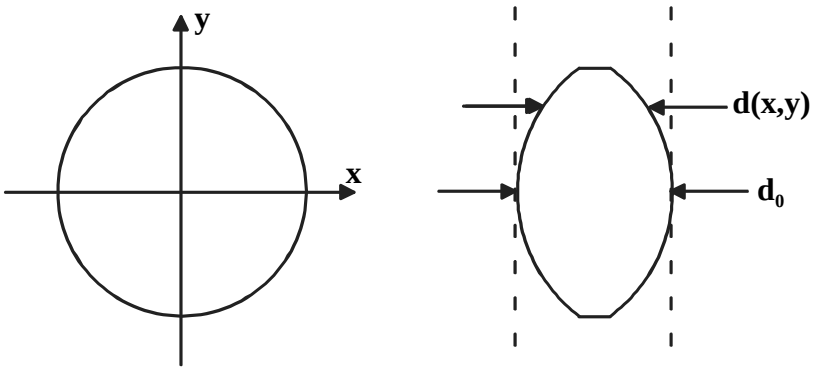


Abbildung 3.2 Schematische Darstellung einer Konvexlinse

Die Phasenlage kennzeichnet die Phase jedes diskreten Punktes in dem eingestrahlten Feld. Bei einer Ausbreitung des elektrischen Feldes durch eine Linse wird die Phase jedes Punktes einzeln verändert. Das Ziel ist es daher den

Phasenunterschied bzw. die Phasenverzögerung zwischen den einzelnen Punkten zu errechnen.

Wird die räumliche Ausbreitung des einfallenden Feldes betrachtet, ergibt sich folgende Besonderheit:
Das einfallende Feld trifft zuerst in der Mitte der Linse auf die Oberfläche. Die entsprechende Welle besitzt damit bei Linsenaustritt eine Phase von kn_Ld_0. Das weiter außenliegende Feld trifft erst verspätet auf die Linsenoberfläche und tritt auch im Vergleich zur Linsenmitte verfrüht aus. Die Phasenverzögerung in Folge des Linsenmediums beträgt $kn_Ld(x,y)$. Die Phasenverzögerung aus dem umgebenden Medium ergibt sich zu $kn_U(d_0\text{-}d(x,y))$. Die gesamte Phasenverzögerung ist dann $kn_Ld(x,y)+ kn_U(d_0\text{-}d(x,y))$.
Die Transmissionsfunktion der Linse erhält man zu

$$\begin{aligned}\underline{t}(x,y) &= \exp\left(\mathrm{j}\left[kn_L d(x,y) + kn_U\left(d_0 - d(x,y)\right)\right]\right) \\ &= \exp(\mathrm{j}kn_U d_0)\exp\left(\mathrm{j}k(n_L - n_U)d(x,y)\right)\end{aligned} \tag{3.3}$$

k, n_U, n_L und d_0 sind bekannte Größen. Der Dickenverlauf $d(x,y)$ muss jedoch noch errechnet werden. Hierzu teilt man die Linse in drei Bereiche auf (Abbildung 3.3). Die Dickenfunktion ergibt sich zu

$$d(x,y) = d_1(x,y) + d_2(x,y) + d_3(x,y) \tag{3.4}$$

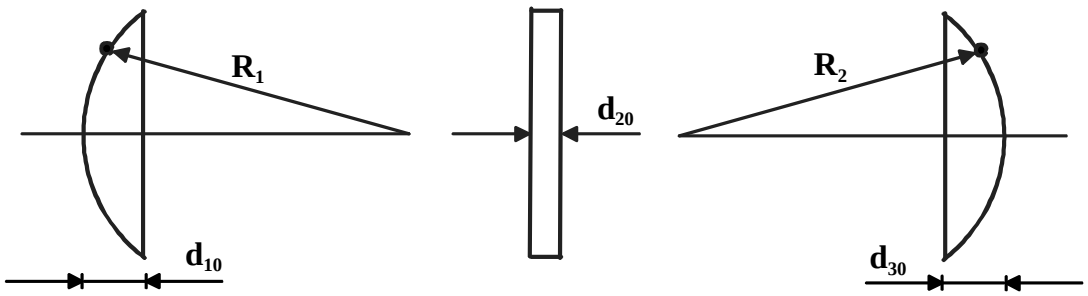


Abbildung 3.3 Dickenverlauf einer Konvexlinse

Stellt man die einzelnen Dickenfunktionen auf, erhält man

$$d_1(x,y) = d_{10} - \left(R_1 - \sqrt{R_1^2 - x^2 - y^2}\right) = d_{10} - R_1\left(1 - \sqrt{1 - \frac{x^2+y^2}{R_1^2}}\right) \tag{3.5}$$

$$d_3(x,y) = d_{30} - \left(-R_2 - \sqrt{R_2^2 - x^2 - y^2}\right) = d_{30} + R_2\left(1 - \sqrt{1 - \frac{x^2+y^2}{R_2^2}}\right) \tag{3.6}$$

Addiert man die einzelnen Terme zur Gesamtdickenfunktion auf, dann ist

$$d(x,y) = d_0 - R_1\left(1 - \sqrt{1 - \frac{x^2 + y^2}{R_1^2}}\right) + R_2\left(1 - \sqrt{1 - \frac{x^2 + y^2}{R_2^2}}\right) \tag{3.7}$$

Diese Funktion kann bereits zur Berechnung der Linse verwendet werden. Aufgrund der Komplexität der Gleichung ist es aber noch hilfreich eine Näherung durchzuführen. Der Wurzelterm lässt sich dann durch eine Reihenentwicklung mit Abbruch nach dem zweiten Term approximieren. Dann gilt

$$\sqrt{1 - \frac{x^2 + y^2}{R^2}} \approx 1 - \frac{x^2 + y^2}{2R^2} \tag{3.7}$$

Arbeitet man diese Näherung in die Gleichung (3.7) ein, ergibt sich

$$d(x,y) = d_0 - \frac{x^2 + y^2}{2}\left(\frac{1}{R_1} - \frac{1}{R_2}\right) \tag{3.9}$$

Zur endgültigen Berechnung der Transmissionsfunktion ist noch die Gleichung (3.9) in (3.3) einzusetzen.

$$\begin{aligned} \underline{t}(x,y) &= \exp(\mathrm{j}kn_U d_0)\exp\left(\mathrm{j}k(n_L - n_U)\left(d_0 - \frac{x^2 + y^2}{2}\left(\frac{1}{R_1} - \frac{1}{R_2}\right)\right)\right) \\ &= \exp(\mathrm{j}kn_U d_0)\exp\left(\mathrm{j}k(n_L - n_U)d_0\right)\exp\left(-\mathrm{j}k(n_L - n_U)\frac{x^2 + y^2}{2}\left(\frac{1}{R_1} - \frac{1}{R_2}\right)\right) \\ &= \exp(\mathrm{j}kn_L d_0)\exp\left(-\mathrm{j}k(n_L - n_U)\frac{x^2 + y^2}{2}\left(\frac{1}{R_1} - \frac{1}{R_2}\right)\right) \end{aligned} \tag{3.10}$$

Den ersichtlichen mathematischen Zusammenhang zwischen den Größen n_L, n_U, R_1 und R_2 im Exponenten kann man in einer weiteren physikalischen Größe zusammenfassen. Dies ist die sogenannte Brennweite. Sie ist definiert als

$$\frac{1}{f} \equiv (n_L - n_U)\left(\frac{1}{R_1} - \frac{1}{R_2}\right) \tag{3.11}$$

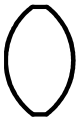
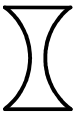
Die Transmissionsfunktion schreibt sich damit verkürzt zu

$$\underline{t}(x, y) = \exp(\mathrm{j}kn_L d_0)\exp\left(-\mathrm{j}\frac{k}{2f}\left(x^2 + y^2\right)\right) \tag{3.12}$$

Diese Gleichung beschreibt nun die weiter oben beschriebene Problematik. Zusammengefasst mit der Gleichung (3.2) erhält man die Transmissionsfunktion einer Konvexlinse.

$$\underline{\mathbf{E}}^{+}(x) = \begin{cases} \exp(\mathrm{j}kn_L d_0)\exp\left(-\mathrm{j}\frac{k}{2f}\left(x^2+y^2\right)\right)\ \underline{\mathbf{E}}^{-}(x) & |x| < a \\ 0 & |x| > a \end{cases} \tag{3.13}$$

Möchte man die Transmissionsfunktionen weiterer Linsen errechnen, so müssen nur in die eben beschriebene Transmissionsfunktion (3.13) vorzeichenbehaftete Radien oder Brennweiten eingefügt werden. Sie können der Tabelle 3.1 entnommen werden. Dabei gilt folgende Vorzeichenkonvention:
Nimmt man an, dass sich die Wellen von links nach rechts durch die Linse fortpflanzen, dann besitzt eine konvexe Oberfläche einen positiven und eine konkave Oberfläche einen negativen Radius.

Linse	Radius R_1	Radius R_2	Brennweite f
	>0	<0	>0
	>0	$\pm\infty$	>0
	>0	$R_2 > R_1$	>0
	<0	>0	<0
	<0	$\pm\infty$	<0
	>0	$R_1 > R_2 > 0$	<0

R_1 = Radius der linken Oberfläche bei einer Wellenausbreitung von links nach rechts
R_2 = Radius der rechten Oberfläche bei einer Wellenausbreitung von links nach rechts

Tabelle 3.1 Tabelle mit den verschiedenen Linsenarten

3.2 Zylinderlinse

Eine Zylinderlinse ist nur ein Spezialfall der bereits beschriebenen Konvexlinse. Für eine konvexe Zylinderlinse müssen alle Gleichungen von Kapitel 3.1 in den ebenen Fall übertragen werden. Betrachtet man gleich die endgültige Gleichung (3.13) ist dies einfach durch weglassen einer Koordinate erreichbar. Die Transmissionsfunktion einer konvexen Zylinderlinse beträgt somit

$$\underline{\mathbf{E}}^{+}(x) = \begin{cases} \exp(\mathrm{j}kn_L d_0)\exp\left(-\mathrm{j}\frac{k}{2f}x^2\right)\ \underline{\mathbf{E}}^{-}(x) & |x| < a \\ 0 & |x| > a \end{cases} \tag{3.14}$$

Analog zu Kapitel 3.1 kann man durch einsetzen vorzeichenbehafteter Werte andere Formen einer Zylinderlinse darstellen. Dies kann ebenfalls der in der bereits aufgeführten Tabelle 3.1 nachgeschlagen werden.

4 Programmentwurf

In diesem Kapitel werden die objektorientierte Analyse, der Klassenentwurf und der Programmablauf erläutert. Ausgangspunkt hierfür sind die Gleichungen (4.1) und (4.2).

$$\begin{aligned}\underline{E}_S(\xi_S,\eta_S,z_0) &= \Im^{-1}\left\{\Im\left\{\underline{E}_\Sigma^+(\xi,\eta,z=0)\right\}\Im\left\{\tfrac{1}{\mathrm{j}\lambda}\,\underline{h}(\xi,\eta,z_0)\right\}\exp\left(\mathrm{j}2\pi\left(x_0 f_\xi + y_0 f\eta\right)\right)\right\} \\ &= \Im^{-1}\left\{\Im\left\{\underline{E}_\Sigma^+(\xi,\eta,z=0)\right\}\underline{H}(\xi,\eta,x_0,y_0,z_0)\right\}\end{aligned} \tag{4.1}$$

mit

$$\underline{h}(\xi,\eta,z_0)=\begin{cases}\dfrac{1}{2}\dfrac{\sqrt{z_0^2+\xi^2+\eta^2}+z_0}{z_0^2+\xi^2+\eta^2}\exp\left(\mathrm{j}k\sqrt{z_0^2+\xi^2+\eta^2}\right) & \text{Kirchhoff – Theorie}\\ \dfrac{z_0}{z_0^2+\xi^2+\eta^2}\exp\left(\mathrm{j}k\sqrt{z_0^2+\xi^2+\eta^2}\right) & \text{Erste RayleighSommerfeld}\\ \dfrac{1}{z_0^2+\xi^2+\eta^2}\exp\left(\mathrm{j}k\sqrt{z_0^2+\xi^2+\eta^2}\right) & \text{Zweite RayleighSommerfeld}\\ \dfrac{1}{z_0}\exp\left(\frac{\mathrm{j}k}{2z_0}\left(2z_0^2+\xi^2+\eta^2\right)\right) & \text{Approximation}\end{cases}$$

$$\underline{E}_\Sigma^+(\xi,\eta,\mathrm{z}=0)=\begin{cases}\underline{E}(\xi,\eta,z=0) & \mathrm{P}(\xi,\eta,\mathrm{z}=0)\in\Sigma\\ 0 & \mathrm{P}(\xi,\eta,\mathrm{z}=0)\notin\Sigma\end{cases}\Big\}\text{Apertur}$$

$$\underline{E}_\Sigma^+(\xi,\eta,\mathrm{z}=0)=\begin{cases}\mathrm{t}(\xi,\eta)\underline{E}(\xi,\eta,z=0) & \mathrm{P}(\xi,\eta,\mathrm{z}=0)\in\Sigma\\ 0 & \mathrm{P}(\xi,\eta,\mathrm{z}=0)\notin\Sigma\end{cases}\Big\}\text{Linse}$$

$$\begin{aligned}I(x,y) &= \frac{1}{2}\varepsilon c\left(\left|\underline{E}_x(x,y)\right|^2+\left|\underline{E}_y(x,y)\right|^2\right)\\ &= \frac{1}{2}\varepsilon c\left(\left(\sqrt{\mathrm{Re}\{\underline{E}_x\}^2+\mathrm{Im}\{\underline{E}_x\}^2}\right)^2+\left(\sqrt{\mathrm{Re}\{\underline{E}_y\}^2+\mathrm{Im}\{\underline{E}_y\}^2}\right)^2\right)\\ &= \frac{1}{2}\varepsilon c\left(\mathrm{Re}\{\underline{E}_x\}^2+\mathrm{Im}\{\underline{E}_x\}^2+\mathrm{Re}\{\underline{E}_y\}^2+\mathrm{Im}\{\underline{E}_y\}^2\right)\end{aligned} \tag{4.2}$$

Gleichung (4.1) ist aus den bereits besprochenen Gleichungen (2.40), (2.44), (2.48) und (3.2) hervorgegangen. Sie beschreibt die Änderungen eines elektrischen Feldes, die durch optische Elemente verursacht werden. Zu beachten ist, dass das Licht als elektrisches Feld zwei Komponenten $\underline{E}_x$ und $\underline{E}_y$ besitzt. Gleichung (4.1) muss infolgedessen auf beide Komponenten angewendet werden. Zusätzlich ist der Einfluss der Verschiebung und die Funktion $\Im\left\{\frac{1}{\mathrm{j}\lambda}\underline{h}(\xi,\eta,z_0)\right\}$ in einer Funktion $\underline{H}(\xi,\eta,x_0,y_0,z_0)$ zusammengefasst worden. Die mathematische Bedeutung dieser

Funktion $\underline{H}$ wird ersichtlich, wenn Gleichung (4.1) durch eine Äquivalenzumformung in einer anderen Form geschrieben wird.

$$\underline{H}(\xi,\eta,x_0,y_0,z_0) = \Im\left\{\frac{1}{\mathrm{j}\lambda}\underline{h}(\xi,\eta,z_0)\right\}\exp\left(\mathrm{j}2\pi\left(x_0 f_\xi + y_0 f\eta\right)\right) = \frac{\Im\left\{\underline{E}_S(\xi_S,\eta_S,z_0\right\}}{\Im\left\{\underline{E}_\Sigma^+(\xi,\eta,z=0)\right\}} \quad (4.3)$$

In Analogie zur Regelungstechnik ergibt sich, dass die Funktion $\underline{H}(\xi,\eta,x_0,y_0,z_0)$ eine Übertragungsfunktion des optischen Elements ist. Sie ist für jedes Element berechenbar und zeitlich konstant, d.h. ist die Übertragungsfunktion für ein optisches Element einmal bestimmt worden, so kann sie auf jedes weitere elektrische Feld angewendet werden. Auf diese Weise interagiert jedes optische Element mit einem elektrischen Feld und verändert dieses.
Es soll jedoch nicht unerwähnt bleiben, dass erst das in den Frequenzbereich transformierte elektrische Feld $\underline{E}_\Sigma^+$ dem Eingangsignal des Systems und das ebenfalls transformierte Feld $\underline{E}_S$ dem Ausgangssignal des Systems entspricht.

Eine ähnliche Betrachtung wie bei der Übertragungsfunktion kann auch mit der Transmissionsfunktion der Linse durchgeführt werden. Hier findet die mathematische Operation aber nicht im Frequenzbereich, sondern im Ortsbereich statt. Als Gleichung erhält man

$$\underline{t}(\xi,\eta) = \frac{\underline{E}_\Sigma^+(\xi,\eta,z=0)}{\underline{E}(\xi,\eta,z=0)} \quad (4.4)$$

Auch hier ist die Funktion $\underline{t}(\xi,\eta)$ jeder Linse eigen. Außerdem muss sie analog zur Übertragungsfunktion nur einmal berechnet werden und kann dann auf jedes elektrische Feld angewendet werden.

Die Detektion der elektrischen Feldstärke wird üblicherweise durch elektrooptische Komponenten, wie PIN-Dioden, CCD-Zeilen o.ä. realisiert. Dabei wird lediglich die Intensität des elektrischen Feldes nach Gleichung (4.2) registriert. Für die Simulation muss somit die Umrechnung der elektrischen Feldstärke in Intensität oder Leistung in einem Programmteil, z.B. einem Schirm realisiert werden. Diese Intensität ist die Eigenschaft, die das menschliche Auge wahrnehmen kann. Sie errechnet sich aus der Addition der Beträge der einzelnen Felder ($\underline{E}_x$, $\underline{E}_y$) multipliziert mit einem konstanten Vorfaktor. Der Faktor $^1/_2\varepsilon c$ ergibt sich durch die Tatsache, dass die Intensität physikalisch nicht durch die Beträge der einzelnen Felder errechnet wird, sondern aus der zeitlichen Mittelung des Poyntingvektors.
Nähere Informationen zu diesem Vorfaktor kann [HEC98] oder [TRO00] entnommen werden.

4.1 Objektorientierte Analyse, Design und Klassenentwurf

Bevor die Gleichungen (4.1) und (4.2) in irgendeiner Form in der geforderten Programmiersprache C++ implementiert werden, ist zunächst eine objektorientierte Analyse notwendig. Diese Analyse dient dazu einen Überblick über das umzusetzende Problem zu gewinnen.
Da in der Literatur für diese Analyse zum Teil Begriffe mit leicht unterschiedlicher Bedeutung verwendet werden, sollen die Schlüsselwörter, die im weiteren Verlauf der Ausarbeitung benötigt werden, kurz erläutert werden. Die Erläuterungen werden dabei zum Teil direkt der einschlägigen Literatur entnommen.
Die Notation bei dieser Begriffsbeschreibung und im Verlauf der weiteren Ausarbeitung ist wie folgt definiert: Innerhalb des Textkörpers werden Funktionen und Variablen **fett** geschrieben, Programmzeilen hingegen in der Schriftart Arial. Datentypen, Bezüge auf Programmzeilen und Dateien erfolgen in der Schriftart `CourierNew`.

Klassen: Klassen umfassen und repräsentieren die Definition von Datentypen mit gemeinsamen Eigenschaften (Daten, Elementvariablen) und Verhaltensweisen (Methoden) [LOU99]. Bei der Definition einer Klasse wird noch kein Speicher belegt.

Instanzen / Objekte: Jede Instanz (Objekt) einer Klasse besitzt ihre eigene Kopie der Elementvariablen, die in der Klasse definiert sind. Objekte / Instanzen einer Klasse bezeichnen real-existierende deklarierte Datentypen (Beispiel: Rechteckapertur).

Datenelemente / Elementvariablen: Sie sind Variablen / Objekte, die in einer Klasse deklariert sind und beschreiben die Eigenschaften (Daten) der Klasse.

Methoden / Funktionen: Sie sind Teil einer Klasse. Sie bestimmen, was die Objekte dieser Klasse leisten können und beschreiben auf diese Art das Verhalten der Klasse.

Abstrakte Klassen: Als abstrakte Klasse bezeichnet man eine Klasse, die mindestens eine abstrakte Methode enthält. Von einer abstrakten Klasse kann keine Instanz erzeugt werden. Es ist zu beachten, dass Klassen, die von einer abstrakten Klasse abgeleitet wurden und für die vergessen wurde, die abstrakten Methoden zu überschreiben, selbst zu abstrakten Klassen werden.

Vererbung: Klassen werden in Klassenhierarchien zusammengefasst. Abgeleitete Klassen können die Eigenschaften der übergeordneten Klasse übernehmen (erben).

Kapselung: Sie ist Eigenschaft der objektorientierten Programmierung. Sie bildet reale Objekte mit ihren Eigenschaften (Daten) und ihrem Verhalten (Methoden) in einem Programm ab. Dabei werden die Eigenschaften und das Verhalten der realen Objekte in einer Klasse, die dieses Objekt repräsentiert, zusammengefasst.

Überladung: Als Überladung werden im weiteren zwei Besonderheiten der objektorientierten Programmierung bezeichnet. Erstens die Zuweisung mehrerer Operationen an einen Operator und zweitens die Mehrfachdefinitionen einer Funktion in einer Klasse [LOU99]. Die überladenen Operatoren und Funktionen müssen daher in einem gemeinsamen Gültigkeitsbereich vorliegen. Bei diesen Mehrfachdefinitionen entscheidet der Compiler bei Funktions-, bzw. Operatoraufruf

anhand der Parameteranzahl und –typen, welche Funktion bzw. welcher Operator aufgerufen wird [AUP97].

Überschreibung: Als Überschreibung wird die Neudefinition einer ererbten (möglicherweise auch überladenen) Funktion in einer abgeleiteten Klasse bezeichnet. Ziel ist es, dass die abgeleitete Klasse in spezifischer Weise auf den Funktionsaufruf reagiert. Diese Funktion muss jedoch als `virtual` definiert sein.

Virtuelle Methoden: Der Begriff `virtual` wird im Bereich der Vererbung verwendet und kennzeichnet die späte Bindung einer Methode [LOU99] [AUP97].

Die Funktionalität der Überladung, der Überschreibung und der virtuellen Methoden sollen an einem Beispiel kurz darglegt werden. Hierzu werden in einer Klasse `Basis` die Funktionen **fA** und **fB** definiert und mit gleichnamigen Funktionen überladen.

```
class Basis {
  public:
    void fA (int a, int b)            { cout << ”Dies ist Basis:fA (int a, int b).\n”; };
    void fA (double a, double b)   { cout << ”Dies ist Basis:fA (double a, double b).\n”; };
    virtual void fB (int a, int b)    { cout << ”Dies ist Basis:fB (int a, int b).\n”; };
    void fB (double a, double b)   { cout << ”Dies ist Basis:fB (double a, double b).\n”; };
};
```

Gleichzeitig wird eine Klasse `Abgeleitet` mit der überschriebenen Funktion **fB (int a, int b)** und die Funktion **main** wie folgt definiert:

```
class Abgeleitet : public Basis {
  public:
    void fA (int a, int b)            { cout << ”Dies ist Abgeleitet:fA (int a, int b).\n”; };
    void fA (double a, double b)   { cout << ”Dies ist Abgeleitet:fA (double a, double b).\n”; };
    virtual void fB (int a, int b)    { cout << ”Dies ist Abgeleitet:fB (int a, int b).\n”; };
    void fB (double a, double b)   { cout << ”Dies ist Abgeleitet:fB (double a, double b).\n”; };
};


#include <iostream>
using namespace std;

int main() {
  Abgeleitet* Ab = new(Abgeleitet);
  Basis* Ba = Ab;
  Ba->fA(2, 3);
  Ba->fA(2.0, 3.0);
  Ba->fB(2, 3);
  Ba->fB(2.0, 3.0);
  return 0;
}
```

Bei einer Programmausführung erhält man dann eine Bildschirmausgabe in der Form

```
> Dies ist Basis:fA (int a, int b).
> Dies ist Basis:fA (double a, double b).
> Dies ist Abgeleitet:fB (int a, int b).
> Dies ist Basis:fB (double a, double b).
```

Auf eine nähere Erläuterung der Funktionsweise von einer Überladung, Überschreibung und virtuelle Methode soll an dieser Stelle verzichtet und statt dessen auf die bereits erwähnte Literatur [LOU99] [AUP97] verwiesen werden.

Abstrakte Methoden / rein virtuelle Methode: Methoden, die als `virtual` deklariert und auf 0 gesetzt sind.

```
virtual Typ name (Parameterliste) = 0;
```

Polymorphismus: Erlaubt es, durch die zuvor definierten Eigenschaften der objektorientierten Programmierung (Überschreibung und virtuelle Methode) auf unterschiedliche Objekte gleichlautende Methoden auszuführen.

Referenzen: Referenzen verweisen wie Zeiger auf Objekte. Sie können (und müssen) mit einem Objekt initialisiert und danach nicht mehr auf andere Objekte umgelenkt werden. Sie werden wie ganz normale Variablen, quasi als Synonyme der Variablen, auf die sie verweisen, verwendet. Beispiel: Wird bei einer Methode eine Parameterübergabe mit Hilfe einer Referenz durchgeführt, so werden innerhalb der Funktion die entsprechenden Werte bei Zugriff auf die Referenz direkt verändert.

Mit Hilfe dieser definierten Schlüsselwörter lässt sich die folgende objektorientierte Analyse leichter beschreiben. Bei der zu entwerfenden Klassenstruktur soll insbesondere die Methode der Kapselung angewendet werden. Um diese Bedingung zu erfüllen, ist gleichzeitig eine Analyse der real existierenden Objekte notwendig.
Wird zuerst die physikalische Anordnung der Aperturen, Linsen und CCD-Chips als eine Einheit betrachtet, erkennt der Anwender nur eine Kamera die aus zwei einfallenden elektrischen Feldern eine Intensitätsverteilung errechnet. Diese Kamera wird bereits als erste Klasse definiert (siehe Abbildung 4.1).
Löst man das reale Objekt der „Kamera“ auf, d.h. wird das Gehäuse der Kamera entfernt, so sind einzelne Kameraelemente innerhalb der Kamera sichtbar. Die Kamera enthält daher mehrere Kameraelemente. Diese Elemente werden im Programm ebenfalls als eigener Klassentyp dargestellt. Wird auch das reale Objekt des Kameraelements näher betrachtet, beobachtet man, dass als nächstes ein optisches Element und ein Schirm zu erkennen sind. Der physikalische Unterschied zwischen diesen beiden Elementtypen ist, dass einerseits das elektrische Feld manipuliert und hindurchgelassen wird, während im Falle des Schirms über die Detektoren ein Bild entsteht.
Führt man dieses Vorgehen innerhalb der Kamera weiter durch, werden zuerst die Aperturen und Linsen und anschließend die Rechteck-, Kreisapertur und die Konvex-, Konkav- und Zylinderlinse sichtbar. Diese Objekttypen können daher als eine Spezialisierung eines Kameraelements betrachtet werden. Infolgedessen bilden sie mit Hilfe der Methode der Vererbung eine eigenständige Klassenhierarchie. An dieser Stelle ist zu erwähnen, dass sich die Eigenschaften einer Konvex- und einer Konkavlinse nur dadurch unterscheiden, dass die Brennweite ein anderes Vorzeichen besitzt. Dieses Vorzeichen wird später vom Anwender mit Hilfe einer Eingabedatei (`Camera.dat`), in der alle wesentlichen Objektmerkmale definiert werden, festgelegt. Er entscheidet daher, um welche Linsenart es sich handelt. Aus diesem Grund können diese beiden Klassen zusammengefasst werden, damit im Verlauf der Programmierung ein doppelter Programmcode in den sonst identischen Klassen vermieden wird.
Um diese Klassenstruktur weiter zu verfeinern, muss zunächst von der anschaulichen Analyse der Kamera weggegangen und die Gleichung 4.1 näher betrachtet werden. In dieser Gleichung werden elektrische Felder, die Übertragungsfunktion und die Transmissionsfunktion verwendet. Diese Felder und Funktionen besitzen mehrere gemeinsame Eigenschaften. Die wichtigste Eigenschaft ist die, dass die Daten in

Form diskreter Werte vorliegen, die in einem Array abgelegt werden können. Sie können daher als eine abgeleitete Klasse von einer Klasse „Feld“ gedeutet werden.
Wie bereits erläutert wurde, ist die Übertragungsfunktion jedem optischem Element und die Transmissionsfunktion jeder Linse eigen, d.h. sie beschreiben die optischen Eigenschaften der entsprechenden Klasse. Diese Funktionalitäten liegen daher als Elementvariablen in diesen Klassen vor. Anstatt in diesen Klassen einzelne Datenelemente zur Beschreibung der Funktionalitäten zu verwenden, ist es übersichtlicher diese Eigenschaften in eigenen Klassentypen („Übertragungsfunktion / CTransferFunction“, „Transmissionsfunktion / CTransmissionFunction“) zu kapseln.

Um die objektorientierte Analyse abschließen zu können, ist – wie in Abbildung 4.1 sichtbar ist – noch die Funktion der Klasse „Plan“ zu erklären. Dabei sind einige Eigenschaften des Programms vorwegzunehmen.
Wie bekannt ist, müssen bei der Berechnung nach Gleichung (4.1) mehrere Fouriertransformationen durchgeführt werden. Diese Transformationen werden mit Hilfe der FFTW (**F**astest **F**ourier **T**ransform in the **W**orld) berechnet. Sie liegt der Studienarbeit in elektronischer Form (Dokumentation und Programmcode) bei. Die FFTW verwendet den Algorithmus einer FFT. Bei dieser FFT wird vor der eigentlichen Berechnung ein Plan erstellt. Durch Benutzung dieses Plans bei der Transformation, wird diese erheblich beschleunigt. Nachteilig ist, dass ein Plan sowohl für die Vorwärts-, als auch für die Rückwärtstransformation erstellt werden muss. Außerdem sind diese Pläne nur für eine vorgegebene Anzahl von Stützstellen optimiert und einsatzfähig. Aus diesem Grund soll eine eigene Klasse die Verwaltung und damit die Berechnung des Plans übernehmen. Diese Klasse „Plan“ wird dabei als Basisklasse von der Klasse „Feld“ angenommen. Die genaue Begründung für diese Anordnung in der Klassenstruktur wird im weiteren Verlauf der Dokumentation ersichtlich.

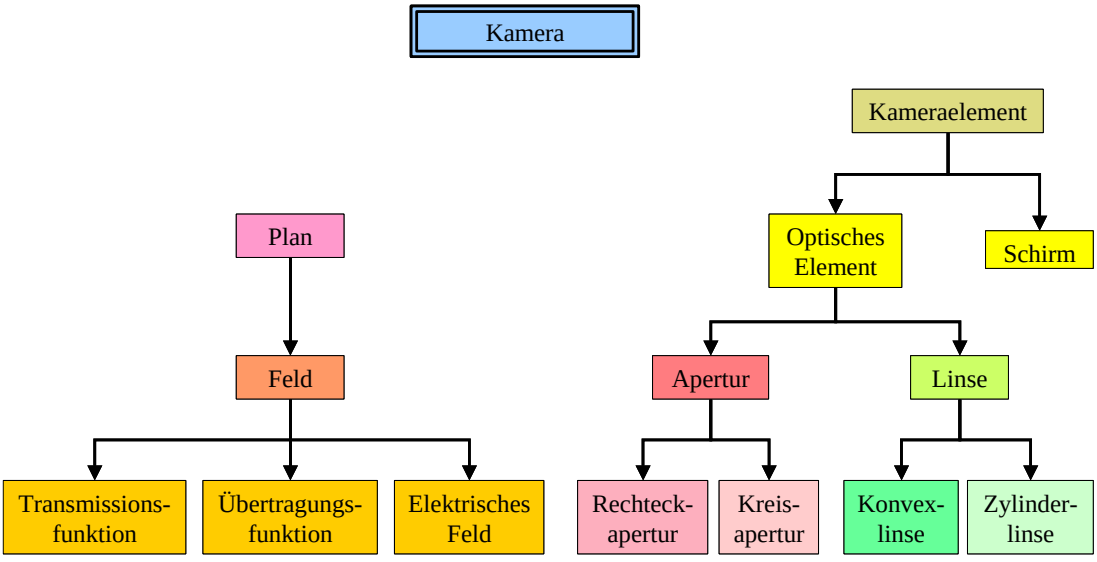


Abbildung 4.1 Darstellung der vollständigen Klassenhierarchie

Nach dieser vollständigen objektorientierten Analyse und Festlegung des Designs wird als nächstes der Klassenentwurf jeder einzelnen Klasse erläutert.

Dabei wird mit der Klasse „Kamera“ begonnen. Sie wurde bei der Analyse als ein reales Objekt beschrieben, welches zwei elektrische Eingangsfelder vom Typ „Elektrisches Feld“ enthält und diese mit Hilfe der Kameraelemente manipuliert. Die Kamera enthält daher zwei elektrische Felder und die in der Kamera physikalisch vorhandenen Kameraelemente als Elementvariablen. Über die beiden Felder verfügt sie von Beginn an. Die Kameraelemente werden aber erst zur Laufzeit erstellt, d.h. der Anwender gibt die Kameraelemente in einer Datendatei (`Camera.dat`) vor. Diese Datendatei ist im Anhang abgedruckt. Die Aufgabe der Kamera ist es diese Werte einzulesen und die passenden Elemente zu erzeugen. Daher wird eine Methode **read** benötigt. Der Parameter dieser Funktion wird bei der Besprechung des Gültigkeitskonzepts näher erläutert. Die Funktion **read** liest die eigenen Datenelemente ein. Dazu gehören die Anzahl der Kameraelemente und die Wellenlänge mit der die Anordnung der Elemente berechnet werden soll. Danach wird der Klassentyp des ersten Kameraelements eingelesen und erzeugt. Diese Erzeugung erfolgt mit Hilfe des Standardkonstruktors. Die Adresse des Elements wird dabei in einem Pointerarray festgehalten, damit im weiteren Verlauf des Programms noch auf dieses Element zugegriffen werden kann. Das anschließende Einlesen der elementspezifischen Datenelemente ist Aufgabe des Elements selber, d.h. das Kameraelement besitzt ebenfalls eine Funktion **read**, die die weiteren eigenen Daten einliest. Dieser Vorgang wird mit allen Kameraelementen durchgeführt.

Nachdem auf diese Art die Kameraelemente mit ihren Eigenschaften in der Klasse „Kamera“ vorliegen, kann mit der Beschreibung der eigentlichen Berechnung begonnen werden. Die Berechnung wird von zwei Funktionen durchgeführt. Dies sind die Methoden **set** und **calc**. Die Aufgabe der Funktion **set** ist es, die größte räumliche Ausdehnung der Kameraelemente in *x*- und *y*-Richtung herauszufinden und in den Elementvariablen **MAXWidthX** und **MAXHeightY** abzuspeichern.
Der Grund für dieses Vorgehen wird ersichtlich, wenn zunächst nur die Grundstruktur eines Berechnungsablaufs betrachtet wird. Bei einem Berechnungsablauf wird zuerst ein elektrisches Feld mit der Größe des ersten Kameraelements angefordert. Eine Anforderung eines größeren Feldes wäre verschwendete Rechenzeit, weil der überstehende Teil vom ersten Kameraelement abgeschnitten würde. Zu beachten ist aber, dass ein Teil der folgenden Berechnung aus einer Fouriertransformation besteht. Es sei hier darauf hingewiesen, dass die elektrischen Felder im Programm nur durch diskrete Werte dargestellt werden. Bei der eigentlichen Fouriertransformation, bestehend aus Hin- und Rücktransformation, ist daher die Anzahl der diskreten Werte und die räumliche Abtastung gleichbleibend. Die Größe und Auflösung des elektrischen Feldes für jedes Kameraelement innerhalb der Kamera ist damit konstant. Wird nur ein elektrisches Eingangsfeld mit der Größe des ersten Kameraelements angefordert, so füllt dieses das größte Element in der Kamera nicht vollständig aus. Eine Möglichkeit wäre, ein entsprechend großes Eingangsfeld anzufordern. Der größte Teil des Feldes würde jedoch am ersten Element abgeschnitten. Infolge dessen wird nur ein elektrisches Eingangsfeld von der Größe des ersten Kameraelements angefordert und anschließend auf die maximale Größe mit Nullen aufgefüllt. Damit wird erreicht, dass bei der Berechnung sowohl am ersten als auch am größten Element ein vollständiges Feld vorliegt.

Auf weitere Aufgaben der Funktion **set** soll erst bei der Erläuterung des Gültigkeitskonzepts eingegangen werden. Die weitere Berechnung der Kamera erfolgt nun vollständig in der Methode **calc**. Die Aufgabe dieser Funktion lässt sich in vier Teile gliedern. Zuerst muss ein Eingangsfeld mit der Größe des ersten Kameraelements angefordert werden. Anschließend wird dieses Feld bis zur maximalen Ausdehnung der Kamera mit Nullen aufgefüllt. Zur Kontrolle dieses Vorgangs existiert die Statusvariable **EFillUp**. Sie ist Teil der Struktur `status` (`Camera.h:87-90`). Ist ihr Wert true, so wurde das Feld bereits aufgefüllt. Die dritte Aufgabe ist es, dem elektrischen Feld seine Wellenlänge zuzuweisen. Dabei sei vorweggenommen, dass die Wellenlänge Eigenschaft des elektrischen Feldes und daher Elementvariable der Klasse „Elektrisches Feld“ ist.
Die letzte Verantwortung, die die Funktion **calc** zu erfüllen hat, besteht darin eine bestimmte Methode jedes Kameraelements aufzurufen und die elektrischen Felder $\underline{E}_x$ und $\underline{E}_y$ als Parameter mit zu übergeben. Die Aufgabe dieser Methode der Kameraelemente ist es, diese elektrischen Felder gezielt nach den Gleichungen (4.1) und (4.2) zu verändern und, wenn gewünscht, das Ergebnis auszugeben. Die Funktion in der Klasse „Kameraelement“ besitzt entsprechend den Namen **calc**.

Nach Beschreibung dieser drei Hauptfunktionen der Klasse „Kamera“ soll abschließend noch die Funktion **writeCamera** erläutert werden. Sie wird aufgerufen, wenn die noch nicht beschriebene Ausgabevariable **writeCamera** in der Struktur `output` (`Camera.h:82-84`) den Wert true besitzt. Der Wert der Variablen wird vom Anwender in der Eingabedatei festgelegt. Die Aufgabe der Methode ist es die Datenelemente der Kamera und der Kameraelemente in eine Datei `Kameradat.dat` auszugeben. Dabei werden die Daten in diese Datei genau so geschrieben, wie sie aus der Eingabedatei gelesen wurden. Zu beachten ist, dass die Ausgabe nicht direkt in eine Datei geschrieben wird, sondern zunächst nur in ein Ausgabeobjekt vom Typ COut. Diese Klasse verwaltet selbständig das Öffnen der Datei, das Schreiben der Daten und das Schließen der Datei. Dem COut-Objekt muss daher nur zu Beginn der Ausgabe einmal der Dateiname mitgeteilt werden. Dieser Vorgang erfolgt bereits beim Aufruf des Standardkonstruktors der Klasse „Kamera“.

Die verschiedenen Methoden und Elementvariablen der Klasse „Kamera“ sind in Abbildung 4.2 dargestellt.

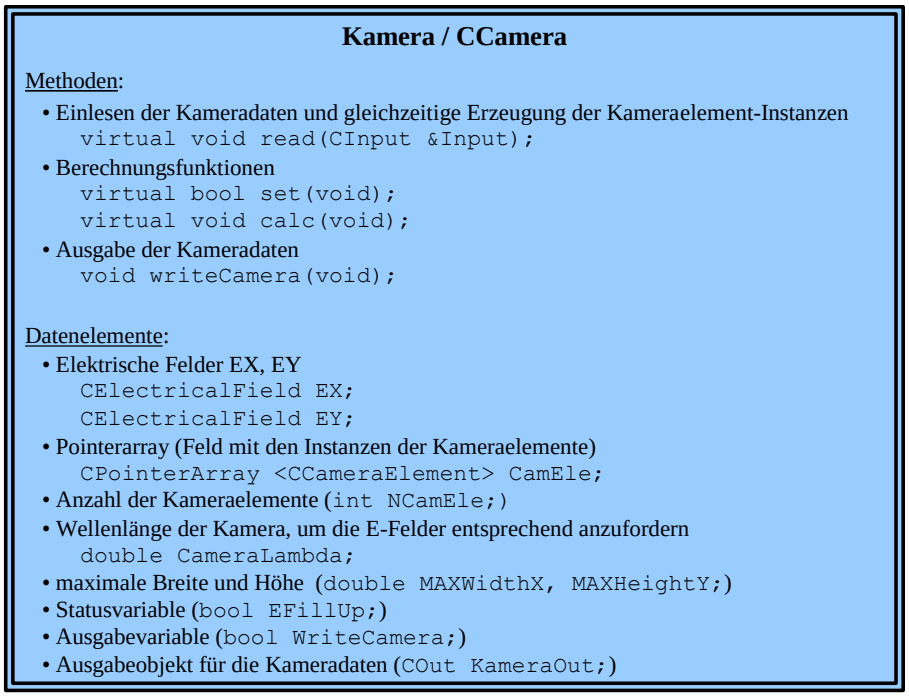


Abbildung 4.2 Entwurf der Klasse „Kamera“

Nach der Beschreibung der Klasse „Kamera“ kann nun die erste Klassenhierarchie erläutert werden. Dabei wird mit den Klassen der unterschiedlichen Felder und des Plans begonnen.

Die Klasse „Plan“ wird – wie bereits beschrieben wurde – benötigt, um die Pläne für die FFT zu verwalten. Sie besitzt einen Plan für die Vorwärts- (**PlanForward**) und einen Plan für die Rückwärtstransformation (**PlanBackward**). Sie sind als Datenelemente in Abbildung 4.3 dargestellt. Da diese Pläne nur für eine spezifische Größe verwendet werden dürfen, können sie sowohl für die Transformation des elektrischen Feldes als auch für die Transformation der Übertragungsfunktion verwendet werden. Dies ist möglich, weil beide Felder bei der Berechnung miteinander elementweise multipliziert werden und daher dieselbe Anzahl von Stützstellen besitzen. Infolge dessen ist die Speicherklasse der Pläne vom Typ `static`. Die Pläne selber sind dabei vom Typ `fftwnd_plan`, welcher von der Klasse `FFTW` zur Verfügung gestellt wird. Näheres hierzu kann [FRI99] entnommen werden. Weiterhin existiert eine Variable **PlanValid** vom Typ `bool`, die angibt, ob die derzeitigen Pläne gültig sind oder nicht. Sie ist analog zu den Plänen ebenfalls als `static` deklariert.

Als Methoden werden nur zwei Funktionen in der Klasse „Plan“ benötigt. Eine dient zur Zerstörung und die andere zur Neuberechnung der Pläne. Die Zerstörung erfolgt in der Methode **destroyPlan**. In ihr werden die Pläne mit Hilfe der `FFTW` zerstört und die Gültigkeit der Variablen **PlanValid** auf false gesetzt.

Die zweite Funktion besitzt die Verantwortung, die Pläne zu berechnen (**calcPlan**). Ihr werden als Parameter die Stützstellenanzahl in *x*- und *y*-Richtung vorgegeben. Bevor diese Funktion die neuen Pläne für die gewünschte Größe berechnet, werden erst die beiden alten Pläne mit der Methode **destroyPlan** zerstört. Erst anschließend werden die Pläne mit Hilfe der entsprechenden Methode der `FFTW` erzeugt und **PlanValid** auf true gesetzt.

An dieser Stelle soll bereits näher auf die Erzeugung der Pläne (`Plan.cpp:33-34`) eingegangen werden. Der Methode **fftw2d_create_plan** werden dabei mehrere Parameter übergeben. Zwei von diesen sollen kurz erläutert werden. **FFTW_IN_PLACE** gibt an, dass bei einer Vorwärtstransformation der Ortsbereich mit den Werten des Frequenzbereichs überschrieben werden soll. Nach der Transformation liegen an der Stelle des Feldes daher die Informationen des Frequenzbereichs. Diese Art der Transformation benötigt zwar mehr Zeit als eine out-place-Transformation, jedoch wird der Speicherplatzbedarf erheblich reduziert. Bei der erwähnten out-place-Transformation werden zwei Felder verwendet. Ein Feld enthält die Werte des Ortsbereichs. In das zweite Feld mit dergleichen Größe wird der Frequenzbereich geschrieben. Zu beachten ist, dass der Speicherplatzbedarf bei der in-place-Variante mehr als halb so groß wie bei out-place ist. Dies ist durch den benötigten temporären Speicherbedarf der `FFTW` bei der Fouriertransformation zu erklären.
Der zweite Parameter ist **FFTW_ESTIMATE**. Er zeigt der Methode **fftw2d_create_plan** an, dass nur ein suboptimaler Plan erstellt werden soll. Für seine Berechnung wird viel weniger Zeit in Anspruch genommen als für den optimalen Plan. Die Erstellung des optimalen Plans kann durch Angabe der Option **FFTW_MEASURE** erreicht werden. Diese Berechnung ist aber erst rentabel, wenn entweder eine hohe Anzahl von Transformationen oder eine Transformation mit einer hohen Anzahl von Stützstellen erfolgen soll. Dieser Parameter ist daher im Verlauf des Gesamtprojekts noch eventuell zu ändern.

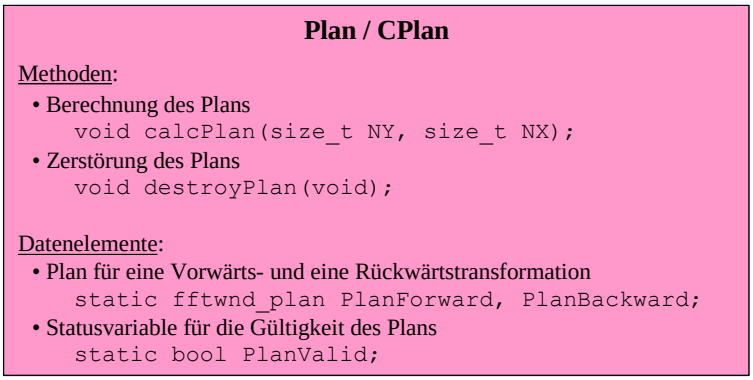


Abbildung 4.3 Entwurf der Klasse „Plan“

Die Klasse Plan ist, wie in Abbildung 4.1 erkennbar ist, Basisklasse für die Klasse „Feld“. Dieser Klassetyp besitzt zwei unterschiedliche Verantwortungsbereiche. Der erste ist die Verwaltung des eigentlichen Feldes, d.h. eines Arrays. Dieses Array ist Elementvariable dieser Klasse und ist vom Typ `CArrayManager`. Da die Daten innerhalb des Arrays vom Typ `fftw_complex` (siehe [FRI99]) sind, ist die Elementvariable vom Typ `CArrayManager<fftw_complex>`. Um von außen auf die Daten innerhalb des Feldes zugreifen zu können, ist ein entsprechender Zugriffsoperator definiert. Hierzu wurde der **()**-Operator überladen. Bevor jedoch mit diesem Operator auf das Array zugegriffen werden kann, muss dem Feld eine Größe zugewiesen werden. Dies ist die Aufgabe der Methode **setSize**. Dieser Methode werden 5 Parameter übergeben. Die ersten beiden (**NewNY**, **NewNX**) bestimmen die neue Stützstellenzahl des Arrays. Die drei anderen werden beim bereits

beschriebenen Auffüllen des Feldes mit Nullen benötigt. Der Parameter **save** gibt dabei an, ob die Daten des alten Arrays in das neue Array übertragen werden sollen. Ist der Wert false, wird ein neues Feld mit der Größe **NewNY** und **NewNX** reserviert. Ist Wert aber true, so werden an das alte Feld noch weitere Arrayelemente mit dem Inhalt „0“ angehängt. Da bei der Berechnung der Ursprung des Koordinatensystems jedoch in der physikalischen Mitte des Feldes angenommen wird, müssen die alten Daten noch innerhalb des neuen Feldes verschoben werden. Diese Verschiebung wird mit den beiden Parametern **OffsetY** und **OffsetX** vorgegeben.

Eine weitere Verhaltensweise der Klasse „Feld“ ist, dass die aktuelle Stützstellenzahl abgefragt werden kann. Dies erfolgt mit Hilfe der Methoden **getNX** und **getNY**.
Die letzten noch vorzustellenden Funktionen dienen dem Einlesen und der Ausgabe der Arraydaten. Für das Einlesen mit Hilfe der Funktion **read** muss dabei nur der Dateiname, in der die Daten gespeichert sind, als Parameter angegeben werden. Bei der Ausgabe mit der Methode **write** hingegen wird ein COut-Objekt als Parameter (**COut& Out**) benötigt. Beide Funktionen werden in ihrem Funktionsrumpf als `virtual` deklariert, weil sie zum Teil von entsprechenden Funktionen in den abgeleiteten Klassen überschrieben werden.
Auf die genaue Implementierung der Funktionen **getNX**, **getNY**, **read** und **write** muss nicht näher eingegangen werden. Sie nutzen nämlich nur die entsprechenden Funktionen des ArrayManagers und der Klasse `COut` (`Field.h:24, 25, 43; Field.cpp:63`).

Der zweite Verantwortungsbereich der Klasse „Feld“ besteht in der Durchführung der Fouriertransformation über das Array. Hierzu existieren die beiden Funktionen **Ffft** (Forward FFT) und **Bfft** (Backward FFT). Werden sie aufgerufen, überprüfen sie zuerst die Gültigkeit der Pläne, bevor sie die entsprechende Transformation über das Feld durchführen. Sind die Pläne ungültig, werden zuerst mit der Methode **CPlan::calcPlan** neue Pläne errechnet (`Field.cpp:30-57`).
Zu beachten ist, dass bei der FFTW die Daten des Arrays nach einer Hin- und Rücktransformation vertauscht werden. Um diese Vertauschung rückgängig zu machen, wird die Funktion **ChangeField** verwendet. Sie tauscht die vier Quadranten des Feldes paarweise miteinander. Dieses Verhalten muss vom Anwender gesteuert werden und wird aus Zeitgründen nicht bei jeder Transformation automatisch durchgeführt.

Diese Eigenschaften und Methoden der Klasse „Feld“ sind in Abbildung 4.4 dargestellt.



Abbildung 4.4 Entwurf der Klasse „Feld“

Von Klasse „Feld“ werden drei weitere Klassen abgeleitet. Eine von diesen ist die Klasse „Elektrisches Feld“ (siehe Abbildung 4.5). Sie erbt unter anderem von der Klasse „Feld“ das Array. Dieses Array besitzt im Falle des elektrischen Feldes jedoch noch weitere Eigenschaften. Hierzu gehören die Wellenlänge (**Lambda**), eine physikalische Höhe und Breite des Feldes (**HeightYE**, **WidthXE**), sowie ein Abstand zwischen den Stützstellen in *x*- und *y*-Richtung (**DeltaXE**, **DeltaYE**).
Passend dazu werden get-Funktionen definiert, die die Werte der entsprechenden Elementvariablen zurückgeben (Bsp: **getLambda**, **getDeltaXE**, ...). Diese Funktionen werden aber um eine zusätzliche Methode ergänzt. Sie besitzt die Wellenzahl k als Rückgabewert, d.h. in der Funktion wird direkt das Produkt $^{2\pi}/_{\lambda}$ berechnet.

Weiterhin existieren zwei Funktionen, die unmittelbar auf die Elementvariablen zugreifen. Einerseits ist dies die Methode **setLambda**, bei der die Wellenlänge des elektrischen Feldes geändert werden kann und anderseits die Methode **equalParameters**. Der Methode **equalParameters** wird als Parameter eine Referenz eines elektrisches Feld übergeben. Ihre Aufgabe ist es die Eigenschaften des Referenzfeldes mit den Eigenschaften der Instanz zu vergleichen. Zu diesen Eigenschaften gehören die geometrische Größe der Felder, die Anzahl der Stützstellen und die Wellenlänge (`ElectricalField.cpp:28-38`). Der Rückgabewert dieser Funktion ist true, wenn die Eigenschaften identisch sind. Andernfalls ist der Rückgabewert false.

Eine weitere Verhaltensweise der Klasse „Elektrisches Feld“ ist, dass sich ein elektrisches Feld selbständig auf eine gewünschte geometrische Breite und Höhe mit Nullen auffüllen kann. Die Methode besitzt den Namen **fillUp**. Ihr werden die gewünschten Größen in x- (Breite) und y-Richtung (Höhe) als Parameter übergeben. Aus Gründen der Rechengeschwindigkeit der FFT wird die Größe des elektrischen

Feldes dabei verdoppelt, vervierfacht oder verachtfacht, usw. (2er-Potenz). Diese Methode ist nach einer Vergrößerung des Feldes auch für die Veränderung der Elementvariablen zuständig und führt dies automatisch durch. Außerdem wird bei einer Veränderung der Größe des Feldes gleichzeitig die Gültigkeit der Pläne auf false gesetzt. Denn sobald sich die Größe des Feldes ändert, muss ein neuer Plan berechnet werden.

Außerdem existieren in der Klasse „Elektrisches Feld“ zwei Funktionen **read** und **write**, die die entsprechenden Funktionen aus der Basisklasse „Feld“ überschreiben. Die Aufgabe der Funktionen wird jedoch nur geringfügig erweitert. Zusätzlich zu den Felddaten müssen in der abgeleiteten Klasse noch die Elementvariablen mit eingelesen bzw. ausgegeben werden. Dieses Einlesen und Ausgeben erfolgt in den Funktionen als erstes, damit anschließend die entsprechende Funktion (**read**, **write**) in der Basisklasse „Feld“ aufgerufen werden kann (`ElectricalField.cpp:213-226, 230-256`). Die Deklaration der Methode **write** bleibt dabei in der Klasse „Elektrisches Feld“ unverändert. Nur bei der Methode **read** werden zwei weitere Parameter deklariert. Dies ist einerseits eine Variable des Datentyps `FILE`, damit die Funktion selbständig die Elementvariablen direkt aus der Datei einlesen kann und andererseits der Parameter **mode**. Durch ihn wird der Methode mitgeteilt, ob die Daten in der Datei als ASCII- oder Binärdaten vorliegen.

Die letzte zu erläuternde Funktion (**setTestEField**) wird nur in der Testphase des Programms benötigt. Ihre Aufgabe ist es ein elektrisches Feld mit einer bestimmten Größe (NX, NY, Breite, Höhe) und einer bestimmten Wellenlänge (Lambda) zu erzeugen. Diese Funktion ist so implementiert, dass das elektrische Feld eine ebene Welle simuliert.

Elektrisches Feld / CElectricalField

Methoden:

- Abfrage der Datenelemente und der Wellenzahl k

```
double getLambda (void);
double getDeltaXE (void);
```

- Ändern der Wellenlänge

```
void setLambda (double _Lambda);
```

- Ausgabe / Einlesen eines E-Feldes

```
virtual void read (string&, dateiname, FILE *fd, char mode);
virtual void write (COut& Out);
```

- Überprüfung, ob 2 E-Felder gleiche Eigenschaften haben

```
bool equalParameters (CElectricalField& E);
```

- Auffüllen mit Nullen bis zu einer best. Breite und Höhe

```
void fillUp (double WishWidth, double WishHight);
```

- Setzen eines Testfeldes

```
void setTestEField (size_t _NX, size_t _NY, double _WidthXE,
                    double _HeightYE, double _Lambda);
```

Datenelemente:

- Wellenlänge (`double Lambda;`)
- Breite, Höhe (`double WidthXE, HeightYE;`)
- Abstand der diskreten Werte in x- und y-Richtung (`double DeltaXE, DeltaYE`)

Abbildung 4.5 Entwurf der Klasse „Elektrisches Feld“

Außer der Klasse „Elektrisches Feld“ wurden noch zwei weitere Klassen von der Klasse „Feld“ abgeleitet. Dies sind die Klassen „Transmissionsfunktion“ und „Übertragungsfunktion“. Sie werden im weiteren gemeinsam besprochen. Beide Klassen besitzen ähnlich wie das elektrische Feld die Eigenschaft, dass sie nur für eine bestimmte Wellenlänge (**LambdaH**, **LambdaT**), Breite (**WidthXH**, **WidthXT**) und Höhe (**HeightYH**, **HeightYT**) gültig sind, für die sie berechnet wurden. Diese Variablen bilden die Datenelemente der jeweiligen Klasse.
Um die volle Funktionalität der Klassen zu nutzen, ist es ausreichend, verschiedene Methoden zu definieren, mit denen man die Werte aller Elementvariablen einzeln abfragen (get-Funktionen) und manipulieren (set-Funktionen) kann.
Zu einem späteren Zeitpunkt des Gesamtprojekts kann es an dieser Stelle hilfreich sein, Methoden zum Einlesen und zur Ausgabe dieser Funktionen zu implementieren.
Die vorhandenen Funktionen und Datenelemente sind in Abbildung 4.6 dargestellt.

Übertragungsfunktion / CTransferFunction
Transmissionsfunktion / CTransmissionFunction

Methoden:

- Abfrage / Setzen der Datenelemente

```
double getLambdaH (void);
double getLambdaT (void);
void setLambdaH (double _Lambda);
void setLambdaT (double _Lambda);
```

Datenelemente:

- Wellenlänge (`double LambdaH, LambdaT;`)
- Breite (`double WidthXH, WidthXT;`)
- Höhe (`double HeightYH, HeightYT;`)

Abbildung 4.6 Entwurf der Klassen „Transmissions-“ und „Übertragungsfunktion“

Nachdem nun der Klassenentwurf der ersten Klassenhierarchie vollständig beschrieben wurde, kann die Vorstellung der zweiten Klassenhierarchie erfolgen. Die Basisklasse dieses Klassenbaums ist die Klasse „Kameraelement“. Eigenschaft dieser Klasse ist, dass jedes Kameraelement innerhalb der Kamera eine Nummer (**No**) erhält. Diese Nummer beschreibt die Position des jeweiligen Kameraelements innerhalb der gesamten Kameraanordnung. Die Nummerierung beginnt bei dem ersten Element mit dem Wert „0“ und endet beim letzten Element mit N-1, wenn N die Gesamtzahl aller Kameraelemente darstellt. Damit nach jeder Erzeugung eines Kameraelements diesem Objekt seine Positionsnummer zugewiesen werden kann, ist eine Funktion **setNo** definiert. Ihr wird nur die Positionsnummer als Parameter übergeben.
Eine zweite Elementvariable der Klasse ist **CalcNumber**. Sie gibt die Anzahl der bereits von diesem Kameraelement durchgeführten Berechnungen an. Sie wird nach jedem Berechnungsdurchgang um den Wert eins erhöht. Die Funktionalität dieser beiden Eigenschaften ist erforderlich, um dynamisch (während der Berechnung) eindeutige Dateinamen zu erzeugen. Diese Dateinamen werden dann zur Ausgabe des Berechnungsergebnisses (elektrisches Feld, Intensitätsverteilung) verwendet.

Damit ist sichergestellt, dass nach Abschluss einer vollständigen Berechnung die erzeugten Ausgabedateien den einzelnen Elementen zugeordnet werden können.

Weiterhin besitzt die Klasse „Kameraelement“ mehrere get-Funktionen. Zwei dieser Methoden werden dazu verwendet, die Größe (Höhe, Breite) des Kameraelements zu bestimmen (**getHeightY**, **getWidthX**). Diese Funktionen werden benötigt, damit die Klasse „Kamera“ die größte räumliche Ausdehnung von allen Objekten ermitteln kann. Damit ist bereits an dieser Stelle vorweggenommen, dass die abgeleiteten Klassen (Rechteckapertur, Kreisapertur, Konvexlinse, Zylinderlinse, Schirm) die Elementvariablen Höhe, Breite bzw. Radius besitzen. Diese Funktionalität ist – wie bereits oben beschrieben wurde – notwendig, damit das elektrische Eingangsfeld auf die maximale geometrische Größe der Kamera ausgedehnt werden kann. Diese Funktionen werden daher von den abgeleiteten Klassen überschrieben. Aus diesem Grund sind diese Methoden als rein virtuelle Methoden definiert. Es soll aber bereits an dieser Stelle erwähnt werden, dass je nach spezifiziertem Typ des Kameraelements (Rechteckapertur, Kreisapertur, Konvexlinse, Zylinderlinse, Schirm) die Breite, Höhe oder der Kreisdurchmesser des Objektes zurückgegeben wird. Daher wird bei der weiteren Beschreibung der restlichen Klassen nicht mehr auf diese Funktion eingegangen. Sie werden jedoch weiterhin im Klassenentwurf der einzelnen Klassen dargestellt.
Zwei weitere get-Funktionen werden verwendet, um die Anzahl von vorgegebenen Stützstellen des letzten Elements (Schirm) zu ermitteln (**getNX**, **getNY**). Dies ist aus folgendem Grund notwendig: Der CCD-Chip besitzt eine Breite und eine Höhe. Zusätzlich enthält er in Analogie zum realen CCD-Chip eine bestimmte Anzahl von Pixel in *x*- und *y*-Richtung (**NX**, **NY**). Auch diese Eigenschaften sind in der Klasse „Schirm“ als Elementvariable vorhanden. Mit Hilfe dieser vier Werte kann die Auflösung des Schirms berechnet werden. Diese Funktionalität wird benötigt, damit die Klasse „Kamera“ in ihrer Berechnungsmethode zuerst die Auflösung des CCD-Schirms ermitteln und anschließend ein Eingangsfeld mit derselben Auflösung anfordern kann. Nur so kann gewährleistet werden, dass das berechnete elektrische Feld dieselbe Auflösung wie der CCD-Chip hat.
Beim Zugriff auf die Funktionen **getNX** und **getNY** des Schirms ist aber zu beachten, dass die Klasse „Kamera“ nur mit Hilfe des Elementverweisoperators -> auf die einzelnen Instanzen der Klasse „Kameraelement“ zugreifen kann (Bsp.: CamEle[NCamEle-1]->getNX();). Würden die Funktionen **getNX** und **getNY** aufgrund eines Fehlers in der Eingabedatei `Camera.dat` auf ein optisches Element angewendet werden (Bsp.: Als letztes Kameraelement wird vom Anwender ein optisches Element definiert.), so würde das gesamte Programm sofort beendet werden. Um dies zu verhindern, werden die beiden Funktionen auch in der Klasse „Kameraelement“ implementiert (`CameraElement.cpp:21-25, 31-35`). Aufgabe dieser Funktionen ist es jedoch nur eine Fehlermeldung mit dem Inhalt „Letztes Kameraelement ist kein Schirm.“ auf dem Bildschirm auszugeben und dem Anwender damit den Fehler anzuzeigen. Das Programm wird aber nicht beendet, sondern normal fortgesetzt. Bei richtiger Verwendung der Eingabedatei werden diese Methoden aber von den entsprechenden Funktionen **getNX** und **getNY** der Klasse „Schirm“ überschrieben und nie aufgerufen.

Die wichtigste Methode der Klasse „Kameraelement“ ist jedoch die zur Berechnung der elektrischen Felder. Sie besitzt den Namen **calc** und benötigt zwei Referenzen auf die beiden elektrischen Felder als Parameter. Der Rückgabetyp dieser Funktion ist `void`, weil – wie bereits beschrieben wurde – die Felder direkt verändert werden

(in-place). Sie existiert in dieser Klasse nur als abstrakte Methode, weil die spezifische Implementierung erst in den abgeleiteten Klassen „Schirm“ und „Optisches Element“ erfolgt.
Der Abbildung 4.7 kann der vollständige Klassenentwurf entnommen werden.

Kameraelement / CCameraElement

Methoden:

- Berechnung ($\underline{E}_x$, $\underline{E}_y$)

```
virtual void calc (CElectricalField& EX,
                   CElectricalField& EY)        =0;
```

- Ändern der Positionsnummer (`void setNo (int _No);`)
- Abfrage der räumlichen Größe

```
virtual double getWidthX(void)                  =0;
virtual double getHeightY(void)                 =0;
```

- Abfrage der Stützstellenzahl NX, NY

```
virtual size_t getNX(void);
virtual size_t getNY(void);
```

- Berechnungsfunktion ($\underline{E}_x$, $\underline{E}_y$)

```
virtual void calc (CElectrical Field& EX,
                   CElectricalFiel& EY)         =0;
```

- Einlesen der Objektdaten (Elementvariablen)

```
virtual void read (CInput& Input)               =0;
```

- Ausgabe der Objektdaten (Elementvariablen)

```
virtual void writeElement (COut& KameraElement) =0;
```

Datenelemente:

- Positionsnummer (`int No;`)
- Anzahl der berechneten Bilder (1 Bild = 2 E-Felder) bzw. Berechnungsdurchgänge

```
int CalcNumber;
```

Abbildung 4.7 Entwurf der Klasse „Kameraelement“

Zwei Methoden dieser Abbildung sind noch nicht näher erläutert worden. Diese Funktionen sind für das Einlesen und die Ausgabe der Objektdaten verantwortlich. Da diese Funktionen in jeder von „Kameraelement“ abgeleiteten Klasse vorhanden sind (Schirm, Optisches Element, Apertur, Zylinderlinse, ...), sollen sie an dieser Stelle vollständig beschrieben werden. Wie bereits bei der Funktion **read** in der Klasse „Kamera“ erklärt wurde, wird in der dortigen Funktion zuerst ein spezialisiertes Kameraelement (Rechtexkapertur, Zylinderlinse, Schirm, ...) mit den jeweiligen Standardkonstruktoren erstellt. Anschließend wird dieses Element direkt angewiesen, seine eigenen Elementvariablen einzulesen. Als Beispiel soll hier eine Konvexlinse behandelt werden. Dazu wird die Funktion **read** in der Klasse „Konvexlinse“ aufgerufen. Ihr wird als Parameter eine Referenz auf ein CInput-Objekt übergeben. Dieses Objekt besitzt Informationen darüber, aus welcher Datei die Daten eingelesen werden sollen. Auf das CInput-Objekt soll an dieser Stelle nicht näher eingegangen werden. Es wird bei der Beschreibung des Gültigkeitskonzepts näher erläutert. Die Funktion **read** der Konvexlinse hat nun die Aufgabe, nicht nur ihre eigenen Elementvariablen aus der Datendatei einzulesen, sondern auch die **read**-Funktionen der übergeordneten „Linse“ aufzurufen. Diese wiederum besitzt dieselbe

Verantwortung wie die **read**-Funktion der Klasse „Konvexlinse“. Sie liest ihre eigenen Datenelemente ein und ruft die **read**-Funktion des optischen Elements auf. Nach Abarbeitung dieser drei **read**-Funktionen (Konvexlinse, Linse, Optisches Element) sind alle Elementvariablen der Konvexlinse eingelesen. Zu beachten ist, dass die **read**-Funktion in der Klasse „Kameraelement“ nicht aufgerufen wird, da diese Klasse keine Elementvariablen zum einlesen besitzt (Hinweis: Das Setzen der Elementnummer des Objektes erfolgt in der Klasse „Kamera“, nach erfolgreichem Einlesen der Datenelemente, mit Hilfe der Funktion **setNo**.). Sie ist somit eine Funktion ohne Anweisungsteil und ist daher als abstrakte Methode definiert. Die Definition ist nur notwendig, damit sichergestellt ist, dass alle abgeleiteten Klassen diese Funktion überschreiben. Erfüllt eine abgeleitete Klasse diese Aufgabe nicht, wird sie selber zur abstrakten Klasse und es ist nicht möglich Instanzen von ihr zu erzeugen.
Ähnlich wie die Funktion **read** ist die Methode **writeElement** implementiert. Wird vom Anwender die Ausgabe aller Elementvariablen und somit der berechneten Kamerakonfiguration gefordert, ruft die Klasse „Kamera“ die Funktion **writeElement** jedes Kameraelements auf. Dabei wird eine Referenz eines COut-Objekt als Parameter übergeben. Dieses Objekt besitzt die notwendigen Informationen über die Datei, in die die Datenelemente geschrieben werden sollen. Außerdem stellt die Klasse `COut` zusätzliche Funktionen zur Verfügung, mit denen die Ausgabe erheblich vereinfacht wird (`OpticalElement.cpp:647-654`). Als Beispiel wird wieder eine Konvexlinse betrachtet werden. Wird ihre **writeElement**-Funktion aufgerufen, dann gibt sie nicht nur ihre eigenen Elementvariablen aus, sondern ruft auch die **writeElement**-Funktion ihrer Basisklasse (Linse) auf. Diese führt daraufhin die gleichen Befehle aus. Auf diese Art wird wie bei der Funktion **read** sichergestellt, dass alle Elementvariablen in das COut-Objekt geschrieben werden. Es ist aber darauf hinzuweisen, dass nicht alle Datenelemente ausgegeben werden, sondern nur die, die auch in der Eingabedatei vorhanden sind. Diese Variablen sind ausreichend, um die berechnete Kamerakonstellation nachzuvollziehen.
Da jetzt die gesamte Struktur zum Einlesen der Daten und zur Ausgabe der Elementvariablen beschrieben wurde, wird bei den weiteren Klassenentwürfen auf eine Erläuterungen dieser Funktionen verzichtet. Die Funktionen werden auch nicht mehr in den Abbildungen dargestellt.

Mit den Funktionen **read** und **writeElement** sind alle Funktionen der Klasse „Kameraelement“ erklärt worden und es kann die Klasse „Schirm“ betrachtet werden (siehe Abbildung 4.8). Zuerst werden die Eigenschaften dieser Klasse genauer erläutert. Zu den Elementvariablen gehören neben den bereits besprochenen Variablen **HeightYS**, **WidthXS**, **NX** und **NY** auch eine Intensitätsverteilung **I** vom Typ `CArrayManager<float>`. Dazugehörend existiert auch ein COut-Objekt **OutI**. Mit diesem kann die Intensitätsverteilung einfach in eine Datei geschrieben werden.
Weiterhin sind Ausgabevariablen definiert. Sie sind in der Struktur `output` gekapselt (`Screen.h:82-88`). Die Variablen beschreiben die zu verwendende Ausgabevariante. So wird über die Variable **WriteI** spezifiziert, ob die Intensitätsverteilung in einer Datendatei ausgegeben werden soll und wenn ja, ob die Ausgabe im ASCII- oder Binärmodus zu erfolgen hat. Die Ausgabevariablen können daher in der Eingabedatei `Camera.dat` vom Benutzer beliebig eingestellt werden. Dort wird ebenfalls festgelegt, ob zusätzlich die Intensitätsverteilung in ein TIFF-

Bild geschrieben werden soll (**WriteTIFF**). Wenn dies gewünscht wird, hat der Anwender noch die Möglichkeit die Intensitätsverteilung zu logarithmieren (**LogTIFF**). Ziel dieser Funktion ist, es niedrige Werte im Vergleich zum Intensitätsmaximum anzuheben. Dadurch werden feinere Strukturen des Bildes sichtbar.

Für die Beschreibung der vier nächsten Elementvariablen ist es notwendig, den allgemeinen Berechnungsablauf zu betrachten. Dort wird in der Funktion **calc** der Klasse „Kamera“ die Auflösung des CCD-Chips verwendet, um ein passendes Eingangsfeld anzufordern. Anschließend wird dieses Feld mit Nullen auf die maximale Größe der Kamera angepasst. Infolgedessen besitzt das „wahre“ berechnete Feld zwar den gleichen Stützstellenabstand wie der Schirm, die geometrische Größe und die Anzahl der Stützstellen wurde jedoch erhöht. Da die Original-Elementvariablen jedoch nicht verändert werden sollen (Begründung folgt in dem Abschnitt „Gültigkeitskonzept“), werden die wahren Werte in den Variablen **CalculatedWidthXS**, **CalculatedHeightYS**, **CalculatedNX** und **CalculatedNY** gespeichert.

Eine weitere Elementvariable ist **DestroyI**. Ihr Wert wird wie die Ausgabevariablen aus der Eingabedatei eingelesen. Sie gibt an, ob nach der Berechnung und Ausgabe der Intensitätsverteilung das Feld **I** gelöscht werden soll. Dies gibt dem Anwender die Möglichkeit den vorhandenen Speicherplatz optimal zu nutzen und unnötige Speicherfreigaben und –reservierungen zu verhindern. Diese Statusvariable liegt in Analogie zu den später erläuternden Statusvariablen der Klasse „Optisches Element“ in einer Struktur `status` vor (`Screen.h:77-79`).

Von den vorhandenen Methoden sollen zunächst die Funktionen beschrieben werden, die `protected` sind. Hierzu gehört unter anderem die Funktion **writeTIFF**. Sie gibt die Intensitätsverteilung im TIFF-Format aus. Als Parameter wird der zu verwendende Dateiname übergeben. Die Methode überprüft bei der Ausgabe selbständig, ob die Intensitätsverteilung logarithmiert werden soll oder nicht. Wenn dies zu erfolgen hat, wird die Funktion **logintensityTIFF** aufgerufen. Sie ist eine weitere Funktion, die den Status `protected` besitzt. Sie führt eine Logarithmierung über die Intensitätsverteilung durch. Ist dieser Vorgang abgeschlossen, ruft die Methode **writeTIFF** die Funktion **scaleIntensity** auf. Ihre Verantwortung ist es, die Intensitätsverteilung für das TIFF-Format zu skalieren. Dafür muss der Maximalwert 255 betragen. Außerdem müssen alle Intensitätswerte auf ganze Zahlen gerundet werden.

Betrachtet man die `public`-Methoden genauer, so ist erkennbar, dass die Funktionen **getWidthX**, **getHeightY**, **getNX** und **getNY** bereits vollständig beschrieben wurden. Nur die Methode **calc** wurde noch nicht klassenspezifisch erörtert. Sie besitzt als Parameter die beiden elektrischen Felder $\underline{E}_x$ und $\underline{E}_y$, die ihr als Referenz übergeben werden. Daraus wird die entsprechende Intensitätsverteilung **I** berechnet und anschließend in eine Datendatei oder Bilddatei ausgegeben. Dabei wird der Dateiname mit Hilfe der Positions- und Berechnungsnummer (**No**, **CalcNumber**) dynamisch erstellt.

Schirm / CScreen

Methoden:

- Berechnung (E_x, E_y)

```
virtual void calc (CElectricalField& EX,
                   CElectricalField& EY);
```

- Ausgabe der Intensität im TIFF-Format

```
void writeTIFF (char Filename[100]);
```

- lineare Skalierung der Intensität

```
void scaleLinearIntensity (void);
```

- Logarithmierung der Intensitätsverteilung

```
void logIntensity (void);
```

- get-Funktionen

```
virtual double getWidthXS (void);
virtual double getHeightYS (void);
virtual double getNX (void);
virtual double getNY (void);
```

Datenelemente:

- Intensitätsverteilung

```
CArrayManager<float> I;
```

- Ausgabeobjekt für die Intensitätsverteilung

```
COut OutI;
```

- Ausgabe der Intensitätsverteilung (ASCII-, Binär- oder TIF-Format)

```
char WriteI;
bool WriteTIFF, LogTIFF;
```

- Angabe, ob das Feld I nach der Berechnung zerstört wird

```
bool DestroyI;
```

- geometrische Größe

```
double WidthXS, HeightYS;
```

- Anzahl der Pixel (Stützstellenanzahl)

```
int NX, NY;
```

- berechnete geometrische Größe

```
double CalculatedWidthXS, CalculatedHeightYS;
```

- berechnete Anzahl der Pixel (Stützstellenanzahl)

```
int CalculatedNX, CalculatedNY;
```

Abbildung 4.8 Entwurf der Klasse „Schirm“

Als nächstes wird die in Abbildung 4.9 dargestellte Klasse „Optisches Element“ beschrieben. Auch bei dieser Klasse werden zuerst die Elementvariablen erläutert. Dazu gehört unter anderem die Ausgabevariable **WriteE**. Sie kann wie bei der Klasse „Schirm“ vom Anwender vorgegeben werden. Der Anwender hat dabei ebenfalls die Auswahlmöglichkeiten zwischen keiner durchzuführenden Ausgabe und einer Ausgabe in eine ASCII- oder Binärdatei. Die Ausgabevariablen sind in der Struktur `output` zusammengefasst (`OpticalElement.cpp:97-99`). Soll eine Ausgabe erfolgen, dann geschieht dies in der Berechnungsfunktion **calc** mit Hilfe der COut-Objekte **OutEFieldX** und **OutEFieldY**.

Weitere Datenelemente sind die, die durch Betrachtung der realen Objekte beobachtet werden können. Hierzu gehören der Abstand (**Distance)** zum nächsten optischen Element, die Verschiebung des darauffolgenden Kameraelements in x- und y-Richtung (**XShift**, **YShift**). Des weiteren ist die Übertragungsfunktion **H** und die

Berechnungsmethode **Modell** Bestandteil jedes optischen Elements. Zu beachten ist, dass die Funktion **H** vom bereits beschriebenen Klassentyp `CTransferFunction` ist.

Zusätzlich zu diesen Datenelementen existieren die Statusvariablen **DestroyH**, **DestroyT**, **HValid** und **TValid**. Sie sind in der Struktur `status` (`OpticalElement.h:89-94`) zusammengefasst. Die ersten beiden Variablen dienen – analog zum „Schirm“ – der Speicherverwaltung. Sie werden ebenfalls in der Eingabedatei vom Anwender vorgegeben. Demgegenüber werden die beiden letzten Variablen ausschließlich programmintern verwendet. Wie bereits erklärt wurde, sind die Übertragungs- und Transmissionsfunktion unveränderlich und brauchen nur einmal berechnet zu werden. Die beiden Variablen geben daher die Gültigkeit der entsprechenden Felder an (gültig = true).
Zu beachten ist, dass die Variablen für die Transmissionsfunktion (**DestroyT**, **TValid**) bei strenger objektorientierter Programmierung nicht in dieser Klasse implementiert werden dürften. Da die Struktur `status` in der abgeleiteten Klasse „Linse“ nicht überschrieben werden soll, wurde sie in der Klasse „Optisches Element“ definiert.
Bei den Methoden der Klasse ist auffällig, dass sowohl das Abschneiden eines Rechtecks (**cutRectangle**), als auch das Abschneiden eines Kreises (**cutCircle**) im optischen Element implementiert ist. Der Grund dafür ist, dass der doppelte Programmcode vermieden wird. Andernfalls müsste zum Beispiel die Funktion **cutRectangle** in den Klassen „Rechteckapertur“ und „Zylinderlinse“ identisch definiert werden. Die Aufgabe der beiden Funktionen ist das Ausblenden des elektrischen Feldes an der Apertur.
Weiterhin sind die Berechnungsfunktionen für die Elementvariable **H** definiert. Die Hauptfunktion ist dabei **calcTransferFunction**. Sie entscheidet anhand der Variable **Modell**, nach welcher Theorie die Übertragungsfunktion berechnet wird. Je nach Wahl des Modells wird dann die entsprechende Berechnungsfunktion (**calcApproximation**, **calcSecondRS**, **calcFirstRS**, **calcKirchhoff**) aufgerufen. Anschließend wird, falls benötigt, die Verschiebung des nachfolgenden Kameraelements berücksichtigt. Für diesen Fall ist die Methode **calcShift** implementiert worden. Zu beachten ist, dass nach der vollständigen Berechnung der Übertragungsfunktion die Statusvariable **HValid** auf true gesetzt wird. Wichtig ist ebenfalls, dass allen Methoden eine Referenz auf ein elektrisches Feld übergeben wird. Die Verwendung dieses Parameters ist notwendig, damit die diskreten Werte der Übertragungsfunktion an den Stellen berechnet werden, an denen auch Werte des elektrischen Feldes vorliegen. Nur so kann gesichert werden, dass später die diskreten Werte der Übertragungsfunktion mit den passenden Werten des elektrischen Feldes multipliziert werden. Für die Berücksichtigung der räumlichen Lage der Werte sind die geometrischen Daten (Höhe, Breite) und die Stützstellenzahl des elektrischen Feldes (NX, NY) erforderlich. Außerdem wird in allen Methoden die Wellenlänge des E-Feldes benötigt, um eine genau abgestimmte Übertragungsfunktion zu berechnen.

Neben diesen Funktionen ist in dieser Klasse auch die Methode **calc(CElectricalField& EX, CElectricalField& EY)** definiert. Der Funktion werden zwei elektrische Felder als Parameter in Form einer Referenz übergeben. Ihre Aufgabe ist es, mit beiden elektrischen Feldern den gleichen Berechnungsablauf durchzuführen (zweimaliger Aufruf derselben Funktion). Dieser Berechnungsablauf unterscheidet sich je nach Klassentyp (Apertur oder Linse) nur in der zusätzlichen

Berücksichtigung der Transmissionsfunktion (siehe Gleichung (4.1)). Dieser gleichartige Berechnungsablauf besitzt ebenfalls den Namen **calc**. Die Methode besitzt aber nur ein elektrisches Feld als Parameter (**calc(CElectricalField& E)**). Aufgrund der notwendigen unterschiedlichen Implementierung der Methode **calc(CElectricalField& E)** in den Klassen „Apertur“ und „Linse“ ist diese in der Klasse „Optisches Element“ nur als abstrakte Funktion definiert.
Der hier beschriebene Berechnungsablauf ist grundsätzlich möglich und richtig. Da in der Funktion **calc(CElectricalField& EX, CElectricalField& EY)** aber nach der Berechnung eventuell die Übertragungs- und Transmissionsfunktion zerstört werden (**DestroyH**, **DestroyT**), ist auch diese Methode in den Klassen „Apertur“ und „Linse“ implementiert. Der Grund hierfür ist, dass das optische Element keine Transmissionsfunktion besitzt. Die Funktion ist daher ebenfalls als eine rein virtuelle Methode deklariert.

Eine weitere Funktion, die Teil des optischen Elements ist, wird ebenfalls zu direkten Berechnung benötigt. Dies ist die Methode **multiplicationAndNormalisation**. Sie benötigt als Parameter eine Referenz auf ein elektrisches Feld und eine Referenz auf die Übertragungsfunktion **H**. Ihre Aufgabe besteht darin das elektrische Feld elementweise mit der Übertragungsfunktion zu multiplizieren. Gleichzeitig wird die Normierung der FFT durchgeführt. Wird nämlich über ein Feld mit der Größe N eine Vorwärts- und anschließend eine Rückwärtstransformation durchgeführt, dann liegen im Feld die Daten des Ortsbereichs multipliziert mit N vor. Diese Multiplikation wird bei der Normierung rückgängig gemacht. Das Ergebnis der Multiplikation wird dabei in das elektrische Feld geschrieben.

Optisches Element / COpticalElement

Methoden:

- Berechnung ($\underline{E_x}$, $\underline{E_y}$)

```
virtual void calc (CElectricalField& EX,
                   CElectricalField& EY)     =0;
```

- Berechnung ($\underline{E}$)

```
virtual void calc (CElectricalField& E)      =0;
```

- Berechnung der Übertragungsfunktion

```
void calcTransferFunction (CElectricalField);
void calcApproximation (CElectricalField);
void calcSecondRS (CElectricalField);
void calcFirstRS (CElectricalField);
void calcKirchhoff (CElectricalField);
void calcShift (CElectricalField);
```

- Multiplikation von E mit H

```
void multiplicationAndNormalization (CElectricalFiel& E,
                                     CTransferFunctionH);
```

- Ausblenden des elektrischen Feldes an der Blende

```
void cutRectangle (CElectricalField& E, double _HeightYRec,
                   double _WidthXRec);
void cutCircle (CElectricalField&E, double _Radius);
```

- Abfrage der räumlichen Größe

```
virtual double getWidthX(void)               =0;
virtual double getHeightY(void)              =0;
```

Datenelemente:

- Übertragungsfunktion $\underline{H}$

```
CTransferfunction H;
```

- Berechnungsmodell

```
char Modell;
```

- Abstand und Verschiebung des nächsten Elements (x_0, y_0, z_0)

```
double XShift, YShift, Distance;
```

- Ausgabeobjekte für das E-Feld (eins für $\underline{E_x}$, eins für $\underline{E_y}$)

```
COut OutEFieldX, OutEFieldY;
```

- Angabe, ob die Felder H und T nach der Berechnung zerstört werden sollen

```
bool DestroyH, DestroyT;
```

- Angabe, ob die Elemente in der Übertragungs- und Tranmissionsfunktion gültig sind

```
bool HValid, TValid;
```

Abbildung 4.9 Entwurf der Klasse „Optisches Element“

Von der Klasse „Optisches Element“ wird die Klasse „Apertur“ abgeleitet. Sie besitzt keine eigenen Elementvariablen. Die Funktion **calc(CElectricalField& E)** führt mit einem elektrischen Feld eine vollständige Berechnung durch. Diese Berechnung besteht im einzelnen aus folgenden Schritten:

1. Ausblenden des elektrischen Feldes an der Apertur (**cutE(E)**)
2. Berechnung der Übertragungsfunktion (**calcTransferFunction(E)**)
3. Transformation des E-Feldes in den Frequenzbereich (**E.Ffft**)
4. Multiplikation des E-Feldes mit der Übertragungsfunktion (**multiplicationAndNormalization(E, H)**)
5. Rücktransformation des E-Feldes in den Ortsbereich (**E.Bfft**)
6. Austausch der vier Quadranten des E-Feldes (**E.ChangeField**)

Dabei wurden fünf der sechs vorkommenden Funktionen bereits erläutert. Nur die Methode **cutE** wurde noch nicht beschrieben. Sie ist aber nur als abstrakte Funktion definiert, da das Ausblenden des elektrischen Feldes in Abhängigkeit von der Blendenform (Rechteck-, Kreisapertur) erfolgt.

Die dritte Methode der Klasse „Apertur“ ist **calc(CElectricalField& EX, CElectricalField& EY)**. Die Aufgabe dieser Funktion lässt sich gliedern in:

1. Berechnung des Feldes EX (**calc(EX)**)
2. Berechnung des Feldes EY (**calc(EY)**)
3. Erhöhung der Zählvariablen (**CalcNumber++**)
4. ggf. Zerstörung der Übertragungsfunktion (**DestroyH**)
5. ggf. Erzeugung eines eindeutigen Dateinamens für die Ausgabe
6. ggf. Ausgabe der elektrischen Felder (**EX.write(OutEFieldX)**)

Apertur / CAperture

Methoden:

• Berechnung von zwei elektrischen Feldern (= 1Bild)

```
virtual void calc (CElectricalField& EX,
                   CElectricalField& EY);
```

• Berechnung eines E-Feldes *E*

```
virtual void calc (CElectricalField& E);
```

• Ausblenden des E-Feldes an der Blende

```
virtual void cutE (CElectricalField& E)   =0;
```

• Abfrage der räumlichen Größe

```
virtual double getWidthX(void)            =0;
virtual double getHeightY(void)           =0;
```

Abbildung 4.10 Entwurf der Klasse „Apertur“

Als nächstes wird die Klasse „Rechteckapertur“ beschrieben. Diese Apertur hat in Analogie zum realen Objekt die Eigenschaft, dass es in x- und y-Richtung eine räumliche Größe besitzt. Die Elementvariablen besitzen die Namen **WidthXRec** und **HeightYRec** und werden in der Datei `Camera.dat` vom Anwender vorgegeben. Sie beschreiben die Fläche der Rechteckapertur durch die das elektrische Feld die Blende ungehindert passieren kann.
Als Methode ist nur die Funktion **cutE** vorhanden. Ihr wird das abzuschneidende elektrische Feld in Form einer Referenz als Parameter übergeben. Die Funktion erfüllt ihre Aufgabe, indem sie die passende Methode aus der Klasse „Optisches Element“ aufruft. Dies ist bei einer Rechteckapertur die Funktion **cutRectangle**. Als Parameter bei diesem Funktionsaufruf werden das E-Feld und die geometrische Größe der Rechteckapertur angegeben.

Rechteckapertur / CRectangle

Methoden:

- Ausblenden des E-Feldes
  ```
  virtual void cutE (CElectricalField& E);
  ```
- Abfrage der räumlichen Größe
  ```
  virtual double getWidthX(void);
  virtual double getHeightY(void);
  ```

Datenelemente:

- geometrische Größe
  ```
  double WidthXRec, HeightYRec;
  ```

Abbildung 4.11 Entwurf der Klasse „Rechteckapertur“

Im Vergleich zur „Rechteckapertur“ besitzt der Klassenentwurf der „Kreisapertur“ nur geringe Unterschiede. Einerseits ist die geometrische Größe der Blende nur durch den Radius bestimmt und andererseits ist die Implementierung der Methode **cutE** unterschiedlich. Bei dieser Funktion wird die Methode **cutCircle** von der Klasse „Optisches Element“ aufgerufen.
Der Klassenentwurf wird in Abbildung 4.12 gezeigt.

Kreisapertur / CCircle

Methoden:

- Ausblenden des E-Feldes
  ```
  virtual void cutE (CElectricalField& E);
  ```
- Abfrage der räumlichen Größe
  ```
  virtual double getWidthX(void);
  virtual double getHeightY(void);
  ```

Datenelemente:

- geometrische Größe
  ```
  double RadiusCircle;
  ```

Abbildung 4.12 Entwurf der Klasse „Kreisapertur“

Nachdem in der Klassenhierarchie der Zweig der „Apertur“ vollständig beschrieben wurden, wird nun die Klasse „Linse“ beschrieben. Sie ist ähnlich wie die „Apertur“ aufgebaut. Daher sollen hier nur die Unterschiede erwähnt werden. Eine reale Linse besitzt einen Brechungsindex und eine Dicke. Diese sind Elementvariablen der Klasse (**RefractiveIndex**, **Thickness**). Sie werden aus der Eingabedatei eingelesen. Weiterhin enthält eine Linse eine Transmissionsfunktion **T** (vgl. Gleichung (4.1)). Sie ist vom bereits beschriebenen Typ `CTransmissionFunction`.

Bei den Methoden **calc(CElectricalField& E)** und **calc(CElectricalField& EX, CElectricalField& EY)** soll der genaue Ablauf innerhalb der Methoden den entsprechenden Funktionen der Apertur gegenübergestellt werden (vgl. Klassenentwurf der Klasse „Apertur).

calc(CElectricalField& E):

1. Ausblenden des elektrischen Feldes an der Apertur (**cutE(E)**)
2. Berechnung der Transmissionsfunktion (**calcTransmissionFunktion(E)**)
3. Multiplikation der Transmissionsfunktion mit dem E-Feld (**applyTransmissionFunctionToElectricalField(E, T)**)
4. Berechnung der Übertragungsfunktion (**calcTransferFunction(E)**)
5. Transformation des E-Feldes in den Frequenzbereich (**E.Ffft**)
6. Multiplikation des E-Feldes mit der Übertragungsfunktion (**multiplicationAndNormalization(E, H)**)
7. Rücktransformation des E-Feldes in den Ortsbereich (**E.Bfft**)
8. Austausch der vier Quadranten des E-Feldes (**E.ChangeField**)

calc(CElectricalField& EX, CElectricalField& EY):

1. Berechnung des Feldes EX (**calc(EX)**)
2. Berechnung des Feldes EY (**calc(EY)**)
3. Erhöhung der Zählvariablen (**CalcNumber++**)
4. ggf. Zerstörung der Übertragungs- und der Transmissionsfunktion (**DestroyH**, **DestroyT**)
5. Erzeugung eines eindeutigen Dateinamens für die Ausgabe
6. ggf. Ausgabe der elektrischen Felder (**EX.write(OutEFieldX)**)

Ein Vergleich mit der „Apertur“ ergibt, dass zwei zusätzliche Methoden existieren. Eine ist die Funktion **calcTransmissionFunction**. Sie berechnet in Abhängigkeit des Objekts (Konvex- oder Zylinderlinse) die passende Transmissionsfunktion. Da diese Funktion typabhängig ist, ist sie in der Klasse Linse nur als rein virtuelle Funktion definiert.
Die andere Funktion multipliziert die Transmissionsfunktion elementweise mit dem elektrischen Feld. Ihr Name ist **applyTransmissionFunctionToElectricalField**.
Die Übersicht über die Klasse „Linse“ ist in Abbildung 4.13 zu sehen.

Linse / CLens

Methoden:

- Berechnung von zwei elektrischen Feldern (= 1 Bild)

```
virtual void calc (CElectricalField& EX,
                   CElectricalField& EY);
```

- Berechnung eines E-Feldes *E*

```
virtual void calc (CElectricalField& E);
```

- Multiplikation der Transmissionsfunktion mit dem elektrischen Feld

```
void applyTransmissionFunctionToElectricalField
     (CElectricalField& E, CTransmissionFunction& T);
```

- Berechnung der Transmissionsfunktion

```
virtual void calcTransmissionFunction
                    (CElectricalField& E)      =0;
```

- Ausblenden des E-Feldes an der Blende

```
virtual void cutE (CElectricalField& E)      =0;
```

- Abfrage der räumlichen Größe

```
virtual double getWidthX(void)               =0;
virtual double getHeightY(void)              =0;
```

Abbildung 4.13 Entwurf der Klasse „Linse“

Die Klasse „Linse“ (Abbildung 4.14) dient als Grundlage für die Klasse „Konvexlinse“. Diese Linse besitzt eine Brennweite (**FocalLenghtCon**) und einen Linsenradius (**RadiusCon**). Eine Methode der Konvexlinse ist identisch mit dergleichen Methode in der Klasse „Kreisapertur“. Es ist die Methode **cutE**. Sie blendet einen Teil des elektrischen Feldes aus. Um diese Aufgabe durchzuführen, wird die Funktion **cutCircle** aus der Klasse „Optisches Element“ aufgerufen.

Die wichtigste Methode der Klasse „Konvexlinse“ ist die Funktion **calcTransmissionFunction**. Sie berechnet die Transmissionsfunktion. Dabei ist zu beachten, dass sie als Parameter eine Referenz auf ein elektrisches Feld benötigt. Das E-Feld wird benötigt, um die diskreten Werte der Transmissionsfunktion genau an den Stützstellen des E-Feldes zu berechnen (vgl. Übertragungsfunktion). Dafür werden verschiedene Informationen des Feldes benötigt. Dies sind die geometrische Größe des Feldes (Höhe, Breite) sowie die Anzahl der Stützstellen in x- und y-Richtung (NX, NY). Außerdem wird die Wellenlänge des elektrischen Feldes benötigt, damit die Eigenschaften der Transmissionsfunktion wirklich mit denen des elektrischen Feldes übereinstimmen. Nach Berechnung der Transmissionsfunktion wird die Statusvariable **TValid** auf true gesetzt.

Konvexlinse / CConvexConcave

Methoden:

- Berechnung der Transmissionsfunktion

```
virtual void calcTransmissionFunction (CElectricalField& E);
```

- Ausblenden des E-Feldes an der Blende

```
virtual void cutE (CElectricalField& E);
```

- Abfrage der räumlichen Größe

```
virtual double getWidthX(void);
virtual double getHeightY(void);
```

Datenelemente:

- Brennweite und Radius der Linse

```
double FocalLenghtCon, RadiusCon;
```

Abbildung 4.14 Entwurf der Klasse „Konvexlinse“

Die letzte noch zu erläuternde Klasse ist die „Zylinderlinse“ (siehe Abbildung 4.15). Bei ihr existieren als Datenelemente die Variablen **WidthXCyl**, **HeightYCyl**, **FocalLenghtCyl** und **Orientation**. Die ersten drei Variablen sind bereits bekannt und benötigen keine weitere Erklärung. Die Variable **Orientation** beschreibt die Lage der Symmetrieachse der Zylinderlinse. Diese kann horizontal oder vertikal sein und wird vom Anwender in der Eingabedatei festgelegt.
Die Funktionsrümpfe der Methoden sind identisch mit denen der „Konvexlinse“. Die Implementierungen der Funktionen sind aber natürlich unterschiedlich. Einerseits, weil das Abschneiden mit Hilfe der Funktion **cutRectangle** erfolgt und andererseits weil die Transmissionsfunktion mit einer anderen mathematischen Gleichung berechnet wird.

Zylinderlinse / CCylinder

Methoden:

- Berechnung der Transmissionsfunktion
  ```
  virtual void calcTransmissionFunction (CElectricalField& E);
  ```
- Ausblenden des E-Feldes an der Blende
  ```
  virtual void cutE (CElectricalField& E);
  ```
- Abfrage der räumlichen Größe
  ```
  virtual double getWidthX(void);
  virtual double getHeightY(void);
  ```

Datenelemente:

- Brennweite, Größe und Orientierung der Linse
  ```
  double FocalLenghtCon, WidthXCyl, HeightYCyl;
  char Orientation;
  ```

Abbildung 4.15 Entwurf der Klasse „Zylinderlinse“

4.2 Gültigkeitskonzept

Nachdem mit den Methoden und Datenelementen der „Zylinderlinse“ die letzten wesentlichen Elemente des Gesamtprogramms vorgestellt worden sind, wird als nächstes das Gültigkeitskonzept des Programms erläutert.
Hierzu ist erst ein Überblick über den Programmablauf notwendig. In der **main**-Funktion werden zuerst die Objektdaten aus der Datei `Camera.dat` eingelesen und die notwendigen Instanzen mit den entsprechenden Datenelementen erzeugt (`main.cpp:12`). Anschließend wird die Berechnung dieser Kameraelemente der Reihe nach gestartet (`main.cpp:18`). Gleichzeitig werden in dieser Routine die Ergebnisse von den jeweiligen Kameraelementen selber ausgegeben. Nach einer Berechnung kann die Eingabedatei verändert werden und das Programm neu gestartet werden.
Nachteilig an dieser Art der Berechnung ist, dass eine Parametervariation nur mit sehr viel Aufwand erfolgen kann. Besser wäre es in der Eingabedatei beliebig viele Schleifen zu definieren, die den zu variierenden Parametern zugeordnet werden können (`Camera.dat:3,8`). Nach jedem Berechnungsdurchgang kann dann der Parameter verändert und eine neue Berechnung automatisch gestartet werden. Dieses Vorgehen wäre, wie bei der im Anhang befindlichen Eingabedatei noch vertetbar, weil bei der Wellenlänge sowohl die Übertragungsfunktion, als auch die Transmissionsfunktion (**T**) neu berechnet werden müssen. Wird jedoch als Parameter nur der Radius der Kreisapertur verändert, können die alten Funktionen (**H**, **T**) weiter verwendet werden. Aus diesem Grund ist es wichtig dem Programm mitzuteilen welche Werte geändert worden sind. Gleichzeitig ist zu koordinieren, welchen Einfluss diese Veränderung hat, d.h. welche Parameter und Felder neu berechnet werden müssen. Dies erfolgt mit Hilfe der Change-Variablen. Dabei wird jeder Eingabewert dahingehend bewertet, welche Bereiche er beeinflusst. Für das gesamte Programm erhält man auf diese Weise fünf unterschiedliche Change-Variablen. Diese sind im weiteren tabellarisch mit den entsprechenden Datenelementen und einer kurzen Erläuterung aufgeführt.

1. **changeE**
   Abhängigkeit: Wellenlänge
   Kommentar: Nur wenn sich die Wellenlänge ändert, muss ein anderes Eingangsfeld angefordert werden. Daher existiert diese Variable nur in der Klasse „Kamera“.
   Vorhanden in: Camera.h

2. **changeH**
   Abhängigkeit: Berechnungsmodell, Abstand zum nächsten Element, Verschiebung
   Kommentar: Nur diese Attribute haben einen Einfluss auf die Übertragungsfunktion.
   Vorhanden in: OpticalElement.h

3. **changeT**
   Abhängigkeit: Brechungsindex, Dicke der Linse, Brennweite, Orientierung
   Kommentar: Diese Datenelemente verändern ausschließlich die Übertragungsfunktion.
   Vorhanden in: OpticalElement.h (Begründung: Die später erläuterten change-Funktionen müssten sonst fast identisch in derKlasse „Lens“ und der „Apertur“ implementiert werden.)

4. **changeSize**
   Abhängigkeit: geometrische Größe (Breite, Höhe, Radius)
   Kommentar: Bei einer Größenänderung müssen die bereits besprochenen Probleme überprüft werden:
   - maximale Ausdehnung der Kamera variiert
     $\Rightarrow$ Größe des E-Feldes ändert sich
   - Auflösung des letzten Schirms wird höher oder niedriger
     $\Rightarrow$ Auflösung des E-Feldes wird verändert
   - Größe des ersten Kameraelements
     $\Rightarrow$ Größe des angeforderten Eingangsfeldes kann sich ändern

   Vorhanden in: OpticalElement.h, Screen.h

5. **changeEle**
   Abhängigkeit: changeSize
   Vorhanden in: Camera.h
   Kommentar: Gibt an, ob mindestens ein Kameraelement in der Größe verändert wurde.

Diese fünf Variablen lassen sich auch bezüglich ihrer Klassenzugehörigkeit gliedern. Die Variablen eins und fünf gehören zu `Camera.h`, zwei bis vier zu `OpticalElement.h` und **changeSize** schließlich noch zu `Screen.h`.
Zusätzlich zu diesen Variablen gibt es in den drei Klassen noch eine weitere Change-Variable. Sie besitzt direkt den Namen **change**. Sie zeigt an, ob mindestens eine der

unter ihr angeordneten change-Variablen (**changeH**, **changeT**, **changeSize** bzw. **changeE** oder **changeEle**) eine Änderung bekannt gibt.

Wird nach einem Berechnungsdurchgang ein Wert durch eine Schleife verändert, so muss auch die entsprechende change-Variable gesetzt werden (= true). Danach werden die entsprechenden Change-Variablen untereinander koordiniert. D.h. wenn **change**=true ist, dann muss auch **changeH**=true und **changeT**=true sein usw. Diese Aufgabe erfüllt eine Funktion mit dem Namen **syncChange**, die in allen drei Klassen vorhanden ist.
Anschließend wird in der Methode **set** die entsprechende Maßnahme getroffen. Wenn zum Beispiel die Variable **changeH** gesetzt ist, dann muss die Statusvariable **HValid** (gibt die Gültigkeit der Werte in der Übertragungsfunktion an) den Wert nicht gültig (= false) annehmen.
Ein weiteres Beispiel ist, wenn **changeEle** den Wert true hat. Daraus kann gefolgert werden, dass mindestens in einem Kameraelement die Größe verändert wurde und daher erst die neue maximale räumlich Ausdehnung der Kamera bestimmt werden muss. Eine Änderung der Größe kann als Resultat ergeben, dass ein neues Eingangsfeld angefordert werden muss.
Die Methode **set** ist damit hauptverantwortlich für die eigentliche Berechnung und ist im Quellcode entsprechend dokumentiert. Erst nach dem Aufruf der beiden beschriebenen Funktionen kann eine neue Berechnung gestartet werden.

Dieses insgesamt sehr aufwendige Verfahren wird durch das Einbinden der bereits vorhandenen Klasse `CInput` erleichtert. Zusätzlich müssen einige Programmmodifikationen durchgeführt werden, um die volle Unterstützung auszunutzen. Für ein besseres Verständnis der Implementierung und der Benutzung des CInput-Objekts werden als nächstes die ersten Zeilen der main-Funktion näher betrachtet.
Zuerst wird nach Erzeugung eines Camera-Objekts (`main.cpp:6`) ein CInput-Objekt erzeugt (`main.cpp:9`). Dem Konstruktor werden dabei zwei Dateinamen übergeben. Einerseits der Dateiname der die zu berechnende Kamera enthält und als zweites eine Datei die dem Anwender die Eingabe von Stringkonstanten ermöglicht (vgl. `TypesCamera.txt:10`).
Anschließend werden aus der Datei `Camera.dat` aus dem Block A die Schleifen geladen (`main.cpp:12`). Damit werden dem Input-Objekt verschiedene Laufindizes mit deren Anfangswert, Bedingung und Reinitialisierung mitgeteilt. Danach erfolgt der Sprung zu Block B (`main.cpp:13`) und die Kameradaten werden eingelesen (`main.cpp:14`). Dieses Einlesen mit Hilfe der Funktion **read** aus der Kameraklasse soll an dieser Stelle näher untersucht werden (`Camera.cpp:80-111`).
Der Funktion **read** wird beim Aufruf eine Referenz auf das Input-Objekt übergeben. Innerhalb des Programms kann dann nicht nur die Eigenschaft der Parametervariation, sondern auch die erweiterte Funktionalität der Klasse zum Einlesen der Datenelemente genutzt werden. Als erstes wird die Anzahl der Kameraelemente eingelesen. Hierzu wird die Methode **readFunction** der Klasse `CInput` mit der Adresse von der Elementanzahl als Parameter aufgerufen (`Camera.cpp:83`). Die Funktion erkennt automatisch den Datentyp, liest den ersten Wert im Block B der Eingabedatei ein und geht in der Datei in die nächste

Zeile. Auf diese Art können alle size_t, integer, double oder float Werte gelesen werden.
Als nächstes wird ein boolscher Wert mit Hilfe der Funktion **readType** im Datenelement **output.WriteCamera** gespeichert (`Camera.cpp:84`). In diesem Datenelement liegt der Wert vor, ob nach einem Berechnungsdurchgang alle Attribute der aktuellen Kamera ausgegeben werden sollen. Er ist Bestandteil der Klasse „Kamera".
Diese beiden eingelesenen Daten können nicht mit Parametervariation verändert werden. Sie sind infolgedessen für alle Berechnungen fest.

Dies gilt nicht für die als nächstes einzulesende Wellenlänge. Sie wird zwar genauso wie die Elementanzahl behandelt (`Camera.cpp:86`), jedoch wird zuvor eine **setChange** Funktion mit der für die Wellenlänge entsprechenden Change-Variablen aufgerufen. Dies zeigt dem Input-Objekt an, dass alle folgenden Werte mit der Change-Variable verknüpft sind. Anschließend werden die einzelnen Objekte eingelesen. D.h. wird nach einem Berechnungsvorgang die Wellenlänge durch eine Schleife variiert, wird automatisch die Variable **changeE** mit Hilfe des Input-Objekts gesetzt (= true). Bevor dann die neue Berechnung mit veränderten Parametern gestartet wird, werden erst die bereits erklärten Funktionen **syncChange** und **set** aufgerufen. Damit wird unter Umständen ein neues Eingangsfeld angefordert und die Übertragungs- und Transmissionsfunktion einzelner Elemente neu berechnet. Ist auch diese Berechnung mit der Ausgabe der elektrischen Felder und der Intensitätsverteilung erfolgreich beendet worden, werden erst alle Change-Variablen mit Hilfe der Funktion **resetAllChange** auf ihre Ausgangswerte (= false) (`main.cpp:20`) gesetzt. Anschließend werden mit Hilfe der Klasse `CInput` die nächsten Schleifenwerte in die Datenelemente übertragen (`main.cpp:23`) und der nächste Berechnungsvorgang wie beschrieben gestartet.

4.3 Programmablauf

Nachdem in den Kapiteln 4.1 und 4.2 alle wichtigen Methoden, Datenelemente und Konzepte vorgestellt worden sind, soll an dieser Stelle die Beschreibung des Programmablaufs folgen. Hierzu wird bewusst eine sehr einfache Konstellation von Kameraelementen gewählt. Es handelt sich um eine Konvexlinse mit einem dahinter befindlichen Schirm. Die genauen Geometriedaten sind für die Beschreibung nebensächlich, und werden nicht näher betrachtet. Trotzdem wird diese Beschreibung alle Bereiche des Programms ansprechen, weil nicht nur die Übertragungsfunktion, sondern auch die Transmissionsfunktion benötigt wird und zusätzlich alle Daten ausgegeben werden sollen.

Zuerst erfolgt in der Funktion **main** das Erzeugen der notwendigen Objekte und das Einlesen der Eingabedatei (`main.cpp:5-12`). Im letzten Unterkapitel wurde das Einlesen bis in die Methode **read** in der Klasse „Kamera" verfolgt (`Camera.cpp:86`). An dieser Stelle soll nun fortgefahren werden. Als nächstes wird in der for-Schleife in den nächsten Block gesprungen (`Camera.cpp:91`) und

zuerst die Art des einzulesenden Kameraelements ermittelt (`Camera.cpp:94`). In Abhängigkeit vom Typ des Kameraelements wird dann ein solches Objekt erzeugt und in die Liste der Kameraelemente aufgenommen (`Camera.cpp:95-104`). Zu beachten ist, dass die Elemente nur mit dem Standardkonstruktor definiert werden. Deswegen müssen als nächstes alle Datenelemente des Objekts eingelesen werden. Dazu gehört, dass das Element seine Positionsnummer innerhalb des Kameraobjektes erhält (`Camera.cpp:108`). Anschließend wird die Funktion **read** des Kameraelements mit dem Input-Objekt als Parameter aufgerufen. Damit ist dem Element direkt bekannt, aus welcher Datei und an welcher Stelle es seine eigenen Attribute einlesen soll (`Camera.cpp:110`). Das Einlesen in dieser Methode erfolgt dann so, wie es im vorherigen Unterkapitel beschrieben wurde (`ConvexConcave.cpp:104-119`).
Nach diesem Einlesen wird die for-Schleife ein zweites Mal durchlaufen, da die Kamera aus zwei Kameraelementen besteht. Bei diesem Durchgang wird dann der Schirm erzeugt, eine Positionsnummer vergeben und seine Attribute eingelesen. Damit ist die Abarbeitung der Schleife beendet. Anschließend werden alle Change-Variablen gesetzt, weil noch kein Feld berechnet oder eingelesen bzw. angefordert wurde.

In der Funktion **main** werden danach die Funktionswerte von eventuell vorhandenen Schleifen auf ihre Anfangswerte gesetzt (`main.cpp:13`).
Nach dem Einlesen der Eingabedatei liegen alle definierten Kameraelemente in einem Vektor **CamEle**. Genau genommen liegen die Adressen der Objekte in einem Vektor vom Typ `CPointerArray`. Damit ist sichergestellt, dass im weiteren Programm die Methoden der Kameraelemente mit dem Elementverweisoperator angesprochen werden können.

Als nächstes kann die Berechnung beginnen. Sie wird mit unterschiedlichen Datenelementen so oft durchgeführt wie die Schleifen verändert werden (`main.cpp:15,23`).
Zuerst wird die Funktion **syncChange** der Klasse Kamera aufgerufen (`main.cpp:16`). In ihr werden zunächst alle Kameraelemente angewiesen, ihre Change-Variablen zu synchronisieren und auf diese Art Änderungen bekant zu geben (`Camera.cpp:62`). Anschließend wird überprüft, ob von einem Element die Größe verändert wurde. Wenn dies zutrifft wird die Variable **changeEle** in der Klasse Kamera gesetzt (`Camera.cpp:68-72`). Danach werden die eigenen Change-Variablen synchronisiert (`Camera.cpp:76`) und die Verantwortung des Programmablaufs an die **main**-Funktion zurückgegeben.
Diese ruft die Methode **set** auf (`main.cpp:17`). Sie ist verantwortlich für das Setzen der richtigen Statusvariablen. Ihre Aufgabe kann in vier Bereiche gegliedert werden. Zuerst wird für den Fall, dass die Größe eines Elements verändert wurde, die neue maximale räumliche Ausdehnung der Kamera errechnet (`Camera.cpp:232-240`).
Der zweite Bereich verändert die Change-Variablen jedes Elements, falls das elektrische Feld von außen verändert wurde. Er setzt automatisch die **changeH** und **changeT** Variablen aller Elemente, weil bei einer Änderung der Eigenschaften des elektrischen Feldes die Felder **H** und **T** ihre Gültigkeit verlieren.
Als drittes werden die einzelnen Kameraelements dazu aufgerufen, ihre eigenen **set** Funktionen abzuarbeiten. In dem hier behandelten Beispiel wird zuerst die Methode

aus dem optischen Element aufgerufen (`OpticalElement.cpp:68-98`). In ihr werden alle Kombinationen von gesetzten Change-Variablen berücksichtigt und dementsprechend die Statusvariablen gesetzt. Diese **set**-Funktion liefert nach ihrer Abarbeitung den Wert true zurück, falls die Statusvariablen verändert wurden. Andernfalls wird der Wert false zurückgegeben. Der Rückgabewert wird aber in diesem Programm nicht weiter berücksichtigt. Er ist jedoch aus Gründen der Kompatibilität zum Gesamtprojekt implementiert.
Im letzten Bereich wird den Statusvariablen **ERead** und **EFillUp** in derKamera der Wert false zugewiesen. Diese Wertzuweisung muss vor jedem Berechnungsvorgang erfolgen, weil das elektrische Feld „in place" berechnet und manipuliert werden. D.h. die Daten des E-Feldes wurden nach einer eventuellen vorherigen Berechnung überschrieben.

Nach dem Setzen der Statusvariablen erfolgt der Rücksprung in die Funktion **main**. Dort wird als nächstes die „richtige" Berechnungsfunktion **calc** aufgerufen (`main.cpp:18`). In dieser Methode wird zuerst das Eingangsfeld aus den Dateien mit den Namen „EingangsfeldX.dat" und „EingangsfeldY.dat" eingelesen (`Camera.cpp:134-145`) oder zu Testzwecken ein passendes Eingangsfeld erzeugt (`Camera.cpp:147-166`). Anschließend wird die Statusvariable **ERead** auf true gesetzt (`Camera.cpp:168`).
Danach wird überprüft, ob die Parameter der beiden elektrischen Felder wirklich identisch sind und die Wellenlänge durch die Wellenlänge von der Kamera ersetzt. An dieser Stelle ist später der Aufruf zur Anforderung eines „passenden" elektrischen Feldes zu implementieren. Als Parameter sind dann folgende Werte anzugeben: Auflösung des letzten Schirms, Höhe und Breite des ersten Elements sowie die Wellenlänge von der Kamera.
Anschließend wird überprüft, ob das Eingangsfeld wirklich größer als das erste Kameraobjekt ist (`Camera.cpp:184`). Erst danach können die elektrischen Felder mit Nullen bis auf die maximale Höhe und Breite der Kamera aufgefüllt werden (`Camera.cpp:190-194`). Nachdem an dieser Stelle das „richtige" Eingangsfeld für die Berechnung vorliegt werden die einzelnen Berechnungsmethoden aller Kameraelemente der Reihe nach aufgerufen (`Camera.cpp:197-199`).

Zuerst wird die **calc**-Funktion der Konvexlinse (`Lens.cpp:24-96`) mit den beiden E-Feldern als Parameter aufgerufen. In dieser Methode werden zuerst die beiden Felder für sich berechnet (`Lens.cpp:40, 41`). Dort wird zunächst das elektrische Feld beim Durchgang durch die Apertur mit der Funktion **cutE** „abgeschnitten" (`Lens.cpp:103, ConvexConcave.cpp:99`). Danach wird die Transmissionsfunktion berechnet (`Lens.cpp:106, ConvexConcave.cpp:29-93`) und auf das E-Feld angewendet (`Lens.cpp:109`).
Anschließend wird das elektrische Feld in den Frequenzbereich transformiert (`Lens.cpp:112`). Bei der Transformation wird zuerst überprüft, ob der aktuelle Plan gültig ist. Ist diese Bedingung nicht erfüllt, wird zunächst ein Plan für die Vorwärtstransformation und ein Plan für die Rückwärtstransformation berechnet (`Field.cpp:35-37`). Anschließend wird die Transformation angewendet (`Field.cpp:41`). Nach dieser Manipulation der Werte im elektrischen Feld wird die Übertragungsfunktion errechnet (`Lens.cpp:115`). Dazu wird die Methode aus der Klasse des optischen Elements aufgerufen und die Eigenschaften des E-Feldes

mit denen der aktuellen H-Funktion verglichen (OpticalElement.cpp:108-114). Diese Überprüfung ist für den Fall notwendig, dass sich die Größe eines Kameraelements im Vergleich zum vorherigen Berechnungsdurchgang geändert hat. Denn nur unter der Bedingung, dass das elektrische Feld im Vergleich zur letzten Berechnung erweitert wurde, muss eine neue Übertragungsfunktion berechnet werden. Andernfalls kann die alte Funktion weiter verwendet werden.
Danach erfolgt in Abhängigkeit von der Statusvariable **HValid** und des gewählten Modells die Berechnung (OpticalElement.cpp:118-143). Dabei wird die Verschiebung gleich mit berücksichtigt. Auf die Implementierung dieser Funktionen soll hier nicht eingegangen werden. Der Quelltext ist dafür ausreichend kommentiert worden.
Nun wird die Übertragungsfunktion auf das elektrische Feld angewendet, die Rücktransformation durchgeführt, sowie die FFT mit ihrer Vertauschung der vier Quadranten rückgängig gemacht (Lens.cpp:118-124).
Damit ist ein elektrisches Feld bei einem Durchgang durch eine Konvexlinse und anschließender freier Propagation berechnet worden und es erfolgt der Rücksprung in die **calc(EX, EY)**-Funktion (Lens.cpp:40). Dort wird gleich das zweite Feld berechnet. Hier muss aber keine Übertragungs- oder Transmissionsfunktion berechnet werden.
Danach wird die Anzahl der berechneten „Bilder“ (1 Bild = 2 E-Felder) um eins erhöht (Lens.cpp:42) und die Funktionen H und T eventuell zerstört (Lens.cpp:47-54). Anschließend werden, sofern die Ausgabe des elektrischen Feldes erwünscht ist, zwei eindeutige Dateinamen generiert und die Felder mit Hilfe der COut-Objekte ausgegeben (Lens.cpp:64-95).

Diese Ausgabe beendet die Bearbeitung der elektrischen Felder durch die Konvexlinse. Als nächstes wird aus dem aktuellen E-Feld die Intensitätsverteilung errechnet (Camera.cpp:198). Hierzu wird die **calc**-Funktion des Schirms aufgerufen (Screen.cpp:87-161). Diese berechnet aus der aktuellen Stützstellenzahl des elektrischen Feldes eine Intensitätsverteilung derselben Größe (Screen.cpp:98-119). Danach wird ein eindeutiger Dateiname für die Ausgabe generiert (Screen.cpp:129-137). Damit kann ein passendes Out-Objekt geöffnet und die Daten können im gewünschten Format gespeichert werden (ASCII, Binär) (Screen.cpp:139-151). Sollte diese Art der Ausgabe nicht erwünscht sein, besteht noch die Möglichkeit eine Bilddatei im TIFF-Format zu erzeugen (Screen.cpp:154). In der Funktion **writeTIFF** wird zuerst die Dateiendung an den Dateinamen angehängt und die Datei geöffnet (Screen.cpp:177, 178). Anschließend wird die Intensitätsverteilung noch logarithmiert und auf 256 Werte skaliert. Die Logarithmierung muss dabei nicht durchgeführt werden. Sie kann vom Anwender in der Eingabedatei ausgeschaltet werden (Screen.cpp:182-189). Nach dieser Aufbereitung der Intensitätsverteilung für das Bildformat wird die Datei geschrieben (Screen.cpp:196-489). Informationen zu dem Ausgabeformat TIF liegen der Studienarbeit in elektronischer Form bei. Ihr kann der gesamte Aufbau und die verschiedenen Möglichkeiten des Formats entnommen werden. Danach erfolgt der Rücksprung in die **calc**-Funktion des Schirms. In dieser wird noch überprüft, ob die Intensitätsverteilung nach einer Berechnung zerstört werden soll (Screen.cpp:158-160). Diese Funktion kann vom Anwender in der Eingabedatei gesteuert werden und dient der Speicherfreigabe.

Danach wird der Programmablauf in der Methode **calc** in der Kamera und, weil diese ebenfalls beendet ist, in der Funktion **main** fortgesetzt.

Dort wird als nächstes die Ausgabe der aktuellen Kameradaten gestartet (`main.cpp:19`). Dabei werden nur die derzeitigen Datenelemente der einzelnen Objekte ausgegeben. Diese Ausgabedatei besitzt die gleiche Struktur wie die Eingabedatei. Auf diese Art kann der Anwender die erzeugten Bilder und Daten auch nach längerer Zeit den Parametern einer berechneten Kamera zuordnen. Für die Daten ist ähnlich wie beim Einlesen jedes Objekt selber verantwortlich. Auf den genauen Ablauf und die Struktur wird hier nicht näher eingegangen.

Als nächstes werden alle Change-Variablen (Kamera, optisches Element, Schirm) in den Ausgangszustand zurückgesetzt (= false) (`main.cpp:20`). Dies ist notwendig, weil ein Berechnungsdurchgang abgeschlossen ist und als nächstes das Input-Objekt die neuen Schleifenwerte lädt (`main.cpp:23`). Bei diesem Laden der neuen Werte werden gleichzeitig die entsprechenden Change-Variablen gesetzt. Danach startet der Berechnungsvorgang von vorne. Erst wenn alle Schleifen ihren Endwert erreicht haben, wird die do-Schleife abgebrochen und das Programm beendet.

5 Anwendungsfälle und Leistungsvermögen

Ziel dieses Kapitels ist es die Ergebnisse von den wichtigsten Testrechnungen zu präsentieren und mit deren Hilfe die Leistungsfähigkeit des Programms darzustellen.

Als erstes wird ein schmaler Spalt betrachtet. Auf diesen trifft eine ebene Welle. In einem bestimmten Abstand hinter der Apertur wird die Intensitätsverteilung mit einem Schirm beobachtet. Durch den räumlichen Abstand des Schirms zur Apertur werden die Theorien auf ihre Leistungsfähigkeit im Fernfeld und im Nahfeld getestet. Dafür wurden sieben unterschiedliche Spaltbreiten mit den 4 Berechnungstheorien nachgerechnet. Bei der Auswahl der Spaltbreite ist zu beachten, dass bei gleichbleibendem Schirmabstand eine Vergrößerung der Spaltbreite so wirkt, als ob der Schirm weiter in das Nahfeld gerückt wird. Der Parameter, der dieses Verhältnis unter Berücksichtigung der Wellenlänge beschreibt heißt Fresnelzahl N_F. Sie ist eine dimensionslose Kennzahl und wird nach folgender Gleichung berechnet:

$$N_F = \frac{w^2}{\lambda z_0} \tag{5.1}$$

w = halbe Spaltbreite
λ = Wellenlänge
z_0 = Abstand des Schirms von der Apertur

In den Abbildungen 5.1 bis 5.7 ist die berechnete und auf den Maximalwert normierte Intensitätsverteilung graphisch dargestellt. Zusätzlich zu den implementierten Theorien wurde der analytisch exakte Verlauf der Intensität eingezeichnet. Sie wurde mit dem eigens implementierten Programm „Exakte_Spaltberechnung.exe“ ausgerechnet und liegt der Studienarbeit in elektronischer Form bei. Auf diese Berechnung soll in dieser Ausarbeitung aber nicht näher eingegangen werden.
Bei allen Bildern ist zu beachten, dass die x-Koordinate dimensionslos aufgetragen wurde (vgl. [GOO68]). Die entsprechende Koordinate wird mit X bezeichnet. Sie berechnet sich aus

$$X = \frac{x}{\sqrt{\lambda z_0}} \tag{5.2}$$

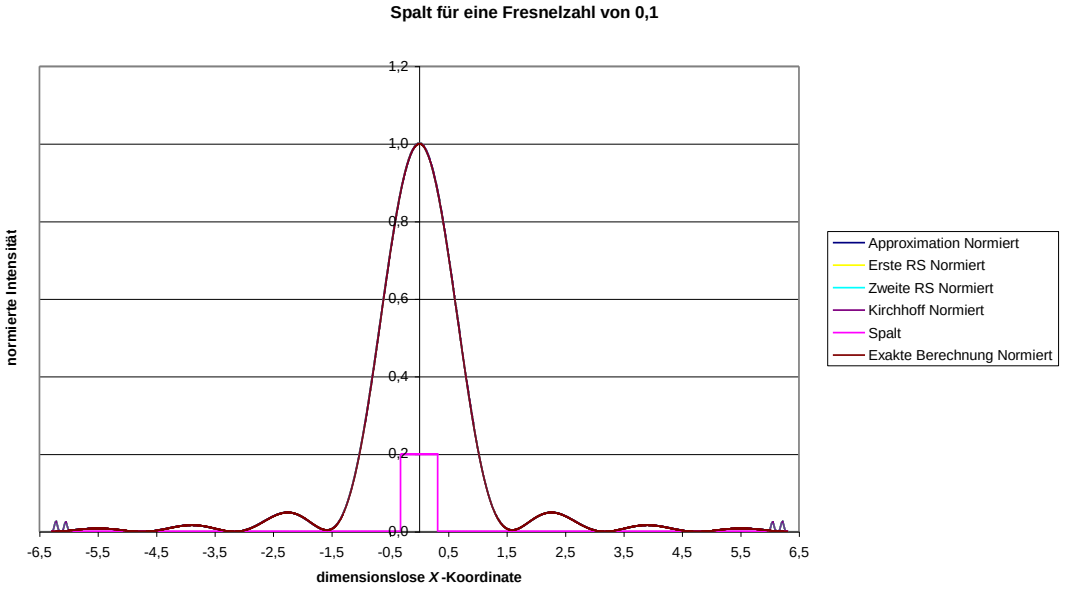


Abbildung 5.1 Spalt mit N_F = 0,1

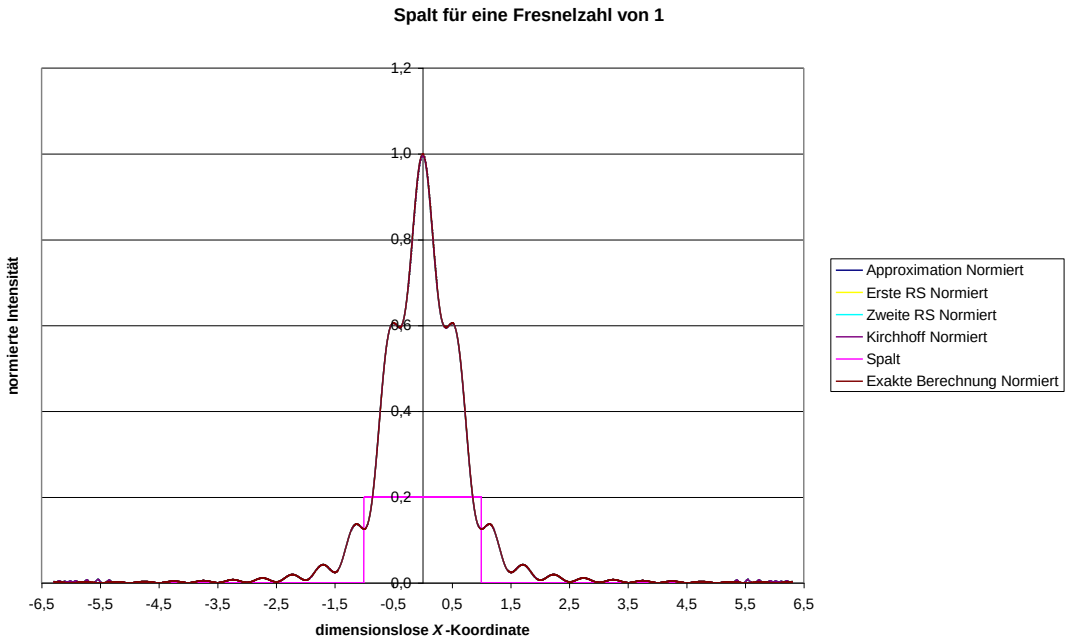


Abbildung 5.2 Spalt mit N_F = 1,0

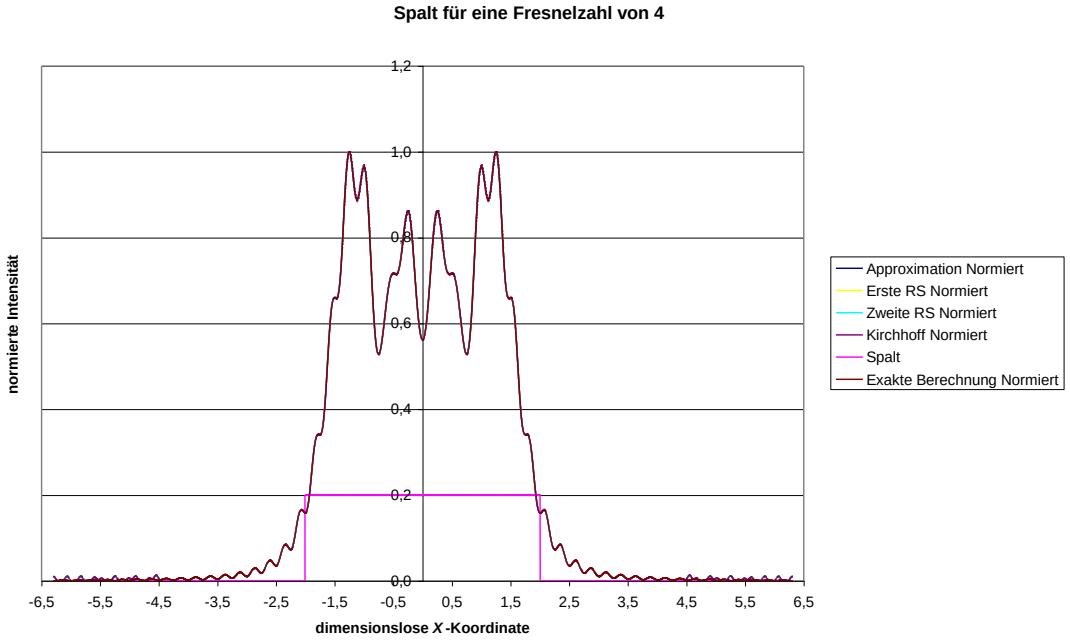


Abbildung 5.3 Spalt mit N_F = 4,0

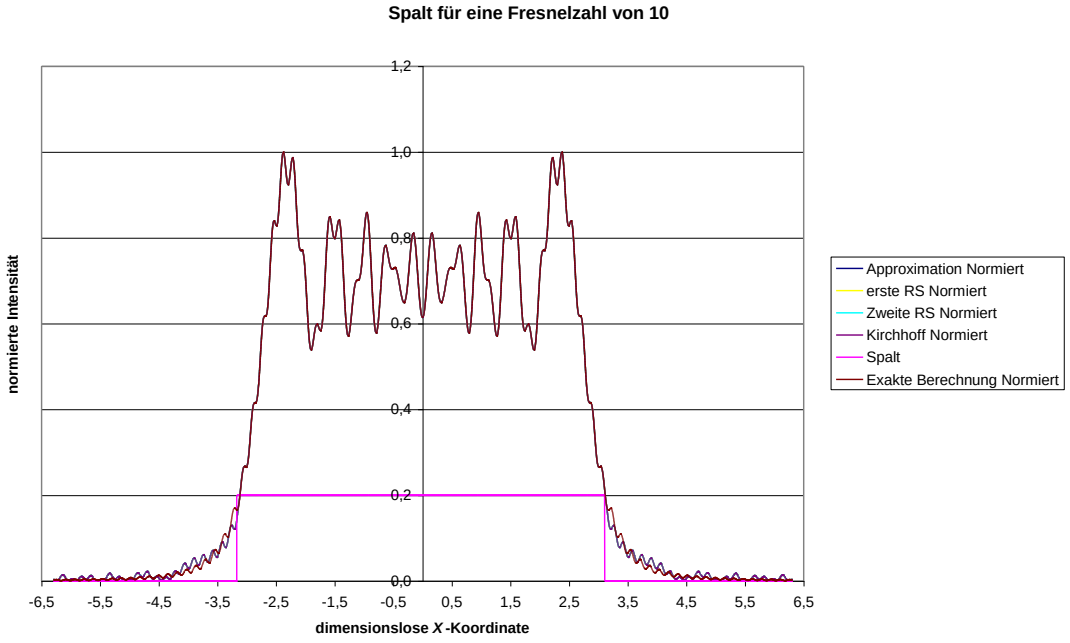


Abbildung 5.4 Spalt mit N_F = 10,0

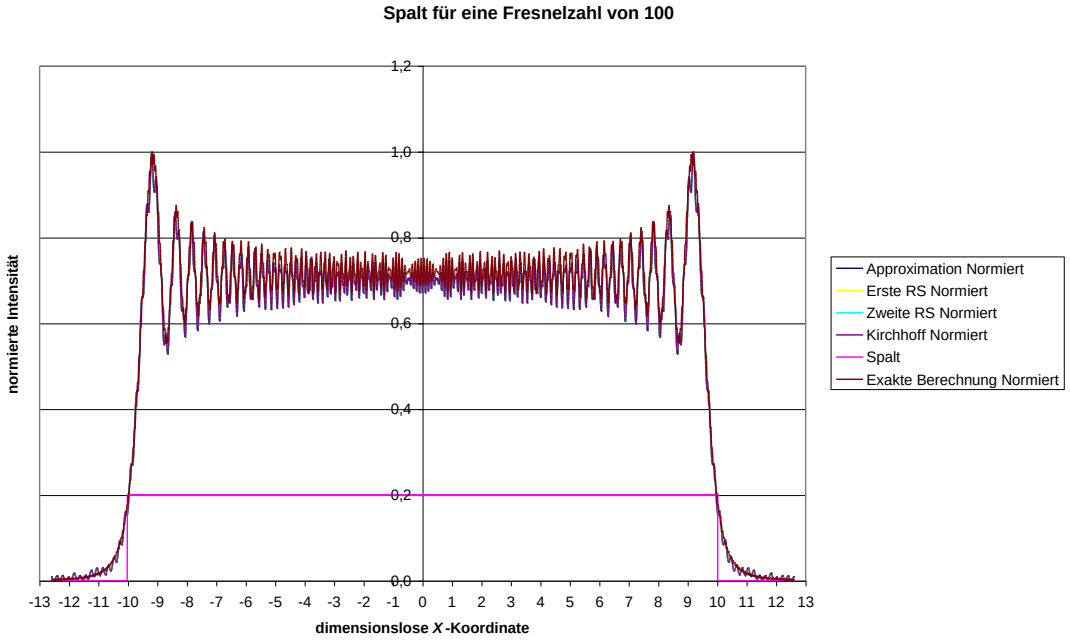


Abbildung 5.5 Spalt mit N_F = 100

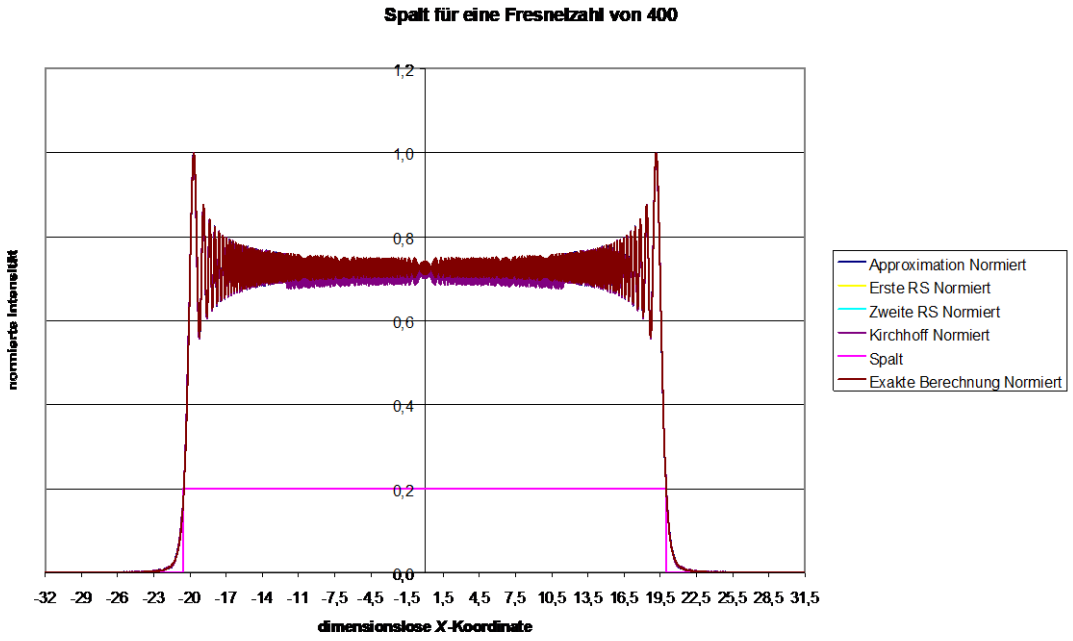


Abbildung 5.6 Spalt mit N_F = 400

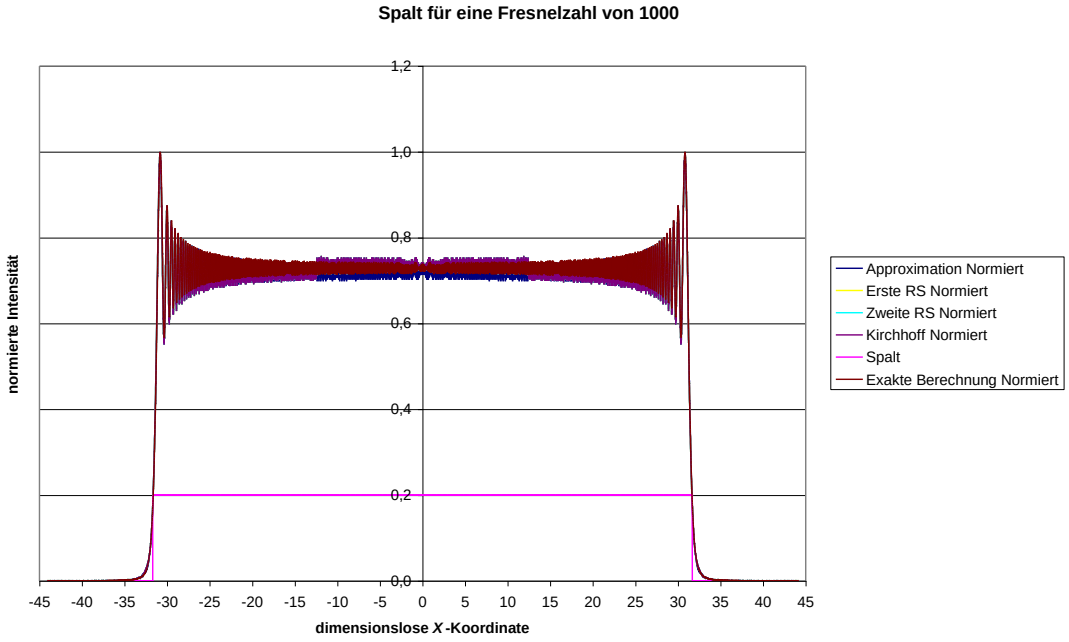


Abbildung 5.7 Spalt mit N_F = 1000

Die graphische Darstellung der Ergebnisse wird im folgenden näher erläutert. In Abbildung 5.1 ist die Intensitätsverteilung im Fernfeld richtig berechnet worden. Lediglich am Rand werden Fehler festgestellt. Diese Fehler sind in Form zweier Peaks leicht erkennbar. Die Ursache dieser Fehler ist jedoch nicht eindeutig und muss daher in weiteren Arbeiten näher untersucht werden.
In den folgenden drei Bildern (N_F = 1; 4; 10) ist kein Unterschied zwischen den verschiedenen Theorien und der exakten Berechnung festzustellen. Lediglich bei einer Fresnelzahl von 10 treten im Bereich außerhalb des Spaltes erste kleinere Abweichungen auf. Hierbei ist bei einer näheren Betrachtung festzustellen, dass die wahren Oszillationen nicht mehr korrekt dargestellt werden können. Es liegt vermutlich eine Unterabtastung vor.
Erst bei einer weiteren Annäherung an die Apertur (N_F = 100) erscheinen auch innerhalb des Spaltbereichs sichtbare Fehler. In diesem Bereich ist auffällig, dass der Verlauf der Intensität („Frequenz“) richtig errechnet wird, aber die Amplitude zu niedrigeren Werten verschoben ist. Die Entstehung dieses Fehlers kann zur Zeit noch nicht gedeutet werden. Hierfür sind weitere Untersuchungen notwendig.
Wird die Spaltbreite weiter erhöht (Abstand erniedrigt) erreicht man die Fresnelzahl von 400. Hier ist außerhalb des Spaltes kaum noch ein Unterschied zwischen Theorie und exakter Berechnung festzustellen. Die Ursache dafür ist, dass die Werte für die Intensität sehr schnell auf unter 0,6% des absoluten Maximums absinken. Aus diesem Grund ist der absolute Fehler in bezug auf die maximale Intensität sehr klein, der relative Fehler an dieser Stelle aber vermutlich genauso groß wie bei N_F = 100.
Innerhalb des Spaltes gilt dasselbe wie für N_F = 100. Der Verlauf der Intensität wird korrekt wiedergegeben. Nur die Amplitude wird zu niedrig errechnet. Zusätzlich ist aber zu beobachten, dass im Bereich von X=-11 bis X=+11 die Amplitude einen plötzlichen Sprung nach unten vollführt. Auch hier ist die Ursache des Fehlersunbekannt. Erwähnenswert ist, dass die Approximation als einzige Theorie einen geringeren Abfall besitzt als die drei anderen Theorien.

Schließlich wurde noch im direkten Nahfeld die Intensitätsverteilung errechnet (N_F = 1000). Damit wird bewiesen, dass die „Frequenzen“ auch direkt hinter der Apertur richtig berechnet werden. Der Verlauf der Amplituden ist ähnlich dem vorher betrachteten Bild. Im Bereich von X=+12 bis X=+30 werden erneut von allen Theorien zu niedrige Werte berechnet. Wird der Abschnitt in der Nähe der optischen Achse (X=-12 bis X=+12) analysiert, beobachtet man wieder einen Sprung im Amplitudenverlauf. Dieser Sprung erfolgt jedoch zu höheren Werten und wird nur von der Kirchhoff-Theorie und den beiden Rayleigh-Sommerfeld-Gleichungen gezeigt.
Diese einfachen Berechnungen zeigen bereits, dass die Anwendung der Approximation als ausreichend bewertet werden kann. Zudem benötigt sie für die Berechnung erheblich weniger Multiplikationen.

Bei der nächsten Testrechnung wird das Problem der Unterabtastung näher betrachtet. Gleichzeitig soll die Funktionstüchtigkeit der Konvexlinse bewiesen werden.
Das Phänomen der Unterabtastung wurde bei einer einfachen Versuchsanordnung beobachtet. Diese Anordnung besteht aus einer Konvexlinse und einem Schirm, der zur Linse einen Abstand von 1,5m besitzt. Als Eingangsfeld wird das elektrische Feld errechnet, welches durch zwei Punktquellen im Abstand von 35cm vor der Linse entsteht. Der Abstand der Punktquellen beträgt ca. 130μm. Das berechnete elektrische Feld und der Schirm haben eine Größe von 1cm × 1cm.
Der Radius der Linse wird größer als der Schirm gesetzt (Einfluss der Kreisapertur wird vernachlässigt). Da dieser Test noch zu einem Zeitpunkt erfolgte, als die Abfrage nach der Größe des ersten Elements noch nicht im Programm implementiert war, wird in der Ebene der Linse nur eine einfache Phasenverschiebung über das gesamte E-Feld durchgeführt. Gleichzeitig dient die Linse als Rechteckapertur mit einer Größe von 1cm × 1cm. Diese Anordnung ist in Abbildung 5.8 graphisch dargestellt.

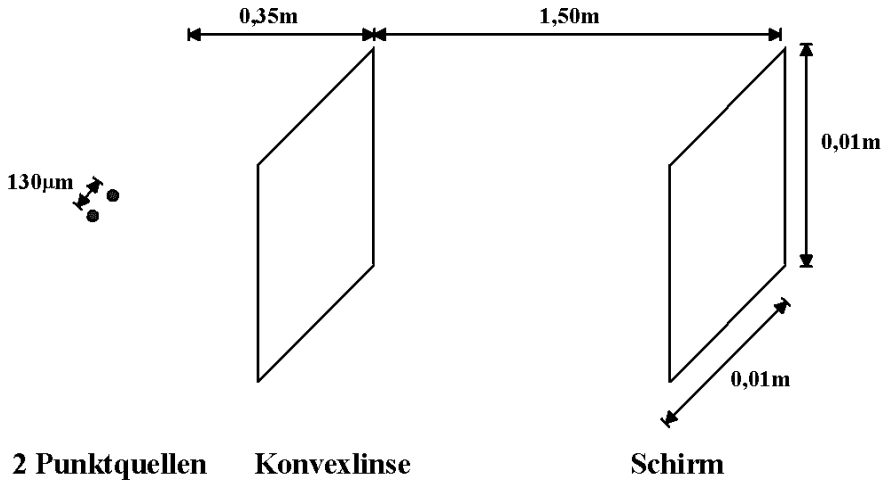
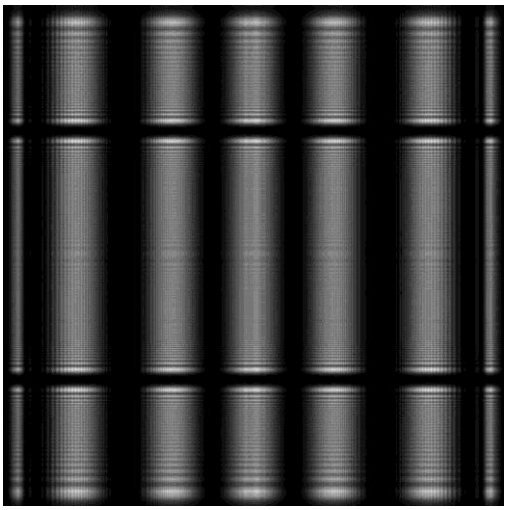


Abbildung 5.8 Berechnete Versuchsanordnung

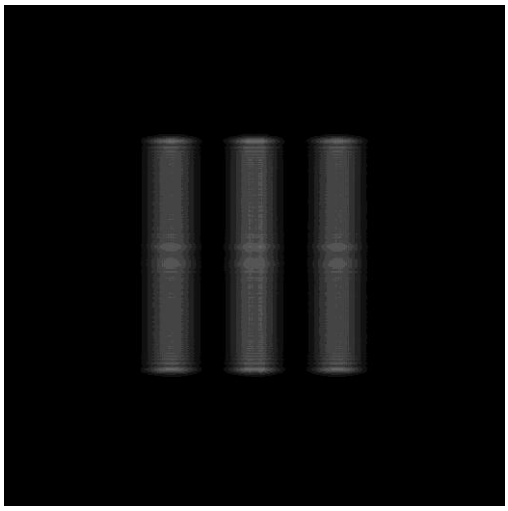
Die Intensitätsverteilung auf dem Schirm wurde mit einer Auflösung von 512, 1024, 1536, 2048 und 4096 nachgerechnet. Die Bilder sehen wie folgt aus:



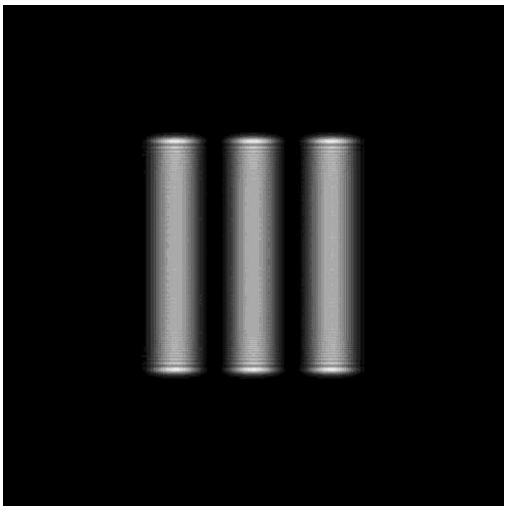
512 × 512



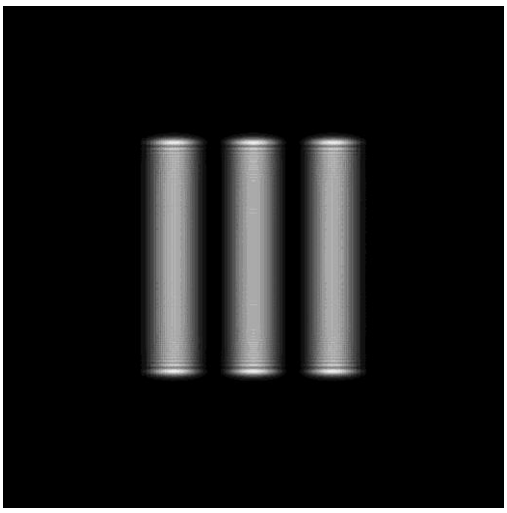
1024 × 1024



1536 × 1536



2048 × 2048



4096 × 4096

Abbildung 5.9 Erhöhung der Auflösung

Wie erkennbar ist, wird erst bei einer Auflösung von 2048 × 2048 der Bereich der Unterabtastung eindeutig verlassen. Da dies bereits eine hohe Auflösung ist, soll nun versucht werden eine Näherungsformel für den Stützstellenabstand bzw. –zahl anzugeben. Dafür ist ein kurzer Einblick in die Entstehung der Unterabtastung notwendig.
Aus Kapitel 1 ist bekannt, dass sich eine Punktquelle in Form einer Kugelwelle ausbreitet. Die einzelnen Kugeln stellen dabei eine Wellenfront mit konstanter Phase dar. Wie in Abbildung 5.10 dargestellt, wird aber nicht ein elektrisches Feld auf einer Kugeloberfläche, sondern auf einer ebenen Fläche errechnet.

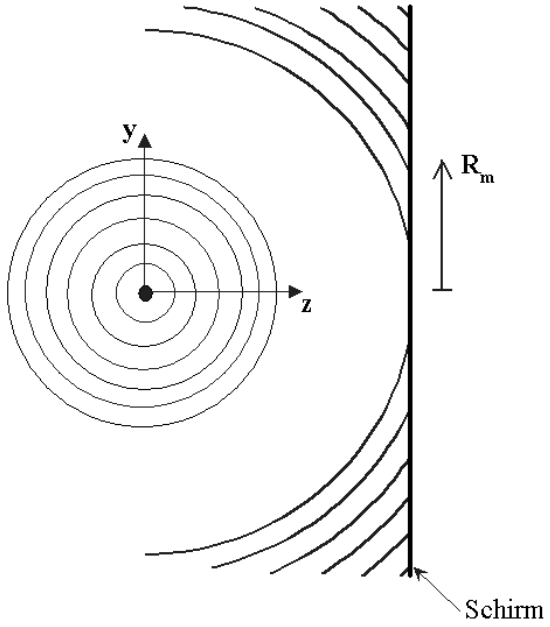


Abbildung 5.10 Ausbreitung einer Kugelwelle

Auf dieser ebenen Fläche wird daher bei Vernachlässigung der Amplitudenabnahme nur der Phasenunterschied von unterschiedlichen Kugelwellen betrachtet. Das entstehende Phasenbild entspricht dem einer Fresnel´schen Zonenplatte. Bei dieser kann man die Newtonschen Ringe beobachten. Die entsprechende Anordnung und weitere Erläuterungen können [HEC98] entnommen werden. Dort gilt für den Radius eines Rings mit maximaler Auslöschung folgende Gleichung:

$$R_m = \sqrt{m\lambda z_0} \qquad \text{m = 1, 2, 3, ...} \qquad (5.3)$$

R_m = Radius des m.-ten Newton´schen Ringes
λ = Wellenlänge
z_0 = Abstand der Punktquelle zum Schirm

Der Abstand zweier Ringe berechnet sich daher zu

$$\Delta R = R_{m+1} - R_m = \sqrt{(m+1)\lambda z_0} - \sqrt{m\lambda z_0} \qquad (5.4)$$

Wird als nächstes die Gleichung (5.3) nach m aufgelöst und in Gleichung (5.4) eingesetzt, ergibt sich

$$\Delta R = \sqrt{\left(\frac{R_m^2}{\lambda z_0} + 1\right)\lambda z_0} - \sqrt{\frac{R_m^2}{\lambda z_0}\lambda z_0}$$
$$= \sqrt{R_m^2 + \lambda z_0} - R_m \tag{5.5}$$

Zusätzlich muss beachtet werden, dass innerhalb eines solchen Streifens das Abtasttheorem erfüllt ist. Daher müssen innerhalb dieser beiden Minima mindestens zwei Stützstellen vorhanden sein.

$$2\Delta x = \sqrt{R_m^2 + \lambda z_0} - R_m \tag{5.6}$$

Wird für Δx noch das Verhältnis aus der Breite des CCD-Chips zur Stützstellenanzahl eingesetzt ($\Delta x = \frac{B}{NX}$), ergibt sich eine Gleichung für die mindestens notwendige Stützstellenzahl in x-Richtung.

$$NX = \frac{2B}{\sqrt{R_m^2 + \lambda z_0} - R_m} \tag{5.7}$$

Dabei stellt R_m den Radius des Newton´schen Ringes dar, welcher gerade noch richtig auf dem CCD-Chip abgebildet wird. Die weiter außen liegenden Ringe werden dann aufgrund der Unterabtastung nicht mehr korrekt von der Kamera wiedergegeben.
Für das in Abbildung 5.8 dargestellte Beispiel errechnet sich eine notwendige Stützstellenzahl von 906 für das erste (B = 0,01m; R_m = ½B; λ = 632nm; z_0 = 0,35m) und 423 für das zweite Kameraelement (B = 0,01m; R_m = B; λ = 632nm; z_0 = 1,5m). Die höchste berechnete Stützstellenzahl von den einzelnen Elementen ist dann für die gesamte Kamera anzuwenden (hier 906). Zu beachten ist, dass für den Ringradius beim ersten Kameraelement aufgrund der Position der Punktquellen (in der Bildmitte) nur die halbe Feldbreite eingesetzt wird. Bei der Berechnung des zweiten Elements ist aber davon auszugehen, dass alle Punkte des ersten Elements Ausgangsort einer Kugelwelle sind. Deswegen müssen auch die Punkte am Rand des elektrischen Feldes mitberücksichtigt und die volle Feldbreite für den Radius eingesetzt werden.
Es ist aber ausdrücklich darauf hinzuweisen, dass diese Gleichung zu niedrige Werte liefert. Dies wird aus einem Vergleich der 906 Stützstellen aus der Gleichung (5.7) mit den in Wirklichkeit benötigten 1536 Stützstellen (Abbildung 5.9) deutlich.

Nachdem mit dieser Testrechnung die richtige Implementierung und die Funktionsfähigkeit der Konvexlinse bewiesen wurde, muss noch die Zylinderlinse untersucht werden. Die entsprechenden Ergebnisse liefern jedoch keine neuen Erkenntnisse. Aus diesem Grund soll an dieser Stelle nur erwähnt werden, dass die vorhandene Implementierung — unter Berücksichtigung der Mindestanzahl von Stützstellen — funktionsfähig ist. Die entsprechenden Testrechnungen können der beiliegenden CD entnommen werden.

Die nächste Testrechnung behandelt die Verschiebungsfunktion (Shifting). Hierzu wird als Eingangsfeld eine ebene Welle (λ = 632nm) betrachtet. Diese trifft auf eine Rechteckapertur mit den Abmessungen von 2×2mm. Anschließend befindet sich mit einem Abstand von 1m ein Schirm mit 2×2cm. Das nicht verschobene Beugungsbild ist in Abbildung 5.11 zu sehen. Am äußeren Rand ist in diesem Fall bereits der Aliasing-Effekt zu erkennen.

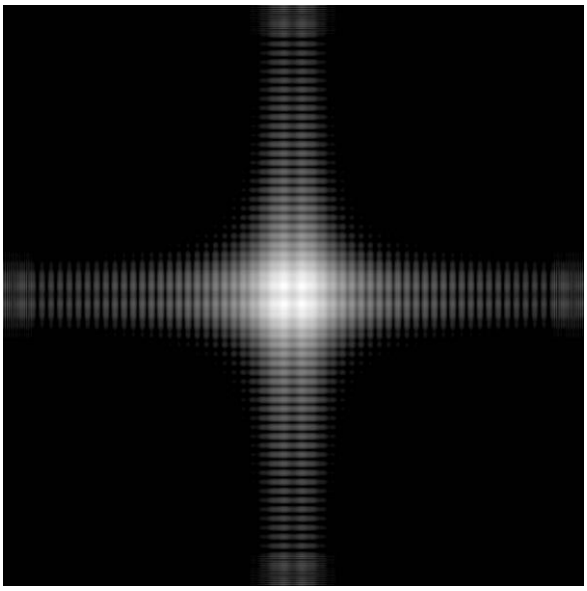


Abbildung 5.11 Beugungsbild der unverschobenen Rechteckapertur

Wird bei der darauffolgenden Rechnung der Schirm um 6cm in die positive *x*- (nach rechts) und 5mm in die positive *y*-Richtung (nach oben) verschoben, sollten in den unteren Quadranten nur die Intensitätsmaxima höherer Ordnung beobachtet werden. Es ergibt sich jedoch Abbildung 5.12.

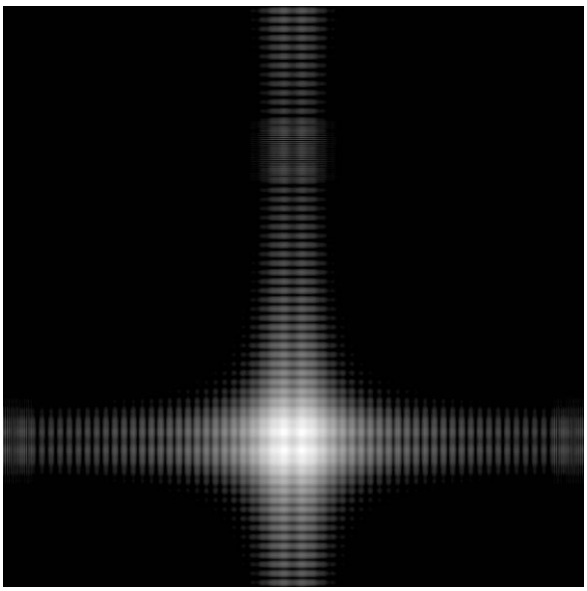


Abbildung 5.12 Verschobenes Beugungsbild

In diesem Bild ist das 0.-, 1.-, 2.-, etc. Intensitätsmaxima zu erkennen. Dieses Phänomen ist durch die diskrete Fouriertransformation zu erklären. In Kapitel 2 wurde beschrieben, dass auf das Eingangsfeld eine Fouriertransformation angewendet wird. Zu beachten ist hierbei, dass die ideale mathematische Transformation, wie sie in der Gleichung dargestellt wird, nicht nur über eine begrenzte Fläche erfolgt, sondern über die von -∞ bis +∞ ausgedehnte ebene Fläche. Bei der diskreten FFT über eine feste Anzahl von Stützstellen wird angenommen, dass die diskreten Werte über die Feldgrenzen hinaus periodisch fortsetzet werden. D.h. mathematisch liegt in der Ebene des Einfallfeldes nicht nur ein Feld vor. Vielmehr existieren unendlich viele Felder die direkt aneinandergereiht sind. Wird bei einer Berechnung ein Kameraelement genau um die Breite und Höhe des eigentlichen Feldes verschoben, so gelangt man zum ursprünglichen Feld. Es kann folglich mit dieser Funktion keine zusätzliche Information gewonnen werden. Aus diesem Grund ist es nicht möglich die Verschiebungsfunktion sinnvoll einzusetzen. Trotzdem sollte diese Methode nicht aus dem Programm genommen werden, weil sie für zukünftige Programmerweiterungen noch eventuell verwendet werden kann.

Die vierte und letzte Testrechnung die in diesem Kapitel vorgestellt wird, ist die Berechnung einer ersten Anordnung von mehreren Kameraelementen. Bei dieser Kombination von Elementen handelt es sich um eine 4f-Anordnung mit zwei gleichen Konvexlinsen. Der Aufbau der Kamera ist Abbildung 5.13 zu entnehmen.

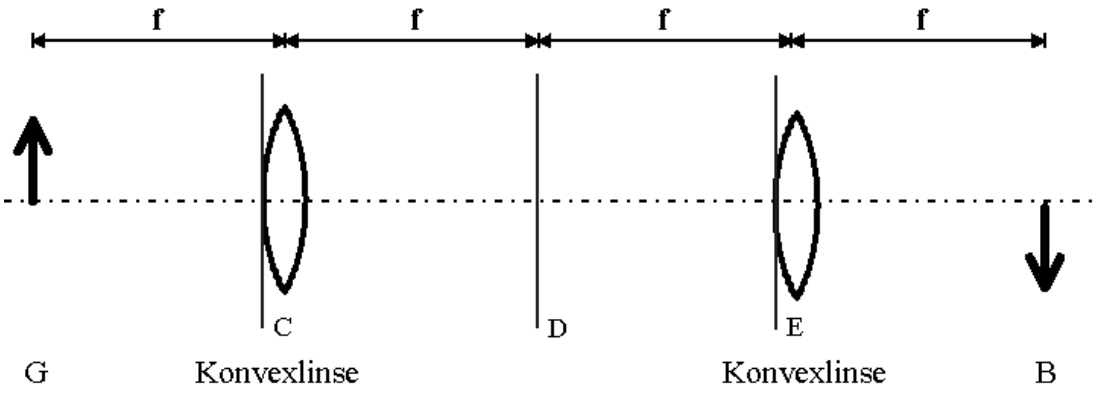


Abbildung 5.13 4f-Anordnung

Als Gegenstand wird ein einfaches Quadrat verwendet. Dieses Objekt ist in Abbildung 5.14 dargestellt. Es wird dadurch erzeugt, dass alle Werte der beiden E-Felder zu null und nur die Realteile im Bereich des Gegenstandes zu eins gesetzt werden. Die genauen Kameradaten, wie z.B. der Brechungsindex der Linsen, die Brennweite, usw. kann der Datei `Kameradat.dat` auf der CD entnommen werden (CD:\ Rechnungen\ Kamera_1\ Approximation\ Apertur_0_1_2_3_Max).

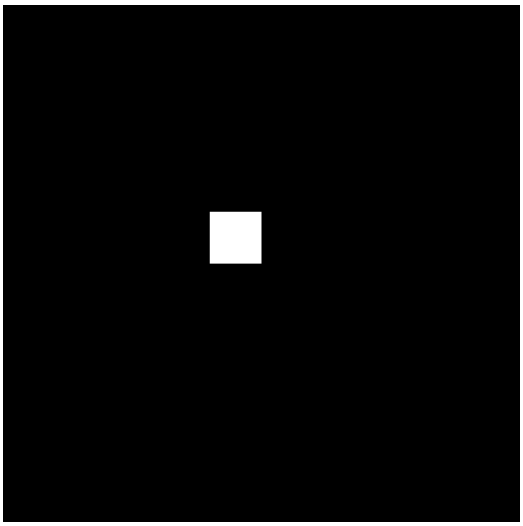


Abbildung 5.14 Betrachteter Gegenstand (G)

Die Eingabedatei wurde so angelegt, dass zur Kontrolle und Anschauung im Verlauf der Kamera vier Bilder berechnet werden. Diese sind:

1. direkt vor der ersten Konvexlinse (Ort C / Abbildung 5.15)
2. im Brennpunkt zwischen den Linsen (Ort D / Abbildung 5.16)
3. direkt vor der zweiten Konvexlinse (Ort E / Abbildung 5.17)
4. im Brennpunkt hinter der zweiten Linse (Ort B / Abbildung 5.18)

Die Intensitätsverteilung wird in diesem Fall mit Hilfe der Approximation errechnet.

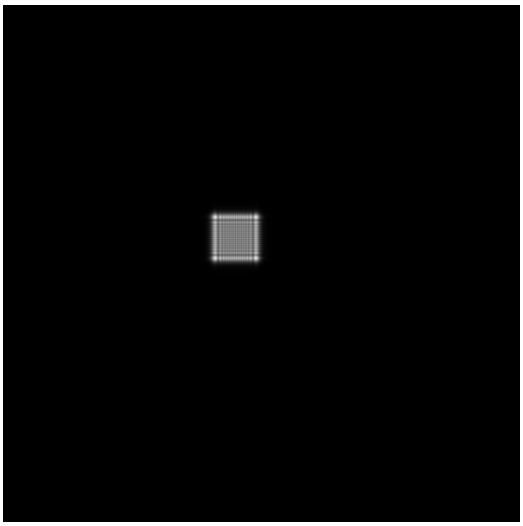


Abbildung 5.15 Bild vor der ersten Linse (C)



Abbildung 5.16 Bild im Brennpunkt beider Linsen (D)

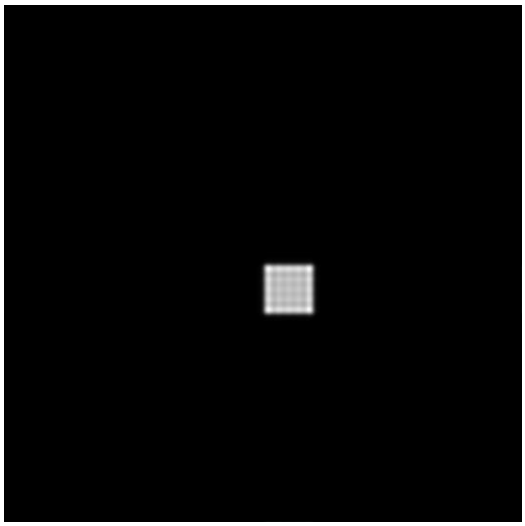


Abbildung 5.17 Bild vor der zweiten Linse (E)

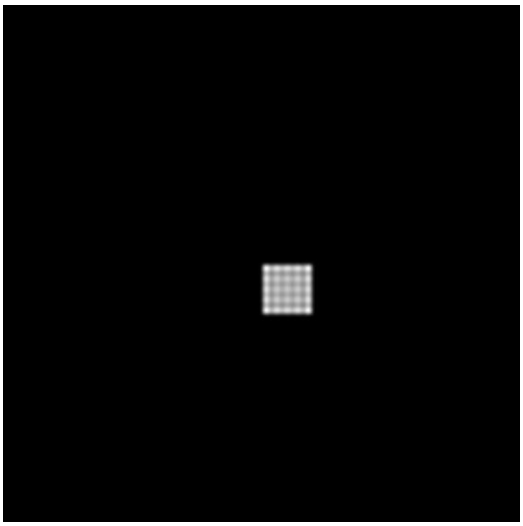


Abbildung 5.18 Beobachtetes Bild (B)

Die Abbildungen 5.15 bis 5.18 liefern das erwartete Ergebnis. Als Bild ist der umgekehrte und gleichgroße Gegenstand zu sehen. Zu beachten ist, dass in Abbildung 5.16 nichts zu erkennen ist, weil sich alle Gegenstandsstrahlen nach der Strahlentheorie im Brennpunkt treffen. Erst wenn dieses Bild vergrößert und nachbearbeitet wird, ist die Detailstruktur sichtbar. Für diese Nachbearbeitung werden die Farben mit einer Gamma-Korrektur verändert. Bei dieser Korrektur werden niedrige Schwarzwerte angehoben. Das Ergebnis ist in Abbildung 5.18 dargestellt. Es ist eindeutig die Fouriertransformierte des Gegenstandes (Rechteck) zu erkennen. Dies ist das Ergebnis nach Wellentheorie. Näheres zu dieser Eigenschaft der 4f-Anordnung, eine optische Fouriertransformation durchzuführen, kann in [HEC98] nachgelesen werden.

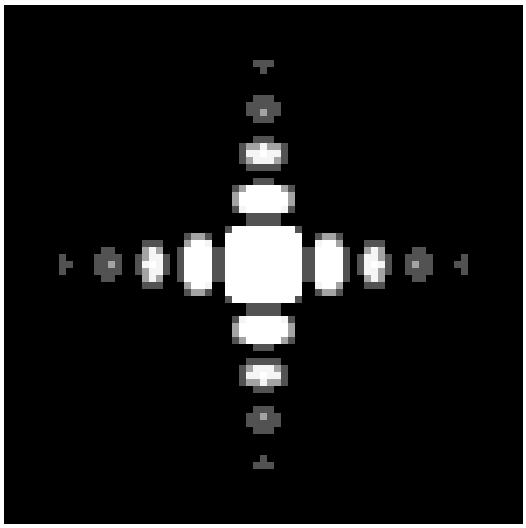


Abbildung 5.19 Vergrößerung des Brennpunktes (D)

Der Test wird im weiteren dadurch verfeinert, dass bei den nächsten Berechnungen im Brennpunkt zwischen den Linsen eine Apertur eingefügt wird. Aufgabe dieser Apertur ist es bestimmte Anteile des Frequenzbereichs herauszufiltern. Dabei ist die Größe der Blende so gestaltet, dass zuerst die Maxima 4. , dann die 3., 2. und schließlich die Maxima 1. Ordnung die Apertur nicht mehr passieren können. Die verschiedenen Bilder des Gegenstandes (B) können in den weiteren Abbildungen betrachtet werden.

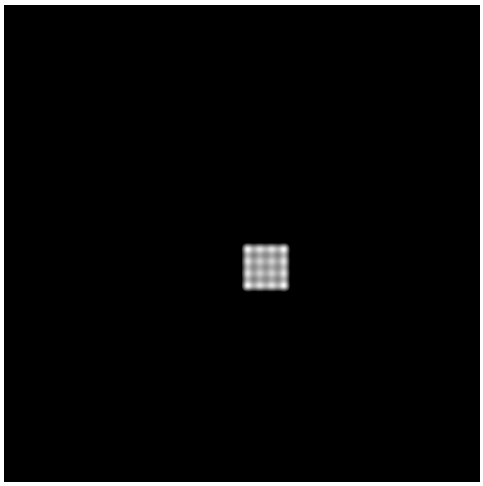


Abbildung 5.20 4.Maximum ist abgeschnitten

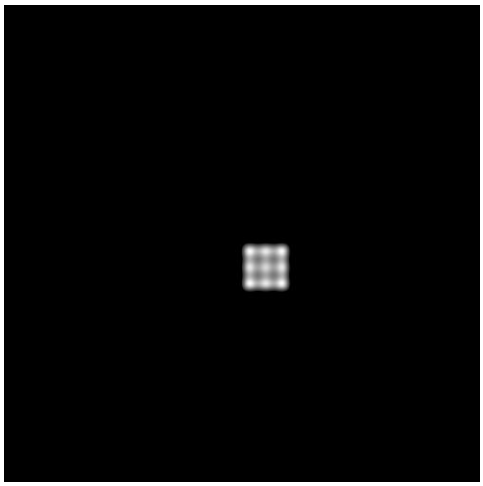


Abbildung 5.21 3.Maximum ist abgeschnitten

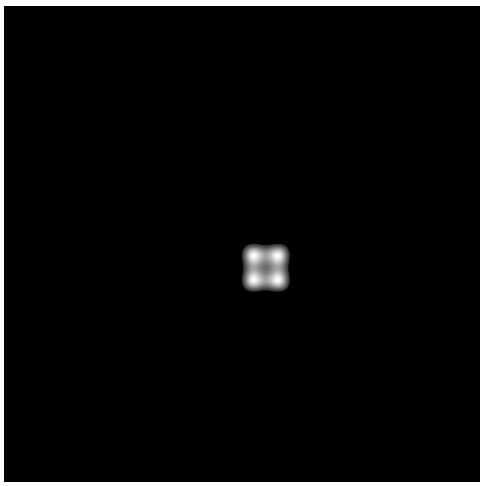


Abbildung 5.22 2.Maximum ist abgeschnitten

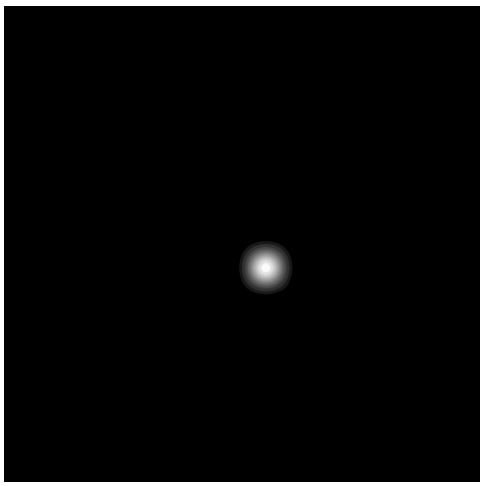


Abbildung 5.23 1.Maximum ist abgeschnitten

Diese Berechnungen unterstreichen, dass die Berechnungsmethode den wahren Verlauf sehr gut simuliert. Nachteilig ist aber, dass die Auflösung sehr fein gewählt werden muss, um ein gutes Ergebnis zu erhalten. Bei einer hohen Stützstellenzahl steigt der Speicherplatzbedarf überproportional an. Als Beispiel sei erwähnt, das ein quadratisches Feld mit 4096 Punkten 256MB Arbeitsspeicher belegt. Während der Berechnung einer Apertur (ohne Transmissionsfunktion) werden damit insgesamt 768MB benötigt. Reicht der Arbeitsspeicher nicht mehr aus, so beginnt das Betriebssystem den Arbeitsspeicher auf die Festplatte auszulagern. Dadurch wird die Berechnung erheblich verlangsamt.

6 Ausblick

Inhalt dieses Kapitels ist es erste Lösungsmöglichkeiten für die vorhandenen Probleme zu finden. Dadurch soll versucht werden einen Ausblick auf die zukünftige Arbeit zu geben.

Hauptproblem bei der Simulation der Kamera ist die extrem hohe Anzahl von Stützstellen bei der späteren Anwendung. Als Beispiel soll ein Spalt mit 200µm Breite betrachtet werden. Soll hinter diesem Spalt ein Bild mit akzeptabler Bildqualität errechnet werden, ist es notwendig, dass ca. 1000 Punkte innerhalb des Spaltes vorhanden sind. Wird weiter angenommen, dass die Kamera eine maximale Ausdehnung vom 0,01m besitzt, so ergibt sich die Anzahl der Stützstellen in Breitenrichtung des Spaltes zu

$$\frac{0{,}01m}{0{,}002m}1000\text{Punkte} = 50.000\text{Punkte}.$$

Ein quadratisches Feld dieser Größe belegt damit 40GB Arbeitsspeicher. Für drei Felder, die für eine vollständige Berechnung eines Elementes notwendig sind, sind es insgesamt 120GB. Eine Reduzierung kann dabei nur durch eine Modifikation des Programms erfolgen. Hierbei wird dann immer nur ein elektrisches Feld im Arbeitsspeicher abgelegt und bearbeitet. Deswegen werden dann nur 80GB benötigt.
Bei dieser Betrachtung wurde der temporäre Arbeitsspeicher, der für die Durchführung der FFT benötigt wird [FRI99] noch nicht berücksichtigt. Dieser muss aber zu den oben angegebenen Zahlen noch addiert werden.

Aufgrund dieses extremen Speicherbedarfs muss die Frage nach einer weiteren Berechnungsmöglichkeit gestellt werden. Hierfür existieren zwei Varianten.
Ausgangspunkt für die erste Möglichkeit ist die Zweite Rayleigh-Sommerfeld-Formel aus Gleichung (2.38). Sie lässt sich schreiben als

$$\underline{\mathbf{E}}(\xi,\eta,z_0) = \int_{-\infty}^{\infty}\int \underline{\mathrm{E}}_{\Sigma}(x,y,z=0)\underline{\mathrm{h}}(x-\xi,y-\eta,z_0)\,d\xi\,d\eta \tag{6.1}$$

$$\text{mit}\quad \underline{\mathrm{h}}(x,y,\mathrm{z}_0) = \frac{\exp(\mathrm{jkz}_0)}{\mathrm{j}\lambda\mathrm{z}_0}\exp\left(\frac{\mathrm{jk}}{2\mathrm{z}_0}\left(x^2+y^2\right)\right)$$

Durch eine Äquivalenzumformung ergibt sich

$$\underline{\mathbf{E}}(\xi,\eta,z_0) = \frac{\exp(jkz_0)}{j\lambda z_0}\exp\left(j\frac{k}{2z_0}\left(x^2+y^2\right)\right) \int_{-\infty}^{\infty}\int\left\{\underline{\mathrm{E}}_{\Sigma}(x,y,z=0)\exp\left(j\frac{k}{2z_0}\left(\xi^2+\eta^2\right)\right)\right\}\exp\left(-j\frac{2\pi}{\lambda z_0}\left(x\xi+y\eta\right)\right)d\xi\,d\eta \tag{6.2}$$

In dieser Form ist leicht zu erkennen, dass es sich bei dem Integral um eine Fouriertransformation handelt. Die Gleichung kann damit weiter vereinfacht werden. Damit ist die Formel der ersten Berechnungsvariante gegeben zu

$$\underline{\mathbf{E}}(\xi,\eta,z_0)=\frac{\exp(jkz_0)}{j\lambda z_0}\exp\left(j\frac{k}{2z_0}\left(x^2+y^2\right)\right)\Im\left\{\underline{\mathbf{E}}_{\Sigma}(x,y,z=0)\exp\left(j\frac{k}{2z_0}\left(\xi^2+\eta^2\right)\right)\right\} \tag{6.3}$$

Der Vorteil dieser Berechnungsart ist neben der Einsparung von zwei Fouriertransformationen, dass kein Speicherplatz für eine Übertragungsfunktion benötigt wird. Trotzdem beträgt der gesamte Speicherplatzbedarf für das oben behandelte Beispiel bei Modifizierung des Programms 40GB.

Eine zweite Variante zur Berechnung des elektrischen Feldes stellt die direkte Berechnung des Beugungsintegrals dar. Dieses Integral wurde bereits in der Gleichung (2.34) vorgestellt. Es wird hier der Vollständigkeit halber nochmals aufgeführt werden.

$$\underline{\mathbf{E}}(P_0)=\frac{1}{\mathrm{j}\lambda}\iint_{\Sigma}\underline{\mathbf{E}}(P_1)\frac{\exp(\mathrm{j}kr_{01})}{r_{01}}\psi ds \tag{6.4}$$

$$\text{mit}\quad \psi=\begin{cases}\frac{1}{2}\left(1+(\mathbf{n}\otimes\mathbf{r}_{01})\right)\\ \mathbf{n}\otimes\mathbf{r}_{01}\\ 1\end{cases}$$

Die Implementierung dieser Formel besitzt entscheidende Vorteile. Der wichtigste Aspekt ist, dass die Lage der Punkte, an denen die elektrische Feldstärke errechnet wird, frei gewählt werden kann. D.h. auf jedem Kameraelement kann die Stützstellenanzahl und deren Abstand frei festgesetzt oder nach bestimmten Kriterien gewählt werden. Außerdem kann eine echte Feldverschiebung oder sogar eine Scheimpflug-Anordnung abgebildet werden.
Aufgrund der freien Wahl der Auflösung auf jedem Element genügt es, ein Feld mit einer Größe von ca. 4000×4000 Punkten zu berechnen. Diese Zahl an Stützstellen ist vermutlich aufgrund der Gefahr der möglichen Unterabtastung notwendig. Bei dieser vergleichsweise geringen Feldgröße beträgt der Speicherplatzbedarf für ein Feld nur 256MB. Aus diesem Grund fällt auch der Nachteil nicht gravierend aus, dass das neue und alte Feld gleichzeitig im Speicher vorliegen müssen.
Wenn aber trotz allem höhere Auflösungen berechnet werden sollen, ist es möglich eine schnelle Berechnung zu gewährleisten. Dies kann mit Hilfe eines Netzwerkes erfolgen. Hierbei verwaltet ein Rechner die vollständige Berechnung. Er stellt jedem Rechner im Netzwerk das Eingangsfeld zur Verfügung und teilt diesen Rechnern mit, welche Teile des neuen Feldes sie zu berechnen haben. Nachdem diese PC´s ihre Teilaufgabe beendet haben, setzt der verantwortliche Rechner alle Ergebnisse zu einem Feld zusammen und startet die nächste Berechnung.
Als wesentlicher Nachteil der Gleichung (6.4) ist jedoch die extrem lange Rechenzeit zu nennen. Die Komplexität dieser Gleichung ist von quadratischer Ordnung (N^2), während die derzeitige Implementierung die Komplexität $N\log N$ besitzt.

Ein entscheidender Vorteil ist jedoch die Effektivität bei der Berechung von dicken Linsen. Bei einer Berechung mit der derzeitigen Implementierung ist für jede Höhenlinie einer Linse eine Übertragungsfunktion zu berechnen (Abbildung 6.1). Bei Berechnung dieser Linsenart nach Gleichung (6.4) müssen aber nur in einem weiteren Feld die räumlichen Koordinaten (x, y, z) jedes Punktes abgelegt werden, damit die Linse vollständig berechnet werden kann.

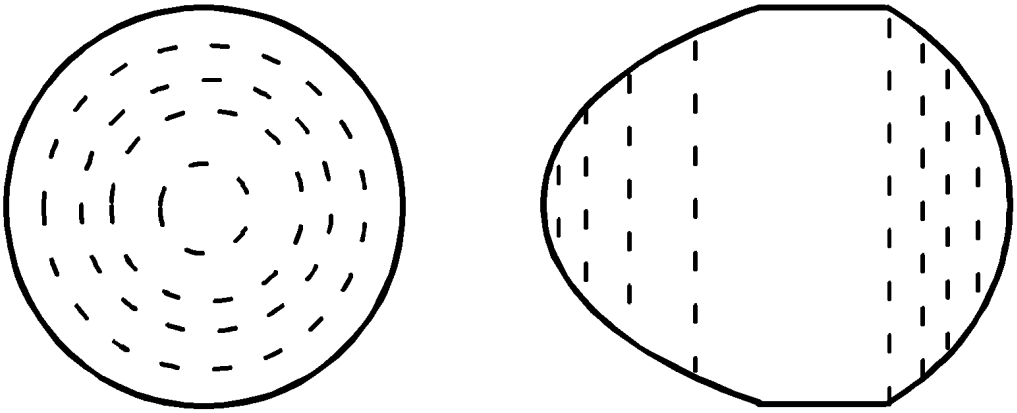


Abbildung 6.1 Darstellung der Höhenlinien zur Berechnung der Übertragungsfunktionen

Weitere Verbesserungsmöglichkeiten betreffen direkt die Klassenstruktur und damit die Implementierung des Programms. Zur Zeit sind die Pläne zur Berechnung der Fouriertansformation Teil der zu transformierenden Felder. Dies hat bei der späteren Einbindung des Programms in das Gesamtprojekt zur Folge, dass bei einer Berechnung von mehreren Kameras es je nach Implementierung vorkommen kann, dass die Pläne sehr häufig neu berechnet werden müssen. Dies kann vermieden werden, indem die Klasse „Kamera“ die Verantwortung über die Pläne übernimmt. Diese Konstellation hat den Vorteil, das jede Kamera ihren eigenen optimierten Plan besitzt. Diese Implementierung ist daher zu favorisieren, wenn die Anzahl der Stützstellen innerhalb der unterschiedlichen Kameras nicht identisch ist. Ist aber diese Anzahl von Stützstellen von Kamera zu Kamera gleichbleibend, dann hat die aktuelle Implementierung Vorteile. Dies ist damit zu begründen, dass die Pläne für alle Kameras zur Zeit nur einmal erstellt würden.
Schließlich bleibt an dieser Stelle daher nur festzuhalten, dass sich die für die Praxis bessere Variante erst bei einer vollständigen Berechnung einer GPD-Anordnung herausfinden lässt.

Anhang

A Fouriertransformation

Die nachfolgenden Formeln können den meisten mathematischen Nachschlagewerken entnommen werden. Auf eine genauere Erläuterung wird daher an dieser Stelle verzichtet.

Fourierintegral:

Fouriertransformation:

$$G(f_x, f_y) = \Im\{g\} = \int_{-\infty}^{\infty}\int g(x,y)\exp\left(-\mathrm{j}2\pi\left(f_x x + f_y y\right)\right)dxdy$$

Inverse Fouriertransformation:

$$g(x,y) = \Im^{-1}\{G\} = \int_{-\infty}^{\infty}\int G(f_x, f_y)\exp\left(\mathrm{j}2\pi\left(f_x x + f_y y\right)\right)df_x df_y$$

Amplitudenspektrum:

$$A(f_x, f_y) = \sqrt{\left(\mathrm{Re}\{G(f_x, f_y)\}\right)^2 + \left(\mathrm{Im}\{G(f_x, f_y)\}\right)^2}$$

Phasenspektrum:

$$\varphi(f_x, f_y) = \arctan\frac{\mathrm{Im}\{G(f_x, f_y)\}}{\mathrm{Re}\{G(f_x, f_y)\}}$$

Linearitätssatz:

$$\Im\{\alpha g + \beta h\} = \alpha\Im\{g\} + \beta\Im\{h\}$$

Faltung:

$$\Im\left\{\int_{-\infty}^{\infty}\int g(\xi,\eta)h(x-\xi, y-\eta)\right\} = G(f_x, f_y)H(f_x, f_y)$$

Verschiebungssatz:

$$\Im\{g(x-\mathrm{a}, y-\mathrm{b}\} = G(f_x, f_y)\exp\left(-\mathrm{j}2\pi\left(f_x\mathrm{a} + f_y\mathrm{b}\right)\right)$$

Ähnlichkeitssatz:

$$\Im\{g(\mathrm{a}x, \mathrm{b}y)\} = \frac{1}{|\mathrm{ab}|}G\left(\frac{f_x}{\mathrm{a}}, \frac{f_y}{\mathrm{b}}\right)$$

An dieser Stelle sollen noch drei wichtige analytisch durchgeführte Fouriertransformationen aufgeführt werden, weil sie im Rahmen dieser Arbeit verwendung finden. Die Formeln sind [GOO68] entnommen worden.

Funktion	Fouriertransformierte
$\exp\left(-\pi\left(\mathrm{a}^2x^2+\mathrm{b}^2y^2\right)\right)$	$\frac{1}{\lvert\mathrm{ab}\rvert}\exp\left(-\pi\left(\frac{f_x^2}{\mathrm{a}^2}+\frac{f_y^2}{\mathrm{b}^2}\right)\right)$
$\exp\left(\mathrm{j}\pi\left(\mathrm{a}^2x^2+\mathrm{b}^2y^2\right)\right)$	$\frac{\mathrm{j}}{\lvert\mathrm{ab}\rvert}\exp\left(-\mathrm{j}\pi\left(\frac{f_x^2}{\mathrm{a}^2}+\frac{f_y^2}{\mathrm{b}^2}\right)\right)$
$\mathrm{rect}(\mathrm{a}x)\,\mathrm{rect}(\mathrm{b}y)$	$\frac{1}{\lvert\mathrm{ab}\rvert}\mathrm{sinc}\left(\frac{f_x}{\mathrm{a}}\right)\mathrm{sinc}\left(\frac{f_y}{\mathrm{b}}\right)$

mit $\quad \mathrm{rect}(x)=\begin{cases}1 & \lvert x\rvert<\frac{1}{2}\\ \frac{1}{2} & \lvert x\rvert=\frac{1}{2}\\ 0 & \lvert x\rvert>\frac{1}{2}\end{cases}$

und $\quad \mathrm{sinc}(x)=\frac{\sin(\pi x)}{\pi x}$

B Quellcode

Der Quellcode ist in folgender Reihenfolge aufgelistet. Dabei ist zuerst die Header-Datei und im Anschluß daran die CPP-Datei aufgeführt.

1. main()
2. Camera
3. Plan
4. Field
5. ElectricalField
6. Transferfunction
7. Transmissionfunction
8. CameraElement
9. Screen
10. OpticalElement
11. Apertur
12. Rectangle
13. Circle
14. Lens
15. ConvexConcave
16. Cylinder
17. TypesCamera.txt
18. Camera.dat

```
1   #include "camera.h"
2
3   int main()
4   {
5     // Erzeugung eines Kameraobjektes
6     CCamera Camera;
7
8     // Erzeugung eines Input-Objektes, von dem die Kameradaten eingelesen werden.
9     CInput Input("Camera.dat","TypesCamera.txt");
10    Input.loadLoops("A");       // Laden der Schleifen
11    Input.gotoBlock("B");       // Gehe zu Beginn der eigentlichen Definition der Kamera
12    Camera.read(Input);         // Einlesen der Kameradaten
13    Input.Reset();              // Setzen der Funktionswerte fuer die SchleifenP
14
15    do {
16      Camera.syncChange();      // Synchronisierung aller change-Variablen
17      Camera.set();             // Setzen der Statusvariablen
18      Camera.calc();            // Einlesen / Anfordern der Felder und Berechnung
19      Camera.writeCamera();     // Ausgabe der aktuelen Kameradaten
20      Camera.resetAllChange();  // Zuruecksetzen aller change-Variablen
21
22      // Input-Objekt veraendert ggf. Werte und setzt die entsprechende change-Variable
23    } while(Input.nextRun());
24
25
26    return 0;
27  }
```

```
1   // Camera.h: Schnittstelle für die Klasse CCamera.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if
    !defined(AFX_CAMERA_H__2EBBF980_A917_11D4_9702_FA70CF88542D__INCLUDE
    D_)
6   #define
    AFX_CAMERA_H__2EBBF980_A917_11D4_9702_FA70CF88542D__INCLUDED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #ifdef _DEBUG
13  #pragma warning(disable: 4786)
14  #endif
15
16  #ifdef _DEBUG
17  #pragma warning(disable: 4275)
18  #endif
19
20  #include <iostream.h>
21  #include <string.h>
22
23  #include "..\Constants\constant.h"
24  #include "..\VectorManager\ArrayManager.h"
25  #include "..\PointerArray\PointerArray.h"
26  #include "..\Read\Input.h"
27  #include "..\Read\Read.h"
28
29  #include "TransferFunction.h"
30  #include "ElectricalField.h"
31  #include "Screen.h"
32  #include "Circle.h"
33  #include "ConvexConcave.h"
34  #include "Rectangle.h"
35  #include "Cylinder.h"
36  #include "Lens.h"
37  #include "Aperture.h"
38  #include "OpticalElement.h"
39  #include "CameraElement.h"
40
41  using namespace std;
42
43  ////////////////////////////////
44  ////   Klassendeklaration   ////
45  ////////////////////////////////
46
47  class CCamera
```

```
48  {
49  public:
50    CCamera();
51    virtual ~CCamera();
52
53    // Change-Funktionen zur Steuerung des automatischen Aenderns von
      Objekteigenschaften
54    virtual void syncChange(void);
55    virtual bool set(void);
56    virtual void resetAllChange(void);
57    virtual void setAllChange(void);
58
59    // Berechnung und Ausgabe der EFelder
60    virtual void calc(void);
61    // Einlesen aller Daten
62    virtual void read(CInput &Input);
63    // Ausgabe der Objektdaten
64    void writeCamera(void);
65
66  protected:
67    void init(void);
68
69    CElectricalField EX;        // Elektrisches Feld mit Polarisierung in x-Richtung
70    CElectricalField EY;        // Elektrisches Feld mit Polarisierung in y-Richtung
71
72    int NCamEle;                // Anzahl der Kameraelemente
73    double CameraLambda;    // Wellenlaenge innerhalb der Kamera
74    double MAXWidthX;          // maximale raeumliche Ausdehnung der Kamera in x-Richtung
75    double MAXHeightY;          // maximale raeumliche Ausdehung der Kamera in y-Richtung
76
77    COut KameraOut;             // Ausgabeobjekt fuer die Kameradaten
78
79    CPointerArray<CCameraElement> CamEle;   // Feld mit erzeugten Kameraobjekten
80
81    // Ausgabevariablen
82    struct {
83     bool WriteCamera; // Angabe ob die tatsaechlich verwendeten Objektdaten ausgegeben
                           werden sollen
84    } output;
85
86    // Stausvariablen
87    struct {
88      bool ERead;      // Angabe, ob das E-Feld bereits gelesen wurde
89      bool EFillUp;    // Angabe, ob das E-Feld berets aufgefüllt wurde
90    } status;
91
92    // Change-Variablen
93    bool change;    // Angabe, ob irgendwelche Eigenschaften der Kameraelemente
94                    // oder des elektrischen Feldes veraendert wurde
95    bool changeE;       // Angabe, ob sich das E-Feld geändert hat
```

```
96      bool changeEle;    // Angabe, ob ein optisches Element geändert wurde
97  };
98  #endif //
    !defined(AFX_CAMERA_H__2EBBF980_A917_11D4_9702_FA70CF88542D__INCLUDE
    D_)
```

```
1   // Camera.cpp: Implementierung der Klasse CCamera.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "Camera.h"
6
7   //////////////////////////////////////////////////////////////////////
8   // Konstruktion/Destruktion
9   //////////////////////////////////////////////////////////////////////
10
11  CCamera::CCamera()
12  {
13    init();
14  }
15
16
17  CCamera::~CCamera()
18  {
19
20  }
21
22
23  void CCamera::init(void)
24  {
25    NCamEle = 0;
26    MAXHeightY = MAXWidthX = 0.0;
27    CameraLambda=MAXWidthX=MAXHeightY=0;
28
29    output.WriteCamera = false;
30    // Definition des Outobjektes zum Schreiben der Kameradaten in eine Datei
31    KameraOut.open("Kamaeradat","wt");
32    status.ERead = status.EFillUp = false;
33    setAllChange();
34  }
35
36
37  // Setzen aller change-Variablen
38  void CCamera::setAllChange(void)
39  {
40    // Setzen der Kamera "eigenen" Variablen
41    change=changeE=changeEle=true;
42    // Setzen der change-Variablen aller Objekte
43    for(int i=0;i<CamEle.Size();i++) CamEle[i]->setAllChange();
44  }
45
46
47  // Zuruecksetzen aller change-Variablen
48  void CCamera::resetAllChange(void)
49  {
50    // "eigene" Variablen
51    change = changeE = changeEle = false;
52
53    // Zuruecksetzen der change-Variablen aller Objekte
54    for(int i=0;i<CamEle.Size();i++) CamEle[i]->resetAllChange();
```

```
55  }
56
57
58  // Synchronisierung der change-Variablen untereinander
59  void CCamera::syncChange(void)
60  {
61    // Alle untergeordneten Componenten werden angewiesen Aenderungen bekannt zu geben
62    for(int i=0; i<CamEle.Size();i++) CamEle[i]->syncChange();
63
64    // Ueberpruefung, ob bei den Kameraelementen eine neue Groesse zugewiesen worden
        ist.
65    // Wenn ja muss die entsprechende change-Variable der Kamera gesetzt werden und
66    // die neue maximale raeumliche Ausdehnung der Kamera, die Aufloesung des letzten
        Schirms
67    // und die Groesse des ersten Elments neu ermittelt werden.
68    for(i=0; i<CamEle.Size();i++) {
69      if(CamEle[i]->getChangeSize()) {
70        changeEle = true;
71      }
72    }
73    // Wenn nach einer spaeteren Integrierung des gesamten Programms in das gesamte
        Projekt
74    // die Kamera von aussen veraendert wurde, muessen alle Vorgaenge zur Berechnung
75    // neu ausgerführt werden.
76    if (change == true)   { changeE = changeEle = true; }
77  }
78
79  // Einlesen aller notwedigen Daten
80  void CCamera::read(CInput &Input)
81  {
82    // Eigenschaften der Camera-Elemente einlesen
83    Input.readFunction(&NCamEle);
84    output.WriteCamera = (bool)Input.readType("bool");
85    Input.setChange(&changeE);
86    Input.readFunction(&CameraLambda);
87
88    // Einlesen der einzelnen Objektdaten
89    for(int i=0; i<NCamEle; i++) {
90
91      Input.gotoNext();
92      // Zunaechst wird die Art des optischen Elementes eingelesen und ein solches
93      // Objekt erzeugt.
94      switch((char)Input.readType("artdesoptischenelementes")) {
95      case 'R':  CamEle.Grow(new CRectangle);
96                 break;
97      case 'C':  CamEle.Grow(new CCircle);
98                 break;
99      case 'Y':  CamEle.Grow(new CCylinder);
100                break;
101     case 'O':  CamEle.Grow(new CConvexConcave);
102                break;
103     case 'S':  CamEle.Grow(new CScreen);
104                break;
105     default: ::cerr << "CCamera::readCameraData(): Der geforderte Typ existiert nicht.";
106     }
107     // Das neu erzeugte Objekt erhaelt die seiner Stelle entsprechenden Elementnummer
```

```
108       CamEle[i]->setNo(i);
109       // Das Kameraelment liest die ih betreffenden Daten selbstaendig ein.
110       CamEle[i]->read(Input);
111     }
112
113     // Setzen aller change-Variablen, da noch nichts berechnet wurde.
114     setAllChange();
115   }
116
117
118   // Berechnungsmethode
119   void CCamera::calc(void)
120   {
121     // lokale Variablen
122     size_t i;
123
124     ////////////////////////////////////////////////////////////////////////
125     // Einlesen / Erzeugung eines Eingangsfeldes, wenn es noch nicht erfolgt ist.
126     FILE *fdX;
127     FILE *fdY;
128     string TestXdat="EingangsfeldX.dat";
129     string TestYdat="EingangsfeldY.dat";
130     int TempNX, TempNY;
131     double TempWidthX, TempHeightY;
132
133     if (status.ERead == false) {
134       // Einlesen eines eingangsfeldes
135   /*  fdX=fopen("EingangsfeldX.dat","r+b");
136       fdY=fopen("EingangsfeldY.dat","r+b");
137
138       if ((fdX==NULL)||(fdY==NULL)) {
139         ::cerr << "Datei zum Einlesen der Felder konnte nicht geöffnet werden";
140         return;
141       }
142       EX.read(TestXdat,fdX,'r+b');
143       EY.read(TestYdat,fdY,'r+b');
144       fclose(fdX);
145       fclose(fdY);
146   */
147       // Erzeugung eines Eingangsfeldes (hier ebene Welle)
148       // Die Art des Eingangsfeldes kann durch Veraenderung der setTestEField-Funktion
149       // einfach erfolgen.
150       // Ermittlung der Aufloesung des letzten Elements (muss immer ein Schirm sein)
151       TempNX = CamEle[NCamEle-1]->getNX();
152       TempNY = CamEle[NCamEle-1]->getNY();
153       TempWidthX = CamEle[NCamEle-1]->getWidthX();
154       TempHeightY = CamEle[NCamEle-1]->getHeightY();
155       // Setzen des Testfeldes mit der Aufloesung des letzten Elements (Schirm)
156       EX.setTestEField(TempNX, TempNY, TempWidthX, TempHeightY, CameraLambda);
157       EY.setTestEField(TempNX, TempNY, TempWidthX, TempHeightY, CameraLambda);
158       // Auffuellen des Testfeldes mit Nullen bis die maximale Groesse der Kamera erreicht ist
159       // Dadurch wird die Groesse des Feldes veraendert!
160       EX.fillUp(MAXWidthX,MAXHeightY);
161       EY.fillUp(MAXWidthX,MAXHeightY);
162       // Da das Eingangsfeld nur mit Nullen aufgefuellt wurde, muss zum Testen des
            Programms
```

```
163     // mit einer ebenen Welle die setTestEField-funktion erneut aufgerufen werden, damit die
164     // ebene Welle die gesamte Breite und Hoehe der Kmera einschliesst.
165     EX.setTestEField(EX.getNX(), EX.getNY(), EX.getWidthXE(), EX.getHeightYE(),
        CameraLambda);
166     EY.setTestEField(EY.getNX(), EY.getNY(), EY.getWidthXE(), EY.getHeightYE(),
        CameraLambda);
167     // Ein spaeter eingelesenes Feld ist zu diesem Zeitpunkt aber noch nicht aufgefuellt
            worden.
168     status.ERead = true;
169     status.EFillUp = false;
170   }
171
172   // Kontrolle ob die eingelesenen Felder dieselben Eigenschaften besitzen
173   if ( !(EX.equalParameters(EY)) ) {
174     ::cerr<<"CCamera::calc(void): Die Felder EX und EY besitzen nicht dieselben
        Eigenschaften !!!";
175   }
176   // Den Feldern wird die Wellenlaenge der Kamera zugewiesen. ACHTUNG!!!
177   // Bei der spaeteren Integration in das Hauptprogramm muesste anstelle des Einlesens und
178   // Zuweisens der Wellenlange ein Aufruf stehen, der von den vorangegangenen
          Rechenschritten ein Feld
179   // mit den Eigenschaften der hier vorliegenden E-Felder und mit der Kamerawellenlaenge
        anfordert.
180   EX.setLambda(CameraLambda);
181   EY.setLambda(CameraLambda);
182
183   // Ueberpruefung ob das E-Feld mindestens die Groesse des ersten Elmentes besitzt.
184   if ( ((CamEle[0]->getWidthX())>EX.getWidthXE()) || ((CamEle[0]-
        >getHeightY())>EX.getHeightYE())) {
185       ::cerr<<"Fehler in CCamera::calc(): Das Eingangsfeld ist kleiner als die Groesse des
        ersten Elementes!";
186       ::cerr<<"Das Feld wird mit Nullen aufgefuellt. Dabei entstehen Rechenfehler!";
187   }
188
189   // Auffuellen der E-Felder mit Nullen auf die gewuenschte Breite und Hoehe
190   if (status.EFillUp == false) {
191     EX.fillUp(MAXWidthX,MAXHeightY);
192     EY.fillUp(MAXWidthX,MAXHeightY);
193     status.EFillUp = true;
194   }
195
196   // Berechnung der einzelnen Objekte
197   for(i=0; i<NCamEle; i++) {
198     CamEle[i]->calc(EX,EY);
199   }
200 }
201
202
203 // Ausgabe der Objektdaten
204 void CCamera::writeCamera(void)
205 {
206   // Ausgabe der eigenen aktuellen Objektdaten
207   if (output.WriteCamera) {
208     KameraOut.open("Kamaeradat","at");
209     KameraOut << "NeueKamera\n";
210     KameraOut << NCamEle;
211     KameraOut << "// Anzahl der Elemente\n";
```

```
212     KameraOut << output.WriteCamera;
213     KameraOut << "// Ausgabe der Kameradaten(JA,NEIN)\n";
214     KameraOut << CameraLambda;
215     KameraOut << "// Wellenlaenge\n";
216     KameraOut << "\n";
217     // Anweisung an die Kameraelemente sich selber auszugeben.
218     for(int i=0; i<NCamEle; i++) {
219       CamEle[i]->writeElement(KameraOut);
220     }
221   }
222 }
223
224
225 // In dieser Funktion werden alle Variationen von gesetzten change-Variablen behandelt
226 // und die entsprechenden Statusvariablen gesetzt bzw. Funktionen durchgefuehrt.
227 bool CCamera::set(void)
228 {
229   size_t i;
230
231   // Die Groesse eines Elementes wurde geaendert.
232   if (changeEle == true) {
233     // Ermittlung der neuen groessten raeumlichen Ausdehnung
234     MAXHeightY = 0;
235     MAXWidthX = 0;
236     for(i=0; i<NCamEle; i++) {
237       if ((CamEle[i]->getHeightY()) >MAXHeightY)  MAXHeightY=(CamEle[i]->getHeightY());
238       if ((CamEle[i]->getWidthX()) >MAXWidthX)  MAXWidthX=(CamEle[i]->getWidthX());
239     }
240   }
241
242   // Das elektrische Feld wurde von aussen veraendert. (Wird erst bei der spaeteren
243   // integration in das Hauptprogramm benoetigt.)
244   if (changeE == true) {
245     for(i=0; i<NCamEle; i++) {
246       CamEle[i]->setChangeHT(true);
247     }
248   }
249
250   // Jedes Kameraobjekt wird angeweisen dasgleiche durchzufuehren.
251   for(i=0; i<NCamEle; i++) {
252     CamEle[i]->set();
253   }
254
255   // Hier werden die Statusvariablen zurueckgesetzt, weil bei Aufruf dieser Funktion
256   // eine Berechnung vollstaendig erfolgt ist und ein neues Eingangsfeld eingelesen
257   // werden muss. (Das alte E-Feld kann nicht mehr verwendet werden, weil es schlielich
258   // von den Kameraelmenten veraendert wurde.)
259   status.ERead = status.EFillUp = false;
260
261   return true;  // Aufruf der set-Funktion ist erfolgt
262 }
```

```
1   // Plan.h: Schnittstelle für die Klasse CPlan.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if !defined(AFX_PLAN_H__58AE4851_AE7C_11D4_8960_0050BF19F9FF__INCLUDED_)
6   #define AFX_PLAN_H__58AE4851_AE7C_11D4_8960_0050BF19F9FF__INCLUDED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #define USE_FFTW_DLL FFTW2dll.dll
13  #include "..\fftw_float\fftw\fftw.h"
14
15  #ifdef _DEBUG
16  #pragma warning(disable: 4786)
17  #endif
18
19  #ifdef _DEBUG
20  #pragma warning(disable: 4275)
21  #endif
22
23  #include <iostream.h>
24
25  #include "..\Constants\constant.h"
26  #include "..\Out\Out.h"
27
28
29
30  class CPlan
31  {
32
33  public:
34    CPlan();
35    virtual ~CPlan() { };
36
37  protected:
38    // Berechnung der Plaene für die Vorwaerts- und Rückwaertstransformation für eine
        vorgegebene Groesse
39    void calcPlan(size_t NY, size_t NX);
40
41    // Plan (Hilfsmittel der FFTW) zur Berechnung einer FFT mit einer festen Groesse
42    // Der Plan muss vor der Verwendung durch die FFT einmal fuer die entsprechende
43    //   Groesse (NX,NY) erstellt werden.
44    static fftwnd_plan PlanForward;  // Plan für die Vorwaertsfouriertransformation
45    static fftwnd_plan PlanBackward; // Plan für die Rueckwärtstransformation
46
47    // Statusvariable gibt an, ob der Plan für die Feldgroesse (für dieses Feld) gueltig ist
48    // gueltig       = true
49    // nicht gueltig = false
50    static bool PlanValid;
51
52    // Initialisierung der Variablen (static-Variablen werden direkt in CPlan.cpp erstmalig
        definiert.
53    void init(void)  { };
```

```
54
55  private:
56    void destroyPlan(void);  // Funktion zur Zerstoerung der beiden Plaene
57
58  };
59  #endif //
    !defined(AFX_PLAN_H__58AE4851_AE7C_11D4_8960_0050BF19F9FF__INCLUDED_)
```

```
1   // Plan.cpp: Implementierung der Klasse CPlan.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "Plan.h"
6
7   //Erste notwendige Initialisierung der static-Variablen
8   fftwnd_plan CPlan::PlanForward  = NULL;
9   fftwnd_plan CPlan::PlanBackward = NULL;
10  bool        CPlan::PlanValid    = false;
11
12
13  CPlan::CPlan()
14  {
15    init();
16  }
17
18
19  // Berechnung der Plaene für die Vorwaerts- und Rueckwaertstransformation fuer eine
      vorgegebene Groesse
20  void CPlan::calcPlan(size_t NY, size_t NX)
21  {
22    // Berechnung der Plaene für die FFT mit einer bestimmten Groesse
23    // Hinweis: Bei einer hohen Aufloesung oder einer groesseren Anzahl (>200) von Bildern,
24    //          die von derselben Kamera berechnet werden, kann es hilfreich sein, als Option
25    //          nicht ESTIMATE sonder MEASURE zu verwenden.
26    //          Hierdurch dauert die Planerstellung zwar ein vielfaches laenger, aber die
27    //          eigentliche Berechnung ist wesentlich kuerzer.
28
29    // Vor der Berechnung der Plaene werden diese sicherheitshalber vollstaendig zerstoert.
30    destroyPlan();
31
32    // Berechnung der Plaene für die Vorwaerts- und Rueckwaertstransformation.
33    CPlan::PlanForward  = fftw2d_create_plan(NY,NX,FFTW_FORWARD,
      FFTW_USE_WISDOM|FFTW_ESTIMATE|FFTW_IN_PLACE);
34    CPlan::PlanBackward =
      fftw2d_create_plan(NY,NX,FFTW_BACKWARD,FFTW_USE_WISDOM|FFTW_ESTIMAT
      E|FFTW_IN_PLACE);
35
36    // Nach Berechnung der Plaene muss die entsprechende Status- (Gueltigkeits-)variable
37    //   auf gueltig (=true) gesetzt werden.
38    CPlan::PlanValid=true;
39  }
40
41
42  // Zerstörung der Pläne
43  void CPlan::destroyPlan()
44  {
45    // Zerstoerung der Plaene für die Vorwaerts- und Rueckwaertstransformation
46    fftwnd_destroy_plan(CPlan::PlanForward);
47    fftwnd_destroy_plan(CPlan::PlanBackward);
48
49    // Nach Zerstoerung der Plaene muss die entsprechende Status- (Gueltigkeits-)variable
50    //   auf nicht gueltig (=false) gesetzt werden.
51    CPlan::PlanValid=false;
52  }
```

```
1   // Field.h: Schnittstelle für die Klasse CField.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if
    !defined(AFX_FIELD_H__E75CA6E0_C007_11D4_9702_A68F8DFD6A71__INCLUDED_
    )
6   #define AFX_FIELD_H__E75CA6E0_C007_11D4_9702_A68F8DFD6A71__INCLUDED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #include "Plan.h"
13  #include "..\VectorManager\ArrayManager.h"
14
15
16  class CField : public CPlan
17  {
18
19  public:
20    CField();
21    virtual ~CField();
22
23    // Funktion zur Ermittlung der Feldgroesse in x- und y- Richtung
24    size_t getNX(void) { return Array.CArrayManager<fftw_complex>::getYSize(); };
25    size_t getNY(void) { return Array.CArrayManager<fftw_complex>::getXSize(); };
26
27    // Anwendung der Vorwaertstransformation auf das entsprechende Feld
28    void Ffft(void);
29
30    // Anwendung der Rueckwaertstransformation auf das entsprechende Feld
31    void Bfft(void);
32
33    // Austausch der 4 Viertel des Feldes, um die numerische Umordnung der FFT
    rueckgaengig zu machen
34    void ChangeField(void);
35
36    // Aenderung der Groesse des Feldes
37    //   Dabei koennen die alten Daten beibehalten werden (save=true)
38    //   Die alten übernommenen Daten (wenn save=true) koennen im neuen Feld
39    //   mit einem Offset versehen werden, der die Daten im neuen Feld "verschiebt"
40    void setSize(size_t NewNY, size_t NewNX, bool save, int OffsetY, int OffsetX);
41
42    // Ausgabe des eigentlichen Feldes
43    virtual void write(COut& Out) { Out << Array; };
44    // Einlesen eines Feldes
45    virtual void read(string& dateiname);
46
47    // Definition des ()-Operator für den Feldzugriff (j,i)
48    // Zu beachten ist, dass in CCameraElement die Elemente wie folgt angesprochen:
49    //       (j,i)=(NY,NX)
50    //       |------>x
51    //       |
52    //       |y
```

```
53    // Im ArrayManager liegen die Daten wie folgt:
54    //       [j][i]=[NX][NY]
55    //       |------>y
56    //       |
57    //       |x
58    const fftw_complex& operator() (size_t Y, size_t X) const { return(Array[Y][X]); };
59    fftw_complex& operator() (size_t Y, size_t X) { return(Array[Y][X]); };
60
61  protected:
62    void init(void)  { };
63
64    // Liefert die Adresse des eigentlichen Feldes
65    // (wird fuer die Anwendung der FFTW benötigt)
66    fftw_complex* getArrayPointer(void) { return Array.getPointer(); };
67
68    // Eigentliches Feld mit den Daten vom Typ fftw_complex
69    CArrayManager<fftw_complex> Array;
70  };
71
72  #endif //
    !defined(AFX_FIELD_H__E75CA6E0_C007_11D4_9702_A68F8DFD6A71__INCLUDED_
    )
```

```
1   // Field.cpp: Implementierung der Klasse CField.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "Field.h"
6
7
8   CField::CField():CPlan()
9   {
10    init();
11  }
12
13
14  CField::~CField()
15  {
16
17  }
18
19
20  // Aenderung der Groesse des Feldes
21  //   Dabei koennen die alten Daten beibehalten werden (save=true)
22  //   Die alten übernommenen Daten (wenn save=true) koennen im neuen Feld
23  //     mir einem Offset versehen werden, der die Daten im neuen Feld "verschiebt"
24  void CField::setSize(size_t NewNY, size_t NewNX, bool save, int OffsetY, int OffsetX)
25  {
26    Array.CArrayManager<fftw_complex>::setSize(NewNY, NewNX, save, OffsetY, OffsetX);
27  }
28
29
30  // Anwendung der Vorwärtstransformation auf das Feld
31  void CField::Ffft()
32  {
33    // Wenn der Plan nicht gueltig (nicht fuer die richtige Groesse berechnet wurde),
34    // dann wird zunaechst ein neuer Plan errechnet.
35    if (PlanValid==false) {
36      calcPlan(getNY(), getNX());
37    }
38    // Anwendung der FFTW
39    // fftwnd_one(Plan gibt die Art der Transformation an, Zeiger auf das Feld, temporaerer
        Speicher);
40    // Hinweis: Der Plan enthält bereits die Informationen zur Feldgroesse
41    fftwnd_one(PlanForward, getArrayPointer(), NULL);
42  }
43
44
45  // Anwendung der Rückwärtstransformation auf das Feld
46  void CField::Bfft()
47  {
48    // Wenn der Plan nicht gueltig (nicht fuer die richtige Groesse berechnet wurde),
49    // dann wird zunaechst ein neuer Plan errechnet.
50    if (PlanValid==false) {
51      calcPlan(getNY(), getNX());
52    }
53    // Anwendung der FFTW
54    // fftwnd_one(Plan gibt die Art der Transformation an, Zeiger auf das Feld, temporaerer
```

```
        Speicher);
55    // Hinweis: Der Plan enthält bereits die Informationen zur Feldgroesse
56    fftwnd_one(PlanBackward, getArrayPointer(), NULL);
57  }
58
59
60  // Einlesen eines Feldes
61  void CField::read(string& dateiname)
62  {
63    Array.read(dateiname);
64  }
65
66
67  // Austausch der 4 Viertel des Feldes, um die numerische Umordnung der FFT
      rueckgaengig zu machen
68  void CField::ChangeField(void)
69  {
70    // Lokale Variablen
71    int i,j;            // Laufvariablen
72    double TempRe;      // temporaerer Realteil
73    double TempIm;      // temporaerer Imaginaerteil
74    int NX = getNX();
75    int NY = getNY();
76    int NXDiv2 = NX /2;
77    int NYDiv2 = NY /2;
78
79    // Nummerierung der Viertel (Links oben ist Element (0,0)):
80    //          |
81    //       2  | 1
82    //    ____|____   |->x
83    //          |       |
84    //       3  | 4     y
85    //          |
86
87    // Die folgenden Variablen kennzeichnen den ersten
88    // bzw. den letzten Index im entsprechenden Quadranten
89    int BeginnFirstQuarterX = NX / 2;
90    int EndSecondQuarterX   = BeginnFirstQuarterX - 1;
91    int BeginnThirdQuarterY = NY / 2;
92    int EndSecondQuarterY     = BeginnThirdQuarterY -1;
93
94    // Tausch des 2. mit dem 4. Viertel
95    for (i=0; i<=EndSecondQuarterX; i++) {
96      for (j=0; j<=EndSecondQuarterY; j++) {
97        // Zwischenspeichern des zweiten Viertels
98        TempRe = Array[j][i].re;
99        TempIm = Array[j][i].im;
100       // Umkopieren der Werte vom vierten in das zweite Viertel
101       Array[j][i].re = Array[j+NYDiv2][i+NXDiv2].re;
102       Array[j][i].im = Array[j+NYDiv2][i+NXDiv2].im;
103       // Speichern der zwischengelagerten Werte in das vierte Viertel
104       Array[j+NYDiv2][i+NXDiv2].re = TempRe;
105       Array[j+NYDiv2][i+NXDiv2].im = TempIm;
106     }
107   }
108
```

```
109     // Tausch des 1. mit dem 3. Viertel
110     for (i=BeginnFirstQuarterX; i<NX; i++) {
111       for (j=0; j<=EndSecondQuarterY; j++) {
112         // Zwischenspeichern des ersten Viertels
113         TempRe = Array[j][i].re;
114         TempIm = Array[j][i].im;
115         // Umkopieren der Werte vom dritten in das erste Viertel
116         Array[j][i].re = Array[j+NYDiv2][i-NXDiv2].re;
117         Array[j][i].im = Array[j+NYDiv2][i-NXDiv2].im;
118         // Speichern der zwischegelagerten Werte in das dritte Viertel
119         Array[j+NYDiv2][i-NXDiv2].re = TempRe;
120         Array[j+NYDiv2][i-NXDiv2].im = TempIm;
121       }
122     }
123   }
```

```
1   // ElectricalField.h: Schnittstelle für die Klasse CElectricalField.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if !defined(AFX_U_H__FE514781_A992_11D4_8960_0050BF19F9FF__INCLUDED_)
6   #define AFX_U_H__FE514781_A992_11D4_8960_0050BF19F9FF__INCLUDED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #include "Field.h"
13  #include <math.h>
14  #include <string.h>
15
16
17  class CElectricalField : public CField
18  {
19
20  public:
21    CElectricalField();
22    virtual ~CElectricalField();
23
24    // get-Funktionen
25    // Abfrage der Wellenlaenge vom elektrischen Feld
26    double getLambda(void)  const { return Lambda; };
27    // Abfrage von k
28    double getK(void)       const { return (2*M_PI)/Lambda; };
29    // Abfrage der physikalischen Breite des elektrischen Feldes
30    double getWidthXE(void) const { return WidthXE; };
31    // Abfrage der physikalischen Hoehe des elektrischen Feldes
32    double getHeightYE(void) const { return HeightYE; };
33    // Abfrage der Aufloesung in x-Richtung
34    double getDeltaXE(void) const { return DeltaXE; };
35    // Abfrage der Aufloesung in y-Richtung
36    double getDeltaYE(void) const { return DeltaYE; };
37
38    // set-Funktionen
39    // Setzen der Wellenlaenge
40    void setLambda(double _Lambda) { Lambda=_Lambda; };
41    // Setzen eines Eingangsfeldes (nur zum Testen des Programms benoetigt)
42    void setTestEField(size_t _NX, size_t _NY, double _WidthXE, double _HeightYE, double
      _Lambda);
43
44    // Seitliches auffuellen des elektrischen Feldes mit Nullen
45    void fillUp(double WishWidth, double WishHeight);
46    // Ausgabe des elektrischen Feldes
47    virtual void write(COut& Out);
48    // Einlesen eines elektrischn Feldes
49    virtual void read(string& dateiname, FILE *fd,char mode);
50    // Ueberpruefung, ob zwei elektrische Felder dieselben Eigenschaften besitzen
51    bool equalParameters(CElectricalField& E);
52
53  protected:
54    void init(void);        // Initialisierung der Variablen
```

```
55
56      double Lambda;          // Wellenlänge des E-Feldes
57      double WidthXE;         // Breite des E-Feldes in x-Richtung
58      double HeightYE;        // Höhe des E-Feldes in y-Richtung
59      double DeltaXE;         // Abstand der Stützstellen in x-Richtung
60      double DeltaYE;         // Abstand der Stützstellen in y-Richtung
61
62  };
63
64  #endif //
    !defined(AFX_U_H__FE514781_A992_11D4_8960_0050BF19F9FF__INCLUDED_)
```

```
1   // ElectricalField.cpp: Implementierung der Klasse CCElectricalField.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "ElectricalField.h"
6
7
8   CElectricalField::CElectricalField():CField()
9   {
10    init();
11  }
12
13
14  CElectricalField::~CElectricalField()
15  {
16
17  }
18
19
20  // Initialisierung der Variablen
21  void CElectricalField::init(void)
22  {
23    Lambda=WidthXE=HeightYE=DeltaXE=DeltaYE=-1.0;
24  }
25
26
27  // Ueberpruefung, ob zwei elektrische Felder dieselben Eigenschaften besitzen
28  bool CElectricalField::equalParameters(CElectricalField& E)
29  {
30    // Ueberprufung, ob die Anzahl der Elemente in x- und y-Richtung, die physikalische
    Feldgroesse
31    // und die Wellenlaenge der Felder identisch sind.
32    if ( (getNX()==E.getNX()          ) && (getNY()==E.getNY()          ) &&
33         (getWidthXE()==E.getWidthXE()) && (getHeightYE()==E.getHeightYE()) &&
34         (getLambda()==E.getLambda()  )   ) {
35      return true;
36    }
37    return false;
38  }
39
40
41  // Seitliches auffüllen des elektrischen Feldes mit Nullen
42  void CElectricalField::fillUp(double WishWidth, double WishHeight)
43  {
44    // Lokale Variablen
45    size_t NewNX;      // Zukuenftiges NX
46    size_t NewNY;      // Zukuenftiges NY
47    int OffsetX;       // Gibt an wie weit die alten Daten in x-Richtung in das neue Feld
                            "hineingeschoben" werden
48    int OffsetY;       // Gibt an wie weit die alten Daten in y-Richtung in das neue Feld
                            "hineingeschoben" werden
49
50    // Aus Gruenden der Rechengeschwindigkeit wird ein zu vergroesserndes Feld nur
51    //   verdoppelt, vervierfacht, verachtfacht, ...(Zweier-Potenz)
52    // Berechnung des Vergroesserungsfaktors in Breiten- und Hoehenrichtung
```

```
53      double ExpWidth = ceil (log(WishWidth/WidthXE) / log(2));
54      double ExpHeight = ceil (log(WishHeight/HeightYE) / log(2));
55
56      // Berechnung der neuen Dimensionen in x-Richtung
57      // Hinweis: Sicherheitshalber wird für den Fall, dass keine Vergroesserung durchgefuehrt
58      //          werden muss, die alten Werte als Neue angenommen.
59      if ( WishWidth > WidthXE ) {
60        NewNX = getNX() * pow(2.0, ExpWidth);
61        OffsetX = (int) ( (NewNX-getNX())/2 );
62      }
63      else if ( WishWidth <= WidthXE ) {
64        NewNX = CField::getNX();
65        OffsetX = 0;
66      }
67
68      // Berechnung der neuen Dimensionen in y-Richtung
69      // Hinweis: Sicherheitshalber wird für den Fall, dass keine Vergroesserung durchgefuehrt
70      //          werden muss, die alten Werte als Neue angenommen.
71      if ( WishHeight > HeightYE ) {
72        NewNY = getNY() * pow(2.0, ExpHeight);
73        OffsetY = (int) ( (NewNY-getNY())/2 );
74      }
75      else if ( WishHeight <= HeightYE ) {
76        NewNY = CField::getNY();
77        OffsetY = 0;
78      }
79
80      // Wenn NX oder NY geaendert werden muessen, wird dies entsprechend durchgefuehrt.
81      if ( !((NewNX==getNX()) && (NewNY==getNY())) ) {
82        CField::setSize(NewNY, NewNX, true, OffsetY, OffsetX);
83        HeightYE = DeltaYE * NewNY;
84        WidthXE = DeltaXE * NewNX;
85        CPlan::PlanValid = false;
86      }
87    }
88
89
90    // Setzen eines Eingangsfeldes (nur zum Testen des Programms benoetigt)
91    void CElectricalField::setTestEField(size_t _NX, size_t _NY, double _WidthXE, double
      _HeightYE, double _Lambda)
92    {
93      // Variablen zu Bestimmung der Feldeigenschaften
94      int NX = _NX;
95      int NY = _NY;
96      HeightYE = _HeightYE;
97      WidthXE = _WidthXE;
98      Lambda = _Lambda;
99
100     // Lokale Variablen
101     double i,j;
102     DeltaYE = HeightYE/NY;
103     DeltaXE = WidthXE/NX;
104     double K = (2*M_PI) /Lambda;
105     CField::setSize(NY,NX,false,0,0); // Festlegung der Feldgroesse
106
107     // Definition einer ebenen Welle:
```

```
108     for (i=0; i<NX; i++) {
109       for (j=0; j<NY; j++) {
110         Array[j][i].re = 1.0;
111         Array[j][i].im = 0.0;
112       }
113     }
114   /*
115     // Definition eines Quadrats als Objekt:
116     // Zur Zeit:
117     // Kamera mit 4096: 1638 bis 2048 (=2,001953125mm)
118     // Kamera mit 5120: 2048 bis 2560 (=2mm)
119     for (i=0; i<NX; i++) {
120       for (j=0; j<NY; j++) {
121         Array[j][i].re = 0.0;
122         Array[j][i].im = 0.0;
123       }
124     }
125     for (i=(2048); i<(NX/2); i++) {
126       for (j=(2048); j<(NY/2); j++) {
127         Array[j][i].re = 1.0;
128         Array[j+1][i].re = 1.0;
129       }
130     }
131   */
132   /*
133     // Definition von vier Punktquellen, die wie folgt angeordnet sind:
134     //          P3 y           P4
135     //            |_x
136     //          P2             P1
137     //
138     float z = 0.15;   // Entfernung der vier Punktquellen zur berechneten Ebene
139     float Phase1 =0;  // Phase des ersten Punktes
140     float Phase2 =0;  // Phase des zweiten Punktes
141     float Phase3 =0;  // Phase des dritten Punktes
142     float Phase4 =0;  // Phase des vierten Punktes
143     float A1 = 6;     // Amplitude der ersten Punktquelle
144     float A2 =10;     // Amplitude der zweiten Punktquelle
145     float A3 =10;     // Amplitude der dritten Punktquelle
146     float A4 = 6;     // Amplitude der vierten Punktquelle
147
148     // Definition der Lage der Punktquellen im Verhaeltnis zur Feldmitte
149     int P1X = (NX/2) + 22;  // x-Abstand von 1 zu 2
150     int P1Y = NY/2;
151     int P2X = (NX/2);
152     int P2Y = (NY/2);
153     int P3X = P2X;
154     int P3Y = P2Y    - 5;  // y-Abstand von 3 zu 2
155     int P4X = P1X;
156     int P4Y = P3Y;
157
158
159     // Berechnung des elektrischen Feldes
160     float z2 = z*z;
161     // Zukuenftige Amplituden
162     A1 = A1*exp(Phase1);
163     A2 = A2*exp(Phase2);
```

```
164     A3 = A3*exp(Phase3);
165     A4 = A4*exp(Phase4);
166
167     // Berechnung der Punkte 1 und 2
168     // temporaere Variablen
169     float Temp1, Temp2;
170     // Raeumliche Abstaende der zu berechnenden Punkte von den Punktquellen
171     float Radius1, Radius2;
172     //Abstaende des zu berechnenden Punktes von der Punktquelle in x- und y-Richtung
173     float AbstandX1, AbstandY1;
174     float AbstandX2, AbstandY2;
175     for (i=0; i<NX; i++) {
176       for (j=0; j<NY; j++) {
177         AbstandX1 = (i-P1X)*DeltaXE;
178         AbstandY1 = (j-P1Y)*DeltaYE;
179         AbstandX2 = (i-P2X)*DeltaXE;
180         AbstandY2 = (j-P2Y)*DeltaYE;
181         Radius1 = sqrt(z2 + (AbstandX1*AbstandX1) + (AbstandY1*AbstandY1));
182         Radius2 = sqrt(z2 + (AbstandX2*AbstandX2) + (AbstandY2*AbstandY2));
183         Temp1 = ((-1)*K*Radius1) + Phase1;
184         Temp2 = ((-1)*K*Radius2) + Phase2;
185         Array[j][i].re = ((A1*cos(Temp1)) + (A2*cos(Temp2)))/Radius1;
186         Array[j][i].im = ((A1*sin(Temp1)) + (A2*sin(Temp2)))/Radius2;
187       }
188     }
189     // Superposition der Punkte 3 und 4 zum elektrischen Feld der Punkte 1 und 2
190     // Hinweis: Die Berechnung erfolgt identisch.
191     float Temp3, Temp4;
192     float Radius3, Radius4;
193     float AbstandX3, AbstandY3;
194     float AbstandX4, AbstandY4;
195     for (i=0; i<NX; i++) {
196       for (j=0; j<NY; j++) {
197         AbstandX3 = (i-P3X)*DeltaXE;
198         AbstandY3 = (j-P3Y)*DeltaYE;
199         AbstandX4 = (i-P4X)*DeltaXE;
200         AbstandY4 = (j-P4Y)*DeltaYE;
201         Radius3 = sqrt(z2 + (AbstandX3*AbstandX3) + (AbstandY3*AbstandY3));
202         Radius4 = sqrt(z2 + (AbstandX4*AbstandX4) + (AbstandY4*AbstandY4));
203         Temp3 = ((-1)*K*Radius3) + Phase3;
204         Temp4 = ((-1)*K*Radius4) + Phase4;
205         Array[j][i].re += ((A3*cos(Temp3)) + (A4*cos(Temp4)))/Radius3;
206         Array[j][i].im += ((A3*sin(Temp3)) + (A4*sin(Temp4)))/Radius4;
207       }
208     }
209   */
210   }
211
212   // Ausgabe des elektrischen Feldes
213   void CElectricalField::write(COut& Out)
214   {
215     Out << "EF";      // ID
216     Out << '0';       // Version
217     Out << DeltaXE;
218     Out << DeltaYE;
219     Out << WidthXE;
```

```
220     Out << HeightYE;
221     Out << Lambda;
222
223     CField::write(Out);
224
225     return;
226   }
227
228
229   // Einlesen der Felddaten
230   void CElectricalField::read(string& dateiname, FILE *fd, char mode)
231   {
232     size_t charCounter=0;
233     char str[2];
234     char version;
235
236     if(mode!='b') {
237       ::cerr << "Warnung CElectricalField::read: Lesen von ASCIIDaten nocht nicht
      implementiert";
238     }
239
240     charCounter+=fread(&str[0],sizeof(char),2,fd);
241     if( !(str[0]=='E' && str[1]=='F') ) {
242       ::cerr << "Fehler CElectricalField::read: Falscher Typ in Datei";
243       return;
244     }
245     charCounter+=fread(&version,sizeof(char),1,fd);
246     charCounter+=fread(&DeltaXE,sizeof(double),1,fd);
247     charCounter+=fread(&DeltaYE,sizeof(double),1,fd);
248     charCounter+=fread(&WidthXE,sizeof(double),1,fd);
249     charCounter+=fread(&HeightYE,sizeof(double),1,fd);
250     charCounter+=fread(&Lambda,sizeof(double),1,fd);
251     fclose(fd);
252
253     CField::read(dateiname);
254
255     return;
256   }
```

```
1   // TransferFunction.h: Schnittstelle für die Klasse CTransferFunction.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if
    !defined(AFX_TRANSFERFUNCTION_H__9EC9EA40_ADAC_11D4_9702_89ACD9685454__INCLUDED_)
6   #define
    AFX_TRANSFERFUNCTION_H__9EC9EA40_ADAC_11D4_9702_89ACD9685454__INCLUDED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #include "Field.h"
13
14
15  // Die Klasse stellt keine weiteren Funktionen oder Eigenschaften zur Verfuegung.
16  // Wird nur wegen aus objektorientierten Gruenden erstellt.
17  class CTransferFunction : public CField
18  {
19  public:
20    CTransferFunction();
21    virtual ~CTransferFunction();
22
23    // Abfrage der Wellenlaenge von der H-Funktion
24    double getLambdaH(void)  const { return LambdaH; };
25    // Setzen der Wellenlaenge fuer die die H-Funktion gueltig ist
26    void setLambdaH(double _LambdaH) { LambdaH = _LambdaH; };
27    // Abfrage der physikalischen Breite der Uebertragungsfunktion
28    double getWidthXH(void) const { return WidthXH; };
29    // Setzen der physikalischen Breite fuer die die Uebertragungsfunktion gueltig ist
30    void setWidthXH(double _WidthXH) { WidthXH = _WidthXH; };
31    // Abfrage der physikalischen Hoehe der Uebertragungsfunktion
32    double getHeightYH(void) const { return HeightYH; };
33    // Setzen der physikalischen Hoehe fuer die die Uebertragungsfunktion gueltig ist
34    void setHeightYH(double _HeightYH) { HeightYH = _HeightYH; };
35
36  protected:
37    void init() { LambdaH = WidthXH = HeightYH = 0; };
38
39    double LambdaH;        // Wellenlänge fuer die die H-Funktion berechnet wurde
40    double WidthXH;        // Breite der H-Funktion in x-Richtung
41    double HeightYH;       // Höhe des H-Funktion in y-Richtung
42
43  };
44  #endif //
    !defined(AFX_TRANSFERFUNCTION_H__9EC9EA40_ADAC_11D4_9702_89ACD9685454__INCLUDED_)
```

```
1   // TransferFunction.cpp: Implementierung der Klasse CTransferFunction.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "TransferFunction.h"
6
7   //////////////////////////////////////////////////////////////////////
8   // Konstruktion/Destruktion
9   //////////////////////////////////////////////////////////////////////
10
11
12  CTransferFunction::~CTransferFunction()
13  {
14
15  }
16
17
18  CTransferFunction::CTransferFunction():CField()
19  {
20    init();
21  }
```

```
1   // TransmissionFunction.h: Schnittstelle für die Klasse CTransmissionFunction.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if
    !defined(AFX_TRANSMISSIONFUNCTION_H__6EC1BDA3_C056_11D4_AF24_00E07D
    006091__INCLUDED_)
6   #define
    AFX_TRANSMISSIONFUNCTION_H__6EC1BDA3_C056_11D4_AF24_00E07D006091_
    _INCLUDED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #include "Field.h"
13
14
15  // Die Klasse stellt keine weiteren Funktionen oder Eigenschaften zur Verfuegung.
16  // Wird nur wegen aus objektorientierten Gruenden erstellt.
17  class CTransmissionFunction : public CField
18  {
19
20  public:
21    CTransmissionFunction();
22    virtual ~CTransmissionFunction();
23
24    // Abfrage der Wellenlaenge von der T-Funktion
25    double getLambdaT(void)  const { return LambdaT; };
26    // Setzen der Wellenlaenge fuer die die T-Funktion gueltig ist
27    void setLambdaT(double _LambdaT) { LambdaT = _LambdaT; };
28    // Abfrage der physikalischen Breite der Transmissionsfunktion
29    double getWidthXT(void) const { return WidthXT; };
30    // Setzen der physikalischen Breite fuer die die Transmissionsfunktion gueltig ist
31    void setWidthXT(double _WidthXT) { WidthXT = _WidthXT; };
32    // Abfrage der physikalischen Hoehe der Transmissionsfunktion
33    double getHeightYT(void) const { return HeightYT; };
34    // Setzen der physikalischen Hoehe fuer die die Transmissionfunktion gueltig ist
35    void setHeightYT(double _HeightYT) { HeightYT = _HeightYT; };
36
37  protected:
38   void init() { LambdaT = WidthXT = HeightYT = 0; };
39
40    double LambdaT;        // Wellenlänge fuer die die T-Funktion berechnet wurde
41    double WidthXT;        // Breite der T-Funktion in x-Richtung
42    double HeightYT;       // Höhe des T-Funktion in y-Richtung
43  };
44
45  #endif //
    !defined(AFX_TRANSMISSIONFUNCTION_H__6EC1BDA3_C056_11D4_AF24_00E07D
    006091__INCLUDED_)
```

```
1   // TransmissionFunction.cpp: Implementierung der Klasse
    CTransmissionFunction.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "TransmissionFunction.h"
6
7   //////////////////////////////////////////////////////////////////////
8   // Konstruktion/Destruktion
9   //////////////////////////////////////////////////////////////////////
10
11  CTransmissionFunction::CTransmissionFunction():CField()
12  {
13    init();
14  }
15
16  CTransmissionFunction::~CTransmissionFunction()
17  {
18
19  }
```

```
1   // CameraElement.h: Schnittstelle für die Klasse CCameraElement.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if
    !defined(AFX_CAMERAELEMENT_H__4FD964F6_C509_11D4_AF24_00E07D006091__
    INCLUDED_)
6   #define
    AFX_CAMERAELEMENT_H__4FD964F6_C509_11D4_AF24_00E07D006091__INCLUD
    ED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #include <iostream.h>
13  #include <string.h>
14  #include <stdlib.h>
15  #include <stdio.h>
16  #include <math.h>
17
18  #include "ElectricalField.h"
19  #include "..\Out\Out.h"
20  #include "..\read\Input.h"
21
22
23  class CCameraElement
24  {
25
26  public:
27    CCameraElement();
28    virtual ~CCameraElement();
29
30    // Berechnung der E-Felder (opt. Element) bzw. der Intensität (Schirm)
31    virtual void calc(CElectricalField& EX, CElectricalField& EY) =0;
32    // Zuweisen der Elementnummer
33    void setNo(int _No) { No=_No; };
34
35    virtual void writeElement(COut& KameraOut)=0;  // Ausgabe der Objektdaten
36    virtual double getWidthX(void)          =0;  // Ausgabe der maximalen Breite des Objektes
37    virtual double getHeightY(void)          =0;   // Ausgabe der maximalen Breite des
    Objektes
38    virtual void read (CInput& Input)       =0;  // Einlesen der Objekt-Daten
39    virtual size_t getNX(void);                  // Ausgabe der Stuetzstellen in x-Richtung
40    virtual size_t getNY(void);                  // Ausgabe der Stuetzstellen in y-Richtung
41
42    // Change-Funktionen zur Steuerung des automatischen Aenderns von
    Objekteigenschaften
43    virtual void setAllChange(void)    =0;
44    virtual void syncChange(void)      =0;
45    virtual bool set(void)             =0;
46    virtual void resetAllChange(void)  =0;
47
48    // Abfrage des Wertes der Variablen change.changeSize
49    virtual bool getChangeSize(void)   =0;
```

```
50      // Setzen der Werte fuer change.changeH und change.changeT
51      virtual void setChangeHT(bool NewChangeHT)  =0;
52
53
54    protected:
55      void init(void) { No=999; CalcNumber = 0; } ;
56
57      int No;                    // Elementnummer innerhalb der Kamera
58      int CalcNumber;        // Anzahl der bereits durchgeführten Berechnungen
59    };
60
61    #endif //
      !defined(AFX_CAMERAELEMENT_H__4FD964F6_C509_11D4_AF24_00E07D006091__
      INCLUDED_)
```

```
1   // CameraElement.cpp: Implementierung der Klasse CCameraElement.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "CameraElement.h"
6
7
8   CCameraElement::CCameraElement()
9   {
10    init();
11  }
12
13  CCameraElement::~CCameraElement()
14  {
15
16  }
17
18  // liefert die Groesse NX des letzten Kamera Elementes zurueck
19  // Diese Funktion wird jedoch nur aufgerufen, falls der Anwender an die letzte Stelle
20  // in der Kamera keinen Schirm gesetzt hat.
21  size_t CCameraElement::getNX(void)
22  {
23    ::cerr<<"CCameraElement::getNX():  Letztes Element ist kein Schirm";
24    return 0;
25  }
26
27
28  // liefert die Groesse NY des letzten Kamera Elementes zurueck
29  // Diese Funktion wird jedoch nur aufgerufen, falls der Anwender an die letzte Stelle
30  // in der Kamera keinen Schirm gesetzt hat.
31  size_t CCameraElement::getNY(void)
32  {
33    ::cerr<<"CCameraElement::getNY():  Letztes Element ist kein Schirm";
34    return 0;
35  }
```

```
1   // Screen.h: Schnittstelle für die Klasse CScreen.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if
    !defined(AFX_SCREEN_H__B4308B20_AB84_11D4_9702_B3D911E63CA8__INCLUDE
    D_)
6   #define
    AFX_SCREEN_H__B4308B20_AB84_11D4_9702_B3D911E63CA8__INCLUDED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #ifdef _DEBUG
13  #pragma warning(disable: 4786)
14  #endif
15
16  #ifdef _DEBUG
17  #pragma warning(disable: 4275)
18  #endif
19
20  #include "CameraElement.h"
21  #include "..\VectorManager\ArrayManager.h"
22
23
24  using namespace std;
25
26  class CScreen : public CCameraElement
27  {
28  public:
29    CScreen();
30    virtual ~CScreen();
31
32    virtual void read(CInput& Input);                    // Einlesen der Objekt-Daten
33    virtual void writeElement(COut& KameraOut);  //Ausgabe der eigenen Objektdaten
34
35    // Berechnung der Intensität fuer zwei senkrecht zueinander polarisierten elektrischen
        Feldern
36    virtual void calc(CElectricalField& EX, CElectricalField& EY);
37
38    virtual double getWidthX(void) { return WidthXS; };  // Ausgabe der maximalen Breite des
    Objektes
39    virtual double getHeightY(void) { return HeightYS; };  // Ausgabe der maximalen Breite
    des Objektes
40    double getDeltaXS(void) { return (WidthXS/NX); };    // Ausgabe der Auflösung
41    double getDeltaYS(void) { return (HeightYS/NY); };    // Ausgabe der Auflösung
42    virtual size_t getNX(void) { return NX; };
43    virtual size_t getNY(void) { return NY; };
44
45    // Change-Funktionen zur Steuerung des automatischen Aenderns von
        Objekteigenschaften
46    virtual void setAllChange(void);
47    virtual void syncChange(void);
48    virtual bool set(void);
49    virtual void resetAllChange(void);
```

```
50
51    virtual bool getChangeSize(void) { return change.changeSize; };  // Gibt an, ob die
                                                                    Schirmgroesse veraendert wurde
52    virtual void setChangeHT(bool NewChangeHT) { };   // Wird nur verwendet, um eine
                                                  Laufzeittypidentifiezierung in CCamera zu verhindern
53
54
55  protected:
56    void init(void);
57
58    // Ausgabe der Intensität im TIF-Format
59    void writeTIFF(char Filename[100]);
60    // lineare Skalierung der Intensität
61    void scaleLinearIntensity();
62    // logarithmische Skalierung der Intensität im TIFF-Ausgabeformat
63    void logIntensityTIFF();
64
65    CArrayManager<float> I;      // Feld mit der Intensitätsverteilung
66    double WidthXS;              // Breite des Schirms
67    double HeightYS;               // Hoehe des Schirms
68    int NX;                      // geplante Anzahl von Stuetzstellen in x-Richtung
69    int NY;                      // geplante Anzahl von Stuetzstellen in y-Richtung
70    double CalculatedWidthXS;  // Endgueltig berechnete Breite des Schirms
71    double CalculatedHeightYS;  // Endgueltig berechnete Hoehe des Schirms
72    int CalculatedNX;          // Endgueltig berechnete Anzahl von Stuetzstellen in x-Richtung
73    int CalculatedNY;          // Endgueltig berechnete Anzahl von Stuetzstellen in y-Richtung
74    COut OutI;                 // Ausgabeobjekt fuer die Intensitaetsveerteilung
75
76    // Statusvariable
77    struct {
78      bool DestroyI;   // Angabe ob die Intensitaetsverteilung nach der Berchnung zerstoert
                              werden soll. (ja=true, nein=false)
79    } status;
80
81    // Ausgabevariablen
82    struct {
83      // Angabe, ob die Intensitaetsverteilung ausgegeben werden soll.
84      // Moeglichkeiten: ASCII:"t", Binaer:"b", Nein "n"
85      char WriteI;
86      bool WriteTIFF;  // Angabe ob die Intensitaetsverteilung in einem TIFF-Format
                              ausgegeben werden soll
87      bool LogTIFF;    // Angabe ob die TIFF-Ausgabe logarithmiert wird
88    } output;
89
90    // Change-Variablen
91    struct {
92      bool change;               // Angabe ob irgentetwas veraendeert wurde
93      bool changeSize;           // Angabe ob die Groesse veraendert wurde
94    } change;
95  };
96  #endif //
    !defined(AFX_SCREEN_H__B4308B20_AB84_11D4_9702_B3D911E63CA8__INCLUDE
    D_)
```

```
1   // Screen.cpp: Implementierung der Klasse CScreen.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "Screen.h"
6
7   //////////////////////////////////////////////////////////////////////
8   // Konstruktion/Destruktion
9   //////////////////////////////////////////////////////////////////////
10
11  CScreen::~CScreen()
12  {
13    init();
14  }
15
16
17  CScreen::CScreen():CCameraElement()
18  {
19    init();
20  }
21
22
23  void CScreen::init(void)
24  {
25    CalculatedWidthXS = CalculatedHeightYS = WidthXS = HeightYS = -1.0;
26    CalculatedNX = CalculatedNY = NX = NY = 1;
27    output.WriteI = output.WriteTIFF = output.LogTIFF = status.DestroyI = true;
28    setAllChange();
29  }
30
31
32  // Zuruecksetzen der Change-Variablen
33  void CScreen::resetAllChange(void)
34  {
35    change.change = change.changeSize = false;
36  }
37
38
39  // Setzen aller Change-Variablen
40  void CScreen::setAllChange(void)
41  {
42    change.change = change.changeSize = true;
43  }
44
45
46  // Setzen der hier nicht vorhandenen Statusvariablen
47  bool CScreen::set(void)
48  {
49    return false;
50  }
51
52
53  // Synchronisierung der Change-Variablen
54  void CScreen::syncChange(void)
```

```
55  {
56    if (change.change==true) change.changeSize=true;
57  }
58
59
60  // Durchfuehrung einer linearen Skalierung der Intensitaetsveteilung I[j][i]
61  // auf einen maximalen Bezugswert
62  void CScreen::scaleLinearIntensity()
63  {
64    // lokale Variablen
65    int i, j;
66    float MAX =0;
67    float Scale =0;
68
69    // Ermittlung der größten Intensität
70    for (j=0; j<CalculatedNY; j++){
71      for (i=0; i<CalculatedNX; i++){
72        if (MAX < I[j][i]) { MAX = I[j][i];};
73      }
74    }
75    // Skalierung der Intensitätsverteilung
76    Scale = 255/MAX;
77    for (j=0; j<CalculatedNY; j++){
78      for (i=0; i<CalculatedNX; i++){
79        I[j][i] = floor((I[j][i])*Scale);
80      }
81    }
82  }
83
84
85  // Berechnung der Intensitaetsverteilung I[j][i] fuer zwei senkrecht zueinander
86  // stehende elektrische Felder
87  void CScreen::calc(CElectricalField &EX, CElectricalField &EY)
88  {
89    // lokale Variable
90    size_t i,j;
91    float TempIX, TempIY;
92    // Const = 0.5 * Epsilon * Lichtgeschwindigkeit = Vorfaktor für die Intensität
93    float Const =0.5*(8.8542E-12)*(2.997925E+8);
94
95    ////////////////////////////////////////////////////////////////
96    // Ueberpruefung der Eigenschaften der beiden elektrischen Felder,
97    // ob sie identisch sind. Wenn ja erfolgt gleich die Berechnung der Intensitaetsverteilung
98    if ( EX.equalParameters(EY) ) {
99      // Anpassen des Schirms an das endgueltig durchgerechnete elektrische Feld
100     CalculatedNX = EX.CField::getNX();
101     CalculatedNY = EX.CField::getNY();
102     CalculatedWidthXS = EX.getWidthXE();
103     CalculatedHeightYS = EX.getHeightYE();
104     I.CArrayManager<float>::setSize(CalculatedNY, CalculatedNX, false, 0, 0);
105
106     // Berechnung der Intensitaetsverteilung
107     for (i=0; i<CalculatedNX; i++) {
108       for (j=0; j<CalculatedNY; j++) {
109         // Einfache Addition der beiden Intensitaeten und Multiplikation mit dem konstanten
              Vorfaktor
```

```
110         TempIX = (EX(j,i).re*EX(j,i).re) + (EX(j,i).im*EX(j,i).im);
111         TempIY = (EY(j,i).re*EY(j,i).re) + (EY(j,i).im*EY(j,i).im);
112         I[j][i] = Const * (TempIX + TempIY);
113       }
114     }
115     CalcNumber++;    // Erhoehung der Anzahl der berechneten "Bilder"
116   }
117   else {
118     ::cerr<<"calcIntensity(EX, EY): Die übergebenen Felder besitzen nicht dieselben
      Eigenschaften.\n";
119   }
120
121   ////////////////////////////////////
122   //Ausgabe der Intensitätsverteilung
123   ////////////////////////////////////
124   // lokale Variablen zur automatischen Erstellung eines Dateinamens
125   // fuer die Outobjekte
126   char No_ASCII[2];                    // enthaelt den Ort (Stelle) des Elements
127   char CalcNumber_ASCII[4];            // enthaelt die Nummer des berechneten Bildes
128   // (generierte Dateinamen duerfen maximal 100 Zeichen enthalten)
129   char Filename[100]="Screen_Place";  // 1.Teil des Dateinamens
130   char *pFilename;                     // Zeiger auf den Dateinamen
131   pFilename=Filename;
132
133   _itoa(No,No_ASCII,10);              // Konvertierung der Elementnummer in ASCII-Zeichen
134   strcat(pFilename,No_ASCII);         // Anhaengen der Elementnummer an den Dateinamen
135   strcat(pFilename,"_Picture");
136   _itoa(CalcNumber,CalcNumber_ASCII,10);  // Konvertierung der Bildnummer in ASCII-
                                                Zeichen
137   strcat(pFilename,CalcNumber_ASCII);         // Anhaengen der Bildnummer
138
139   // Definition des Outobjektes
140   if ( output.WriteI=='t') {
141     OutI.open(Filename,"wt");
142   }
143   if ( output.WriteI=='b') {
144     OutI.open(Filename,"wb");
145   }
146   // Ausgabe der Intensitaetsverteilung mit Hilfe des ArrayManagers
147   if ( output.WriteI=='t' || output.WriteI=='b' ) {
148     OutI << WidthXS;
149     OutI << HeightYS;
150     OutI << I;
151   }
152   //Ausgabe der Intensitaetsverteilung als TIF-Bild
153   if (output.WriteTIFF==true) {
154     writeTIFF(Filename);
155   }
156
157   // Abfrage, ob das Intensitätsfeldes zerstört werden soll
158   if (status.DestroyI==true) {
159     I.CArrayManager<float>::setSize(1, 1, false, 0, 0);
160   }
161 }
162
163
```

```
164   // Ausgabe der Intensitaetsvereilung in einem TIF-Bild
165   // als RGB-Bild mit drei gleichen Farbanteilen
166   // Fuer spaeter zu berechnende Bilder mit einer hoeheren Aufloesung
167   // kann hier eine einfache Komprimierung erfolgen
168   void CScreen::writeTIFF(char Filename[100])
169   {
170     // lokale Variablen
171     size_t i, j;
172     int Offset;
173     FILE *fptr;
174     char *pFilename;
175     pFilename=Filename;
176
177     strcat(pFilename,".tif");       // Anhaengen der Dateiendung
178     fptr = fopen(Filename,"wb");    // Oeffnen der Datei
179
180     // Durchführung der benötigten Skalierung (logarithmisch oder linear),
181     // damit der maximale Intensitaetswert für das Bild bei 255 liegt.
182     if (output.LogTIFF==false) {
183       scaleLinearIntensity();
184     }
185     if (output.LogTIFF==true) {
186       // Zu beachten ist die Reihenfolge der logarithmischen und linearen Skalierung
187       logIntensityTIFF();
188       scaleLinearIntensity();
189     }
190
191
192     // Schreiben der Kopfdaten gemaess des TIFF 6.0 - Standards (liegt der Ausarbeitung
193     // in Form einer PDF-Datei bei.)
194     // Dieser Datei kann die gesamte Beschreibung des TIFF-Formats entnommen werden.
195     // "4d 4d 00 2a" = Little endian & TIFF identifier
196     putc(77,fptr);
197     putc(77,fptr);
198     putc(0,fptr);
199     putc(42,fptr);
200
201     Offset = CalculatedNX * CalculatedNY *3  + 8;
202     putc((Offset & 0xff000000) / 16777216,fptr);
203     putc((Offset & 0x00ff0000) / 65536,fptr);
204     putc((Offset & 0x0000ff00) / 256,fptr);
205     putc((Offset & 0x000000ff),fptr);
206
207     // Schreiben der Binärdaten
208     for (j=0;j<CalculatedNY;j++) {
209       for (i=0;i<CalculatedNX;i++) {
210         putc((int)floor(I[j][i]),fptr);
211         putc((int)floor(I[j][i]),fptr);
212         putc((int)floor(I[j][i]),fptr);
213       }
214     }
215
216     // Schreiben der Fußdaten
217     // "000e"
218     putc(0,fptr);
219     putc(14,fptr);  // Anzahl der Einträge ind die Directory
```

```
220
221     // Width tag, short int
222     // "01 00 00 03 00 00 00 01"
223     putc(1,fptr);
224     putc(0,fptr);
225     putc(0,fptr);
226     putc(3,fptr);
227     putc(0,fptr);
228     putc(0,fptr);
229     putc(0,fptr);
230     putc(1,fptr);
231     fputc((CalculatedNX & 0xff00) / 256,fptr);    // Bildbreite
232     fputc((CalculatedNX & 0x00ff),fptr);
233     // "0000"
234     putc(0,fptr);
235     putc(0,fptr);
236
237     // Height tag, short int
238     // "01 01 00 03 00 00 00 01"
239     putc(1,fptr);
240     putc(1,fptr);
241     putc(0,fptr);
242     putc(3,fptr);
243     putc(0,fptr);
244     putc(0,fptr);
245     putc(0,fptr);
246     putc(1,fptr);
247     fputc((CalculatedNY & 0xff00) / 256,fptr);    // Bildhöhe
248     fputc((CalculatedNY & 0x00ff),fptr);
249     // "0000"
250     putc(0,fptr);
251     putc(0,fptr);
252
253     // Bits per sample tag, short int
254     // "01 02 00 03 00 00 00 03"
255     putc(1,fptr);
256     putc(2,fptr);
257     putc(0,fptr);
258     putc(3,fptr);
259     putc(0,fptr);
260     putc(0,fptr);
261     putc(0,fptr);
262     putc(3,fptr);
263     Offset = CalculatedNX * CalculatedNY *3 + 182;
264     putc((Offset & 0xff000000) / 16777216,fptr);
265     putc((Offset & 0x00ff0000) / 65536,fptr);
266     putc((Offset & 0x0000ff00) / 256,fptr);
267     putc((Offset & 0x000000ff),fptr);
268
269     // Compression flag, short int
270     // "01 03 00 03 00 00 00 01 00 01 00 00"
271     putc(1,fptr);
272     putc(3,fptr);
273     putc(0,fptr);
274     putc(3,fptr);
275     putc(0,fptr);
```

```
276     putc(0,fptr);
277     putc(0,fptr);
278     putc(1,fptr);
279     putc(0,fptr);
280     putc(1,fptr);
281     putc(0,fptr);
282     putc(0,fptr);
283
284     // Photometric interpretation tag, short int
285     // "01 06 00 03 00 00 00 01 00 02 00 00"
286     putc(1,fptr);
287     putc(6,fptr);
288     putc(0,fptr);
289     putc(3,fptr);
290     putc(0,fptr);
291     putc(0,fptr);
292     putc(0,fptr);
293     putc(1,fptr);
294     putc(0,fptr);
295     putc(2,fptr);
296     putc(0,fptr);
297     putc(0,fptr);
298
299     // Strip offset tag, long int
300     // "01 11 00 04 00 00 00 01 00 00 00 08"
301     putc(1,fptr);
302     putc(17,fptr);
303     putc(0,fptr);
304     putc(4,fptr);
305     putc(0,fptr);
306     putc(0,fptr);
307     putc(0,fptr);
308     putc(1,fptr);
309     putc(0,fptr);
310     putc(0,fptr);
311     putc(0,fptr);
312     putc(8,fptr);
313
314     // Orientation flag, short int
315     // "01 12 00 03 00 00 00 01 00 01 00 00"
316     putc(1,fptr);
317     putc(18,fptr);
318     putc(0,fptr);
319     putc(3,fptr);
320     putc(0,fptr);
321     putc(0,fptr);
322     putc(0,fptr);
323     putc(1,fptr);
324     putc(0,fptr);
325     putc(1,fptr);
326     putc(0,fptr);
327     putc(0,fptr);
328
329     // Sample per pixel tag, short int
330     // "01 15 00 03 00 00 00 01 00 03 00 00"
331     putc(1,fptr);
```

```
332     putc(21,fptr);
333     putc(0,fptr);
334     putc(3,fptr);
335     putc(0,fptr);
336     putc(0,fptr);
337     putc(0,fptr);
338     putc(1,fptr);
339     putc(0,fptr);
340     putc(3,fptr);
341     putc(0,fptr);
342     putc(0,fptr);
343
344     // Rows per strip tag, short int
345     // "01 16 00 03 00 00 00 01"
346     putc(1,fptr);
347     putc(22,fptr);
348     putc(0,fptr);
349     putc(3,fptr);
350     putc(0,fptr);
351     putc(0,fptr);
352     putc(0,fptr);
353     putc(1,fptr);
354     fputc((CalculatedNY & 0xff00) / 256,fptr);
355     fputc((CalculatedNY & 0x00ff),fptr);
356     // "0000"
357     putc(0,fptr);
358     putc(0,fptr);
359
360
361     // Strip byte count flag, long int
362     // "01 17 00 04 00 00 00 01"
363     putc(1,fptr);
364     putc(23,fptr);
365     putc(0,fptr);
366     putc(4,fptr);
367     putc(0,fptr);
368     putc(0,fptr);
369     putc(0,fptr);
370     putc(1,fptr);
371     Offset = CalculatedNX * CalculatedNY *3;
372     putc((Offset & 0xff000000) / 16777216,fptr);
373     putc((Offset & 0x00ff0000) / 65536,fptr);
374     putc((Offset & 0x0000ff00) / 256,fptr);
375     putc((Offset & 0x000000ff),fptr);
376
377     // Minimum sample value flag, short int
378     // "01 18 00 03 00 00 00 03"
379     putc(1,fptr);
380     putc(24,fptr);
381     putc(0,fptr);
382     putc(3,fptr);
383     putc(0,fptr);
384     putc(0,fptr);
385     putc(0,fptr);
386     putc(3,fptr);
387     Offset = CalculatedNX * CalculatedNY *3  + 188;
```

```
388     putc((Offset & 0xff000000) / 16777216,fptr);
389     putc((Offset & 0x00ff0000) / 65536,fptr);
390     putc((Offset & 0x0000ff00) / 256,fptr);
391     putc((Offset & 0x000000ff),fptr);
392
393     // Maximum sample value tag, short int
394     // "01 19 00 03 00 00 00 03"
395     putc(1,fptr);
396     putc(25,fptr);
397     putc(0,fptr);
398     putc(3,fptr);
399     putc(0,fptr);
400     putc(0,fptr);
401     putc(0,fptr);
402     putc(3,fptr);
403     Offset = CalculatedNX * CalculatedNY *3 + 194;
404     putc((Offset & 0xff000000) / 16777216,fptr);
405     putc((Offset & 0x00ff0000) / 65536,fptr);
406     putc((Offset & 0x0000ff00) / 256,fptr);
407     putc((Offset & 0x000000ff),fptr);
408
409     // Planar configuration tag, short int
410     // "01 1c 00 03 00 00 00 01 00 01 00 00"
411     putc(1,fptr);
412     putc(28,fptr);
413     putc(0,fptr);
414     putc(3,fptr);
415     putc(0,fptr);
416     putc(0,fptr);
417     putc(0,fptr);
418     putc(1,fptr);
419     putc(0,fptr);
420     putc(1,fptr);
421     putc(0,fptr);
422     putc(0,fptr);
423
424     // Sample format tag, short int
425     // "01 53 00 03 00 00 00 03"
426     putc(1,fptr);
427     putc(83,fptr);
428     putc(0,fptr);
429     putc(3,fptr);
430     putc(0,fptr);
431     putc(0,fptr);
432     putc(0,fptr);
433     putc(3,fptr);
434     Offset = CalculatedNX * CalculatedNY *3 + 200;
435     putc((Offset & 0xff000000) / 16777216,fptr);
436     putc((Offset & 0x00ff0000) / 65536,fptr);
437     putc((Offset & 0x0000ff00) / 256,fptr);
438     putc((Offset & 0x000000ff),fptr);
439
440     // Ende der Directory-Einträge
441     // "00 00 00 00"
442     putc(0,fptr);
443     putc(0,fptr);
```

```
444     putc(0,fptr);
445     putc(0,fptr);
446
447     // Bits for each colour channel
448     // "00 08 00 08 00 08"
449     putc(0,fptr);
450     putc(8,fptr);
451     putc(0,fptr);
452     putc(8,fptr);
453     putc(0,fptr);
454     putc(8,fptr);
455
456     // Minimum value for each component
457     // "00 00 00 00 00 00"
458     putc(0,fptr);
459     putc(0,fptr);
460     putc(0,fptr);
461     putc(0,fptr);
462     putc(0,fptr);
463     putc(0,fptr);
464
465     // Maximum value per channel
466     // "00 ff 00 ff 00 ff"
467     putc(0,fptr);
468     putc(255,fptr);
469     putc(0,fptr);
470     putc(255,fptr);
471     putc(0,fptr);
472     putc(255,fptr);
473
474     // Samples per pixel for each channel
475     // "00 01 00 01 00 01"
476     putc(0,fptr);
477     putc(1,fptr);
478     putc(0,fptr);
479     putc(1,fptr);
480     putc(0,fptr);
481     putc(1,fptr);
482
483     // IFD
484     putc(0,fptr);
485     putc(0,fptr);
486     putc(0,fptr);
487     putc(0,fptr);
488
489     fclose(fptr);
490   }
491
492
493   // Logarithmierung der Intensitaet
494   // Werte zwischen null und 1 werden zu null gesetzt.
495   void CScreen::logIntensityTIFF()
496   {
497     size_t i, j;
498
499     for (j=0; j<CalculatedNY; j++){
```

```
500       for (i=0; i<CalculatedNX; i++){
501         if (I[j][i]<=1.0) { I[j][i] = 0.0; }
502         else I[j][i] = log(I[j][i]);
503       }
504     }
505   }
506
507
508   // Einlesen der Objekt-Daten
509   void CScreen::read(CInput& Input)
510   {
511     status.DestroyI = (bool)Input.readType("bool");
512     output.WriteI   = (char)Input.readType("ausgabe");
513     output.WriteTIFF= (bool)Input.readType("bool");
514     output.LogTIFF  = (bool)Input.readType("bool");
515
516     Input.setChange(&change.changeSize);
517     Input.readFunction(&NX);
518     Input.readFunction(&NY);
519     Input.readFunction(&WidthXS);
520     Input.readFunction(&HeightYS);
521
522     setAllChange();
523   }
524
525
526   // Ausgabe der Schirmdaten in eine Kameraausgabedatei
527   void CScreen::writeElement(COut &KameraOut)
528   {
529     KameraOut << "Screen // Art des optischen Elements\n";
530     KameraOut << status.DestroyI;
531     KameraOut << "// Intensitaetsverteilung nach der Berechnung zerstoeren (JA, NEIN)\n";
532     KameraOut << output.WriteI;
533     KameraOut << "// Ausgabe der unskalierten Intensitaetsveteilung (ASCII, NEIN, JA
    (=BINAER))\n";
534     KameraOut << output.WriteTIFF;
535     KameraOut << "// Ausgabe der Intensitaetsverteilung im TIF-Format\n";
536     KameraOut << output.LogTIFF;
537     KameraOut << "// Soll die Ausgabe im TIF-Format logarithmiert werden (JA, NEIN)\n";
538     KameraOut << CalculatedNX;
539     KameraOut << "// Anzahl der Stuetzstellen in X-Richtung\n";
540     KameraOut << CalculatedNY;
541     KameraOut << "// Anzahl der Stuetzstellen in Y-Richtung\n";
542     KameraOut << CalculatedWidthXS;
543     KameraOut << "// Breite des Schirms in X-Richtung\n";
544     KameraOut << CalculatedHeightYS;
545     KameraOut << "// Hoehe des Schirms in Y-Richtung\n\n";
546   }
```

```
1   // OpticalElement.h: Schnittstelle für die Klasse COpticalElement.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if
    !defined(AFX_OPTICALELEMENT_H__2EBBF981_A917_11D4_9702_FA70CF88542D_
    _INCLUDED_)
6   #define
    AFX_OPTICALELEMENT_H__2EBBF981_A917_11D4_9702_FA70CF88542D__INCLUD
    ED_
7
8
9   #if _MSC_VER > 1000
10  #pragma once
11  #endif // _MSC_VER > 1000
12
13  #ifdef _DEBUG
14  #pragma warning(disable: 4786)
15  #endif
16
17  #include "..\Constants\constant.h"
18
19  #include "CameraElement.h"
20  #include "TransferFunction.h"
21
22
23  class COpticalElement : public CCameraElement
24  {
25  public:
26    COpticalElement();
27    virtual ~COpticalElement();
28
29    // Berechnung von zwei elektrischen Feldern (= 1 Bild)
30    virtual void calc(CElectricalField& EX, CElectricalField& EY) =0;
31    // Ausgabe der Objektdaten
32    virtual void writeElement(COut& KameraOut);
33    // Einlesen der Objekt-Daten
34    virtual void read(CInput& Input);
35
36    // Change-Funktionen zur Steuerung des automatischen Aenderns von
        Objekteigenschaften
37    virtual void syncChange(void);
38    virtual bool set(void);
39    virtual void resetAllChange(void);
40    virtual void setAllChange(void);
41
42    // Abfrage des Wertes der Variablen change.changeSize
43    virtual bool getChangeSize(void) { return change.changeSize; };
44    // Setzen der Werte fuer change.changeH und change.changeT
45    virtual void setChangeHT(bool NewChangeHT);
46
47    // Ausgabe der maximalen Breite des Objektes
48    virtual double getWidthX(void)   =0;
49    // Ausgabe der maximalen Breite des Objektes
50    virtual double getHeightY(void)   =0;
```

```
51
52
53  protected:
54    // Berechnung eines E-Feldes
55    virtual void calc(CElectricalField& E)   =0;
56
57    // Berechnung der vorgegebenen Übertragungsfunktion
58    void calcTransferFunction(CElectricalField& E);
59    // Berechnung einer Approximation der Übertraguungsfunktion
60    void calcApproximation(CElectricalField& E);
61    // Berechnung der Übertragungsfunktion nach der zweiten Rayleigh-Sommerfeld
        Gleichung
62    void calcSecondRS(CElectricalField& E);
63    // Berechnung der Übertragungsfunktion nach der ersten Rayleigh-Sommerfeld
        Gleichung
64    void calcFirstRS(CElectricalField& E);
65    // Berechnung der Übertragungsfunktion nach der Kirchhoffschen Gleichung
66    void calcKirchhoff(CElectricalField& E);
67    // Berücksichtigung der Verschiebung im Frequenzbereich
68    void calcShift(CElectricalField& E);
69
70    // Multiplikkation der Übertragungsfunktion mit dem elektrischen Feld und gleichzeitige
        Normierung der FFT
71    void multiplicationAndNormalization(CElectricalField& E, CTransferFunction& H);
72
73    // Ausschneiden eines Rechtecks
74    void cutRectangle(CElectricalField& E,double _HeightYRec, double _WidthXRec);
75    // Ausschneiden eines Kreises
76    void cutCircle(CElectricalField& E, double _Radius);
77
78    void init(void);
79
80    COut OutEFieldX;      // Ausgabeobjekt fuer das E-Feld mit Polarisierung in x-Richtung
81    COut OutEFieldY;      // Ausgabeobjekt fuer das E-Feld mit Polarisierung in y-Richtung
82    CTransferFunction H;  // Uebertragungsfunktion H
83    double Distance;      // Abstand des optischen Elementes zum darauffolgenden Element
84    double XShift;      // Verschiebung des nachfolgenden optischen Elementes in x-Richtung
85    double YShift;      // Verschiebung des nachfolgenden optischen Elementes in y-Richtung
86    char Modell;          // Berechnungsmethode (kann 'A'=Approximation, 'F'=First-Rayleigh-
                          Sommerfeld, 'S'=Second-Rayleigh-Sommerfeld oder 'K'=Kirchhoff sein
87
88    // Statusvariablen
89    struct {
90      bool DestroyH;      // Zerstoerung der Uebertragungsfunktion H nach der Berechnung
                             (ja=true, nein=false)
91      bool DestroyT;      // Zerstoerung der Transmissionsfunktionsfunktion T nach der
                             Berechnung (ja=true, nein=false)
92      bool HValid;        // Gibt die aktuellen Gueltigkeit der Uebertragungsfunktion H an
                             (true=gueltig, false=ungueltig)
93      bool TValid;        // Gibt die aktuellen Gueltigkeit der Transmissionsfunktion T an
                             (true=gueltig, false=ungueltig)
94    } status;
95
96    // Ausgabevariablen
97    struct {
98      char WriteE;        // Angabe, ob und wie die elektrischen Felder ausgegeben werden
                             sollen (a,b,n)
```

```
 99      } output;
100
101     // Change-Variablen
102     struct {
103       bool change;        // Angabe, ob irgendetwas an dem Objekt (Element) veraendert
                                wurde
104       bool changeH;       // Angabe, ob etwas veraendert wurde, was nur die
                                Uebertragungsfunktion beeinflusst
105       bool changeT;       // Angabe, ob etwas veraendert wurde, was nur die
                                Transmissionsfunktion beeinflusst
106       bool changeSize;    // Angabe, ob etwas an der Groesse des Objektes veraendert wurde
107     } change;
108   };
109
110   #endif //
      !defined(AFX_OPTICALELEMENT_H__2EBBF981_A917_11D4_9702_FA70CF88542D_
      _INCLUDED_)
```

```
1   // OpticalElement.cpp: Implementierung der Klasse COpticalElement.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "OpticalElement.h"
6
7
8   COpticalElement::COpticalElement(void):CCameraElement()
9   {
10    init();
11  }
12
13
14  COpticalElement::~COpticalElement()
15  {
16
17  }
18
19
20  void COpticalElement::init(void)
21  {
22    Distance = XShift = YShift = 0.0;
23    Modell='\0';
24
25    status.HValid = status.TValid = output.WriteE = false;
26    status.DestroyH = status.DestroyT = true;
27    setAllChange();
28  }
29
30
31  // Zuruecksetzen aller change-Variablen
32  void COpticalElement::resetAllChange(void)
33  {
34    change.change = change.changeH = change.changeT = change.changeSize = false;
35  }
36
37
38  // Setzen aller change-Variablen
39  void COpticalElement::setAllChange(void)
40  {
41    change.change = change.changeH = change.changeT = change.changeSize = true;
42  }
43
44
45  // Veraendern der change-Variablen fuer die Transmissions- und Uebertragungsfunktion
46  // auf einen vorgegebenen Wert
47  void COpticalElement::setChangeHT(bool NewChangeHT)
48  {
49    change.changeH = change.changeT = NewChangeHT;
50  }
51
52
53  // Synchronisierung der change-Variablen untereinander
54  void COpticalElement::syncChange(void)
```

```
55  {
56    // Wenn die Hauptvariable change.change gesetzt ist, ist das gesamte Element
57    // veraendeert worden. Dementsprechend muessen auch die "untergeordneten"
58    // change-Variablen gesetzt werden.
59    if (change.change==true) {
60      change.changeH=true;
61      change.changeT=true;
62      change.changeSize=true;
63    }
64  }
65
66
67  // Setzen der Statusvariablen in folge der gesetzten change-Variablen
68  bool COpticalElement::set(void)
69  {
70    // Es werden in dieser Funktion alle Varianten von gesetzten change-Variablen
71    // berucksichtigt und die entsprechenden Statusvariablen gesetzt.
72    // 1. Hauptvariable change.change ist gesetzt
73    if (change.change==true) {
74      status.HValid=false;
75      status.TValid=false;
76      return true;
77    }
78    // 2. Die Variable change.changeH ist gesetzt
79    if (change.changeH==true) {
80      status.HValid=false;
81      // Gleichzetig kann zusaetzlich noch die change.changeT-Variable gesetzt sein.
82      // Diese tritt bei gleichzeitigem Aendern von mindestens zwei Variablen ein.
83      // Der Fall ist vorhanden sobald in Kamera.dat zwei Schleifen definiert sind,
84      // von denen eine nur die Uebertragungsfunktion und die andere nur die
85      // Transmissionsfunktion beeinflussen.
86      if (change.changeT==true) {
87        status.TValid=false;
88      }
89      return true;
90    }
91    // 3. nur die change-Variable fuer die Transmissionsfunktion ist true.
92    if (change.changeT==true) {
93      status.TValid=false;
94      return true;
95    }
96
97    return false;
98  }
99
100
101 // Auswahl des benötigten Modells zur Berechnung der Übertragungsfunktion
102 void COpticalElement::calcTransferFunction(CElectricalField& E)
103 {
104   // Ueberpruefung, ob die vorhandene Uebertragungsfunktion bereits dieselben
    Eigenschaften
105   // wie das elektrische Feld besitzen.
106   // Wenn "ja" wird die Uebertragungsfunktion nicht erneut berechnet. Wenn "nein" wird
107   // die Stausvarible HValid auf false gesetzt.
108   if ( (H.CField::getNX() != E.CField::getNX()) ||
109        (H.CField::getNY() != E.CField::getNY()) ||
```

```
110         (H.getLambdaH()    != E.getLambda())     ||
111         (H.getHeightYH()    != E.getHeightYE())       ||
112         (H.getWidthXH()    != E.getWidthXE())      ) {
113       status.HValid = false;
114     }
115
116     // Abfrage, ob die aktuelle Uebertragungsfunktion gueltig ist.
117     // Wenn "nein" wird sie sofort berechnet und die Statusvariable auf true gesetzt
118     if ( status.HValid == false ) {
119       H.CField::setSize(E.getNY(), E.getNX(), false, 0, 0);
120       // Berechnung der vorgegebenen Uebertragungsfunktion
121       switch(Modell) {
122         case 'K': calcKirchhoff(E);
123                   break;
124         case 'F': calcFirstRS(E);
125                   break;
126         case 'S': calcSecondRS(E);
127                   break;
128         case 'A': calcApproximation(E);
129                   break;
130       default: ::cerr << "COpticalElement::calcTransferFunction: Es wurde kein Modell
    definiert!";
131       }
132       // Berechnung der Verschiebung
133       if ((XShift!=0)||(YShift!=0)) calcShift(E);
134
135       // Setzen der Eigenschaften der Uebertragungsfunktion
136       H.setLambdaH(E.getLambda());
137       H.setHeightYH(E.getHeightYE());
138       H.setWidthXH(E.getWidthXE());
139       // Setzen der statusvariable nach erfolgreicher Berechnung
140       if ( ((Modell=='K')||(Modell=='F'))  ||  ((Modell=='S')||(Modell=='A')) ) {
141         status.HValid = true;
142       }
143     }
144   }
145
146
147   // Berechnung der Übertragungsfunktion nach Kirchhoff
148   void COpticalElement::calcKirchhoff(CElectricalField& E)
149   {
150     // lokale Variablen
151     size_t i, j;
152     double Temp;
153     double TempSqrt;
154     double TempX;
155     double TempY;
156
157     size_t NX = E.getNX();
158     size_t NY = E.getNY();
159     double DeltaX = E.getDeltaXE();
160     double DeltaY = E.getDeltaYE();
161     double NXTrans = (1.0 - NX)/2.0;    // Hilfsvariable zur Umrechnung der Feldindizes in
                                              reale Dimensionen
162     double NYTrans = (NY-1.0)/2.0;      // Hilfsvariable zur Umrechnung der Feldindizes in
                                              reale Dimensionen
```

```
163     double Lambda = E.getLambda();
164     double K = (2*M_PI) / Lambda;
165     double MinLambda = (-1)*Lambda;
166     double Distance2 = pow(Distance,2);
167
168     // Berechnung der Kirchhoffschen Gleichung =>h
169     // Es wird gleich durch den Faktor jLambda geteilt !!!
170     // Hinweis: Implementierung erfolgt nur für eine gerade Anzahl von Abtastungen
171     for (i=0; i<NX; i++) {
172       for (j=0; j<NY; j++) {
173         // Berechnung der realen Lage fuer den betrachteten Index
174         TempX = (NXTrans + i) *DeltaX;
175         TempY = (NYTrans - j) *DeltaY;
176         // Berechnung der Uebertragungsfunktion h*(1/jLambda)
177         Temp      = Distance2 + (TempX*TempX) + (TempY*TempY);
178         TempSqrt  = sqrt(Temp);
179         Temp      = ((TempSqrt+Distance)/(2*Temp));
180         H(j,i).re = (Temp*sin(K*TempSqrt)) / Lambda;
181         H(j,i).im = (Temp*cos(K*TempSqrt)) / MinLambda;
182       }
183     }
184
185     // Anwendung der FFT => Uebertragungsfunktion H
186     H.Ffft();
187   }
188
189
190   // Berechnung der Übertragungsfunktion nach der ersten Rayleigh-Sommerfeld-Gleichung
191   void COpticalElement::calcFirstRS(CElectricalField& E)
192   {
193     // lokale Variablen
194     size_t i, j;
195     double Temp;
196     double KTempSqrt;
197     double TempX;
198     double TempY;
199
200     size_t NX = E.getNX();
201     size_t NY = E.getNY();
202     double DeltaX = E.getDeltaXE();
203     double DeltaY = E.getDeltaYE();
204     double NXTrans = (1.0-NX)/2.0;      // Hilfsvariable zur Umrechnung der Feldindizes in
    reale Dimensionen
205     double NYTrans = (NY-1.0)/2.0;      // Hilfsvariable zur Umrechnung der Feldindizes in
    reale Dimensionen
206     double Lambda = E.getLambda();
207     double MinLambda = (-1)*Lambda;
208     double K = (2*M_PI) /Lambda;
209     double Distance2 = pow(Distance,2);
210
211     // Berechnung der ersten RS =>h
212     // Es wird gleich durch den Faktor jLambda geteilt !!!
213     // Hinweis: Implementierung erfolgt nur für eine gerade Anzahl von Abtastungen
214     for (i=0; i<NX; i++) {
215       for (j=0; j<NY; j++) {
216         // Berechnung der realen Lage fuer den betrachteten Index
```

```
217         TempX = (NXTrans+i)*DeltaX;
218         TempY = (NYTrans-j)*DeltaY;
219         // Berechnung der Uebertragungsfunktion h*(1/jLambda)
220         Temp  = Distance2 + TempX*TempX + TempY*TempY;
221         KTempSqrt = sqrt(Temp) * K;
222         Temp = Distance / Temp;
223         H(j,i).re = (Temp*sin(KTempSqrt)) / Lambda;
224         H(j,i).im = (Temp*cos(KTempSqrt)) / MinLambda;
225       }
226     }
227
228     // Anwendung der FFT => Uebertragungsfunktion H
229     H.Ffft();
230   }
231
232
233   // Berechnung der Übertragungsfunktion nach der zweiten Rayleigh-Sommerfeld-
      Gleichung
234   void COpticalElement::calcSecondRS(CElectricalField& E)
235   {
236     // lokale Variablen
237     size_t i, j;
238     double TempSqrt;
239     double TempX;
240     double TempY;
241
242     size_t NX = E.getNX();
243     size_t NY = E.getNY();
244     double DeltaX = E.getDeltaXE();
245     double DeltaY = E.getDeltaYE();
246     double NXTrans = (1.0-NX)/2.0;    // Hilfsvariable zur Umrechnung der Feldindizes in
                                             reale Dimensionen
247     double NYTrans = (NY-1.0)/2.0;    // Hilfsvariable zur Umrechnung der Feldindizes in
                                             reale Dimensionen
248     double Lambda = E.getLambda();
249     double MinLambda = (-1)*Lambda;
250     double K = (2*M_PI) / Lambda;
251     double Distance2=pow(Distance,2);
252
253     // Berechnung der zweiten RS =>h
254     // Es wird gleich durch den Faktor jLambda geteilt !!!
255     // Hinweis: Implementierung erfolgt nur für eine gerade Anzahl von Abtastungen
256     for (i=0; i<NX; i++) {
257       for (j=0; j<NY; j++) {
258         // Berechnung der realen Lage fuer den betrachteten Index
259         TempX = (NXTrans+i)*DeltaX;
260         TempY = (NYTrans-j)*DeltaY;
261         // Berechnung der Uebertragungsfunktion h*(1/jLambda)
262         TempSqrt = sqrt (Distance2 + TempX*TempX + TempY*TempY );
263         H(j,i).re =(sin(K*TempSqrt)) / (TempSqrt*Lambda);
264         H(j,i).im =(cos(K*TempSqrt)) / (TempSqrt*MinLambda);
265       }
266     }
267
268     // Anwendung der FFT => Uebertragungsfunktion H
269     H.Ffft();
```

```
270   }
271
272
273   // Berechnung der Übertragungsfunktion mit Hilfe der Näherung
274   void COpticalElement::calcApproximation(CElectricalField& E)
275   {
276     // lokale Variablen
277     size_t i, j;
278     double Temp;
279     double TempX;
280     double TempY;
281
282     size_t NX = E.getNX();
283     size_t NY = E.getNY();
284     double DeltaX = E.getDeltaXE();
285     double DeltaY = E.getDeltaYE();
286     double NXTrans = (1.0-NX)/2.0;          // Hilfsvariable zur Umrechnung der Feldindizes
                                                   in reale Dimensionen
287     double NYTrans = (NY-1.0)/2.0;          // Hilfsvariable zur Umrechnung der Feldindizes
                                                   in reale Dimensionen
288     double DisLambda = Distance*E.getLambda();  // = Abstand zum nächsten
      Element*Lambda
289     double MinDisLambda = (-1)*DisLambda;
290     double TwoDistance2 = 2*pow(Distance,2);
291     double Const   = E.getK() / (2*Distance);
292
293     // Berechnung der Approximation  =>h
294     // Es wird gleich durch den Faktor jLambda geteilt !!!
295     // Hinweis: Implementierung erfolgt nur für eine gerade Anzahl von Abtastungen
296     for (i=0; i<NX; i++) {
297       for (j=0; j<NY; j++) {
298         // Berechnung der realen Lage fuer den betrachteten Index
299         TempX = (NXTrans+i)*DeltaX;
300         TempY = (NYTrans-j)*DeltaY;
301         // Berechnung der Uebertragungsfunktion h*(1/jLambda)
302         Temp  = Const * (TwoDistance2 + TempX*TempX + TempY*TempY);
303         H(j,i).re =(sin(Temp)) /    DisLambda;
304         H(j,i).im =(cos(Temp)) / MinDisLambda;
305       }
306     }
307
308     // Anwendung der FFT => Uebertragungsfunktion H
309     H.Ffft();
310   }
311
312
313   // Berechnung der räumlichen Verschiebung des E-Feldes im Frequenzbereich
314   void COpticalElement::calcShift(CElectricalField& E)
315   {
316     // lokale Variablen
317     size_t i, j;
318     double Temp;
319     double TempHRe;
320     double ShiftRe, ShiftIm;
321
322     size_t NX = E.getNX();
```

```
323     size_t NY = E.getNY();
324     double NXDiv2 = NX/2.0;
325     double NYDiv2 = NY/2.0;
326     double WidthX = E.getWidthXE();
327     double HeightY = E.getHeightYE();
328
329     double TwoPi = 2*M_PI;
330     double XShiftDivWidth = XShift/WidthX;
331     double YShiftDivHeight = -YShift/HeightY;
332
333     // Fuer die Berechnung der Verschiebung des zu berechnenden elektrischen Feldes
334     // muss nach Gleichung (2.50) die Uebertragungsfunktion im Frequenzbereich
335     // mit einer Exponentialfunktion multipliziert werden.
336     // Die Berechnung muss dabei beruecksichtigen, dass die Frequenzen im Feld
337     // aufgrund der FFTW raeumlich anders vorliegen. Genau genommen liegen die
          Frequenzen
338     // "in-order" vor. D.h. das k-te Element entspricht der Frequenz k/n (n= NX oder NY).
339     // Dies bedeutet, dass links oben im Feld (Zweiter Quadrant) die positiven Frequenzen
340     // von der x- und y-Richtung vorliegen. Gleichzeitig liegen die negativen Frequenzen
341     // in den anderen Quadranten in umgekehrter Reihenfolge vor.
342     // Naeheres kann der Beschreibung der FFTW entnommen werden.
343
344     // Berechnung des Bereichs, in dem beide Frequenzen positiv sind (2.Quadrant)
345     for (i=0; i<NXDiv2; i++) {
346       for (j=0; j<NYDiv2; j++) {
347         // Berechnung des Exponentialterms
348         Temp = TwoPi * ( (XShiftDivWidth*i) + (YShiftDivHeight*j) );
349         ShiftRe = cos(Temp);
350         ShiftIm = sin(Temp);
351         // Multiplikation des Real- und Imaginäranteils der Shiftfunktion
352         // mit der Übertragungsfunktion
353         TempHRe = H(j,i).re;
354         H(j,i).re = (H(j,i).re*ShiftRe) - (H(j,i).im*ShiftIm);
355         H(j,i).im = ( TempHRe *ShiftIm) + (H(j,i).im*ShiftRe);
356       }
357     }
358
359     // Berechnung für fx<0 und fy>0 (1.Quadrant)
360     for (i=NXDiv2; i<NX; i++) {
361       for (j=0; j<NYDiv2; j++) {
362         // Berechnung des Exponentialterms
363         Temp = TwoPi * ( (XShiftDivWidth*(i-NX)) + (YShiftDivHeight*j) );
364         ShiftRe = cos(Temp);
365         ShiftIm = sin(Temp);
366         // Multiplikation des Real- und Imaginäranteils der Shiftfunktion
367         // mit der Übertragungsfunktion
368         TempHRe = H(j,i).re;
369         H(j,i).re = (H(j,i).re*ShiftRe) - (H(j,i).im*ShiftIm);
370         H(j,i).im = ( TempHRe *ShiftIm) + (H(j,i).im*ShiftRe);
371       }
372     }
373     // Berechnung für fx>0 und fy<0 (3.Quadrant)
374     for (i=0; i<NXDiv2; i++) {
375       for (j=NYDiv2; j<NY; j++) {
376         // Berechnung des Exponentialterms
377         Temp = TwoPi * ( (XShiftDivWidth*i) + (YShiftDivHeight*(j-NY)) );
```

```
378         ShiftRe = cos(Temp);
379         ShiftIm = sin(Temp);
380         // Multiplikation des Real- und Imaginäranteils der Shiftfunktion
381         // mit der Übertragungsfunktion
382         TempHRe = H(j,i).re;
383         H(j,i).re = (H(j,i).re*ShiftRe) - (H(j,i).im*ShiftIm);
384         H(j,i).im = ( TempHRe *ShiftIm) + (H(j,i).im*ShiftRe);
385       }
386     }
387     // Berechnung für fx<0 und fy<0 (4.Quadrant)
388     for (i=NXDiv2; i<NX; i++) {
389       for (j=NYDiv2; j<NY; j++) {
390         // Berechnung des Exponentialterms
391         Temp = TwoPi * ( (XShiftDivWidth*(i-NX)) + (YShiftDivHeight*(j-NY)) );
392         ShiftRe = cos(Temp);
393         ShiftIm = sin(Temp);
394         // Multiplikation des Real- und Imaginäranteils der Shiftfunktion
395         // mit der Übertragungsfunktion
396         TempHRe = H(j,i).re;
397         H(j,i).re = (H(j,i).re*ShiftRe) - (H(j,i).im*ShiftIm);
398         H(j,i).im = ( TempHRe *ShiftIm) + (H(j,i).im*ShiftRe);
399       }
400     }
401   }
402
403
404   // Multiplikation der Übertragungsfunktion mit dem E-Feld und
405   // gleichzeitige Normierung der FFT
406   void COpticalElement::multiplicationAndNormalization(CElectricalField &E,
      CTransferFunction &H)
407   {
408     // lokale Variablen
409     size_t i,j;
410     double TempRe;
411     size_t NX = E.getNX();
412     size_t NY = E.getNY();
413     double NXNY = (double)(NX*NY);
414     double Lambda = E.getLambda();
415
416     // Nach der Ueberpruefung, dass das elektrische Feld und die Uebertragungsfunktion
417     // dieselben Eigenschaften besitzen, werden die Felder elementweise miteinander
418     // multipliziert und normiert.
419     if( (E.getNX()==H.getNX())          &&  (E.getNY()==H.getNY())          &&
420         (E.getWidthXE()==H.getWidthXH)  &&  (E.getHeightYE()==H.getHeightYH()) &&
421         (E.getLambda()==H.getLambdaH())   ) {
422         for (i=0; i<NX; i++) {
423           for (j=0; j<NY; j++) {
424             TempRe = E(j,i).re;
425             E(j,i).re =( (E(j,i).re * H(j,i).re)  -  (E(j,i).im * H(j,i).im) ) /NXNY;
426             E(j,i).im =( ( TempRe   * H(j,i).im)  +  (E(j,i).im * H(j,i).re) ) /NXNY;
427           }
428         }
429     }
430     else ::cerr <<"CLens::calc(CElectricalField& E): Die Eigenschaften von H und E stimmen
      nicht überein.\n";
431   }
```

```
432
433
434  // Abschneiden des E-Feldes bei einem Durchgang durch eine Rechteckapertur
435  void COpticalElement::cutRectangle(CElectricalField& E,double _HeightYRec, double
     _WidthXRec)
436  {
437    // lokale Variablen
438    size_t i,j;
439    double Tmp;
440    size_t NX = E.getNX();
441    size_t NY = E.getNY();
442    double WidthXE = E.getWidthXE();
443    double HeightYE = E.getHeightYE();
444    double DeltaXE = E.getDeltaXE();
445    double DeltaYE = E.getDeltaYE();
446
447    // Hilfsvariablen
448    double WidthLeftScreen;         // Breite des Schirms links vom Rechteck
449    double BeginnRightScreen;   // Index des ersten Feldes (Elements) rechts des Rechtecks
450    double HeightUpperScreen;      // Höhe des Schirms oberhalb des Rechtecks
451    double BeginnBottomScreen;  // Index des Feldes (Elements) unterhalb des Rechtecks
452    size_t NXBeginnRec;        // Enthält den Index des ersten im Rechteck befindlichen
                                     Feldes (Elements)
453    size_t NXEndRec;            // Enthält den Index des letzten im Rechteck befindlichen
                                     Feldes (Elements)
454    size_t NYBeginnRec;         // Enthält den Index des ersten im Rechteck befindlichen
                                     Feldes (Elements)
455    size_t NYEndRec;           // Enthält den Index des letzten im Rechteck befindlichen
                                     Feldes (Elements)
456
457    // Berechnung der Hilfvariablen:
458    // Hinweis: Die gleiche Berechnung von Hilfsvariablen erfolgt auch in cutCircle!
459    WidthLeftScreen    = (0.5*WidthXE) - (0.5*_WidthXRec);
460    BeginnRightScreen  = WidthLeftScreen + _WidthXRec;
461    HeightUpperScreen   = (0.5*HeightYE) - (0.5*_HeightYRec);
462    BeginnBottomScreen = HeightUpperScreen + _HeightYRec;
463
464    Tmp=fmod(WidthLeftScreen,DeltaXE);
465    if (Tmp<=0.5) {  // Fallunterscheidung, wie weit das Element in das Rechteck hineingeht
466      NXBeginnRec = (size_t)(floor(WidthLeftScreen/DeltaXE));
467    }
468    else {
469      NXBeginnRec = (size_t)(ceil(WidthLeftScreen/DeltaXE));
470    }
471
472    Tmp=fmod(BeginnRightScreen,DeltaXE);
473    if (Tmp<0.5) {  // Fallunterscheidung, wie weit das Element in das Rechteck hineingeht
474      NXEndRec    = (size_t)(floor(BeginnRightScreen/DeltaXE)-1);
475    }
476    else {
477      NXEndRec    = (size_t)(floor(BeginnRightScreen/DeltaXE));
478    }
479
480    Tmp=fmod(HeightUpperScreen,DeltaYE);
481    if (Tmp<=0.5) {  // Fallunterscheidung, wie weit das Element in das Rechteck hineingeht
482      NYBeginnRec = (size_t)(floor(HeightUpperScreen/DeltaYE));
```

```
483     }
484     else {
485       NYBeginnRec = (size_t)(ceil(HeightUpperScreen/DeltaYE));
486     }
487
488     Tmp=fmod(BeginnBottomScreen,DeltaYE);
489     if (Tmp<0.5) {  // Fallunterscheidung, wie weit das Element in das Rechteck hineingeht
490       NYEndRec    = (size_t)(floor(BeginnBottomScreen/DeltaYE)-1);
491     }
492     else {
493       NYEndRec    = (size_t)(floor(BeginnBottomScreen/DeltaYE));
494     }
495
496     // Fuer den Fall das eine einfache freie Propagation berechnet wird
497     // Breite des Rechtecks ist mit der Groesse des elektrischen Feldes identisch
498     if ( (NXBeginnRec<0) || (NXBeginnRec>NX-1) ) { NXBeginnRec=0; }
499     if ( (NYBeginnRec<0) || (NYBeginnRec>NY-1) ) { NYBeginnRec=0; }
500     if (   (NXEndRec<0)  ||   (NXEndRec>NX-1)  )  { NXEndRec=NX-1; }
501     if (   (NYEndRec<0)  ||   (NYEndRec>NX-1)  )  { NYEndRec=NY-1; }
502
503     // Eliminierung des Anteils, der sich links vom Rechteck befindet
504     for (i=0; i<NXBeginnRec ; i++) {
505       // Manipulation der gesamten Spalte
506       for(j=0; j<NY; j++) {
507         E(j,i).re = 0.0;
508         E(j,i).im = 0.0;
509       }
510     }
511     // Eliminierung des Anteils, der sich rechts vom Rechteck befindet
512     for (i=NXEndRec+1; i<NX ; i++) {
513       // Manipulation der gesamten Spalte
514       for(j=0; j<NY; j++) {
515         E(j,i).re = 0.0;
516         E(j,i).im = 0.0;
517       }
518     }
519     // Eliminierung des Anteils, der sich direkt ober- und unterhalb des
520     // Rechtecks befindet
521     for (i=NXBeginnRec; i<=NXEndRec ; i++) {
522       // Manipulation der Spalte oberhalb des Rechtecks
523       for(j=0; j<NYBeginnRec; j++) {
524         E(j,i).re = 0.0;
525         E(j,i).im = 0.0;
526       }
527       // Manipulation der Spalte unterhalb des Rechtecks
528       for(j=NYEndRec+1; j<NY; j++) {
529         E(j,i).re = 0.0;
530         E(j,i).im = 0.0;
531       }
532     }
533   }
534
535
536   // Abschneiden des E-Feldes bei einem Durchgang durch eine Kreisapertur
537   void COpticalElement::cutCircle(CElectricalField &E, double _Radius)
538   {
```

```
539     // lokale Variablen
540     size_t i,j;
541     double TempX;
542     double TempY;
543     double Tmp;
544     size_t NX = E.getNX();
545     size_t NY = E.getNY();
546     double NXTrans = (1.0-NX)/2.0;  //Hilfe zur Transformation der Indizes in reale
        Dimensionen
547     double NYTrans = (NY-1.0)/2.0;  //Hilfe zur Transformation der Indizes in reale
        Dimensionen
548     double WidthXE = E.getWidthXE();
549     double HeightYE = E.getHeightYE();
550     double DeltaXE = E.getDeltaXE();
551     double DeltaYE = E.getDeltaYE();
552     double Radius2 = _Radius * _Radius;
553
554     // Zuerst werden alle Elemente zu Null gesetzt, die sich außerhalb eines
555     // umschreibenden Quadrates befinden. Anschließend wird innerhalb des Quadrates
556     // jeder Punkt einzeln überprüft, ob er sich inner- oder außerhalb des Kreises
557     // befindet.
558     cutRectangle(E, 2*_Radius, 2*_Radius);
559
560
561     // Hilfsvariablen fuer die Definition des umschreibenden Quadrates
562     double WidthLeftScreen;     // Breite des Schirms links vom Kreis
563     double BeginnRightScreen;   // Index des ersten Feldes rechts vom Kreis
564     double HeightUpperScreen;    // Breite des Schirms oberhalb des Kreises
565     double BeginnBottomScreen;  // Index des ersten Feldes unterhalb des Kreises
566     size_t NXBeginnQ;          // Enthält den Index des ersten im Quadrat befindlichen Feldes
567     size_t NXEndQ;             // Enthält den Index des letzten im Quadrat befindlichen Feldes
568     size_t NYBeginnQ;          // Enthält den Index des letzten im Quadrat befindlichen Feldes
569     size_t NYEndQ;             // Enthält den Index des letzten im Quadrat befindlichen Feldes
570
571     // Zunaechts wird das umschreibende Rechteck
572     // Berechnung der Hilfsvariablen:
573     // Hinweis: Die gleiche Berechnung von Hilfsvariablen erfolgt auch in cutRectangle!
574     HeightUpperScreen   = (0.5*HeightYE) - _Radius;
575     BeginnBottomScreen = HeightUpperScreen + (2*_Radius);
576     WidthLeftScreen    = (0.5*WidthXE) - _Radius;
577     BeginnRightScreen  = WidthLeftScreen + (2*_Radius);
578
579     Tmp=fmod(WidthLeftScreen,DeltaXE);
580     if (Tmp<=0.5) {  // Fallunterscheidung, wie weit das Element in das Rechteck hineingeht
581       NXBeginnQ = (size_t)(floor(WidthLeftScreen/DeltaXE));
582     }
583     else {
584       NXBeginnQ = (size_t)(ceil(WidthLeftScreen/DeltaXE));
585     }
586
587     Tmp=fmod(BeginnRightScreen,DeltaXE);
588     if (Tmp<0.5) {  // Fallunterscheidung, wie weit das Element in das Rechteck hineingeht
589       NXEndQ    = (size_t)(floor(BeginnRightScreen/DeltaXE)-1);
590     }
591     else {
592       NXEndQ    = (size_t)(floor(BeginnRightScreen/DeltaXE));
```

```
593     }
594
595     Tmp=fmod(HeightUpperScreen,DeltaYE);
596     if (Tmp<=0.5) {  // Fallunterscheidung, wie weit das Element in das Rechteck hineingeht
597       NYBeginnQ = (size_t)(floor(HeightUpperScreen/DeltaYE));
598     }
599     else {
600       NYBeginnQ = (size_t)(ceil(HeightUpperScreen/DeltaYE));
601     }
602
603     Tmp=fmod(BeginnBottomScreen,DeltaYE);
604     if (Tmp<0.5) {  // Fallunterscheidung, wie weit das Element in das Rechteck hineingeht
605       NYEndQ    = (size_t)(floor(BeginnBottomScreen/DeltaYE)-1);
606     }
607     else {
608       NYEndQ    = (size_t)(floor(BeginnBottomScreen/DeltaYE));
609     }
610
611     // Fuer den Fall das eine einfache freie Propagation berechnet wird.
612     // Radius des Kreises ist groesser als die halbe Feldbreite des elektrischen Feldes
613     if ( (NXBeginnQ<0) || (NXBeginnQ>NX-1) ) { NXBeginnQ=0; }
614     if ( (NYBeginnQ<0) || (NYBeginnQ>NY-1) ) { NYBeginnQ=0; }
615     if (   (NXEndQ<0)  ||   (NXEndQ>NX-1)  )  { NXEndQ=NX-1; }
616     if (   (NYEndQ<0)  ||   (NYEndQ>NX-1)  )  { NYEndQ=NY-1; }
617
618     // Überprüfung jedes einzelnen Punktes im Quadrat, ob er außerhalb oder
619     // direkt auf dem Kreis liegt.
620     for (i=NXBeginnQ; i<=NXEndQ; i++) {
621       for (j=NYBeginnQ; j<=NYEndQ; j++) {
622         TempX = (NXTrans+i)*DeltaXE;
623         TempY = (NYTrans-j)*DeltaYE;
624         if (  (TempX*TempX + TempY*TempY) > Radius2 ) {
625           E(j,i).re = 0.0;
626           E(j,i).im = 0.0;
627         }
628       }
629     }
630   }
631
632
633   //Einlesen der Objekt-Daten
634   void COpticalElement::read(CInput& Input)
635   {
636     Input.setChange(&change.changeH);
637     Modell=(char)Input.readType("modell");
638     Input.readFunction(&Distance);
639     Input.readFunction(&XShift);
640     Input.readFunction(&YShift);
641   }
642
643
644   //Ausgabe der eigenen Objektdaten in die Kameraausgabedatei
645   void COpticalElement::writeElement(COut &KameraOut)
646   {
647     KameraOut << Modell;
648     KameraOut << "// Modell (A, K, F, S)\n";
```

```
649     KameraOut << Distance;
650     KameraOut << "// Abstand zum naechsten Element\n";
651     KameraOut << XShift;
652     KameraOut << "// X-Shift\n";
653     KameraOut << YShift;
654     KameraOut << "// Y-Shift\n";
655   }
```

```
1   // Aperture.h: Schnittstelle für die Klasse CAperture.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if
    !defined(AFX_APERTURE_H__7CEFE1D3_A978_11D4_8960_0050BF19F9FF__INCLU
    DED_)
6   #define
    AFX_APERTURE_H__7CEFE1D3_A978_11D4_8960_0050BF19F9FF__INCLUDED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #include "OpticalElement.h"
13
14
15  class CAperture : public COpticalElement
16  {
17  public:
18    CAperture();
19    virtual ~CAperture();
20
21    // Berechnung von zwei elektrischen Feldern (= 1 Bild)
22    virtual void calc(CElectricalField& EX, CElectricalField& EY);
23
24    // Ausgabe der Objektdaten
25    virtual void writeElement(COut& KameraOut);
26    // Einlesen der Objekt-Daten
27    virtual void read(CInput& Input);
28
29    // get-Funktionen fuer die Ermittlung der raeumlichen Ausdehnung
30    virtual double getWidthX(void) =0;          // Ausgabe der maximalen Breite des Objektes
31    virtual double getHeightY(void) =0;          // Ausgabe der maximalen Hoehe des Objektes
32
33  protected:
34    void init(void) {  };
35
36    // Abschneiden des E-Feldes
37    virtual void cutE(CElectricalField& E)=0;
38    // Berechnung der Propagation eines elektrischen Feldes
39    virtual void calc(CElectricalField& E);
40
41  };
42
43  #endif //
    !defined(AFX_APERTURE_H__7CEFE1D3_A978_11D4_8960_0050BF19F9FF__INCLU
    DED_)
```

```
1   // Aperture.cpp: Implementierung der Klasse CAperture.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "Aperture.h"
6
7
8   CAperture::CAperture():COpticalElement()
9   {
10    init();
11  }
12
13
14  CAperture::~CAperture()
15  {
16
17  }
18
19
20  // Einlesen der Objektdaten
21  void CAperture::read(CInput& Input)
22  {
23    COpticalElement::read(Input);
24  }
25
26
27  // Berechnung von zwei elektrischen Feldern (= 1 Bild) mit eventueller Ausgabe
28  void CAperture::calc(CElectricalField& EX, CElectricalField& EY)
29  {
30    // lokale Variablen zur automatischen Erstellung eines Dateinamens für die Outobjekte
31    char No_ASCII[2];                    // enthaelt den Ort (Stelle) des Elementes
32    char CalcNumber_ASCII[4];            // enthaelt die Bildnummer des Elementes
33    // (generierte Dateinamen duerfen maximal 100 Zeichen enthalten)
34    // erster gleichnamiger Teil des Dateinamens, der unabhaengig vom auszugebenden
        Feld ist
35    char Filename[100]="Aperture_Place";
36    char FilenameX[100];                 // Dateiname fuer das EFeld mit Polarisierung in X-
                                             Richtung
37    char FilenameY[100];                 // Dateiname fuer das EFeld mit Polarisierung in X-
                                             Richtung
38    char *pFilename;                   // Zeiger auf die zu generierenden Dateinamen
39    char *pFilenameX;
40    char *pFilenameY;
41
42
43    ////////////////////////////////////////////
44    // Berechnung der einzelnen elektrischen Felder
45    calc(EX);
46    calc(EY);
47    CalcNumber++;  // Anzahl der berechneten "Bilder" wird um eins erhoeht
48
49    ////////////////////////////////////////////////////////////////////
50    // eventuelle Zerstörung der Übertragungsfunktion zwecks Speicherfreigabe
51    if ( status.DestroyH==true ) {
52      status.HValid = false;
```

```
53      H.CField::setSize(1, 1, false, 0, 0);
54    }
55
56    ////////////////////////////////////////////////////////////////////////
57    // Ausgabe des E-Feldes in eine Datei, sofern es vom Anwender gefordert wird
58    // Generierung des Dateinamens
59    // Zusammensetzung des Namens: "Aperture_PlaceB_EFieldX_CalcNumber"
60    // Dies bedeutet, dass das Kameraelement, welches an der Stelle B sitzt, vom Typ
61    // einer Apertur ist. Die Nummerierung des Ortes (der Stelle) beginnt dabei mit 0
      (=1.Element).
62    // Desweiteren enthaelt diese Datei das elektr. Feld X (kann X oder Y). Schliesslich
63    // gibt die CalcNumber die Nummer des berechneten Bildes an (Zaehlbeginn bei 1).
64    if ( output.WriteE=='b' || output.WriteE=='t') {  // Abfrage ob eine Ausgabe des E-Feldes
                                                           erfolgen soll
65      // Generierung des Dateinamens
66      pFilename=Filename;
67      _itoa(No,No_ASCII,10);                 // Umwandlung der Elementnummer in ein ASCII-
                                                  Zeichen
68      strcat(pFilename,No_ASCII);            // Anhaengen der Elementnummer
69      strcat(pFilename,"_EField");
70      for (int i=0; i<99 ; i++) {
71        FilenameX[i]=FilenameY[i]=Filename[i];
72      }
73      pFilenameX=FilenameX;
74      pFilenameY=FilenameY;
75      strcat(pFilenameX,"X");                // Anhaengen der Spezifizierung fuer das E-Feld
76      strcat(pFilenameY,"Y");
77
78      _itoa(CalcNumber,CalcNumber_ASCII,10);
79      strcat(pFilenameX,CalcNumber_ASCII);     // Anhaengen der Bildnummer
80      strcat(pFilenameY,CalcNumber_ASCII);
81
82      // Definition der Outobjekte für den ASCII-Modus
83      if (output.WriteE=='t') {
84        OutEFieldX.open(FilenameX,"wt");
85        OutEFieldY.open(FilenameY,"wt");
86      }
87      // Definition der Outobjekte für den Binär-Modus
88      if (output.WriteE=='b') {
89        OutEFieldX.open(FilenameX,"wb");
90        OutEFieldY.open(FilenameY,"wb");
91      }
92      // Ausgabe der elektrischen Felder in das entsprechende Outobjekt
93      EX.write(OutEFieldX);
94      EY.write(OutEFieldY);
95
96    }
97  }
98
99
100 // Vollständige Berechnung eines E-Feldes
101 void CAperture::calc(CElectricalField& E)
102 {
103   // Abschneiden des Feldes, der nicht durch den Schirm hindurchgeht
104   cutE(E);
105
```

```
106     // Berechnung der Übertragungsfunktion H
107     calcTransferFunction(E);
108
109     // Durchführung der FFT über das Eingangsfeld
110     E.Ffft();
111
112     // Elementweise Multiplikation von H mit der FFT des Eingangsfeldes
113     // und gleichzeitige Normierung der FFT
114     multiplicationAndNormalization(E, H);
115
116     // Anwendung der inversen FFT auf das oben berechnete ´Eingangssignal´
117     E.Bfft();
118
119     // Vertauschen der 4 Viertel, um die FFT auszugleichen
120     E.ChangeField();
121   }
122
123
124   //Ausgabe der eigenen Objekteigenschaften in die Kameraausgabedatei
125   void CAperture::writeElement(COut &KameraOut)
126   {
127     COpticalElement::writeElement(KameraOut);
128   }
```

```
1   // Rectangle.h: Schnittstelle für die Klasse CRectangle.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if
    !defined(AFX_RECTANGLE_H__AA978420_AA02_11D4_9702_D2702F72A653__INCLU
    DED_)
6   #define
    AFX_RECTANGLE_H__AA978420_AA02_11D4_9702_D2702F72A653__INCLUDED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #include "Aperture.h"
13
14  class CRectangle : public CAperture
15  {
16  public:
17    CRectangle();
18    virtual ~CRectangle();
19
20    virtual void read(CInput& Input);                    // Einlesen der Objekt-Daten
21    virtual void writeElement(COut& KameraOut);  // Ausgabe der Objektdaten
22
23    virtual double getWidthX(void) { return WidthXRec; };  // Ausgabe der maximalen Breite
                                                                des Objektes
24    virtual double getHeightY(void) { return HeightYRec; };  // Ausgabe der maximalen Breite
                                                               des Objektes
25
26
27  protected:
28    virtual void cutE(CElectricalField& E);  // Abschneiden des E-Feldes
29    void init(void);
30
31    double WidthXRec;    // Breite des Rechteckes in x-Richtung
32    double HeightYRec;    // Hoehe des Rechteckes in y-Richtung
33  };
34
35  #endif //
    !defined(AFX_RECTANGLE_H__AA978420_AA02_11D4_9702_D2702F72A653__INCLU
    DED_)
```

```
1   // Rectangle.cpp: Implementierung der Klasse CRectangle.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "Rectangle.h"
6
7
8   CRectangle::~CRectangle()
9   {
10
11  }
12
13
14  CRectangle::CRectangle():CAperture()
15  {
16    init();
17  }
18
19
20  void CRectangle::init(void)
21  {
22    WidthXRec = HeightYRec = 0.0;
23  }
24
25
26  // Einlesen der Objekt-Daten
27  void CRectangle::read(CInput& Input)
28  {
29    output.WriteE = (char)Input.readType("ausgabe");
30    status.DestroyH = (bool)Input.readType("bool");
31
32    CAperture::read(Input);
33
34    Input.setChange(&change.changeSize);
35    Input.readFunction(&WidthXRec);
36    Input.readFunction(&HeightYRec);
37
38    setAllChange();
39  }
40
41
42  // Abschneiden des E-Feldes bei Durchgang durch eine Rechteckapertur
43  void CRectangle::cutE(CElectricalField& E)
44  {
45    cutRectangle(E, HeightYRec, WidthXRec);
46  }
47
48
49  // Ausgabe der Objekteigenschaften in die Kameraausgabedatei
50  void CRectangle::writeElement(COut &KameraOut)
51  {
52    KameraOut << "Rectangle // Art des optischen Elements\n";
53    KameraOut << output.WriteE;
54    KameraOut << "// Ausgabe des EFeldes (ASCII, NEIN, JA (=BINAER))\n";
```

```
55      KameraOut << status.DestroyH;
56      KameraOut << "// Uebertragungsfunktion nach der Berechnung zerstoeren (JA, NEIN)\n";
57
58      CAperture::writeElement(KameraOut);
59
60      KameraOut << WidthXRec;
61      KameraOut << "// Breite der Rechteckapertur\n";
62      KameraOut << HeightYRec;
63      KameraOut << "// Hoehe der Rechteckapertur\n\n";
64    }
```

```
1   // Circle.h: Schnittstelle für die Klasse CCircle.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if
    !defined(AFX_CIRCLE_H__AA978421_AA02_11D4_9702_D2702F72A653__INCLUDED
    _)
6   #define AFX_CIRCLE_H__AA978421_AA02_11D4_9702_D2702F72A653__INCLUDED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #include "Aperture.h"
13
14
15  class CCircle : public CAperture
16  {
17  public:
18    CCircle();
19    virtual ~CCircle();
20
21    virtual void writeElement(COut& KameraOut);    // Ausgabe der Objekteigenschaften
22    virtual void read(CInput& Input);                    // Einlesen der Objekt-Daten
23
24    virtual double getWidthX(void) { return 2*RadiusCircle; };  // Ausgabe der maximalen
                                                                    Breite des Objektes
25    virtual double getHeightY(void) { return 2*RadiusCircle; };  // Ausgabe der maximalen
                                                                    Breite des Objektes
26
27
28  protected:
29    void init(void);
30    virtual void cutE(CElectricalField& E);       // Abschneiden des E-Feldes
31
32    double RadiusCircle;                    // Radius der runden Blende
33  };
34
35  #endif //
    !defined(AFX_CIRCLE_H__AA978421_AA02_11D4_9702_D2702F72A653__INCLUDED
    _)
```

```
1   // Circle.cpp: Implementierung der Klasse CCircle.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "Circle.h"
6
7   //////////////////////////////////////////////////////////////////////
8   // Konstruktion/Destruktion
9   //////////////////////////////////////////////////////////////////////
10
11  CCircle::CCircle():CAperture()
12  {
13    init();
14  }
15
16
17  CCircle::~CCircle()
18  {
19
20  }
21
22
23  void CCircle::init(void)
24  {
25    RadiusCircle = 0.0;
26  }
27
28
29  // Aufruf der benoetigten Funktion zum "Abschneiden" des E-Feldes
30  void CCircle::cutE(CElectricalField& E)
31  {
32    cutCircle(E, RadiusCircle);
33  }
34
35
36  // Einlesen der eigenen Objektdaten
37  void CCircle::read(CInput& Input)
38  {
39    output.WriteE = (char)Input.readType("ausgabe");
40    status.DestroyH = (bool)Input.readType("bool");
41
42    CAperture::read(Input);
43
44    Input.setChange(&change.changeSize);
45    Input.readFunction(&RadiusCircle);
46
47    setAllChange();
48  }
49
50
51  // Schreiben der eigenen Objektdaten in die Kameraausgabedatei
52  void CCircle::writeElement(COut &KameraOut)
53  {
54    KameraOut << "Circle // Art des optischen Elements\n";
```

```
55      KameraOut << output.WriteE;
56      KameraOut << "// Ausgabe des EFeldes (ASCII,NEIN,JA(=BINAER))\n";
57      KameraOut << status.DestroyH;
58      KameraOut << "// Uebertragungsfunktion nach der Berechnung zerstoeren (JA,NEIN)\n";
59
60      CAperture::writeElement(KameraOut);
61
62      KameraOut << RadiusCircle;
63      KameraOut << "// Radius der Kreisapertur\n";
64      KameraOut << "\n";
65    }
```

```
1   // Lens.h: Schnittstelle für die Klasse CLens.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if
    !defined(AFX_LENS_H__AA978422_AA02_11D4_9702_D2702F72A653__INCLUDED_)
6   #define AFX_LENS_H__AA978422_AA02_11D4_9702_D2702F72A653__INCLUDED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #include "OpticalElement.h"
13  #include "TransmissionFunction.h"
14
15
16  class CLens : public COpticalElement
17  {
18  public:
19    CLens();
20    virtual ~CLens();
21
22    //Berechnung von zwei elektrischen Feldern (= 1 Bild)
23    virtual void calc(CElectricalField& EX, CElectricalField& EY);
24
25    virtual void writeElement(COut& KameraOut);       // Ausgabe der Objektdaten
26    virtual void read(CInput& Input);                 // Einlesen der Objekt-Daten
27
28    virtual double getWidthX(void)  =0;        // Ausgabe der maximalen Breite des Objektes
29    virtual double getHeightY(void)  =0;        // Ausgabe der maximalen Breite des Objektes
30
31
32  protected:
33    // Multiplikation der Transmissionsfunktion mit dem E-Feld
34    void applyTransmissionFunctionToElectricalField(CElectricalField& E,
    CTransmissionFunction& T);
35    // Berechnung eines E-Feldes
36    virtual void calc(CElectricalField& E);
37
38    // Berechnung der Transmissionsfunktion für die Linse
39    virtual void calcTransmissionFunction(CElectricalField& E) =0;
40    // Abschneiden des E-Feldes
41    virtual void cutE(CElectricalField& E) =0;
42
43    void init(void) { RefractiveIndex = Thickness = 0.0; };
44
45    CTransmissionFunction T;    // Transmissionsfunktion T
46    double RefractiveIndex;     // Brechungsindex der Linse
47    double Thickness;           // Dicke der Linse (Wird nur für die Berechnung der konstanten
                                     Phasendrehung bei der Transmission benötigt.)
48  };
49  #endif //
    !defined(AFX_LENS_H__AA978422_AA02_11D4_9702_D2702F72A653__INCLUDED_)
```

```
1   // Lens.cpp: Implementierung der Klasse CLens.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "Lens.h"
6
7   //////////////////////////////////////////////////////////////////////
8   // Konstruktion/Destruktion
9   //////////////////////////////////////////////////////////////////////
10
11  CLens::CLens():COpticalElement()
12  {
13    init();
14  }
15
16
17  CLens::~CLens()
18  {
19
20  }
21
22
23  // Verwaltung der Berechnung und der selbständigen Ausgabe des E-Feldes von den
      Objekten
24  void CLens::calc(CElectricalField& EX, CElectricalField& EY)
25  {
26    // lokale Variablen zur automatischen Erstellung eines Dateinamens für die Outobjekte
27    char No_ASCII[2];                    // enthaelt den Ort (Stelle) des Elementes
28    char CalcNumber_ASCII[4];            // enthaelt die Bildnummer des Elements
29    // generierte Dateinamen duerfen maximal 100 Zeichen enthalten
30    // erster gleichnamiger Teil des Dateinamens, der unabhaengig vom auszugebenden
        Feld ist
31    char Filename[100]="Lens_Place";
32    char FilenameX[100];              // Dateiname fuer das E-Feld mit Polarisierung in x-
                                          Richtung
33    char FilenameY[100];              // Dateiname fuer das E-Feld mit Polarisierung in y-
                                          Richtung
34    char *pFilename;                  // Zeiger auf die zu generierenden Dateinamen
35    char *pFilenameX;
36    char *pFilenameY;
37
38    ////////////////////////////////////////////////
39    // Berechnung der einzelnen elektrischen Felder
40    calc(EX);
41    calc(EY);
42    CalcNumber++;   // Anzahl der berechneten "Bilder" wird um eins erhoeht
43
44    //////////////////////////////////////////////////////////////////////
45    // eventuelle Zerstörung der Übertragungs- und der Transmissionsfunktion
46    // (Zweck: Speicherfreigabe!!!)
47    if ( status.DestroyH==true ) {
48      status.HValid = false;
49      H.CField::setSize(1, 1, false, 0, 0);
50    }
51    if ( status.DestroyT==true ) {
```

```
52      status.TValid = false;
53      T.CField::setSize(1, 1, false, 0, 0);
54    }
55
56    ////////////////////////////////////////////////////////////////////////
57    // Ausgabe des E-Feldes in eine Datei sofern es vom Anwender gefordert wird
58    // Generierung des Dateinamens
59    // Zusammensetzung des Namens: "Lens_PlaceB_EFieldX_CalcNumber"
60    // Dies bedeutet, dass das Kameraelement, welches an der Stelle B sitzt, vom Typ
61    // einer Linse ist. Die Nummerierung des Ortes (der Stelle) beginnt dabei mit 0
      (=1.Element).
62    // Desweiteren enthaelt diese Datei das elektr. Feld X (kann X oder Y). Schliesslich
63    // gibt die CalcNumber die Nummer des berechneten Bildes an (Zaehlbeginn bei 1).
64    if ( output.WriteE=='b' || output.WriteE=='t') {   // Abfrage ob eine Ausgabe erfolgen soll
65      // Generierung des Dateinamens
66      pFilename=Filename;
67      _itoa(No,No_ASCII,10);          // Umwandlung der Elementnummer in ein ASCII-Zeichen
68      strcat(pFilename,No_ASCII);                          // Anhaengen der Elementnummer
69      strcat(pFilename,"_EField");
70      for (int i=0; i<99 ; i++) {
71        FilenameX[i]=FilenameY[i]=Filename[i];
72      }
73      pFilenameX=FilenameX;
74      pFilenameY=FilenameY;
75      strcat(pFilenameX,"X");
76      strcat(pFilenameY,"Y");                    // Anhaengen der Spezifitzierung fuer das E-Feld
77
78      _itoa(CalcNumber,CalcNumber_ASCII,10);
79      strcat(pFilenameX,CalcNumber_ASCII);          // Anhaengen der Bildnummer
80      strcat(pFilenameY,CalcNumber_ASCII);
81
82      // Definition der Outobjekte für den ASCII-Modus
83      if (output.WriteE=='t') {
84        OutEFieldX.open(FilenameX,"wt");
85        OutEFieldY.open(FilenameY,"wt");
86      }
87      // Definition der Outobjekte für den Binär-Modus
88      if (output.WriteE=='b') {
89        OutEFieldX.open(FilenameX,"wb");
90        OutEFieldY.open(FilenameY,"wb");
91      }
92      // Ausgabe der elektrischen Felder in das entsprechende Outobjekt
93      EX.write(OutEFieldX);
94      EY.write(OutEFieldY);
95    }
96  }
97
98
99  // Vollständige Berechnung des E-Feldes bei einem Durchgang durch die Linse und die
    Apertur
100 void CLens::calc(CElectricalField& E)
101 {
102   // Abschneiden des Feldes, der nicht durch den Schirm hindurchgeht
103   cutE(E);
104
105   // Berechnung der Transmissionsfunktion der Linse
```

```
106     calcTransmissionFunction(E);
107     // Multiplikation der Transmissionsfunktion mit dem E-Feld
108     // (Hinweis: Es wird nur dort etwas berechnet, wo das E-Feld nicht abgeschnitten wurde)
109     applyTransmissionFunctionToElectricalField(E, T);
110
111     // Durchführung der FFT über das Eingangsfeld
112     E.Ffft();
113
114     // Berechnung der Übertragungsfunktion H
115     calcTransferFunction(E);
116     // Elementweise Multiplikation von H mit der FFT des Eingangsfeldes
117     // und gleichzeitige Normierung der FFT
118     multiplicationAndNormalization(E, H);
119
120     // Anwendung der inversen FFT auf das oben berechnete ´Eingangssignal´
121     E.Bfft();
122
123     // Vertauschen der 4 Viertel, um die FFT auszugleichen
124     E.ChangeField();
125   }
126
127
128   // Anwendung der Transformationsfunktion auf das E-Feld
129   // (einfache elementweise Multiplikation mit Beruecksichtigung, dass nur
130   // ein Viertel der Transmissionsfunktion berechnet wurde
131   void CLens::applyTransmissionFunctionToElectricalField(CElectricalField& E,
      CTransmissionFunction& T)
132   {
133     // lokale Variablen
134     int i, j;
135     float TempRe;
136
137     size_t NX = E.getNX();
138     size_t NY = E.getNY();
139     size_t NX_1 = NX -1;
140     size_t NY_1 = NY -1;
141     size_t NXDiv2 = NX /2;
142     size_t NYDiv2 = NY /2;
143
144
145     // Multiplikation der Transmissionsfunktion (zweiter Quadrant) mit den
146     // vier Quadranten des elektrischen Feldes.
147     // Hinweis: Es wird nur dort etwas berechnet, wo das E-Feld nicht abgeschnitten wurde.
148     //         Dies ist identisch mit der Forderung, dass entweder der Real- oder
149     //         der Imaginaerteil oder beide ungleich Null sind.
150     // Zweiter Quadrant:
151     for (i=0; i<NXDiv2; i++) {
152       for (j=0; j<NYDiv2; j++) {
153         // Berechnung des E-Feldes erfolgt nur, wenn entweder der Real-
154         // oder der Imaginäranteil ungleich Null ist
155         if ((E(j,i).re!=0) || (E(j,i).im!=0)) {
156           // Berechnung des elektrischen Feldes im zweiten Quadranten.
157           TempRe = E(j,i).re;
158           E(j,i).re = (E(j,i).re * T(j,i).re) - (E(j,i).im * T(j,i).im);
159           E(j,i).im = ( TempRe   * T(j,i).im) + (E(j,i).im * T(j,i).re);
160         }
```

```
161       }
162     }
163
164     // Die weiteren drei Quadranten werden analog zum ersten berechnet. Jedoch
165     // sind die Laufindizes und Laufgrenzen entsprechend angepasst.
166     // Erster Quadrant:
167     for (i=NXDiv2; i<NX; i++) {
168       for (j=0; j<NYDiv2; j++) {
169         if ( (E(j,i).re!=0) || (E(j,i).im!=0) ) {
170           TempRe = E(j,i).re;
171           E(j,i).re = (E(j,i).re * T(j,NX_1-i).re) - (E(j,i).im * T(j,NX_1-i).im);
172           E(j,i).im = (  TempRe  * T(j,NX_1-i).im) + (E(j,i).im * T(j,NX_1-i).re);
173         }
174       }
175     }
176
177     // Dritter Quadrant:
178     for (i=0; i<NXDiv2; i++) {
179       for (j=NYDiv2; j<NY; j++) {
180         if ( (E(j,i).re!=0) || (E(j,i).im!=0) ) {
181           TempRe = E(j,i).re;
182           E(j,i).re = (E(j,i).re * T(NY_1-j,i).re) - (E(j,i).im * T(NY_1-j,i).im);
183           E(j,i).im = (  TempRe  * T(NY_1-j,i).im) + (E(j,i).im * T(NY_1-j,i).re);
184         }
185       }
186     }
187
188     // Vierter Quadrant:
189     for (i=NXDiv2; i<NX; i++) {
190       for (j=NYDiv2; j<NY; j++) {
191         if ( (E(j,i).re!=0) || (E(j,i).im!=0) ) {
192           TempRe = E(j,i).re;
193           E(j,i).re = (E(j,i).re * T(NY_1-j,NX_1-i).re) - (E(j,i).im * T(NY_1-j,NX_1-i).im);
194           E(j,i).im = (  TempRe  * T(NY_1-j,NX_1-i).im) + (E(j,i).im * T(NY_1-j,NX_1-i).re);
195         }
196       }
197     }
198   }
199
200
201   // Einlesen der Objekt-Daten
202   void CLens::read(CInput& Input)
203   {
204     COpticalElement::read(Input);
205
206     Input.setChange(&change.changeT);
207     Input.readFunction(&RefractiveIndex);
208     Input.readFunction(&Thickness);
209   }
210
211
212   // Ausgabe der eigenen Objekteigenschaften in die Kameraausgabedatei
213   void CLens::writeElement(COut &KameraOut)
214   {
215     COpticalElement::writeElement(KameraOut);
216
```

```
217     KameraOut << RefractiveIndex;
218     KameraOut << "// Brechungsindex des Mediums\n";
219     KameraOut << Thickness;
220     KameraOut << "// Dicke der Linse\n";
221   }
```

```
1   // ConvexConcave.h: Schnittstelle für die Klasse CConvexConcave.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if !defined(AFX_CONVEXCONCAVE_H__AA978424_AA02_11D4_9702_D2702F72A653__INCLUDED_)
6   #define AFX_CONVEXCONCAVE_H__AA978424_AA02_11D4_9702_D2702F72A653__INCLUDED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #include "Lens.h"
13
14
15  class CConvexConcave : public CLens
16  {
17  public:
18    CConvexConcave();
19    virtual ~CConvexConcave();
20
21    virtual void read(CInput& Input);                    // Einlesen der Objekt-Daten
22    virtual void writeElement(COut& KameraOut);          // Ausgabe der eigenen Objekteigenschaften
23
24    virtual double getWidthX(void) { return 2*RadiusCon; };  // Ausgabe der maximalen Breite des Objektes
25    virtual double getHeightY(void) { return 2*RadiusCon; };  // Ausgabe der maximalen Breite
                                                               des Objektes
26
27
28  protected:
29    virtual void cutE(CElectricalField& E);              // Abschneiden des E-Feldes
30    virtual void calcTransmissionFunction(CElectricalField& E);  // Berechnung der
                                                                    Transmisionsfunktion
31    void init(void);
32
33    double RadiusCon;                                    // Radius der Linse
34    double FocalLenghtCon;                               // reelle Brennweite
35  };
36
37  #endif // !defined(AFX_CONVEXCONCAVE_H__AA978424_AA02_11D4_9702_D2702F72A653__INCLUDED_)
```

```
1   // ConvexConcave.cpp: Implementierung der Klasse CConvexConcave.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "ConvexConcave.h"
6
7   //////////////////////////////////////////////////////////////////////
8   // Konstruktion/Destruktion
9   //////////////////////////////////////////////////////////////////////
10
11  CConvexConcave::~CConvexConcave()
12  {
13
14  }
15
16
17  CConvexConcave::CConvexConcave():CLens()
18  {
19    init();
20  }
21
22  void CConvexConcave::init(void)
23  {
24    RadiusCon = FocalLenghtCon=0.0;
25  }
26
27
28  // Berechnung der Transmissionsfunktion
29  void CConvexConcave::calcTransmissionFunction(CElectricalField& E)
30  {
31    // Ueberpruefung, ob die vorhandene Transmissionsfunktion bereits dieselben
        Eigenschaften
32    // wie das elektrische Feld besitzen
33    // Wenn "ja" wird die Transmissionsfunktion nicht erneut berechnet. Wenn "nein" wird
34    // die Stausvarible TValid auf false gesetzt.
35    if ( (T.CField::getNX() != (E.CField::getNX()/2) ) ||
36         (T.CField::getNY() != (E.CField::getNY()/2) ) ||
37         (T.getLambdaT()    != E.getLambda()         ) ||
38         (T.getHeightYT()   != E.getHeightYE()       ) ||
39         (T.getWidthXT()    != E.getWidthXE()        )  ) {
40      status.TValid = false;
41    }
42
43    // Abfrage, ob die aktuelle Transmissionsfunktion gueltig ist.
44    // Wenn "nein" wird sie sofort berechnet und die Statusvariable auf true gesetzt
45    if( status.TValid==false ) {
46      // lokale Variablen
47      size_t i,j;
48      float Temp;
49      float TempX;
50      float TempY;
51
52      // =-0.5*K/FocalLenght <= Konstanter Faktor im Exponenten der Transmissionsfunktion
53      float Constant= E.getK() / ( (-2)*FocalLenghtCon );
54      float ConstantTwo = E.getK() * RefractiveIndex * Thickness;  // = knd0 in der
```

Ausarbeitung

```
55      size_t NX = E.getNX();
56      size_t NY = E.getNY();
57      size_t NXDiv2 = NX/2;
58      size_t NYDiv2 = NY/2;
59      size_t NX_1 = NX-1;
60      size_t NY_1 = NY-1;
61      float NXTrans = (1.0-NX)/2.0;                  // Hilfe zur Transformation der Indizes
                                                          in reale Dimensionen
62      float NYTrans = (NY-1.0)/2.0;                  // Hilfe zur Transformation der Indizes
                                                          in reale Dimensionen
63      float DeltaX = E.getDeltaXE();
64      float DeltaY = E.getDeltaYE();
65
66
67      // Berechnung der Transmissionsfunktion für ein Viertel der Fläche
68      // Hinweis: Mathematisch wird hier aus Symmetriegründen nur der zweite
69      //          Quadrant berechnet
70      T.CField::setSize(NYDiv2, NXDiv2, false,0,0);
71
72      for (i=0; i<NXDiv2; i++) {
73        for (j=0; j<NYDiv2; j++) {
74          // Berechnung der Transmissionsfunktion für den zweiten Quadranten
75          // Berechung der realen Lage (dimensionsbehaftet) der Indizes
76          TempX = (NXTrans+i)*DeltaX;  // in x-Richtung
77          TempY = (NYTrans-j)*DeltaY;  // in y-Richtung
78          // Berchnung des Exponenten
79          Temp  = Constant*( TempX*TempX + TempY*TempY ) + ConstantTwo;
80          // Berchnung des Real- und Imaginaerteils aus der Exponentialdarstellung
81          T(j,i).re = cos(Temp);
82          T(j,i).im = sin(Temp);
83        }
84      }
85
86      // Setzen der Eigenschaften der Transmissionsfunktion
87      T.setLambdaT(E.getLambda());
88      T.setHeightYT(E.getHeightYE());
89      T.setWidthXT(E.getWidthXE());
90      // Nach der erfolgreicher Berechnung wird die Statusvariable gesetzt.
91      status.TValid=true;
92    }
93  }
94
95
96  // Aufruf der notwendigen Funktion zum "Abschneiden" des E-Feldes
97  void CConvexConcave::cutE(CElectricalField& E)
98  {
99    cutCircle(E, RadiusCon);
100 }
101
102
103 // Einlesen der Objekt-Daten
104 void CConvexConcave::read(CInput& Input)
105 {
106   output.WriteE = (char)Input.readType("ausgabe");
107   status.DestroyH = (bool)Input.readType("bool");
```

```
108     status.DestroyT = (bool)Input.readType("bool");
109
110     CLens::read(Input);
111
112     Input.setChange(&change.changeT);
113     Input.readFunction(&FocalLenghtCon);
114
115     Input.setChange(&change.changeSize);
116     Input.readFunction(&RadiusCon);
117
118     setAllChange();
119   }
120
121
122   //Ausgabe der eigenen Objekteigenschaften in die Kameraausgabedatei
123   void CConvexConcave::writeElement(COut &KameraOut)
124   {
125     KameraOut << "ConvexConcave // Art des optischen Elementes\n";
126     KameraOut << output.WriteE;
127     KameraOut << "// Ausgabe des EFeldes (ASCII, NEIN, JA (=BINAER))\n";
128     KameraOut << status.DestroyH;
129     KameraOut << "// Uebertragungsfunktion nach der Berechnung zerstoeren (JA, NEIN)\n";
130     KameraOut << status.DestroyT;
131     KameraOut << "// Transmissionsfunktion nach der Berechnung zerstoeren (JA, NEIN)\n";
132
133     CLens::writeElement(KameraOut);
134
135     KameraOut << FocalLenghtCon;
136     KameraOut << "// relle Brennweite\n";
137     KameraOut << RadiusCon;
138     KameraOut << "// Radius der Linse (der Kreisapertur)\n";
139     KameraOut << "\n";
140   }
```

```
1   // Cylinder.h: Schnittstelle für die Klasse CCylinder.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #if
    !defined(AFX_CYLINDER_H__AA978423_AA02_11D4_9702_D2702F72A653__INCLUD
    ED_)
6   #define
    AFX_CYLINDER_H__AA978423_AA02_11D4_9702_D2702F72A653__INCLUDED_
7
8   #if _MSC_VER > 1000
9   #pragma once
10  #endif // _MSC_VER > 1000
11
12  #include "Lens.h"
13
14
15  class CCylinder : public CLens
16  {
17  public:
18    CCylinder();
19    virtual ~CCylinder();
20
21    virtual void read(CInput& Input);            // Einlesen der Objekt-Daten
22    virtual void writeElement(COut& KameraOut);   // Ausgabe der Objektdaten
23
24    virtual double getWidthX(void) { return WidthXCyl; };  // Ausgabe der maximalen Breite
                                                                des Objektes
25    virtual double getHeightY(void) { return HeightYCyl; };  // Ausgabe der maximalen Breite
                                                                des Objektes
26
27
28  protected:
29    virtual void cutE(CElectricalField& E);                  // Abschneiden des E-Feldes
30    virtual void calcTransmissionFunction(CElectricalField& E);  // Berechnung der
                                                                     Transmissionsfunktion
31
32    void init(void);
33
34    double WidthXCyl;       // Breite der Linse in x-Richtung
35    double HeightYCyl;       // Höhe der Linse in y-Richtung
36    double FocalLenghtCyl;  // reelle Brennweite
37    char Orientation;       // Orientierung der Zylinderlinse (kann 'H'=Horizontal und
                                 'V'=Vertikal sein)
38
39  };
40
41  #endif //
    !defined(AFX_CYLINDER_H__AA978423_AA02_11D4_9702_D2702F72A653__INCLUD
    ED_)
```

```
1   // Cylinder.cpp: Implementierung der Klasse CCylinder.
2   //
3   //////////////////////////////////////////////////////////////////////
4
5   #include "Cylinder.h"
6
7   //////////////////////////////////////////////////////////////////////
8   // Konstruktion/Destruktion
9   //////////////////////////////////////////////////////////////////////
10
11  CCylinder::~CCylinder()
12  {
13
14  }
15
16
17  CCylinder::CCylinder():CLens()
18  {
19    init();
20  }
21
22
23  void CCylinder::init(void)
24  {
25    WidthXCyl = HeightYCyl = FocalLenghtCyl = 0.0;
26    Orientation = '\0';
27  }
28
29
30  // Berechnung der Transmissionsfunktion
31  void CCylinder::calcTransmissionFunction(CElectricalField& E)
32  {
33    // Ueberpruefung, ob die vorhandene Transmissionsfunktion bereits dieselben
    Eigenschaften
34    // wie das elektrische Feld besitzen
35    // Wenn "ja" wird die Transmissionsfunktion nicht erneut berechnet. Wenn "nein" wird
36    // die Stausvarible TValid auf false gesetzt.
37    if ( (T.CField::getNX() != (E.CField::getNX()/2) ) ||
38         (T.CField::getNY() != (E.CField::getNY()/2) ) ||
39         (T.getLambdaT()    != E.getLambda()         ) ||
40         (T.getHeightYT()   != E.getHeightYE()       ) ||
41         (T.getWidthXT()    != E.getWidthXE()        )  ) {
42      status.TValid = false;
43    }
44
45    // Abfrage, ob die aktuelle Transmissionsfunktion gueltig ist.
46    // Wenn "nein" wird sie sofort berechnet und die Statusvariable auf true gesetzt
47    if( status.TValid==false ) {
48      // lokale Variablen
```

```
49      size_t i,j;
50      float Temp;
51      float TempX;
52      float TempY;
53
54      float Constant = E.getK() / ( (-2)*FocalLenghtCyl );        // = -0.5*K/FocalLenghtCyl
55      float ConstantTwo = E.getK() * RefractiveIndex * Thickness; // = knd0 in der
                                                                       Ausarbeitung
56      size_t NX = E.getNX();
57      size_t NY = E.getNY();
58      size_t NXDiv2 = NX/2;
59      size_t NYDiv2 = NY/2;
60      size_t NX_1 = NX-1;
61      size_t NY_1 = NY-1;
62      float NXTrans = (1.0-NX)/2.0;    // Hilfe zur Transformation der Indizes in reale
                                            Dimensionen
63      float NYTrans = (NY-1.0)/2.0;    // Hilfe zur Transformation der Indizes in reale
                                            Dimensionen
64      float DeltaX = E.getDeltaXE();
65      float DeltaY = E.getDeltaYE();
66
67
68      // Hinweis: Mathematisch wird hier aus Symmetriegründen nur der zweite Quadrant der
69      //          Transmissionsfunktion berechnet
70      T.CField::setSize(NYDiv2, NXDiv2,false ,0,0);
71
72      // Abfrage, ob die Linse horizontal angeordnet ist
73      // (Symmetrieachse parallel zur x-Achse)
74      if (Orientation=='H'){
75        // Berechnung fuer die horizontale Orientierung
76        for (j=0; j<NYDiv2; j++){
77          // Berechnung der realen Dimension des betrachteten Feldindex
78          TempY = (NYTrans-j)*DeltaY;
79          // Berechnung des Exponenten
80          Temp  = Constant* TempY*TempY + ConstantTwo;
81          // Berechnung des Real- und Imaginaerteils aus der Exponentialdarstellung
82          T(j,0).re = cos(Temp);
83          T(j,0).im = sin(Temp);
84          // Kopieren des Wertes in die gesamte Zeile
85          for (i=1; i<NXDiv2; i++){
86            T(j,i).re = T(j,0).re;
87            T(j,i).im = T(j,0).im;
88          }
89        }
90      }
91      // Berechnung fuer den Fall, dass die Symmetrieachse  parallel zur y-Achse ist
92      else if (Orientation=='V'){
93        // Berechnung für die vertikale Anordnung
94        for (i=0; i<NXDiv2; i++){
95          // Berechnung der realen Dimension des betrachteten Feldindex
```

```
96          TempX = (NXTrans+i)*DeltaX;
97          // Berechnung des Exponenten
98          Temp  = Constant* TempX*TempX;
99          // Berechnung des Real- und Imaginaerteils aus der Exponentialdarstellung
100         T(0,i).re = cos(Temp);
101         T(0,i).im = sin(Temp);
102         // Kopieren des Wertes in die gesamte Spalte
103         for (j=1; j<NYDiv2; j++){
104           T(j,i).re = T(0,i).re;
105           T(j,i).im = T(0,i).im;
106         }
107       }
108     }
109     else ::cerr<<"Cylinder.calcTransmissionFunction: Angabe der Orientierung ist
    fehlerhaft.";
110
111     // Setzen der Eigenschaften der Transmissionsfunktion und der Statusvariablen
112     if ( (Orientation=='V')||(Orientation=='H') ) {
113       T.setLambdaT(E.getLambda());
114       T.setHeightYT(E.getHeightYE());
115       T.setWidthXT(E.getWidthXE());
116       status.TValid=true;
117     }
118   }
119 }
120
121
122 // Aufruf der richtigen Funktion zum "Abschneiden" des E-Feldes
123 void CCylinder::cutE(CElectricalField& E)
124 {
125   cutRectangle(E, HeightYCyl, WidthXCyl);
126 }
127
128
129 // Einlesen der Objekt-Daten
130 void CCylinder::read(CInput& Input)
131 {
132   output.WriteE = (char)Input.readType("ausgabe");
133   status.DestroyH = (bool)Input.readType("bool");
134   status.DestroyT = (bool)Input.readType("bool");
135
136   CLens::read(Input);
137
138   Input.setChange(&change.changeT);
139   Orientation=(char)Input.readType("orientation");
140   Input.readFunction(&FocalLenghtCyl);
141
142   Input.setChange(&change.changeSize);
143   Input.readFunction(&WidthXCyl);
144   Input.readFunction(&HeightYCyl);
```

```
145
146   setAllChange();
147 }
148
149
150 //Ausgabe der eigenen Objekteigenschaften in die Kameraausgabedatei
151 void CCylinder::writeElement(COut &KameraOut)
152 {
153   KameraOut << "Cylinder // Art des optischen Elements\n";
154   KameraOut << output.WriteE;
155   KameraOut << "// Ausgabe des EFeldes (ASCII, NEIN, JA(=BINAER))\n";
156   KameraOut << status.DestroyH;
157   KameraOut << "// Uebertragungsfunktion nach der Berechnung zerstoeren (JA, NEIN)\n";
158   KameraOut << status.DestroyT;
159   KameraOut << "// Transmissionsfunktion nach der Berechnung zerstoeren (JA, NEIN)\n";
160
161   CLens::writeElement(KameraOut);
162
163   KameraOut << Orientation;
164   KameraOut << "// Orientierung (H=Symmetrieachse ist die X-Achse, V)\n";
165   KameraOut << FocalLenghtCyl;
166   KameraOut << "// reelle Brennweite\n";
167   KameraOut << WidthXCyl;
168   KameraOut << "// Breite des Zylinders\n";
169   KameraOut << HeightYCyl;
170   KameraOut << "// Hoehe des Zylinders\n\n";
171 }
```

TypesCamera.txt

```
1   bool
2   "NEIN"N"FALSE"NO"
3   "JA"J"TRUE"YES"Y"
4
5   orientation
6   "HORIZONTAL"H":H
7   "VERTICAL"V":V
8
9   modell
10  "APPROXIMATION"A":A
11  "KIRCHHOFF"K":K
12  "FIRSTRS"FIRST"F":F
13  "SECONDRS"SECOND"S":S
14
15  artdesoptischenelementes
16  "RECHTECK"RECTANGLE"REC"R":R
17  "KREIS"CIRCLE"CIRC"CI"K":C
18  "ZYLINDER"CYLINDER"CYL"CY"Z":Y
19  "CONVEXCONCAVE"XV":O
20  "SCREEN"S":S
21
22  ausgabe
23  "ASCII","A","a":t
24  "BINAER","B","b":b
25  "NEIN"N"FALSE"NO":n
26  "JA"J"TRUE"YES"Y":b
```

Camera.dat

```
1   A) Schleifen
2   1 // Anzahl der Schleifen
3   FOR x=0.000000632 TO 0.000000650 STEP 0.000000002  // Schleife 1
4
5   B) Number of Elements
6   5  // Number of Elements (Integer)
7   JA // Ausgabe der Kameradaten ("JA" oder "NEIN")
8   x // Wellenlaenge (nm)
9
10  B.1) Erstes Element
11  Rectangle // Art des optischen Elements (kann
                "Rectangle","Circle","Cylinder","ConvexConcave" oder "Screen")
12  // Ausgabe des E-Feldes (ASCII,NEIN, JA [Standard ist Binaerausgabe])
13  // Uebertragungsfunktion nach der Berechnung loeschen ("NEIN" oder "JA")
14  // Modell (kann "K"=Kirchhoff, "A"=Approximation, "F"=Erste Rayleigh Sommerfeld oder
              "S"=Zweite Rayleigh Sommerfeld sein)
15  // Abstand zum naechsten Element
16  // X-Shift (Verschiebung des naechsten Elements in X-Richtung)
17  // Y-Shift (Verschiebung des naechsten Elements in Y-Richtung)
18  // Breite der Rechteckapertur
19  // Hoehe der Rechteckapertur
20
21  B.2) Zweites Element
22  Circle // Art des optischen Elements (kann
            "Rectangle","Circle","Cylinder","ConvexConcave" oder "Screen")
23  // Ausgabe des E-Feldes (ASCII,NEIN, JA [Standard ist Binaerausgabe])
24  // Uebertragungsfunktion nach der Berechnung loeschen ("NEIN" oder "JA")
25  // Modell (kann "K"=Kirchhoff, "A"=Approximation, "F"=Erste Rayleigh Sommerfeld oder
              "S"=Zweite Rayleigh Sommerfeld sein)
26  // Abstand zum naechsten Element
27  // X-Shift (Verschiebung des naechsten Elements in X-Richtung)
28  // Y-Shift (Verschiebung des naechsten Elements in Y-Richtung)
29  // Radius der Kreisapertur
30
31  B.3) Drittes Element
32  Cylinder// Art des optischen Elements (kann
              "Rectangle","Circle","Cylinder","ConvexConcave" oder "Screen")
33  // Ausgabe des E-Feldes (ASCII,NEIN, JA [Standard ist Binaerausgabe])
34  // Uebertragungsfunktion nach der Berechnung loeschen ("NEIN" oder "JA")
35  // Transmissionsfunktion nach der Berechnung loeschen ("NEIN" oder "JA")
36  // Modell (kann "K"=Kirchhoff, "A"=Approximation, "F"=Erste Rayleigh Sommerfeld oder
              "S"=Zweite Rayleigh Sommerfeld sein
37  // Abstand zum naechsten Element
38  // X-Shift (Verschiebung des naechsten Elements in X-Richtung)
39  // Y-Shift (Verschiebung des naechsten Elements in Y-Richtung)
40  // Brechungsindex des Mediums
41  // Dicke der Linse
42  // Orientierung (kann "H"=horizontal(Symmetrieachse ist die X-Achse) oder
                     "V"=vertikal(Symmetrieachse ist die Y-Achse) sein)
43  // reelle Brennweite
44  // Breite des Zylinders
45  // Hoehe des Zylinders
46
47  B.4) Viertes Element
```

```
48    ConvexConcave // Art des optischen Elements (kann
                          "Rectangle","Circle","Cylinder","ConvexConcave" oder "Screen")
49    // Ausgabe des E-Feldes (ASCII,NEIN, JA [Standard ist Binaerausgabe])
50    // Uebertragungsfunktion nach der Berechnung loeschen ("NEIN" oder "JA")
51    // Transmissionsfunktion nach der Berechnung loeschen ("NEIN" oder "JA")
52    // Modell (kann "K"=Kirchhoff, "A"=Approximation, "F"=Erste Rayleigh Sommerfeld oder
            "S"=Zweite Rayleigh Sommerfeld sein
53    // Abstand zum naechsten Element
54    // X-Shift (Verschiebung des naechsten Elements in X-Richtung)
55    // Y-Shift (Verschiebung des naechsten Elements in Y-Richtung)
56    // Brechungsindex des Mediums
57    // Dicke der Linse
58    // reelle Brennweite
59    // Radius der Linse (der Kreisapertur)
60
61    B.5) Fueenftes Element
62    Screen // Art des optischen Elements (kann
            "Rectangle","Circle","Cylinder","ConvexConcave" oder "Screen")
63    // Intensitaetsverteilung nach der Ausgabe loeschen ("NEIN" oder "JA")
64    // Ausgabe der Intensitaetsverteilung (ASCII,NEIN, JA [Standard ist die Binaerausgabe])
                                (Hinweis: Es wird die unskalierte Verteilung ausgegeben.)
65    // Ausgabe der Intensitaetsverteilung im TIF-Format ("JA" oder "NEIN")
66    // Soll die Ausgabe im TIF-Format logarithmiert werden ("JA" oder "NEIN")
67    // Anzahl der Stuetzstellen in X-Richtung
68    // Anzahl der Stuetzstellen in Y-Richtung
69    // Breite des Schirms in X-Richtung
70    // Hoehe des Schirms in Y-Richtung
```

Bezeichnungen

1. Lateinische Buchstaben

c	[ms^{-1}]	Lichtgeschwindigkeit in einem Medium
c_o	[ms^{-1}]	Lichtgeschwindigkeit im Vakuum
$d(x,y)$	[m]	Dickenverlauf einer Linse
d_0	[m]	Dicke einer Linse
d_{10}, d_{20}, d_{30}	[m]	Dicken einer aufgeteilten Linse
$\underline{\mathbf{E}}$	[Vm^{-1}]	Elektrisches Feld
$\underline{E}_x$, $\underline{E}_y$	[Vm^{-1}]	Elektrische Felder mit Polarisation in x- und y-Richtung
$\underline{\mathbf{E}}_\Sigma$	[Vm^{-1}]	Elektrisches Feld in der Apertur
$\underline{\mathbf{E}}^-$	[Vm^{-1}]	Elektrisches Feld vor der Anwendung der Transmissionsfunktion
$\underline{\mathbf{E}}^+$	[Vm^{-1}]	Elektrisches Feld nach der Anwendung der Transmissionsfunktion
f	[s^{-1}, m]	Frequenz, Brennweite
G		Hilfsfunktion
$\underline{H}$		Übertragungsfunktion
j		imaginäre Einheit
k	[m^{-1}]	Wellenzahl
n		Brechungsindex
n_L		Brechungsindex einer Linse
n_U		Brechungsindex des Umgebungsmediums um eine Linse
$\mathbf{n}$		Normalenvektor
$\mathbf{r}_{01}$		Radiusvektor von Punkt 0 zu Punkt 1
R_1, R_2	[m]	Oberflächenradien einer Linse
S	[m^2]	Oberfläche
$\underline{t}(x,y)$		Transmissionsfunktion einer Linse
U		Hilfsfunktion
x,y,z	[m]	Ortskoordinaten
x_0, y_0	[m]	Verschiebung des nachfolgenden Kameraelements in x- und y-Richtung
z_0	[m]	Abstand des nachfolgenden Kameraelements

2. Griechische Buchstaben

ε_0	[$CV^{-1}m^{-1}$]	Dielektrizitätskonstante des Vakuums
ε_r		relative Dielektrizitätszahl des Mediums
λ	[m]	Wellenlänge
μ_0	[$VsA^{-1}m^{-1}$]	Permeabilitätskonstante des Vakuums
μ_r		relative Permeabilität des Mediums
ν	[s^{-1}]	Frequenz
ξ,η	[m]	Ortskoordinaten
ξ_S, η_S	[m]	Ortskoordinaten im verschobenen Koordinatensystem
Σ	[m^2]	Oberfläche
ψ		Neigungsfaktor

Literatur

[ALB01] Albracht H-E, Tropea C, Borys M, Damaschke N (2001) Laser Doppler and Phase Doppler Measurement Techniques. Springer, Berlin Heidelberg New York

[AUP97] Martin Aupperle (1997) Die Kunst der objektorientierten Programmierung mit C++ - Exakt Grundlagen und zuverlässige Techniken der Softwareentwicklung. Vieweg

[DUB97] Dubbel (1997) Taschenbuch für den Maschinenbau, Hrgb. Beitz W., Grote K-H, Springer, Berlin Heidelberg New York

[FRI99] http://www.fftw.org

[HEC98] Hecht E (1998) Optics. Addison-Wesley Publ Co, 3rd ed.

[GOO68] Goodman J (1968) Introduction to Fourier Optics. The McGraw-Hill Companies, inc.

[LOU99] Dirk Louis (1999) C/C++ - Referenz & Anwendungen. Markt und Technik Verlag

[PAP67] Papoulis A (1967) Systems and Transforms with Applications in Optics. McGraw-Hill Book Company

[PÉR??] Pérez J-P (??) Optik. Spektrum Akademischer Verlag

[TRO00] Tropea C Skript Moderne optische Meßtechnik in der Strömungsmechanik und Verbrennungstechnik. Technische Universität Darmstadt

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit eigenständig, ohne Hilfe Dritter, durchgeführt und verfasst habe. Weiterhin habe ich die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach aus anderen Dokumenten entnommen worden sind, durch Angabe der Quellen kenntlich gemacht. Ebenso habe ich keine anderen als die angegebenen Hilfsmittel verwendet.

Bruchköbel, 19. April 2001

Jens Ritzert