TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR SOFTWARE- UND MULTIMEDIATECHNIK
PROFESSUR FÜR COMPUTERGRAPHIK UND VISUALISIERUNG
PROF. DR. STEFAN GUMHOLD

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Medieninformatiker

Modellierung und Simulation optischer Strahlführungen

Timon Bätz
(Geboren am 2. November 1985 in Ingolstadt)

Betreuer: Dr. Michael Bussmann

Dresden, 24. Oktober 2012

Aufgabenstellung

Ziel der Diplomarbeit ist die Entwicklung eines virtuellen optischen Baukastens für Hochleistungslaser, der dem Benutzer erlaubt, den Laseraufbau zu visualisieren und das erstellte Modell zu simulieren. Der Baukasten soll es dem Benutzter ermöglichen, möglichst schnell und intuitiv ein Lasermodell durch lineares Aneinanderreihen optischer Komponenten virtuell zu erstellen. Dabei können Elemente, zur besseren Übersichtlichkeit von komplexen Aufbauten, zu Gruppen zusammengefasst werden. Die Simulation des Laserpulses soll auf dem physikalischen Lichtausbreitungsmodell basieren. Somit ist es möglich für jeden Punkt auf dem Pfad die Eigenschaften des Lasers zu berechnen und zu visualisieren.

Im einzelnen sind folgende Teilaufgaben zu lösen:

- Implementierung eines übersichtlichen Laserbaukasten.
- Simulation und Visualisierung nach dem Gaußstrahl-Modell.
- Simulation und Visualisierung mittels Fourieroptik.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Arbeit zum Thema:

Modellierung und Simulation optischer Strahlführungen

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 24. Oktober 2012

Timon Bätz

Kurzfassung

In dieser Arbeit wird eine Simulationssoftware für Hochleistungslaser vorgestellt. Ein virtueller optischer Baukasten gestattet es dem Benutzter durch aneinanderreihen optischer Elemente einfache Laseraufbauten nachzubilden. Diese können anschließend entweder durch das Gaußstrahl-Modell oder die Fourieroptik simuliert werden. Rechenintensive mathematische Operationen werden durch ein Plugin-System ausgelagert, wodurch es möglich, ist plattformspezifische Implementierungen der Plugin-Schnittstelle zu entwickeln. Überdies hinaus wird untersucht wie die parallele Rechenkraft moderner Grafikkarten ausgenutzt werden kann um die komplexen Operationen zu beschleunigen.

Abstract

This diploma thesis presents a simulation software for high performance lasers. The graphical interface provides the user with the ability to string together optical components to build simple beam lines. The propagation of light through the system can be simulated with the gaussian beam model and the fourier optics model. Intensive mathematical operations are sourced out from the main application into plugins. As a consequence of this design it is possible to implement different versions of the plugin interface tailored towards specific system configurations. It is further investigated how the parallel processing power of modern GPUs can be used to improve computation time.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Zur Modellierung und Simulation optischer Strahlführungen fehlen bis heute einfache visuelle Konzepte. Die Verwendung der bisher vorhandenen Software ist zum einen mit hohen Kosten verbunden, zum anderen ist sie für den Benutzer schwer verständlich und bei komplizierten Aufbauten meist sehr unübersichtlich. Die Komplexität eines optischen Systems lässt sich oft durch einen linearen Aufbau abstrakt vereinfachen. Ferner kann durch Gruppierung mehrerer Elemente zu Funktionsgruppen die Übersicht auf das Gesamtsystem weiter verbessert werden. Im Rahmen dieser Arbeit soll ein System entwickelt werden, welches außer den oben genannten Eigenschaften zur Verbesserung der Bedienung die parallelen Berechnungsmöglichkeiten moderner Systeme ausnutzt, um hochwertige Simulationen in Echtzeit zu erstellen.

Die Ausarbeitung ist wie folgt strukturiert: Abschnitt 1.2 erläutert grundlegende Begriffe, welche im Laufe dieser Arbeit benutzt werden. Kapitel 2 erläutert die physikalischen Hintergründe der verwendeten Simulationsmodelle. Kapitel 3 und 4 befassen sich jeweils mit der Konzipierung und Umsetzung von performanten Simulations- und Visualisierungsstrategien, welche in der vorgestellten Anwendung implementiert wurden. Die Validierung der Simulationsergebnisse und die Auswertung der Berechnungszeiten werden in Kapitel 5 und 6 besprochen. Abschließend (Kapitel 7) werden mögliche Erweiterungen und weitere Optimierungen der Anwendung diskutiert.

1.2 Begriffe

Im folgenden Abschnitt sind Begriffe erläutert, die im späteren Verlauf dieser Arbeit benutzt werden. Es soll als Nachschlagekapitel für den Rest der Ausarbeitung dienen.

1.2.1 Beugung

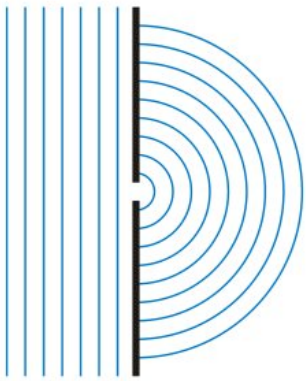

Abbildung 1.1: Beugung einer Welle an einer Öffnung.

Den Effekt der Beugung beobachtet man beim Durchgang von Wellen durch Öffnungen und an Hindernissen. Wegen der Wellennatur des Lichtes, gemäß des Huygensschen Prinzips, kann sich dieses in Raumbereiche ausbreiten, die auf geradem Weg durch das Hindernis versperrt wären (siehe Abbildung 1.1). Nach diesem Prinzip kann jeder Punkt einer Wellenfront als Ausgangspunkt einer sogenannten Elementarwelle (Kugelwellen im dreidimensionalen Raum) betrachtet werden, deren Überlagerung die weiteren Wellenfronten bilden [Mes08].

1.2.2 Interferenz

Interferenz beschreibt die Überlagerung von zwei oder mehr Wellen nach dem Superpositionsprinzip, also die Addition ihrer Amplituden. Löschen sich die Wellen dabei gegenseitig aus, spricht man von destruktiver Interferenz. Verstärken sich die Amplituden, spricht man von konstruktiver Interferenz.

1.2.3 Kohärenz

Vollständige zeitliche Kohärenz liegt vor, wenn die Differenz zwischen der Phase eine Welle gemessen im Zeitpunkt t_1 und die Phase der selben Welle gemessen im Zeitpunkt $t_2 = t_1 + \Delta t$ konstant ist. Monochromatische Wellen sind durch hohe zeitliche Kohärenz charakterisiert.

Vollständige Räumliche Kohärenz liegt vor, wenn entlang der Raumachse zwischen zwei Teilwellen eine feste Phasendifferenz besteht.

1.2.4 Paraxiale Strahlen

Voraussetzung für paraxiale Strahlen ist, dass der Abstand von der optischen Achse zum Strahl klein ist und damit auch nur kleine Winkel mit der optischen Achse vorkommen. Es gilt:

$$\sin \alpha \approx \alpha \tag{1.1}$$

wobei α der Winkel zwischen Strahlverlauf und optischer Achse ist.

1.2.5 Brechungsindex

Im Vakuum breitet sich Licht mit einer Geschwindigkeit von $c_0 = 299792458 \frac{m}{s}$ aus. Der Brechungsindex n eines transparenten homogenen Mediums beschreibt die Geschwindigkeit des Lichts relativ zu c_0. Es gilt:

$$n = \frac{c_0}{c} \geq 1. \tag{1.2}$$

1.2.6 Rayleighlänge

Die Rayleighlänge z_0 ist die Distanz entlang der optischen Achse, die ein Laserstrahl braucht, bis seine Querschnittfläche sich ausgehend von dem Fokus verdoppelt. Der Radius ist dort um den Faktor $\sqrt{2}$ größer.

Wenn man den Laser als Gaußstrahl betrachtet lässt dich die Rayleighlänge wie folgt berechnen:

$$z_0 = \frac{\pi w_0^2}{\lambda} \tag{1.3}$$

wobei w_0 der Radius des Strahls im Fokus und λ die Wellenlänge ist.

1.2.7 Fouriertransformation

Nach der Fourieranalyse kann eine komplexwertige Funktion $f(t)$ als Überlagerung harmonischer Funktionen verschiedener Frequenzen und Amplituden dargestellt werden. Mehrere Funktionen der Form

$\mathcal{F}(\nu)\exp(-i2\pi\nu t)$ (mit komplexer Amplitude $\mathcal{F}(\nu)$ und Frequenz ν) können somit addiert werden um $f(t)$ zu bilden. Es gilt:

$$f(t) = \int_{-\infty}^{\infty} \mathcal{F}(\nu)\exp(-i2\pi\nu t)d\nu. \tag{1.4}$$

Gleichung 1.4 wird als **inverse Fouriertransformation** bezeichnet. Die komplexer Amplitude $\mathcal{F}(\nu)$ kann wiederum durch die **Fouriertransformation** von $f(t)$ berechnet werden:

$$\mathcal{F}(\nu) = \int_{-\infty}^{\infty} f(t)\exp(i2\pi\nu t)dt. \tag{1.5}$$

Analog zu Gleichung 1.4 kann die **zweidimansionale inverse Fouriertransformation** wie folgt definiert werden:

$$f(x,y) = \int_{-\infty}^{\infty} \mathcal{F}(\nu_x,\nu_y)\exp(-i2\pi(\nu_x x + \nu_y y))d\nu_x d\nu_y. \tag{1.6}$$

Demzufolge lautet die **zweidimensionale Fouriertransformation**:

$$\mathcal{F}(\nu_x,\nu_y) = \int_{-\infty}^{\infty} f(x,y)\exp(i2\pi(\nu_x x + \nu_y y))dxdy. \tag{1.7}$$

Die Parameter x und y sind dabei oft die räumlichen Koordinaten des zweidimensionalen Raumes während $\nu_x = k_x/2\pi$ und $\nu_y = k_y/2\pi$ den Ortsfrequenzen in x- und y-Richtung entsprechen.

1.2.8 Faltung

In der Mathematik beschreibt die Faltung eine mathematische Operation auf zwei Funktionen f und g welche eine dritte Funktion liefert. Anschaulich kann die Faltung dadurch beschrieben werden, dass jeder Wert von f durch das mit g gewichtete Mittel der ihn umgebenden Werte ersetzt wird (siehe Abbildung 1.2). Die Faltung zweier Funktionen ist definiert durch:

$$(f * g)(x) = \int_{-\infty}^{\infty} f(\tau)g(x-\tau)d\tau. \tag{1.8}$$

Das **Faltungstheorem** [ST91] besagt das die Fouriertransformierte einer Faltung der punktweisen Multiplikation der einzelnen fouriertransformierten Funktionen entspricht. Somit entspricht die Faltung in den einen Raum einer punktweisen Multiplikation im anderen:

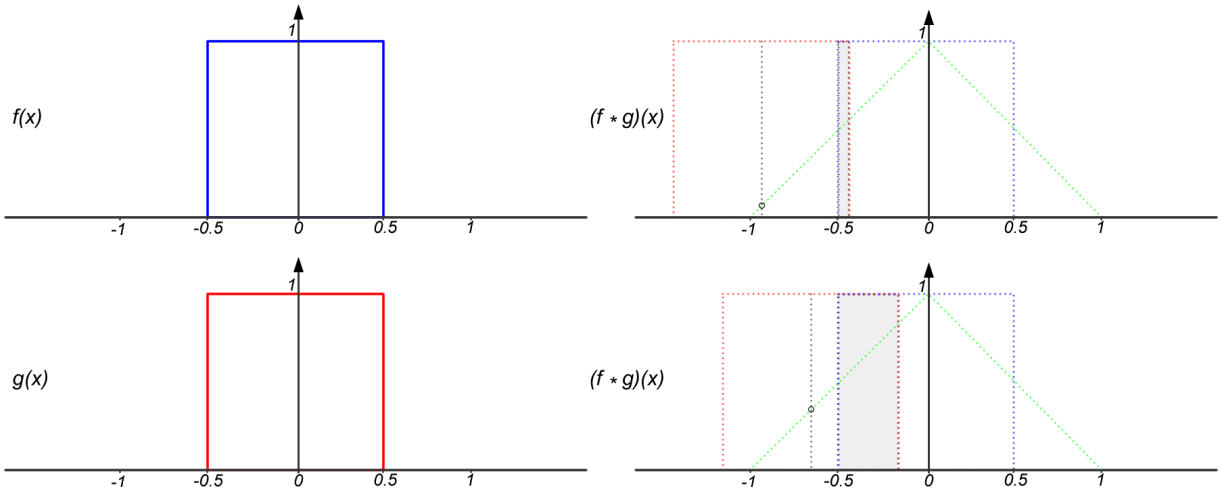

Abbildung 1.2: Veranschaulichung der Faltung einer Funktion f und g. Jeder Punkt der resultierenden Funktion $(f * g)(x)$ entspricht der gemeinsamen Fläche der Funktion f und der um x verschobenen Version von g. Die Ausdehnung dieser Fläche bzw. das Ergebnis der Faltung ist durch die grün gepunktete Linie dargestellt.

$$\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\} \tag{1.9}$$

1.2.9 Lineares System

Nach der **Systemtheorie** ist ein **System** durch eine Eingangsfunktion $f(t)$, einer Ausgangsfunktion $g(t)$ und einer Transformationsregel $g(t) = \mathcal{L}\{f(t)\}$ definiert [ST91]. Erfüllt $\mathcal{L}$ das Superpositionsprinzip, so dass die Antwort des Systems auf eine beliebige Linearkombination zweier Eingangsfunktionen der Linearkombination der einzelnen Antworten entspricht, so wird das System **linear** genannt:

$$\alpha g_1 + \beta g_2 = \alpha\mathcal{L}\{f_1\} + \beta\mathcal{L}\{f_2\} = \mathcal{L}\{\alpha f_1 + \beta f_2\}. \tag{1.10}$$

Die Ausgangsfunktion kann allgemein durch eine gewichtete Überlagerung der Eingangsfunktion an verschiedenen Zeitpunkten τ beschrieben werden [ST91]:

$$g(t) = \mathcal{L}\{f(t)\} = \int_{-\infty}^{\infty} h(t;\tau) f(\tau) d\tau \tag{1.11}$$

wobei $h(t;\tau)$ der Gewichtungsfaktor ist, mit dem die Eingangsfunktion zum Zeitpunkt τ in die Ausgangsfunktion zum Zeitpunkt t einfließt.

Setzt man als Eingangsfunktion des linearen Systems einen Impuls der Form $f(t) = \delta(t - \tau)$ (wobei $\delta(t - \tau)$ die Dirac-Funktion kennzeichnet) ein, erhält man aus Gleichung 1.11 $g(t) = h(t; \tau)$. Die Gewichtungsfunktion wird somit als **Impuls-Antwort-Funktion** bezeichnet.

Ein System wird als **zeitinvariant** bezeichnet, wenn eine zeitliche Verschiebung der Eingangsfunktion eine gleiche Verschiebung der Ausgangsfunktion bewirkt. Demnach hängt die Impulsantwortfunktion nur noch von der Zeitdifferenz $t - \tau$ ab [ST91]. Aus Gleichung 1.11 erhält man:

$$g(t) = \int_{-\infty}^{\infty} h(t - \tau) f(\tau) d\tau. \tag{1.12}$$

Somit entspricht $g(t)$ der Faltung (siehe Abschnitt 1.2.8) der Eingangsfunktion $f(t)$ und der Impulsantwortfunktion $h(t)$. Durch das Faltungstheorem (Gleichung 1.9) erhält man:

$$\mathcal{G}(\nu) = \mathcal{H}(\nu)\mathcal{F}(\nu). \tag{1.13}$$

wobei $\mathcal{G}(\nu)$, $\mathcal{H}(\nu)$ und $\mathcal{F}(\nu)$ den Fouriertransformierten von $g(t)$, $h(t)$ und $f(t)$ entsprechen. $\mathcal{H}(\nu)$ wird als **Übertragungsfunktion** bezeichnet.

2 Physikalische Eigenschaften optischer Strahlführungen

Licht ist elektromagnetische Strahlung. Damit bezeichnet man Wellen aus gekoppelten elektrischen (E) und magnetischen (H) Feldern, welche kein Medium benötigen um sich auszubreiten.

Gegenüber herkömmlichen Lichtquellen zeichnet sich Laserstrahlung durch hohe zeitliche und örtliche Kohärenz (siehe Abschnitt 1.2.3) aus. Solche Wellen weisen ein sehr enges Frequenzspektrum und einen sehr kleinen Divergenzwinkel des Strahls auf. Durch diese Eigenschaften finden sich zahlreiche Anwendungsmöglichkeiten in der Technik und Forschung sowie im täglichen Leben.

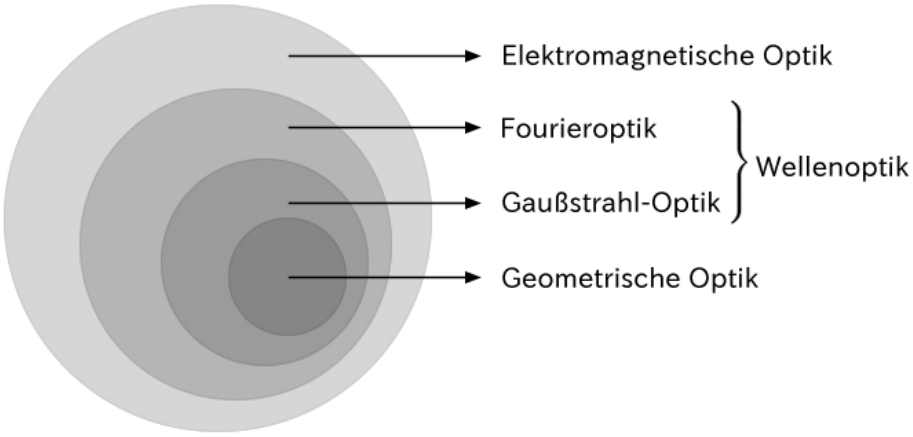

Abbildung 2.1: Die geometrische Optik beschreibt Licht als Strahlen, berücksichtigt jedoch nicht deren wellenartige Natur. In der Wellenoptik wird Licht durch eine skalare Funktion beschrieben, wobei die vektoriellen Eigenschaften des elektrischen und magnetischen Feldes aus der elektromagnetischen Optik vernachlässigt werden.

In der Optik wird eine Vielzahl an mathematischen Modellen für die Berechnung der Ausbreitung eines Lichtstrahls verwendet. Die meisten Theorien sind Vereinfachungen der eigentlichen elektromagnetischen Natur dieser Wellen. Diese Modelle unterscheiden sich anhand der optischen Phänomene die sich damit beschreiben lassen und anhand der Komplexität der mathematischen Berechnung. Welches Modell

für ein gewisses Problem in Frage kommt, hängt von dem Aufbau und der Genauigkeit der geforderten Ergebnisse ab.

In den folgenden Abschnitten werden drei optische Modelle dargestellt. Das Erste, die geometrische Optik, ist zwar das Einfachste, stellt jedoch eine starke Abstraktion der Realität dar, welche Lichtausbreitung als Ausbreitung von Strahlen beschreibt (siehe Abschnitt 2.1). Die Wellenoptik hingegen ist das allgemeinere Modell, mit dem sich optische Phänomene als Wellenphänomäne beschreiben lassen (siehe Abschnitt 2.2). Allgemein können die Ergebnisse aus einem einfacheren auch durch ein komplexeres Modell berechnet werden, jedoch nicht umgekehrt. Ein einfacheres Modell kann wiederum durch Anwendung einer Näherung aus einem komplexeren Modell hergeleitet werden.

2.1 Geometrische Optik

Die geometrische Optik ist das einfachste Modell, das die Ausbreitung von Licht beschreibt. Wenn man elektromagnetische Wellen betrachtet, deren Wellenlängen viel kleiner als die Ausdehnung der wechselwirkenden Strukturen (Spiegel, Linsen, Blenden, ...) sind, können die Welleneigenschaften des Lichtes vernachlässigt werden. Dadurch können jedoch Phänomene aus der Wellenlehre, wie Beugung und Interferenz (siehe Abschnitt 1.2.1, 1.2.2), nicht beschrieben werden. Mathematisch ergibt sich die geometrische Optik aus der Wellenoptik als Grenzfall für verschwindende Wellenlänge $\lambda \to 0$.

In dieser Näherung wird die Propagation von Licht durch Strahlen beschrieben, welche sich geradlinig im Raum ausbreiten. Optische Komponenten bewirken einfache geometrische Transformationen auf die Strahlen. In der Ebene wird solch ein Strahl vollständig durch seinen Startpunkt und einen Winkel definiert (siehe Abbildung 2.2).

$$v_0 = \begin{pmatrix} y_0 \\ \alpha_0 \end{pmatrix} \tag{2.1}$$

Im homogenen Medium (konstanter Brechungsindex n, siehe Abschnitt 1.2.5), nach Durchlaufen der Strecke $L = z_1 - z_0$ und paraxialer Näherung ($sin\alpha = \alpha$, siehe Abschnitt 1.2.4), ändert sich der Strahlvektor v aus Gleichung 2.1 wie folgt:

$$v_1 = \begin{pmatrix} y_1 \\ \alpha_1 \end{pmatrix} = \begin{pmatrix} y_0 + L\alpha_0 \\ \alpha_0 \end{pmatrix}. \tag{2.2}$$

Diese Beziehung lässt sich auch als Matrixoperation darstellen:

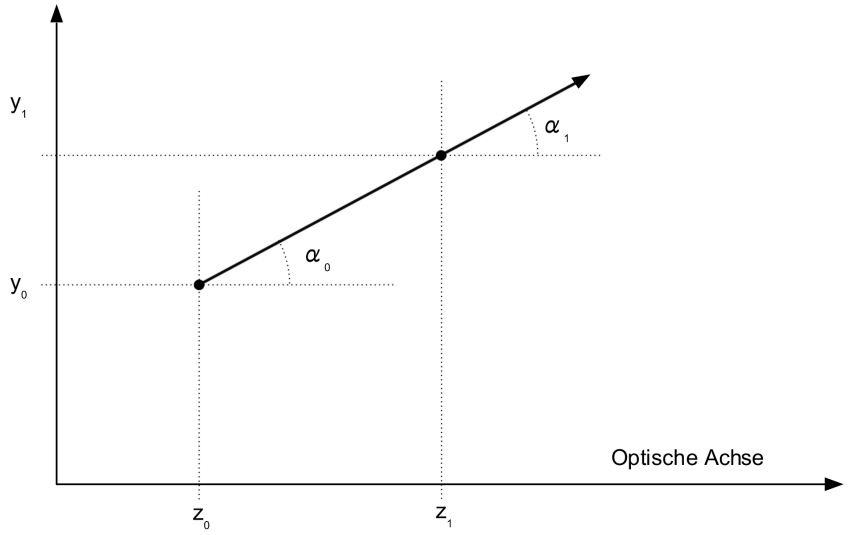

Abbildung 2.2: Beschreibung der freien Lichtstrahlausbreitung nach der Geometrischen Optik.

$$v_1 = \begin{pmatrix} 1 & L \\ 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ \alpha_0 \end{pmatrix}. \tag{2.3}$$

Zu jedem optischen Element lässt sich eine solche 2×2-Matrix definieren, welche die Transformation des Lichtstrahls beschreibt. Diese Beziehung wird als **ABCD-Gesetz** bezeichnet:

$$v_1 = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} y_0 \\ \alpha_0 \end{pmatrix}. \tag{2.4}$$

Beim Übergang eines Lichtstrahls von einem Medium mit Brechungsindex n_1 in eines mit Brechungsindex n_2 wird die Ausbreitungsrichtung des Strahls nach dem Brechungsgesetz [ST91] verändert (siehe Abbildung 2.3):

$$n_1 \sin \alpha_1 = n_2 \sin \alpha_2. \tag{2.5}$$

Unter Berücksichtigung der paraxialen Näherung ($sin\alpha = \alpha$, siehe Abschnitt 1.2.4) lautet das Brechungsgesetz:

$$\alpha_2 \approx \sin \alpha_2 = \frac{n_1}{n_2} \sin \alpha_1 \approx \frac{n_1}{n_2} \alpha_1 \tag{2.6}$$

und somit lautet die Matrix, die diese Brechung beschreibt:

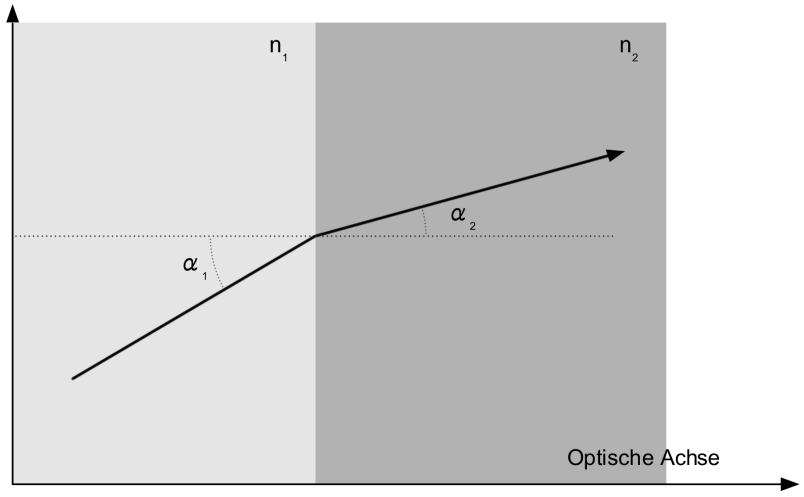

Abbildung 2.3: Brechung eines Strahls beim Übergang von einem Medium mit Brechungsindex n_1 in ein anderes Medium mit Brechungsindex n_2.

$$M = \begin{pmatrix} 1 & 0 \\ 0 & n_1/n_2 \end{pmatrix}. \tag{2.7}$$

Wenn der Übergang zwischen den Medien eine sphärische Fläche mit Krümmungsradius ρ ist (siehe Abbildung 2.4), gilt für die Winkel β_1 und β_2 das Brechungsgesetz aus Gleichung 2.6. Mit:

$$\alpha_1 = \beta_1 - \gamma\,, \quad \alpha_2 = \beta_2 - \gamma \quad \text{und} \quad \gamma = \frac{y_1}{\rho} \tag{2.8}$$

erhält man:

$$\alpha_2 = \frac{n_1 - n_2}{n_2 \rho} y_1 + \frac{n_1}{n_2} \alpha_1, \tag{2.9}$$

woraus sich die Matrix für den sphärischen Übergang ergibt:

$$M = \begin{pmatrix} 1 & 0 \\ \frac{n_1 - n_2}{n_2 \rho} & \frac{n_1}{n_2} \end{pmatrix}. \tag{2.10}$$

Für den Grenzfall $\rho \to \infty$ ist diese mit Gleichung 2.7 äquivalent.

Durchläuft ein Strahl mehrere optische Elemente, wird der Strahlvektor v nacheinander mit den einzelnen Transformationsmatrizen (M_1 bis M_n) multipliziert:

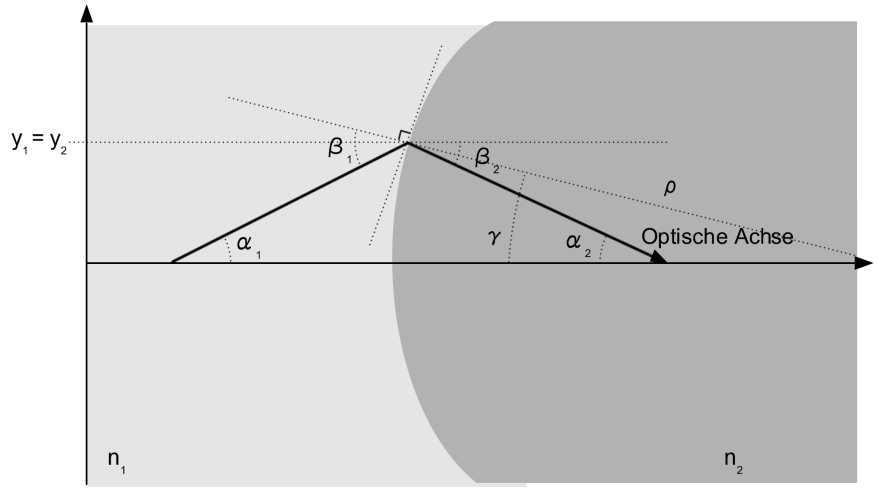

Abbildung 2.4: Brechung eines Strahls beim Übergang von einem Medium mit Brechungsindex n_1 in ein anderes Medium mit Brechungsindex n_2 mit sphärischer Grenzfläche.

$$v_2 = M_1\ v_1$$

$$v_3 = M_2\ v_2$$

$$v_4 = M_3\ v_3$$

$$\vdots$$

$$v_{n+1} = M_n\ v_n$$

Setzt man die Gleichungen ineinander ein, erhält man:

$$v_{n+1} = M_n\ M_{n-1}\ \ldots\ M_3\ M_2\ M_1\ v_1 = M\ v_1.$$

Demnach können komplexere optische Elemente aus diesen Grundmatrizen zusammengestellt und mit einer einzigen $2 \times 2-$ Matrix beschrieben werden. Eine dünne Linse besteht beispielsweise aus zwei sphärischen Grenzübergängen mit Krümmungsradien ρ_1 und ρ_2. Da die Breite einer dünnen Linse vernachlässigbar ist, ergibt sich:

$$M = \begin{pmatrix} 1 & 0 \\ \frac{n_1-n_2}{n_2\rho} & \frac{n_1}{n_2} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \frac{n_2-n_1}{n_1\rho} & \frac{n_2}{n_1} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -1/f & 1 \end{pmatrix}, \tag{2.11}$$

wobei für die Brennweite f:

$$\frac{1}{f} = \frac{n_2 - n_1}{n_1}\left(\frac{1}{\rho_1} + \frac{1}{\rho_2}\right) \tag{2.12}$$

folgt.

2.2 Wellenoptik

Da die Wellenlänge des sichtbaren Lichts viel kleiner als die Ausdehnung der sichtbaren Objekte ist, reicht die Näherung der geometrischen Optik im täglichen Leben oft aus. Will man jedoch die wellenartige Natur des Lichtes beschreiben, greift man zur Wellenoptik. Hierbei handelt es sich immer noch um eine Näherung zur elektromagnetischen Optik. Anstelle der zwei gekoppelten Vektorwellen (elektrisches und magnetisches Feld) [ST91], wird Licht durch eine skalare Wellenfunktion beschrieben, welche die sogenannte **Wellengleichung** erfüllen muss:

$$\nabla^2 E - \frac{1}{c^2}\frac{\partial^2 E}{\partial t^2} = 0 \tag{2.13}$$

Diese lässt sich unter anderem aus den Maxwell-Gleichungen herleiten.

Da es sich bei den Lösungen der Wellengleichung um skalare Funktionen handelt, können mit diesem Modell keine optischen Phänomene beschrieben werden, die eine vektorielle Darstellung des Elektrischen Feldes verlangen (z.B. Polarisationseffekte).

Prinzipiell gibt es unendlich viele Lösungen der Wellengleichung [HW92]. Für monochromatisches Licht ist die Wellenfunktion eine harmonische Funktion der Form:

$$E(\mathbf{r}, t) = a(\mathbf{r}) \cos\left(\omega t + \varphi(\mathbf{r})\right). \tag{2.14}$$

Amplitude $a(\mathbf{r})$ und Phase $\varphi(\mathbf{r})$ sind im allgemeinen ortsabhängig; $\omega = 2\pi f$ stellt die Kreisfrequenz dar.

In den meisten Fällen ist es bequemer, die trigonometrische Funktion durch die komplexe Exponentialfunktion

$$\mathbf{E}(\mathbf{r}, t) = a(\mathbf{r}) e^{i\varphi(\mathbf{r})} e^{i\omega t} \tag{2.15}$$

zu ersetzen, welche ebenfalls die Wellengleichung 2.13 erfüllt. Die physikalische Feldstärke $E(\mathbf{r}, t)$ ergibt sich damit aus dem komplexen Feld $\mathbf{E}(\mathbf{r}, t)$:

$$E(\mathbf{r},t) = Re\{\mathbf{E}(\mathbf{r},t)\} = \frac{1}{2}(\mathbf{E}(\mathbf{r},t) + \mathbf{E}(\mathbf{r},t)^*). \tag{2.16}$$

In Gleichung 2.15 kann nun der zeitabhängige von dem ortsabhängigen Faktor getrennt werden. Man erhält

$$\mathbf{E}(\mathbf{r},t) = \mathbf{E}(\mathbf{r})e^{i\omega t}. \tag{2.17}$$

Die ortsabhängige Funktion $\mathbf{E}(\mathbf{r})$ wird als **komplexe Amplitude** des elektrischen Feldes bezeichnet. Daraus lassen sich mehrere physikalische Eigenschaften der Welle ableiten:

- **Amplitude**: $a(\mathbf{r}) = |\mathbf{E}(\mathbf{r})|$
- **Phase**: $\varphi(\mathbf{r}) = arg\{\mathbf{E}(\mathbf{r})\}$
- **Intensität**: $I(\mathbf{r}) = |\mathbf{E}(\mathbf{r})|^2$

Durch das Einsetzen von Gleichung 2.17 in die Wellengleichung 2.13, erhält man die **Helmholtz-Gleichung**:

$$(\nabla^2 + k^2)\mathbf{E}(\mathbf{r}) = 0, \tag{2.18}$$

wobei:

$$k = \frac{2\pi f}{c} = \frac{2\pi}{\lambda} = \frac{\omega}{c} \tag{2.19}$$

als **Wellenzahl** bezeichnet wird. Anschaulich ist sie die Anzahl der Schwingungen, die die Welle pro Meter durchführt.

Lösungen der Helmholtz-Gleichung beschreiben somit monochromatische Wellen mit harmonischer Zeitabhängigkeit, deren Wellenfunktionen die Wellengleichung 2.13 erfüllen.

Die einfachsten Lösungen der Helmholtz-Gleichung sind die ebene Welle und die Kugelwelle. Beide sind streng genommen nicht realisierbar, liefern jedoch gute Näherungen, mit denen sich viele Phänomene beschreiben lassen (siehe Abbildung 2.5).

Ebene Welle

Für die ebene Welle, welche sich in z-Richtung ausbreitet, lautet die komplexe Amplitude:

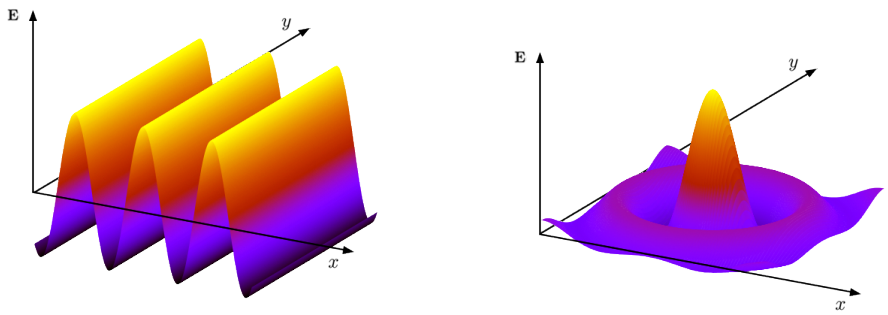

Abbildung 2.5: Links: Darstellung der Intensitätsverteilung einer ebenen Welle; Rechts: Darstellung der Intensitätsverteilung einer Kreiswelle.

$$\mathbf{E}(\mathbf{r}) = \mathbf{A}e^{-ikz}, \tag{2.20}$$

wobei A die konstante **komplexe Einhüllende** ist. Die Intensität dieser Welle, $I(\mathbf{r}) = |\mathbf{A}|^2$, ist in jedem Punkt des Raumes konstant. Somit ist der Energieinhalt der Felder unendlich groß.

Aus Gleichung 2.20 folgt, dass die zugehörige Wellenfunktion sowohl zeitlich als auch räumlich periodisch ist (zeitliche Periode: $1/f$, räumliche Periode: $1/\lambda$).

Kugelwelle

Die komplexe Amplitude der Kugelwelle wird wie folgt berechnet:

$$\mathbf{E}(\mathbf{r}) = \frac{\mathbf{A}}{r}e^{-ikr}, \tag{2.21}$$

wobei r der Betrag des Ortsvektors $\mathbf{r} = (x, y, z)$ ist. Die Intensität $I(\mathbf{r}) = |\mathbf{A}|^2/r^2$ ist hierbei invers proportional zum Quadrat des Abstands r.

2.2.1 Gaußstrahl-Optik

Eine weitere wichtige Lösung der Helmholtz-Gleichung 2.18 ist der Gaußstrahl. Bei dieser Art von Welle ist der Energieinhalt in einem Zylinder um den Mittelpunkt des Strahls konzentriert und die transversale Intensitätsverteilung entspricht einer Gaußfunktion (siehe Abbildung 2.6). Unter idealen Bedingungen entspricht das Licht eines Lasers einem Gaußstrahl.

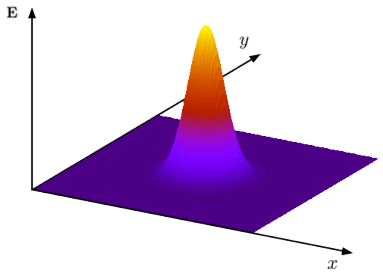

Abbildung 2.6: Transversale Intensitätsverteilung eines Gauß-Strahls.

Um die komplexe Amplitude eines Gaußstrahls herzuleiten, geht man von einer ebenen Welle aus (Gleichung 2.20). Die Konstante $\mathbf{A}$ wird durch eine ortsabhängige Funktion $\mathbf{A}(\mathbf{r})$ ersetzt, welche sich, um die paraxiale Näherung beizubehalten, entlang der Ausbreitungsrichtung z nur langsam verändert. Es gilt:

$$\mathbf{E}(\mathbf{r}) = \mathbf{A}(\mathbf{r})e^{-ikz} \tag{2.22}$$

mit:

$$\frac{\partial^2 \mathbf{A}}{\partial z^2} \approx 0. \tag{2.23}$$

Durch die Funktion $\mathbf{A}(\mathbf{r})$ erhält der Strahl eine ortsabhängige Intensitätverteilung und unterscheidet sich somit von der ebenen Welle.

Durch das Einsetzen von Gleichung 2.22 in die Helmholtz-Gleichung 2.18 unter Berücksichtigung der Näherung aus Gleichung 2.23 erhält man die **paraxiale Helmholtz-Gleichung**:

$$\nabla_T^2 \mathbf{A} - i2k\frac{\partial \mathbf{A}}{\partial z} = 0 \tag{2.24}$$

Eine Lösung dieser Gleichung ist die Gauß-Funktion der Form:

$$\mathbf{A}(\mathbf{r}) = \frac{A_0}{q(z)} e^{-ik\frac{x^2+y^2}{2q(z)}}. \tag{2.25}$$

Die von z abhängige Funktion $q(z) = z + q_0 = z + iz_0$ bezeichnet man als **Strahlparameter**, z_0 als **Rayleigh-Länge** (siehe Abschnitt 1.2.6). Die komplexe Funktion $1/q(z)$ kann, durch Einführen zweier weiterer z-abhängiger Funktionen $w(z)$ und $R(z)$, in Real- und Imaginärteil unterteilt werden:

$$\frac{1}{q(z)} = \frac{1}{z + iz_0} = \frac{1}{R(z)} - i\frac{\lambda}{\pi w^2(z)}. \tag{2.26}$$

$w(z)$ kennzeichnet den Radius bei dem die Amplitude auf $1/e$ und die Intensität auf $1/e^2$ fällt. Der Radius im Fokus wird mit w_0 bezeichnet. Es gilt:

$$w(z) = w_0\sqrt{1 + \left(\frac{z}{z_0}\right)^2}. \tag{2.27}$$

Die Funktion $R(z)$ bestimmt anschaulich wie stark die Wellenfronten gekrümmt sind und wird als **Krümmungsradius** bezeichnet. Außerdem gilt folgende Beziehung:

$$R(z) = z\left(1 + \left(\frac{z_0}{z}\right)^2\right). \tag{2.28}$$

Durch Gleichung 2.22 und 2.25 erhält man einen Ausdruck für die komplexe Amplitude eines Gauß-Strahls [ST91]:

$$\mathbf{E}(\mathbf{r}) = E_0\frac{w_0}{w(z)}\exp\left(-ikz - ik\frac{x^2 + y^2}{2q(z)} + i\zeta(z)\right), \tag{2.29}$$

wobei $E_0 = |E(0)|$ die elektrische Feldstärke in Punkt $r = (0, 0, 0)$ ist.

Ähnlich wie bei der Kugelwelle (Gleichung 2.21) wird der Betrag der elektrischen Feldstärke $|\mathbf{E}(\mathbf{r})|$ durch den Gaußstrahl-Radius skaliert ($E_0 w_0/w(z)$). Vergrößert sich der Durchmesser des Strahles verringert sich die elektrische Feldstärke in jedem Punkt (x, y) der Schnittebene. Die komplexe Amplitude des Gaußstrahls kann aus der ebenen Welle hergeleitet werden (siehe Gleichung 2.21), welche durch eine Gauß-Verteilung moduliert wird. Zusätzlich besitzt der Gaußstrahl eine z-abhängige Phasenverschiebung $\zeta(z) = \arctan(z/z_0)$, die sogenannte **Gouy-Phase**.

Durch Gleichung 2.26 kann 2.29 nun wie folgt umgeschrieben werden:

$$\mathbf{E}(\mathbf{r}) = E_0\frac{w_0}{w(z)}\exp\left(-\frac{x^2 + y^2}{w^2(z)}\right)\exp\left(-ikz - ik\frac{x^2 + y^2}{2R(z)} + i\zeta(z)\right). \tag{2.30}$$

Solange ein Gaußstrahl achsensymmetrische optische Komponenten durchläuft, bleibt dieser ein Gaußstrahl. Es ändern sich jedoch der Strahlradius $w(z)$ und der Krümmungsradius $R(z)$ [ST91]. Da beide Eigenschaften aus dem Strahlparameter $q(z)$ berechnet werden, reicht diese Größe aus, um die Propagation des Gaußstrahls zu beschreiben.

Das für die geometrische Optik hergeleitete **ABCD-Gesetz** lässt sich auf dieses Modell übertragen [ST91]. Komplexe optische Aufbauten können auch hier durch eine einzelne 2×2-Matrix beschrieben

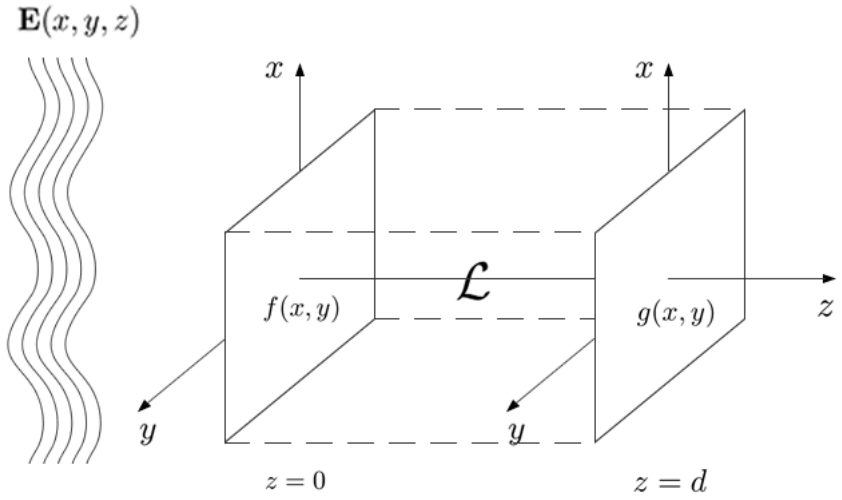

Abbildung 2.7: Ausbreitung einer Welle $\mathbf{E}(x,y,z)$ durch das lineare zeitinvariante System $\mathcal{L}$ mit Eingangsebene $z = 0$, Ausgangsebenen $z = d$, $f(x,y) = \mathbf{E}(x,y,0)$ und $g(x,y) = \mathcal{L}\{f(x,y)\}$.

werden, welche sich durch aneinander Multiplizieren der einzelnen Grundmatrizen ergibt. Diese wendet man jedoch nicht auf den Strahlvektor an, sondern per Multiplikation auf den komplexen Strahlparameter gemäß folgender Vorschrift:

$$q_2 = \frac{Aq_1 + B}{Cq_1 + D}. \tag{2.31}$$

2.2.2 Fourieroptik

Durch die Gaußstrahl-Optik können einfache Laseraufbauten oft gut simuliert werden. Voraussetzung hierbei ist, dass die Intensität des Eingangsstrahls gaußverteilt ist, die Abstände zwischen den Komponenten wesentlich größer als der Strahlradius (paraxial Näherung) und das die optischen Elemente achsensymmetrisch entlang des Strahlverlaufs angeordnet sind. Will man komplexere Läseraufbauten simulieren, reicht das Gaußstrahl-Modell nicht aus.

Die Fourieroptik beschreibt die Ausbreitung von Licht mithilfe von Methoden aus der Fourieranalyse (siehe Abschnitt 1.2.7) und der Systemtheorie (siehe Abschnitt 1.2.9). Eine beliebige Zusammensetzung optischer Elemente kann als lineares zeitinvariantes System $\mathcal{L}$ angesehen werden, welches als Ein- und Ausgabe die komplexen Amplitudenverteilungen auf der xy-Ebene $f(x,y)$ und $g(x,y)$ besitzt (siehe

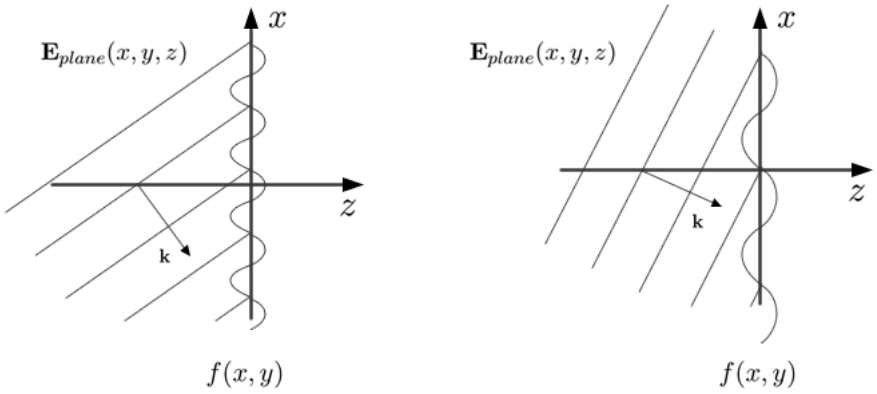

Abbildung 2.8: Projektion einer ebenen Welle auf die (xy)-Ebene. Wellen, die große Winkel mit der z-Achse bilden, projizieren harmonische Funktionen mit kleineren Frequenzen auf die xy-Ebene, als ebene Wellen, die mit der z-Achse kleine Winkel bilden.

Abbildung 2.7). Angenommen die Eingangsebene liegt bei $z = 0$, so entspricht $f(x, y)$ der komplexen Amplitude der eingehenden Welle $\mathbf{E}(x, y, z = 0)$. Die Funktion $g(x, y)$ kann durch Anwenden der Transformation von $\mathcal{L}$ auf $f(x, y)$ berechnet werden: $g(x, y) = \mathcal{L}\{f(x, y)\}$. Das System $\mathcal{L}$ ist linear da die Helmholtz-Gleichung, die $\mathbf{E}(x, y, z)$ erfüllen muss, linear ist. Weil der freie Raum invariant gegenüber Verschiebungen des Koordinatensystems ist, ist $\mathcal{L}$ auch verschiebungsinvariant.

Betrachtet man als Eingangswelle des Systems $\mathcal{L}$ eine ebene Welle

$$\mathbf{E}_{plane}(x, y, z) = \mathbf{A} \exp(-i(k_x x + k_y y + k_z z)) \tag{2.32}$$

mit Wellenvektor $\mathbf{k} = (k_x, k_y, k_z)$, so entspricht $f(x, y)$ einer räumlich harmonischen Funktion (siehe Abbildung 2.8) der Form:

$$f(x, y) = \mathbf{E}_{plane}(x, y, 0) = \mathbf{A} \exp(-i2\pi(\nu_x x + \nu_y y)). \tag{2.33}$$

mit den Ortsfrequenzen $\nu_x = k_x/2\pi$ und $\nu_y = k_y/2\pi$. Die ebene Eingangswelle kann wiederum wie folgt aus $f(x, y)$ berechnet werden:

$$\mathbf{E}_{plane}(x, y, z) = f(x, y) \exp(-ik_z z) \tag{2.34}$$

Betrachtet man als Eingangswelle eine nicht harmonische Funktion, so ist auch $f(x, y) = \mathbf{E}(x, y, 0)$ eine nicht harmonische Funktion. Letztere kann jedoch durch Fouriertransformation als Summe harmonischer Funktionen der Amplituden $\mathcal{F}(\nu_x, \nu_y)$ zerlegt werden:

$$f(x,y) = \iint_{-\infty}^{\infty} \mathcal{F}(\nu_x, \nu_y) \exp(-i2\pi(\nu_x x + \nu_y y)) d\nu_x d\nu_y. \tag{2.35}$$

Da jede harmonische Funktion $\mathcal{F}(\nu_x, \nu_y) \exp(-i2\pi(\nu_x x + \nu_y y))$ der Projektion einer ebenen Welle auf die $(z = 0)$-Ebene entspricht, kann eine beliebige nicht harmonische Eingangswelle $\mathbf{E}(x, y, z)$ als Summe ebener Wellen mit unterschiedlichen Wellenvektoren dargestellt werden [ST91]. Anschaulich wird jede Ortsfrequenz der harmonischen Funktionen durch die Richtung (Wellenvektor $\mathbf{k}$) einer ebenen Welle bestimmt. Ebene Wellen, die große Winkel mit der z-Achse bilden, projizieren harmonische Funktion mit kleineren Frequenz auf die xy-Ebene, als ebene Wellen, die mit der z-Achse kleine Winkel bilden (siehe Abbildung 2.8).

Multipliziert man jede harmonische Funktion im Integral aus Gleichung 2.35 mit $\exp(-ik_z z)$, erhält man die zugehörige ebene Welle (siehe Gleichung 2.34). Es folgt:

$$\mathbf{E}(x,y,z) = \iint_{-\infty}^{\infty} \mathcal{F}(\nu_x, \nu_y) \exp(-i2\pi(\nu_x x + \nu_y y)) \exp(-ik_z z) d\nu_x d\nu_y \tag{2.36}$$

mit:

$$k_z = \sqrt{k^2 - k_x^2 - k_y^2} = 2\pi\sqrt{\frac{1}{\lambda^2} - \nu_x^2 - \nu_y^2}. \tag{2.37}$$

Kennt man die Transformationsgesetze, welche das System $\mathcal{L}$ auf die einzelnen ebenen Wellen anwendet, kann die ausgehende Welle durch Überlagerung der einzelnen transformierten ebenen Wellen berechnet werden.

2.2.2.1 Ausbreitung im freien Raum

Wie im Abschnitt 1.2.9 erläutert, wird jedes zeitinvariantes lineares System durch die zugehörige Impuls-Antwort-Funktion $h(x, y)$ oder die Übertragungsfunktion $\mathcal{H}(\nu_x, \nu_y)$ charakterisiert. Letztere beschreibt den Gewichtungsfaktor, mit dem die harmonischen Teilfunktionen aus der Fourieranalyse von $f(x, y)$ in die Ausgangsfunktion $g(x, y)$ einfließen. Betrachtet man als Eingangswelle von $\mathcal{L}$ eine eben Welle, so ist $f(x, y)$ eine harmonische Funktion, die Gleichung 2.33 entspricht. Besteht das System $\mathcal{L}$ ausschließlich aus freier Propagationsstrecke der Länge d, so gilt:

$$g(x,y) = \mathbf{E}(\mathbf{x}, \mathbf{y}, \mathbf{d}) = \mathbf{A} \exp(-i(k_x x + k_y y + k_z d)). \tag{2.38}$$

In diesem Fall kann die Übertragungsfunktion wie folgt berechnen werden:

$$\mathcal{H}(\nu_x, \nu_y) = \frac{g(x,y)}{f(x,y)} = \exp(-ik_z d) = \exp\left(-i2\pi \sqrt{\frac{1}{\lambda^2} - \nu_x^2 - \nu_y^2}\ d\right). \tag{2.39}$$

Für $\frac{1}{\lambda^2} - \nu_x^2 - \nu_y^2 \geq 0$ beziehungsweise $\nu_x^2 + \nu_y^2 \leq \frac{1}{\lambda^2}$ ist der Betrag der Übertragungsfunktion $|\mathcal{H}(\nu_x, \nu_y)| = 1$, während die Phase $arg\{\mathcal{H}(\nu_x, \nu_y)\}$ abhängig von der Ortsfrequenzen ν_x und ν_y ist. Bei Propagation im freien Raum wirkt somit auf jede harmonische Funktion eine räumliche Phasenverschiebung, während die Amplitude gleich bleibt. Für größere Ortsfrequenzen, $\nu_x^2 + \nu_y^2 > \frac{1}{\lambda^2}$, wird der Term unter der Quadratwurzel negativ und somit der gesamte Exponent reell. Die Übertragungsfunktion wird demzufolge zu einem Dämpfungsfaktor für die Amplituden der harmonischen Funktionen.

Mithilfe dieser Übertragungsfunktion kann $g(x, y)$ auch für komplexere Eingangswellen berechnet werden. Da $\mathcal{L}$ ein lineares zeitinvariantes System ist, entspricht die Transformation $\mathcal{L}\{f(x, y)\}$ einer Faltung der Funktionen $h(x, y)$ und $f(x, y)$:

$$g(x,y) = \iint_{-\infty}^{\infty} f(x', y')h(x - x', y - y')dx'dy'. \tag{2.40}$$

Nach dem Faltungstheorem (Gleichung 1.9) gilt:

$$G(\nu_x, \nu_y) = \mathcal{H}(\nu_x, \nu_y)\mathcal{F}(\nu_x, \nu_y), \tag{2.41}$$

wobei $G(\nu_x, \nu_y)$, $\mathcal{H}(\nu_x, \nu_y)$ und $\mathcal{F}(\nu_x, \nu_y)$ die fouriertransformierten Funktionen $g(x, y)$, $h(x, y)$ und $f(x, y)$ sind. Zerlegt man $g(x, y)$ in harmonische Funktionen, erhält man:

$$g(x,y) = \iint_{-\infty}^{\infty} G(\nu_x, \nu_y) \exp(-i2\pi(\nu_x x + \nu_y y))d\nu_x d\nu_y \tag{2.42}$$

$$= \iint_{-\infty}^{\infty} \mathcal{H}(\nu_x, \nu_y)\mathcal{F}(\nu_x, \nu_y) \exp(-i2\pi(\nu_x x + \nu_y y))d\nu_x d\nu_y. \tag{2.43}$$

2.2.2.2 Fresnelsche Näherung

Betrachtet man nur kleine Ortsfrequenzen, die $\nu_x^2 + \nu_y^2 \ll \frac{1}{\lambda^2}$ erfüllen, kann die Übertragungsfunktion $\mathcal{H}(\nu_x, \nu_y)$ weiter vereinfacht werden. Setzt man $\theta^2 = \lambda^2(\nu_x^2 + \nu_y^2)$, kann der Exponent aus Gleichung 2.39 durch Binomialentwicklung wie folgt angenähert werden:

$$2\pi\left(\frac{1}{\lambda^2}-\nu_x^2-\nu_y^2\right)^{1/2} d = 2\pi\frac{d}{\lambda}(1-\theta^2)^{1/2} \tag{2.44}$$

$$= 2\pi\frac{d}{\lambda}\left(1-\frac{\theta^2}{2}+\frac{\theta^4}{8}-\ldots\right). \tag{2.45}$$

θ entspricht dem Winkel, den die ebene Welle mit der optischen Achse unter paraxialer Näherung bildet [ST91]. Vernachlässigt man alle außer die ersten zwei Therme, erhält man die **Fresnelsche Näherung** der Übertragungsfunktion:

$$\mathcal{H}(\nu_x,\nu_y) \approx \mathcal{H}_0 \exp(i\pi\lambda d(\nu_x^2+\nu_y^2)) \tag{2.46}$$

mit:

$$\mathcal{H}_0 = \exp\left(-i2\pi\frac{d}{\lambda}\right) = \exp(-ikd). \tag{2.47}$$

Durch einsetzen in Gleichung 2.43 ergibt sich:

$$g(x,y) = \mathcal{H}_0 \iint_{-\infty}^{\infty} \mathcal{F}(\nu_x,\nu_y)\exp(i\pi\lambda d(\nu_x^2+\nu_y^2))\exp(-i2\pi(\nu_x x+\nu_y y))d\nu_x d\nu_y. \tag{2.48}$$

Damit die Fresnelsche Näherung anwendbar ist, muss der dritte Term aus Gleichung 2.45 gegen Null gehen, beziehungsweise:

$$\frac{\theta^4 d}{4\lambda} \ll 1. \tag{2.49}$$

Diese Approximation wird somit benutzt um Propagation im **Nahfeld** zu berechnen.

Definiert man $\theta_m \approx a/d$ (maximaler Winkel) und $N_F = a^2/\lambda d$ (**Fresnel-Zahl**), wobei a dem maximalen radialen Abstand in der xy-Ebenen $z = d$ entspricht, kann Gleichung 2.49 wie folgt umgeschrieben werden:

$$\frac{N_F\theta_m^2}{4} \ll 1. \tag{2.50}$$

2.2.2.3 Frauenhofer Näherung

Setzt man die inverse Fourier-Transformierte der Fresnel-approximierten Übertragungsfunktion $\mathcal{H}(\nu_x,\nu_y)$

$$h(x,y) = h_0 \exp\left(-ik\frac{x^2+y^2}{2d}\right), \qquad h_0 = \frac{i}{\lambda d}\exp(-ikd), \tag{2.51}$$

in Gleichung 2.40, ein erhält man:

$$\begin{aligned} g(x,y) &= h_0 \iint_{-\infty}^{\infty} f(x',y') \exp\left(-i\pi\frac{(x-x')^2+(y-y')^2}{\lambda d}\right) dx'dy' && (2.52)\\ &= h_0 \iint_{-\infty}^{\infty} f(x',y') \exp\left(-i\pi\frac{(x^2+y^2)+(x'^2+y'^2)-2(xx'+yy')}{\lambda d}\right) dx'dy'. && (2.53) \end{aligned}$$

Ist $f(x,y)$ nur in einem Kreis um den Ursprung mit Radius b ungleich Null und die Propagationsstrecke d groß genug, so dass

$$N_F = \frac{b^2}{\lambda d} \ll 1, \tag{2.54}$$

kann der Term $\pi/(\lambda d)(x'^2+y'^2) \leq \pi b^2/(\lambda d)$ vernachlässigt werden. Gleichung 2.53 kann wie folgt umgeschrieben werden:

$$g(x,y) = h_0 \exp\left(-i\pi\frac{(x^2+y^2)}{\lambda d}\right) \iint_{-\infty}^{\infty} f(x',y') \exp\left(-i2\pi\frac{(xx'+yy')}{\lambda d}\right) dx'dy'. \tag{2.55}$$

Mit $\nu_x = x/\lambda d$ und $\nu_y = y/\lambda d$ entspricht das Integral der fouriertransformierten von $f(x,y)$. Man erhält den Ausdruck für die **Frauenhofer Näherung** welche die Propagation im **Fernfeld** beschreibt:

$$g(x,y) = h_0 \exp\left(-i\pi\frac{(x^2+y^2)}{\lambda d}\right) \mathcal{F}\left(\frac{x}{\lambda d}, \frac{y}{\lambda d}\right). \tag{2.56}$$

2.2.2.4 Optische Elemente

Optische Komponenten mit Ausdehnung $d \approx 0$ können durch eine einfache **Transmissionsfunktion** beschrieben werden [ST91], die in jedem Punkt dem Proportionalitätsfaktor zwischen der Eingangsfunktion $f(x,y)$ und der Ausgangsfunktion $g(x,y)$ entspricht:

$$t(x,y) = \frac{g(x,y)}{f(x,y)}. \tag{2.57}$$

Blende: Unter starker Näherung wird eine Welle direkt hinter einer Blende in jedem Punkt außerhalb der Öffnung ausgelöscht. Innerhalb der Öffnung bleibt die Welle unverändert. Diese Beziehung kann durch folgende Transmissionsfunktion beschrieben werden:

$$t(x,y) = \begin{cases} 1, & \text{innerhalb der Öffnung.} \\ 0, & \text{außerhalb der Öffnung.} \end{cases} \tag{2.58}$$

Dünne Linse: Die Fokussierung einer dünnen Linse entspricht einer punktabhängigen Phasenverschiebung. Die entsprechende Transmissionsfunktion lautet:

$$t(x,y) = \exp\left(ik\frac{x^2+y^2}{2f}\right) = \exp\left(i\pi\frac{x^2+y^2}{\lambda f}\right), \tag{2.59}$$

wobei f die Brennweite ist [ST91].

3 Konzept

Aus der Aufgabenstellung ergeben sich zwei Teilaufgaben: Die Erstellung des Baukastens und die physikalische Simulation der optischen Strahlführungen. In den folgenden beiden Abschnitten wird auf jede Teilaufgabe einzeln eingegangen.

3.1 Optischer Baukasten

Reale Laseraufbauten sind oft sehr komplex und bestehen aus einer Vielzahl von optischen Elementen und Propagationsstrecken welche in bestimmten Winkeln zueinander angeordnet sind (siehe Abbildung 3.1). Solche komplexen Strahlführungen können meist für theoretische Berechnungen auf lineare Aufbauten vereinfacht werden und sind somit deutlich übersichtlicher.

Der Baukasten soll dem Benutzer erlauben über eine 2D GUI solche lineare Laseraufbauten schnell zu modellieren. Dafür können optische Komponenten per *Drag & Drop* aus einer Liste auf einem Zeitstrahl bzw. Filmstreifen in sogenannten *Slots* aneinandergereiht werden. Der reale relative Abstand zwischen den Elementen wird hierbei nicht maßstabsgetreu dargestellt. Geometrische Eigenschaften wie Ausdehnung und Position relativ zur optischen Achse werden mit den restlichen Parametern (z.B. Brennweite einer Linse) als Eigenschaften der einzelnen Komponenten gespeichert.

Abbildung 3.1: Laserlabor, Helmholtz-Zentrum Dresden-Rossendorf.

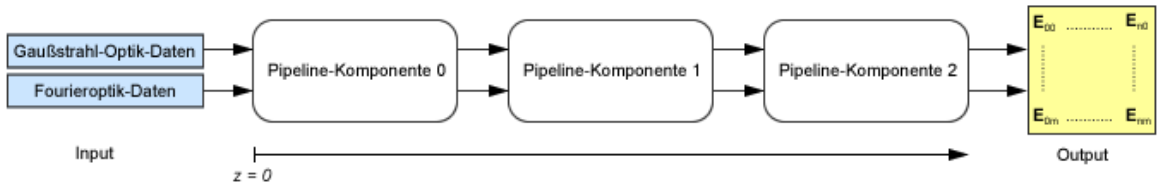

Abbildung 3.2: Transformation der modellabhängigen Datenstruktur nach dem Pipeline-Prinzip.

Weiterhin soll es dem Benutzer möglich sein, mehrere aufeinander folgende Elemente zu Gruppen zusammenzufassen, welche sitzungsunabhängig gespeichert und bei Programmstart automatisch geladen werden. Durch Gruppierung zweier Linsen, getrennt durch eine Propagationsstrecke, kann beispielsweise eine Teleskopkomponente erstellt und in verschiedenen Aufbauten wiederverwendet werden. Eine solche zusammengesetzte Komponente wird wie eine einfache Komponente (auch Basiskomponente genannt) behandelt und kann, wie oben beschrieben, in einen beliebigen *Slot* eingefügt werden. Außerdem ist es somit möglich, gruppierte Komponenten wiederum zu gruppieren.

Die Anwendung soll außerdem die Möglichkeit bieten, erstellte Laseraufbauten in einer Datei zu speichern bzw. aus einer solchen Datei Strahlführungen zu laden.

3.2 Physikalische Simulation optischer Strahlführungen

Alle im Kapitel 2 vorgestellten Modelle besitzen die Eigenschaft, dass die Transformationen der einzelnen optischen Komponenten auf den Laserstrahl unabhängig vom gewählten Laseraufbau sind. Die Transformationsvorschriften sind einzig und allein von den Parametern der Komponente abhängig. Somit können alle optischen Elemente modular implementiert und in beliebiger Reihenfolge aneinandergereiht werden. Allgemein können anhand der Ausdehnung zwei Arten von Komponenten unterschieden werden: Komponenten mit vernachlässigbarer Ausdehnung ($d \approx 0$) und Komponenten, deren Ausdehnung größer Null ist. Erstere wirken als punktuelle Transformationen auf den eingehenden Strahl. Im Fall der dünnen Linse entspricht dies einer Phasenverschiebung. Komponenten mit räumlicher Ausdehnung sind meist Propagationsstrecken bzw. Komponentengruppen mit Propagationsstrecken deren Transformation ortsabhängig ist.

Die Strahlführung kann nach dem **Pipeline-Prinzip** modelliert werden, nachdem eine modellabhängige Datenstruktur von den durchlaufenen Pipeline-Komponenten verändert wird (siehe Abbildung 3.2).

Somit ist es möglich, an einem beliebigen Punkt entlang der Strahlausbreitungsrichtung z die Eigenschaften des Lasers zu berechnen.

Implementiert werden beide aus der Wellenoptik vorgestellten Modelle: die Gaußstrahl-Optik und die Fourieroptik. Als Ergebnis beider Berechnungen erhält man eine komplexwertige 2×2-Matrix welche die diskretisierte komplexe Amplitudenverteilung des Laserstrahls auf einen Schnitt entlang der Ausbreitungsrichtung darstellt. Aus der komplexen Amplitude können Realteil, Imaginärteil, Betrag (bzw. Intensität des Lasers) und Argument (Phase des Lasers) berechnet werden (siehe Abschnitt 2.2).

3.2.1 Simulation mittels Gaußstrahl-Optik

Solange ein Gaußstrahl achsensymmetrische optische Komponenten durchläuft, bleibt dieser ein Gaußstrahl, es ändern sich jedoch der Strahlradius $w(z)$ und der Krümmungsradius $R(z)$. Nach dem Gaußstrahlmodell können beide Größen aus dem Strahlparameter q berechnet werden (siehe Gleichung 2.26). Betrachtet man Gleichung 2.29, erkennt man, dass die komplexe Amplitude eines beliebigen Punktes im Raum durch Angabe der initialen elektrischen Feldstärke E_0, des Radius im Fokus w_0, der Wellenlänge λ und des Strahlparameters q komplett definiert ist. Zuzüglich zweier Diskretisierungsparameter bilden diese Größen die Datenstruktur des Gaußstrahl-Modells:

```
1 struct GaussPipelineData {
2     double W0;                  // Radius im Fokus
3     double E0;                  // Initiale elektrische Feldstaerke
4     double Lambda;              // Wellenlaenge
5     complex<double> q;          // Strahlparameter
6     int Resolution;             // Aufloesung
7     double SamplingStep;        // Abtastfrequenz
8 }
```

wobei `Resolution` die Größe der Ergebnis-Matrix und `SamplingStep` die Abtastfrequenz darstellen.

Bei der im Abschnitt 2.2.1 beschriebenen Berechnungsvorschrift wird vorausgesetzt, dass der Fokus des Laserstrahls im Punkt $z = 0$ liegt. Somit kann der aus dem Laseraufbau berechnete z-Wert nicht verwendet werden. Vielmehr muss z relativ zum aktuellen Fokus berechnet werden (siehe Abbildung 3.3). Aus dem Imaginärteil des Strahlparameters kann der Strahlradius ermittelt werden, aus dem wiederum, durch umstellen von Gleichung 2.27, der Parameter z wie folgt berechnet werden kann:

$$z = z_0 \sqrt{\frac{w^2}{w_0^2} - 1}. \tag{3.1}$$

Der berechnete Wert ist stets positiv. Da jedoch der Strahl links und rechts des Fokus symmetrisch ist, spielt das Vorzeichen keine Rolle. Mit diesen Wert und denen aus der Datenstruktur kann die komplexe Amplitude nach Gleichung 2.29 berechnet werden. Die x- und y-Werte laufen dabei von

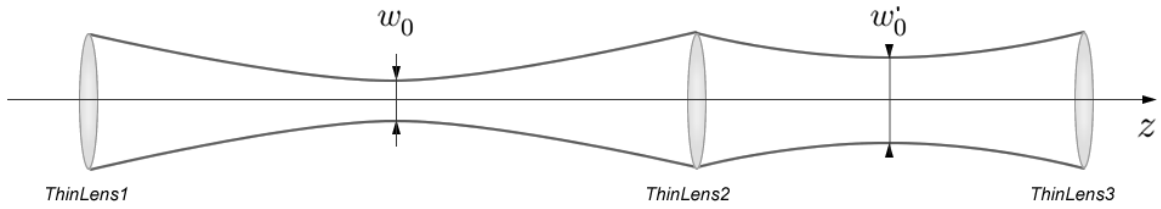

Abbildung 3.3: Gaußstrahl Fokussierung.

`-SamplingStep*Resolution/2` bis `SamplingStep*Resolution/2`.

In den folgenden Abschnitten werden die Transformationen der einzelnen optischen Komponenten näher beschrieben:

Quelle: Die Quelle initialisiert die Datenstruktur. Vom Benutzer einstellbare Parameter sind die initiale elektrische Feldstaerke E_0, die Wellenlänge λ, der Radius W und Krümmungsradius R des Strahls und die Gitterauflösung. Ist der Krümmungsradius $R \neq \infty$, so generiert die Quelle einen Strahl dessen Fokus nicht im Punkt $z = 0$ liegt. Der z-Offset kann wie folgt berechnet werden:

$$z = \frac{R}{1 + (\lambda R/\pi W^2)^2}. \tag{3.2}$$

Statt `GaussPipelineData.W0 = W` zu setzen muss auch der Radius im Fokus unter Berücksichtigung der Krümmung neu berechnet werden:

$$W_0 = \frac{W}{(1 + (\pi W^2/\lambda R)^2)^{1/2}}. \tag{3.3}$$

Der initiale Strahlparameter kann aus Gleichung 2.26 berechnet werden.

Die Transformation der Quelle lässt sich somit in Pseudocode wie folgt schreiben:

```
1 void SourceComponent::compute(GaussPipelineData data)
2 {
3     data.E0 = params.E0;
4     data.Lambda = params.Lambda;
5     data.Resolution = params.Resolution;
6     data.W0 = params.W / (1 + sqrt(PI * params.W^2 / (params.Lambda * params.R))^2);
7     data.q = 1 / params.R - i * params.Lambda / (PI * params.W^2)
8     data.Resolution = params.Resolution;
```

```
9     data.SamplingStep = params.W * sqrt(PI*params.Resolution) / params.Resolution;
10 }
```

Propagationsstrecke: Nach der Gaußstrahl-Optik ändert sich bei der Propagation im freien Raum der Strahlparameter q gemäß dem ABCD-Gesetz (Gleichung 2.4) mit $A = 1, B = L, C = 0$ und $D = 1$. Man erhält:

$$q_2 = q_1 + L, \tag{3.4}$$

wobei L die Länge der Propagationsstrecke ist, welche als Parameter der Komponente einstellbar ist.

Somit kann die Transformation dieser Komponente wie folgt beschrieben werden:

```
1 void PropagationComponent::compute(GaussPipelineData data)
2 {
3     data.q += params.Length; // ABCD-Gesetz
4 }
```

Dünne Linse: Eine dünne Linse (Ausdehnung ≈ 0) bewirkt eine ortsabhängige Phasenverschiebung des Laserstrahls, welche sich durch das ABCD-Gesetz wie folgt beschreiben lässt:

$$q_2 = \frac{q_1}{-q_1/f + 1}. \tag{3.5}$$

Außerdem ändert sich die Position und der Radius des Fokus (siehe Abbildung 3.3) welche nach Gleichung 3.2 und 3.3 neu berechnet werden müssen. Demzufolge lautet der Pseudocode für die Transformation:

```
1 void ThinLenseComponent::compute(GaussPipelineData data)
2 {
3     data.q = data.q / (-data.q / params.FocalLength + 1);       // ABCD-Gesetz
4     double W = sqrt(-data.Lambda / (PI * imag(1/data.q)));      // Berechnung des Radius
5     double R = 1 / real(1 / data.q);                    // Berechnung des Kruemmungsradius
6     data.WO = W / (1 + sqrt(PI * W^2 / (data.Lambda * R))^2);
7 }
```

3.2.2 Simulation mittels Fourieroptik

Möchte man komplexere Laseraufbauten simulieren, z.B. solche mit nicht achsensymmetrischen optischen Elementen, reicht das Gaußstrahlmodell nicht aus. Demzufolge wird ein auf der Fourieroptik basierendes Modell implementiert, durch das eine Vielzahl an weiteren Komponenten simuliert werden kann, welches jedoch rechnerisch deutlich aufwendiger ist. Im Gegensatz zum Gaußstrahlmodell wird hierbei jeder Gitterpunkt des diskretisierten Laserschnittes durch das Pipelinesystem geführt und transformiert. Die benötigte Datenstruktur und Berechnung sind somit im Gegensatz zum vorherigen Modell deutlich komplexer. Um den Strahl an einen Querschnitt entlang der z-Achse zu beschreiben, werden die Wellenlänge λ und eine Maxtrix mit den Werten der komplexen Amplitude benötigt. Zusätzlich werden, wie im vorherigen Modell, die Auflösung und Abtastfrequenz gespeichert:

```
1 struct FourierPipelineData {
2     Matrix E;                   // Komplexe Amplitude
3     double Lambda;              // Wellenlaenge
4     int Resolution;             // Aufloesung
5     double SamplingStep;        // Abtastfrequenz
6 }
```

In den folgenden Abschnitten werden die Transformationen der einzelnen Komponenten näher beschrieben:

Quelle: Die Quellkomponente setzt die Wellenlänge und initialisiert die Amplitudenmatrix:

```
1  void SourceComponent::compute(FourierPipelineData data)
2  {
3      data.Lambda = params.Lambda;
4      data.Resolution = params.Resolution;
5      data.SamplingStep = ... // Berechnung der Abtastfrequenz
6      data.E = new Matrix(params.Resolution, params.Resolution);
7
8      for(int i = 0; i < data.E.rows; i++)
9      {
10         for(int j = 0; j < data.E.cols; j++)
11         {
12             double x = (i - params.Resolution/2) * data.SamplingStep;
13             double y = (j - params.Resolution/2) * data.SamplingStep;
14             data.E[i,j] = ... // Berechnung von E an der stelle (x,y)
15         }
16     }
17 }
```

Die Werte der komplexen Amplitude können verschiedenster Herkunft sein. Diese können beispielsweise analytisch durch eine Gaußverteilung berechnet werden. Denkbar ist es auch Intensität und Phase aus einem Bild eines abgetasteten realen Laser einzulesen.

Propagationsstrecke: Gemäß Abschnitt 2.2.2.1 kann die Propagation im Nahfeld als Faltung der komplexen Amplitudenverteilung mit der Impulsantwortfunktion berechnet werden. Nach dem Faltungstheorem entspricht diese Operation einer punktweisen Multiplikation im Frequenzraum. Benötigt wird somit eine Matrix, deren Werte der diskretisierten Übertragungsfunktion (fouriertransformierte der Impulsantwortfunktion) entsprechen. Für die Propagation im freien Raum können diese Werte durch Gleichung 2.46 (Fresnelsche Näherung) berechnet werden. Durch inverse Fouriertransformation des Ergebnisses aus der Multiplikation mit der komplexen Amplitudenverteilung, erhält man die komplexe Amplitude des propagierten Strahls.

```
1  void PropagationComponent::compute(FourierPipelineData data)
2  {
3      // Nahfeldpropagation
4      Matrix H = new Matrix(data.Resolution, data.Resolution);
5      double freq = 1 / data.SamplingStep;
6      double freqSamplingStep = freq / data.Resolution;
7      // Initialisierung der Uebertragungsfunktion
8      for(int i = 0; i < H.rows; i++)
9      {
10         for(int j = 0; j < H.cols; j++)
11         {
12             double fx = (i - data.Resolution/2) * freqSamplingStep;
13             double fy = (j - data.Resolution/2) * freqSamplingStep;
14             H[i,j] = exp(-2 * i * PI * params.Length / data.Lambda) *
15                      exp(i * PI * data.Lambda * params.Length * (fx^2 + fy^2));
16         }
17     }
18     data.E = ifft(H .* fft(data.E)); // .* Komponentenweise Multiplikation
19 }
```

Im Fernfeld kann der propagierte Strahl durch die Frauenhofer Näherung berechnet werden. Bis auf den Vorfaktor entspricht das Ergebnis aus Gleichung 2.56 der fouriertransformierten Amplitudenverteilung nach Skalierung des Koordinatensystems. Die neue Abtastfrequenz kann wie folgt berechnet werden:

$$\Delta x = \Delta y = \frac{\lambda d}{L} \tag{3.6}$$

wobei L = `FourierPipelineData.Resolution * FourierPipelineData.SamplingStep` und d der Propagationsstrecke entspricht.

```
void PropagationComponent::compute(FourierPipelineData data)
{
    // Fernfeldpropagation
    // Skalierung des Koordinatensystem nach Gleichung 3.6
    double newSamplingStep = data.Lambda * params.Length /
                                (data.SamplingStepSize * data.Resolution);
    complex<double> h0 = i / (data.Lambda * params.Length) *
                         exp(-i * 2 * PI * params.Length / data.Lambda);
    Matrix prefactor = new Matrix(data.Resolution, data.Resolution);
    for(int i = 0; i < H.rows; i++)
    {
        for(int j = 0; j < H.cols; j++)
        {
            double x = (i - data.Resolution/2) * newSamplingStep;
            double y = (j - data.Resolution/2) * newSamplingStep;
            prefactor[i,j] = h0 * exp(-i * PI * (x^2 + y^2) / (data.Lambda * params.Length);
        }
    }
    data.E = prefactor .* fft(data.E) * data.SamplingStep^2; // .* Komponentenweise Mult.
    data.SamplingStep = newSamplingStep;
}
```

Dünne Linse: Wie im Abschnitt 2.2.2.4 erläutert, können alle Elemente mit vernachlässigbarer Ausdehnung durch Multiplikation der komplexen Amplitude mit einer komponentenspezifischen Transmissionsfunktion simuliert werden. Durch Diskretisierung der Transmissionsfunktion erhält man die Transmissionsmatrix. Im Fall der dünnen Linse kann diese gemäß Gleichung 2.59 berechnet werden.

```
void ThinLenseComponent::compute(FourierPipelineData data)
{
    Matrix trans = new Matrix(data.Resolution, data.Resolution);
    for(int i = 0; i < H.rows; i++)
    {
        for(int j = 0; j < H.cols; j++)
        {
            double x = (i - data.Resolution/2) * data.SamplingStep;
            double y = (j - data.Resolution/2) * data.SamplingStep;
            trans[i,j] = exp(i * PI * (x^2 + y^2) / (params.FocalLength * data.Lambda));
        }
    }
    data.E *= trans;
}
```

Blende: Um den Effekt einer Blende zu simulieren werden alle Elemente der Matrix `FourierPipelineData.E` auf 0 gesetzt, die außerhalb der Blendenöffnung liegen:

```
1  void ThinLenseComponent::compute(FourierPipelineData data)
2  {
3      Matrix trans = new Matrix(data.Resolution, data.Resolution);
4      for(int i = 0; i < H.rows; i++)
5      {
6          for(int j = 0; j < H.cols; j++)
7          {
8              double x = (i - data.Resolution/2) * data.SamplingStep;
9              double y = (j - data.Resolution/2) * data.SamplingStep;
10             if(/* Punkt (x,y) liegt ausserhalb der Oeffnung */)
11                 trans[i,j] = 0;
12         }
13     }
14     data.E *= trans;
15 }
```

Es können somit Blenden verschiedenster Formen erstellt werden (z.B. kreisförmig oder rechteckig).

Hindernis: Als Hindernis wird im Kontext dieser Arbeit das Inverse einer Blende bezeichnet. Alle Elemente der Matrix `FourierPipelineData.E`, welche innerhalb der geometrischen Ausdehnung des Hindernisses liegen, werden auf 0 gesetzt.

3.3 Parallelisierung und Caching

Aus dem vorherigen Abschnitt erkennt man dass, abgesehen von den Transmissionsmatrizen-Initialisierungen, die benötigten Operationen hauptsächlich aus komponentenweisen Matrizenmultiplikationen und zweidimensionalen Fouriertransformationen bestehen.

Da die Transmissionsmatrizen hauptsächlich von Komponentenparametern abhängig sind, können diese nach einmaliger Berechnung zwischengespeichert werden und, falls die Komponenten nicht geändert wurden, für den nächsten Datensatz wiederverwendet werden. Somit wird Rechenaufwand eingespart.

Bei der komponentenweisen Matrizenmultiplikation handelt es sich um eine Operation mit Komplexität $\mathcal{O}(n^2)$. Da diese n^2 einzelnen Multiplikationen entspricht kann der Rechenaufwand für dicht besetzte Matrizen nicht weiter Optimiert werden. Die Operation kann jedoch, dank der Unabhängigkeit der einzelnen Teilmultiplikationen, leicht auf Mehrkernprozessoren parallelisiert werden.

Die diskrete zweidimensionale Fouriertransformation (DFT) einer beliebigen Matrix der Form:

$$A = \begin{pmatrix} a_{0,0} & \dots & a_{0,M-1} \\ \vdots & \ddots & \vdots \\ a_{N-1,0} & \dots & a_{N-1,M-1} \end{pmatrix} \tag{3.7}$$

wird mathematisch definiert durch:

$$\hat{a}_{k,l} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} a_{m,n} \exp\left(-2\pi i \frac{mk}{M}\right) \exp\left(-2\pi i \frac{nl}{N}\right), \tag{3.8}$$

für $k = 0, \dots, M-1$ und $l = 0, \dots, n-1$. Die Rücktransformation (iDFT) lautet entsprechend:

$$a_{m,n} = \frac{1}{MN} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} \hat{a}_{k,l} \exp\left(2\pi i \frac{mk}{M}\right) \exp\left(2\pi i \frac{nl}{N}\right) \tag{3.9}$$

für $m = 0, \dots, M-1$ und $n = 0, \dots, n-1$. Die Vorzeichen der Exponenten und die Normalisierungsfaktoren (in diesen Fall 1 für die DFT und $1/MN$ für die iDFT) sind Konventionen und können sich von anderen Definitionen unterscheiden. Einzige Anforderung ist dass sich die Vorzeichen bei Hin- und Rücktransformation unterscheiden und dass das Produkt der Normalisierungsfaktoren $1/MN$ ergibt.

Da der transformierte Datensatz räumlich beschrenkt ist, die Fouriertransformation jedoch auf den gesamten Raum wirkt, kann die DFT als Fouriertransformation einer diskretisierten periodischen Funktion angesehen werden deren Periode dem abgetasteten Raum entspricht. Somit sind Ein- und Ausgangsmatrix verschiebungsinvariant.

Betrachtet man eine quadratische Matrix ($M = N$) ergibt sich die Komplexität der Berechnung eines einzelnen Wertes von $\hat{a}_{k,l}$ zu $\mathcal{O}(n^2)$. Wendet man Gleichung 3.8 auf alle Elemente der Matrix an erhält man eine Komplexität von $\mathcal{O}(n^4)$. Der Berechnungsaufwand wird jedoch durch die sogenannten FFT-Algorithmen (Fast Fourier Transform) deutlich verringert welche teilweise eine Komplexität von $\mathcal{O}(n^2 \log n)$ aufweisen [spa]. Bei einer 512×512-Matrix kann die FFT-Transformation 30,000 mal schneller als die DFT-Transformation sein. Die ohnehin schon schnellen FFT-Algorithmen können außerdem noch parallelisiert werden um die Rechenkapazität moderner System besser ausnutzen zu können.

Außer durch Optimierung der Komponenten kann die Berechnungszeit der Simulationspipeline durch ein Caching-System verringert werden (siehe Abbildung 3.4). Durch das Einführen eines Puffers zwischen den einzelnen Komponenten, welcher eine Kopie des zuletzt weitergeleiteten Datensatzes speichert, muss nach Änderung eines Parameters einer bestimmten Komponente k nicht mehr die gesamte Pipeline neu berechnet werden. Da der zwischengespeicherte Datensatz vor der geänderten Komponente nicht von der Parameteränderung betroffen ist, kann dieser als Ausgangspunkt für die Neuberechnung

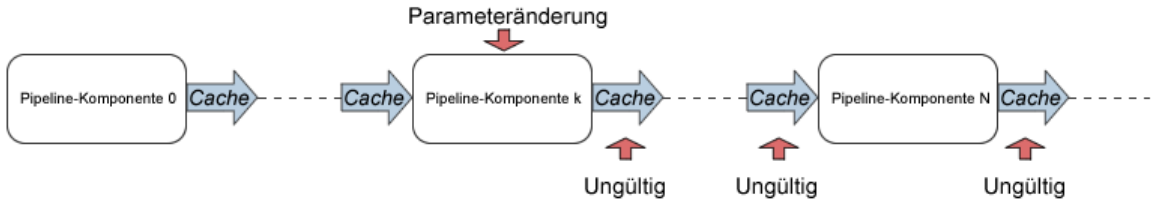

Abbildung 3.4: Caching-System: Ändert sich Komponente k müssen ausschließlich die nachfolgenden Komponenten aktualisiert werden.

verwendet werden. Die Puffer der Komponenten k bis N sind hingegen invalide. Somit müssen nur die Transformationen der Komponenten k bis N berechnet werden, während die ersten $k-1$-Komponenten übersprungen werden können.

Der Algorithmus startet bei der letzten Komponente der Simulationspipeline, welche aus ihren Eingangskanal Daten abruft. Ist der Puffer valide, wird dieser in der Transformation der Komponente verwendet. Andernfalls wird ein neuer Datensatz aus der vorgeschalteten Komponente abgerufen. Im Fall dass alle Puffer ungültig oder leer sind, wird somit die gesamte Pipeline durchlaufen. Da für große Auflösungen eine solche Caching-Strategie viel Platz einnehmen kann, ist es möglich diese abzuschalten um Speicherbedarf gegen Berechnungszeit einzutauschen.

4 Implementierung

Dieses Kapitel befasst sich mit der verwendeten Entwicklungsumgebung und der Architektur der Anwendung die erstellt wurde, um die in vorherigen Kapitel eingeführten Modelle und Konzepte zu implementieren.

4.1 Entwicklungsrichtlinien

Um es einem breiten Spektrum von Benutzern zu erlauben, die Software in gewohnter Umgebung zu benutzten, wurde die Anwendung so konzipiert dass sie möglichst plattform- und hardwareunabhängig ist. Bei Design und Programmierung wurde zudem auf Flexibilität und Erweiterbarkeit geachtet. Durch Modularisierung mehrerer Softwarekomponenten können solche relativ einfach ausgetauscht oder hinzugefügt werden. Dadurch ist es beispielsweise möglich mit geringen Zeitaufwand noch nicht implementierte optische Komponenten der Anwendung hinzuzufügen oder für parallele Berechnungen optimierte Rechenschnittstellen an die Simulation anzubinden.

4.2 Entwicklungsumgebung

Die Applikation wurde vollständig in C++ geschrieben. Diese plattformübergreifende Programmiersprache ermöglicht sowohl die Programmierung auf hohem Abstraktionsniveau, als auch die Implementierung effizienter und maschinennaher Berechnungen. Außerdem ist diese Programmiersprache weit verbreitet und bietet durch die Open-Source-Community eine Vielzahl von plattformunabhängigen Bibliotheken an.

Die Anwendung wurde in einem *Qt*-Kontext [qt] entwickelt. Außer einer leicht zu programmierenden und plattformübergreifenden grafischen Benutzeroberfläche stellt *Qt* unter anderen umfangreiche Bibliotheken zur Internationalisierung, XML-Unterstützung und Threading bereit. Alle plattform-spezifischen Schnittstellen, wie beispielsweise die OpenGL Erweiterung oder der Zugriff auf das Dateisystem, werden durch Qt-Klassen maskiert und dem Programmierer zur Verfügung gestellt. Außerdem bietet *Qt* eine Vielzahl an *Java*-ähnlichen, jedoch *STL* (c++ Standard Template Library) kompatiblen Containerklassen

und Hilfsfunktionen an, welche die Entwicklungszeit und Fehlerquote stark verringern. Alle Klassen und Funktionen sind durch eine umfangreiche Onlinedokumentation beschrieben.

Für die Visualisierung wurde die *VTK*-Bibliothek (Visualization Toolkit) [vtk] in die Anwendung eingebunden. Diese bietet eine Vielzahl an *VTK* Datenverarbeitungs- und Visualisierungs-Algorithmen für verschiedenste Zwecke an. Weiterhin wird dieses Framework aktiv weiterentwickelt und gehört zu den am weitesten verbreiteten Bibliotheken für wissenschaftliche Visualisierung.

4.3 Architektur und Design

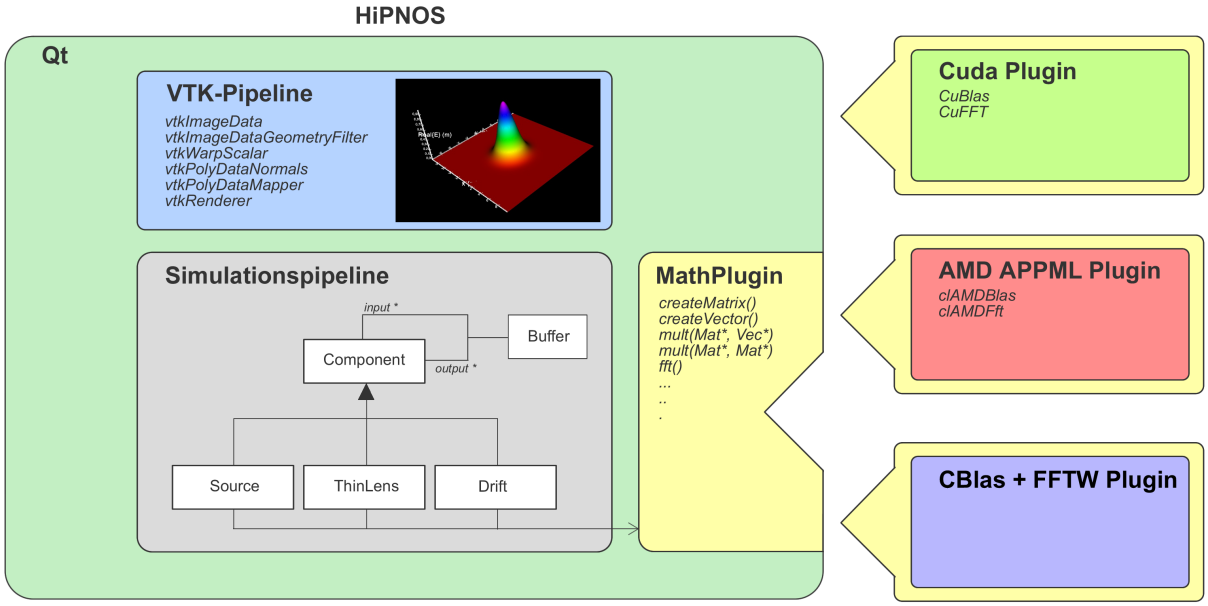

Abbildung 4.1: Hipnos, Architektur.

Die Architektur der Anwendung ist in Abbildung 4.1 vereinfacht dargestellt. Sie besteht hauptsächlich aus drei Modulen, eingebettet in die *Qt*-Umgebung.

Die optischen Komponenten werden durch die Simulationspipeline miteinander verbunden (siehe Abschnitt 3.2). Um für spätere Erweiterungen der Software kompatibel zu bleiben, besitzt jede dieser Komponenten mehrere Eingangs- und Ausgangskanäle. Somit können durch die Datenstruktur auch verzweigte Laseraufbauten als gerichtete Graphen repräsentiert werden. Die einzelnen optischen Komponenten sind durch Pufferklassen miteinander verbunden, welche die Berechnungszeit der Pipeline verringern (siehe Abschnitt 3.3).

Optische Komponenten müssen ein einfaches Interface implementieren und referenzieren nur wenige weitere Klassen. Somit bilden sie eigenständige Module, welche durch Schnittstellen mit dem System interagieren. Um die Komponenten einfach und übersichtlich zu halten, werden allgemeine mathematische Operationen von einem Mathematik-Modul übernommen, dessen Funktionen den Komponentenklassen zu Verfügung stehen. Implementierungen der mathematischen Operationen werden somit getrennt von den physikalischen Transformationsalgorithmen entwickelt (siehe Abschnitt 4.5).

Das Ergebnis aus der Simulationspipeline ist eine komplexwertige Matrix, welche der Amplitudenverteilung des Lasers auf einen Strahlquerschnitt entlang der Ausbreitungsrichtung entspricht (siehe Abschnitt 3.2). Diese Daten bildet die Eingabe der *VTK*-Pipeline deren wichtigste Algorithmen in Abbildung 4.1 aufgelistet sind. Die Schnittstelle der Anwendung zur *VTK*-Pipeline wird durch eine einzelne Klasse definiert. Somit kann diese durch andere Visualisierungsstrategien einfach ausgetauscht werden.

4.4 Benutzeroberfläche

Funktion	**Beschreibung**
$real\{E(\mathbf{r})\}$	Realteil der komplexen Amplitude
$imag\{E(\mathbf{r})\}$	Imaginärteil der komplexen Amplitude
$arg\{E(\mathbf{r})\}$	Argument der komplexen Amplitude bzw. Phase des Laserstrahls
$abs\{E(\mathbf{r})\}^2$	Quadrat des Betrages der komplexen Amplitude bzw. Intensität des Laserstrahls

Tabelle 4.1: Visualisierungsfunktionen.

Die Trennung von Laserbaukasten und Analyse findet sich auch in der Benutzerführung wieder. Durch eine vertikale Tab-Navigation am linke Fensterrand kann der Benutzer zwischen diesen beiden Ansichten wechseln (siehe Abbildung 4.2).

Die Baukastenansicht ist in drei Bereiche unterteilt. Am rechten Rand befinden sich Listen mit verfügbaren optischen Komponenten, welche per *Drag & Drop* auf einen freien *Slot* entlang des Strahlverlaufs gezogen werden können. Basiskomponenten und gruppierte Komponenten werden getrennt aufgelistet. Die Titelleiste des mittleren Fensterausschnitt beinhaltet mehrere *Buttons* um folgende Aktionen durchzuführen:

- In den Strahlverlauf hineinzoomen,

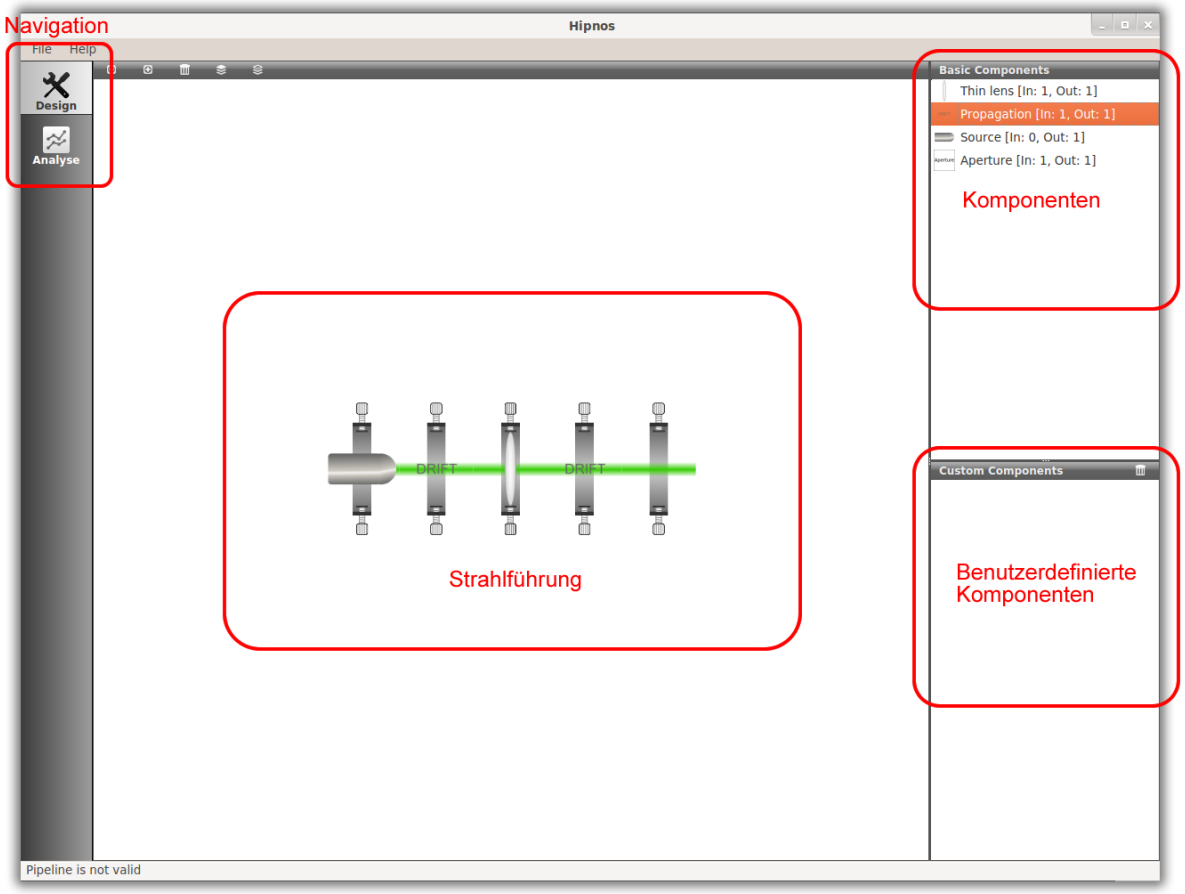

Abbildung 4.2: Benutzeroberfläche, Laserbaukasten.

- Aus dem Strahlverlauf rauszoomen,
- Ausgewählte Komponenten löschen,
- Ausgewählte Komponenten gruppieren,
- Ausgewählte Komponentengruppe auflösen.

Die Analyseansicht (siehe Abbildung 4.3) besteht aus vier Bereichen. Der zentrale und größte Fensterausschnitt dient der Visualisierung des Strahlquerschnitts. Im dreidimensionalen Raum werden Lasereigenschaften durch Färbung und z-Verschiebung dargestellt. Im rechts angrenzenden Fensterbereich können Visualisierungsparameter eingestellt werden. Der Benutzer kann hier auswählen, welche der in Tabelle 4.1 aufgelisteten Funktionen auf die Färbung und z-Verschiebung der 3D-Visualisierung übertragen werden. Somit ist es möglich, zwei physikalische Eigenschaften des Lasers über den Strahlverlauf parallel zu betrachten.

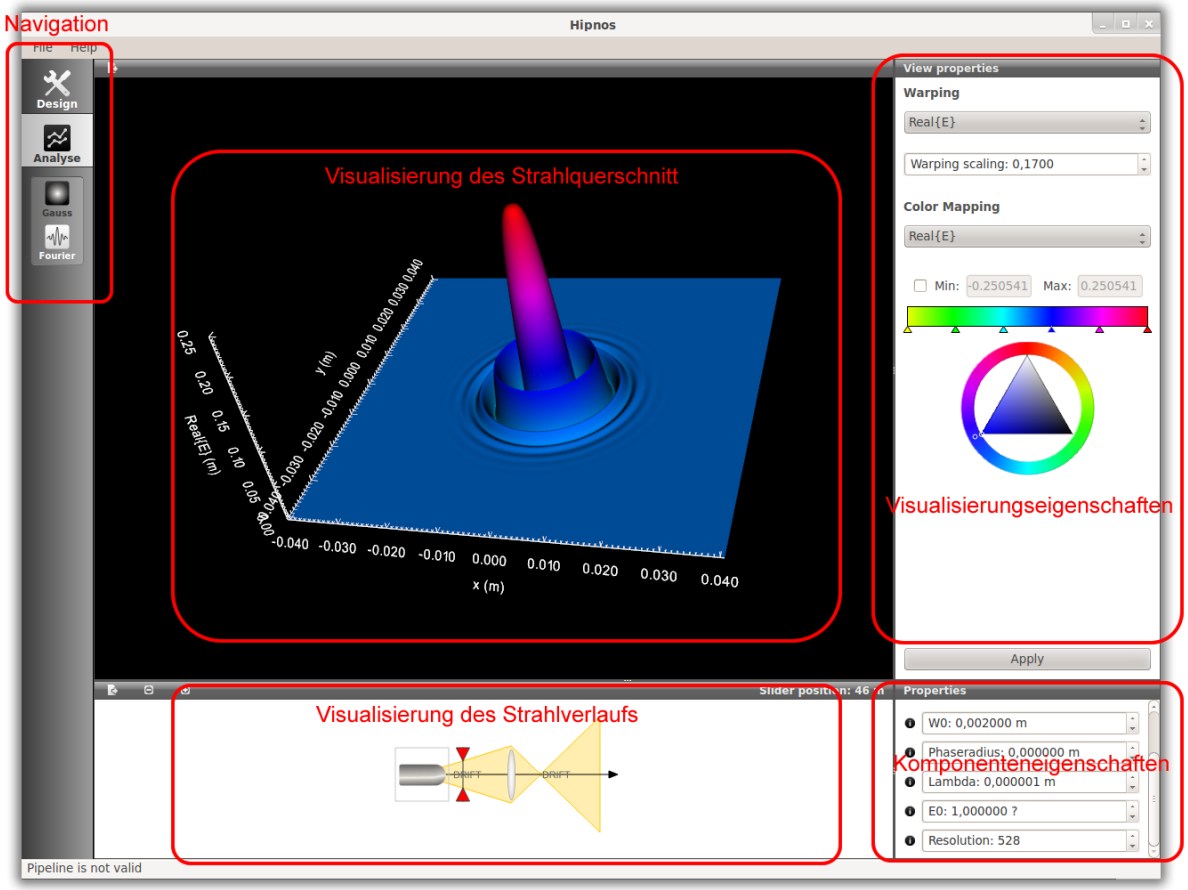

Abbildung 4.3: Benutzeroberfläche, Analyse.

Der untere Abschnitt des Anwendungsfensters zeigt den Strahlverlauf und dessen optische Komponenten. Durch Bewegen eines Schiebers entlang der optischen Achse kann der Benutzer die Position entlang des Strahls festlegen, die als Referenzpunkt für die Berechnung des Strahlquerschnitts dient. Im rechten unteren Fensterbereich können die Eigenschaften der auf dem Strahlverlauf selektierten Komponenten eingestellt werden. Jede Eigenschaft wird durch einen Popup-Dialog näher beschrieben. In dieser Ansicht verändert sich außerdem noch die Tab-Navigation. Hinzu kommen zwei *Buttons* um das hinter der Visualisierung stehende mathematische Berechnungsmodell zu selektieren. Sowohl der dreidimensionale Strahlquerschnitt als auch der Strahlverlauf können über die jeweiligen Titelleisten als Bild exportiert werden.

Außerdem können durch die Hauptmenüleiste weitere Aktionen durchgeführt werden, wie beispielsweise das Laden und Speichern von Strahlverläufen. Die Statusleiste am unteren Fensterrand zeigt Hinweise und Fehlermeldungen an.

4.5 Mathematische Berechnungen

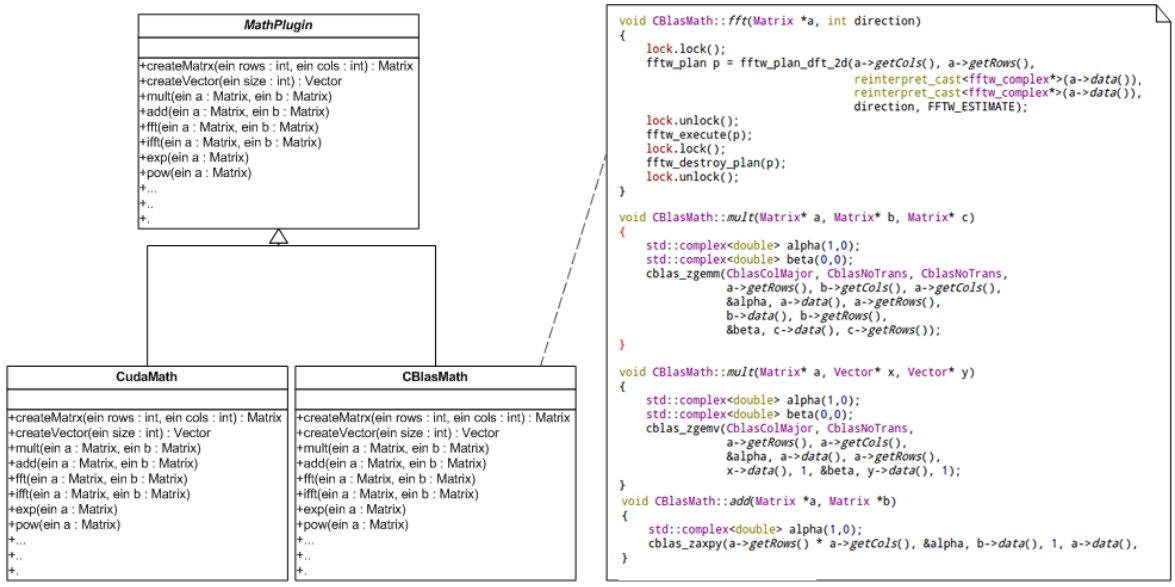

Abbildung 4.4: Plugin-Implementierung.

Das Mathematik-Modul liefert der Anwendung ein Schnittstelle, welche die für die physikalischen Berechnungen benötigten Operationen beschreibt. Die Funktionsimplementierung erfolgt jedoch in externen Plugins, welche durch das Modul beim Anwendungsstart dynamisch geladen werden (siehe Abbildung 4.1). Somit können verschiedenen Optimierungsstrategien für die einzelnen Operationen hinter der Modulschnittstelle eingesetzt werden.

Die entwickelten Plugins dienen überwiegend als *Proxy* zwischen der Plugin-Schnittstelle und einer Mathematik-Bibliothek. Durch diese Architektur bleibt die Anwendung ausschließlich von *Qt* und *VTK*, beides plattformübergreifende Bibliotheken, abhängig. Hardware- und softwareabhängige Implementierungen des Plugin-Interfaces können unabhängig entwickelt und je nach Systemkonfiguration mit der Hauptanwendung mitgeliefert werden.

Wie in Abschnitt 3.3 beschrieben bestehen die Transformationen der optischen Elemente nach der Fourieroptik zum größten Teil aus Matrizenoperationen und zweidimensionalen Fouriertransformationen. Jedes entwickelte Plugin referenziert somit eine oder mehrere Bibliotheken, welche diese Funktionalitäten implementieren. In Tabelle 4.2 sind die implementierten Plugins und deren verwendete Bibliotheken aufgelistet.

Name	Matrizenop.	FFT	Gerät	Genauigkeit
HipnosCBlasMathPlugin	CBlas	FFTW	CPU	`double`
HipnosACMLMathPlugin	ACML-BLAS	ACML-FFT	CPU	`double`
HipnosGSLMathPlugin	GSL-BLAS	GSL-FFT	CPU	`double`
HipnosCudaMathPlugin	cuBLAS	cuFFT	GPU	`double`
HipnosCudaMathSinglePrecisionPlugin	cuBLAS	cuFFT	GPU	`float`
HipnosAPPMLMathPlugin	clAmdBlas	clAmdFft	GPU	`double`

Tabelle 4.2: Übersicht der implementierten Plugins und deren Eigenschaften.

Matrizenoperationen: Matrizenoperationen werden durch das BLAS-Interface (Basic Linear Algebra Subprograms) [bla] beschrieben. Dieses bezeichnet einen Industriestandard [1] für Softwarebibliotheken, die elementare Operationen der linearen Algebra wie Vektor- und Matrixmultiplikationen implementieren. Man unterscheidet drei Funktionsgruppen:

- **Level 1** Funktionen berechnen Skalar-, Vektor- und Vektor-Vektor-Operationen
- **Level 2** Funktionen berechnen Matrix-Vektor-Operationen
- **Level 3** Funktionen berechnen Matrix-Matrix-Operationen

Da es sich nicht um einen echten Standard handelt, wird dieses Interface von vielen Herstellern nur als Richtlinie für eine eigene Funktionsschnittstellen verwendet. Oft unterscheiden sich die Methoden durch herstellerabhängige Präfixe, Parameter und Rückgabewerte, wodurch die Bibliotheken nicht binärkompatibel zur Referenzimplementierung sind. Obwohl eine BLAS-Bibliothek für die Entwicklung eines Plugins nicht zwingend notwendig ist, wird dadurch die Programmierung vereinfacht. Die zu implementierenden Funktionen sind so definiert, dass die Umsetzung mittels einer BLAS-Bibliothek trivial ist.

Fouriertransformationen: Für Fouriertransformationen hat sich bis heute noch kein Standard durchgesetzt. Jede Bibliothek bietet unterschiedliche Schnittstellen an, die oft sogar unterschiedliche Resultate liefern. Grund dafür sind unterschiedliche Normalisierungsfaktoren. Viele Hersteller implementieren die in Gleichung 3.7 und 3.8 beschriebenen Berechnungsvorschriften und skalieren die inverse Fouriertransformation mit dem Faktor $1/MN$ (siehe Abschnitt 3.3). Andere überlassen beispielsweise die Normali-

[1] Industriestandard, seltener auch Quasi-Standard, ist ein Standard, der sich im Laufe der Jahre als technisch nützlich und sinnvoll erwiesen hat und von verschiedenen Anwendern und Herstellern benutzt wird.

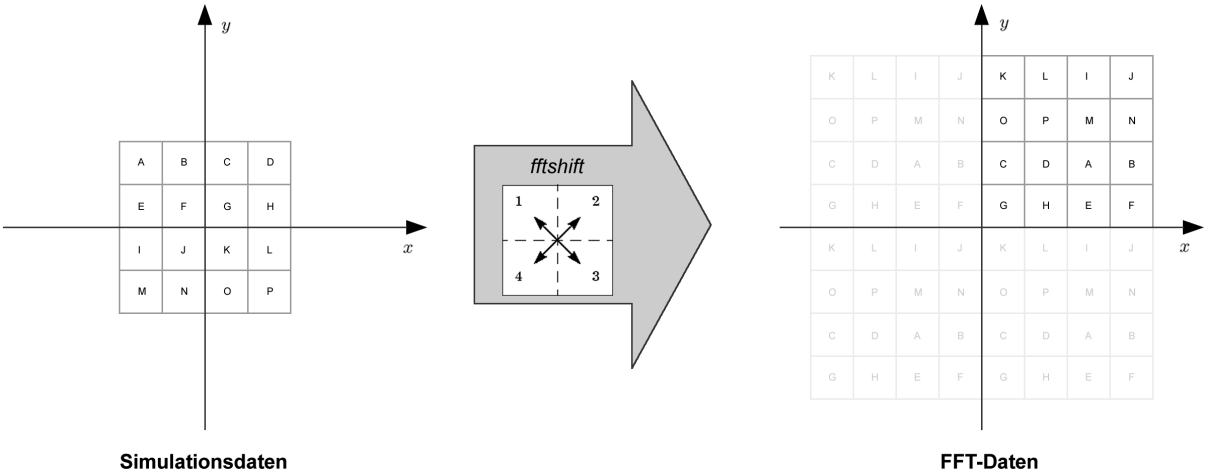

Abbildung 4.5: Veranschaulichung Nullpunktzentrierung mittels der *fftshift* Operation.

sierung dem Entwickler.

Zweidimensionale FFT-Algorithmen betrachten das Element $(0, 0)$ der Ein- und Ausgabematrix als Nullpunkt-Element. Um die Visualisierung benutzerfreundlicher zu gestalten, sind jedoch die von der Anwendung berechneten Datensätze auf den Nullpunkt zentriert (siehe Abbildung 4.5). Das Element $(0, 0)$ ist hierbei dem Punkt $(-N/2 \cdot SamplingStep, N/2 \cdot SamplingStep)$ zugeordnet. Da die DFT verschiebungsinvariant ist, kann die Eingangsmatrix um $N/2$ Elemente pro Achse zyklisch verschoben werden und ist somit mit den FFT-Algorithmen kompatibel. Nach der Transformation muss die Verschiebung rückgängig gemacht werden.

In den folgenden Abschnitten werden die implementierten Plugins und die verwendeten Bibliotheken näher erläutert.

4.5.1 CBlas und FFTW

Die in Fortran77 geschriebene BLAS-Referenzimplementierung von **netlib** [bla] bietet eine C-Schnittstelle (CBlas genannt) an, gegen welche dieses Plugin kompiliert wurde. Sämtliche Bibliotheken, welche mit dieser Schnittstelle binärkompatibel sind, können von diesem Plugin benutzt werden. Die bekanntesten und performantesten sind:

ATLAS: ATLAS (Automatically Tuned Linear Algebra Software) [atl] ist ein Forschungsprojekt, welches empirische Methoden anwendet um eine leistungsstarke und portable BLAS-Bibliothek zu erschaffen. ATLAS kann für Windows und die meisten Unix-Varianten kompiliert und auf beliebiger Hardware ausgeführt werden. Während des Kompilierungsprozesses kommen verschiedene Heuristiken zum Ein-

satz, wodurch die BLAS-Routinen für die installierte Hardware optimiert werden. Auf Mehrkernprozessoren erstellte Bibliotheken sind somit auch *multithreaded*.

OpenBlas: OpenBlas [opea] ist eine auf GotoBLAS2 [got] basierende optimierte BLAS-Bibliothek. GotoBLAS2 wurde als quelloffenes Projekt am *Texas Advanced Computing Center* entwickelt. Das Projekt verfolgte das Ziel neue Techniken und Algorithmen, gegeben durch moderne CPUs, für die BLAS-Routinen zu verwenden. OpenBlas ist die Weiterentwicklung des nicht mehr weitergeführten GotoBLAS2-Projektes. Zur Laufzeit wählt die Bibliothek einen für die CPU optimierten sogenannten *Kernel* aus, welcher die BLAS-Routinen ausführt. Wie auch ATLAS benutzt OpenBlas für Mehrkernprozessoren mehrere Threads.

Als FFT-Bibliothek verwendet dieses Plugin **FFTW** (**Fastest Fourier Transform in the West**) [fft]. Wie auch OpenBlas bietet FFTW portable Leistung durch automatische Optimierung zur Laufzeit. Die Bibliothek besteht aus sogenannten *solvers*, welche verschiedenen Algorithmen und Implementierungsstrategien für bestimmte Aufgaben bereitstellen. Während der Laufzeit können durch Heuristiken oder Zeitmessungen die besten *solvers* ausgewählt werden.

4.5.2 AMD Core Math Library

ACML (AMD Core Math Library) [acm] ist eine von AMD entwickelte und für AMD-Prozessoren optimierte frei erhältliche Mathematikbibliothek. ACML besteht aus vier Komponenten: einer BLAS-Implementierung, lineare Algebra Routinen, FFT-Algorithmen und einen Zufallszahlengenerator. Somit bietet sie alle für ein Plugin nötigen Funktionen an.

ACML ist Windows- und Linux-kompatibel und für 32-Bit und 64-Bit Architekturen verfügbar. Die Bibliothek ist als single-threaded oder als **OpenMP** [opec] parallelisierte multithreaded Version erhältlich.

4.5.3 GNU Scientific Library

GSL (GNU Scientific Library) [gsl] ist eine freie Mathematikbibliothek für C und C++. Sie ist unter der GNU General Public License erhältlich. Die Bibliothek bietet zahlreiche mathematischen Routinen an, unter anderem eine eigene BLAS-Implementierung und FFT-Algorithmen, welche von diesem Plugin verwendet werden. Allgemein bietet die Bibliothek keine hardwarespezifischen Optimierungen oder eine Multithreading-Unterstützung an.

Durch das **GnuWin** [gnu] Projekt ist eine auf Windows portierte Version dieser Bibliothek verfügbar.

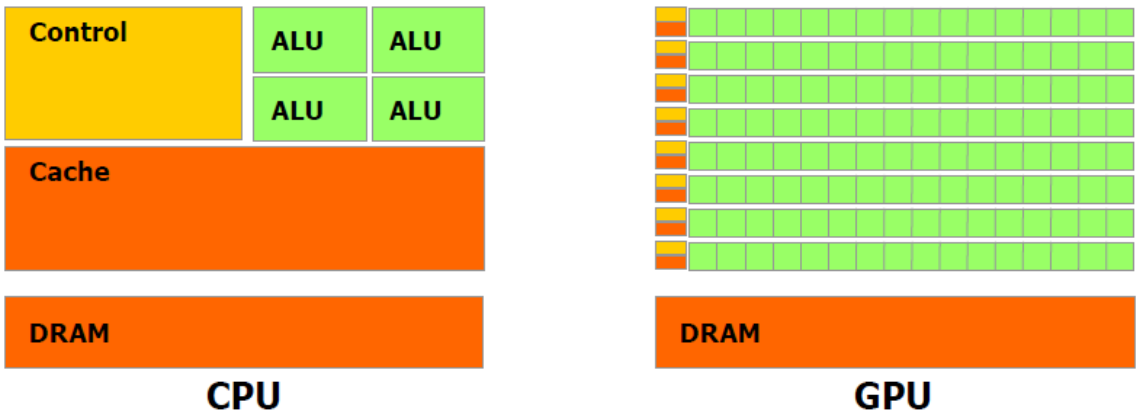

Abbildung 4.6: ALU- und Speicherverteilung in modernen GPUs im vergleich zu CPUs, Quelle: [NVI12c].

4.5.4 NVIDIA Compute Unified Device Architecture

Die bisher beschriebenen Plugins führen Berechnungen auf der CPU aus und speichern Daten im Hauptspeicher. Durch NVIDIAs CUDA (Compute Unified Device Architecture) [cud] ist es möglich die von der Grafikkarte bereitgestellte Rechenkapazität für eigene Algorithmen zu nutzen. Durch die immer weiter steigenden Anforderungen an die Graphik-Pipeline in modernen 3D-Anwendungen haben sich GPUs zu hochperformanten Multicore-Systemen (über 100 Kerne) entwickelt, welche auf datenparalleles Rechnen spezialisiert sind. Allgemein sind in GPUs deutlich mehr Transistoren den Berechnungen als dem Cachen von Daten zugewiesen (siehe Abbildung 4.6). Somit sind GPUs besonders effektiv wenn das Verhältnis von arithmetischen Operationen zu Speicheroperationen sehr hoch ist.

Da GPU-Code, sogenannte *Kernels*, nur auf den Grafikkartenspeicher zugreifen können, besteht eine GPU-beschleunigte Operation allgemein aus drei Schritten:

1. Die Daten werden von der Anwendung auf die Grafikkarten kopiert.
2. Die GPU-Funktionen werden aufgerufen.
3. Das Ergebnis wird in den Hauptspeicher zurück kopiert und der Anwendung zu Verfügung gestellt.

Um die Performance zu maximieren, müssen die zeitaufwendigen Kopieroperationen zwischen Hauptspeicher (*Host-Memory*) und Grafikkartenspeicher (*Device-Memory*) in der Anwendung minimiert werden. In diesem Plugin wird das Problem mithilfe zweier Flags (`hostDataChanged` und `deviceDataChanged`) in den Matrix- und Vektorklassen gelöst. Diese Klassen referenzieren im Host- und Device-Speicher gespiegelte Daten und markieren durch die Flags welcher der beiden valide ist. Somit ergeben sich vier Zustände und drei Transitionen welche den folgenden Methoden entsprechen:

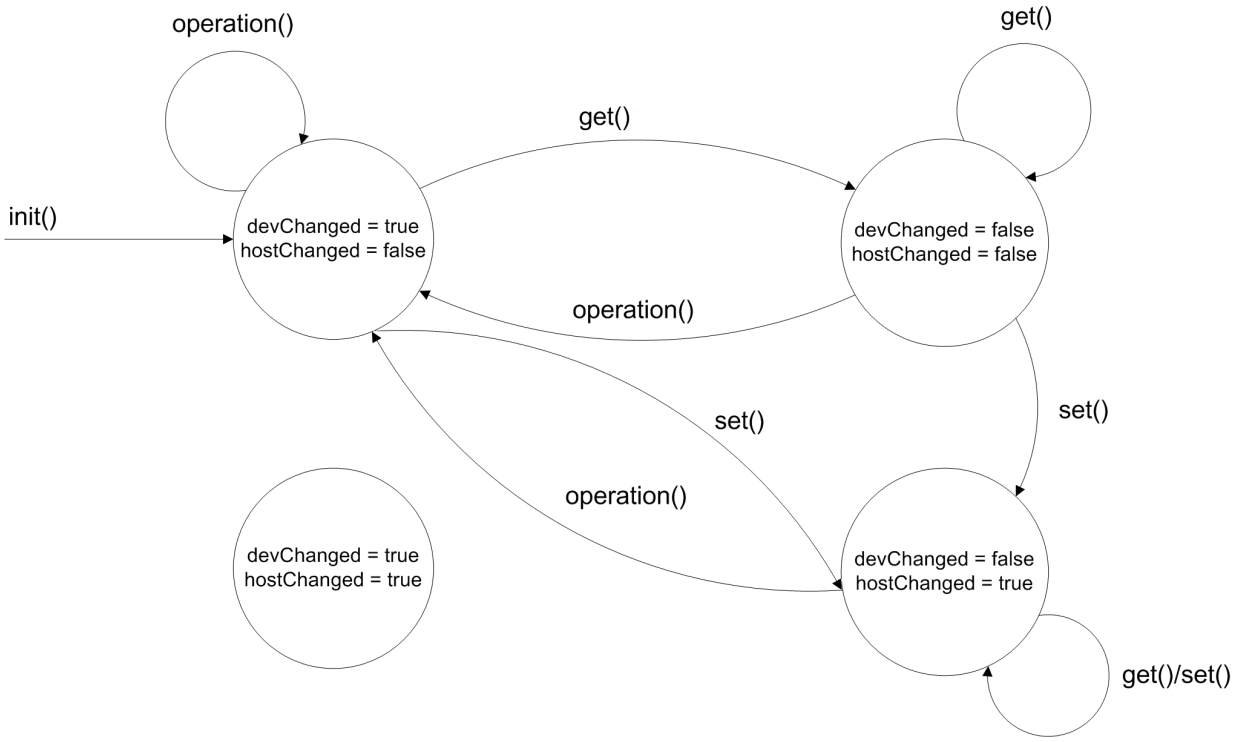

Abbildung 4.7: Zustandsdiagramm der Speicherverwaltung des CUDA-Plugins.

- `get()`: Die Anwendung fragt die Matrix/Vektordaten ab. Wenn die Daten im GPU-Speicher geändert wurden (`deviceDataChanged = true`), müssen diese erst in den Hauptspeicher kopiert werden.
- `set()`: Die Anwendung initialisiert oder ändert bestimmte Matrix/Vektorelemente. Um die Konsistenz zwischen den zwei Speicherreferenzen zu bewahren, müssen, falls der GPU-Speicher geändert wurden (`deviceDataChanged = true`), die Host-Daten zuvor aktualisiert werden.
- `operation()`: Die Anwendung ruft eine beliebige GPU-basierte Operation auf den Daten auf. Falls die Device-Daten nicht mehr aktuell sind (`hostDataChanged = true`), müssen diese aus dem Hauptspeicher in den Grafikkartenspeicher kopiert werden.

Die Zustandsübergänge können durch das in Abbildung 4.7 dargestellte Diagramm beschrieben werden.

Außer der Möglichkeit eigene GPU-beschleunigte Funktionen zu schreiben, bietet NVIDIAs CUDA SDK auch verschiedenen Bibliotheken an, die es dem Entwickler erlauben, ausschließlich Host-Programmcode zu schreiben um daraus die vordefinierten GPU-Routinen aufzurufen. Die von diesen Plugin verwendeten Bibliotheken sind:

cuBLAS: cuBLAS ist eine CUDA-beschleunigte Implementierung der BLAS-Routinen. Abhängig von der verwendeten BLAS-Funktion kann diese bis zu sechzehn Mal schneller als für Mehrkernprozes-

soren optimierte CPU-Implementierungen sein.

cuFFT: cuFFT ist eine CUDA-beschleunigte FFT-Bibliothek, modelliert nach der FFTW-Bibliothek [fft]. Für große Datenmengen ist cuFFT bis zu zehn Mal schneller als CPU-basierten Algorithmen.

Außer dem mit doppelter Genauigkeit arbeitenden *HipnosCudaMathPlugin* wurde für ältere Grafikkarten das *HipnosCudaMathSinglePrecisionPlugin* entwickelt, welches die CUDA-beschleunigten BLAS-Routinen mit einfacher Genauigkeit aufruft.

4.5.5 AMD Accelerated Parallel Processing Math Libraries

CUDA kann rechenintensiven Anwendungen erhebliche Berechnungszeit einsparen, ist jedoch auf Systeme mit NVIDIA-Grafikkarten beschränkt. Eine weitere Möglichkeit um Operationen auf Grafikkarten auszulagern ist **OpenCL** (**Open Computing Language**) [opeb]. Der OpenCL-Standard beschreibt eine Schnittstelle und C-ähnliche Programmiersprache für heterogene Parallelrechner. Im Gegensatz zu CUDA ist diese Technologie nicht auf spezielle Grafikkartenhersteller beschränkt. Das abstrakte Programmiermodell unterscheidet sich nur geringfügig zu dem von CUDA. Das System besteht aus einem Host und einem OpenCL-Gerät mit eigenem Speicher. Um Operationen auf dem Gerät durchzuführen, müssen die Quelldaten auf den Gerätespeicher kopiert werden. Die Host-Anwendung wird durch ein OpenCL-Event informiert, wann die Operation abgeschlossen ist und kann anschließend das Ergebnis aus dem Gerätespeicher auslesen. Auch hier gilt es die Datentransferoperationen zu minimieren um die Leistung zu steigern.

Die von AMD entwickelten APPML (AMD Accelerated Parallel Processing Math Libraries) [app] bestehen aus OpenCL-beschleunigten und für AMD GPUs optimierten BLAS- und FFT-Routinen. Die Bibliothek befindet sich noch im Beta-Stadium. Bisher wurden alle Level 2 und Level 3 BLAS-Funktionen implementiert. Die FFT-Schnittstelle ist auf Dimensionslängen beschränkt, deren Primfaktorzerlegung ausschließlich 2, 3 und 5 enthält.

5 Ergebnisse

Dieses Kapitel beschäftigt sich mit den Ergebnissen der Simulation, wobei besonderes Augenmerk auf deren Korrektheit und Berechnungszeit gelegt wird. Der erste Abschnitt geht auf die Verifizierungsmethoden ein, welche benutzt wurden, um die Ergebnisse der implementierten Modelle zu überprüfen. Abschnitt 5.2, 5.3 und 5.4 zeigen Performance-Messungen der verwendeten Bibliotheken und deren Einfluss auf die Berechnungszeit der Simulationspipeline. Alle in diesem Abschnitt vorgestellten Performance-Messungen wurden mit der in Tabelle 5.1 dargestellten Systemkonfiguration durchgeführt. Im Anhang A befinden sich Messergebnisse weiterer Systeme.

Prozessor	AMD Athlon 64 X2 6400+, 2x 3.20GHz
Mainboard	ASUS M2N4-SLI
Speicher	6GB DDR2-800
Grafikkarte	NVIDIA GeForce GTS 450
Betriebssystem	Linux Mint 13 Maya, 64-bit

Tabelle 5.1: Testsystem 1 Spezifikation.

5.1 Verifizierung

Für das fourieroptische Berechnungsmodell konnte die Korrektheit der Transformationen der einzelnen Komponenten durch numerischen Vergleich der Ausgangsdaten mit Testdaten verifiziert werden. Als Quelle dieser Testdaten diente eine vom *Helmholtz Zentrum Dresden-Rossendorf* [hzd] bereitgestellte MatLab-Script-Sammlung und die MatLab-Funktionen aus [Voe11] und [Sch10]. Unter Berücksichtigung der Rundungsfehler stimmen die Ergebnisse der Komponententransformationen mit den vorberechneten Testdaten überein.

Betrachtet man ausschließlich achsensymmetrische optische Elemente, entspricht die Fourioeroptik unter Verwendung der Fresnelschen Näherung der Gaußstrahl-Optik. Somit kann letztere ebenfalls verifiziert

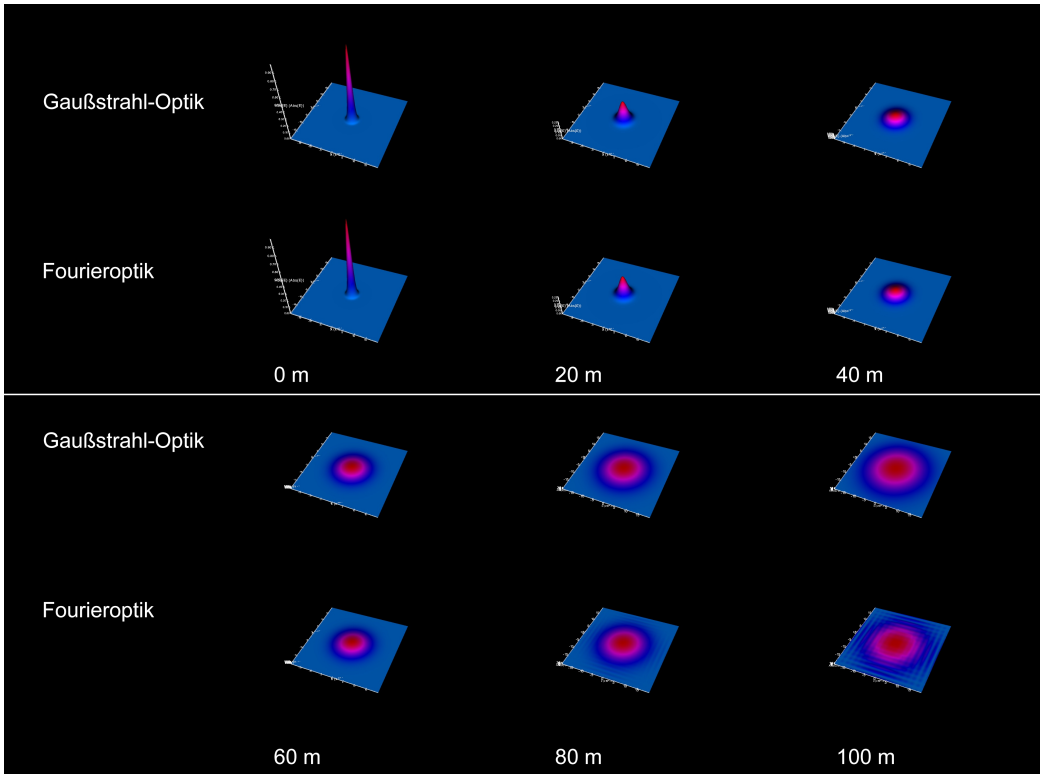

Abbildung 5.1: Gaußstrahl-Optik und Fourieroptik im Vergleich.

werden. Abbildung 5.1 zeigt die Intensitätsänderung eines Gaußstrahls entlang einer freien Propagationsstrecke, jeweils berechnet nach der Gaußstrahl- und Fourieroptik. Gewählt wurde ein Strahl mit folgender Anfangskonfiguration: $\lambda = 1030nm$, $w(0) = w_0 = 0.002m$, $R(0) = 0$, $E_0 = 1V/m$ und einer Gitterauflösung von 256 Punkten. Man erkennt, dass mit wachsender Propagationsstrecke der Unterschied zwischen den Berechnungsmodellen größer wird. Insbesondere bilden sich ab einem bestimmten z-Wert Artefakte durch die FFT-Transformationen, welche bei der Berechnung nach dem Gaußstrahl-Modell nicht auftreten. Abbildung 5.2 zeigt den maximalen Intensitätsunterschied $max\left(\frac{|\mathbf{I}_{gauss}(\mathbf{r})-\mathbf{I}_{fourier}(\mathbf{r})|}{\mathbf{I}_0}\right)$ zwischen den von beiden Modellen berechneten Werten entlang der Ausbreitungsrichtung z.

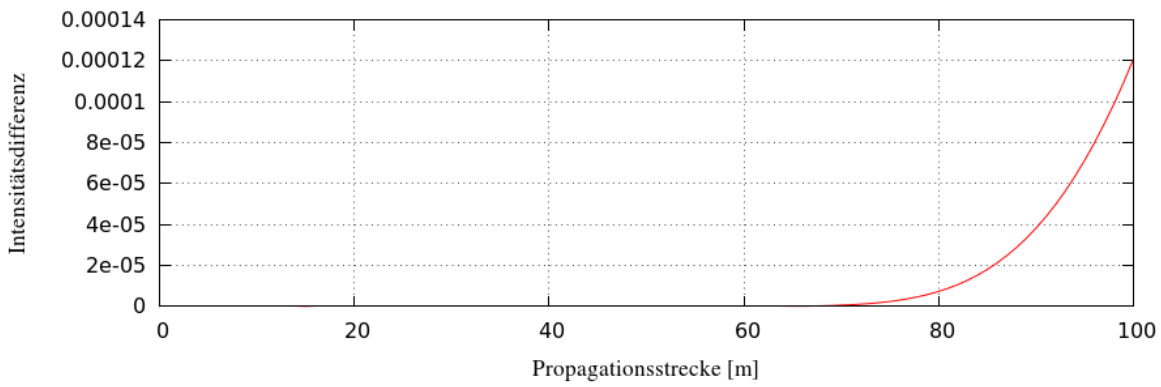

Abbildung 5.2: Maximale Intensitätsunterschied zwischen Gaußstrahl- und Fourierpropagation.

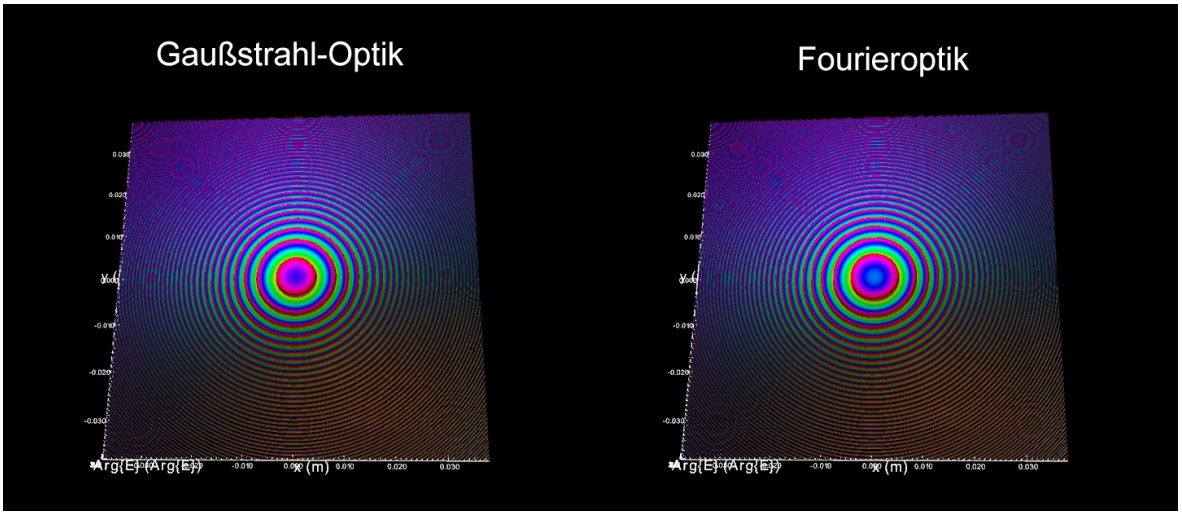

Abbildung 5.3: Konstante Phasenverschiebung zwischen Gaußstrahl- und Fourieroptik nach der Transformation durch eine dünnen Linse.

Bis zu einer Propagationsstrecke von $65m$ bleibt die Differenz der Beträge der komplexen Amplitude unterhalb 10^{-7}. Ab diesem Punkt werden die numerischen Fehler der FFT sichtbar und die Kurve steigt deutlich an.

Untersucht man die Wirkung einer dünnen Linse mit Brennweite $f = 25m$ auf demselben Strahl, erhält man eine Phasenverschiebung von $0.964945 \approx \pi/3$ zwischen den Berechnungsmodellen (siehe Abbildung 5.3). Da dieser Wert konstant über alle Matrixelemente ist, wird die Intensitätsänderung entlang der folgenden Propagationsstrecke nicht beeinflusst. Abbildung 5.4 zeigt die maximale Differenz zwischen der mittels dem Gaußstrahl-Modell und der Fourieroptik berechneten Intensität entlang der Ausbreitungsrichtung z.

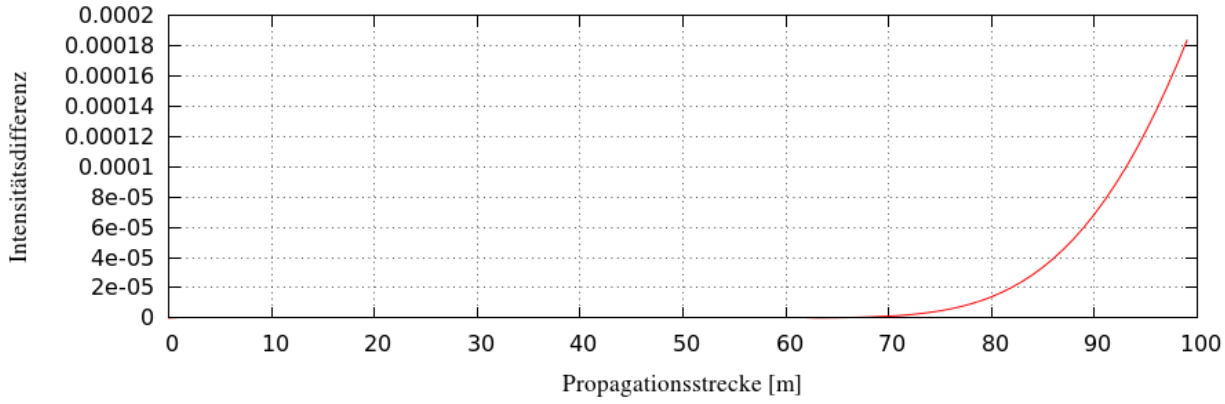

Abbildung 5.4: Maximale Intensitätsdifferenz zwischen Gaußstrahl- und Fourierpropagation nach der Transformation durch eine dünnen Linse.

5.2 BLAS-Benchmarks

Name	Plugin	Gerät	Beschreibung
netlib	*HipnosCBlasMathPlugin*	CPU	BLAS-Referenzimplementierung
ATLAS	*HipnosCBlasMathPlugin*	CPU	Dynamisch optimierte BLAS-Bibliothek
OpenBlas	*HipnosCBlasMathPlugin*	CPU	Auf moderne CPUs optimierte BLAS-Implementierung
ACML	*HipnosACMLMathPlugin*	CPU	Von AMD entwickelte BLAS-Implementierung
GSL	*HipnosGSLMathPlugin*	CPU	BLAS-Implementierung der GNU Scientific Library
cuBLAS	*HipnosCudaMathPlugin, HipnosCudaMathSinglePrecisionPlugin*	GPU	Von NVIDIA entwickelte CUDA-beschleunigte BLAS-Bibliothek
clAmdBlas	*HipnosAPPMLMathPlugin*	GPU	Bestandteil der OpenCL-beschleunigten APPML-Bibliothek

Tabelle 5.2: Übersicht der getesteten BLAS-Bibliotheken.

Durch die implementierten Plugins konnten fünf CPU- und zwei GPU-basierte BLAS-Implementierungen miteinander verglichen werden. Diese sind in Tabelle 5.2 aufgelistet. Anhand unterschiedlicher Matrizengrößen (Dimensionslänge einer $N \times N$-Matrix) wurde die Ausführungszeit (in Millisekunden) dreier Routinen mit unterschiedlicher Komplexität und unterschiedlichem Operations-Speicher Verhältnis gemessen. Die Routinen sind:

- Matrixaddition,
- Matrix-Vektor-Multiplikation,
- Matrixmultiplikation.

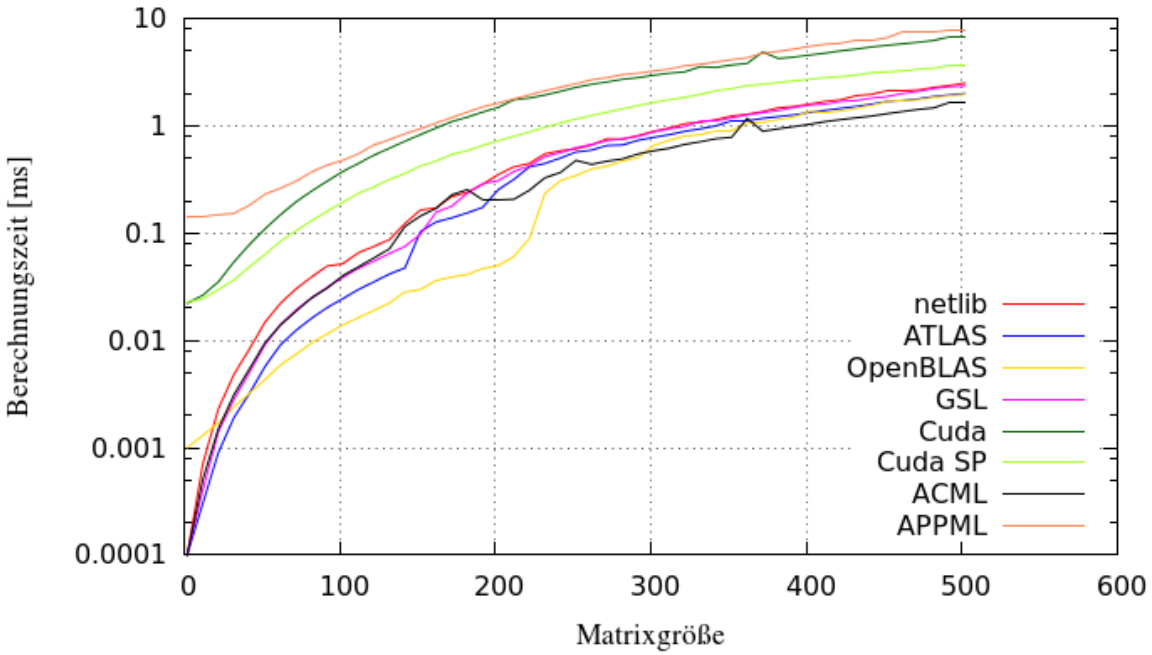

Abbildung 5.5: Benchmark der Matrixaddition.

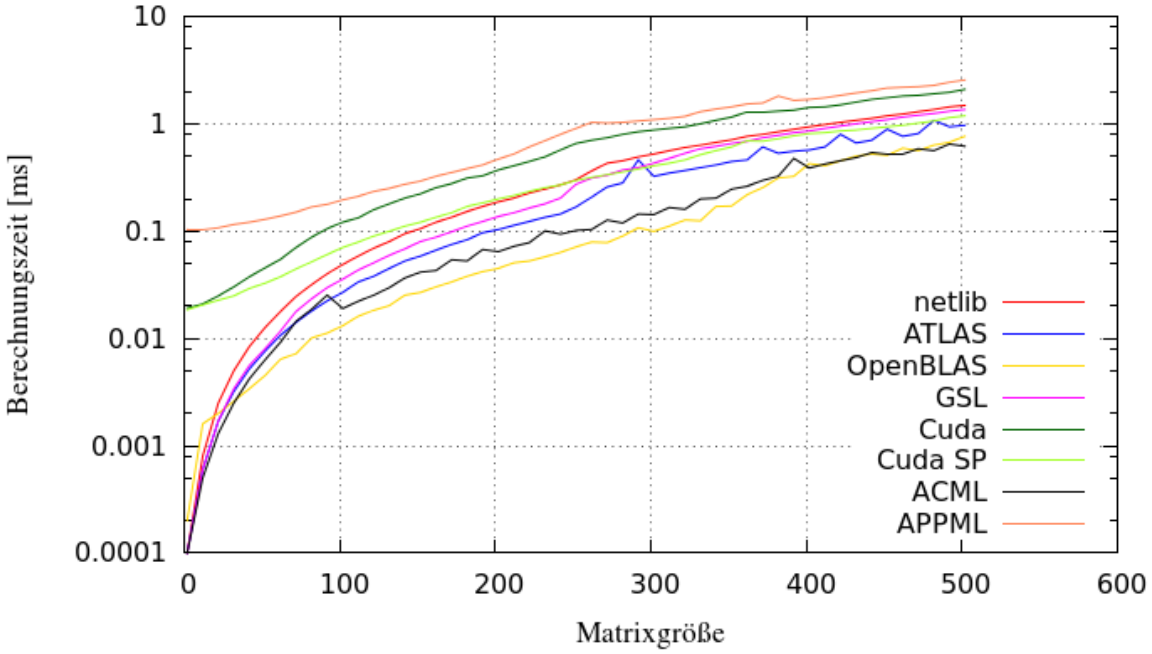

Abbildung 5.6: Benchmark der Matrix-Vektor Multiplikation.

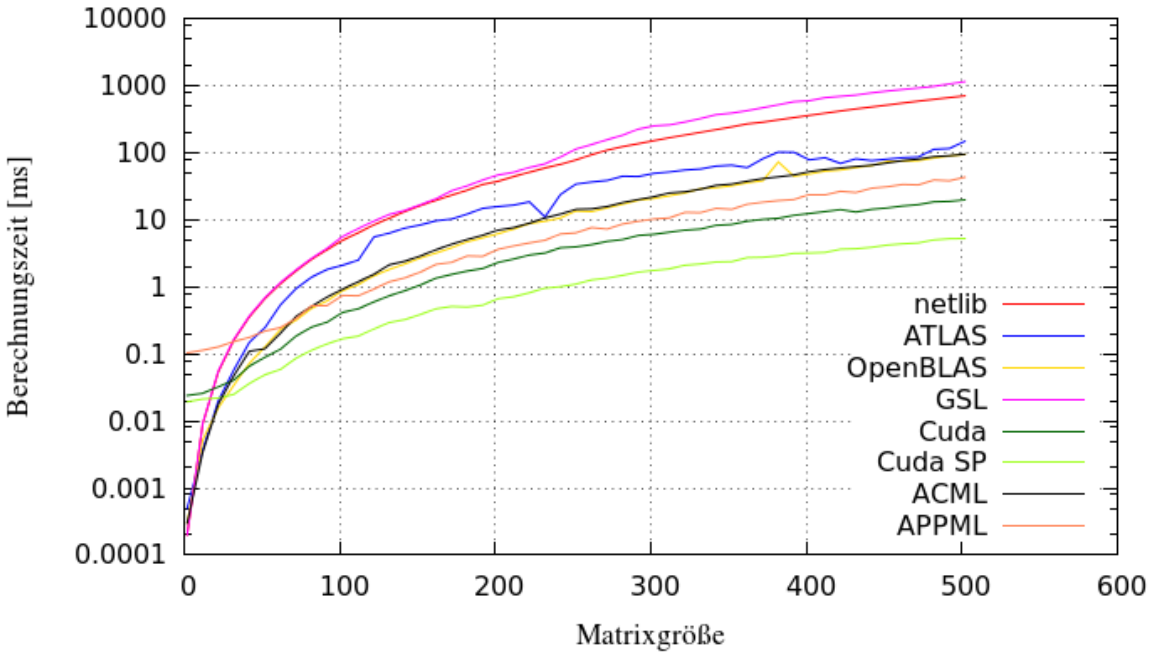

Abbildung 5.7: Benchmark der Matrixmultiplikation.

Man erkennt, dass die parallele Berechnungskraft, gegeben durch multicore-fähige oder GPU-beschleunigte Plugins, erst ab einer gewissen Komplexität der Operation bedeutend wird. Betrachtet man die Messergebnisse der Matrixaddition, bemerkt man dass die GPU-beschleunigten Plugins für diese Operation besonders ungeeignet sind. Grund dafür ist das geringe Operations-Speicher Verhältnis, durch welches viel Zeit bei der Übertragung der Daten auf die Grafikkarte verloren geht.

An den Messergebnissen der Matrix-Vektor-Multiplikation erkennt man, dass die Komplexität dieser Routine ausreicht, um die Rechenkapazität von Multicoresystemen durch Threading besser auszunutzen. Somit sind die ACML-, ATLAS- und OpenBlas-Implementierungen annähernd doppelt so schnell wie die nicht optimierten netlib- und GSL-Implementierungen. Für performante GPU-beschleunigte Berechnungen ist der Speicherbedarf im Verhältnis zur Komplexität der Operation immer noch zu gering. Sogar das *HipnosCudaMathSinglePrecisionPlugin*, welches nur halb soviele Daten auf die Grafikarten übertragen muss (siehe Abbildung 5.9), kann den Overhead durch die parallelisierte Berechnung nicht kompensieren.

Anders als bei den vorherigen Routinen ist die Komplexität der Matrixoperation so hoch, dass Parallelisierung einen wesentlichen Einfluss auf die Ausführungszeit hat. Somit sind GPU-beschleunigte Plugins wesentlich schneller als CPU-basierte. Wie zu erwarten, steigt für einzelprozeßgestützte Implementierungen die Berechnungszeit exponentiell mit der Matrixgröße an.

Abbildung 5.8 verdeutlicht den Einfluss des Operations-Speicher Verhältnisses auf die Ausführungszeit

der Matrix-Vektor-Multiplikation. Verglichen werden das GPU-beschleunigte *HipnosCudaMathPlugin* und die CPU-basierte OpenBlas-Implementierung. Je größer die Dimension und die Anzahl an Operationen auf eine Matrix, desto performanter ist die GPU- gegenüber der CPU-Berechnung.

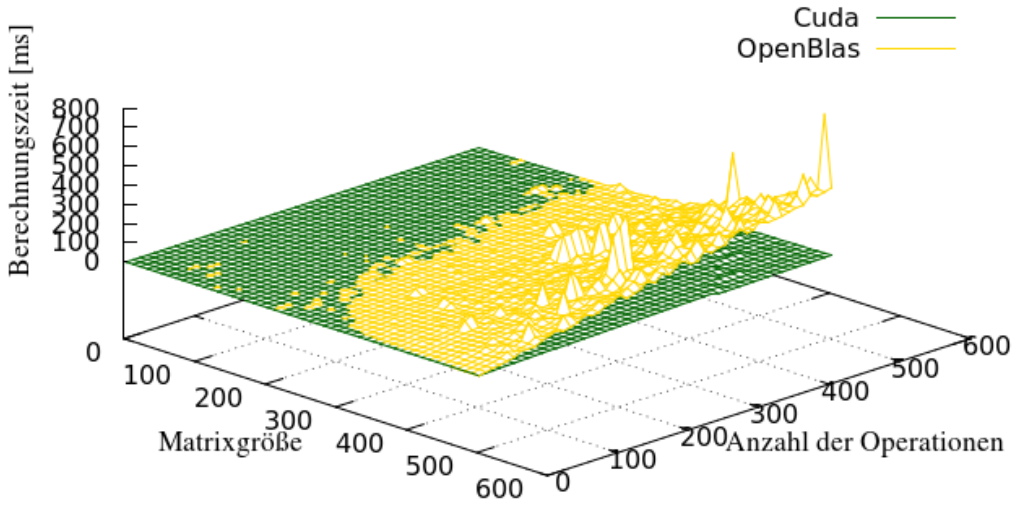

Abbildung 5.8: Veranschaulichung des Einflusses des Operations-Speicher Verhältnises anhand der Matrix-Vektor-Multiplikation.

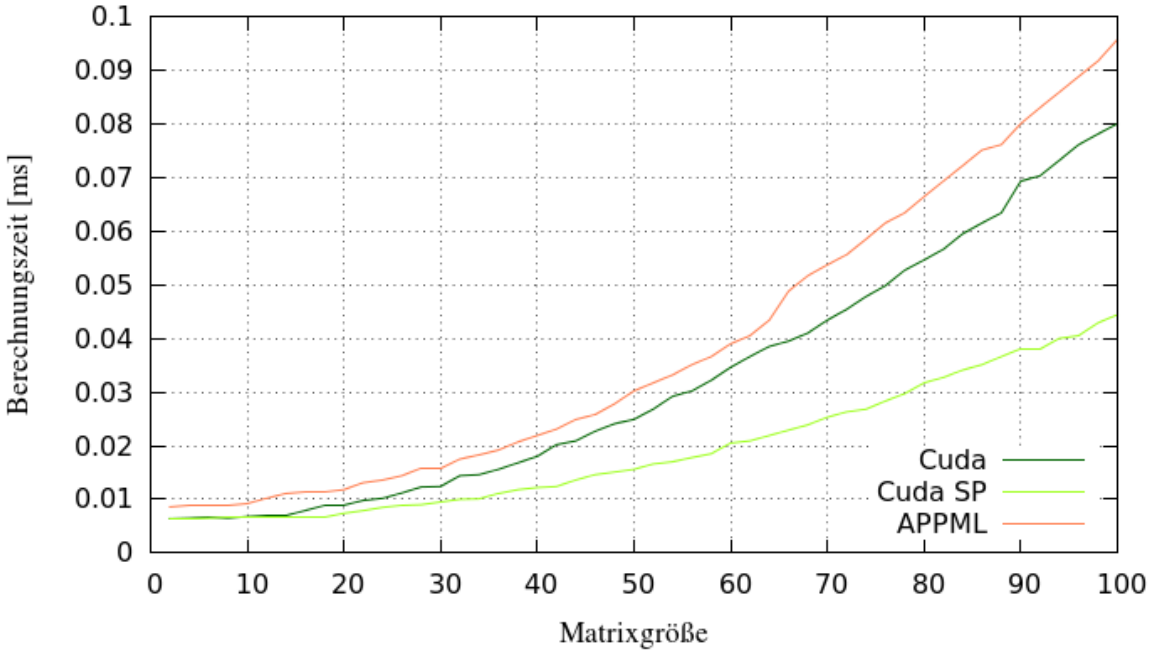

Abbildung 5.9: Vergleich der Ausführungszeit der Datentransferoperation für die GPU-basierenden Plugins.

5.3 FFT-Benchmark

Name	Plugin	Gerät	Beschreibung
FFTW	*HipnosCBlasMathPlugin*	CPU	Dynamisch optimierte FFT-Bibliothek
ACML	*HipnosACMLMathPlugin*	CPU	Von AMD entwickelte FFT-Bibliothek
GSL	*HipnosGSLMathPlugin*	CPU	FFT-Implementierung der GNU Scientific Library
cuFFT	*HipnosCudaMathPlugin, HipnosCudaMathSinglePrecisionPlugin*	GPU	Von NVIDIA entwickelte CUDA-beschleunigte FFT-Bibliothek
clAmdFft	*HipnosAPPMLMathPlugin*	GPU	Bestandteil der OpenCL-beschleunigten APPML-Bibliothek

Tabelle 5.3: Übersicht der getesteten FFT-Bibliotheken.

Tabelle 5.3 listet die von den implementierten Plugins verwenden FFT-Bibliotheken auf, welche in diesem Abschnitt verglichen werden. Abbildung 5.10 zeigt die gemessene Berechnungszeit einer zweidimensionalen Vorwärtstransformation für unterschiedliche Matrizengrößen. Ähnlich wie die Matrixmultiplikation kann diese Operation parallelisiert werden und profitiert somit von mehrkern- und GPU-basierten Implementierungen.

Weiterhin fällt auf, dass abhängig von den Matrixdimensionen unterschiedlich optimierte Algorithmen verwendet werden. Entsprechen die Höhe und Breite der Eingansmatrix einer Zweierpotenz, so benutzt beispielsweise die GSL-Bibliothek die performantere *radix-2* Routine [gsl].

Von den CPU-basierten Implementierungen ist die FFTW-Bibliothek am leistungsstärksten. Durch die Laufzeitoptimierung ist die Berechnungszeit dieser Bibliothek mit der von CPU-Herstellern entwickelten Implementierungen vergleichbar. Außerdem kann die Bibliothek auf einem beliebigen System mit C-Compiler erstellt werden und ist nicht für spezielle Hardware optimiert.

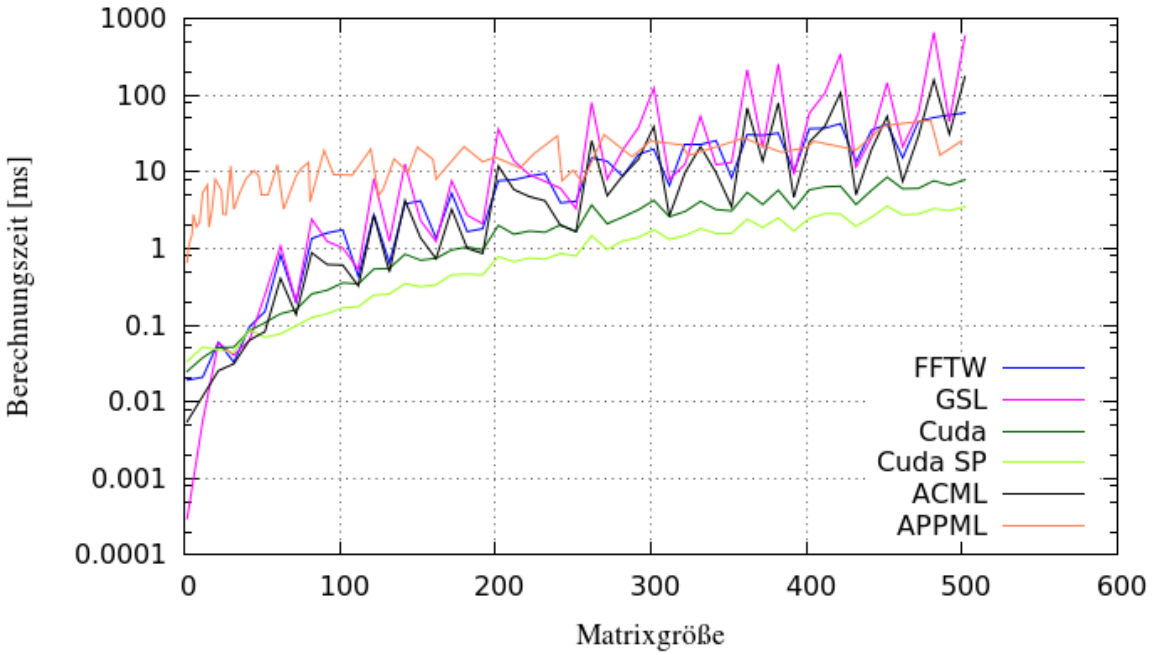

Abbildung 5.10: Benchmark der FFT-Bibliotheken.

5.4 Pipelinekomponenten-Benchmark

Die parallele Rechenkraft der CUDA-beschleunigten Plugins wird besonders gut ersichtlich wenn mehrere Operationen auf den gleichen Daten ausgeführt werden. Abbildung 5.11 zeigt die Ergebnisse aus der Messung der Ausführungszeit einer der rechenintensivsten Komponenten: die Propagation nach der Fresnelschen Näherung. Diese besteht aus sieben komponentenweisen Operationen (z.B. *pow(...)* und *exp(...)*) und zwei Fouriertransformationen.

Man erkennt den Einfluss der datenparallelen Algorithmen auf die allgemeine Berechnungszeit. Somit ist das einzelprozeßgestützte *HipnosGSLMathPlugin* die langsamste Implementierung und skaliert besonders schlecht mit der Auflösung der Simulationsdaten. Die CUDA-beschleunigten Plugins hingegen verarbeiten große Datensätze ohne besonderen Performanceeinbußungen.

Weiterhin fällt auf, dass die Kurve hauptsächlich von der Berechnungszeit der FFT-Algorithmen definiert wird. Dadurch ergibt sich eine starke Ähnlichkeit zwischen Abbildung 5.10 und 5.11.

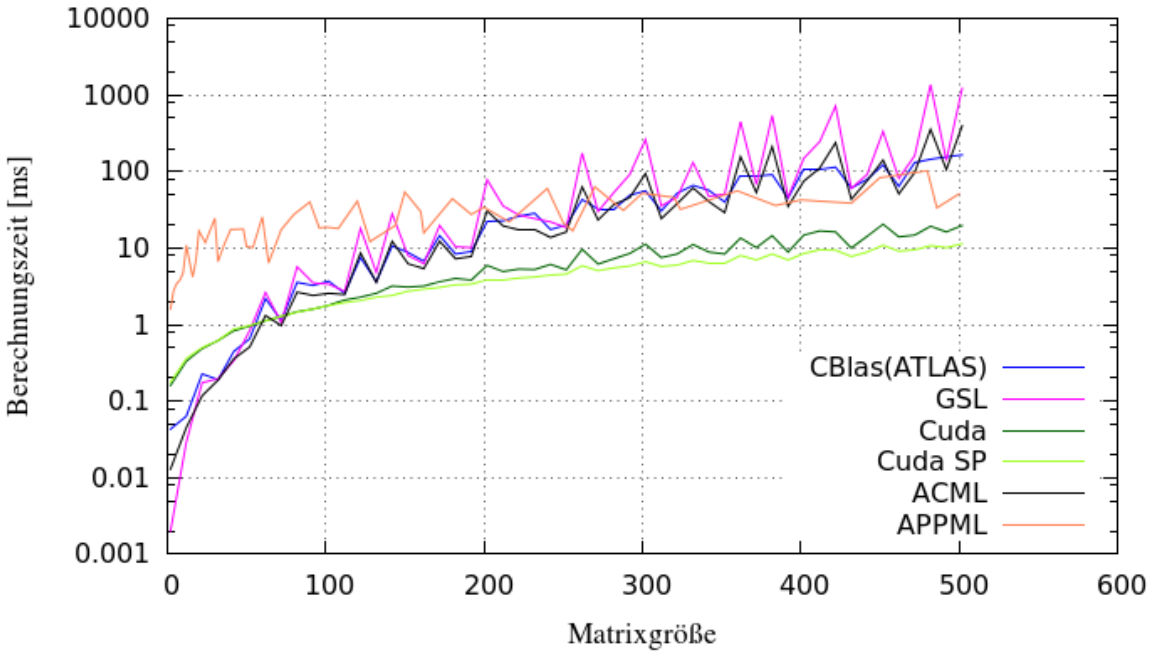

Abbildung 5.11: Messung der Berechnungszeit der Propagationskomponente (Fresnelsche Näherung).

6 Diskussion

Ausgehend von den Ergebnissen aus Kapitel 5 werden in den folgenden Abschnitten die gewonnenen Erkenntnisse diskutiert. Dabei werden zunächst die physikalischen Berechnungsmodelle miteinander verglichen und anschließend auf das implementierte Plugin-System eingegangen.

6.1 Physikalische Modelle

Wie in Kapitel 2 erläutert, können Ergebnisse aus einem bestimmten optischen Modell auch durch ein komplexeres Modell berechnet werden. Betrachtet man ausschließlich achsensymmetrische optische Komponenten und Propagation nach der Fresnelschen Näherung, können die Resultate aus der Fourieroptik mit denen aus der Gaußstrahl-Optik verglichen werden.

Wählt man Gitterauflösung N und Abtastrate Δl der Simulationsdaten, so dass der Radius des Gaußstrahl $w(z) \ll L$ ist (mit $L = \Delta l(N-1)$), so bleibt die Betragsdifferenz zwischen den mit beiden Modellen berechneten komplexen Amplituden unter $0,0001\%$ des Maximalwerts. Durch diesen geringen Fehler ist es möglich, die Rechenwege untereinander zu verifizieren.

Sind die genannten Bedingungen nicht erfüllt, ergeben sich durch die komplexen numerischen Berechnungen der Fourieroptik größere Abweichungen. Hauptursache dieses Fehlers sind Störfrequenzen, die sich bei der zweidimensionalen diskreten Fouriertransformation bilden, wenn sich die Eingangsdaten nahe der Matrixränder deutlich von Null unterschieden. Um physikalisch korrekte Resultate zu erhalten, müssen Auflösung und Abtastrate stets so gewählt werden, dass die Hauptmerkmale der Eingangsdaten in einem kleinen Bereich um den Mittelpunkt des zweidimensionalen Gitters konzentriert sind.

Die numerischen Fehler der komplexen Berechnungen nach der Fourieroptik sind ebenfalls bei Betrachtung der Wellenfronten eines Laserstrahls nach einer kurzen Propagationsstrecke sichtbar. Da die Phase dem Winkel zwischen Real- und Imaginärteil der komplexen Amplitude entspricht, ergeben sich für Werte nahe Null große Phasensprünge (siehe Abbildung 6.1). Dadurch ergibt sich in Bereichen, an denen die Genauigkeit der Fließkommazahlen nicht genügt, eine fehlerhafte Darstellung der Wellenfronten.

Für die Simulation einfacher Laseraufbauten liefert somit das analytische Gaußstrahl-Modell bessere Er-

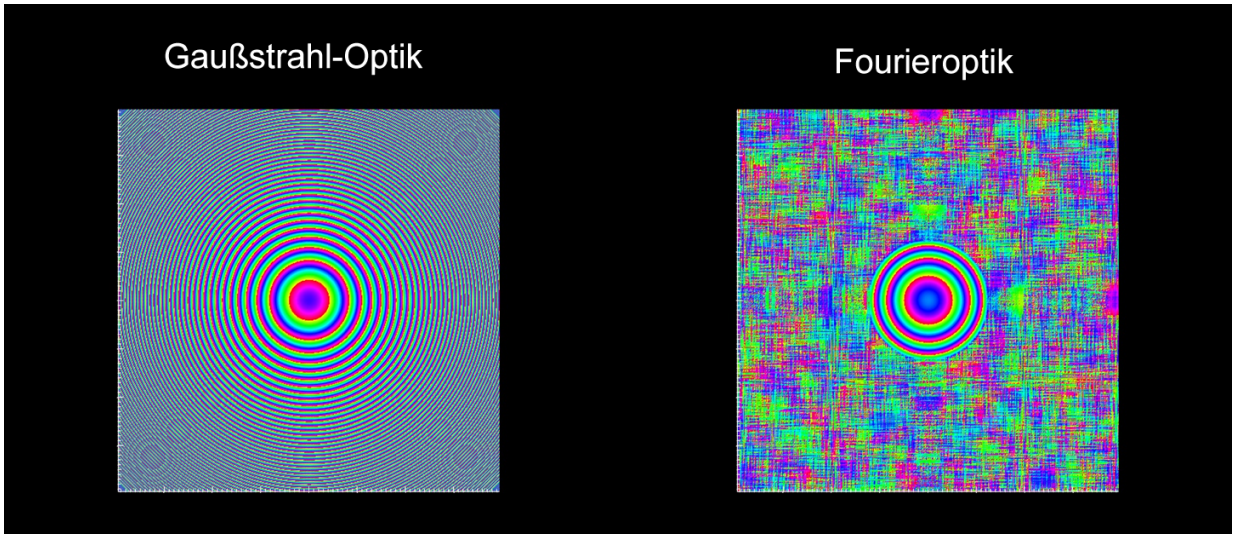

Abbildung 6.1: Darstellung der Auswirkung der numerischen Fehler auf die Berechnung der Phase nach der Fourieroptik.

gebnisse gegenüber der numerisch intensiveren Fourieroptik. Letztere kann jedoch wesentlich komplexere Strahlführungen simulieren, insbesondere auch solche mit nicht rotationssymmetrischen optischen Elementen (beispielsweise eine Spaltblende).

Ein weiterer Unterschied zwischen den implementierten Modellen ergibt sich bei der Berechnung der Phase. Bei der Herleitung der Transformationsvorschriften der optischen Elemente werden gemäß der Fourieroptik konstante Phasenverschiebungen oft vernachlässigt, um die Berechnungen zu vereinfachen. Diese Vereinfachung hat keinen Einfluss auf die weitere Berechnung der Intensitätsverteilung. Die vollständige Transferfunktion einer dünnen Linse lautet beispielsweise:

$$t(x,y) = \exp(-ikd)\exp\left(ik\frac{x^2+y^2}{2f}\right) = h_0\exp\left(ik\frac{x^2+y^2}{2f}\right), \tag{6.1}$$

wobei d die durch die Krümmung der Linsenoberflächen entstehende Breite des optischen Elements ist [ST91]. Der Phasensprung h_0 wird jedoch bei der numerischen Berechnung vernachlässigt. Es ergibt sich der in Abschnitt 5.1 diskutierte konstante Phasenunterschied zwischen den beiden Berechnungsmodellen.

6.2 Mathematische Berechnungen

Durch die implementierte Plugin-Schnittstelle wurde ein System entwickelt, welches zum einen die komplexe Umsetzung der parallelen Berechnungen von den Pipelinekomponenten trennt und zum anderen wesentlich zur Hard- und Software-Unabhängigkeit der Anwendung beiträgt. Außerdem bleibt letztere leicht erweiterbar. Neue Komponenten können mit wenig Aufwand implementiert werden und profitieren automatisch von den optimierten Berechnungen. Plattform-spezifische Plugins können unabhängig von der Anwendung entwickelt werden, um die Berechnungen der Pipeline für spezielle Systeme zu optimieren.

Durch die im Abschnitt 5.4 vorgestellten Messergebnisse wird deutlich, dass die intensiven Berechnungen der Lichtausbreitung nach der Fourieroptik effektiv parallelisiert werden können. Systeme mit CUDA- oder OpenCL-fähigen Geräten sollten somit die entsprechenden Plugins der Anwendung zur Verfügung stellen. Die hohe Leistungssteigerung gegenüber CPU-basierten Implementierungen wurde durch Auslagern der gesamten Berechnung auf die GPU erreicht. Die Ergebnisdaten aus der Simulation werden einmalig in den Hauptspeicher kopiert und der Anwendung für die Auswertung zur Verfügung gestellt. Die zeitintensive Übertragung der Daten zwischen Haupt- und Gerätespeicher wurde somit minimiert.

7 Ausblick

Die in dieser Ausarbeitung vorgestellte Anwendung bietet zahlreiche Weiterentwicklungsmöglichkeiten, die aus Zeitgründen nicht implementiert wurden. Diese werden in den folgenden Abschnitten näher beschrieben:

Komponenten: Es gibt, vor allem in der Fourieroptik, noch zahlreiche optische Elemente die ohne großen Aufwand durch die Anwendung simuliert werden können, jedoch noch nicht Implementiert sind. Beispiele dafür sind Prismen, Beugungsgitter oder verschiedene nicht-ideale Linsen.

Gauß-Moden: Der in Abschnitt 2.2.1 vorgestellte Gaußstrahl ist nur eine der vielen Lösungen der paraxialen Helmholtz-Gleichung. Gleichung 2.30 ist ein Spezialfall der sogenannten **Hermite-Polynome**, welche alle die paraxialen Helmholtz-Gleichung erfüllen (siehe Abbildung 7.1). Weitere sogenannte Gauß-Hermite Moden können problemlos in die Simulation eingebunden werden, um ein breiteres Spektrum an Lasern zu approximieren.

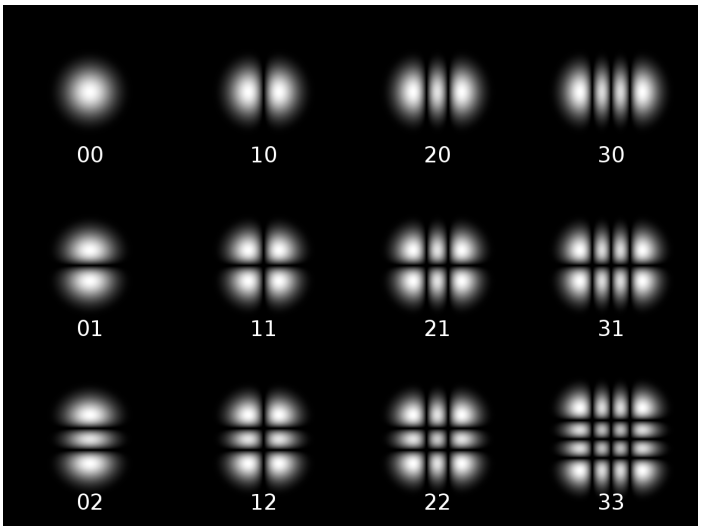

Abbildung 7.1: Gauß-Hermite Modem, Quelle [Wik12].

Berechnugsmodelle: Wie im Kapitel 2 bereits erwähnt, gibt es in der Optik weitaus mehr Berechnungsmodelle als die drei, die in dieser Arbeit vorgestellt wurden. Denkbar ist es weitere solche Modelle, wie beispielsweise das Beamlet-Modell, in die Anwendungsstruktur zu implementieren.

Komponenten in Javascript: Um die Erstellung neuer Komponenten zu erleichtern, wäre es denkbar eine Javascript-Schnittstelle an die Anwendung anzubinden. Optische Transformationen könnten somit leicht implementiert und von der Anwendung bei Programmstart dynamisch geladen werden. Qt bietet dafür ein eigenes Modul (*QtScript Module*) mit vielen Hilfsfunktionen an. Ein- und Ausgangskanäle, sowie das Mathematikmodul, könnten über Javascript native Objekte zugänglich gemacht werden.

Octave Integration: GNU Octave [oct] ist eine freie Software zur numerischen Lösung mathematischer Probleme, dessen Syntax der von Matlab [Mat] ähnelt. Octave wird durch die eigene CLI (Command-Line Interface) gesteuert, welche in der vorgestellten Anwendung eingebunden werden könnte. Denkbar wäre es, mittels der von Octave bereitgestellten Befehlen, dem Benutzer zu erlauben die aus der Simulation stammenden Daten weiter zu bearbeiten und analysieren.

Beamsplitter: Mit dem jetzigen Stand der Anwendung ist die physikalische Lasersimulation auf lineare Aufbauten beschränkt. Eine mögliche Weiterentwicklung wäre die Einbindung von Komponenten die Verzweigungen im Strahlverlaufs erzeugen (z.B. Beamsplitter). Die Daten- und Pipeliene-Struktur wurden nach dieser Anforderung bereits modelliert und können baumartige Strahlführungen abbilden und berechnen. Die Schwierigkeit besteht somit in der Konzeptionierung und Umsetzung der Benutzeroberfläche, und Benutzerführung um stark verzweigte Strahlführungen übersichtlich zu präsentieren.

3D-Visualisierung des gesamten Laseraufbaus: Der jetzige Stand der Anwendung zeigt dem Benutzer, als Ergebnis der Simulation, die Eigenschaften des Lasers auf einem Querschnitt entlang des Strahlverlaufs. Denkbar ist es die Eigenschaften und Form des Strahls, des kompletten Laseraufbaus, in einer dreidimensionalen Szene zu visualisieren.

CUDA - Zero-Copy Memory: In manchen modernen Systemen befinden sich CPU und GPU auf dem selben Chip und benutzten den gleichen Speicher. CUDA-beschleunigte Funktionen können somit direkt auf Daten im Hauptspeicher zugreifen und die zeitaufwendigen Kopieroperationen einsparen. Durch die CUDA-API kann Speicher als *Mapped Pinned Memory* angelegt werden. Solche Speicherregionen werden vom Betriebssystem nie ausgelagert (*Page-Locked*). Außerdem werden diese durch die CUDA-API in den Grafikspeicher gespiegelt. Greifen Host und Device auf den selben physischen

Speicher zu, wird die Spiegelung überflüssig. Solch eine Speicherregion wird als *Zero-Copy Memory* bezeichnet. Die Ausnutzung dieser Technik könnte die Berechnungen durch das CUDA-Plugin deutlich beschleunigen.

Parallele Visualisierung verschiedener Wellenlängen: Für den Benutzer interessant ist es Strahlen verschiedener Wellenlängen parallel zu visualisieren und zu vergleichen. Da die Wellenlänge ein Parameter der Quellkomponenten ist, kann eine Visualisierungsstrategie entwickelt werden, welche den Strahlquerschnitt für unterschiedliche Parameterwerte einer beliebigen Komponente gleichzeitig anzeigt.

Optimierung der 3D-Visualisierung: Durch den Berechnungsaufwand der VTK-Pipeline entsteht ein Overhead der durch ein schlankeres, optimiertes und Shaderbasiertes Visualisierungsverfahren eingespart werden kann. Da sich die Ausdehnung der zu visualisierenden Matrix im Normalfall nur selten ändert, kann ein durch Dreiecke tessiliertes Viereck in GPU-nahen Speicher, wie *Display Lists* oder *Vertex Buffer Objects*, gespeichert werden. Die Matrixwerte können einem Vertex-Shader in Form einer 2D-Textur übergeben werden. Dieser kann anhand der Texturwerte die Verschiebung der Vertices entlang der z-Achse und aus den umliegenden Punkten die Normalen berechnen. Anhang B listet eine mögliche Implementierung der benötigten Shader auf.

Literaturverzeichnis

[acm] *ACML - AMD Core Math Library*. http://developer.amd.com/libraries/acml/pages/default.aspx. – Accessed: 01/10/2012

[app] *APPML - AMD Accelerated Parallel Processing Math Librariesy*. http://developer.amd.com/tools/hc/appmathlibs/Pages/default.aspx. – Accessed: 04/10/2012

[atl] *ATLAS - Automatically Tuned Linear Algebra Software*. http://math-atlas.sourceforge.net/. – Accessed: 29/09/2012

[bla] *BLAS - Basic Linear Algebra Subprograms*. http://www.netlib.org/blas/. – Accessed: 29/09/2012

[cud] *CUDA - Nvidia*. http://www.nvidia.de/object/cuda_home_new_de.html. – Accessed: 01/10/2012

[fft] *FFTW - Fastest Fourier Transform in the West*. http://www.fftw.org/. – Accessed: 01/10/2012

[gnu] *GnuWin*. http://gnuwin32.sourceforge.net/. – Accessed: 01/10/2012

[Goo96] GOODMAN, J.W.: *Introduction To Fourier Optics*. Stanford Universit, 1996 (McGraw-Hill physical and quantum electronics series). – ISBN 0070242542

[got] *GotoBLAS*. http://www.tacc.utexas.edu/tacc-projects/gotoblas2. – Accessed: 01/10/2012

[gsl] *GSL - GNU Scientific Library*. http://www.gnu.org/software/gsl/. – Accessed: 01/10/2012

[Hec01] HECHT, Eugene: *Optics (4th Edition)*. Addison Wesley, 2001. – ISBN 0805385665

[HW92] HODGSON, N. ; WEBER, H.: *Optische Resonatorne: Grundlagen eigenschaften Optimierung*. Springer-Verlag, 1992 (Laser in Technik und Forschung). – ISBN 3–540–54404–6

[hzd] *HZDR - Helmholtz-Zentrum Dresden-Rossendorf*. http://www.hzdr.de/. – Accessed:

07/10/2012

[Mat] MATHWORKS: *MatLab*. http://www.mathworks.de/products/matlab/. – Accessed: 05/10/2012

[Mes08] MESCHEDE, D.: *Optik, Licht und Laser*. Teubner, 2008 (Teubner Studienbücher). – ISBN 9783835101432

[NVI12a] NVIDIA: *CUDA Toolkit 4.2 - CUBLAS Library*. 2012

[NVI12b] NVIDIA: *CUDA Toolkit 4.2 - CUFFT Library*. 2012

[NVI12c] NVIDIA: *NVIDIA CUDA C Programming Guide*. 2012

[oct] *GNU Octave*. http://www.gnu.org/software/octave/. – Accessed: 05/10/2012

[opea] *OpenBLAS*. https://github.com/xianyi/OpenBLAS. – Accessed: 01/10/2012

[opeb] *OpenCL - Open Computing Language*. http://khronos.org/opencl/. – Accessed: 04/10/2012

[opec] *OpenMP*. http://www.openmp.org/. – Accessed: 01/10/2012

[qt] *Qt*. http://qt.nokia.com/products/. – Accessed: 29/09/2012

[Sch10] SCHMIDT, J.D.: *Numerical Simulation of Optical Wave Propagation With Examples in MATLAB*. SPIE, 2010 (Press Monograph). – ISBN 9780819483263

[Sie86] SIEGMAN, A.E.: *Lasers*. University Science Books, 1986. – ISBN 0935702115

[spa] *Spatial Frequency Domain*. http://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic1.htm

[ST91] SALEH, B.E.A. ; TEICH, M.C.: *Fundamentals of photonics*. Wiley-Interscience, 1991 (Electronic and Electrical Engineering). – ISBN 0471213748

[Voe11] VOELZ, David: *Computational Fourier Optics: A Matlab Tutorial*. SPIE Press, 2011 (Tutorial Text Series). – ISBN 9780819482044

[vtk] *VTK - The Visualization Toolkit*. http://www.vtk.org. – Accessed: 29/09/2012

[Wik12] WIKIPEDIA: *Gaussian beam — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Gaussian_beam&oldid=508395851. Version: 2012. – Accessed: 05/10/2012

Abbildungsverzeichnis

Tabellenverzeichnis

A Weitere Benchmarks

Im folgenden Abschnitt werden Performance-Messungen, welche im bisherigen Teil der Arbeit noch nicht betrachtet wurden, dargestellt.

Im *Testsystem 2* ist eine NVIDIA GeForce 405 Grafikkarte mit *CUDA compute capability 1.2* verbaut, welche keine Operationen mit doppelter Genauigkeit unterstützt. Das einzige GPU-basierte Plugin welches, auf diesem System getestet werden könnte, ist das *HipnosCudaMathSinglePrecisionPlugin* (in den Legenden als *Cuda SP* bezeichnet). Das *HipnosAPPMLMathPlugin* benutzt die CPU als OpenCL-Gerät.

Der Thinkpad X61s Laptop besitzt eine Intel-Grafikkarte und profitiert somit nicht von der CUDA-Technologie. Außerdem unterstützt diese auch keine OpenCL-Schnittstelle, womit sich die auf diesem System durchgeführten Tests auf CPU-basierte Plugins beschränken.

A.1 System 2

Spezifikationen:

Prozessor	AMD Athlon II X2 255, 2x 3.10GHz
Mainboard	Fujitsu D2981-A1
Speicher	4GB DDR3-1066
Grafikkarte	NVIDIA GeForce 405
Betriebssystem	Ubuntu 11.10 Oneric Ocelot, 64-bit

Tabelle A.1: Testsystem 2 Spezifikation.

Messungen:

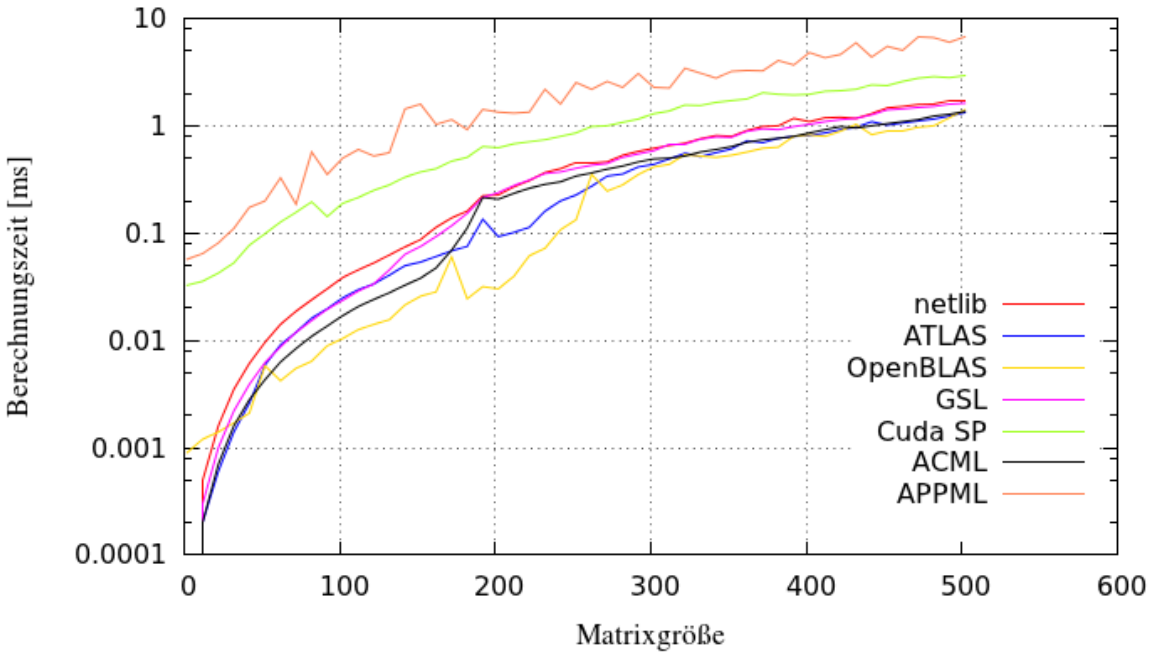

Abbildung A.1: Benchmark der Matrixaddition.

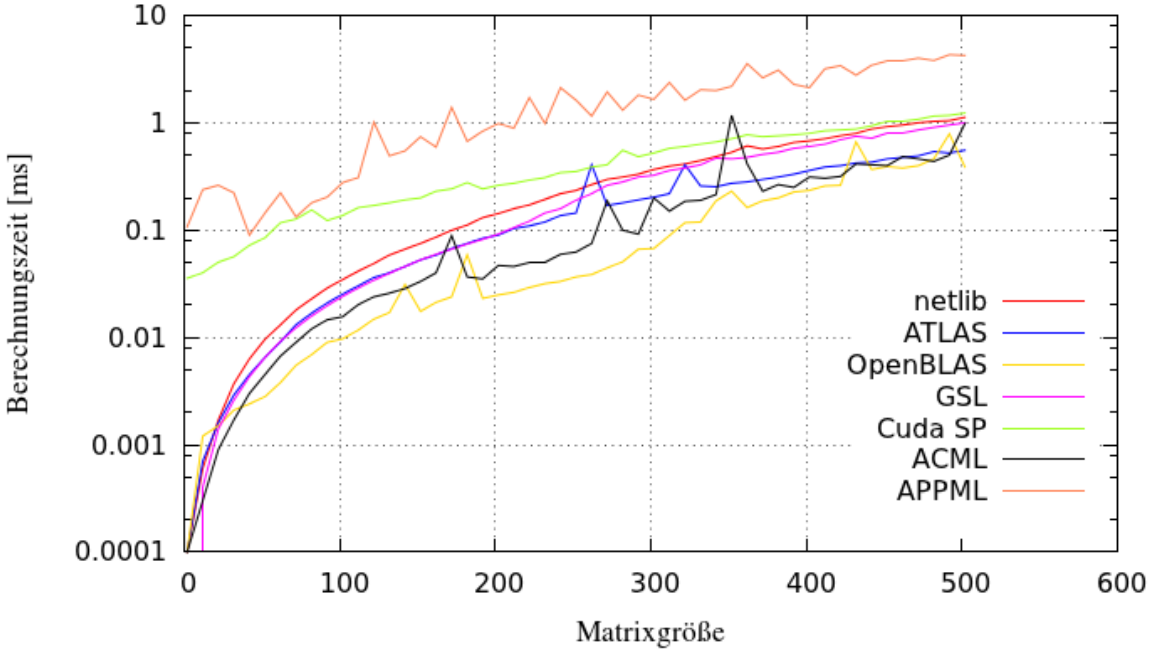

Abbildung A.2: Benchmark der Matrix-Vektor Multiplikation.

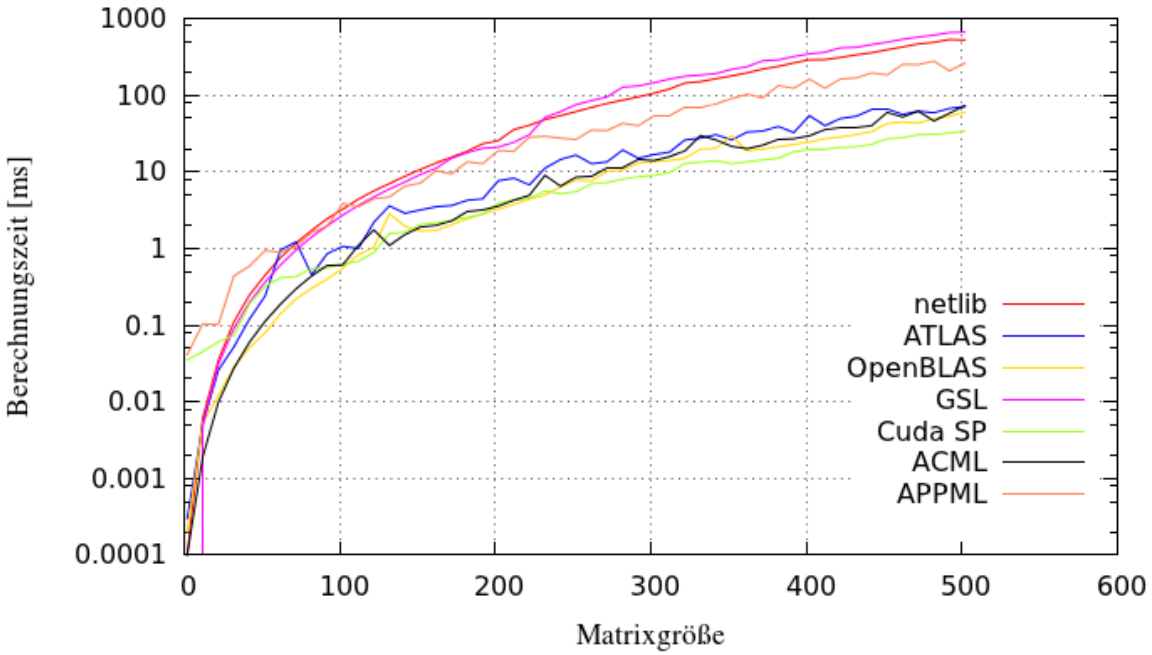

Abbildung A.3: Benchmark der Matrixmultiplikation.

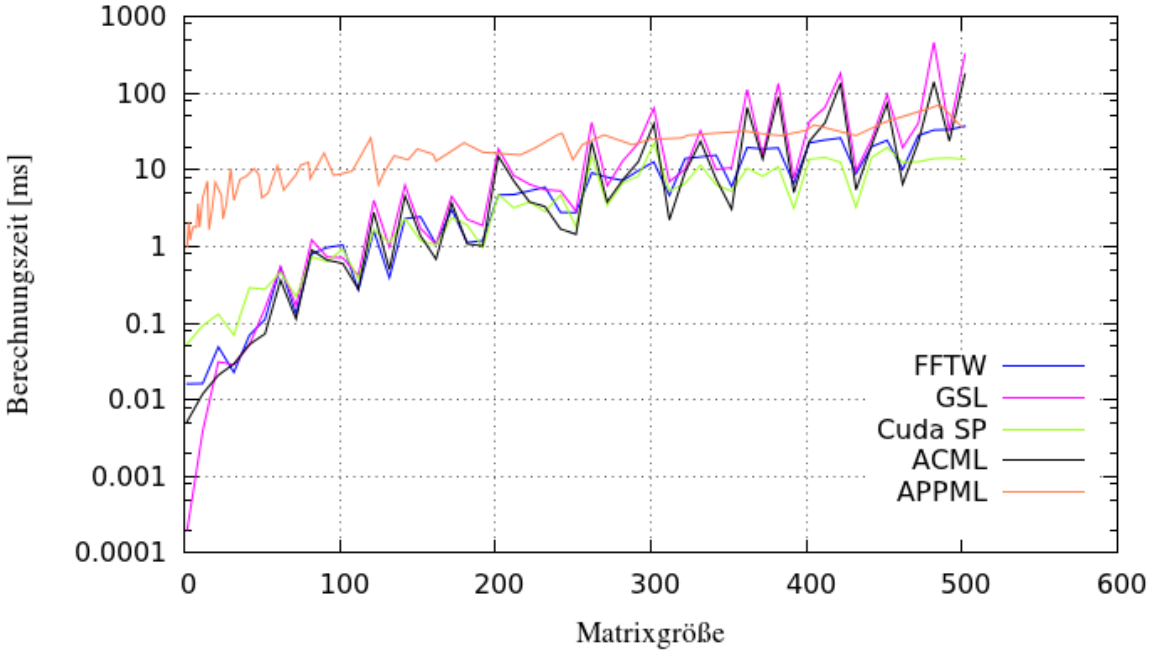

Abbildung A.4: Benchmark der FFT-Bibliotheken.

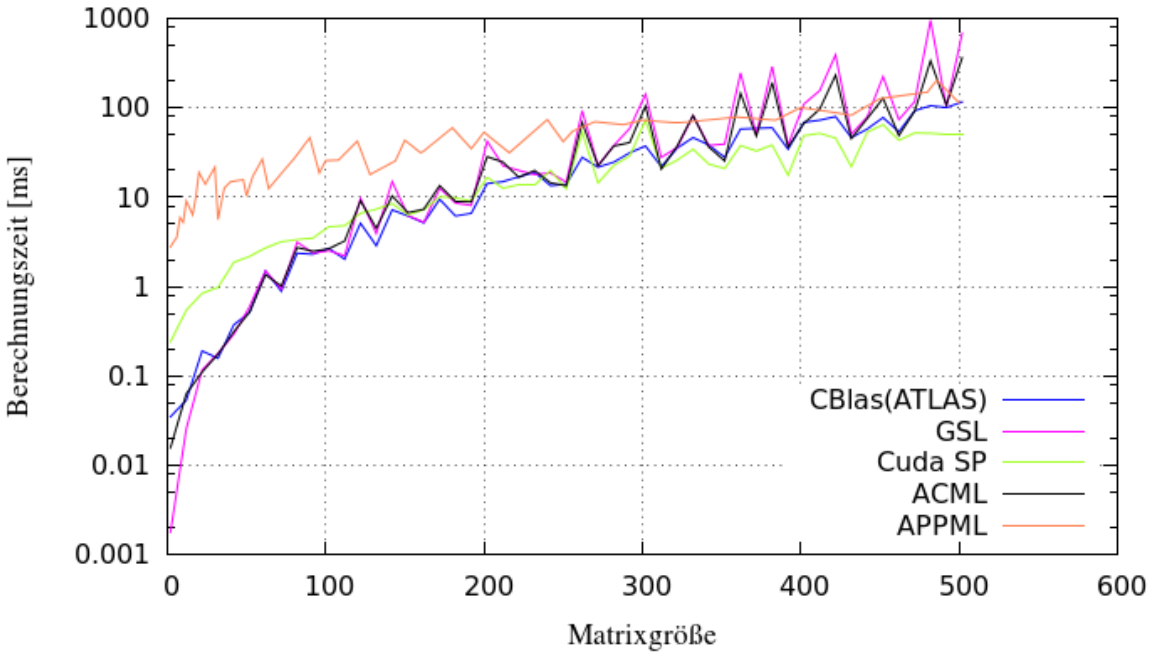

Abbildung A.5: Messung der Berechnungszeit der Propagationskomponente (Fresnelsche Näherung).

A.2 System 3

Spezifikationen:

Laptop	Lenovo Thinkpad X61s
Prozessor	Intel Core 2 Duo L7500, 2x 1.60GHz
Speicher	4GB DDR2-667
Grafikkarte	Intel Graphics Media Accelerator (GMA) X3100 128 MB
Betriebssystem	Ubuntu 12.04 LTS Precise Pangolin, 64-bit

Tabelle A.2: Testsystem 3 Spezifikation.

Messungen:

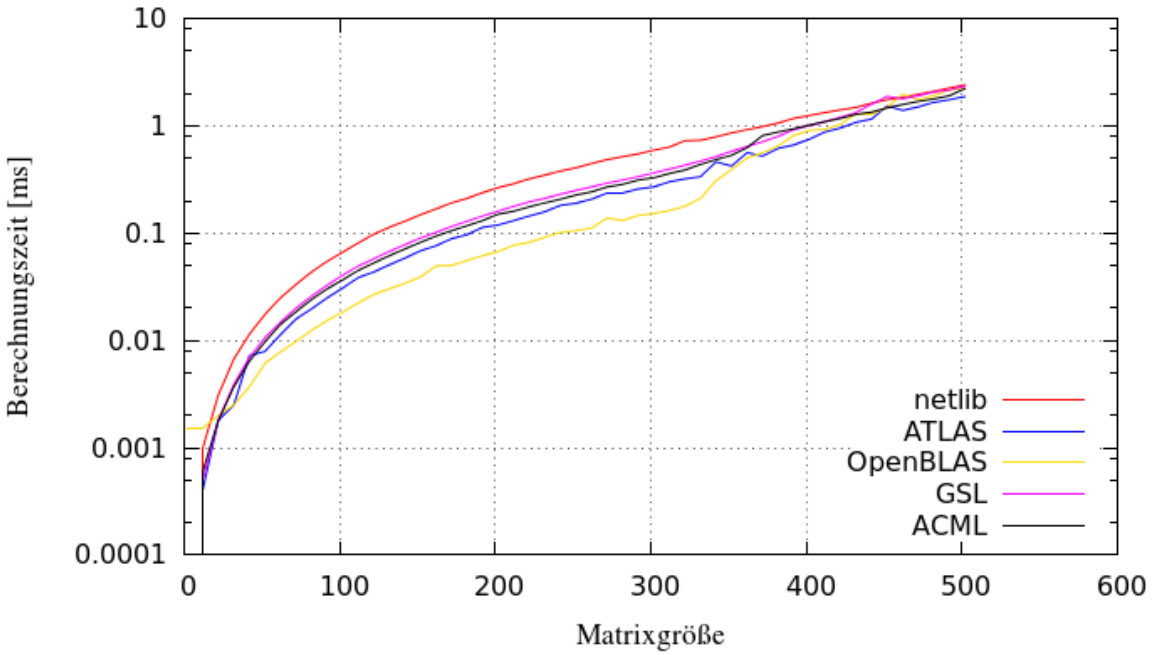

Abbildung A.6: Benchmark der Matrixaddition.

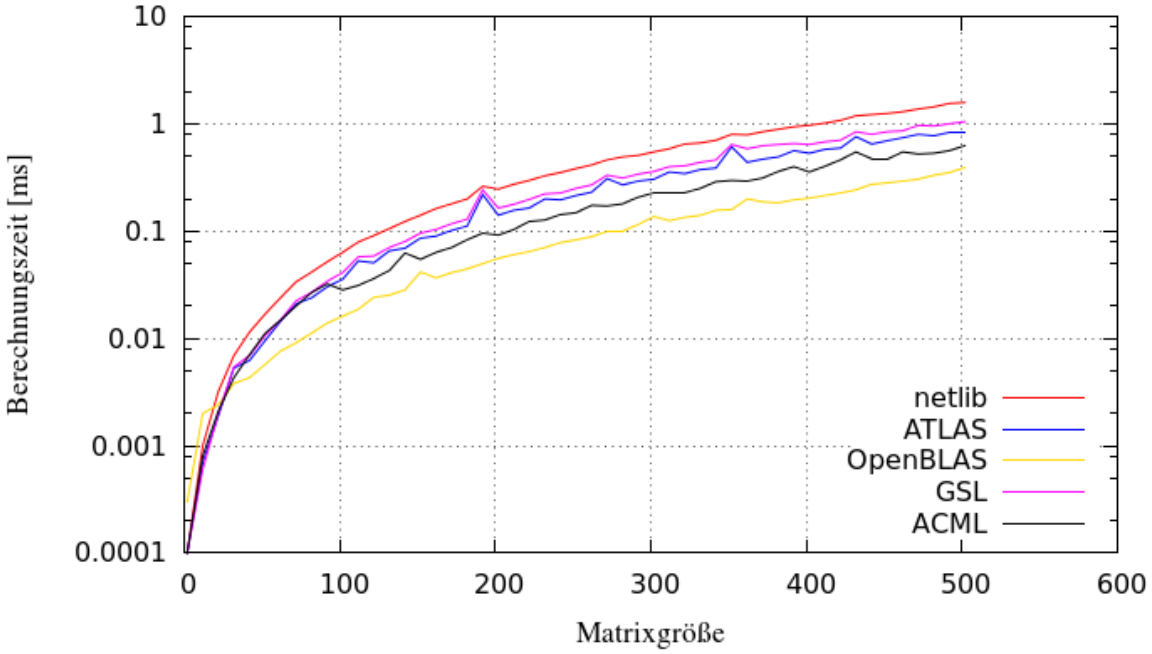

Abbildung A.7: Benchmark der Matrix-Vektor Multiplikation.

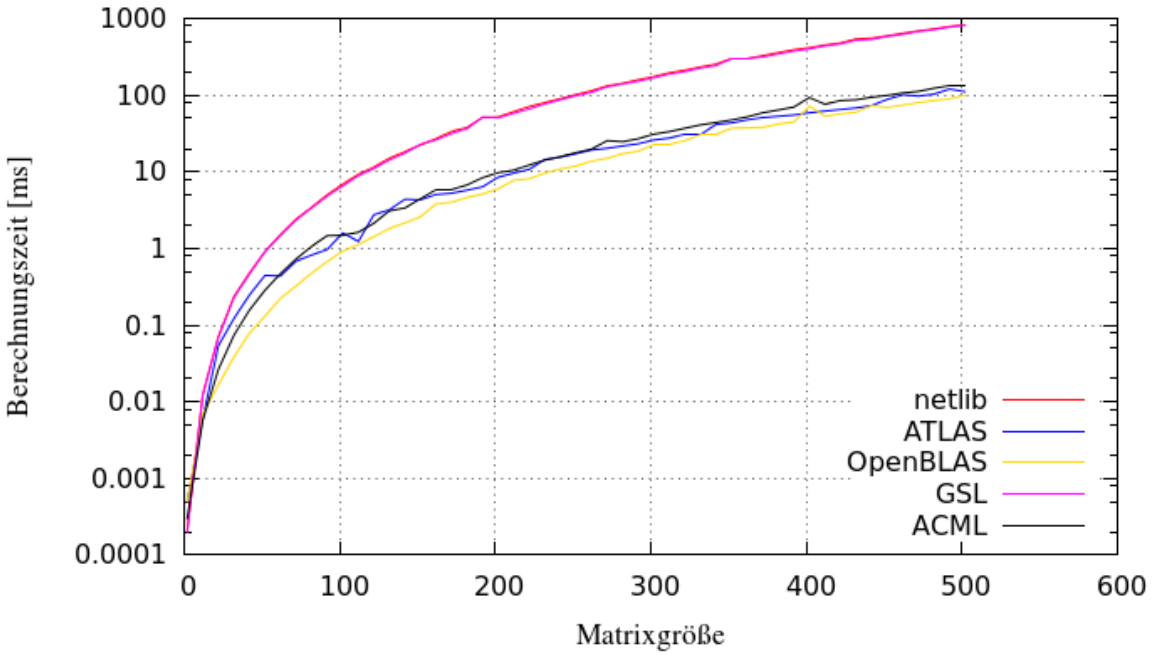

Abbildung A.8: Benchmark der Matrixmultiplikation.

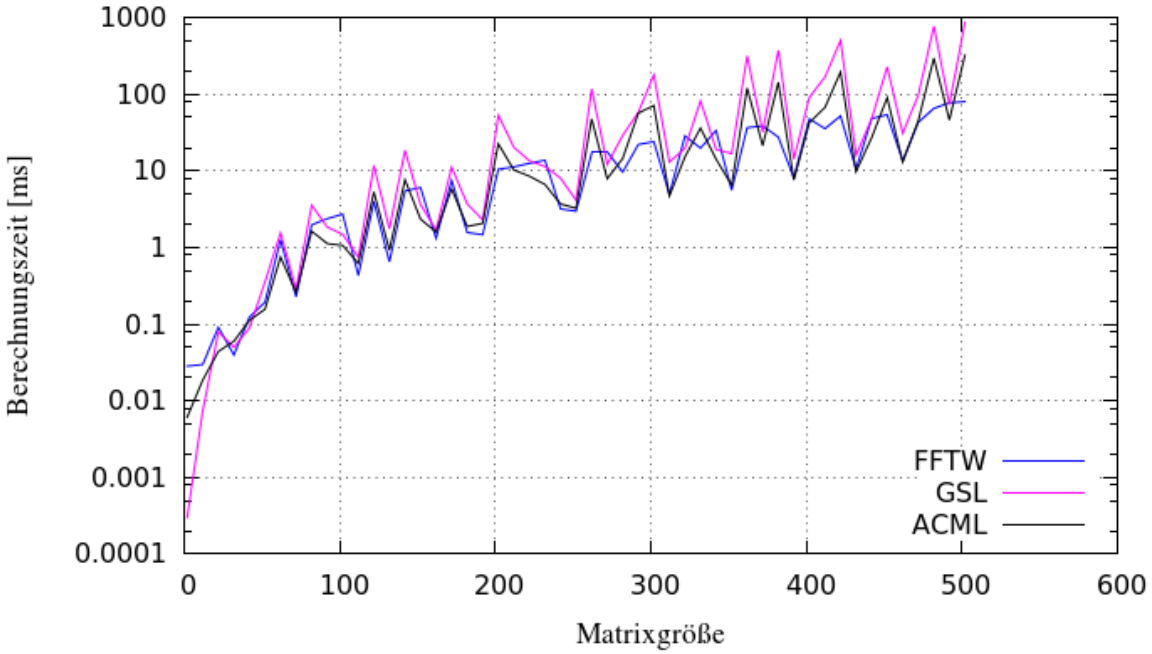

Abbildung A.9: Benchmark der FFT-Bibliotheken.

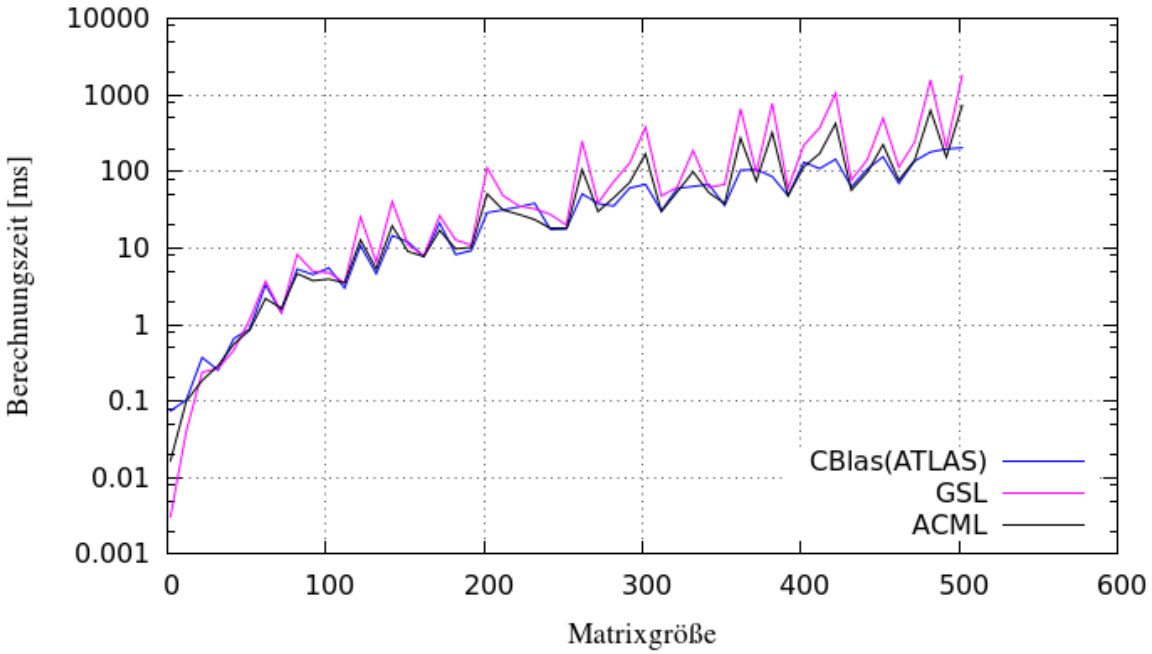

Abbildung A.10: Messung der Berechnungszeit der Propagationskomponente (Fresnelsche Näherung).

B Prototypischer Visualisierungs-Shader

```
1  // GLSL Vertex shader
2  uniform sampler2D dataImage;
3
4  varying vec4 diffuse, ambient;
5  varying vec3 normal, lightPosition;
6
7  float offsetAt(vec2 coords) {
8      return texture2D(dataImage, coords).x;
9
10 }
11
12 vec3 cross4(vec4 a, vec4 b) {
13     return cross(a.xyz, b.xyz);
14 }
15
16
17 void main() {
18
19     float size = 1.0;
20     float resolution = 5.0;
21     float texStep = 1.0/resolution;
22     float step = size/resolution;
23
24
25     float x = gl_MultiTexCoord0.x;
26     float y = gl_MultiTexCoord0.y;
27
28     // p0 is the current point, p1-6 the surrounding points
29     vec4 p0 = gl_Vertex + vec4(0.0  , 0.0  , offsetAt(vec2(x        , y     )), 0);
30
31     vec4 p1 = gl_Vertex + vec4(step , 0.0  , offsetAt(vec2(x+texStep, y     )), 0);
32     vec4 p2 = gl_Vertex + vec4(step , step , offsetAt(vec2(x+texStep, y+texStep)), 0);
33     vec4 p3 = gl_Vertex + vec4(0.0  , step , offsetAt(vec2(x        , y+texStep)), 0);
34     vec4 p4 = gl_Vertex + vec4(-step, 0.0  , offsetAt(vec2(x-texStep, y     )), 0);
35     vec4 p5 = gl_Vertex + vec4(-step, -step, offsetAt(vec2(x-texStep, y-texStep)), 0);
36
37     vec4 p6 = gl_Vertex + vec4(0.0  , -step, offsetAt(vec2(x        , y-texStep)), 0);
38
39     // Set the position of the current vertex with warping
```

```
40      gl_Position = gl_ModelViewProjectionMatrix * p0;
41
42      // Calc normals from adjacent faces
43
44      normal = cross4(p1-p0, p2-p0) + cross4(p2-p0, p3-p0) + cross4(p3-p0, p4-p0) +
45               cross4(p4-p0, p5-p0) + cross4(p5-p0, p6-p0) + cross4(p6-p0, p1-p0);
46
47      // Calculate the normal in world coordinates (multiply by gl_NormalMatrix)
48      normal = normalize(gl_NormalMatrix * normal);
49
50
51      // Calculate the light position for this vertex
52      lightPosition = normalize(gl_LightSource[0].position.xyz);
53
54      // Set the front color to the color passed through with glColor*f
55      gl_FrontColor = gl_Color * gl_LightSource[0].ambient + gl_Color * gl_LightModel.ambient;
56
57
58      // Compute the diffuse, ambient and globalAmbient terms
59      diffuse = gl_Color * gl_LightSource[0].diffuse;
60      ambient = gl_Color * gl_LightSource[0].ambient;
61      ambient += gl_Color * gl_LightModel.ambient;
62  }
```

```
1   // GLSL Fragment shader
2   varying vec4 diffuse, ambient;
3   varying vec3 normal, lightPosition;
4
5
6   void main() {
7
8       // Set the diffuse value
9       float diffuseCoeff = abs(dot(normal, lightPosition));
10
11
12      // Set the output color of our current pixel
13      gl_FragColor = ambient + diffuse * diffuseCoeff;
14  }
```