Hochschule für angewandte Wissenschaft und Kunst
Fakultät Naturwissenschaft und Technik
Göttingen

Bachelorarbeit

Entwicklung einer Software zur stereoskopischen Darstellung von Volumendaten

Ort der Durchführung:
Max-Planck-Institut für Dynamik und Selbstorganisation
Bunsenstraße 10, 37073 Göttingen

Eingereicht von:
Pascal Mues, Matrikel Nr.: 444996

1. Prüfer: Prof. Dr.-Ing. Bernd Stock, HAWK Göttingen
2. Prüfer: Dr. Ulrich Degenhardt, MPI-DS Göttingen

Göttingen, Mai 2010

Eidesstattliche Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen verwendet zu haben. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt.

…….…… ……
(Datum) (Unterschrift/en)

Einverständniserklärung

Ich bin / wir sind damit einverstanden, dass von meiner / unserer Diplomarbeit /Masterarbeit gem. §§ 16 und 17 UrhG Vervielfältigungsstücke erstellt werden können, um sie an Dritte weiterzugeben.
Mein / unser Einverständnis erstreckt sich auch darauf, dass die Diplomarbeit zu diesem Zweck an Dritte weitergegeben werden kann.

☐ einverstanden

☐ nicht einverstanden

Dauer der Sperre:

☐ Jahre

☐ unbefristet

…….…… ……
(Datum) (Unterschrift/en)

Inhaltsverzeichnis

1 Einleitung

Die einfache und verständliche Darstellung von komplexen Messdaten ist häufig ein Problem. Am Max-Planck-Institut für Dynamik und Selbstorganisation in Göttingen [Göt] werden granulare Packungen mit Hilfe eines Computertomographiesystems (Nanotom) auf ihre physikalischen Eigenschaften, wie zum Beispiel die Packungsdichte [EH02] untersucht. Ein Beispiel für eine granulare Packung ist ein Zylinder gefüllt mit gleich großen Tetraedern, die kristallähnliche Strukturen bilden. Das Nanotom liefert Bildstapel, die aus Röntgenaufnahmen der Materialprobe bestehen (siehe Abbildung 1.1). Eine Röntgenaufnahme liegen in Graustufen vor, die im Prinzip über die Dichte des Materials Auskunft geben.

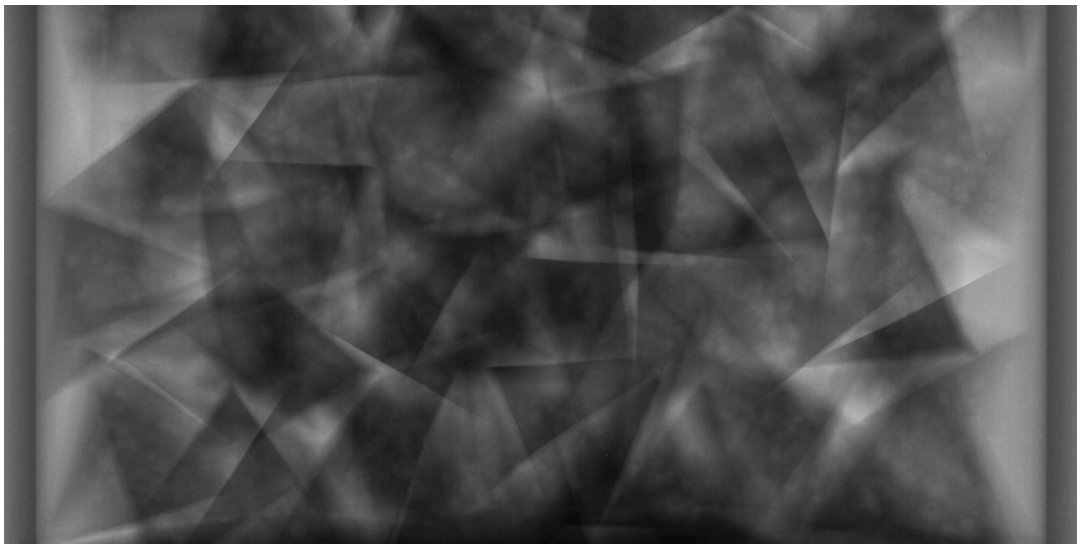

Abbildung 1.1: Radiogramm mit Nanotom aufgenommen. Im Bild zu sehen ist ein Probenzylinder gefüllt mit Tetraedern (Quelle: MPI-DS 2009)

Aus diesen Röntgenaufnahmen werden Schichtbilder der Materialprobe rekonstruiert, die zur weiteren Verarbeitung oder Anzeige verwendet werden können.
Diese Bachelorarbeit führt ein Projekt weiter, das während der Praxisprojektphase begonnen wurde. Die Praxisprojektphase ist in einer Praxisprojektarbeit [Mue09] dokumentiert und dient als Grundlage für diese Bachelorarbeit. In der Praxisprojektarbeit wurde eine Änderung an der Software VolumeRover [Cen] geplant, die eine stereoskopische Darstellung von Volumendaten ermöglichen sollte. Die Softwareänderungen sollten

im Rahmen dieser Bachelorarbeit umgesetzt werden. Wärend der Umsetzung der geplanten Änderungen wurde klar, dass der Sourcecode des VolumeRover-Projekts mit einer veralteten Shadertechnik arbeitet und das gesamte Konzept der Anwendung unzureichend dokumentiert ist. Aus diesen Gründen wurde entschieden, eine neue Software zu entwickeln, mit der die geforderte stereoskopische Betrachtungsmöglichkeit von volumetrischen Messdaten möglich ist.
Ziel dieser Bachelorarbeit ist die Neuentwicklung einer Software zur stereoskopischen Darstellung von Volumendaten. Als erstes werden die Anforderungen an die neue Software definiert. Um einen Neuentwurf sinnvoll planen zu können, ist es notwendig Wissen über die zu verwendenden Grundlagentechniken zu beschreiben. Die Gesetzmäßigkeiten und Verfahren der Stereoskopie sollen dargelegt werden, um diese im neuen Programm umsetzen zu können. Ein Überblick über gängige Methoden zur Volumen-Darstellung ist bei einer späteren Selektion im Entwurf unumgänglich. Zur 3D-Darstellung wird OpenGL verwendet, dies hat den Hintergrund, dass OpenGL unter Qt, gut auf Linux oder Mac portierbar ist. Die grundlegende Funktion und relevante Elemente von OpenGL sollen kurz erläutert werden. Wenn alle Schlüsseltechniken erläutert sind, soll eine Auswahl der Methoden stattfinden und ein erster theoretischer Entwurf der Anwendung zustande kommen. Der primäre Teil der Arbeit ist die Implementierung des erdachten Entwurfs in eine Software. Am Schluss wird das Ergebnis der Arbeit zusammengefasst und ein Ausblick, wie die Software zukünftig entwickelt wird, gegeben.

2 Anforderungen

Die Anforderungen an das Programm, das in dieser Bachelorarbeit erstellt werden soll, basieren auf einer Praxisprojektarbeit [Mue09]. Die Basisanforderungen der Praxisprojektarbeit werden mit den Anforderungen an die neu zu entwickelnde Software verknüpft. Grundlegender Gedanke hinter den folgenden Anforderungen ist die stereoskopische Darstellung von volumetrischen Messdaten, wie sie zum Beispiel in einem Röntgentomographen entstehen.
Aus der Praxisprojektarbeit werden hier folgende Anforderungen übernommen. In der Stereoskopie gibt es verschiedene Projektionsverfahren 3.1.2, es wurde festgelegt, dass vorerst nur das Farb-Anaglypheprojektionsverfahren umgesetzt werden soll. Die Benutzeroberfläche soll das Ein- bzw. Ausschalten und eine Konfiguration der stereoskopischen Projektion ermöglichen. Eine Betrachtung der Probe von allen Seiten soll möglich sein. Um den Blick auf die Probe zu verändern, soll die Maus verwendet werden. Die Struktur für Programmteile, die die stereoskopische Projektion behandeln, sollen möglichst einfach an andere stereoskopische Projektionsverfahren anpassbar sein.
Für die neu zu entwickelnde Software wurden folgende Anforderungen festgelegt. Die Volumendaten sollen aus einem Bilderstapel geladen und dargestellt werden können. Die minimale Größe des Bildstapels wurde auf 512px (Pixel) Höhe, 512px Breite und 512 Bilder Tiefe festgelegt, wobei das Laden kleiner Dimensionen ebenfalls möglich sein soll. Es soll eine Benutzeroberfläche angeboten werden, die es dem Benutzer zur Laufzeit ermöglicht, die Transferfunktion des Volumens zu verändern. Das Programm soll in einer gut verständlichen und erweiterbaren Klassenstruktur programmiert werden.
Fasst man die Anforderungen zusammen gelangt man zu folgenden Punkten:

- 3D-Darstellung möglichst groß
- Bildstapel von minimal 512x512x512 darstellbar
- Oberfläche zur Änderung der Transferfunktion zur Laufzeit
- Verständliche Programmstruktur
- Stereoskopische Darstellung zur Laufzeit Ein-/Ausschaltbar
- Farb-Anaglyphesprojektionsverfahren
- Volumen von allen Seite betrachtbar
- Gut erweiterbar um anderes Stereoprojektionsverfahren

3 Grundlagen

3.1 Stereoskopie

Die Stereoskopie [Kuh99] [Röd07] ist ein Verfahren zur optischen Darstellung, wobei nicht einfach nur ein flaches Bild wiedergegeben wird, sondern mit Hilfe spezieller Aufnahme- und Projektionstechniken ein räumliches Bild mit Tiefeneindruck erzeugt wird. Das Verfahren der Stereoskopie ist schon seit Mitte des 19ten Jahrhunderts bekannt, und im Laufe der Zeit wurde es immer wieder zeitweise zum Trend. Der erste große 3D-Boom fand in den 50er Jahren statt, hier wurde meinst das sogenannte Farb-Anagylphenverfahren zur Projektion von Kinofilmen eingesetzt, welches später noch erläutert wird. Seit einigen Jahren ist die Stereoskopie wieder auf dem Vormarsch, es werden wieder vermehrt Kinofilme in 3D gezeigt, jedoch größten Teils immernoch unter Verwendung des Farb-Anagylenverfahren. Dieser neue Boom wird zu großen Teilen computeranimierten Filmen zugesprochen. Da bei der Computeranimation ohnehin dreidimensionale Daten verarbeitet werden, ist die Aufnahme von stereoskopischen Bildern einfacher als in der realen Welt.
Die stereoskopische Sicht ist die natürliche Sicht des Menschen aus der das Gehirn Informationen über die räumliche Tiefe gewinnt. Dies geschieht durch die Zusammenarbeit der beiden Augen, die den gleichen Punkt fixieren. Aus den beiden erzeugten Bildern und dem Abstand zwischen den Augen kann das Gehirn dann den Abstand zum Fixpunkt bestimmen. Daraus ergibt sich der Grundgedanke hinter der Stereoskopie, es werden zwei Bilder aus verschiedenen Positionen aufgenommen, die jeweils für ein Auge des Betrachters bestimmt sind. Es gibt verschiedene Verfahren zur Aufnahme bzw. zur Projektion die in den nächsten Abschnitten erläutert werden.

3.1.1 Das Aufnahmeverfahren

In diesem Abschnitt soll beleuchtet werden, wie stereoskopische Bildaufnahmen entstehen. Es sollen grundlegende Begriffe definiert werden, die im Umgang mit dem Thema geläufig sind. Wie einleitend erwähnt, müssen zwei Bilder aufgenommen werden, diese werden als *Halbbilder* bezeichnet. Die Aufnahme kann entweder mit zwei Kameras oder mit einer geschehen. Die Kameras werden in einem gewissen Abstand voneinander fixiert bzw. verschoben, wenn es sich um dieselbe Kamera handelt, diesen Abstand nennt man *Stereobasis*. Aus den Maßen der Aufnahme ergibt sich das *Scheinfenster*, es ist das Fenster, welches später das Bild ergibt, der Abstand von Kamera zum Scheinfenster wird als *Scheinfensterweite* bezeichnet. Der Punkt in der Szene, der am weitesten von der

Kamera entfernt ist, nennt man *Fernpunkt*, auf ihm sind die Kameras fixiert. Aus dem Fernpunkt ergibt sich, dass der Abstand von der Kamera zum Fernpunkt als *Fernpunktweite* bezeichnet wird. Analog dazu, resultiert aus dem Objekt das sich am nächsten an der Kamera befindet, der *Nahpunkt*. Der Abstand des Nahpunktes zur Kamera wird als *Nahpunktweite* bezeichnet. Abbildung 3.1 soll den Sachverhalt verdeutlichen.

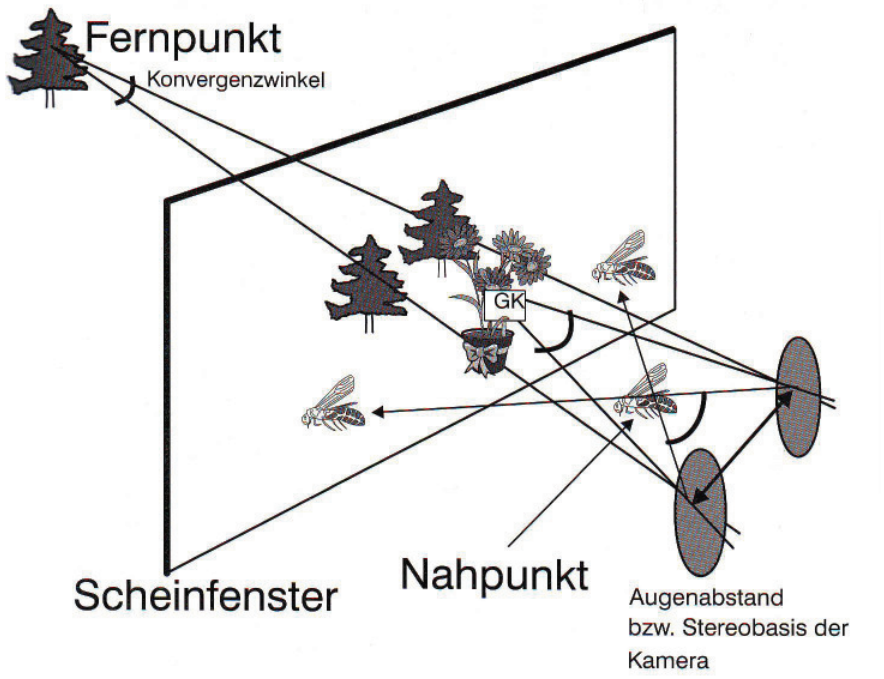

Abbildung 3.1: Der stereoskope Aufnahmeraum (Quelle: [Kuh99])

Was nun für eine Aufnahme bestimmt werden muss, ist die Stereobasis und die Fernpunktweite, alles andere ergibt sich aus der Szene heraus. Es gilt hier folgender mathematischer Zusammenhang:

$$A = p \cdot v \tag{3.1}$$

Diese Gleichung steht für den Zusammenhang zwischen Aufnahme und Projektion, für eine richtige Aufnahme muss man also schon vorher wissen, wie die Projektion gestaltet ist. Der durchschnittliche Augenabstand A beträgt im Durchschnitt 6,3cm [Kuh99], p wird durch die Projektion vorgegeben und berechnet sich zum Beispiel wie folgt:

$$p = \frac{x'}{x} = \frac{1.75m}{35mm} = 50fach \tag{3.2}$$

wobei x' die maximale Bildbreite auf der Leinwand und x die Größe des Dia.

Normal Aufnahme Für die *normale Aufnahme*, das heisst bei Aufnahmen bei dem der Hintergrund weit entfernt und Scheinfensterweite s gleich der Nahpunktweite n ist, gilt folgende Formel:

$$S = v \cdot \left(\frac{s}{f} - 1 \right) \tag{3.3}$$

S bezeichnet hier die Stereobasis, s die Scheinfensterweite und f die Brennweite des Objektives. Der Fernpunkt liegt bei dieser Formel im Unendlichen, das bedeutet die Blickachsen beider Halbbilder verschieben sich hier parallel.

Nahaufnahme Liegt ein aufzunehmendes Objekt vor der Nahpunktweite, muss bei der Formel 3.3 für die Stereobasis zusätzlich beachtet werden, dass der Fernpunkt nicht mehr im Unendlichen liegt. Für die Fernpunktweite w gilt folgende Formel:

$$w = \frac{n}{1 - v \cdot \frac{n-f}{f \cdot S}} \tag{3.4}$$

daraus ergibt sich für die Stereobasis bei der *Nahaufnahme*:

$$S = \frac{v \cdot n - f}{f \cdot \left(1 - \frac{n}{w}\right)} \tag{3.5}$$

wobei n hier die Nahpunktweite und w die Fernpunktweite bezeichnet. Für die Herleitung der Formeln in diesem Abschnitt wird auf [Kuh99] verwiesen. Im Anhang unter A.2 ist ein ein Beispiel für Stereoaufnahmen, die mit Hilfe einer Kamera gemacht wurden, zu finden, in dem die Formeln für die normal Aufnahme zur Anwendung gekommen sind.

Makroaufnahme Bei der *Makroaufnahme* wird der Fernpunkt ins Unendliche gelegt und die Sicht-Axen beider Halbbilder sind wieder parallel. Die entscheidene Variable ist wieder die Stereobasis mit der aufgenommen wird. Da der Fernpunkt im Unendlichen liegt, muss die Stereobasis soweit verkleinert werden, dass Nahpunktweite und Scheinfensterweite gleich groß sind. Es gilt folgende Gleichung:

$$S = v \cdot \left(\frac{n}{T} + 1 \right) \cdot \left(\frac{n}{f} - 1 \right) \tag{3.6}$$

Da T die Tiefenausdehnung gleich w - n ist und die Fernpunktweite w gleich unendlich, kann die Gleichung vereinfacht werden:

$$S = v \cdot \left(\frac{n}{f} - 1 \right) \tag{3.7}$$

Diese Vereinfachung verringert jedoch die Tiefenwirkung der Aufnahme.

3.1.2 Verfahren zur Stereoprojektion

Es gibt verschiedene Verfahren um stereoskopisch aufgenommene Bilder darzustellen. Das bekanntesten Verfahren wird in diesem Abschnitt vorgestellt, auf das Farb-Anaglyphenverfahren wird besonders eingegangen, da dies im Projekt verwendet wird.
Generell verlaufen die Verfahren zur Projektion immer gleich. Man erzeugt zwei Halbbilder, die auf einer Fläche übereinander projiziert werden. Um den räumlichen Eindruck zu erzeugen müssen die Halbbilder jedoch den Augen zugeordnet werden. Diese Zuordnung erfordert, bei den meisten Verfahren, eine Instrumentierung des Benutzers, der eine Brille tragen muss, welche die Halbbilder wieder von einander trennt. Diese Trennung geschieht mit Hilfe von optischen Filtern, die als Brillenglas verwendet werden, diese varieren je nach Projektionsverfahren.
Eine Schwierigkeit bei allen Projektionsverfahren ist es die Filter von Projektion und Brillen richtig abzustimmen. Durch ungleiche Filterung entstehen sogenannte *Ghosting-effekte*. Das bedeutet, dass ein Halbbild nicht vollständig weggefiltert wird und eine Art Schatten im anderen Halbbild erzeugt.

Farb-Anaglyphenverfahren Das wohl bekannteste und älteste Projektionsverfahren ist das *Farb-Anaglyphenverfahren*. Dies kann in einem bestehenden System, wie zum Beispiel einem Kino oder Hörsaal, umgesetzt werden. Hier ist der ausschlaggebende Punkt, dass man als zusätzliche Hardware lediglich die Filterbrille benötigt. Ein weiterer Grund für die weite Verbreitung des Verfahrens ist, dass es mit geringsten finanziellen Mitteln angewendet werden kann.
Beim Farb-Anaglypenverfahren wird meist mit Rot und Cyan gefiltert, früher auch mit Rot und Grün, das heisst es gibt für jedes Halbbild einen Projektor mit einem vorgeschalteten Filter. Diese werfen die beiden Halbbilder übereinander auf eine Projektionswand. Der Betrachter trägt eine Brille mit zwei Filtern die analog zu den Filtern der Projektoren sind, Rot und Cyan. Der Rotfilter ist undurchlässig für das Rote Halbbild, somit wird das rote Halbbild nur von dem Auge hinter dem Cyanfilter wahrgenommen und umgekehrt das cyangefärbte Halbbild nur von dem Auge hinter dem Rotfilter. Es ist auch möglich die Darstellung mit einem Projektor zu realisieren, indem man die Halbbilder übereinander projeziert. Das Übereinanderlegen der Halbbilder kann, wenn die Halbbilder in digitaler Form auf einem Computer vorliegen, durch Addieren der Farbkomponente, Pixel für Pixel, beider Halbbilder realisiert werden.
Der Nachteil dieses Verfahrens ist, dass durch die Farbfilterung die Farben der Halbbilder falsch dargestellt werden. Für Bilder, bei denen es auf korrekte Farbdarstellung ankommt, ist dieses Verfahren ungeeignet, da eigentlich nur Graustufen korrekt zu erkennen sind. Wobei angemerkt werden muss, dass die Farbunterscheidung im Vergleich zu anderen Filterkombinationen, wie zum Beispiel Rot/Grün [Kuh99], wesentlich besser ist.

3.2 Volumengrafik

3.2.1 Allgemein

Die *Volumengrafik* [Had06] wird genutzt Objekte räumlich darzustellen, sie ist ein Fachgebiet der 3D-Computergrafik. Mit Hilfe der Volumengrafik ist es möglich, das Innere eines Objekts darzusellen. Es existieren spezielle Verfahren zum Erzeugen von Volumengrafik. Die Volumengrafik-Darstellung lässt sich in Schritte [Had06] unterteilen, wie in Abbildung 3.2 gezeigt.

Abbildung 3.2: Schritte der Volumengrafik-Darstellung.

Rasterung Um das Innere eines Objekts darzustellen, müssen zuvor Messdaten gewonnen werden die eine Aussage über das Innere ermöglichen. Messdaten über das Objektinnere können zum Beispiel aus einem Röntgenthomographen stammen. Durch spezielle mathematische Rekonstruktionsverfahren [LFK83] werden die Daten aufbereitet. Diese Verfahren basieren auf dem *Gesetz der Strahlungsenergie* [Had06],

$$I = \frac{dQ}{dA_{\perp}} d\Omega dt \tag{3.8}$$

und dem *Volumen-Rendering-Integral* [Had06] das in Gleichung 3.9 zu sehen ist.

$$I(D) = I_0 \exp^{-\int_{s_0}^{D} \kappa(t)dt} + \int_{s_0}^{D} q(s) \exp^{-\int_{s}^{D} \kappa(t)dt} ds \tag{3.9}$$

Eine detailierte Behandlung der Gleichungen ist in [Had06] zu finden. Das Ergebnis der Rekonstruktionsverfahren ist ein dreidimensional gerastertes Gitter des Objekts. Es findet also eine Diskretisierung statt, welche als *Rasterung* bezeichnet wird.

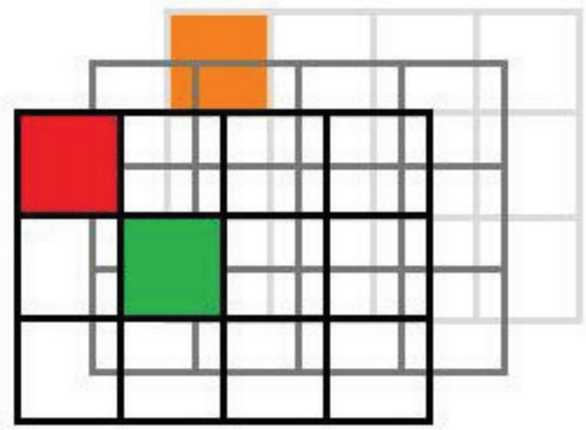

Abbildung 3.3: Voxelgitter eines Volumens.

Eine Zelle dieses Gitters wird als *Voxel* bezeichnet. Ein Voxel kann in diesem System über seine Koordinaten x, y, z eindeutig identifiziert werden. Jeder Voxel enthält zusätzlich zu seinen Koordinaten einen Messwert, zum Beispiel die Röntgendichte. Das Voxel-Gitter gibt also Informationen über das Objektinnere wieder. Von nun an wird dieses Voxel-Gitter als *Volumen* bezeichnet.

Interpolation Aufgabe der Volumengrafik ist es nun, mit Hilfe einer 3D-Grafikschnittstelle wie *OpenGL* dieses Volumen darzustellen. Auf OpenGL wird in Abbschnitt 3.3 eingegangen. Da der Renderaufwand um alle Voxel, zum Beispiel $512^3 = 134217728$, des Volumens anzuzeigen enorm wäre, findet eine Interpolation zwischen den einzelnen diskretisierten Werten statt. Der Typ der *Interpolation* hängt vom jeweilig verwendeten Verfahren ab.

Klassifizierung Bei der Darstellung in einer 3D-Grafik werden den Messwerten des Volumens Farb- und Transparenzwerte zugeordnet. Diese Zuordnung wird als *Transferfunktion* bezeichnet.

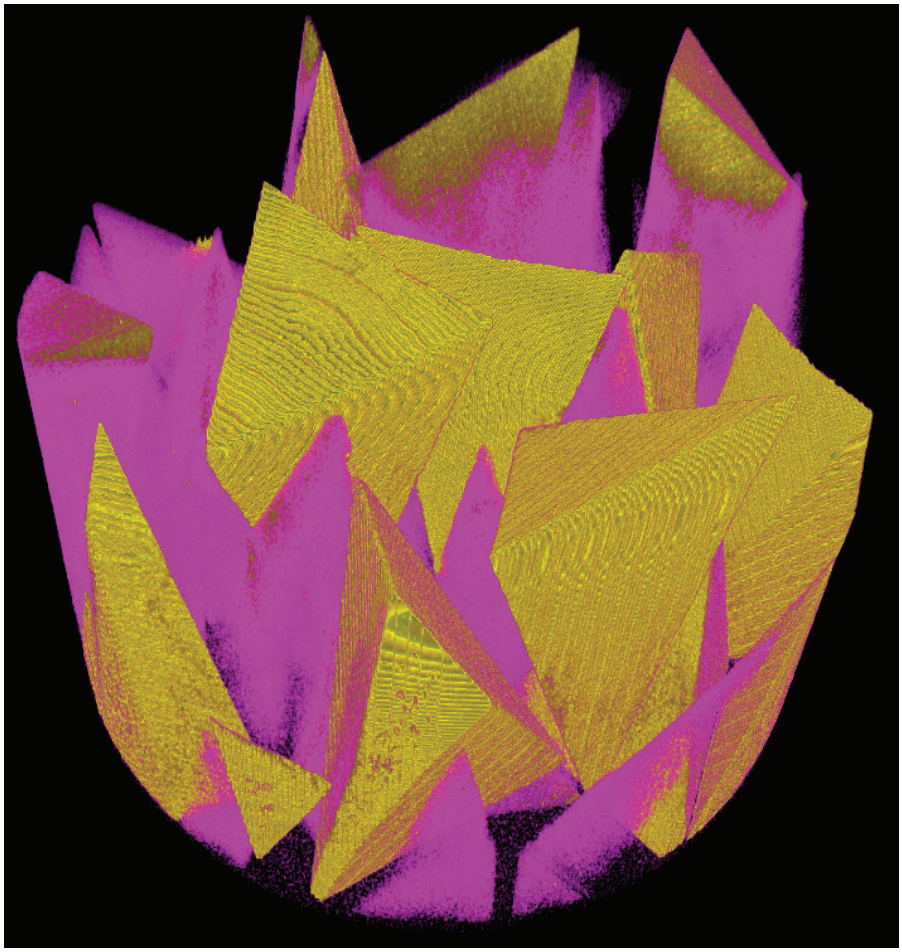

Abbildung 3.4: Volumengrafik einer Tetreaderpackung. In der Probe befinden sich zwei Klassen Tetraeder die aus Materialen mit unterschiedlicher Dichte gefertigt sind. Eine Klasse ist Violett eingefärbt, die andere Gelb.

Wie im Abbildung 3.4 zu sehen sind Gebiete mit geringer Röntgendichte, wie die Freiräume zwischen den Tetraedern, transparenter als Gebiete mit hoher Röntgendichte. Diese Unterscheidung wird als *Klassifizierung* bezeichnet. Eine Klassifizierung wird mit Hilfe einer Transferfunktion realisiert. Es gibt ein- bis dreidimensionale Transferfunktionen mit verschiedenen Dateninterpolationen, näheres hierzu in [Had06]. Die eindimensionale Transferfunktion soll in Abschnitt 3.2.2.1 näher erläutert werden.

Shading In der Übersetzung bedeutet *Shading* soviel wie Schattierung. In der Volumengrafik wird hier berechnet in welcher Farbe und wieviel Licht ein Voxel ausstrahlt. Es gibt verschiedene *Beleuchtungsmodelle*, ein Überblick ist in [Had06] zu sehen, ein weit verbreitetes ist das *Blinn-Phong Modell* auf welches Abschnitt 3.2.2.2 eingegangen wird.

Composing *Composing* bedeutet soviel wie Zusammensetzen oder Mischen. Hier wird, abhängig von der Position des Betrachters, ein Bild auf das Volumen zusammengesetzt. Für jeden Bildpunkt wird sozusagen ein Strahl durch das Volumen geschickt auf dem die Farbe, abhängig von der jeweiligen Transparenz, gemischt wird.

3.2.2 Methodenüberblick

3.2.2.1 Eindimensionale Transferfunktionen

Die *eindimensionale Transferfunktion* gilt für jeden Voxel im Volumen gleichermaßen, daher die Bezeichnung eindimensional. Wie schon erwähnt werden den Datenwerten, der Voxel, mit der Transferfunktion Farb- und Transparenzwerte zugeordnet. Hierfür ist es erforderlich das die Transferfunktion den gesamten Wertebereich der Datenwerte abdeckt und so eine eindeutige Zuordnung der Werte ermöglicht. In der Praxis wird eine Transferfunktionen aus sogenannten *Knoten* generiert. Knoten sind hier als Stützstellen zu verstehen, zwischen denen linear interpoliert wird. Aus der Forderung dass der Wertebereich immer abgedeckt sein muss folgt, dass der Anfangs- und Endknoten immer festgelegt sein müssen. Das folgende Beispiel soll das Prinzip veranschaulichen.

Beispiel Transferfunktion Angenommen der Wertebereich der Daten, die in den Voxeln definiert sind reicht von 0 bis 255. Daraus folgt dass die zugehörige Transferfunktion auch einen Bereich von 0 bis 255 abdecken muss. Um das Beispiel anschaulicher zu machen, werden zunächst *Farb- und Transparenzknoten* definiert 3.1 die dann analysiert werden sollen.

Knoten	**Datenwert**	**Rot**	**Grün**	**Blau**	**Sichbarkeit**	**Resultierende Farbe**
K1	0	0	0	0	0%	Schwarz
K2	99	0	0	0	0%	Schwarz
K3	100	0	255	0	100%	Grün
K4	200	255	0	0	100%	Rot
K5	201	0	0	0	0%	Schwarz
K6	255	0	0	0	0%	Schwarz

Tabelle 3.1: Knoten der Beispiel Transferfunktion

In diesem Beispiel haben die beiden Teilfunktionen für Farbe und Transparenz die gleiche Anzahl von Knoten, diese könnten sich auch unterscheiden. Die Transferfunktion, die durch diese Knoten definiert wird, legt fest dass ein Teilbereich des Wertebereichs zwischen 100 und 200 sichtbar ist. Im Teilbereich wird für die Datenwerte ein Farbverlauf von Grün zu Rot interpoliert. Abbildung 3.5 zeigt wie sich die resultierende Transferfunktion zusammensetzt.

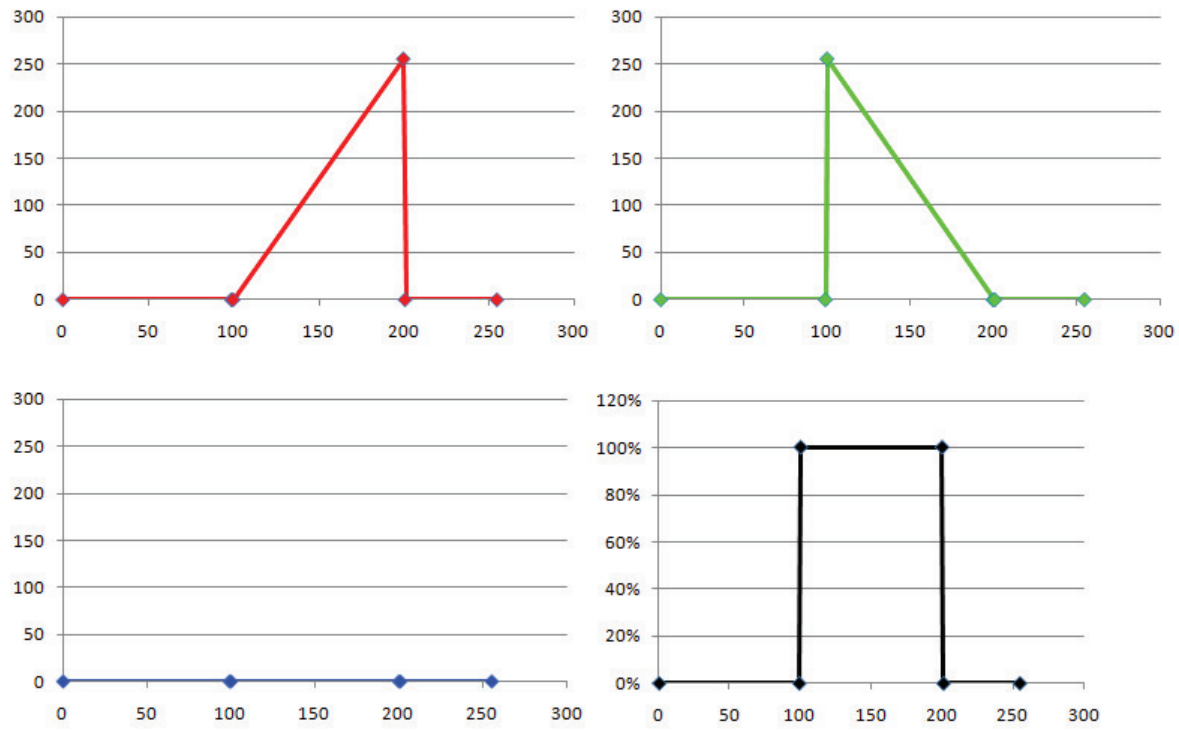

Abbildung 3.5: Darstellung der einzelnen Komponenten der Transferfunktion. Links oben der Rot-Kanal, rechts oben der Grün-Kanal, links unten der Blau-Kanal und rechts unten der Alpha-Kanal der Transferfunktion.

Abbildung 3.6: Beispiel Transferfunktion als Grafik. Die X-Achsen definieren den Datenwert, die Y-Achsen jeweils Rot, Grün, Blau und Alpha (Schwarz). Oben ohne Alpha-Kanal. Unten mit Alpha-Kanal, die schwarzen Ränder der Transferfunktion werden Transparent.

Die schwarzen Ränder der Transferfunktion werden durch den *Alphakanal* ausgeblendet. Diese Ausblendung sorgt in der Praxis dafür das Voxel, deren Datenwert zwischen 0 und 99 oder zwischen 201 und 255 liegen, vollständig transparent sind. Angenommen das Beispiel Volumen besitzt einen Datenwert von 125, so wird ihm ein Transparenzwert von 100%, ein Rotwert von 191, ein Grünwert von 64 und ein Blauwert von 0 zugeordnet.

3.2.2.2 Blinn-Phong Beleuchtungsmodell

Das *Blinn-Phong Modell*[Had06] gehört zu den *lokalen Beleuchtungsmodellen*. Lokale Beleuchtungsmodelle berücksichtigen nur den Lichtanteil, der auf einen Punkt fällt, der direkt von der Lichtquelle ausgesandt wird. Jeder Punkt ist unabhängig von allen anderen Punkten beleuchtet. Durch lokale Beleuchtungsmodelle können keine Schattierungen oder Spiegelungen abgebildet werden. Um Schattierungen oder Spiegelungen abzubilden müsste einfallendes Licht von anderen Punkten berücksichtigt werden, lokale Beleuchtungsmodelle tun dies nicht. Da der mathematische Hintergrund zu lokalen Beleuchtungsmodellen in [Had06] ausführlich beschrieben ist, es soll hier nicht weiter drauf eingegangen werden.

Das Licht, das von einem Punkt aus reflektiert wird, ist durch Gleichung(3.10) definiert und bildet die Basis des Blinn-Phong Modells.

$$I_{Phong} = I_{Ambiente} + I_{Diffuse} + I_{Spekular} \tag{3.10}$$

Das reflektierte Licht besteht aus drei Komponenten, die unterschiedliche Reflektionsmerkale des Materials wiedergeben. Für den *ambienten Teil* der Reflektion gibt es keinen physikalischen Hintergrund, er wird aus praktischen Gründen eingesetzt um nicht jeden Punkt, der evtl. nicht beleuchtet würde, mit einer zusätzlichen Lichtquelle zu beleuchten. Der ambiente Reflektionsanteil entspricht einer globalen Beleuchtung. Gleichung(3.11) zeigt wie der ambiente Teil bestimmt wird.

$$I_{Ambiente} = k_a M_a I_a \tag{3.11}$$

k_a ist hier eine frei wählbare Konstante die angibt wieviel ambientes Licht von dem Punkt reflektiert werden soll. M_a ist die ambiente Materialfarbe. I_a ist die globale ambiente Lichtfarbe. Die hier definierten Farben werden komponentenweise(RGB) multipliziert. Hinter dem *diffusen* und dem *spekularen Anteil* verbergen sich physische Reflektionen, diese werden in Abbildung 3.7 dargestellt.

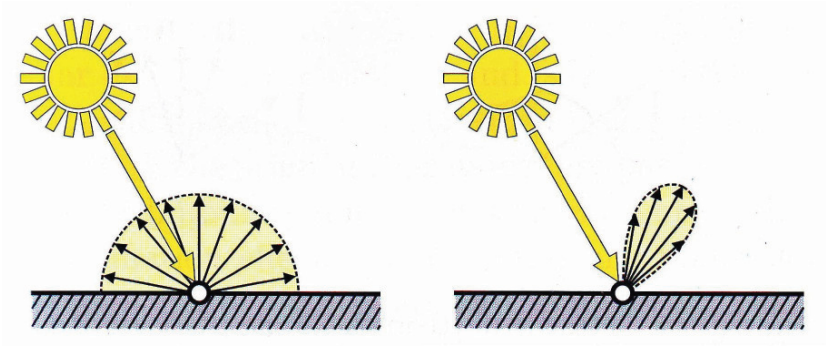

Abbildung 3.7: Das Prinzip der Diffusenreflektion(links) und das Prinzip der Specularenreflektion(rechts). (Quelle: [Had06])

Bevor diffuser und spekularer Lichtanteil durch Gleichungen beschrieben werden, soll die zu Grunde liegende Geometrie, Abbildung 3.8, dargestellt werden.

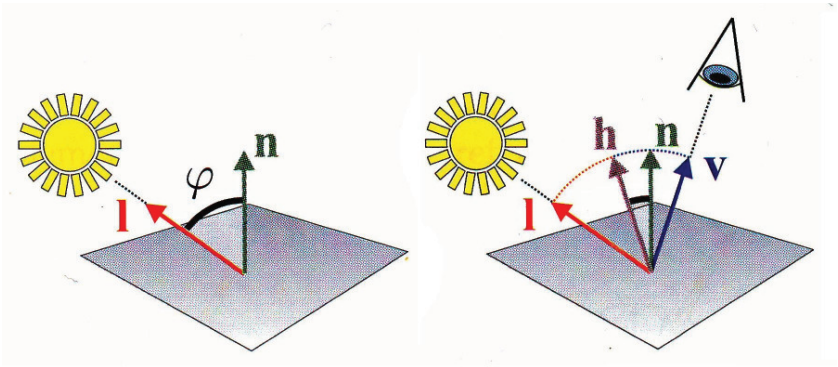

Abbildung 3.8: Vektorgeometrie der Diffusenreflektion(links) und Vektorgeometrie der Spekularenreflektion(rechts). (Quelle: [Had06])

Der Vektor $\vec{n}$ ist die Normale die senkrecht auf dem reflektierenden Punkt steht. $\vec{l}$ ist der Vektor der das auf den Punkt fallende Licht definiert. $\vec{v}$ definiert den Vektor von dem aus der Punkt betrachtet wird. Der Vektor $\vec{l}$ liegt auf halben Weg zwischen $\vec{l}$ und $\vec{v}$, wird aber als Symbol nicht weiter verwendet. Gleichung(3.12) zeigt die Berechnungsvorschrift für den diffusen Reflektionsanteil, Gleichung 3.13 die für den specularen Anteil.

$$I_{Diffuse} = k_d M_d I_d \cos(\varphi) = k_d M_d I_d max((r \cdot v), 0) \tag{3.12}$$

$$I_{Specular} = k_s M_s I_s \cos^n(\rho) = k_s M_s I_s (normalize(v + l) \cdot n)^n \tag{3.13}$$

Alle Teile von Gleichung(3.10) sind somit beschrieben und die Reflektion kann berechnet werden.

3.2.2.3 Das Slicing-Verfahren

Slicing [Had06] ist ein Verfahren zur geometrischen Darstellung eines Volumens. Es wird eine Geometrie definiert auf der die Volumendaten sichtbar gemacht werden können. Diese Geometrie besteht aus Schnitten durch den Raum des Volumens. Von nun an werden diese Schnitt als Slices bezeichnet. In Abbildung 3.9 ist eine Slice-Geometrie dargestellt.

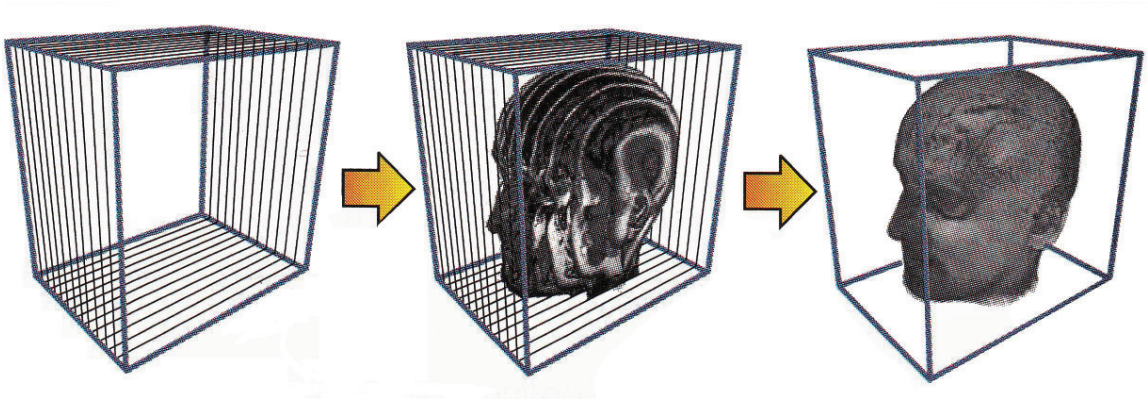

Abbildung 3.9: Slice-Geometrie zur Darstellung eines Volumens. (Quelle: [Had06])

Slices werden durch Polygonen erzeugt. Der Begriff *Polygon* wird in Abschnitt 3.3 erläutert. In einem definierten Abstand werden die Slices durch das Volumen gezeichnet. An den Positionen, an denen die Slices die Volumendaten schneiden, werden die Daten für den Betrachter sichtbar.
Nachteil des Verfahrens ist, dass die Slices in der fertigen Volumendarstellung sichtbare *Aliasing-Artefakte*, in Form von Strichen die über Objekte laufen, erzeugen. Vorteil des Verfahrens ist, dass das Prinzip leicht vorstellbar ist und einfacher umgesetzt werden kann als andere Verfahren.

3.3 OpenGL

3.3.1 3D Grafik mit OpenGL

Die Grundlage für eine geometrische Form ist ein *Koordinatensystem* in dem Punkte eines Objekts definiert werden können. In OpenGL [Gro] [Wol09] [JB09] oder auch DirectX [Cora] wird ein kartesisches Koordniatensystem mit 3 Achsen, X, Y und Z, verwendet.

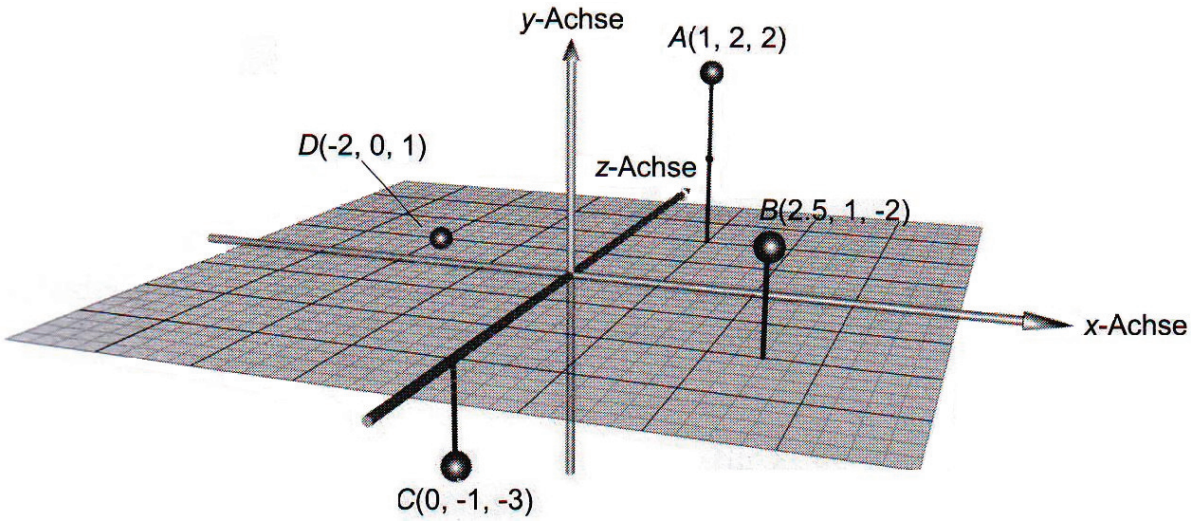

Abbildung 3.10: 3D Koordinatensystem, Grafik entnommen aus [Sch06]

Der Koordinatenursprung in OpenGL liegt hat die Koordinaten X=0, Y=0 und Z=0. Ein Punkt im Raum wird in der Computergrafikwelt als *Vertex* bezeichnet, genauer dienen Vertex als Eckpunkte, zwischen denen weitere Punkte interpoliert werden können. Ein Vertex kann neben den drei Koordinaten noch andere Informationen enthalten, wie zum Beispiel welche Farbe er hat oder zu welcher Primitive er gehört. Eine *Primitive* ist ein Element der Vektorgrafik welches Informationen über die geometrische Form eines Objekts speichert, ein Bauplan wie die Vertizes mit einander verbunden werden. Ein Kreis würde in Vektorgrafik zum Beispiel durch seine Mittelpunkt und seinen Durchmesser definiert und könnte noch zusätzliche Informationen bereit stellen. In Rastergrafik würde sich der Kreis erst aus dem gesamten Bild aller Pixel ergeben. OpenGL stellt solche Primitiven als Grundlage zur Verfügung, die Auswahl an Primitiven ist jedoch begrenzt auf einige Grundtypen aus denen komplexere Gebilde zusammengesetzt werden können. Diese Grundtypen sind im folgenden Bild aufgelistet:

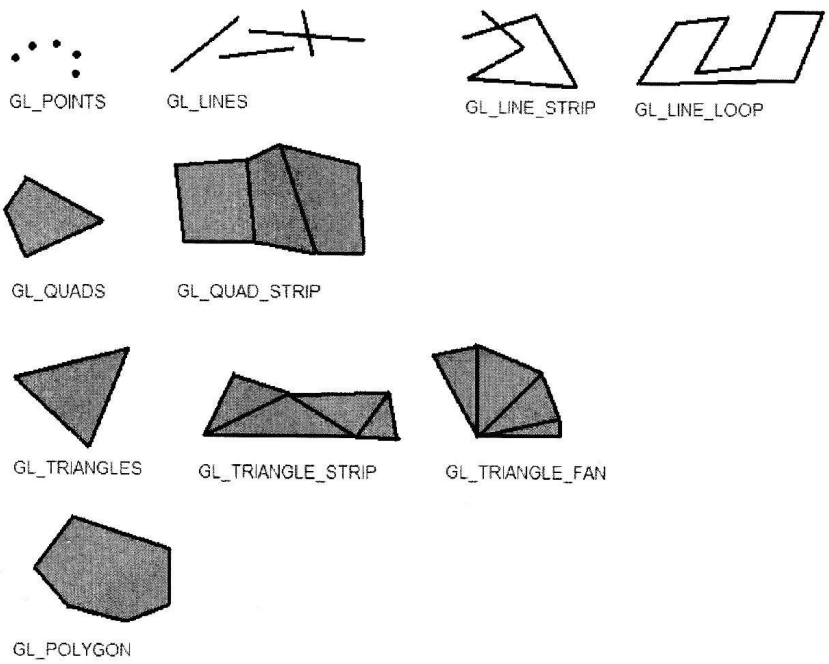

Abbildung 3.11: OpenGL Zeichenkonstanten für Primitiven. (Quelle: [Cha08])

Werden Gebilde aus Primitiven in einem Koordinatensystem dargestellt, nennt man dies *Szene*.

3.3.2 Die Kamera

In diesem Abschnitt soll erklärt werden wie mit OpenGL ein ausgegebenes Bild erzeugt wird. Das Erzeugen eines Bildes geschied mit Hilfe einer *virtuellen Kamera* die in die Szene eingefügt wird. Um eine virtuelle Kamera unter OpenGL zu definieren werden die zwei Funktionen *glFrustum(...)* und *gluLookAt(...)* verwendet. *glFrustum(...)* erzeugt aus den übergebenen Parametern eine perspektivische Projektionsmatrix [Shr09]. *gluLookAt(...)* definiert von und auf welchem Punkt in der Szene geblickt wird und wie die Kamera auf der sich daraus ergebenen Blickachse gedreht ist.

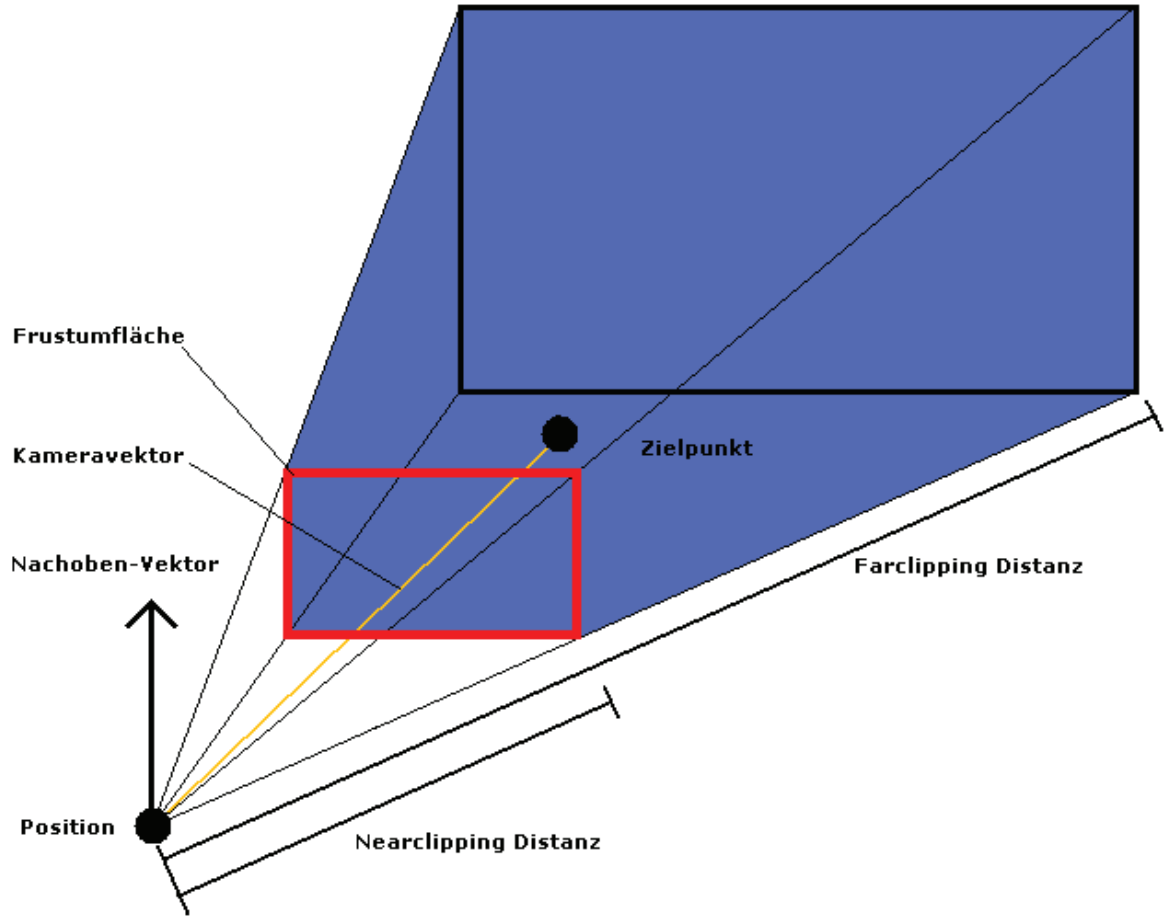

Abbildung 3.12: Skizze der Kamera

Der blau markierte Bereich in Abbildung 3.12 wird durch die Funktion *glFrustum(...)* definiert, von nun an als *sichtbarer Bereich* bezeichnet. Um den sichtbaren Bereich zu erzeugen wird zum Einen die Fläche des *Frustums* und zum Anderen die *Near-* bzw. *Farclipping Distanz* benötigt. Die Frustumfläche wird durch die Abstände ihrer Seiten vom Kameravektor beschrieben. Near- und Farclipping Distanz begrenzen, den sich aus der Frustumfläche ergebene pyramidenförmigen, sichtbaren Bereich. Um den sichtbaren Bereich in der Szene auszurichten wird die Funktion *gluLookAt(...)* verwendet. Die Ausrichtung wird durch den Punkt auf dem sich die Kamera befindet (Position), durch den Punkt auf den die Kamera ausgerichtet ist (Zielpunkt) und durch den Nachoben-Vektor, welcher relativ zur Position ist, definiert.

3.3.3 OpenGL unter Qt

Qt [Corb] stellt Elemente zur Verwendung von OpenGL zur Verfügung. OpenGL wiederrum bietet eine Softwareschnittstelle zur Garfikkartenhardware des Rechners. Unabhängig von der im System vorhandenen Hardware, kann immer gleiche Funktionalität geboten werden. Die wichtigste Klasse um OpenGL unter Qt zu verwenden ist *QGLWidget*, sie dient als abstrakte Basisklasse für ein OpenGL-Fenster. *QGLWidget* stellt drei virtuelle Methoden bereit die in der abgeleiteten Klasse implementiert werden müssen:

- *initializeGL()*, dient zur Initialisierung von OpenGL und wird nur einmal vor *resizeGL()* oder *paintGL()* aufgerufen.
- *resizeGL()*, hier wird unteranderem der View initialisiert. *resizeGL()* wird nach *initializeGL()* und vor *paintGL()* aufgerufen, nach jeder Größenänderung des Fensters ebenfalls.
- *paintGL()*, wird benutzt um die eigentliche Szene zu erstellen, das heisst hier könnten zum Beispiel Primitiven definiert werden, *paintGL()* ist der Painteventhandler des QGLWidgets. *paintGL()* muss später, zum Beispiel durch einen Timer oder Mausmoveevent, aufgerufen werden um eine Animation der Szene zu ermöglichen.

Dieses Beispiel Programm soll die Benutzung der Basisklasse verdeutlichen:

Listing 3.1: Main.cpp:

```
1  #include <QApplication>
2  #include <QMessageBox>
3  #include "mainWindow.h"
4
5  int main(int argc, char *argv[])
6  {
7          // Erstellen einer GUI Anwendung
8          QApplication app(argc, argv);
9          // Prüfen ob OpenGL vorhanden
10         if (!QGLFormat::hasOpenGL())
11         {
12                 QMessageBox::critical(0, "Kein OpenGL", "Auf diesem System gibt es kein OpenGL!");
13                 return 1;
14         }
15         // Erstellen, initialisieren und anzeigen einer TestGL-Instanz
16         TestGL myTestGL;
17         myTestGL.setWindowTitle("TestGL");
18         myTestGL.resize(300, 300);
19         myTestGL.show();
20         return app.exec();
21 }
```

Listing 3.2: mainWindow.h:

```
1  #include <QGLWidget>
2
3  class TestGL : public QGLWidget {
4  public:
5          TestGL(QWidget *parent = 0);
6  protected:
7          void initializeGL();
8          void resizeGL(int width, int height);
9          void paintGL();
10         void mousePressEvent(QMouseEvent *event);
11         void mouseMoveEvent(QMouseEvent *event);
12 private:
13         void draw();
14         GLfloat rotationX;
15         GLfloat rotationY;
```

```
16          GLfloat rotationZ;
17          GLuint texture[1];
18          QPoint lastPos;
19  };
```

Listing 3.3: mainWindow.h:

```
1   #include <QtGui>
2   #include <QtOpenGL>
3   #include "mainWindow.h"
4
5   TestGL::TestGL(QWidget *parent) : QGLWidget(parent)
6   {
7       // Aktiviert den Double- und den Tiefenbuffer für dieses Fenster
8       setFormat(QGLFormat(QGL::DoubleBuffer | QGL::DepthBuffer) );
9       // Festlegen der Rotationsfaktoren. Quad liegt in 0,0,0 -> keine Z-Rotation.
10      rotationX = 10.0;
11      rotationY = 10.0;
12      rotationZ = 0.0;
13  }
14  void TestGL::initializeGL()
15  {
16      // Sorgt dafür das ein Farbverlauf zwischen den Endpunkten der Primitiven gezeichnet wird
17      glShadeModel(GL_SMOOTH);
18      // Legt die Farbe zum leeren des Colorbuffers auf Schwarz fest, erzeugt den also Hintergrund
19      glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
20      // Legt den Wert zum leeren des Tiefenbuffers auf 1.0f fest
21      glClearDepth(1.0f);
22      // Aktiviert die Tiefenprüfung, das heisst welches Element im Vorderung ist und gezeichnet werden muss
23      glEnable(GL_DEPTH_TEST);
24      // Legt die Funktion für die Tiefenprüfung fest
25      // GL_LEQUAL sorgt dafür das alles neugezeichnet wird was einen kleineren oder gleichen Tiefenwert hat
26      glDepthFunc(GL_LEQUAL);
27      // Polygone werden nach ihrer Wicklungs angezeigt, dies bewirkt das man das Polygon nicht mehr sieht wenn
28      // man es herum dreht
29      glEnable(GL_CULL_FACE);
30      // Setzt Hinweise für die Optimierung von OpenGL
31      // GL_PERSPECTIVE_CORRECTION_HINT setzt die Interpolation von Farben und Texturen auf
32      // GL_NICEST, das heisst Interpolation mit bestmöglicher Qualität -> langsamer
33      glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
34  }
35  void TestGL::resizeGL(int width, int height)
36  {
37      // Beschreibt das Betrachtungsfenster, x/y Position der linken unteren Ecke, Höhe und Breite
38      glViewport(0,0,width,height);
39      // Legt die aktive Matrix auf die Projektionsmatrix fest
40      glMatrixMode(GL_PROJECTION);
41      // Läd die Identitätsmatrix der aktiven Matrix
42      glLoadIdentity();
43      // Erstellt eine perspektivische Projektionsmatrix
44      // der Betrachungswinkel zur y-Achse beträgt 45°
45      // das Verhältnis von Breite zu Höhe wird aus der Eigenschaften des Fensters bestimmt
46      // die Nahbereichtsgrenze wird auf 0.1f, die Fernbereichsgrenze auf 100.0f festgelegt
47      gluPerspective(45.0f, (GLfloat)width/(GLfloat)height, 0.1f, 100.0f);
48      // Legt die aktive Matrix auf die Modelview fest
49      glMatrixMode(GL_MODELVIEW);
50      glLoadIdentity();
51  }
52  void TestGL::paintGL()
53  {
54      // Leert den Color- und Tiefen-Puffer
55      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
56      draw();
57  }
58  void TestGL::draw()
59  {
60      // Leert den Color- und Tiefen-Puffer
61      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
62      glLoadIdentity();
63      // Verschiebt die komplette Matrix um -5 in Z-Richtung, dh in den Bildschirm hinein
64      glTranslatef(0.0f,0.0f,-5.0f);
65      // Legt fest um wieviel Grad, auf der X-, Y- und Z-Achse Rotiert wird
66      glRotatef(rotationX,1.0f,0.0f,0.0f);
67      glRotatef(rotationY,0.0f,1.0f,0.0f);
68      glRotatef(rotationZ,0.0f,0.0f,1.0f);
69      // Legt fest das die nächsten Vertex, zu einer GL_QUATS Primitive verbunden werden
70      glBegin(GL_QUADS);
71          // Vorderseite
72          glColor3f(1.0f,1.0f,0.0f);          // Gelb
73          glVertex3f(1.0f, -1.0f,  0.0f);     // Eckpunkt mit x=1, y=-1, z=0
74          glColor3f(0.0f,0.0f,1.0f);          // Blau
75          glVertex3f( 1.0f,  1.0f,  0.0f);    // Eckpunkt mit x=1, y=1, z=0
76          glColor3f(0.0f,1.0f,0.0f);          // Grün
77          glVertex3f( -1.0f, 1.0f,  0.0f);    // Eckpunkt mit x=-1, y=1, z=0
```

```
78          glColor3f(1.0f,0.0f,0.0f);          // Rot
79          glVertex3f(-1.0f, -1.0f,  0.0f);    // Eckpunkt mit x=-1, y=-1, z=0
80      glEnd();
81  }
82
83  // Sorgen dafür das man das Quadrad drehen kann
84  void TestGL::mousePressEvent(QMouseEvent *event)
85  {
86      lastPos = event->pos();
87  }
88  void TestGL::mouseMoveEvent(QMouseEvent *event)
89  {
90      GLfloat dx = GLfloat(event->x() - lastPos.x()) / width();
91      GLfloat dy = GLfloat(event->y() - lastPos.y()) / height();
92      if (event->buttons() & Qt::LeftButton)
93      {
94          rotationX += 180 * dy;
95          rotationY += 180 * dx;
96          updateGL();
97      }
98      else if (event->buttons() & Qt::RightButton)
99      {
100         rotationX += 180 * dy;
101         rotationZ += 180 * dx;
102         updateGL();
103     }
104     lastPos = event->pos();
105 }
```

Dieses Beispielprogramm erzeugt folgendes Fenster:

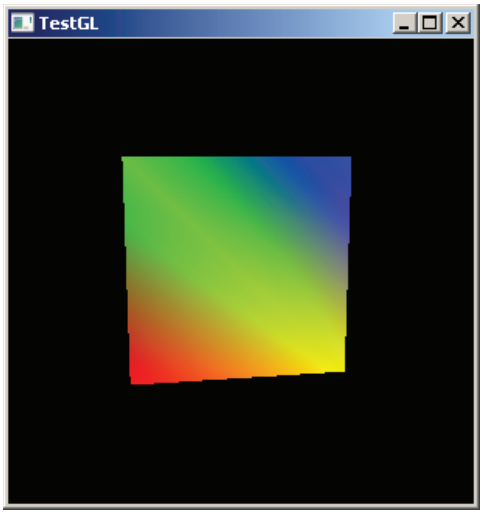

Abbildung 3.13: Fenster des OpenGL-Testprogramms

Hier kann man durch Klicken (gedrückt halten) und Ziehen mit der Maus das angezeigte Polygon drehen.

4 Entwurf

In diesem Kapitel soll aus den zuvor aufgeführten Anforderungen ein erster theoretischer Entwurf der neuen Software entstehen. Es wurde entschieden dass die Qt Bibiothek als Basis für das neue Programm dienen soll, da diese leicht auf andere Betriebssysteme portiert werden kann. Mit der Entscheidung für Qt ist auch die Wahl der Programmiersprache getroffen, hier wird C++ verwendet. Der Programmcode soll objektorientiert strukturiert werden. An geeigneten Punkten im Programm sollen Softwareentwurfsmuster [EF08] zum Einsatz kommen um den Quelltext besser zu strukturieren.
In den folgenden Abschnitten wird auf das Design der Benutzeroberfläche eingegangen. Es wird entschieden welche Techniken zur Volumengrafik im Programm verwendet werden sollen. Eine grobe Klassenstruktur des Programms soll entworfen werden, die in der Implementierung im Detail umgesetzt wird.

4.1 Benutzerinterface

Das Benutzerinterface soll eine möglichst große Fläche zur dreidimensionalen Darstellung enthalten. Auf Grund dieser Anforderung soll die Software aus zwei primären Fenstern bestehen. Dies bietet zum Einen den Vorteil, das Fenster der 3D-Ansicht wärend einer Präsentation auf einen zweiten Bildschirm (Beamer) zu schieben. Abbildung 4.1 zeigt das Prinzip der Anwendung.

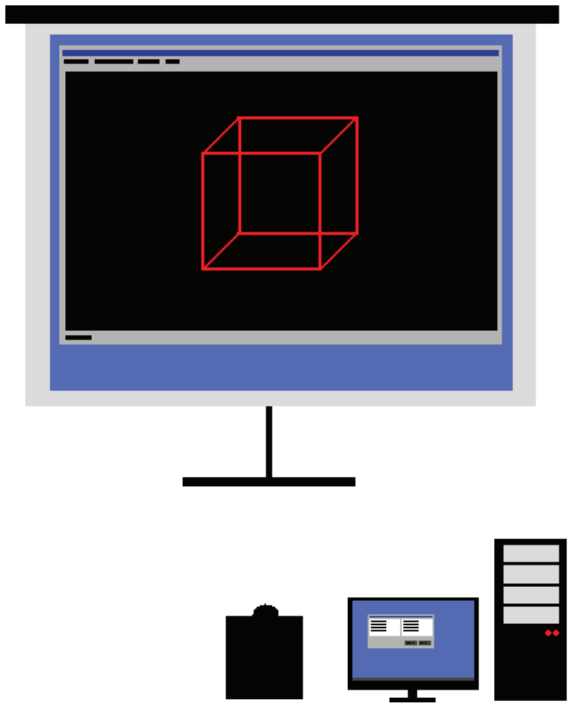

Abbildung 4.1: Das Anwendungsprinzip.

Zum Anderen könnte die Anwendung, bei einer Anpassung an ein Stereoprojektionsverfahren mit polariastions Filtern, einfach um ein zweites 3D-Fenster erweitert werden. Die Anwendung soll also aus zwei primären Fenstern bestehen, dem „Anzeigefenster“ und dem „Konfigurationfenster“.
Um die Anwendung mit Daten zu versorgen wird ein „Datei öffnen“-Dialog benötigt. Datei öffnen heisst in diesem Fall dass ein *Bildstapel*, der sich auf der Festplatte befindet, geladen wird. Die Definition eines Bildstapels ist in Abschnitt 4.1.3 zufinden. Damit allgemeine Anwendungseinstellungen, wie zum Beispiel der Pfad unter dem das Anwendungs-Logfile[1] gespeichert wird, vom Anwender verändert werden können, soll ein

[1]Im Anwendungs-Logfile werden Informationen, wie zum Beispiel Fehler im Programmablauf, gespeichert.

„Einstellungen-Dialog“ umgesetzt werden.
Für die Benutzeroberflächen werden von nun an werden folgende Namen verwendet:

- Anzeigefenster
- Konfigurationsfenster
- Bildstapel-Dialog
- Einstellungen-Dialog

In den folgenden Abschnitten werden die geplanten Funktionen der einzelnen Oberflächenteile im Detail beschrieben.

4.1.1 Anzeigefenster

Das Hauptelement des Anzeigefensters ist ein Element zur 3D-Darstellung. Für die 3D-Darstellung wird ein *OpenGL-Fenster* verwendet. Auf den Begriff OpenGL-Fenster wird in Abschnitt 4.2 eingegangen. Das OpenGL-Fenster soll über die gesamte Fläche des Hauptfensters ausgedehnt und an allen vier Seiten am Seitenrand des Fensters verankert sein. Bei einer Größenänderung, zum Beispiel dem Maximieren des Fensters, wird das OpenGL-Fenster an die Ausdehnung des Fensters angepasst. Am unteren Rand des Fensters soll sich eine Statusleiste befinden, in der Informationen wie zum Beispiel die gerenderten *FPS*[2] des OpenGL-Fensters angezeigt werden. Das Fenster soll über eine Menüleiste verfügen, die über folgende Struktur verfügt:

- Datei
  - Bildstapel öffnen
  - Beenden
- Anzeige
  - Konfigurationsfenster anzeigen
- Bearbeiten
  - Einstellungen
- Hilfe
  - Info

[2] Steht für „frames per second“, und ist somit eine Anzeige dafür mit welcher Geschindigkeit der Inhalt des OpenGL-Fenster neu gerendert wird. [Sch06]

„Bildstapel öffnen“ zeigt hier den Bildstapel-Dialog an und löst die Verarbeitung aus. „Beenden“ schließt das Programm. „Konfigurationsfenster anzeigen“ funktioniert wie eine Checkbox und soll das Konfigurationsfenster einblenden oder ausblenden. „Einstellungen“ zeigt den Einstellungen-Dialog an und löst Verarbeitung aus. „Info“ zeigt ein Informationen über das aktuelle Programm an, wie zum Beispiel die Versionsnummer oder das Datum der letzten Änderung.

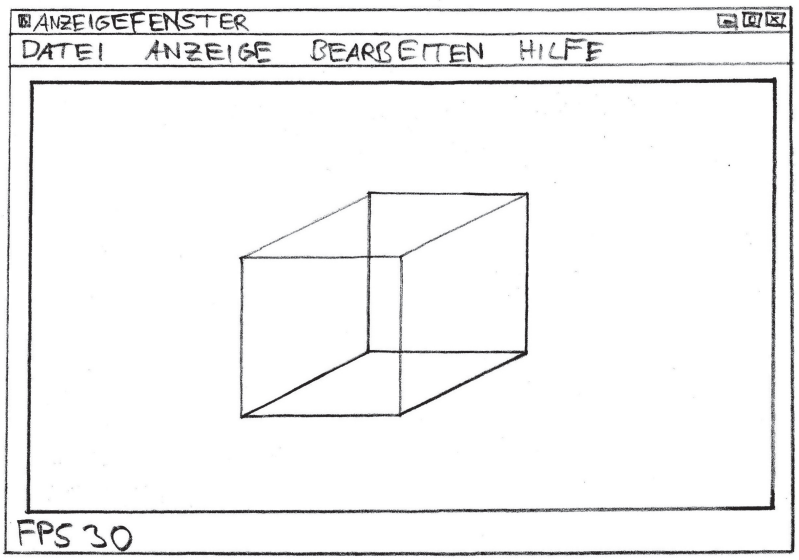

Abbildung 4.2: Skizze des Anzeigefensters.

Abblildung 4.2 zeigt eine Skizze des Fensterlayouts.

4.1.2 Konfigurationsfenster

Das Konfigurationsfenster bietet die meisten Einstellungsmöglichkeiten der Software, das heisst es verfügt auch über die aufwendigste Benutzeroberflächenstruktur. Im Konfigurationsfenster müssen sowohl die Daten der Transferfunktion 3.2.2.1, als auch Parameter für die stereoskopische Darstellung 3.1.1, also im Prizip für die Kamera 3.3.2 der 3D-Darstellung, zugreifbar gemacht werden. Es ergeben sich also zwei Gruppen von Parametern. Parameter für die Transferfunktion und Parameter für die Kameraeinstellungen. Um die Parameter von einander abzugrenzen soll ein Karteireiter-Element benutzt werden. Die Parametergruppen sollen nun getrennt betrachtet werden.

Reiter-Transferfunktion Hier müssen die Farb- und Transparenzknoten der Transferfunktion angezeigt und editierbar gemacht bzw. es müssen neue Knoten hinzugefügt und Vorhandene gelöscht werden können. Um die vorhandenen Knoten anzuzeigen sollen einfache Listenelemente verwendet werden. Die Eingabe von Werten könnte über Textboxen realisiert werden. Das Ändern eines Knotens soll durch Hinzufügen eines Knotens mit dem gleichen Grauwert, wie der des zu ändernden Knotens, erfolgen. Das Hinzufügen bzw. Ändern soll über das Klicken auf einen „Setzen-Button“ ausgelöst werden. Um einen Knoten aus einer Liste zu löschen, soll dieser in der entsprechenden Liste selektiert werden, durch ein Klick auf einen „Löschen-Button“ soll das Löschen ausgeführt werden. Die Anordnung der Steuerelemente ist in Abbildung 4.3 zu sehen.

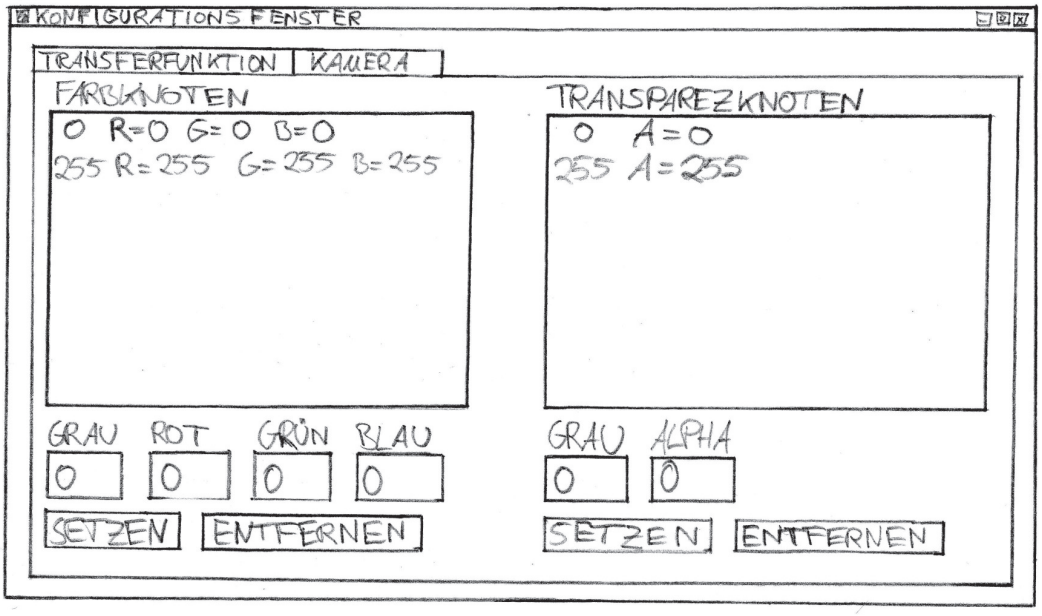

Abbildung 4.3: Skizze des Konfigurationsfensters mit Reiterstellung auf der Transferfunktion.

Reiter-Kamera Die Kameraeinstellungen lassen sich in zwei Gruppen unterscheiden, einmal die generellen Kameraeinstellungen und die Kameraeinstellungen für die stereoskopische Darstellung. Die generellen Kameraeinstellungen wirken sich ebenfalls auf die stereoskopische Darstellung aus. Generelle Kameraeinstellungen sind die Near- und Farclipping-Distanz und das Blickfeld. Die Stereoeinstellungen müssen zu allererst eine Möglichkeit bieten, die Stereoprojektion ein- und auszuschalten. Das Projekttionsverfahren sollte wählbar sein, auch wenn zunächst nur das Farbanaglyphe-Verfahren implementiert werden soll. Die zwei ausschlaggebenden Parameter für eine Stereoprojektion, die Stereobasis und die Fernpunktweite, sollen durch die Benutzeroberfläche zugreifbar gemacht werden. Begriffserklärungen zur Stereoskopie sind in Kapitel 3.1.2 zu finden. Begriffsdefinitionen zur Kamera in Abschnitt 3.3.2 zu finden. Als Steuerelemente zur Eingabe sollen Check-, Text- und Comboboxen verwendet werden. Die Anordnung der Steuerelemente ist in Abbildung 4.4 zu sehen.

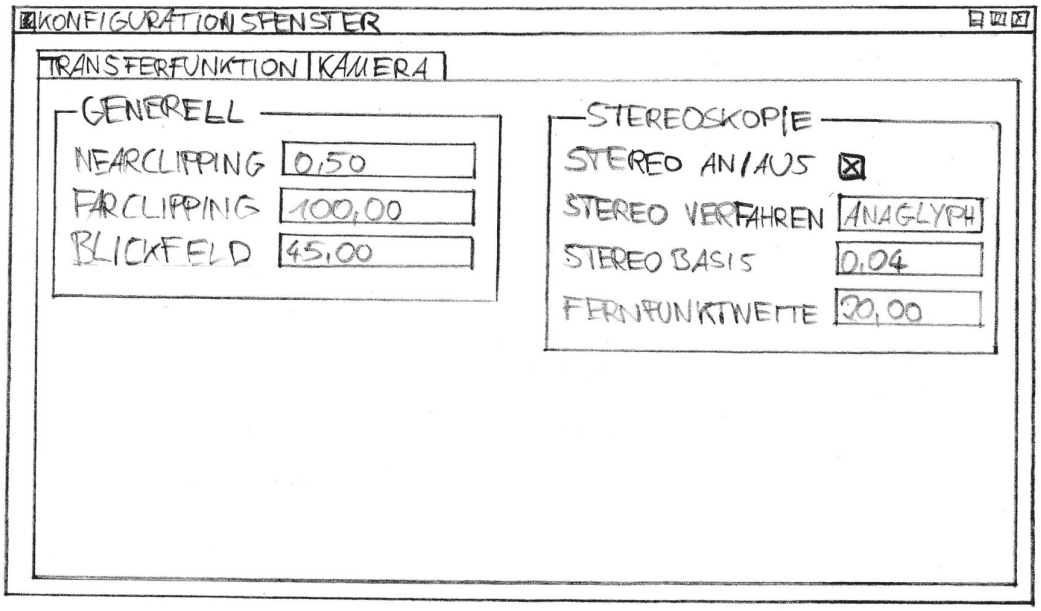

Abbildung 4.4: Skizze des Konfigurationsfensters mit Reiterstellung auf den Kameraeinstellungen.

4.1.3 Bildstapel-Dialog

Mit diesem Dialog sollen Informationen über den zu ladenden Bildstapel erfasst werden. Um festzustellen welche Informationen benötigt werden, ist es notwendig festzustellen wodurch definiert wird.

Bildstapel Ein Bildstapel besteht aus einer bestimmten Anzahl von Bilddateien die im Dateisystem abgelegt sind, wie in Tabelle 4.1 beispielhaft aufgelistet.

C:/Daten/Bildstapel/Tetraeder/Tetraeder-8-bit000.tif
C:/Daten/Bildstapel/Tetraeder/Tetraeder-8-bit001.tif
C:/Daten/Bildstapel/Tetraeder/Tetraeder-8-bit002.tif
...
C:/Daten/Bildstapel/Tetraeder/Tetraeder-8-bit509.tif
C:/Daten/Bildstapel/Tetraeder/Tetraeder-8-bit510.tif
C:/Daten/Bildstapel/Tetraeder/Tetraeder-8-bit511.tif

Tabelle 4.1: Dateiliste eines beispiel Bilderstapels.

Die Dateinamen des Bildstapels bestehen aus einem konstanten Teil, hier „Tetraeder-8-bit“, und einem fortlaufenden Index, im Beispiel von 0 bis 511. Die Dateien sind alle im gleichen Ordner des Dateisystems abgelegt, hier „C:/Daten/Bildstapel/Tetraeder/“. Alle Bilddateien haben die gleich Dateiendung, im Beispiel „.tif“. Zusätzlich wird noch die Breite und Höhe des Bildes in Pixel benötigt. Zudem wäre es auch möglich den Index nicht mit Nullen aufzufüllen, also anstelle von „001“ nur eine „1“ zu schreiben. Der Index könnte auch mit führenden Nullen definiert werden, zum Beispiel statt „511“ eine „0511“, aus diesem Grund sollte die Länge des Indexes abgefragt werden. Die Abfrage der Länge ist natürlich nur notwendig, wenn der Index mit Nullen aufgefüllt werden soll.

Die erforderlichen Paramenter, die der Dialog liefern muss, sind also:
Dateiname, Verzeichnis, Dateiendung, Bildbreite, Bildhöhe, Startindex, Endindex, Index mit Nullen auffüllen, Indexlänge.

Um diese Werte im Dialog abzufragen können Standard-Textboxen verwendet werden. Ein mögliches Design der Benutzeroberfläche des Bildstapel-Dialogs ist in Abbildung 4.5 zu sehen.

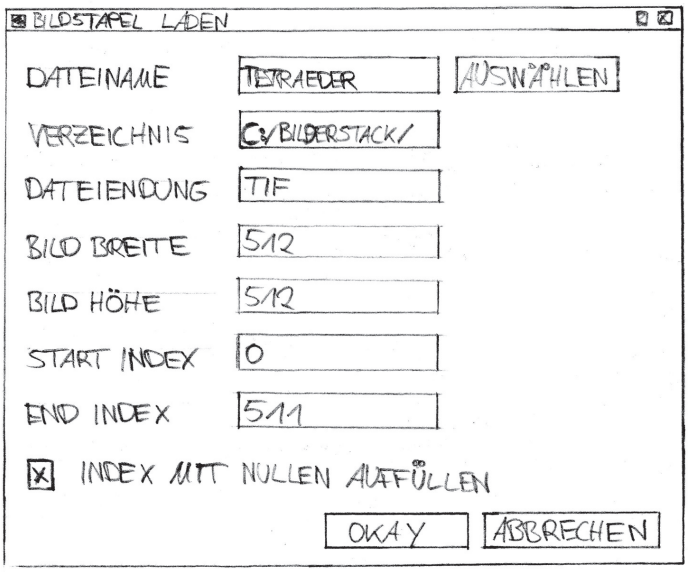

Abbildung 4.5: Skizze der Benutzeroberfläche des Bildstapel-Dialogs.

4.1.4 Einstellungen-Dialog

Im Einstellungen-Dialog sollen Einstellungen die die gesamte Anwendung betreffen änderbar sein. Zur Zeit ist erst eine Verwendung absehbar, und zwar der Pfad unter dem das *Anwendungslogfile* gespeichert wird. Der Speicherpfad kann über eine Textbox angezeigt bzw. eingegeben werden. Ein „Auswahl-Button“ zum Durchsuchen des Dateisystems soll angeboten werden. Der Dialog soll aber um weitere Einstellungen erweiterbar sein. Die Benutzeroberfläche des Dialogs könnte dann wie in Abbildung 4.5 aussehen.

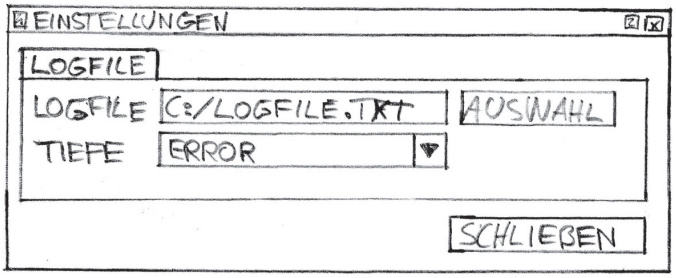

Abbildung 4.6: Skizze der Benutzeroberfläche des Einstellungen-Dialogs.

4.2 Klassenstruktur

In Abbildung 4.7 wird die Klassenstruktur der Anwendung dargestellt. Die Klassenstruktur zeigt nur Oberklassen ohne eventuelle Schnittstellen und Basisklassen.

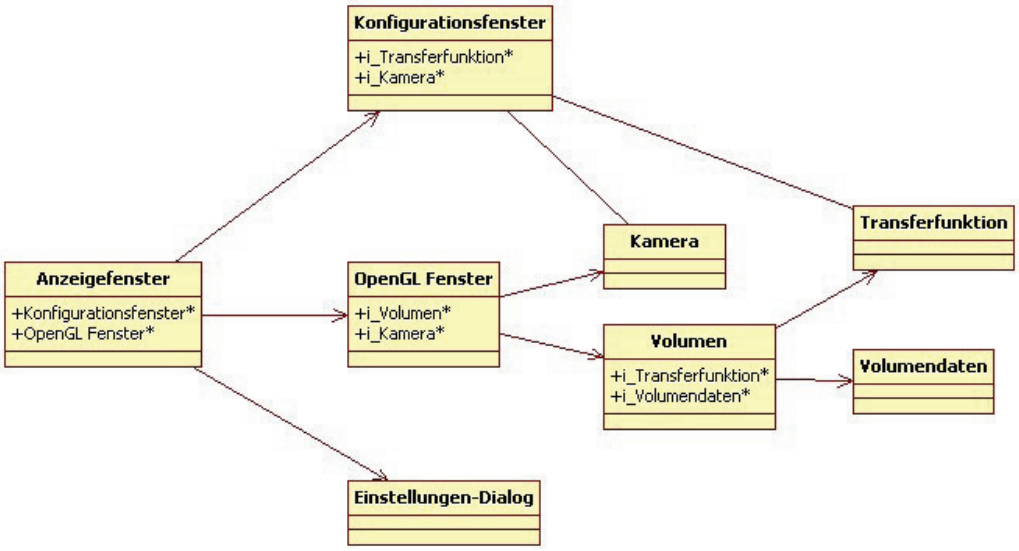

Abbildung 4.7: Die geplante Klassenstruktur der Anwendung.

Die Benutzeroberflächen für Anzeigefenster und Konfigurationsfenster wurden bereits im Abschnitt 4.1 erläutert, im folgenden soll die Klassenstruktur, die sich hinter der Benutzeroberfläche verbirgt, beschrieben werden.

OpenGL-Fenster Das *OpenGL-Fenster* initialisiert OpenGL und sorgt dafür dass die Volumendarstellung aktualisiert wird, wenn der Benutzer die Kameraeinstellung, die Transferfunktion oder die Volumendaten verändert. Die Klasse OpenGL-Fenster enthält einen Verweis auf ein Volumen und die Kamera. Benutzereingaben, im OpenGL-Fenster, sollen an die Kamera-Klasse zur weiteren Verarbeitung übergeben werden.
Das OpenGL-Fenster unterscheidet zwischen normaler Kameraführung und Stereoskoperdarstellung.

Kamera In der *Kamera*-Klasse soll die gesamte Kameralogik gekapselt werden. Die Kamera wird durch ihre Position, Ziel und den Obenvektor definiert. Methoden um Benutzereingaben, zum Verändern des Kamerablickfelds, zu verarbeiten sollen bereitgestellt werden.

Volumen Ein *Volumen* wird durch die Volumendaten und eine zugehörige Transferfunktion charakerisiert, daraus resultiert die Klasse *Volumen* welche über Verweise auf die Volumendaten und eine Transferfunktion zugreift. Die *Volumen-Klasse* definiert wie die untergeordneten Daten gezeichnet werden, sie repräsentiert das Verfahren zur Volumengrafikdarstellung 4.3.

Volumendaten Die Klasse *Volumendaten* enthält die aus einem Bildstapel geladenen Daten und sorgt für die Verwaltung der Volumendaten auf der Grafikhardware. Die Verwaltung umfasst das Initialisieren, das Füllen mit Daten und das Freigeben der notwendigen 3D-Textur.

Transferfunktion Aufgabe der *Transferfunktion*-Klasse ist es Farb- und Transparenzknoten zu verwalten, die Interpolation zwischen den Knoten durchzuführen und die 1D-Textur der Transferfunktion auf die Grafikkarte zu laden.

4.3 Auswahl der Volumengrafikmethoden

Der grundlegende Gedanke hinter der Auswahl der Methoden ist die Frage auf welche Hardware der Großteil der Rechenarbeit geleistet werden soll. Da es sich um eine grafische Anwendung handelt und heute in den meisten Rechnern leistungsfähige Grafikprozessoren vorhanden sind, ist es sinnvoll eine Grafikprozessor orientierte Varianten zu wählen. Die Umsetzung der Volumengrafikverfahren auf Grafikhardware entspricht dem Stand der Technik [Had06]. Auf der Grafikkarte wird die Datenhaltung von *Texturen* [Shr09] übernommen. Volumendaten sollen in einer *3D-Textur*, die Transferfunktion in einer *1D-Textur* abgelegt werden. Die Verarbeitung der Daten wird durch *Vertex-* und *Fragmentshader* auf der Grafikkarte durchgeführt. Auf den Begriff Shader wird in Abschnitt 5.3.2 noch eingegangen. Verarbeitung bedeutet hier die Schritte Interpolation, Klassifizierung, Shading und Composing der Volumengrafik-Darstellung. Als Geometrie auf der die 3D-Textur dargestellt wird, soll das Slicing-Verfahren verwendet werden.

4.4 Auswahl des Farbanaglypheprojektions-Verfahren

Hier soll kurz beleuchtet werden warum in diesem Projekt das Farbanaglyphe-Verfahren verwendet werden soll. Der große Vorteil des Farb-Anaglyphensystems gegenüber anderen Verfahren ist, dass es schnell und mit geringem Aufwand zur Präsentation verwendet werden kann. Es ist kein Problem, ein vorbereitetes Anaglyphenbild auf einem Monitor oder über einen Beamer zu zeigen. Der Aufwand für die Erzeugung des Bildes wurde also schon im Vorfeld erbracht. Diese Vorarbeit soll in unserem Fall softwaretechnisch, durch die geplante Programmerweiterung, durchgeführt werden, das heisst die Bilder sollen in nahezu Echtzeit Farb-Anaglyph erzeugt und ausgegeben werden.

Ein weiterer Vorteil des Farb-Anaglyphenverfahrens ist sein Preis. Es wird zum Beispiel kein zweiter Projektor benötigt wie beim Polarisationsverfahren, sondern nur die Farbfilterbrille. Eine Rot/Cyan-Fitlerbrille aus Karton ist im Mittel für 1,10 € erhältlich, im Vergleich dazu kostet eine Zirkularpolarisations-Filterbrille aus Karton im Mittel 2,49 € und damit mehr als das Doppelte.[3]

4.5 Prozesse

4.5.1 Laden von Volumendaten

Um Daten von einem Datenträger in das Programm zu laden, wird über die Menüleiste des Anzeigefensters der Bildstapel-Dialog ausgeführt. Um eine 3D-Textur in den Speicher der Grafikkarte zu laden, kann die OpenGL-Funktion *glTexImage3D(..)* verwendet werden, welche ein eindimensionales Array mit den Texturdaten erwartet. Mit Hilfe der im Dialog erfassten Basisdaten soll der Bilderstapel zunächst in ein eindimensionales Array geladen werden. Das Programm muss jedes Bild des Bildstapel durchlaufen und für jeden Pixel den jeweiligen Grauwert in das Array speichern, von nun an als *Voxel-Array* bezeichnet, welches *glTexImage3D(..)* übergeben werden kann. Der Datentyp für das Voxel-Array soll *unsigend char* sein. Ein Voxel-Array enthält für jeden Voxel des Volumens einen Grauwert von 0 bis 255.

[3]Herkunft der Preise im Anhang unter A.1

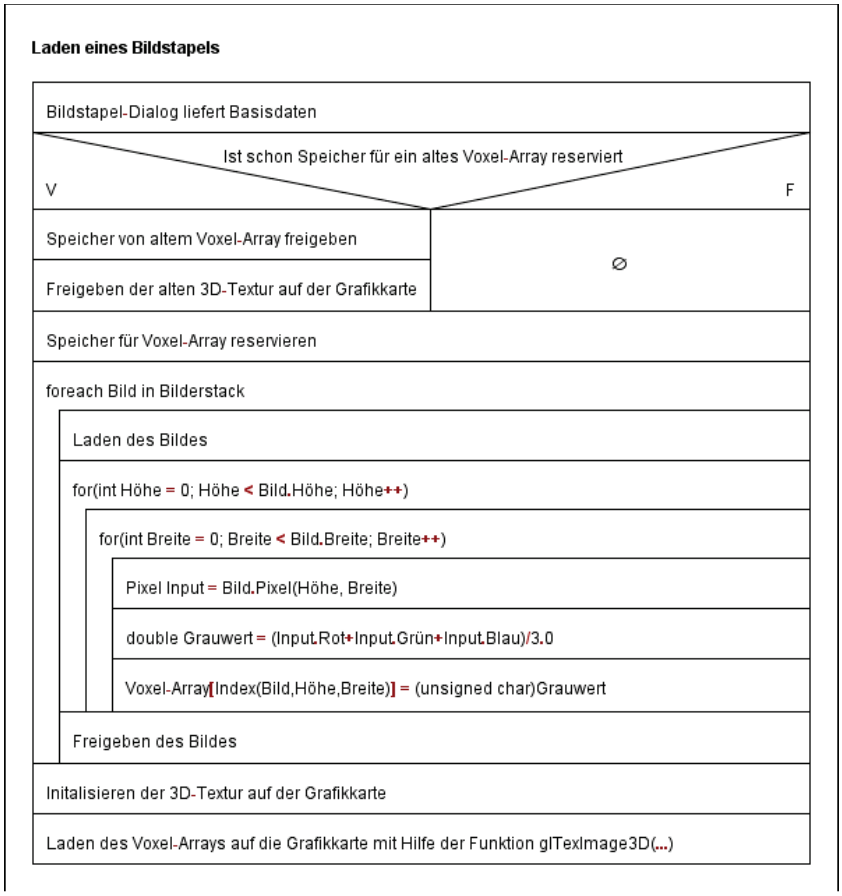

Abbildung 4.8: Programmablauf zum Laden eines Bildstapels.

Nach dem Laden des Voxel-Arrays in den Arbeitsspeicher wird die Textur auf der Grafikkarte inistialisiert. Im Anschluss an die Initialisierung soll das Voxel-Array mit *glTexImage3D(...)* in den Grafikspeicher übertragen werden.
Am Ende dieses Prozesses liegen die Volumendaten in Form einer 3D-Textur im Speicher der Grafikkarte und kann dort weiter verarbeitet werden.

4.5.2 Laden einer Transferfunktion

Am Anfang des Prozesses stehen die Farb- und Transparenzknoten der Transferfunktion 3.2.2.1. Um eine 1D-Textur in den Speicher der Grafikkarte zu laden, kann die OpenGL-Funktion *glTexImage1D(...)* verwendet werden.

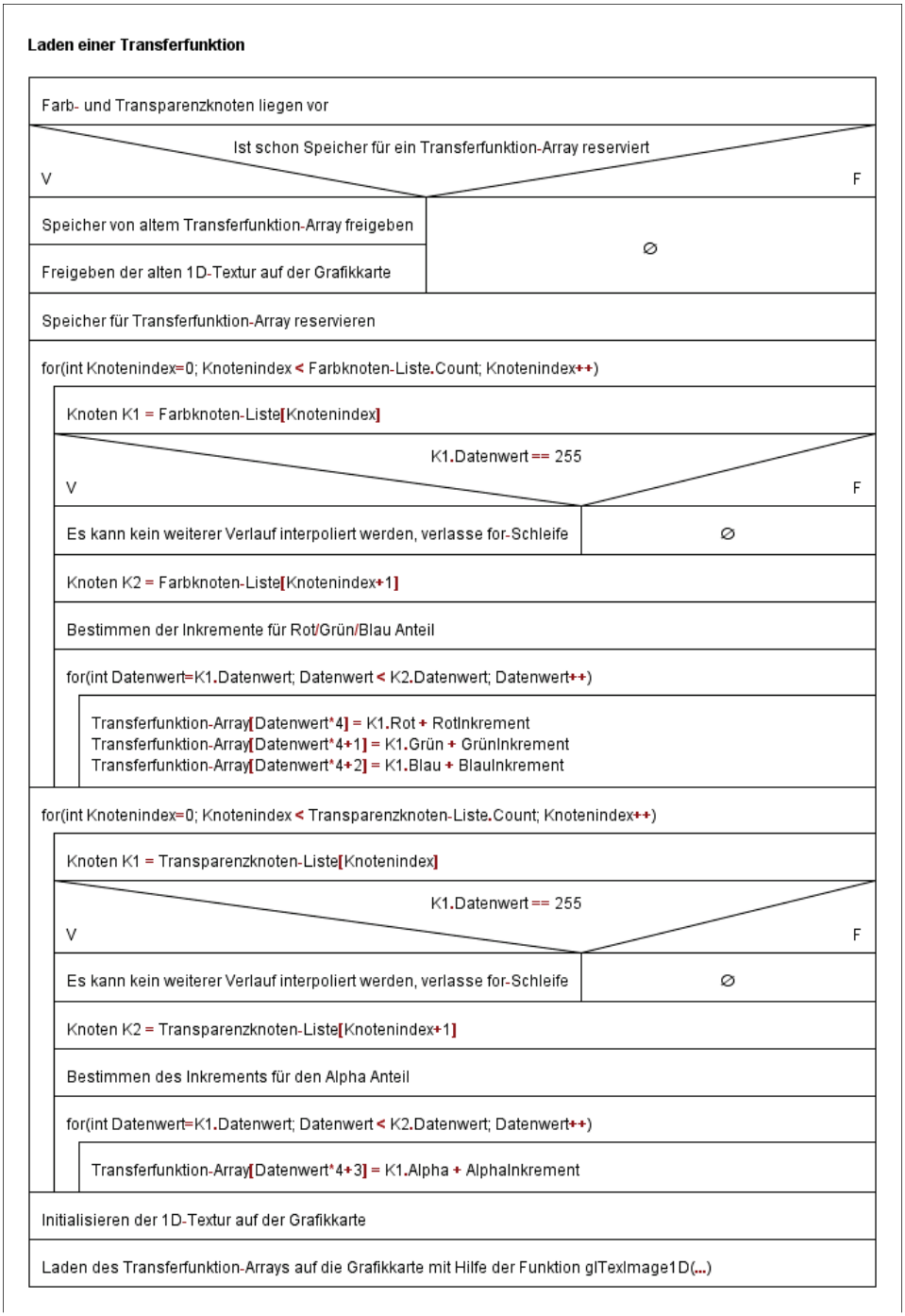

Abbildung 4.9: Programmablauf zum Laden einer Transferfunktion.

Die Funktion erwartet in ihrem letzten Parameter ein Array, das die Textur-Daten enthält. Es soll ein eindimensionales Array erzeugt werden in dem die aus den Konten interpolierte Transferfunktion gespeichert ist. Von nun an wird dieses Array als

Transferfunktion-Array bezeichnet. Die Werte für Rot, Grün, Blau und Alpha der Transferfunktion sollen einen Wertebereich von 0 bis 255 abdecken. Der Datentyp *unsigned char* deckt diesen Bereich optimal ab. Das Transferfunktion-Array muss für jeden Datenwert, den ein Voxel im Volumen annehmen kann, eine Referenzfarbe (RGB) mit Alphawert (A) enthalten. Das Transferfunktion-Array ist $4 * 256 = 1024$ Werte vom Typ *unsigned char* groß.

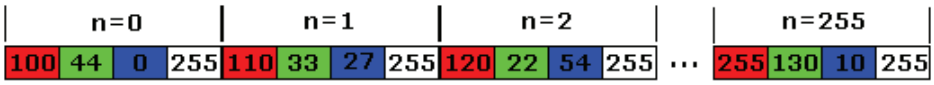

Abbildung 4.10: Datenformat des Transferfunktion-Arrays.

In Abbildung 4.10 ist das Datenformat des Transferfunktion-Arrays dargestellt. Der Index n zählt hierbei die Anzahl Pixel der späteren Textur. Liegt das Array in der dargestellten Form vor, wird die Textur auf der Grafikkarte initialisiert. Die Texturdaten werden anschließend mit *glTexImage1D* aus dem Transferfunktion-Array in den Speicher der Grafikkarte geladen.
Am Ende dieses Prozesses liegen die Transferfunktion in Form einer 1D-Textur im Speicher der Grafikkarte und kann dort weiter verarbeitet werden.

4.5.3 Ein- und Ausschalten der Stereoprojektion

Um eine Stereoprojektion umzusetzen, muss die gleiche Szene von unterschiedlichen Betrachtungspunkten dargestellt werden. In der Anwendung bedeutet das, dass das Volumen zweimal mit verschiedenen Kameraeinstellungen gerendert werden muss. An einer geeigneten Stelle muss also zwischen zwei Kameraeinstellungen unterschieden werden. Eine geeignete Stelle ist die *paintGL()*-Methode des OpenGL-Fensters.

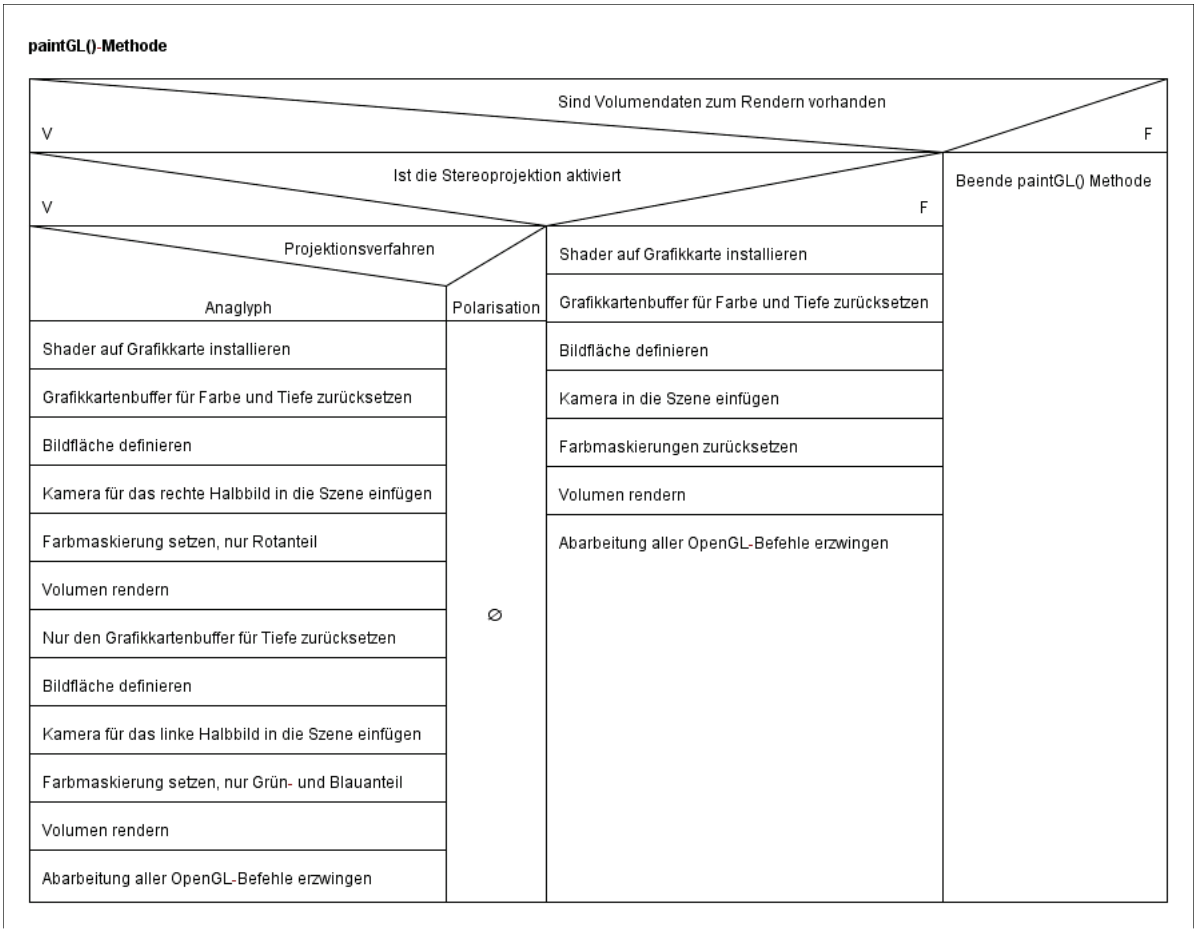

Abbildung 4.11: Ablauf der paintGL()-Methode zur Unterscheidung zwischen Normal- und Stereoprojektion.

Um die Einstellungen für die beiden Stereohalbbilder zu erhalten, soll eine Stereokamera erzeugt werden die auf der normalen Kamera basiert. Das Akkumulieren beider Halbbilder erfolgt automatisch dadurch, dass der *Colorbuffer* [Shr09] zwischen den zwei Render Operationen, nicht zurückgesetzt wird.

5 Implementierung

In diesem Kapitel wird die aus dem Entwurf entstandene Programmimplementierung dokumentiert. Im ersten Abschnitt werden die Benutzeroberflächen mit ihren zugehörigen Klassen erläutert. Anschließend werden die Klassen, die das System im Hintergrund repräsentieren, beschrieben. Auf spezielle Problemstellungen, die bei der Implementierung aufgetreten sind, wird im letzten Abschnitt eingegangen.

5.1 Benutzeroberflächen

Im Abschnitt Benutzeroberfläche werden die relevanten Oberflächenelemente erläutert. Die Schnittstellen zu den Klassen die im Hintergrund arbeiten soll beschrieben werden. Um die Oberflächenelemente auf einem Formular anzuordnen werden von Qt sogenannte *Layoutklassen* bereitgestellt. Die Layoutklassen, sowie die darin anzuordnenden Elemente, können mit Hilfe des *Qt-Designers* ohne großen Aufwand kombiniert werden. Auf Grund der einfachen Handhabung wurden die Benutzeroberflächen mit dem Qt-Designer erstellt und von Hand um benutzerdefinierte Elemente, wie das *OpenGlWidget*, erweitert.

5.1.1 Anzeigefenster

In Abbildung 5.1 ist die Klassenstruktur des Anzeigefensters dargestellt, welche im folgenden erläutert wird.

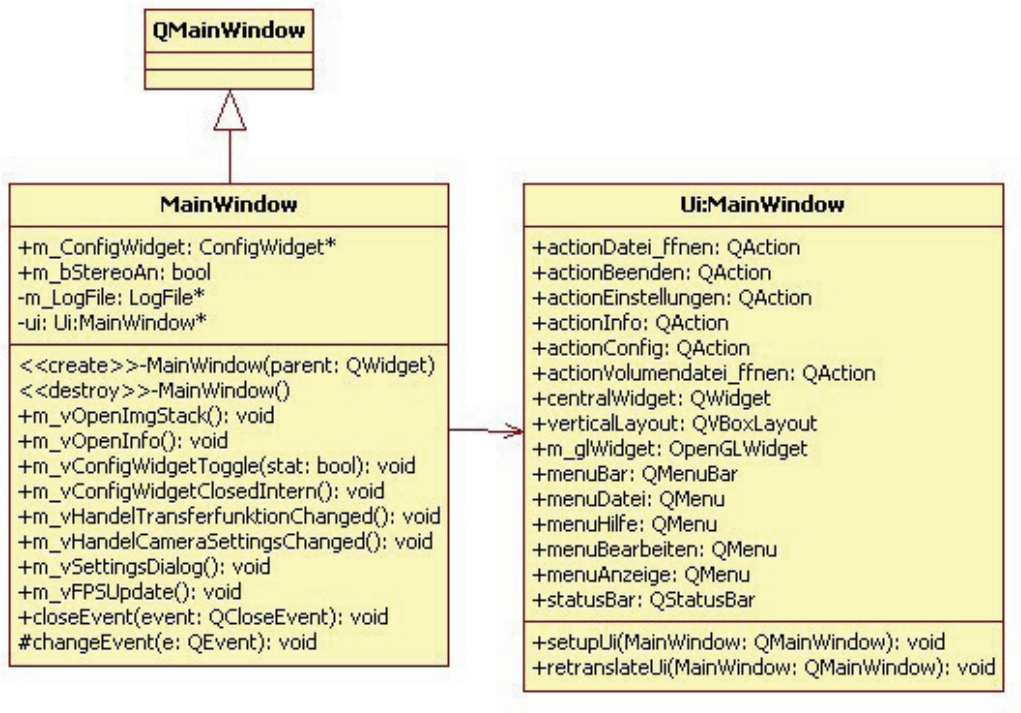

Abbildung 5.1: Klassendiagramm der Anzeigefenster-Klasse.

Das primäre Element des Anzeigefensters ist das 3D-Fenster, welches vom Typ *OpenGL-Widget* ist. Basisklasse des Anzeigefensters ist das *QMainWindow*. Zum erzeugen einer Menüleiste stellt Qt die Klasse *QMenuBar* zur Verfügung, welcher man Objekte von Typ *QMenu* zuordnen kann um die einzelnen Menüs, wie den Reiter Datei, zu erzeugen. Die Statusbar ist vom Typ *QStatusBar*. In Abbildung 5.2 ist das Layout des Anzeigefensters mit allen Steuerelementen zu sehen.

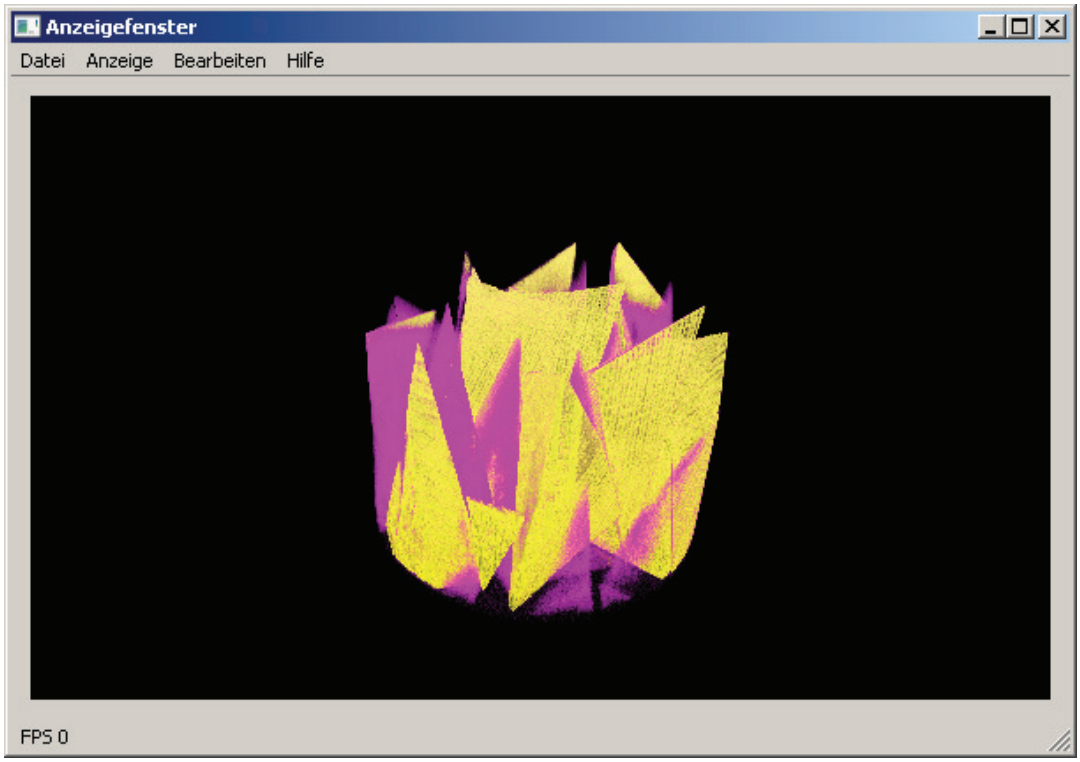

Abbildung 5.2: Screenshot des Anzeigefensters.

Signale[1] die der Benutzer im Anzeigefenster auslöst, werden an die verarbeitenden Klassen weitergegeben. Die Signale der Menüleiste werden direkt vom Anzeigefenster abgearbeitet. Für die Abarbeitung sind *Slots*[2] definiert, dies sind Methoden die bei dem Auftreten eines Signals ausgeführt werden. In Tabelle 5.1 sind die vom Anzeigefenster implementierten Slots mit ihren Funktionsbeschreibungen aufgeführt.

[1]Signale sind die Qt Variante von Events. Es wird ein Variante des Observer-Musters [EF08] verwendet, die Signale sind hierbei die Subjekte des Muster.
[2]Slots sind die Beobachter im Observer-Muster des Signal/Slot-Systems.

Slotname	**Signalherkunft**	**Beschreibung**
m_vOpenImgStack()	Menüleiste	Zeigt den Bildstapel-Dialog an. Wenn der Dialog erfolgreich abgeschlossen wurde wird das im Dialog erzeugt Datenfile an das Volumen weitergegben, anschließend wird die Initialisierungsmethode des Volumens ausgeführt.
m_vOpenInfo()	Menüleiste	Zeigt den Informations-Dialog an.
m_vConfigWidgetToggle(bool stat)	Menüleiste	Sorgt dafür das die Anzeige des Konfigurationsfensters entweder Ein- oder Ausgeschaltet wird, das heisst ist es angezeigt wird es ausgeblendet und umgekehrt.
m_vConfigWidgetClosedIntern()	Konfigurationsfenster	Wird das Konfigurationsfenster intern geschlossen, wird hier die Anzeige des „Akiviert“-Hakens im Menü entfernt.
m_vHandelTransferfunktionChanged()	Konfigurationsfenster	Führt eine Aktualisierung der Transferfunktion aus und sorgt dafür das das *OpenGLWidget* neu gezeichnet wird.
m_vHandelCameraSettingsChanged()	Konfigurationsfenster	Löst eine neu Berechnung der Kameraeinstellung aus und sorgt dafür das das *OpenGLWidget* neu gezeichnet wird.
m_vSettingsDialog()	Menüleiste	Ruft den Settingsdialog auf.
m_vFPSUpdate()	OpenGLWidget	Aktuallisiert die FPS-Anzeige in der Statusleiste.
closeEvent(QCloseEvent *event)	Selbst	Schließt das Konfigurationsfenster, wenn das Anzeigefenster geschlossen wird.

Tabelle 5.1: Slots des Anzeigefensters

Slot werden mit der Methode *connect(...)* Klasse *QObject* einem Signal zugeordnet, in Codelisting 5.1 wird dies gezeigt. Bei der Zuordnung ist Voraussetzung das die Funktionsköpfe identische Parametertypen aufweisen.

Listing 5.1: Beispiel für QObject::connect(...):

```
1 QObject::connect(ui->actionDatei_ffnen, SIGNAL(triggered()), this, SLOT(m_vOpenImgStack()));
```

5.1.2 Konfigurationsfenster

In Abbildung 5.3 ist die Klassenstruktur des Konfigurationsfenster dargestellt, welche im folgenden erläutert wird.

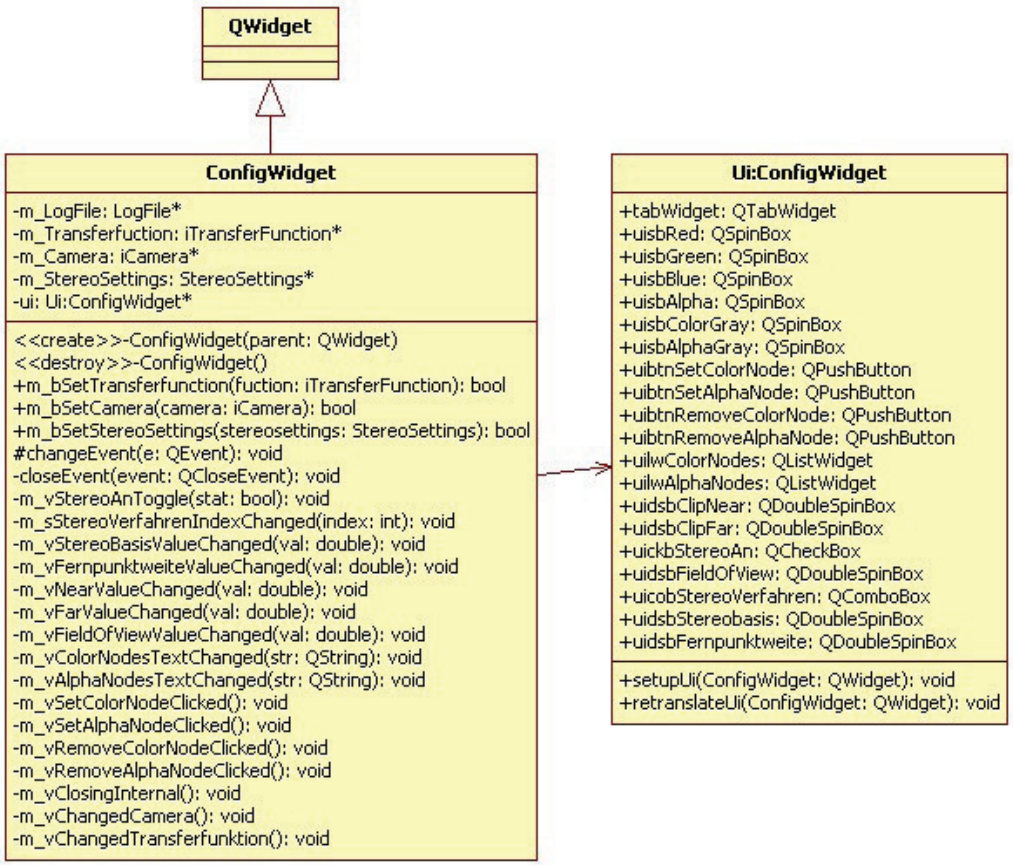

Abbildung 5.3: Klassendiagramm der Konfigurationsfenster-Klasse. Im UI-Teil des Klassendiagramms wurden Label, die der Beschriftung der Steuerelemente dienen, und Layoutelemente entfernt.

Als Basisklasse für das Konfigurationsfenster wurde der allgemeine Fenstertyp, *QWidget*, verwendet. Das Karteikarten-System wird durch ein *QTabWidget* Objekt erzeugt. Die Funktionalitiäten der Reiter Transferfunktion und Kamera werden in den folgenden zwei Abschnitten getrennt betrachtet.
In Tabelle 5.2 sind die vom Konfigurationsfenster ausgelösten Signals mit ihren Auslösebedingungen aufgeführt.

Signalname	Empfangen von	Bedingung
m_vChangedCamera()	Anzeigefenster	Wird ausgelöst wenn eine Kamera-Einstellung geändert wurde.
m_vChangedTransferfunktion()	Anzeigefenster	Wird ausgelöst wenn ein Farb- oder Transparenzknoten der Transferfunktion verändert, gelöscht oder hinzugefügt wurde.

Tabelle 5.2: Signals des Konfigurationsfensters

Transferfunktion Auf dem Reiter Transferfunktion sollen die Farb- und Transparenzknoten der Transferfunktion verändert, gelöscht und neue hinzugefügt werden können. In Abbildung 5.4 ist das Layout des Transferfunktionreiters mit allen Steuerelementen zu sehen.

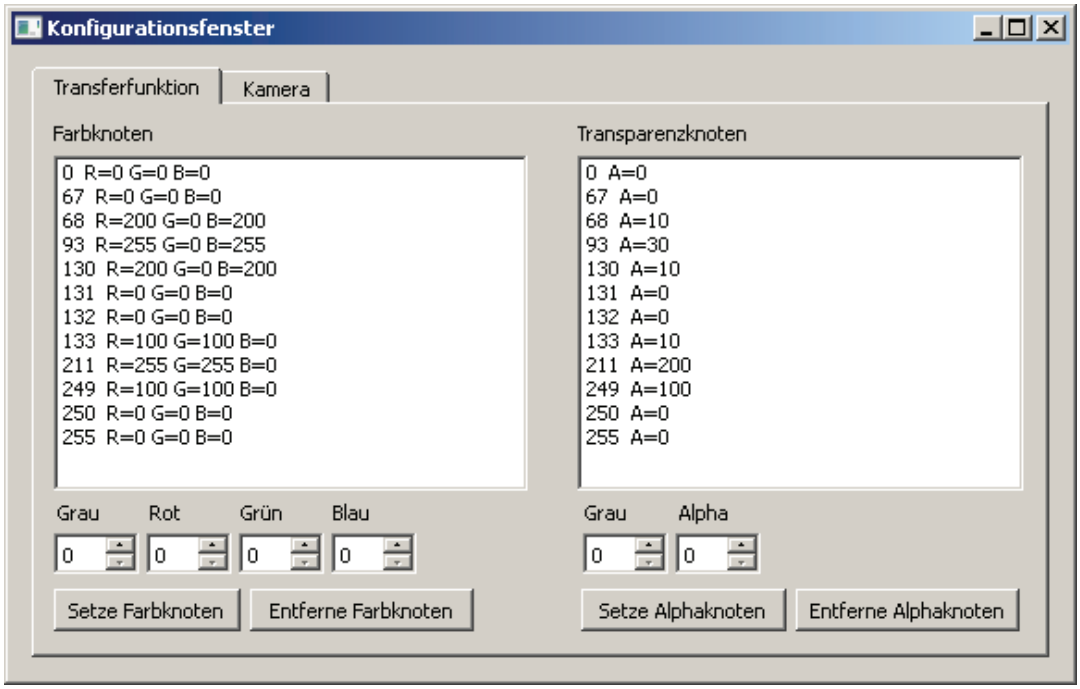

Abbildung 5.4: Screenshot des Konfigurationsfenster(Reiter: Transferfunktion).

Für die Anzeige der vorhandenen Knoten werden *QListWidget* Elemente verwendet. Die Listen stellen im Grunde ein Array von Strings dar, welche mit der Maus selektiert werden können. Die aufgelisteten Strings haben folgende Formate:
<Grauwert> R=<Rotanteil> G=<Grünanteil> B=<Blauanteil>
<Grauwert> A=<Alpha>
Zur Eingabe neuer Knoten werden Elemente vom Typ *QSpinBox* verwendet. *QSpinBox* arbeiten mit einen Integerwert. Der Wertebereich wurde auf 0 bis 255 begrenzt, sowohl für Grauwert als auch für Farb- und Transparenzwert. Das Ändern eines bestehenden Knotens erfolgt durch Klicken auf den gewünschten Knoten in der jeweiligen

Liste. Anschließend werden die Werte in die unter der Liste befindlichen Spinboxen geschrieben. Die Werte können nun verändert werden und durch einen Klick auf den zur Liste gehörenden „Setzen-Button“ werden die neuen Werte in die Liste übernommen. Das Löschen eines Knoten erfolgt analog zum Ändern, ein Klick auf den Knoten in der Liste selektiert diese der anschließende Klick auf den „Löschen-Button“ löscht diesen aus der Liste. Die Knoten mit den Grauwerten 0 und 255 können nur editiert, jedoch nicht gelöscht werden. Um einen neuen Knoten zur Liste hinzuzufügen werden die Werte in die entsprechenden Spinboxen eingetragen und mit dem „Hinzufügen-Button“ in die Liste hinzugefügt.
Auf die Instanz der Transferfunktion wird über ein referenziertes Feld vom Typ *iTransferFunction** zugriffen. Das Objekt welches referenziert wird, wird von Volumen erzeugt. Die Referenz wird beim Programmstart vom Anzeigefenster mit der Methode *m_bSetTransfercunction(...)* gesetzt.
In Tabelle 5.3 sind die vom Transferfunktionsreiter implementierten Slots mit ihren Funktionsbeschreibungen aufgeführt.

Slotname	**Signalherkunft**	**Beschreibung**
m_vColorNodesTextChanged(QString str)	Farbknoten-Listbox	Schreibt den Grau-, Rot-, Grün- und Blauwert des, in der Farbknoten-Listbox ausgewählten, Farbknotens in die zugehörigen Spinboxen.
m_vAlphaNodesTextChanged(QString str)	Transparenzknoten-Listbox	Schreibt den Grau- und Alphawert des, in der Transparenzknoten-Listbox ausgewählten, Transparenzknotens in die zugehörigen Spinboxen.
m_vSetColorNodeClicked()	Setze Farbknoten-Button	Erstellt einen Farbknoten und fügt diesen der Farbknotenliste der Transferfunktion hinzu. Um einen Farbknoten zu erstellen wird die Methode *m_bColorNodeAdd(...)* des *iTransferfunction*-Interfaces verwendet. Anschließend wird die Transferfunktion erneut in die Listenfelder geladen. Das Signal *m_vChangedTransferfunktion()* wird ausgelöst.
m_vSetAlphaNodeClicked()	Setze Alphaknoten-Button	Erstellt einen Transparenzknoten und fügt diesen der Transparenzknotenliste der Transferfunktion hinzu. Um einen Alphaknoten zu erstellen wird die Methode *m_bAlphaNodeAdd(...)* des *iTransferfunction*-Interfaces verwendet. Anschließend wird die Transferfunktion erneut in die Listenfelder geladen. Das Signal *m_vChangedTransferfunktion()* wird ausgelöst.
m_vRemoveColorNodeClicked()	Entferne Farbknoten-Button	Entfernt den Farbknoten aus der Farbknotenliste der Transferfunktion, welcher den Grauwert aufweist der in der entsprechenden Grauwert-Spinbox ausgewählt wurde. Um einen Farbknoten zu löschen wird die Methode *m_bColorNodeRemove(...)* des *iTransferfunction*-Interfaces verwendet. Anschließend wird die Transferfunktion erneut in die Listenfelder geladen. Das Signal *m_vChangedTransferfunktion()* wird ausgelöst.
m_vRemoveAlphaNodeClicked()	Entferne Alphaknoten-Button	Entfernt den Transparenzknoten aus der Transparenzknotenliste der Transferfunktion, welcher den Grauwert aufweist der in der entsprechenden Grauwert-Spinbox ausgewählt wurde. Um einen Alphaknoten zu löschen wird die Methode *m_bAlphaNodeRemove(...)* des *iTransferfunction*-Interfaces verwendet. Anschließend wird die Transferfunktion erneut in die Listenfelder geladen. Das Signal *m_vChangedTransferfunktion()* wird ausgelöst.

Tabelle 5.3: Slots des Transferfunktionsreiters

Kamera Auf dem Reiter Kamera sollen zum Einen die generellen Kameraeinstellungen konfiguriert werden können, zum Andern soll auch die stereoskopische Projektion aktiviert und konfigurierbar gemacht werden. In Abbildung 5.5 ist das Layout des Kamerareiters mit allen Steuerelementen zu sehen.

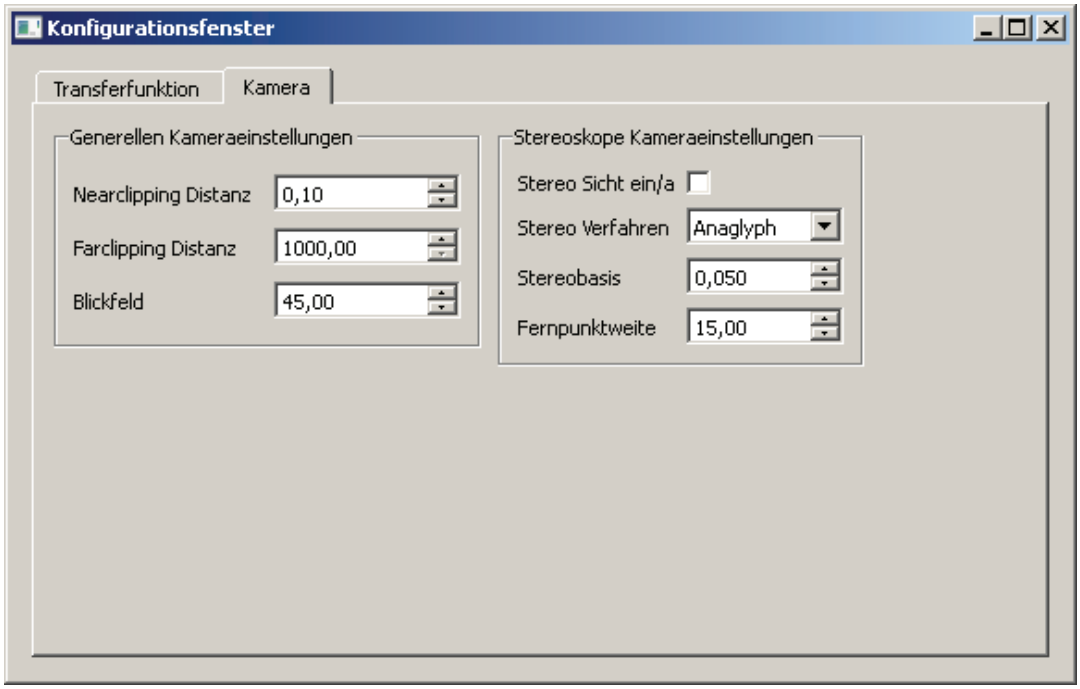

Abbildung 5.5: Screenshot des Konfigurationsfenster(Reiter: Kamera).

Für die numerischen Werte Nearclipping Distanz, Farclipping Distanz, Blickfeld, Stereobasis und Ferpunktweite werden Elemente von Typ *QDoubleSpinBox* verwendet, da die Funktionen in denen sie verwendet werden Werte vom Typ *double* erwarten. Das Ein- und Ausschalten der Stereoprojektion erfolgt über eine Controll vom Typ *QCheckBox*. Zur Auswahl des Stereoprojektionsverfahrens wird eine *QComboBox* verwendet, da der Benutzer hier nur eine Auswahl angeboten bekommen muss. Im Programm werden zwei Verfahren angeboten, Anaglyph und Polarisation. Das Polarisationsverfahren ist im Programm nicht implementiert, der Eintrag soll nur die Möglichkeit zeigen wie das Programm später um ein anderes Projektionsverfahren erweitert werden kann.
Auf die Instanz der Kamera wird über ein referenziertes Feld, vom Typ *iCamera**, zugegeriffen. Die referenzierte Instanz wird, von *OpenGLWidget* erzeugt. Die Referenz wird beim Programmstart vom Anzeigefenster mit der Methode *m_bSetCamera(...)* gesetzt.
Alle Einstellungen für die Stereoprojektion sind in einem referenzierten Feld vom Typ *StereoSettings** abgelegt. Das Objekt auf welches referenziert wird, wird von *OpenGLWidget* erzeugt. Die Referenz wird beim Programmstart vom Anzeigefenster mit der Methode *m_bSetStereoSettings(...)* gesetzt.
In Tabelle 5.4 sind die vom Kamerareiter implementierten Slots mit ihren Funktionsbeschreibungen aufgeführt.

Slotname	**Signalherkunft**	**Beschreibung**
m_vStereoAnToggle(bool stat)	Stereoprojektion An/Aus-Checkbox	Setzt in den referenzierten Stereosettings den Parameter ob die Stereoprojektion ein- oder ausgeschaltet ist. Das Signal *m_vChangedCamera()* wird ausgelöst.
m_sStereoVerfahrenIndexChanged(int index)	Stereoverfahren Droppdownbox	Verändert das Stereoprojektionsverfahren in den referenzierten Stereosettings. Das Signal *m_vChangedCamera()* wird ausgelöst.
m_vStereoBasisValueChanged(double val)	Stereobasis Spinbox	Übernimmt den Wert für die Stereobasis aus der Spinbox in die referenzierten Stereosettings. Das Signal *m_vChangedCamera()* wird ausgelöst.
m_vFernpunktweiteValueChanged(double val)	Fernpunkt Spinbox	Die veränderte Fernpunktweite wird in die referenzierten Stereosettings geschrieben. Das Signal *m_vChangedCamera()* wird ausgelöst.
m_vNearValueChanged(double val)	Nearclipping Spinbox	Mit der Methode *m_bSetNear(...)* des *iCamera*-Interfaces wird der neue Wert für die Nearclipping Distanz gesetzt. Das Signal *m_vChangedCamera()* wird ausgelöst.
m_vFarValueChanged(double val)	Farclipping Spinbox	Der neue Wert für die Farclipping Distanz wird mit der Methode *m_bSetFar(...)* des *iCamera*-Interfaces gesetzt. Das Signal *m_vChangedCamera()* wird ausgelöst.
m_vFieldOfViewValueChanged(double val)	Blickfeld Spinbox	Die Methode *m_SetFieldOfView(...)* des iCamera-Interfaces wird aufgerufen um den neuen Wert für das Blickfeld zu setzen. Das Signal *m_vChangedCamera()* wird ausgelöst.

Tabelle 5.4: Slots des Kamerareiters

5.1.3 Bildstapel-Dialog

In Abbildung 5.6 ist die Klassenstruktur des Bildstapel-Dialogs dargestellt, welche im folgenden erläutert wird.

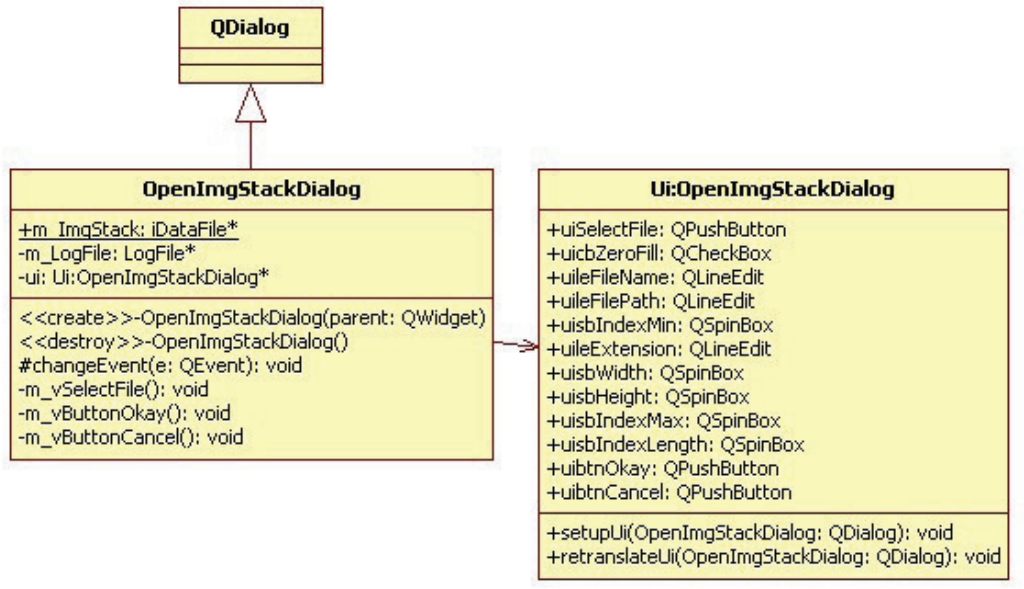

Abbildung 5.6: Klassendiagramm des Bildstapel-Dialogs. Im UI-Teil des Klassendiagramms wurden Label, die der Beschriftung der Steuerelemente dienen, und Layoutelemente entfernt.

Der Bildstapel-Dialog dient als Eingabemaske aller zum Laden eines Bildstapels notwendigen Daten. In Abbildung 5.7 ist das Layout des Bildstapel-Dialogs mit allen Steuerelementen zu sehen.

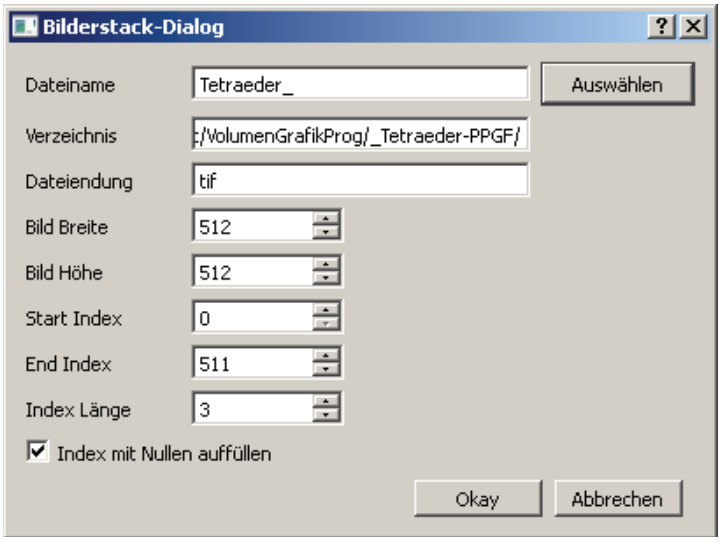

Abbildung 5.7: Screenshot des Bildstapel-Dialogs.

Die Bildstapelparameter Dateiname, Verzeichnis und Dateiendung erfordern String-Variablen, aus diesem Grund werden für ihre Erfassung *QLineEdit*-Elemente verwendet. Höhe und Breite des Bildstapels, Start/End-Index und Index-Länge sind immer ganzzahlig, zur Erfassung werden Controlls vom Typ *QSpinBox* verwendet.
Besonderheit des Bildstapel-Dialogs ist das statische Feld *m_ImgStack*. Das Attribut *static* soll hier bewirken das, der erzeugte Bildstapel, auch nach dem Beenden des Bildstapel-Dialogs noch im Speicher zugegriffen werden kann.
In Tabelle 5.5 sind die vom Bildstapel-Dialog implementierten Slots mit ihren Funktionsbeschreibungen aufgeführt.

Slotname	**Signalherkunft**	**Beschreibung**
m_vSelectFile()	Dateiname-Auswahl-Button	Öffnet einen Dateiauswahl-Dialog. Abhängig von der ausgewählten Datei werden Eingabewerte für den Dialog approximiert, die auf dem Inhalt des Verzeichnises, in dem sich die ausgewählte Datei befindet, beruhen. Die approximierten Werte erfordern jedoch eine Überprüfung.
m_vButtonOkay()	Okay Button	Erzeugt einen Bildstapel, unter Verwendung der Klasse ImgStack. Das erzeugt *ImgStack*-Objekt wird im Feld *m_ImgStack*, das von Typ *iDataFile* ist, abgelegt. Anschließend wird der Dialog mit dem Ergebnis Accept beendet.
m_vButtonCancel()	Abbrechen Button	Setzt das Feld *m_ImgStack* gleich 0 und beenden den Dialog mit dem Ergebnis *Reject*.

Tabelle 5.5: Slots des Bildstapel-Dialogs

5.1.4 Einstellungen-Dialog

In Abbildung 5.8 ist die Klassenstruktur des Einstellungen-Dialogs dargestellt, welche im folgenden erläutert wird.

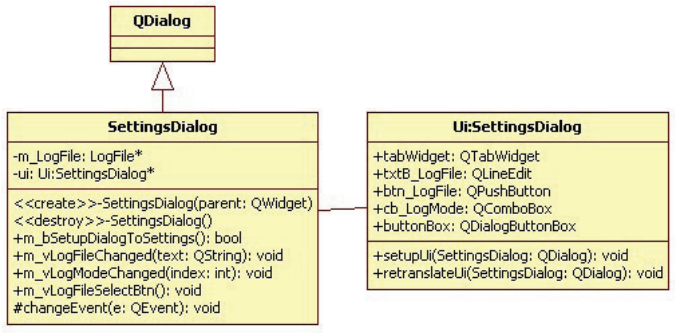

Abbildung 5.8: Klassendiagramm des Einstellungen-Dialogs. Im UI-Teil des Klassendiagramms wurden Label, die der Beschriftung der Steuerelemente dienen, und Layoutelemente entfernt.

Der Einstellungen-Dialog dient dazu generelle Anwendungseinstellungen vornehmen zu können. In Abbildung 5.9 ist das Layout des Bildstapel-Dialogs mit allen Steuerelementen zu sehen.

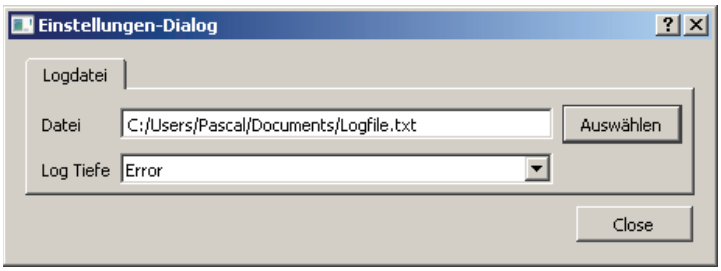

Abbildung 5.9: Screenshot des Einstellungen-Dialogs.

Das oberste Element des Dialogs ist ein *QTabWidget*, zur Zeit enthält es nur einen Reiter für die Verwaltung der Logdatei. In diesem Dialog könnte später eine Auswahl des Verfahrens zur Volumendarstellung statt finden oder eine Konfiguration des aktiven Verfahrens. Der Dateiname der *Logdatei* wird über ein *QLineEdit* editiert. Auf den Begriff Logdatei wird in Abschnitt 5.2.3.1 noch eingegangen. Die *Log-Tiefe* hat vordefinierte Werte, *Error*, *Warning* und *Debug*, die über eine *QComboBox* wählbar sind.
Die Besonderheit an dem Einstellungen-Dialog ist, dass die geänderten Einstellungen sofort beim Verlassen des geänderten Felds übernommen werden, ohne das ein Klicken auf

einen „Dialog-Okay“-Button notwendig ist. In Tabelle 5.6 sind die vom Einstellungen-Dialog implementierten Slots mit ihren Funktionsbeschreibungen aufgeführt.

Slotname	**Signalherkunft**	**Beschreibung**
m_vLogFileChanged(QString text)	Logdatei Lineedit	Ändert, in der Klasse *LogFile*, den Pfad unter dem die Logdatei gespeichert wird. Da die Klasse *LogFile* implementiert das Singelton-Muster, die Änderung wirkt sich also auf die gesamte Anwendung aus.
m_vLogModeChanged(int index)	Log-Tiefe Combobox	Ändert, in der Klasse *LogFile*, die Tiefe des Logings.
m_vLogFileSelectBtn()	Logdatei-Auswahl Button	Öffnet einen „Dateiauswahl-Dialog“ und schreibt den ausgewählten Pfad mit Dateiname in das Logdatei-Lineedit.

Tabelle 5.6: Slots des Einstellungen-Dialogs

5.2 Klassen

In diesem Abschnitt werden die Teile des Programms beleuchtet, die im Hintergrund arbeiten. Das System kann in drei Unterabschnitte gegliedert werden, das Volumen, das OpenGLFenster und einen Abschnitt der sich mit Hilfsklassen beschäftigt die in mehreren Klassen zur Anwendung kommen. Im Anhang unter A.4 ist eine Übersicht der Gesamten Klassenstruktur des Programms zu finden.

5.2.1 Volumen

Das Volumen repräsentiert die Daten der Anwendung. Es werden Methoden zur Verfügung gestellt die die 3D-Darstellung der Daten ausführen. In Abbildung 5.10 ist die Klassenstruktur der *Volume*-Klasse dargestellt, welche im folgenden erläutert wird.

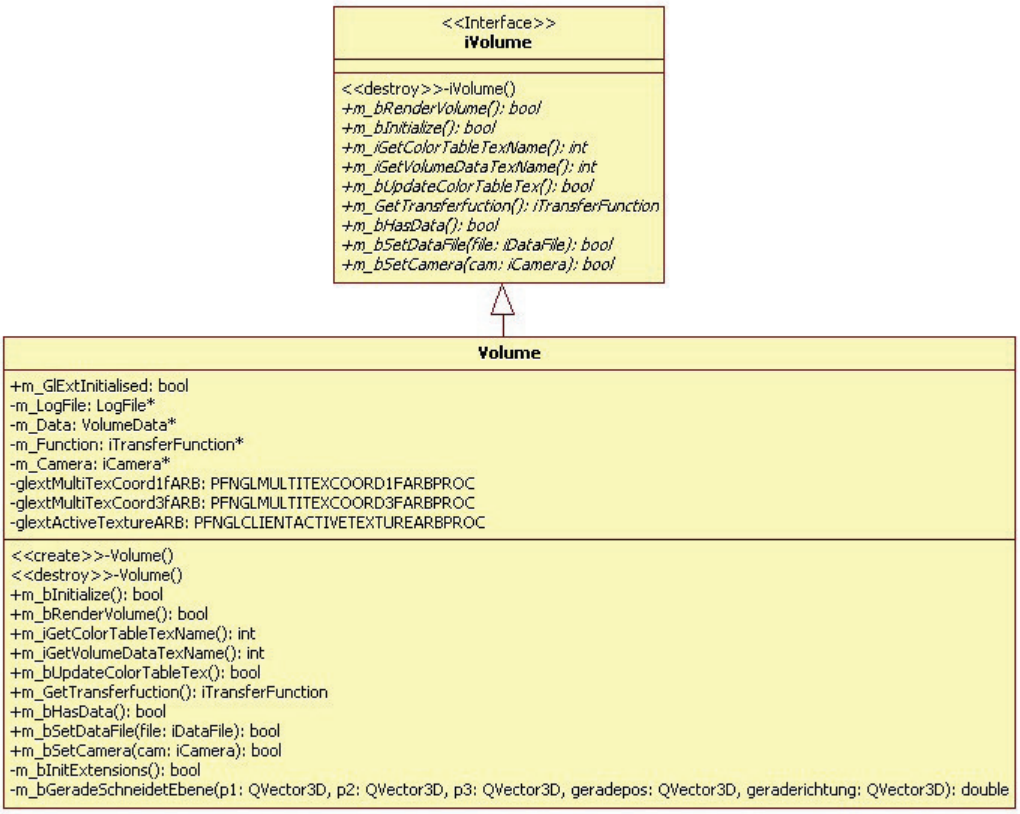

Abbildung 5.10: Klassendiagramm der *Volume*-Klasse.

iVolume Um später andere Volumenverfahren implementieren zu können, wurden die Methoden die von anderen Klassen benötigt werden in das Interface *iVolume* gekapselt.

Die Methoden des Interfaces werden in Tabelle 5.7 beschrieben.

Methode	**Beschreibung**
m_bRenderVolume()	Zeichnet das Volumen mit Hilfe von OpenGL-Funktionen.
m_bInitialize()	Initialisiert die Transferfunktion- und Datentextur und läd die vorhandenen Daten in den Grafikkartenspeicher.
m_iGetColorTableTexName()	Liefert die Adresse der *Textureinheit* [Shr09] in der die ein dimensionale Transferfunktiontextur abgelegt wurde.
m_iGetVolumeDataTexName()	Liefert die Adresse der Textureinheit in der die dreidimensionale Volumendatentextur abgelegt wurde.
m_bUpdateColorTableTex()	Interpoliert die Transferfunktiontextur und läd sie erneut in den Grafikkartenspeicher.
iTransferFunction* m_GetTransferfuction()	Gibt eine Referenz auf das im Volumen verwendete Transferfunktion-Objekt.
m_bHasData()	Prüft ob das Volumen Daten zur Darstellung enthält.
m_bSetDataFile(iDataFile* file)	Die als Referenz übergebene Datendatei, wird als Datenquelle für das Volumen verwenden.
m_bSetCamera(iCamera* cam)	Setzt eine Referenz auf die verwendete Kamera. Diese wird zur Berechnung der 3D-Darstellung benötigt.

Tabelle 5.7: Interfacebeschreibung von *iVolume*

Volume Die *Volume*-Klasse implementiert das *iVolume* Interface. Als Verfahren zur Volumendarstellung wurde das Slicing-Verfahren gewählt. Die Slices, die hierfür durch die 3D-Textur gezeichnet werden, werden in der Methode *m_bRenderVolume()* gerendert. In Codelisting 5.2 wird gezeigt wie die Slices gezeichnet werden.

Listing 5.2: Codeausschnitt zum Zeichnen von Slices in Y-Richtung von -0.5 bis 0.5

```
1  GLdouble high=-0.5, texhigh = 0;
2  double Count = 700;
3  // Zeichnet von -y nach +y
4  for(GLdouble i=0; i<=Count; i++)
5  {
6      glBegin(GL_QUADS);
7      glextMultiTexCoord3fARB(GL_TEXTURE0_ARB, 0.0, 0.0, texhigh);
8      glVertex3d(-0.5, high, -0.5);
9      glextMultiTexCoord3fARB(GL_TEXTURE0_ARB, 1.0, 0.0, texhigh);
10     glVertex3d(0.5, high, -0.5);
11     glextMultiTexCoord3fARB(GL_TEXTURE0_ARB, 1.0, 1.0, texhigh);
12     glVertex3d(0.5, high, 0.5);
13     glextMultiTexCoord3fARB(GL_TEXTURE0_ARB, 0.0, 1.0, texhigh);
14     glVertex3d(-0.5, high, 0.5);
15     glEnd();
16
17     high += 1.0/Count;
18     texhigh += 1.0/Count;
19 }
```

Die im Listing gezeigte Schleife zeichnet 700 *QL_QUADS*, Viereck-Polygone. Das ablaufenden Programm legt einen Stapel von 700 Schnitten durch die 3D-Textur, vergleichbar mit einem Stapel Fotos die übereinander gestapelt werden.
Bei der Implementierung trat hier das Problem auf, dass die Transparenz durch die Slices nur dann richtig berechnet wird, wenn die Slices in einer bestimmten Richtung ausgerichtet und gezeichnet werden. Dies Problematik soll in Abschnitt 5.3.3 mit ihrer Lösung erläutert werden.

Wie schon in der Beschreibung des *iVolume*-Interfaces erwähnt, werden mehrere Textureinheiten auf der Grafikkarte verwendet. Um die Nutzung mehrerer Textureinheiten zu ermöglichen, muss eine sogenannte *OpenGL-Extension*[3] verwendet werden. Hier werden drei relevante Methoden importiert, deren Funktion in Tabelle 5.8 beschrieben werden.

Methode	**Beschreibung**
glextMultiTexCoord1fARB	Setzt Texturkoordinaten einer ein Dimensionalen Textur, abhängig von der verwendeten Textureinheit.
glextMultiTexCoord3fARB	Setzt Texturkoordinaten einer drei Dimensionalen Textur, abhängig von der verwendeten Textureinheit.
glextActiveTextureARB	Legt die aktive Textureinheit fest.

Tabelle 5.8: Importierte Methoden der Multitexturing-Extension von OpenGL.

Das Importierten der aufgelisteten Extensionmethoden wird in der Methode *m_bInitExtensions()* durchgeführt. In Codelisting 5.3 gezeigt eine auf das wesendliche beschränkte Version der Funktion *m_bInitExtensions()*. Es werden unter Verwendung der OpenGL-Funktion *wglGetProcAddress(...)* Funktionspointer auf die entsprechenden Extension-Methodes importiert.

Listing 5.3: Codeausschnitt zum Importieren der Extension-Methode *glMultiTexCoord3fARB* auf den Funktionszeiger glextMultiTexCoord3fARB.

```
1  bool Volume::m_bInitExtensions()
2  {
3      // Multi-Texturing
4      glextMultiTexCoord1fARB = (PFNGLMULTITEXCOORD1FARBPROC)
5                                wglGetProcAddress("glMultiTexCoord1fARB");
6      glextMultiTexCoord3fARB = (PFNGLMULTITEXCOORD3FARBPROC)
7                                wglGetProcAddress("glMultiTexCoord3fARB");
8      glextActiveTextureARB = (PFNGLCLIENTACTIVETEXTUREARBPROC)
9                               wglGetProcAddress("glActiveTextureARB");
10     return true;
11 }
```

Die *Volumen*-Klasse besitzt referenzierte Felder der Typen *iTransferFunction*, *iVolumeData* und *iCamera* die im folgenden erklärt werden sollen.

5.2.1.1 Transferfunktion

In Abbildung 5.11 ist die Klassenstruktur der *Transferfunktion*-Klasse dargestellt, welche im folgenden erläutert wird.

[3]Mit OpenGL-Extensions [Shr09] wird der Funktionsumfang von OpenGL erweitert.

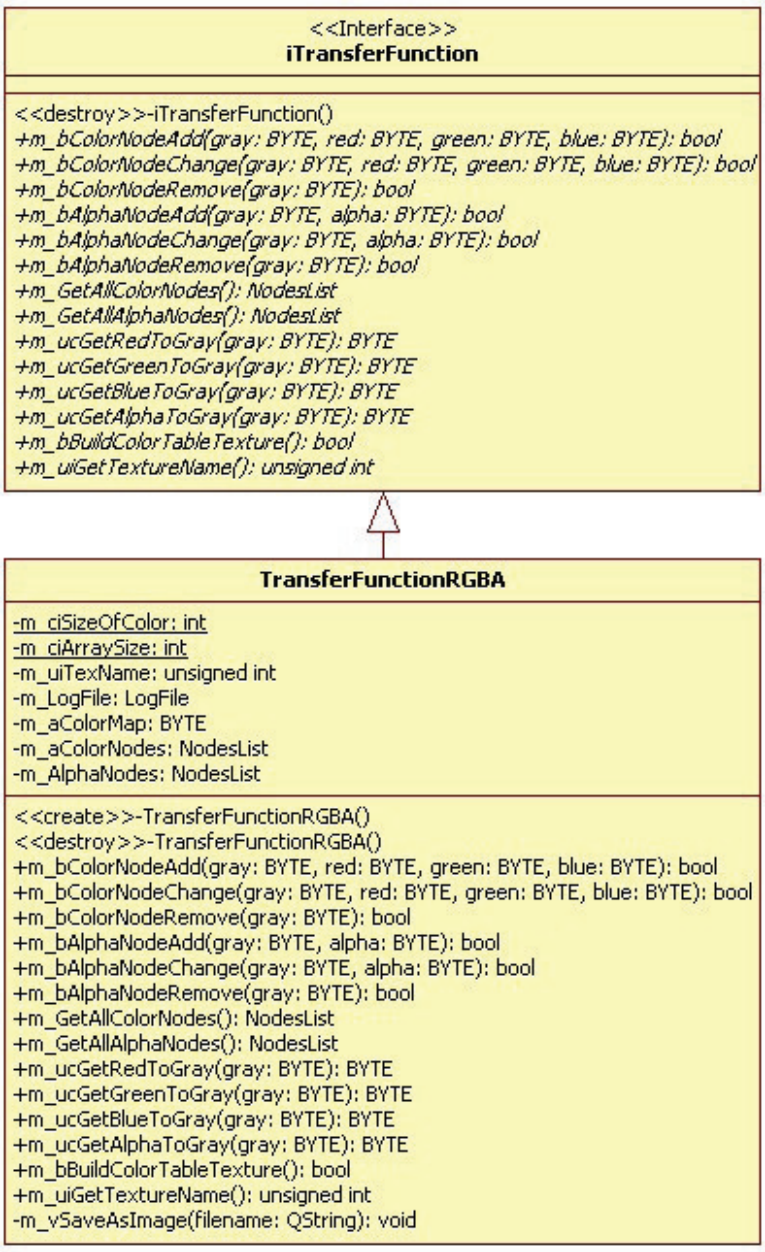

Abbildung 5.11: Klassendiagramm der *Transferfunktion*-Klasse.

iTransferFunction Um später Transferfunktionen mit zum Beispiel anderen Interpolationsverfahren oder Farbtiefen, implementieren zu können wurden die Methoden, die für die Arbeit mit der Transferfunktion notwendig sind, in das Interface *iTransferFunction* gekapselt. Die Methoden des Interfaces werden in Tabelle 5.9 beschrieben.

Methode	Beschreibung
m_bColorNodeAdd(BYTE gray, BYTE red, BYTE green, BYTE blue)	Fügt der Transferfunktion einen Farbknoten mit den übergebenen Rot-/Grün- und Blauwerten hinzu.
m_bColorNodeChange(BYTE gray, BYTE red, BYTE green, BYTE blue)	Ändert den Farbknoten mit dem entsprechenden Grauwert, auf den übergebenen Rot-/Grün- und Blauwert.
m_bColorNodeRemove(BYTE gray)	Entfernt den Farbknoten mit dem übergebenen Grauwert aus der Transferfunktion.
m_bAlphaNodeAdd(BYTE gray, BYTE alpha)	Fügt der Transferfunktion einen Transparenzknoten mit den übergebenen Werten hinzu.
m_bAlphaNodeChange(BYTE gray, BYTE alpha)	Ändert den Transparenzknoten mit dem entsprechenden Grauwert, auf den übergebenen Transparenzwert.
m_bAlphaNodeRemove(BYTE gray)	Entfernt den Transparenzknoten mit dem übergebenen Grauwert aus der Transferfunktion.
m_GetAllColorNodes()	Liefert einen Verweis auf die Farbknotenliste.
m_GetAllAlphaNodes()	Liefert einen Verweis auf die Transparenzknotenliste.
m_ucGetRedToGray(BYTE gray)	Liefert den Rotwert zum übergebenen Grauwert.
m_ucGetGreenToGray(BYTE gray)	Liefert den Grünwert zum übergebenen Grauwert.
m_ucGetBlueToGray(BYTE gray)	Liefert den Blauwert zum übergebenen Grauwert.
m_ucGetAlphaToGray(BYTE gray)	Liefert den Transparenzwert zum übergebenen Grauwert.
m_bBuildColorTableTexture()	Interpoliert aus den vorhandenen Knoten der Transferfunktion das Transferfunktion-Array und läd die daraus erzeugte eindimensionale Transferfunktiontextur in den Speicher der Grafikkarte.
m_uiGetTextureName()	Gibt die Adresse der Textureinheit zurück in der die Transferfunktiontextur abgelegt ist.

Tabelle 5.9: Interfacebeschreibung von *iTransferFunction*

TransferFunctionRGBA Die Klasse *TransferFunctionRGBA* implementiert das *iTransferFunction* Interface. Zum Speichern der Farb- und Transparenzknoten dienen die Felder *m_aColorNodes* und *m_AlphaNodes*. Die Listen zur Verwaltung der Knoten sind vom Typ *NodesList*, die Klasse *NodesList* wird in Abschnitt 5.2.3.2 erläutert. Ein *NodesList* Objekt kann jedoch nur eine Datenkomponente der Farbe verwalten, aus diesem Grund gibt es für Rot-, Grün-, Blau- und Alphakomponente jeweils ein *NodesList* Instanz. Das Transferfunktions-Array aus dem die eindimensionale Transferfunktion-Textur erzeugt wird ist im Feld *m_aColorMap* abgelegt. Die Größe des Transferfunktions-Array wird wiefolgt berechnet $Arraygröße = m\_ciSizeOfColor * m\_ciArraySize = 4 * 256 = 1024$. Da der Datentyp des Transferfunktions-Array *unsigend char* ist, was einem *Byte* entspricht, ist das Transferfunktions-Array 1kByte groß.

5.2.1.2 Volumedaten

In Abbildung 5.12 ist die Klassenstruktur der *VolumeData*-Klasse dargestellt, welche im folgenden erläutert wird.

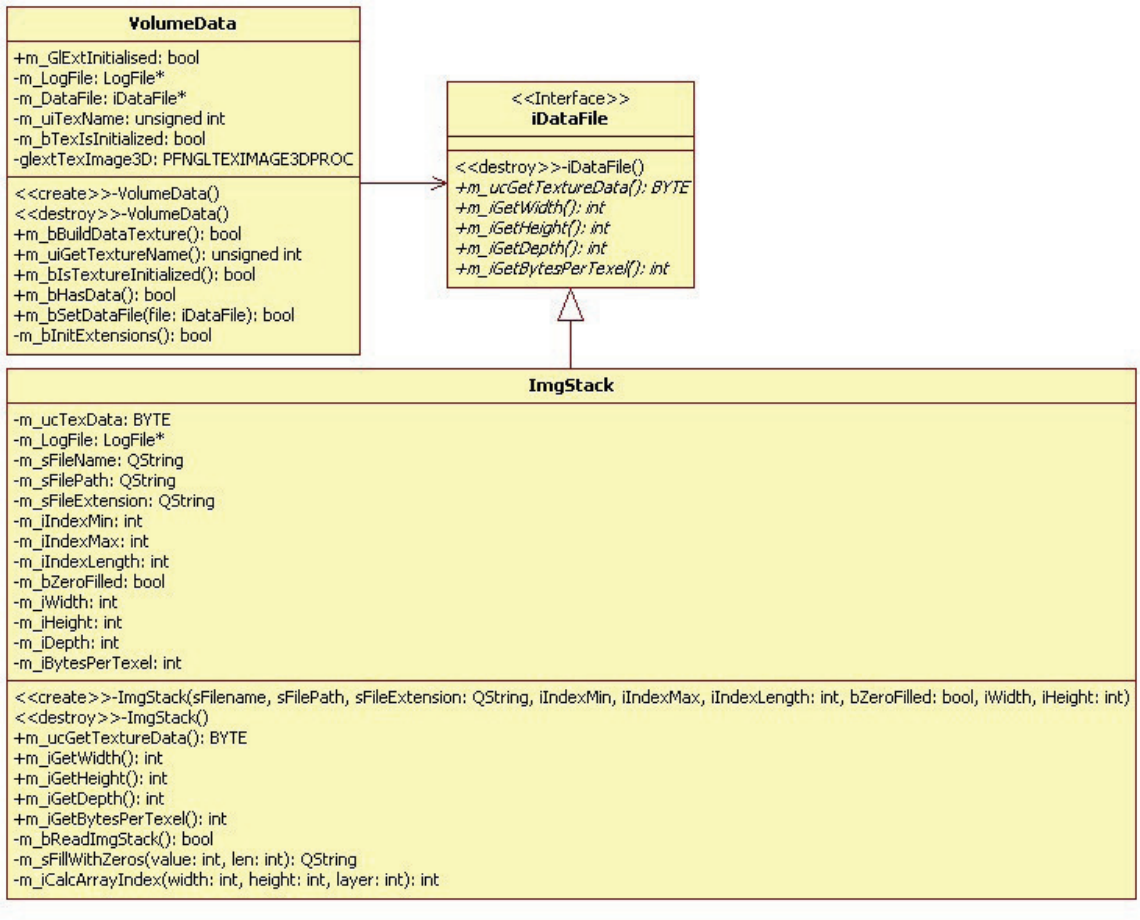

Abbildung 5.12: Klassendiagramm der VolumeData-Klasse.

VolumeData Die Klasse *VolumenData* ist für das Verwalten der Volumendaten-Textur auf der Grafikkarte zuständig. Um eine 3D-Textur auf der Grafikkarte erzeugen zu können ist es notwendig die OpenGL-Extension Funktion *glTexImage3D* zu verwenden. Der Funktionszeiger auf die *glTexImage3D*-Funktion wird auf das Feld *glextTexImage3D* importiert. Zum Zugriff auf die Basisdaten für die Volumen-Textur wird auf ein Objekt, welches das *iDataFile* Interface implementiert, zugegriffen.

iDataFile Um später auf andere Dateitypen oder Dateistrukturen zugreifen zu können wurden die Methoden, die für den Zugriff auf die Basis Volumendaten notwendig sind, in das Interface *iDataFile* gekapselt. Die Methoden des Interfaces werden in Tabelle 5.10 beschrieben.

Methode	Beschreibung
m_ucGetTextureData()	Liefert einen Verweis auf das Volumen-Array in dem sich die Basisdaten für die Volumen-Textur befinden.
m_iGetWidth()	Gibt die Breite der Textur zurück.
m_iGetHeight()	Gibt die Höhe der Textur zurück.
m_iGetDepth()	Gibt die Tiefe der Textur zurück.
m_iGetBytesPerTexel()	Gibt die Anzahl der Farbwerte die für jeden Texel im Array gespeichert wurde zurück. Wird für jeden Texel nur ein Grauwert aus jedem Voxel gespeichert, ist dieser Wert gleich eins.

Tabelle 5.10: Interfacebeschreibung von *iDataFile*

ImgStack Die Klasse *ImgStack* implementiert das *iDataFile* Interface. Aufgabe der *ImgStack* Klasse ist es einen Bildstapel in ein Array einzulesen, welches zum Erzeugen der dreidimensionalen Volumen-Textur geeigent ist. Im Konstruktur der *ImgStack* Klasse werden die notwendigen Basisdaten des Bildstapels übergeben. Die Methode *m_bReadImgStack()* liest den Bilderstack in das Array-Feld *m_ucTexData* ein. Die Elemente des Array-Felds *m_ucTexData* sind vom Typ *unsigned char*. In Codelisting 5.4 wird, in verkürzter Form, gezeigt wie ein Bildstapel von der Methode *m_bReadImgStack()* geladen wird.

Listing 5.4: Codeausschnitt der Methode *m_bReadImgStack()* der *ImgStack* Klasse.

```
1  bool ImgStack::m_bReadImgStack()
2  {
3      m_ucTexData = (BYTE *)malloc(m_iWidth * m_iHeight * m_iDepth * m_iBytesPerTexel);
4                  ...
5      for(int ImgIndex = m_iIndexMin; ImgIndex <= m_iIndexMax; ImgIndex++)
6      {
7          ...
8          QImage* image = new QImage();
9          image->load(pfad);
10         for(int s=0; s < m_iWidth; s++)
11         {
12             for(int t=0; t < m_iHeight; t++)
13             {
14                 double Rot, Gruen, Blau;
15                 Rot = qRed(image->pixel(s,t));
16                 Gruen = qGreen(image->pixel(s,t));
17                 Blau = qBlue(image->pixel(s,t));
18                 double gray = (Rot+Gruen+Blau)/3.0;
19                 m_ucTexData[m_iCalcArryIndex(s,t,ImgIndex)] = static_cast<BYTE>(gray);
20             }
21         }
22         delete image;
23     }
24     return true;
25 }
```

Wie im Codelisting der *m_bReadImgStack()* zu sehen, wird die Arrayindizierung von der Methode *m_iCalcArryIndex(...)* durchgeführt. Die Indexberechung hängt von der Nummer des Bildes und der Höhen-/Breitenkoordinate des Pixels im Bild ab, es wird eine Sortierung wie in Tabelle 5.11 gezeigt benötigt.

Arrayindex	Bildnummer	X-Koordinate	Y-Koordinate
0	1	0	0
1	1	1	0
2	1	2	0
...			
262144	1	511	511
262145	2	0	0
262146	2	1	0
262147	2	2	0
...			
134217727	512	510	511
134217728	512	511	511

Tabelle 5.11: Array Indizierung des Volumen-Array. Die X-/Y-Koordinaten beziehen sich immer auf das Bild mit der Bildnummer.

5.2.2 OpenGL-Fenster

Das OpenGL-Fenster ist das Element der Anwendung welches die eigentlich dreidimensionale Darstellung übernimmt. In Abbildung 5.13 ist die Klassenstruktur der *OpenGL-Widget*-Klasse dargestellt, welche im folgenden erläutert wird.

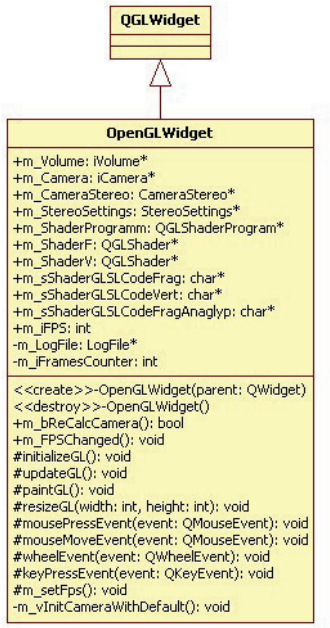

Abbildung 5.13: Klassendiagramm der *OpenGLWidget*-Klasse.

Die *OpenGLWidget* Klasse erbt von der Qt-Basisklasse *QGLWidget*. Das Feld *m_Volume* vom Typ *iVolume** verweist auf die aktuell anzuzeigenden Volumendaten. Über das Feld *m_Camera* vom Typ *iCamera** wird die im Fenster verwendete Kamera zugegriffen. Im Feld *m_CameraStereo* vom Typ *CameraStereo** wird auf die verwendete Stereokamera

verwiesen, welche die stereoskopischen Kameraeinstellungen verwaltet.
OpenGLWidget implementiert einen Slot, *m_setFps()*, und ein Signal, *m_FPSChanged()*. Der Slot ist privat, er wird alle 500ms Sekunden von einem Timer aufgerufen und aktuallisiert das Feld, *m_iFPS*, in dem die Anzahl der gezeichneten Bilder gespeichert wird. Im Anschluss an die Frameberechnung wird das Signal *m_FPSChanged()* ausgelöst, welches vom Anzeigefenster empfangen wird um die Statusleiste zu aktuallisieren.
Die *OpenGLWidget* Klasse verwaltet ebenfalls die Shader, welche für die Volumendarstellung verantwortlich sind. Die Shader werden über ein Objekt vom Typ *QGLShaderProgram*, welches im Feld *m_Shaderprogramm* gehalten wird, auf den Fragment- bzw. Vextexprozessor der Grafikkarte geladen. In den Feldern *m_sShaderGLSLCodeVert*, *m_sShaderGLSLCodeFrag* und *m_sShaderGLSLCodeFragAnaglyp* sind die Quelltexte der Shader abgelegt, diese sind vom Typ *char** und werden im Konstruktor der *OpenGLWidget* Klasse mit dem jeweiligen Code befüllt. Im Abschnitt 5.3.2 wird die Implementierung und Verwendung der Shader im Detail betrachtet.
Eine der Hauptaufgaben der *OpenGLWidget* Klasse ist die Unterscheidung zwischen normaler und stereoskopischer Kameraführung. Im Feld *m_StereoSettings* vom Typ *StereoSettings** werden die Einstellungen für die stereoskopische Darstellung gespeichert. Das Feld *m_StereoSettings* wird, über eine Referenz, direkt vom Konfiguration-Fenster geändert. Die Unterscheidung der Kamera wird in der *paintGL()* Methode durchgeführt, die dafür notwendige Struktur wird in Codelisting 5.5 dargestellt.

Listing 5.5: Codeausschnitt der Methode *paintGL()* der *OpenGLWidget* Klasse. Der Code für das Binden der Shader wurde entfernt.

```
1  void OpenGLWidget::paintGL()
2  {
3      if(m_Volume->m_bHasData() == true)
4      {
5          if(m_StereoSettings->m_bStereoOn == true)
6          {
7              switch(m_StereoSettings->m_StereoMethod)
8              {
9                  case Anaglyp:
10                 {
11                     ...
12
13                     // render the right part of stereoscreen
14                     glDrawBuffer(GL_BACK);
15                     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
16                     glMatrixMode(GL_PROJECTION);
17                     glLoadIdentity();
18
19                     m_CameraStereo->m_glbSetupFrustum();
20                     glMatrixMode(GL_MODELVIEW);
21                     glLoadIdentity();
22                     m_CameraStereo->m_glbSetupLookAt();
23
24                     glPushMatrix();
25                     glColorMask(GL_TRUE, GL_FALSE, GL_FALSE, GL_TRUE);
26                     m_Volume->m_bRenderVolume();
27                     glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
28                     glPopMatrix();
29                     glFlush();
```

```
30
31
32                      // render the left part of stereoscreen
33                      glDrawBuffer(GL_BACK);
34                      // dont clear the colorbuffer, so the both image parts will be added
35                      glClear(GL_DEPTH_BUFFER_BIT);
36
37                      glMatrixMode(GL_PROJECTION);
38                      glLoadIdentity();
39
40                      m_CameraStereo->m_glbSetupFrustum();
41                      glMatrixMode(GL_MODELVIEW);
42                      glLoadIdentity();
43                      m_CameraStereo->m_glbSetupLookAt();
44                      glPushMatrix();
45
46                      glColorMask(GL_FALSE, GL_TRUE, GL_TRUE, GL_TRUE);
47                      m_Volume->m_bRenderVolume();
48                      glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
49                      glPopMatrix();
50                      glFlush();
51
52                      break;
53                  }
54              }
55          }
56          else
57          {
58              ...
59
60              glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
61              glDrawBuffer(GL_BACK);
62              glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
63
64              glMatrixMode(GL_PROJECTION);
65              glLoadIdentity();
66              m_Camera->m_glbSetupFrustum();
67
68              glMatrixMode(GL_MODELVIEW);
69              glLoadIdentity();
70              m_Camera->m_glbSetupLookAt();
71              glPushMatrix();
72              m_Volume->m_bRenderVolume();
73              glPopMatrix();
74
75              glFlush();
76          }
77      }
78  }
```

Auf das Gesamtkonzept der Farb-Anaglyphenprojektion soll in Abschnitt 5.3.1 eingegangen werden. Das Rendern des nicht stereoskopisch dargestellten Volumens läuft nach folgenden Schritten ab:

1. Binden der Shader (im Codelisting nicht dargestellt)
2. Rücksetzen der Farbmaskierung
3. Rücksetzen des Farb- und Tiefenbuffers
4. Setzen des Frustums, siehe Abschnitt 3.3.2, in auf der Projektionsmatrix [Shr09]

5. Setzen der Kameraeinstellung in der Modelviewmatrix [Shr09]

6. Rendern der Geometrie des Volumens

Die *OpenGLWidget* Klasse implementiert drei Slot Methoden zum Verarbeiten von Mausevents, die vom Nutzer im Fenster ausgelöst werden können. Durch Weitergabe dieser Mouseevents an das Kamera Objekt wird die Kamerabewegung realisiert. Wie die Bewegung der Kamera im Detail implementiert ist, wird im Abschnitt 5.2.2.1 beschrieben.

5.2.2.1 Kamera

In Abbildung 5.14 ist die Klassenstruktur der *Kamera*-Klasse dargestellt, welche im folgenden erläutert wird.

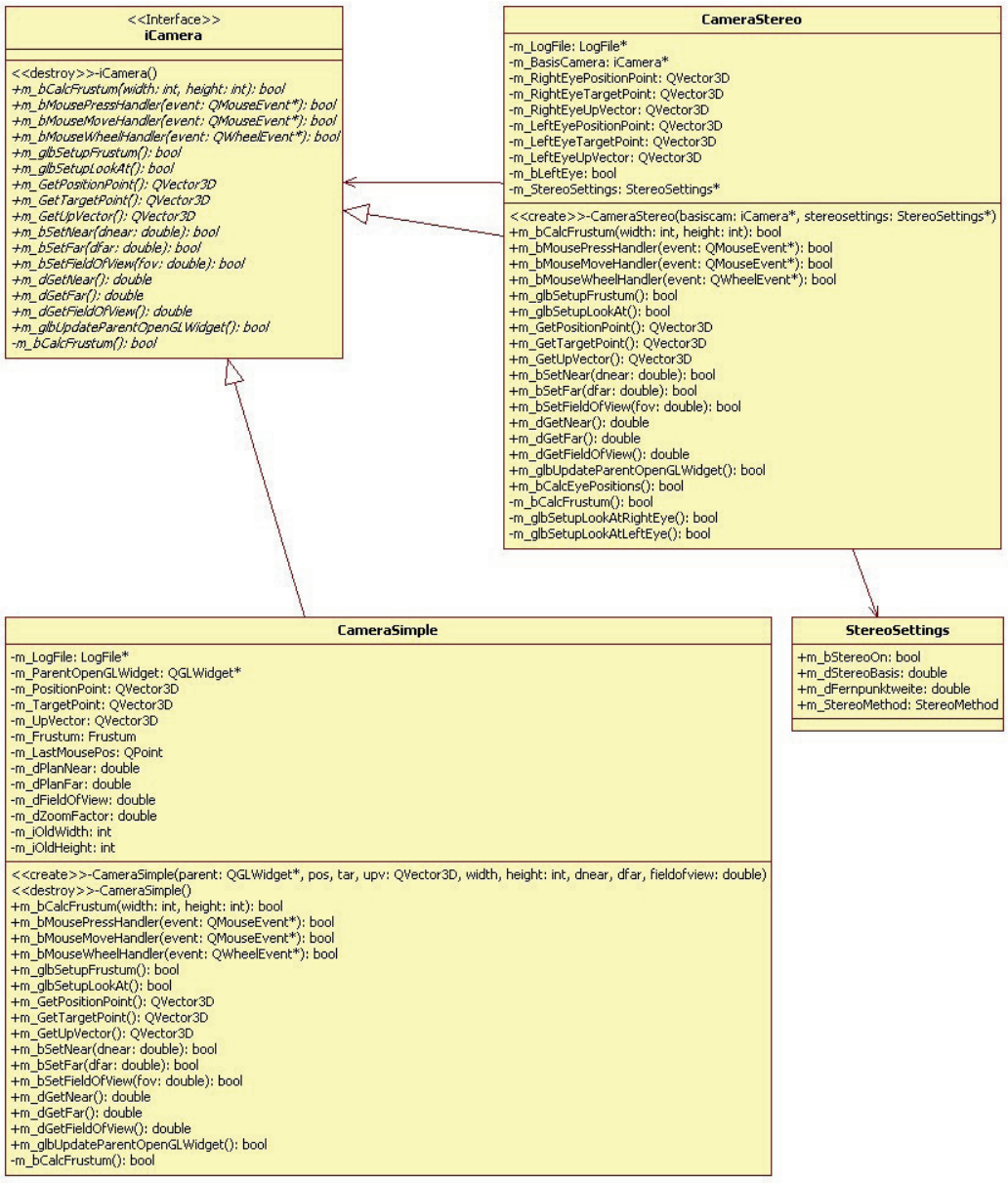

Abbildung 5.14: Klassendiagramm der *Kamera*-Klasse.

iCamera Um die Funktionalität der Kamera besser handhaben zu können, wurden Methoden zum Zugriff in das Interface *iCamera* gekapselt. Durch die Kapelung ist es

später Beispielsweise leichter möglich eine andere Kamera-Steuerung umzusetzen. Die Methoden des Interfaces werden in Tabelle 5.12 beschrieben.

Methode	**Beschreibung**
m_bCalcFrustum(int width, int height)	Berechnet die Werte des Frustums.
m_bMousePressHandler(QMouseEvent *event)	Verarbeitet Mausevents, bei denen eine Maustaste gedrückt wurde.
m_bMouseMoveHandler(QMouseEvent *event)	Verarbeitet Mausevents, bei denen die Maus bewegt wurde.
m_bMouseWheelHandler(QWheelEvent *event)	Verarbeitet Mausevents, bei denen das Mausrad bewegt wurde.
m_glbSetupFrustum()	Über diese Methode wird das Frustum der Kamera mit OpenGL-Funktionen erzeugt. Die Methode wird wärend des Renderns aufgerufen.
m_glbSetupLookAt()	Mit dieser Methode wird die eigentliche Kamerasicht erzeugt. Die Methode wird wärend des Renderns aufgerufen.
m_GetPositionPoint() const	Gibt die Position auf der sich die Kamera befindet zurück.
m_GetTargetPoint() const	Gibt die Koordinaten auf welche die Kamera blickt zurück.
m_GetUpVector() const	Gibt den Nachoben-Vektor der Kamera zurück.
m_bSetNear(double dnear)	Setzt die Near-Distanz der Kamera auf den übergebenen Wert.
m_bSetFar(double dfar)	Setzt die Far-Distanz der Kamera auf den übergebenen Wert.
m_bSetFieldOfView(double fov)	Setzt den Blickwinkel der Kamera auf den übergebenen Wert.
m_dGetNear() const	Gibt den aktuellen Wert der Near-Distanz der Kamera zurück.
m_dGetFar() const	Gibt den aktuellen Wert der Far-Distanz der Kamera zurück.
m_dGetFieldOfView() const	Gibt den aktuellen Wert des Blickwinkels der Kamera zurück.
m_glbUpdateParentOpenGLWidget() const	Sorgt dafür das das OpenGLWidget, welchem die Kamera zugeordnet wurde, neu gezeichnet wird.
m_bCalcFrustum()	Berechnet die Werte des Frustums neu. Wird intern genutzt wenn Near-/Fardistanz bzw. der Zoomfaktor geändert wurde.

Tabelle 5.12: Interfacebeschreibung von *iCamera*

Normale Kamera Die Klasse *CameraSimple* implementiert das *iCamera* Interface. Um die Werte die das *iCamera* Interface benötigt zu speichern werden folgende Typ verwendet. Für die *Kameraposition*, das *Kameraziel* und den *Nachoben-Vektor* wird der Typ *QVector3D* verwendet. Near-/Fardistanz und Blickfeld werden in Feldern vom Typ *double* abgelegt. Um das Frustum speichern wird eine selbst definierte Struktur verwendet. Die *Frustum-Struktur* enthält vier Werte vom Typ *double*, Top, Bottom, Left und Right. Der Verweise auf das *OpenGLWiget*, in welchem sich die Kamera befindet, wird in einem Feld vom Typ *OpenGLWidget** gehalten. Im Konstruktor der Klasse werden die Felder, mit übergebenen Werten, initialisiert oder aus ihnen berechnet.
Die Implementierung der Kamerabewegung durch Mausevents ist der Schwerpunkt der *CameraSimple* Klasse. Für die Kamerabewegung sind die Methoden *m_bMousePressHandler*, *m_bMouseMoveHandler* und *m_bMouseWheelHandler* schon durch das *iCamera* Interface vorgesehen. Da eine Betrachtung des Volumens von allen Seite ermöglicht werden soll, muss der Nutzer der Kamera auf jeder der drei Achsen rotieren können. Die Rotation der Kamera, um das Volumen, soll durch Drücken einer Maustaste initiiert werden. Die Rotationsrichtungen wurde wie in Tabelle 5.13 festgelegt.

Maustaste	Bewegungsrichtung der Maus	Rotation der Kamera um Achse
Links	Nach Links/Nach Rechts	Y-Achse gegen Uhrzeigersin/Y-Achse im Uhrzeigersin
Links	Nach Oben/Nach Unten	X-Achse gegen Uhrzeigersin/X-Achse im Uhrzeigersin
Rechts	Nach Links/Nach Rechts	Y-Achse gegen Uhrzeigersin/Y-Achse im Uhrzeigersin
Rechts	Nach Oben/Nach Unten	Z-Achse gegen Uhrzeigersin/Z-Achse im Uhrzeigersin

Tabelle 5.13: Definition der Bewegungsrichtungen, die von der *CameraSimple* Klasse durch geführt werden.

Grundlage für die Drehung der Kamera ist die räumliche Drehung von Vektoren [Luh03]. Für die Anwendung ist es notwendig den Blickvektor, das ist der Vektor der von der Position der Kamera zum Zielpunkt geht, und den Nachoben-Vektor, ist ein Vektor der aus der Kameraposition heraus die Nachoben-Richtung definiert, um den Zielpunkt zu rotieren. In Codelisting 5.6 ist die Umsetzung mit der räumlichen Drehung mit trigonometrischen Funktionen zu sehen.

Listing 5.6: Codeausschnitt der Methode *m_bMouseMoveHandler* der *CameraSimple* Klasse. Dargestellt wird nur der Teil der Funktion der für die Nutzereingabe der linken Maustaste zuständig ist.

```
1  bool CameraSimple::m_bMouseMoveHandler(QMouseEvent *event)
2  {
3      double dx = event->x() - m_LastMousePos.x();
4      double dy = event->y() - m_LastMousePos.y();
5      double degreex = 0, degreey = 0, degreez = 0;
6      if (event->buttons() & Qt::LeftButton)
7      {
8          if(dx != 0)
9              degreey = (dx) / (2*PI);
10         if(dy != 0)
11             degreex = (dy) / (2*PI);
12
13         // rotation of position-vector
14         QVector3D PosRot = m_PositionPoint - m_TargetPoint;
15         QVector3D oldvector = PosRot;
16
17         // rot Y-achse
18         oldvector = PosRot;
19         PosRot.setX( oldvector.x()*cos(degreey) + oldvector.z()*sin(degreey));
20         PosRot.setY( oldvector.y());
21         PosRot.setZ(-oldvector.x()*sin(degreey) + oldvector.z()*cos(degreey));
22
23         // rot X-achse
24         oldvector = PosRot;
25         PosRot.setX(oldvector.x());
26         PosRot.setY(oldvector.y()*cos(degreex) - oldvector.z()*sin(degreex));
27         PosRot.setZ(oldvector.y()*sin(degreex) + oldvector.z()*cos(degreex));
28
29         m_PositionPoint = PosRot + m_TargetPoint;
30
31         // rotation of up-vector
32         PosRot = m_UpVector - m_TargetPoint;
33
34         // rot Y-achse
35         oldvector = PosRot;
36         PosRot.setX( oldvector.x()*cos(degreey) + oldvector.z()*sin(degreey));
37         PosRot.setY( oldvector.y());
38         PosRot.setZ(-oldvector.x()*sin(degreey) + oldvector.z()*cos(degreey));
```

```
39
40          // rot X-achse
41          oldvector = PosRot;
42          PosRot.setX(oldvector.x());
43          PosRot.setY(oldvector.y()*cos(degreex) - oldvector.z()*sin(degreex));
44          PosRot.setZ(oldvector.y()*sin(degreex) + oldvector.z()*cos(degreex));
45
46          m_UpVector = PosRot + m_TargetPoint;
47      }
48
49      ...
50
51      m_LastMousePos = event->pos();
52      m_ParentOpenGLWidget->updateGL();
53      return true;
54  }
```

Die *CameraSimple* Klasse implementiert eine Zoomfunktion. Um einen Zoom durchzuführen wird bei der Berechnung des *Frustum*, also der Bildfläche, siehe Codelisting 5.7, ein Skalierungsfaktor, *m_dZoomFactor*, einbezogen.

Listing 5.7: Die Methode *m_bCalcFrustum* der *CameraSimple* Klasse. Es wird eine perspektivische Bildfläche [Sch06] berechnet.

```
1  bool CameraSimple::m_bCalcFrustum(int width, int height)
2  {
3      m_iOldWidth = width;
4      m_iOldHeight = height;
5
6      // generate a perspectiv frustum with zoomfaktor
7      double top = m_dPlanNear * tan(m_dFieldOfView * PI /360) * m_dZoomFactor;
8      m_Frustum.Top = top;
9      m_Frustum.Bottom = (-top);
10     m_Frustum.Left = (-top * (double)width/(double)height);
11     m_Frustum.Right = (top * (double)width/(double)height);
12
13     m_ParentOpenGLWidget->updateGL();
14     return true;
15 }
```

Der Zoomfaktor wird wiederum durch das Drehen des Mausrades variiert. Zur Behandung der Mausraddrehung wird die Interfacefunktion *m_bMouseWheelHandler* implementiert. In der Methode *m_bMouseWheelHandler* wird der Zoomfaktor, abhängig von der Drehung des Mausrades, um einen konstanten Wert in- bzw. dekrementiert.

Stereokamera Kamera Die Klasse *CameraStereo* implementiert das *iCamera* Interface. Hauptaufgabe der *CameraStereo* Klasse ist es die Kamerapositionen für die zwei Stereohalbbilder zu berechnen, zu speichern und die Kameraeinstellung beim Rendern durchzuführen. Eine Kamera wird durch ihre Position, ihr Ziel und den Nachoben-Vektor beschrieben. Für das Linke wie das Rechte Halbbild werden Position, Zeil und Nachoben-Vektor berechnet und gespeichert. Als Basis dient ein Objekt welches das *iCamera* Interface implementiert, von nun an als Basiskamera bezeichnet. In Abbildung 5.15 ist die Geometrie der Vektoren dargestellt. Das dargestellte Vektorsystem ist aus Abbildung 3.1 des Kapitels über die Grundlagen der Stereoskopie abgeleitet.

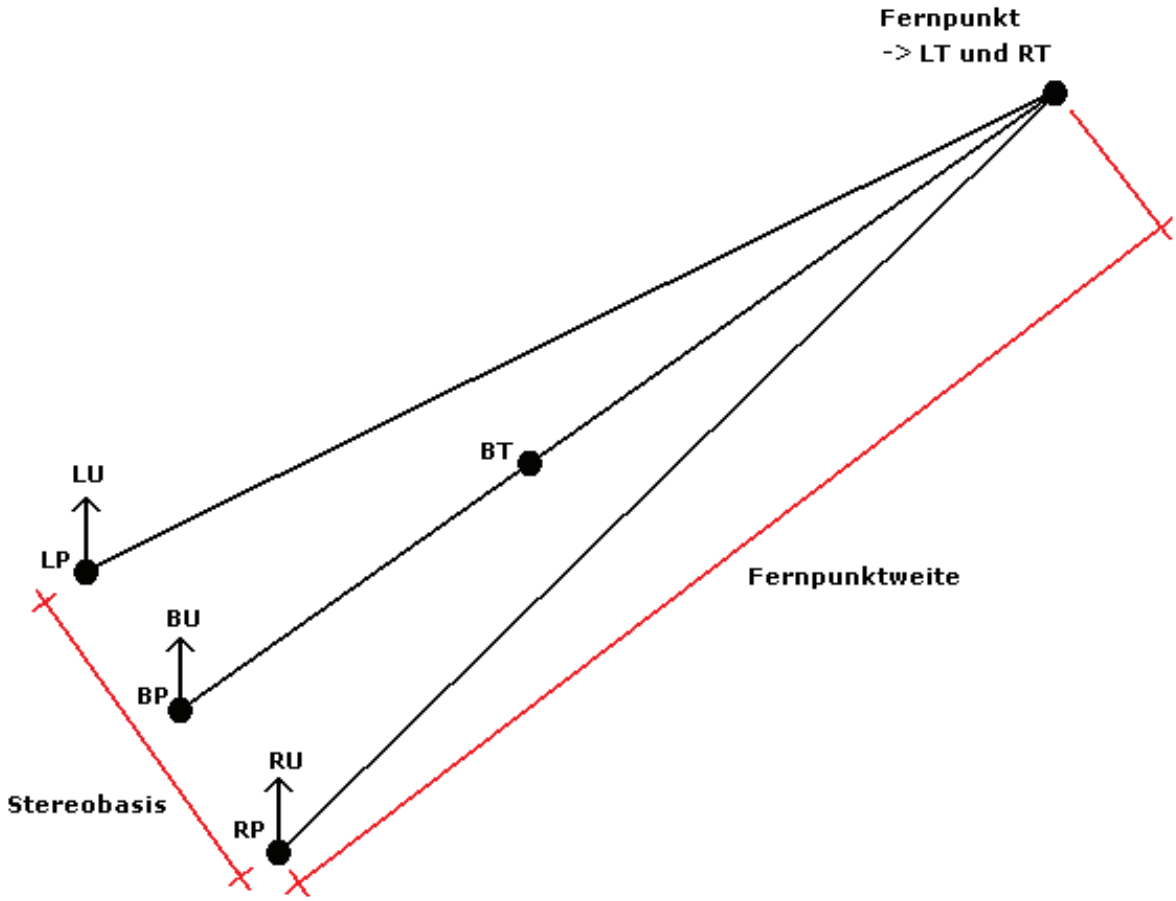

Abbildung 5.15: Skizze der Vektorgeometrie für die *CameraStereo* Klasse. Die Abkürzungen, zum Beispiel BP, sind wie folgt definiert. B=Basis, R=Rechte Halbbildkamera, L=Linke Halbbildkamera, P=Positionspunkt, T=Zielpunkt und U=Nachoben-Vektor.

Die Basiskamera liefert die Basisposition (BP), das Basisziel (BT) und den Basisupvektor (BU). In der Klasse *CameraStereo* werden die Positionen der beiden Stereokameras gehalten, eine Zuordnung der Felder zu den Vektor ist in Tabelle 5.14 aufgeführt.

Feldname	Vektor	Beschreibung
m_RightEyePositionPoint	RP	Punkt in der Szene auf dem sich die Kamera, für das rechte Halbbild, befindet.
m_RightEyeTargetPoint	RT	Punkt in der Szene auf den die Kamera, für das rechte Halbbild, blickt.
m_RightEyeUpVector	RU	Vektor der aus der Position der Kamera, für das rechte Halbbild, nach oben zeigt.
m_LeftEyePositionPoint	LP	Punkt in der Szene auf dem sich die Kamera, für das linke Halbbild, befindet.
m_LeftEyeTargetPoint	LT	Punkt in der Szene auf den die Kamera, für das linke Halbbild, blickt.
m_LeftEyeUpVector	LU	Vektor der aus der Position der Kamera, für das linke Halbbild, nach oben zeigt.

Tabelle 5.14: Zuordnung und Beschreibung der Punkte und Vektoren im CameraStereo-System.

Die Abstände *Stereobasis* und *Fernpunktweite* sind die Parameter die über das Konfigurationsfenster eingestellt werden können. Was es nun zu Berechnen gilt, sind die Punkte RP und LP der Halbbildkameras, sowie den Fernpunkt, welcher für die Punkte RT und LT verwendet wird. Um RP und LP zu berechnen muss zunächst ein Vektor bestimmt werden auf dem der Punkt BP verschoben werden kann. Der Verschiebevektor wird durch das Kreuzprodukt aus dem Nachoben-Vektor (BU) und dem Kameravektor bestimmt. Der Kameravektor ist der Vektor zwischen den Punkten BP und BT. Durch das Faktorisieren mit positiven/negativen Vorzeichen der Stereobasis und addieren zur Basiskameraposition (BP), erhält man die Punkte RP und LP. Der Fernpunkt wird bestimmt in dem man den Kameravektor bestimmt, diesen normalisiert und mit dem Parameter Fernpunktweite multipliziert. RT und LT entsprechen dem Fernpunkt. Da die Positionen der Halbbildkameras nur parallel verschoben werden, sind die Nachoben-Vektoren (RU und LU) identisch zu Nachoben-Vektor der Basiskamera (BU). Die beschriebenen Berechnungen werden in der Methode *m_bCalcEyePositions()* der *CameraStereo* Klasse durchgeführt.

Eine Besonderheit der *CameraStereo* Klasse ist die Methode, *m_glbSetupLookAt()*, zum Einstellen der Kameraposition wärend des Render-Vorgangs. Die *CameraStereo* Klasse soll über das *iCamera* Interface zugegriffen werden können. Das *iCamera* Interface stellt jedoch nur die Funktion *m_glbSetupLookAt()* zur Verfügung um die Kameraposition einzustellen. Die Lösung, um mit einer Funktion beide Positionen darzustellen, ist ein Schalter welcher nach jedem Aufruf der Funktion umgeschaltet wird und dafür sorgt das beim nächsten Aufruf die andere Kameraposition eingestellt wird. Die Funktion *m_glbSetupLookAt()* wird beim Rendervorgang zweimal, für jedes Halbbild einmal, aufgerufen. Im Konstruktor der *CameraStereo* Klasse wird der Schalter so initialisiert, dass als erstes die Kameraposition des linken Halbbildes eingestellt wird.

5.2.3 Hilfesklassen und Strukturen

5.2.3.1 Logdatei

Die Klasse *LogFile* ist für das Schreiben der Anwendungslogfiles verantwortlich. In Abbildung 5.16 ist die Klassenstruktur der *LogFile*-Klasse dargestellt, das im folgenden erläutert wird.

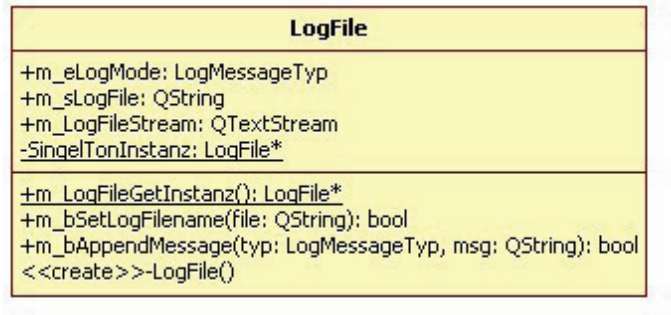

Abbildung 5.16: Klassendiagramm der *LogFile*-Klasse.

Es gibt im Programm nur ein Logfile, welches in jeder Klasse verwendet werden soll. Da also immer die selbe Instanz des Logfiles verwendet werden soll, wurde in der *LogFile* Klasse das *Singelton Muster* [EF08] angewendet. Das Prinzip des Singelton Musters ist es den Klassenkonstruktor privat zu deklarieren und einen Verweis auf eine statische Instanz der Klasse nur über eine extra dafür eingerichtete Methode zurückzugeben. Die statische Instanz der Klasse wird beim ersten Aufruf der Referenz-Methode erstellt und in einem privaten statischen Feld abgelegt. Im Fall der *LogFile* Klasse wird der Verweis auf die statische Instanz im Feld *SingelTonInstanz* vom Typ *LogFile** abgelegt und durch die Methode *m_LogFileGetInstanz()* verwaltet.
Das Schreiben in die Logdatei erfolgt mit Hilfe der Methode *m_bAppendMessage(...)*. Beim Schreiben der Lognachrichten wird zwischen Fehler-, Warnung- und Debuginformationen unterschieden. Der jeweilige Nachrichtentyp und Text der Nachricht wird der *m_bAppendMessage(...)* Methode übergeben. Es findet ein Abgleich des Nachrichtentyps, der Nachricht mit dem Nachrichtentyp zu dem Lognachrichten geschrieben werden sollen, statt. Die Ebene zu der Logeinträge geschrieben werden sollen, wird im Feld *m_eLogMode* gespeichert. Die Hierarchie der Nachrichtentypen lautet wie folgt:

1. Debug
2. Warnung
3. Fehler

Bei der Hierarchie ist zu beachten das hohe Ebenen, am höchsten ist Debug, die darunter liegenden einschließen, es werden zum Beispiel auch Fehlernachrichten in die Logdatei geschrieben wenn als Logebene Debug ausgewählt ist.

5.2.3.2 Knotenliste

Die Klasse *NodesList* ist eine selbst definierte *Listen-Klasse*. In Abbildung 5.17 ist die Klassenstruktur der *NodesList*-Klasse dargestellt, welche im folgenden erläutert wird.

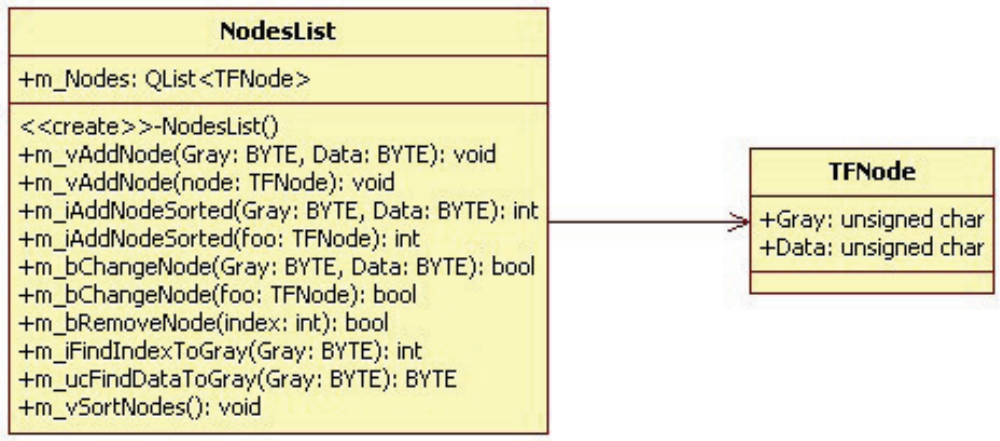

Abbildung 5.17: Klassendiagramm der *NodesList*-Klasse.

Basis der *NodesList*-Klasse ist das Feld *m_Nodes* von Typ *QList*, in dem Objekte von Typ *TFNode* aufgelistet werden. Für die spätere Interpolation der Datenwerte ist es notwendig das die Knoten in sortierter Reihenfolge vorliegen. Die sortierte Reihenfolge bezieht sich auf die Grauwerte der *TFNodes*, diese müssen in aufsteigender Reihenfolge, nach dem Grauwert, in der Liste sortiert sein. Eine Sortierung der Liste wird von den Methode *m_vSortNodes()* durch geführt, es wird der *Bubble-Sort Algorithmus* [Lou02] verwendet. Die *NodesList*-Klasse implementiert desweiteren Methoden zur Verwaltung der Knotenliste. Um die Konten zu verwalten, stehen Methoden zum Hinzufügen, Ändern und Löschen zur Verfügung.

5.3 Spezielle Problemstellungen

In diesem Abschnitt werden spezielle Lösungsansätze, die im Programm umgesetzt wurden im Detail beschrieben.

5.3.1 Umsetzung der Farb-Anaglyphestereoprojektion

In diesem Abschnitt werden die einzelnen Schritte der Farb-Anaglypenstereoprojektion, erläutert, es wird gezeigt an, welchen Stellen in der Software diese abgearbeitet werden. Die Farb-Anaglypenstereoprojektion lässt sich in folgende Schritte unterteilt:

1. Erzeugen der Halbbilder
2. Farbfilterung
3. Mischen der Halbbilder

Im Programm wird diese Schrittkette etwas abgewandelt angewendet. Da es nicht möglich ist beide Halbbilder gleichzeitig zu erzeugen, werden sie nacheinander gerendert und gefiltert. Der genaue Ablauf wird im Folgenden mit Codeauszügen beschrieben.
Ein Bestandteil der Farbfilterung ist die Grauwertbildung, diese wird im Programm direkt auf der Grafikkarte durch geführt, nämlich vom Fragment-Shader. Da im Programm ohnehin Shader zur Volumendarstellung verwendet werden, brauchen diese nur modifiziert zu werden. In Codelisting 5.8 sind die Zeilen des Fragment-Shaders zu sehen die für die Grauwertbildung der Farb-Anaglyphenprojektion verantwortlich sind.

Listing 5.8: Codeauszug aus dem Fragment-Shader zur Farb-Anaglyphenprojketion.

```
1 ...
2 float gray = (vecres.r + vecres.g + vecres.b)/3.0;
3 vecres.r = gray;
4 vecres.g = gray;
5 vecres.b = gray;
6 ...
```

Als nächstes wird eine initiale Löschung des Farb- und Tiefenbuffer [Shr09] durchgeführt. Der nächste Schritt ist das Einstellen der Kamera zum Erzeugen des linken Halbbilders. Zum Kamerasetup werden die Methoden *m_glbSetupFrustum()* und *m_glbSetupLookAt()* der Stereokamera aufgerufen. Wie im Abschnitt 5.2.2.1 zur Klasse *CameraStereo* erwähnt, wird die Kameraposition nach jedem Aufruf von *m_glbSetupLookAt()* intern gewechselt. Das linke Halbbild soll nur rote Farbkomponenten enthalten. Zum Ausblenden der unerwünschten Farbkomponenten wird die OpenGL Funktion *glColorMask(...)* verwendet. Alle Vorbereitungen zum Zeichnen der Geometrie sind nun abgeschlossen, die Methode *m_bRenderVolume()*, die das Volumen zeichnet wird aufgerufen. Das linke Halbbild des Farb-Anaglyphenstereobildes liegt nun im Buffer der Grafikkarte. In Codelisting 5.9 ist zusehen wie dies im Programm umgesetzt wurde.

Listing 5.9: Codeauszug aus der *paintGL()* Methode. Dieser Ausschnitt erzeugt das linke Halbbild.

```
1  glDrawBuffer(GL_BACK);
2  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
3  glMatrixMode(GL_PROJECTION);
4  glLoadIdentity();
5
6  m_CameraStereo->m_glbSetupFrustum();
7  glMatrixMode(GL_MODELVIEW);
8  glLoadIdentity();
9  m_CameraStereo->m_glbSetupLookAt();
10
11 glPushMatrix();
12 // objekte rendern
13 glColorMask(GL_TRUE, GL_FALSE, GL_FALSE, GL_TRUE);
14 m_Volume->m_bRenderVolume();
15 glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
16 glPopMatrix();
17 glFlush();
```

Um das zweite, rechte, Halbbild zu erzeugen muss zunächst die Kameraposition gewechselt werden, dies erfolgt analog zum Kamerasetup des linken Halbbildes. Der ausschlaggebende Unterschied ist das beim zweiten Halbbild nur die Grüne- und Blauefarbkomponenten dargestellt werden, dies erfolgt wieder durch den Aufruf der OpenGL Funktion *glColorMask(...)* mit den entsprechenden Parametern. Das rechte Halbbild ist somit auch erzeugt. Der letzte Schritt ist die Mischung beider Halbbilder. In Codelisting 5.10 ist zu sehen wie dies im Programm umgesetzt wurde.

Listing 5.10: Codeauszug aus der *paintGL()* Methode. Dieser Ausschnitt erzeugt das rechte Halbbild.

```
1  glDrawBuffer(GL_BACK);
2  glMatrixMode(GL_PROJECTION);
3  glLoadIdentity();
4
5  m_CameraStereo->m_glbSetupFrustum();
6  glMatrixMode(GL_MODELVIEW);
7  glLoadIdentity();
8  m_CameraStereo->m_glbSetupLookAt();
9  glPushMatrix();
10 // objekte rendern
11 glColorMask(GL_FALSE, GL_TRUE, GL_TRUE, GL_TRUE);
12 m_Volume->m_bRenderVolume();
13 glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
14 glPopMatrix();
15 glFlush();
```

Da zwischen den beiden Renderprozessen der Farbbuffer nicht zurück gesetzt wird, werden beiden Bilder in den gleichen Farbbuffer gerendert, dies hat zur Folge das beide Halbbilder auf akkumuliert werden wie gefordert. Das fertige Farb-Anaglyphestereobild liegt nun im Hintergrundbuffer der Grafikkarte. Der Hintergrundbuffer wird beim Verlassen der *paintGL()* Methode in den Vordergrund gebracht.

5.3.2 Shader zur Volumendarstellung

In diesem Abschnitt sollen die Shader die im Programm für die Volumendarstellung verantwortlich sind im Detail betrachtet werden. Die grundlegenden Komponenten, das Blinn-Phong Beleuchtungsmodel sowie die Umsetzung der Transferfunktion, stammen aus [Had06]. Der Basiscode für die Shader liegt in der Literatur [Had06] in der allgemeinen Shadersprache *CG* [NVI] vor. OpenGL besitzt eine eigene Shadersprache, *GLSL* [RJR10], in die der CG-Code vor der Verwendung portiert werden musste.
Wie in der Bearbeitungsfolge der Grafikkarte, soll mit dem *Vertex-Shader* begonnen werden. Aufgabe des Vextex-Shaders im Programm ist es die Position des Fragments zu berechnen und die Texturkoordinaten an den *Fragment-Shader* zu übergeben. Eine Begriffserklärung zu Vertex- und Fragmentshader ist in [RJR10] zu finden. In Codelisting 5.11 ist zu sehen wie der Vertex-Shader, in GLSL-Code umgesetzt aussieht.

Listing 5.11: Der im Programm verwendete Vertex-Shader.

```
1 uniform sampler3D VolumeTexture;
2 uniform sampler1D ColorTable;
3 void main()
4 {
5     gl_Position = ftransform();
6     gl_FrontColor = gl_Color;
7     gl_TexCoord[0] = gl_MultiTexCoord0;
8     gl_TexCoord[1] = gl_MultiTexCoord1;
9 }
```

Der Fragment-Prozessor beschäftigt sich mit der Berechnung der Farbe eines Fragments, aus diesem Grund sind die Aufgaben des Fragment-Shaders im Programm umfangreicher und rechenintensiver wie die des Vertex-Shaders. Die erste Aufgabe des Fragment-Shaders ist es, abhängig von der Koordinate in der Datentextur und dem daraus resultierenden Grauwert, die Zuordnung vom Grauwert zur Transferfunktion durchzuführen. Nach der Zuordnung des Grauwertes, muss die Beleuchtung die auf des Fragment fällt berechnet werden. Wie Eingangs erwähnt wird das Blinn-Phong Beleuchtungsmodel 3.2.2.2 verwendet. Die Berechnung der Beleuchtung nach Blinn-Phong ist in die Funktion *Blinn-Phong(...)* gekapselt. In Codelisting 5.12 ist zu sehen wie der Fragment-Shader, umgesetzt wurde.

Listing 5.12: Der im Programm verwendete Fragement-Shader. Ohne Grauwertbildung für die Farb-Anaglypheprojektion.

```
1  uniform sampler3D VolumeTexture;
2  uniform sampler1D ColorTable;
3  uniform vec3 PosLicht;
4  uniform vec3 PosCamera;
5
6  vec3 BlinnPhong(vec3 N, vec3 V, vec3 L)
7  {
8      vec3 Ka = vec3(0.1, 0.1, 0.1);
9      vec3 Kd = vec3(0.6, 0.6, 0.6);
10     vec3 Ks = vec3(0.2, 0.2, 0.2);
11     float n = 100.0;
12     vec3 LightColor = vec3(1.0, 1.0, 1.0);
```

```
13
14      vec3 LightColorAmbient = vec3(0.3, 0.3, 0.3);
15      vec3 H = normalize(L+V);
16      vec3 AmbientTeil = Ka * LightColorAmbient;
17
18      float DiffuseLicht = max(dot(L,N), 0.0);
19      vec3 DiffuseTeil = Kd * LightColor * DiffuseLicht;
20
21      float SpecularLicht = pow(max(dot(H,N), 0.0), n);
22      if(DiffuseLicht <= 0.0) SpecularLicht = 0.0;
23      vec3 SpecularTeil = Ks * LightColor * SpecularLicht;
24
25      return AmbientTeil + DiffuseTeil + SpecularTeil;
26  }
27  void main()
28  {
29      vec4 vecpos = texture3D(VolumeTexture, gl_TexCoord[0].stp);
30      float gray = (vecpos.r + vecpos.g + vecpos.b)/3.0;
31      vec4 vecres = texture1D(ColorTable, gray);
32
33      vec3 N = normalize(2.0*vecres.xyz - 1..xxx);
34      vec3 L = normalize(PosLicht - gl_TexCoord[0].stp);
35      vec3 V = normalize(PosCamera - gl_TexCoord[0].stp);
36      vec3 IlluminationTeil = BlinnPhong(N,V,L);
37      vecres.x += IlluminationTeil.x;
38      vecres.y += IlluminationTeil.y;
39      vecres.z += IlluminationTeil.z;
40
41      gl_FragColor = vecres;
42  };
```

Schnittstelle von den Grafikprozessoren zum Programm sind die globalen *uniform*-Variablen. Diesen werden in der *paintGL()* Methode, aus der Klasse *OpenGLWidget*, Werte zugewiesen. In Codelisting 5.13 ist zu sehen wie die *uniform*-Variablen der Shader mit Werten versorgt werden.

Listing 5.13: Wertzuweisung für die in der Shadern verwendeten *uniform*-Variablen.

```
1  m_ShaderProgramm->setUniformValue("ColorTable", 1);
2  m_ShaderProgramm->setUniformValue("VolumeTexture", 0);
3  m_ShaderProgramm->setUniformValue("PosLicht", QVector3D(0.0, 2.0, 0.0));
4  m_ShaderProgramm->setUniformValue("PosCamera", m_Camera->m_GetPositionPoint());
```

5.3.3 Bestimmen der Blickrichtung auf das Volumen

Hintergrund warum es notwendig ist, die Blickrichtung auf das Volumen zu berechnen, ist die Berechnung der Transparenz in OpenGL. Um eine korrekte Transparenzdarstellung in OpenGL zu erhalten, müssen die zu rendernden Primitiven, in diesem Fall die Slices, in sortierter Reihenfolge gezeichnet werden. Die Primitiven die am weitesten weg von der Position der Kamera sind, müssen als erstes gezeichnet werden, da diese als Hintergrund vorhanden sein müssen, wenn im Vordergrund eine neue Primitive gezeichnet werden.
Die Slices durch die 3D-Datentextur werden immer in Würfelform gezeichnet. Der Lösungsansatz ist, zu prüfen welche Seiten-Ebene des Würfels als erstes vom Kameravektor

geschnitten wird. Als Kameravektor wird der Vektor bezeichnet der von der Position der Kamera zum Zielpunkt der Kamera verläuft. Als mathematischer Ansatz dient das Verfahren „Schnittpunkt und Schnittwinkel einer Gerade mit einer Ebene“ aus [Pap01].

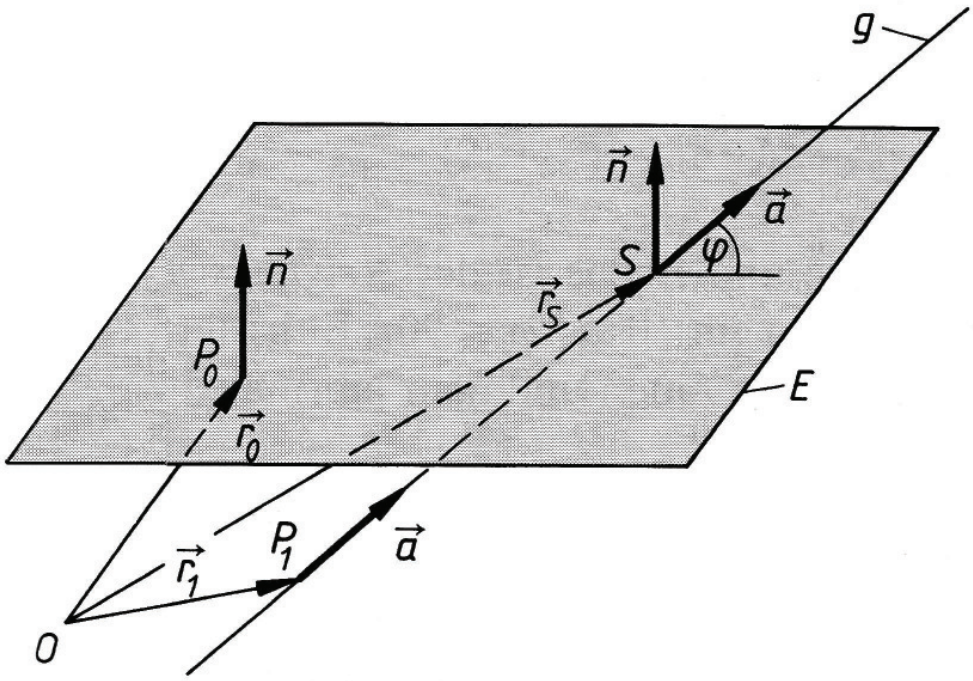

Abbildung 5.18: Skizze zur Berechnung des Schnittpunktes einer Geraden mit einer Ebene. (Quelle: [Pap01]

Im Ansatz ist die Ebene E und die Geradengleichung $\vec{r}(\lambda)$ gegeben.

$$E = \vec{n}(\vec{r} - \vec{r}_0) \tag{5.1}$$

$$\vec{r}(\lambda) = \vec{r}_1 + \lambda\vec{a} \tag{5.2}$$

Aus ihr resultiert der gerichtete Ortsvektor $\vec{r}_s$, wobei $\vec{r}$ nicht parallel zur Ebene verlaufen darf, das heisst $\vec{n}\vec{a} \neq 0$.

$$\vec{r}_s = \vec{r}_1 + \left(\frac{\vec{n}(\vec{r}_0 - \vec{r}_1)}{\vec{n}\vec{a}}\right)\vec{a} = \vec{r}_1 + \lambda_s\vec{a} \tag{5.3}$$

Der Faktor λ_s gibt hierbei Auskunft darüber nach welcher Entfernung die Ebene vom Vektor $\vec{r}(\lambda)$ geschnitten wird. Ist der Wert positiv schneidet der Vektor die Ebene in positiver Vektorrichtung, ist er negativ schneidet er sie „rückwärts“. Um festzustellen welche

Seite des Volumenwürfels zuerst vom Kameravektor geschnitten wird, müssen also sechs λ_s Faktoren, einer für jede Ebene des Würfels, bestimmt werden. In Codelisting 5.14 ist die implementierte Methode zur Bestimmung des λ_s-Faktors zu sehen.

Listing 5.14: Mehtode zur Berechnung des λ_s-Faktors.

```
1  double Volume::m_bGeradeSchneidetEbene(QVector3D ebenep1, QVector3D ebenep2,
2  QVector3D ebenep3, QVector3D geradepunkt, QVector3D geraderichtungsvector)
3  {
4      QVector3D n;
5      n = QVector3D::crossProduct((ebenep2-ebenep1), (ebenep3-ebenep1));
6      if((n*geraderichtungsvector).isNull() == true) return -1;
7      double zaehler = QVector3D::dotProduct(n,ebenep1-geradepunkt);
8      double nenner = QVector3D::dotProduct(n, geraderichtungsvector);
9
10     // plane and vektor are parallel, return -1 so faktor doesnt matter anymore!
11     if(nenner == 0.0) return -1;
12     return (zaehler/nenner);
13 }
```

Drei von sechs berechneten λ_s, fallen durch ihr negatives Vorzeichen weg. Aus den drei verbliebenen λ_s muss der kleinste Faktor gewählt werden, da die zum ihm gehörende Gerade die Ebene als erstes schneidet. Welche Ebene als erstes geschnitten wird ist also bekannt, nun müssen die Slices nur noch entsprechend ausgerichtet werden. Wird zum Beispiel die vordere Ebene des Würfels zuerst geschnitten, müssen die Slices aus Richtung +Z nach +Z gezeichnet werden. In Abbildung 5.19 ist das System des Würfels dargestellt, die rot umrandete Fläche ist vordere Ebene.

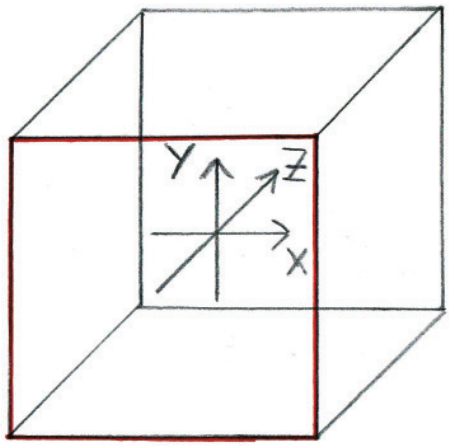

Abbildung 5.19: Koordinatensystem des Würfels.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde eine Software entwickelt, mit der es möglich ist Volumendaten stereoskopisch darzustellen.

Um das Ziel dieser Arbeit zu erreichen, musste Grundlagenwissen auf diversen Themengebieten erarbeitet werden. Ein Überblick über die behandelten Grundlagen wurde im Kapitel 3 gegeben. Die Softwareanforderungen wurden erfasst und in Kapitel 2 festgeschrieben. Ein Softwareentwurf wurde nach den gestellten Anforderungen angefertigt. Der geplante Entwurf wurde, mit Erweiterungen der Klassenstruktur, in eine Software umgesetzt. Bei der Implementierung traten verschiedene Probleme auf, deren Lösungen in Kapitel 5.3 beschrieben wurden.

Mit dem entstandenen Programm können Bildstapel von einem Datenträger geladen und in Form einer Volumengrafik wahlweise in normaler oder stereoskopischer Ansicht betrachtet werden. Es besteht die Möglichkeit Regionen des dargestellten Volumens mit Hilfe einer eindimensionalen Transferfunktion ein- oder auszublenden. Zur Darstellung wird das Volumen von Slices geschnitten, die die Volumendaten sichtbar machen. Es wurden Shader-Programme umgesetzt mit deren Hilfe die Volumendarstellung zum Teil auf den Grafikprozessoren abläuft.

Es ist geplant die Software in Zukunft weiter zu entwickeln und zu verbessert. Die Transferfunktion sollte zur einfacheren Bedienung durch ein grafisches Benutzerinterface ergänzt werden. Da das Slicing-Verfahren sichtbare Aliasing-Artefakt in der Darstellung hervorruft, sollte es zum Beispiel durch das Grafikprozessor basierende Raytracing-Verfahren [Had06] ersetzt werden. Eine Möglichkeit die eingestellte Transferfunktion zu speichern bzw. zu laden wäre sehr sinnvoll um Konfigurationszeiten während einer Präsentation zu sparen. Das Laden eines Bildstapels erfordert einen relativ großen Zeit und Speicheraufwand, eine Speichermöglichkeit für eine erzeugte Datentextur könnte den Ladeaufwand verkürzen. Da die dreidimensionalen Datentexturen einen enormen Speicherbedarf haben, $512^3 Byte \approx 134 MByte$, wäre es sinnvoll eine Datenkompression zum Beispiel durch Wavlets [KDK06] durchzuführen. Um die Präsentation mit der Software zu erleichtern, wäre eine Funktionalität um einen Kamera-Bewegungsablauf zu speichern und abzuspielen eine nützliche Programmerweiterung.

Literaturverzeichnis

[Cen] Center, Computational V.: *Projekt VolumeRover*. http://cvcweb.ices.utexas.edu/ccv/projects/project.php?proID=9, Abruf: 09.11.2009

[Cha08] Charles, Oliver: *Einführung und Überblick: OpenGL*. GRIN Verlag, 2008

[Cora] Corp., Microsoft: *DirectX*. www.microsoft.com/windows/directx, Abruf: 22.05.2010

[Corb] Corporation, Nokia: *Qt*. http://qt.nokia.com/products, Abruf: 19.05.2010

[EF08] Eric Freeman, Kathy Sierra Bert B. Elisabeth Freeman F. Elisabeth Freeman: *Entwurfsmuster von Kopf bis Fuß*. 4. O'Reilly, 2008

[EH02] Ekbert Hering, Martin S. Rolf Martin M. Rolf Martin: *Physik für Ingenieure*. 8. Springer, 2002

[Gro] Group, Khronos: *OpenGL*. http://www.opengl.org, Abruf: 22.05.2010

[Göt] Göttingen, Max-Plank-Institut D.: *Max-Plank-Institut für Dynamik und Selbstorganisation*. http://www.ds.mpg.de/Forschung/index.php, Abruf: 09.11.2009

[Had06] Hadwiger, Markus: *Real-time Volume Graphics*. 1. AK Peters, 2006

[JB09] Jasmin Blanchette, Marksummerfield: *C++ GUI Programmierung mit Qt 4*. 2. Addison-Wesley, 2009

[KDK06] K. D. Kammeyer, K. K.: *Digitale Signalverarbeitung*. 6. B. G. Teubner Verlag, 2006

[Kuh99] Kuhn, Gerhard: *Stereo-Fotografie und Raumbild-Projektion*. VFV Verlag, 1999

[LFK83] L.A. Feldkamp, L.C. D. ; Kress, J.W.: *Partial cone-beam algorithm*. Version: 1983. http://www.opticsinfobase.org/DirectPDFAccess/DDB81450-BDB9-137E-C96C927A49E3D5E2_996.pdf, Abruf: 10.11.2009

[Lou02] Louis, Dirk: *C/C++ Professionell programmieren mit aktuellen Standards*. 1. Mark+Technik Verlag, 2002

[Luh03] Luhmann, Thomas: *Nahbereichsphotogrammetrie - Grundlagen, Methoden und Anwendungen*. 2. Herbert Wichtmann Verlag, 2003

[Mue09] Mues, Pascal: *Projektvorbereitung zur Modifikation an der Software VolumeRover*. 2009

[NVI] NVIDIA: *CG Shader*. http://developer.nvidia.com/page/cg_main.html, urldate={25.05.2010}, Abruf: 25.05.2010

[Pap01] Papula, Lothar: *Mathematik für Ingenieure und Naturwissenschaftlar - Band 1*. 10. Vieweg, 2001

[Röd07] Röder, Oliver: *Grundlagen der Stereoskopie*. VDM Verlag Dr. Müller, 2007

[RJR10] Randi J. Rost, Bill Licea-Kane: *Open GL Shading Language(Orangebook)*. 3. Addison-Wesley, 2010

[Sch06] Schwerfgen, David: *3D Spiele Programmierung mit DirectX 9 und C++*. Hanser, 2006

[Shr09] Shreiner, Dave: *Open GL Programming Guide - The Offical Guide to Learning OpenGL, Versions 3.0 and 3.1(Redbook)*. 7. Addison-Wesley, 2009

[Wol09] Wolf, Jürgen: *Qt 4 GUI-Entwicklung mit C++*. 1. Galileo Press, 2009

Anhang

A.1 Kosten Farb- und Polfilterbrillen

Die Preise wurden den beistehenden Internetseite am 18.09.2009 entnommen.

	Internetseite	**Preis**
1	http://s165588531.e-shop.info	1.25 €
2	http://www.perspektrum.de/3d-brillen.htm	0.85 €
3	http://www.ozhobbies.eu/gadgets	1.20 €
	Mittelwert	1.10 €

Tabelle A.1: Preise für Rot/Cyan-Farbfilterbrillen aus Karton

	Internetseite	**Preis**
1	http://www.3d-foto-shop.de	2.60 €
2	http://www.perspektrum.de/3d-brillen.htm	2.50 €
3	http://www.webmart.de	2.38 €
	Mittelwert	2.49 €

Tabelle A.2: Preise für Zirkularpolarisationsbrillen aus Karton

A.2 Stereoskopische Testaufnahme

Diese Testaufnahme wurde durchgeführt um den Unterschied zwischen der Normalaufnahme und der Nahaufnahme in der Praxis zu testen. Bei der aufgenommen Szene handelt es sich um einen Büroraum am Max Plank Institut. Abbildung A.1 enthält zum Vergleich den maßstäblichen Grundriß des Raumes.

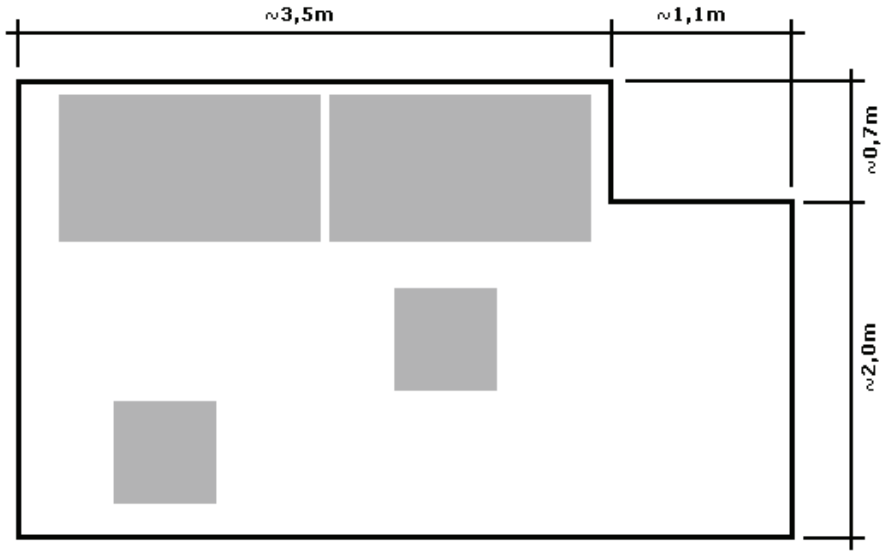

Abbildung A.1: Skizze des Grundrisses des aufgenommenen Büroraums.

Der Unterschied der beiden Aufnahmen liegt darin, dass bei der Nahaufnahme der Fernpunkt auf die Tafel gelegt wurde und bei der Normalaufnahme im Unendlichen. Hier müssen für beide Konfigurationen die Stereobasen bestimmt werden, für die Normalaufnahme gilt:

$$S = v \cdot \left(\frac{s}{f} - 1\right) \approx 7cm \tag{A.1}$$

und für die Nahaufnahme:

$$S = \frac{v \cdot n - f}{f \cdot \left(1 - \frac{n}{w}\right)} \approx 7cm \tag{A.2}$$

Löst man die Gleichungen auf, erhält man nahezu die gleichen Werte für die Stereobasen. Aus diesem Grund ist der ausschlag gebende Unterschied bei der Aufnahme die lage des Fixpunktes.
Die Aufnahme wurde mit einer Kamera durchgeführt die auf einem Dreibeinstativ befestigt war. Das Stativ wurde an einem am Boden befestigten Richtstab um die Stereobasen verschoben, gemessen wurde mit einem Zollstock. Hierbei war es sehr schwierig die Kamera nur mit dem Sucher auf das gewünschte Ziel zu fixieren, um die Blickachse

genau parallel zu verschieben oder immer den gleichen Punkt zu fixieren.
Abbildung A.2 zeigt die Aufnahme in der der Fixpunkt auf der Tafel liegt und in Abbildung A.3 liegt der Fixpunkt im Unendlichen. Die Filterung und das Mischen der Bilder wurde mit einem selbst entwickelten Tool durchgeführt. Mit der Entwicklung des Tools konnte das erworbene Grundwissen zur Stereoskopie in Code umgesetzt werden.
Bei der Betrachtung der Bilder fällt auf, das der Raum unterschiedliche tief wirkt. Die Raumtiefe bei der Normalaufnahme wird wesentlich intensiver wargenommen.

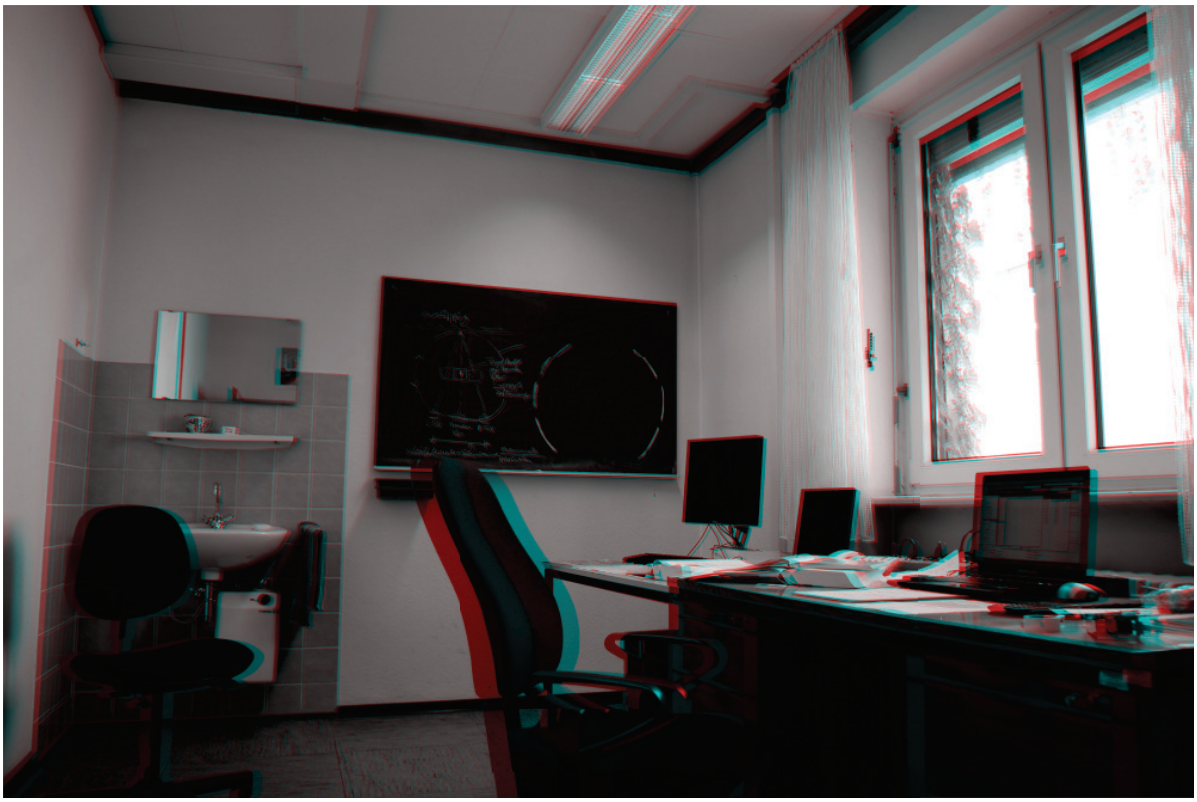

Abbildung A.2: Aufgenommen mit einer Stereobasis von 7cm. Der Fixpunkt liegt auf Tafel die im Bild zu sehen ist.

Abbildung A.3: Aufgenommen mit einer Stereobasis von 7cm. Die Blickachse der Kamera wurde um die Stereobasis parallel verschoben, der Fixpunkt liegt also im Unendlichen.

A.3 CD zum Inhalt

Dieser Bachelorarbeit liegt eine CD bei. Auf dieser CD befinden sich die Quelltexte des entwickelten Programms, die Bachelorarbeit in PDF-Format, alle Bilder die bei der Stereotestaufnahme entstanden sind sowie das Tool mit dem die Farb-Anaglyphen Bilder erstellt wurden und das Programm zum Test der OpenGL Funktionalität unter Qt.

A.4 Klassenübersicht des Programms

Da das Klassendiagramm mit einem externen Programm erstellt wurde und nicht auf A4 Format darstellbar ist, liegt dies auf der nächsten Seite vor.