Humboldt-Universität zu Berlin

Institut für Informatik

Integration von 3D-Algorithmen der C++-Klassenbibliothek VTK in eine medizinische Java-Visualisierungsbibliothek (viskit) für die dreidimensionale Visualisierung von DICOM-Bildern

Studienarbeit von Oliver Sanftleben

Dezember 2014

Betreuer: Prof. Eisert

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Aufgabenstellung

1.1 Thema

Integration von 3D-Algorithmen der C++-Klassenbibliothek VTK in eine medizinische Java-Visualisierungsbibliothek (viskit) für die Visualisierung von DICOM-Bildern.

1.2 Zielstellung

Good Image ist eine modulare Software der Firma sofd GmbH, die sämtliche Prozesse zur Vorbereitung (preparation) und Durchführung (conduct) von Blinded Readings unterstützt. Blinded Readings stellen eine Standard-Auswertungsmethode bei klinischen Prüfungen mit Daten aus bildgebenden Verfahren dar.

Die radiologischen Bilder werden von an den klinischen Studien partizipierenden Kliniken erstellt und den Pharmaunternehmen bereitgestellt. Die Software importiert das Bildmaterial und unterstützt die Mitarbeiter des Pharmaunternehmens bei der detaillierten visuellen Qualitätskontrolle. Diese umfasst die komplette Sichtung der Bilder. Die Mitarbeiter dokumentieren die Ergebnisse der Qualitätssicherung unter Verwendung von Good Image und stellen das Bildmaterial über eine generische Schnittstelle für die Durchführung von Blinded Readings bereit. Bei einem Blinded Reading werden die qualitätsgesicherten Bilder anonymisiert und von unabhängigen Experten gesichtet. Die Experten können dabei Operationen auf dem Bildmaterial vornehmen, wie z. B. Annotationen und Messungen, und Befunde in elektronischen Formularen (eCRFs) eingeben.

Die Firma sofd GmbH hat eine Java-Bibliothek (viskit) speziell zur Visualisierung medizinischer Bilder im DICOM-Format entwickelt, die bisher auf zweidimensionale Darstellung und Verarbeitung beschränkt ist. Diese soll nun derart erweitert werden, dass aus 2D-Bilderserien die aufgenommenen Körper und darin enthaltene Strukturen dreidimensional rekonstruiert werden.

Im Rahmen der im vorliegenden Dokument beschriebenen Studienarbeit soll analysiert werden, welche Algorithmen hierfür geeignet sind und wie diese mit Hilfe der Open-Source-Bibliothek VTK in viskit integriert werden können. Eine große Herausforderung wird dabei sein, wie man VTK in viskit anbindet, d.h. wie man eine Brücke von der C- zur Java-Welt bauen kann.

Dabei sind folgende Fragen zu klären :

- Was ist VTK, welche Möglichkeiten und welche Vor- und Nachteile bietet diese Bibliothek?
- Was ist viskit und was kann viskit?
- Was ist DICOM?
- Der viskit I/O Reader erzeugt eine Liste von 2D dcm-Objekten (JAVA), die als Input für die VTK-Visualisierungsalgorithmen dienen soll. Wie können diese dcm-Objekte in die interne VTK-Struktur umgewandelt werden?
- Wie können die 2D-Schichten anhand von geeigneten VTK Visualisierungsalgorithmen in 3D visualisiert werden?
- Gibt es Alternativen zu VTK, die besser geeignet sind? (OpenGL, JOGL)

1.3 Aufgaben

(a) Identifikation der geeigneten 3D-Algorithmen aus VTK, die die Rekonstruktion aus 2D in 3D vollziehen (d.h. 3D-Objekte aus 2D Schichten rekonstruieren)

(b) Entwicklung einer Benutzeroberfläche zur Steuerung der Parameter der 3D-Visualisierungsalgorithmen

(c) Evaluation der Ergebnisse anhand einer Testanwendung (Was könnte eine sinnvolle Testanwendung sein?)

(d) Suche nach Alternativen zu VTK

Kapitel 2

Überblick

Im nachfolgenden Kapitel *Grundlagen* werde ich zunächst Grundbegriffe und Techniken erläutern, die für die Bewältigung der Aufgabenstellung von Bedeutung sind. Es werden die beiden wichtigsten bildgebenden Verfahren, CT und MRT, vorgestellt, die von viskit unterstützt werden und dazu das Datenformat DICOM erläutert. Das Kapitel *3D-Rekonstruktionen* geht auf die beiden verbreitetsten Verfahren zur dreidimensionalen Rekonstruktion von Bilddatensätzen ein. Die darauf aufbauenden eigenen Implementierungen, sowohl mit Hilfe von VTK als auch mit JOGL, werde ich im Kapitel •Implementierung vorstellen. Anschließend wird von mir ausgewertet, welche Techniken sich am besten für die Erledigung der Aufgabenstellung geeignet haben, welche Vor- und Nachteile diese besitzen und welche Aussichten und Verbesserungsvorschläge sich im Laufe der Studienarbeit ergeben haben.

Kapitel 3

Grundlagen

3.1 Medizinische Bilder

Das vorliegende Bildmaterial für die geplante *3D-Rekonstruktion* umfasst hauptsächlich innere Aufnahmen eines Menschen. Je nach eingesetzter Technik handelt es sich um *tomographische* Aufnahmen des Skelettes, innerer Organe (Gehirn, Lunge, Leber), von innerem Gewebe oder Arterien. Die Bilder liegen in digitalisierter Form im *DICOM-Format* vor und beinhalten neben den reinen Messdaten auch Metadaten, wie z.B. geräteabhängige Parameterwerte. Pro Aufnahme (Scan) entsteht eine Sequenz von Bildern, die auch **Bildserie** genannt wird.

Ziel der Rekonstruktion ist die Erstellung eines 3D-Modells für den gescannten Körper oder des Organs aus der Bildserie.

Bei den zur Verfügung stehenden Bildern handelt es sich überwiegend um Bilder aus der *Computertomographie (CT)* oder *Magnetresonanztomographie (MRT)*.

3.1.1 CT

Im Gegensatz zu klassischen Röntgenbildern ist bei der ***Computertomographie*** eine genaue räumliche Differenzierung von Weichteilen, wie z.B. inneren Organen, Gewebe usw. möglich.

Bei der Computertomographie werden Röntgenstrahlen von einer frei beweglichen Quelle durch ein Objekt geschossen und auf der gegenüberliegenden Seite von einem Detektor aufgefangen und gemessen.[1,2] Röntgenquelle und Detektor können sich dabei 360° um das Objekt entlang einer Ebene bewegen und dieses umkreisen. Je nach Absorptionseigenschaft des Objektmaterials, die von der Dichte abhängt, werden vom Detektor unterschiedliche Restintensitäten gemessen. Aus den Messwerten der verschiedenen

[1] Lipinski: *Einführung in die medizintechnische Informatik*, 1999, S.252-261.
[2] Castleman: *Digital Image Processing*, 1996, S.582-585.

Stellungen des Detektors ist es wiederum möglich, die Absorptionswerte einzelner Punkte innerhalb des Zielobjektes zu bestimmen und diese als Bildmatrix in einem Computer zu betrachten und abzuspeichern. Um größere Objekte, wie z.B. den menschlichen Körper zu messen, müssen mehrere solcher Schnittbilder angefertigt werden, der Patient wird dabei schrittweise durch eine Messröhre geschoben.

Um eine höhere Bildauflösung zu erzielen, ist es meistens notwendig nicht nur einzelne, sondern gleich mehrere Röntgenstrahlen parallel oder fächerförmig durch das Objekt zu schießen.

Eine besondere Rolle bei CT-Bildern spielen die sog. ***Hounsfield-Einheiten***, mit deren Hilfe CT-Bilder aus unterschiedlichen Modalitäten (CT-Scannern) und Aufnahmeverfahren miteinander verglichen werden können. Dabei werden die gemessenen Absorptionswerte relativ zum Absorptionswert von Wasser berechnet.

Die genaue Umrechnungsformel für *Hounsfield-Units (HU)* lautet :

$$\mu_{rel} = k \cdot \frac{\mu_{Obj} - \mu_W}{\mu_W} \tag{3.1}$$

(μ_{Obj}, μ_W - Absorptionskoeffizienten für Objekt und Wasser, k - Konstante, μ_{rel} - Absorptionswert in Houndsfield-Einheiten)

Die Konstante k wird häufig auf 1000 gesetzt. Damit erhält man einen Wertebereich von -1000 (Luft) bis $+2000$ (Knochen), der sich gut in einem 12-Bit-Format abspeichern lässt (maximal 4096 Werte).[3]

Folgende Tabelle listet die HUs für verschiedene Materialien auf (mit $k = 1000$) :

HU	**Material**
-1000	Luft
-200 - 0	Lungen- und Fettgewebe
0	Wasser
50 - 250	Haut, innere Organe, Muskeln
500 - 2000	Knochen

Tabelle 3.1: Houndsfield-Einheiten für unterschiedliche Materialien

3.1.2 MRT

MRT ist die Abkürzung für Magnet-Resonanz-Tomographie und wird an manchen Stellen auch als Kernspin-Resonanz-Tomographie bezeichnet.[4] Im Gegensatz zu CT wird bei diesem Verfahren keine für den Menschen gefährliche ionisierende Strahlung eingesetzt.

[3] Materialien wie etwa Metall haben einen noch höheren Hounsfield-Wert, so können z.B. auch Einschlüsse von Fremdkörpern abgespeichert werden.

[4] Lipinski: *Einführung in die medizintechnische Informatik*, 1999, S.261-263.

MRT beruht auf dem Prinzip der kernmagnetischen Resonanz. Atome mit ungerader Ladungszahl[5] werden elektromagnetisch angeregt und es wird gemessen, nach wie viel Zeit (sog. *Relaxationszeit*) sich das induzierte Magnetfeld des Atomkerns wieder abbaut.[6] Jedes Gewebe hat dabei eine bestimmte spezifische Relaxationzeit, mit der die Gewebearten im Körper unterschieden werden können. Ähnlich wie bei CT kann die Gewebedifferenzierung durch Zugabe von *Kontrastmitteln* verbessert werden.

3.2 DICOM

DICOM ist ein Standard für die elektronische Bildkommunikation in der Medizin und definiert ein Standardformat für die Speicherung medizinischer Bild- und zugehöriger Metadaten. Der Standard wurde 1985 von einer Arbeitsgruppe des *ACR* und der *NEMA* gemeinsam entwickelt und wird heute von vielen bildgebenden Modalitäten weitgehend unterstützt.[7,8]

DICOM ist sehr umfangreich aber auch komplex, sodass in der Praxis meist nur eine Teilmenge des Standards unterstützt wird. Z.B. sind bei den Bildformaten komprimierte und unkomprimierte Bilddaten, sowie verlustbehaftete und verlustlose Kompression vorgesehen, die jedoch nur von wenigen *DICOM-Readern* komplett unterstützt werden.

Neben den reinen Bilddaten werden pro Bild zugehörige Metadaten wie z.B. Bildgröße, Bildformat, *Schichtabstand* (bei *3D-Scans*), Aufnahmezeit, aber auch Patientendaten wie Alter und Geschlecht abgespeichert. Für CT- und MRT gehören dazu auch spezielle Geräteparameter, z.B. die Feldstärke bei MRT oder die Einheiten und die Umrechnungswerte für die Darstellung auf dem Monitor[9].

3.3 VISKIT

VISKIT (VISualisation toolKIT) ist eine Java-Bibliothek zur Visualisierung medizinischer Bilder im DICOM-Format, die zur Zeit von der Firma sofd GmbH entwickelt wird.

Eingesetzt wird Viskit speziell bei der Kontrolle und Auswertung medizinischer Studien sowie bei der Durchführung von „*Blinded Readings*“[10] Viskit nutzt die Bibliothek *dcm4che* zum Laden und Speichern von *DICOM-Dateien* und umfasst verschiedene Routinen zur Anzeige, Messung und zum Vergleich medizinischer Bildserien. Zu den bisherigen Features zählt u.A. auch die Gruppierung ($m \times n$) von Bildserien, *Windowing* und das Zeichnen von *ROIs*.

Zukünftig soll in Viskit die Anzeige mit Hilfe von *OpenGL* beschleunigt werden. Meine Aufgabe im Rahmen der Studienarbeit wird es sein, Viskit um 3D-Funktionalitäten wie z.B. Rekonstruktion und *Volumenrendering* zu erweitern.

---

[5] überwiegend Wasserstoff, jedoch nicht Kohlenstoff oder Sauerstoff

[6] Lipinski: *Einführung in die medizintechnische Informatik*, 1999, S.261-263.

[7] Dugas/Schmidt: *Medizinische Informatik und Bioinformatik*, 2002, S.96.

[8] van Bemmel/Musen: *Handbook of Medical Informatics*, 1997, S.523.

[9] Windowing, Lookup-Table (LUT)

[10] Kontrolle und Auswertung der Bildserien durch Fachspezialisten unabhängig von weiteren Informationen.

3.4 VTK

VTK ist eine C++-Bibliothek zur Visualisierung von wissenschaftlichen Daten in verschiedenen Anwendungsgebieten wie z.B. der Geologie, Physik oder Medizin.
Sie bietet Algorithmen für das Laden, die Verarbeitung und Darstellung *bildgebender Verfahren.*
Ursprünglich wurde VTK in der Forschungsabteilung von General Electric entwickelt und wird nun von der Firma Kitware Inc. weiter gepflegt.
Aktuell umfasst VTK nach eigenen Angaben 700 Klassen und 350000 Zeilen Code.
Obwohl Kernfunktionalitäten in C++ geschrieben sind, existieren auch Schnittstellen für flexible Interpretersprachen wie z.B. Tcl/Tk und Java.

3.4.1 Design

VTK wurde möglichst unabhängig von der zugrundeliegenden 3D-Grafikbibliothek und dem Fenstersystem entworfen, weil sich zur Zeit der Entstehung[11] noch keine 3D-Standardbibliotheken wie z.B. OpenGL durchgesetzt hatten.

Das Design der Bibliothek[12] wird grundsätzlich von einem Grafikmodell und einem Visualisierungsmodell bestimmt.

3.4.2 Das Grafikmodell

Das Grafikmodell umfasst eine Menge von Kernobjekten, mit denen die grafische Darstellung der Daten erfolgt.
Dazu gehören u.a. Render-Fenster, Render-Objekte (ein Render-Fenster kann mehrere Render-Objekte enthalten), Aktoren (3D-Modell- oder Volumenobjekte), Objekte für Licht und Kamera, für 3D-Transformationen (z.B. Rotation, Verschiebung und Skalierung) und für die Interaktion mit dem Nutzer.
Vorrangig dienen die Objekte dazu eine interaktive 3D-Szene zu erzeugen.

3.4.3 Das Visualisierungsmodell

Das Visualisierungsmodell von VTK folgt dem Datenflussparadigma. Das bedeutet, dass Ein- und Ausgabe der Module Datenobjekte sind, auf die Algorithmen angewendet werden. Auf diese Art können einzelne Module verbunden werden und sich zu einem Netzwerk zusammenschließen (dem sog. Visualisierungsnetzwerk).[13]

Die Module im Visualisierungsnetzwerk von VTK sind entweder Daten- oder Prozessobjekte. Datenobjekte speichern nur die reinen Daten (z.B. Bild- oder Geometriedaten), bieten aber keine Funktionen zur Weiterverarbeitung. Prozessobjekte hingegen repräsentieren einzelne Algorithmen und dienen dazu, Datenobjekte zu erzeugen oder umzuwandeln.
Insgesamt gibt es drei Arten von Prozessobjekten : Quellen, Filter und Mapper.
Quellobjekte initiieren das Visualisierungsnetzwerk und generieren Ausgabedatenmengen. Typische Quellobjekte sind z.B. Reader, die Bilddaten von der Festplatte einlesen.
Filter wandeln ein- oder mehrere Datenobjekte in andere Datenobjekte um.[14]

[11] 1996

[12] Schroeder/Martin/Lorensen: *The design and implementation of an object-oriented toolkit for 3D graphics and visualization*, 1996

[13] Schroeder/Martin/Lorensen: *The Visualization Toolkit*, 2004, S.79-101.

[14] In einzelnen Fällen können die Datenobjekte der Ein- und Ausgabe auch identisch sein.

Mapper hingegen repräsentieren die Endpunkte des Netzwerkes, das heißt sie reichen keine Datenobjekte mehr innerhalb des Visualisierungsmodells weiter. Typische Mapper geben die Daten an das Grafikmodell weiter, oder schreiben sie z.B. wieder zurück auf die Festplatte.

3.4.4 Aktivierung des Netzwerkes

Die Verknüpfung zweier Module `A` und `B` erfolgt konkret mittels `B->setInput(A->getOutput())`.
Wichtig ist, dass nur Module mit gleichem Ein- und Ausgabeobjekt[15] verbunden werden können. Außerdem ist zu beachten, dass die Verknüpfung allein noch nicht zur Ausführung der Prozessobjekte führt.

Die Ausführung oder Aktivierung des Netzwerkes erfolgt erst in einem späteren Schritt und ist entweder *demand-driven* oder *event-driven*.
Ersteres bedeutet, dass die Ausführung nur erfolgt, wenn wirklich ein Ergebnis benötigt wird, z.B. bei der Darstellung mit Hilfe des Grafikmodells.
In diesem Fall aktivieren die Mapper die Filterobjekte und diese wiederum die Quellobjekte.
Event-driven bedeutet, dass das Netzwerk auch dann ausgeführt wird, wenn sich die Parameterwerte eines Prozessobjektes ändern[16]. Die Datenobjekte der Ausgabe werden neu berechnet und nachfolgende Knoten des Netzwerks implizit aktiviert usw.

Damit die Datenobjekte nicht immer wieder unnötig neu berechnet werden müssen, speichert sich jedes Prozessobjekt einen Änderungs- und einen Ausführzeitpunkt ab.
Bei Anforderung einer Ausgabe wird zuerst der Änderungszeitpunkt und der Änderungszeitpunkt eventueller Eingabeobjekte mit dem Ausführzeitpunkt verglichen.
Zur Neuberechnung kommt es nur, wenn eine Änderung aktueller als die letzte Ausführung war.

3.4.5 Speicherverwaltung

Mehrere Prozessobjekte können sich ein Datenobjekt teilen, falls dieses nicht von außen verändert wird.
Ein interner Referenzzähler sorgt für die automatische Löschung eines nicht mehr benötigten Datenobjektes.

Um nicht unnötig viele Daten im Hauptspeicher zu halten, hat man die Möglichkeit einem Prozessobjekt mitzuteilen, ob die Ausgaben einmalig oder wiederholt erfolgen sollen.
Wird die Ausgabe nur einmal erzeugt, muss das Prozessobjekt nach der Berechnung keine Referenz auf die Ausgabeobjekte halten. Der Referenzzähler kann für diese Objekte runtergezählt werden. Falls das Prozessobjekt die letzte Referenz hatte, können die Objekte automatisch aus dem Speicher entfernt werden.

[15] Die Objekte müssen derselben Klasse angehören oder in der Vererbungshierarchie zumindestens dieselbe Oberklasse besitzen.
[16] Z.B. durch Nutzereingabe.

3.5 JOGL/OpenGL

OpenGL ist eine der bekanntesten und verbreitetsten Programmierschnittstellen (API) für die Erstellung und Darstellung von 2D- und 3D-Grafik. OpenGL ist im Gegensatz zu DirectX, das von Microsoft ausschließlich für die Windows-Plattform entwickelt wurde, plattformunabhängig und wird von einer Vielzahl von Grafikkarten- und Grafiktreiberherstellern unterstützt. Die API wurde zum ersten Mal 1992 veröffentlicht und seither stetig weiterentwickelt. Aktuell existiert sie in der Version 4.5 (Stand Dezember 2014) und unterstützt dabei Features wie etwa *3D-Texturierung*, Framebuffer-Objekte und die Programmierung eigener *Shader*.

JOGL ist eine Implementierung von OpenGL für die Programmiersprache Java. Als Java Specification Request (JSR) 231 ist JOGL eine offizielle Erweiterung des Java-Standards. Zu einigen Vorteilen gegenüber anderen OpenGL-Implementierungen in Java wie etwa LJWGL zählt die gleichzeitige Erstellung mehrerer OpenGL-Kontexte, sowie die gute Zusammenarbeit mit Java2D und der *Swing*-GUI. So lässt sich z.B. ein OpenGL-Fenster in JOGL einfach als GUI-Komponente einbinden. Zudem ist die aktuelle Version (2.2.0) dank der Unterstützung von OpenGL ES 1 und 2 auch für die 2D- und 3D-Entwicklung auf mobilen Geräten ausgelegt.

OpenGL wird auch von der Bibliothek VTK für die Darstellung eingesetzt.

Kapitel 4

3D-Rekonstruktionen

In der medizinischen Bildverarbeitung dienen *3D-Rekonstruktionen* vorwiegend zur Gewinnung neuer (räumlicher) Informationen aus vorliegenden 2D-Bilddatensätzen.
Häufiges Ziel ist ein Modell der untersuchten Person oder des untersuchten Körperteils. Dabei möchte man entweder eine freie Ansicht aus unterschiedlichen Perspektiven wählen oder alternative Querschnitte generieren.
Hinzu kommt der Wunsch sog. ROI's bzw. VOI's besser lokalisieren zu können.

Fortgeschrittene Anwendungen dienen der computer- und roboterassistierenden Chirugie oder dem Einsatz in Trainingssimulationen. Möglich ist z.B. auch die Erzeugung einer passgenauen Knochenprothese aus einem Polygonmodell.

In *viskit* soll die 3D-Rekonstruktion in erster Linie Bildserienvergleiche unterstützen.[1]

Ausgangspunkt ist eine Serie von Schichtbildern (Bildstacks) aus CT- oder MR-Scans, die hintereinander mit meist gleichmäßigem Schichtabstand angeordnet sind.
Die Bildserie an sich kann auch als 3D-Datenfeld oder Datenwürfel interpretiert werden, allerdings müssen Schicht- und Pixelabstand in den 2D-Bildern nicht unbedingt identisch sind, d.h. es handelt sich eher um einen Datenquader[2].

Mögliche Probleme, die eine Rekonstruktion im Vorfeld eventuell erschweren könnten, liegen in verfälschten Messdaten, welche z.B. durch Noise (Störrauschen) oder Fehler in Messinstrumenten hervorgerufen werden.
Weiterhin ist zu beachten, dass als Grundlage ein diskreter 3D-Datensatz dient, der nur durch Interpolation in ein stetiges Modell überführt werden kann.
Eine zu niedrige Slicingzahl[3] oder eine zu geringe Auflösung könnten dazu führen, dass eine originalgetreue Objektnachbildung nahezu unmöglich wird.

Will man einen einheitlichen Algorithmus entwickeln, stellen sich weitere Hindernisse in den Weg, z.B. können die vorliegenden Bilder grundsätzlich aus verschiedenen *Modalitäten* mit unterschiedlichen Einstellungen stammen.
Theoretisch kann es sich um Bilder mit unterschiedlicher Auflösung und beliebigem Speicherfor-

[1] in sog. Vorher-/Nachher-Studien
[2] bestehend aus einzelnen quaderförmigen Voxeln
[3] Anzahl der Schichtbilder

mat[4] handeln.
Hochauflösende Bilder können darüber hinaus schnell auf Speicher- und Berechnungsgrenzen stoßen.

Leider gibt es keine einheitliche Interpretation der Bilddaten. Während das CT-Verfahren durch die *Hounsfield-Einheiten* standardisiert ist, sind die Werte bei MRT unterschiedlich und hängen z.B. von den Aufnahmezeiten (T1, T2), aber auch von anderen Parametern wie z.B. der eingesetzten Feldstärke ab.

Die Rekonstruktion wird sich somit nicht vollautomatisch sondern nur mit Hilfe des interagierenden Benutzers durchführen lassen.

Das einfachsten Verfahren der Rekonstruktion besteht in der orthogonal frontalen oder seitlichen Bildrekonstruktion, auch multiplanare Reformatierung[5] kurz *MPR* genannt.
In diesem Fall handelt es sich um eine ebene Rekonstruktion von Sekundarschnittbildern.

Für die echte 3D-Konstruktion hingegen existieren 2 Hauptverfahren : die Generierung eines Polygonmodells und das Volumenrendering.
Eine dritte Methode ist das sog. Shell-Rendering[6], das eine Kombination aus Polygon- und Volumenrendering darstellt und hier nicht weiter betrachtet wird.

4.1 Oberflächenbasierte Visualisierung

Das Hauptziel bei der oberflächenbasierten Visualisierung ist die Generierung eines Oberflächenmodells, dass durch ein Polygonnetz[7] approximiert wird. Die Vorteile gegenüber dem voxelbasiertem Volumenrendering liegen in der besseren Oberflächenapproximation und der Laufzeit, da das Modell nur einmalig aus dem vorhandenen Datensatz erzeugt werden muss. Allerdings muss dafür in Kauf genommen werden, dass man ein starres Modell erhält, welches meist nicht gut für die Interaktion geeignet ist[8]. Oberflächenbasierte Visualisierung eignet sich gut für CT-Scans, insbesondere für die Darstellung von Knochen und Haut, die eine relativ glatte geschlossene Oberfläche aufweisen und mit wenigen Polygonen auskommen.

4.1.1 Der Marching-Cubes-Algorithmus

Der ***Marching-Cubes-Algorithmus*** ist ein Standardverfahren zur Generierung eines Oberflächenmodells aus einem medizinischen 3D-Datensatz.
Er ist neben der konturbasierten Triangulation[9] das wichtigste Verfahren zur Erstellung eines Polygonmodells.

Beim Marching-Cubes-Algorithmus wird der vorliegende Datenquader als binärer Datensatz betrachtet.

[4] Z.B. 8 Bit, 12 Bit oder 16 Bit.
[5] HANDELS: *Medizinische Bildverarbeitung*, 2009, S.299-301.
[6] VAN BEMMEL/MUSEN: *Handbook of Medical Informatics*, 1997, S.419.
[7] Meist hardwareunterstützte Dreiecke.
[8] Z.B. bei Änderung der Werte im Windowing.
[9] HANDELS: *Medizinische Bildverarbeitung*, 2009, S.304.

Nachdem ein Schwellwert t vorgegeben wurde, werden die Punkte in Objekt- und Nichtobjektvoxel unterteilt.[10]
Dann werden alle benachbarten Nachbarpunkte betrachtet, und zwar so, dass immer 8 Punkte die Ecken eines Quaders bilden.
Diese lassen sich in $2^8 = 256$ verschiedene Kombinationen von Objekt-/Nichtobjektvoxelkonfigurationen klassifizieren, die aus Symmetriegründen auf 15 verschiedene Varianten reduziert werden können.
Jede Konfiguration wird durch eine Dreiecksmenge repräsentiert, wobei jeder Dreieckspunkt immer auf einer Kante dieses Quaders liegt, dessen Endpunkte verschiedene binäre Werte haben.
Der Dreieckspunkt kann dabei entweder genau auf der Mitte der Kante liegen oder abhängig vom Schwellwert und den Funktionswerten der Kanten f_O[11] und f_NO [12] interpoliert werden, was zu einem besseren Ergebnis führt.
Der genaue Interpolationsfaktor berechnet sich als $\alpha = \frac{(t-f_NO)}{f_O-f_NO} \in [0,1]$.
Setzt man schließlich alle Dreiecksmengen zusammen, erhält man das gewünschte Dreiecksnetz zur Beschreibung der Objektoberfläche.

Mögliche Probleme, die sich aus Mehrdeutigkeiten bei bestimmten Voxelkonfigurationen ergeben und zu „Löchern“ innerhalb des Modells führen, lassen sich durch eine Erweiterung des Basisalgorithmus beheben.[13]

Ein weiteres Problem ist die relativ hohe Anzahl von generierten Dreiecken.
Das Polygonnetz lässt sich jedoch durch Ausdünnungsverfahren[14] vereinfachen, wobei Oberflächen mit geringer Krümmung auf wenige Dreiecke reduziert werden, was kaum zu sichtbaren Qualitätseinbußen führt.[15]

4.2 Volumenrendering

Oberflächenbasierte Visualisierungsverfahren beschränken sich auf die Darstellung einer zweidimensionalen Oberfläche. Polygonmodelle sind nur in der Lage einen Teil des Volumendatensatzes zu repräsentieren, während direktes ***Volumenrendering*** zur Darstellung des gesamten Datensatzes dient. Dabei ist es auch möglich Zwischenstrukturen wie Hirn, Muskel oder Gefäße gleichzeitig anzuzeigen. Nachteilig ist, dass immer auf dem gesamten Datensatz gerechnet wird, was bei größeren Datensätzen eine gute Hardware und jede Menge Optimierungen in den Algorithmen erfordert.[16]

Beim Volumenrendering geht es darum, aus den vorliegenden medizinischen Daten[17] optische Eigenschaften wie *Lichtabsorption*[18] und *Lichtemission*[19] des vorliegenden Mediums abzuleiten.[20]

[10] $t \leq f(x,y,z)$ bzw. $t > f(x,y,z)$
[11] Wert des Objektvoxels der Kante
[12] Wert des Nichtobjektvoxels
[13] Preim/Bartz: *Visualization in Medicine*, 2007.
[14] engl. decimation algorithms
[15] Schroeder/Zarge/Lorensen: *Decimation of triangle meshes*, 1992.
[16] Hansen/Johnson: *Visualization Handbook*, 2004, S.127-260.
[17] Häufig skalare Werte, wie z.B. Dichtewerte bei CT
[18] Bei Flüssigkeiten oder Luft
[19] Farbe des Materials
[20] Engel et al.: *Real-time Volume Graphics*, 2006, S.4-8.

Von den vielen verschiedenen physikalischen optischen Modellen wird für das Volumenrendering häufig das *Emissions-Absorptions-Modell* benutzt. Dabei handelt es sich um ein vereinfachtes Modell, welches nicht sehr rechenaufwändig ist, aber dennoch gute Resultate liefert.

4.2.1 Das Volumenrenderingintegral

Das *Emissions-Absorptions-Modell* basiert auf folgender Integralgleichung:[21]

$$I(D) = I_0 \cdot e^{-\int_{s_0}^{D} k(t)\,dt} + \int_{s_0}^{D} q(s) \cdot e^{-\int_{s}^{D} k(t)\,dt}\,ds. \tag{4.1}$$

Die Gleichung beschreibt die sichtbare Intensität eines Punktes $I(D)$, welche sich als Integration entlang eines Sichtstrahls vom Eintrittspunkt des Volumenmediums s_0 zum Austrittspunkt D in Richtung der Kamera ergibt.
Der erste Teil der Gleichung gibt an, wie viel Licht vom Volumenmedium absorbiert wird und welcher Anteil vom Hintergrundlicht I_0 noch die Kamera erreicht.
Der zweite Teil beschreibt, wie stark das vom Material emittierte Licht $q(s)$ innerhalb des Volumens abgeschwächt wird. Die Absorptionsfähigkeit des Material wird dabei durch die Funktion $k(t)$ beschrieben.
Sowohl $q(s)$ als auch $k(t)$ hängen direkt von den gemessenen Daten $f(x, y, z)$ ab und müssen durch eine geeignete *Transferfunktion* angegeben werden.

Der Term $\tau(s1, s2) = \int_{s_1}^{s_2} k(t)\,dt$ wird als *optische Tiefe* bezeichnet und gibt an, wie viel Licht zwischen den Punkten s_1 und s_2 absorbiert wird.
Ein niedriger Wert besagt, dass das Material sehr transparent ist, während ein hoher Wert Lichtundurchlässigkeit bedeutet.

Fasst man die Transparenz als $T(s_1, s_2) = e^{-\tau(s_1,s_2)}$ zusammen, lässt sich das *Volumenrenderingintegral* auch vereinfacht schreiben als :

$$I(D) = I_0 \cdot T(s_0, D) + \int_{s_0}^{D} q(s) \cdot T(s, D)\,ds. \tag{4.2}$$

Diskretisierung

Da das Volumen aus einem diskreten Datensatz besteht, ist eine analytische Integralberechnung nicht möglich.
Stattdessen muss das Integral diskretisiert werden, z.B. durch eine *Riemannsche Summe*[22]:

$$\tau(s_0, D) = \tau(s_0, s_n) = \int_{s_0}^{s_n} k(t)\,dt = \sum_{i=0}^{n-1} \int_{s_i}^{s_{i+1}} k(t)\,dt \approx \sum_{i=0}^{n-1} \delta \cdot k(s_i) = \delta \cdot \sum_{i=0}^{n-1} k(s_i) \tag{4.3}$$

mit $\delta = \frac{s_n - s_0}{n-1}$

[21] Engel et al.: *Real-time Volume Graphics*, 2006, S.8-10.
[22] Mit equidistanter Intervalleinteilung.

Die Grundidee ist es, das Intervall $[s_0, D = s_n]$ in kleinere Intervalle $[s_0, s_1], \cdots [s_{n-1}, s_n]$ aufzuteilen.
Für jeden Abschnitt gilt dann:

$$I(s_i) = I(s_{i-1}) \cdot T(s_{i-1}, s_i) + \int_{s_{i-1}}^{s_i} q(s) \cdot T(s, s_i)\, ds \tag{4.4}$$

Mit $T_i = T(s_{i-1}, s_i)$ und $c_i = \int_{s_{i-1}}^{s_i} q(s) \cdot T(s, s_i)\, ds$ lässt sich das Volumenrenderingintegral schreiben als :

$$I(D) = I(s_n) = I(s_{n-1}) \cdot T_n + c_n = (I(s_{n-2}) \cdot T_{n-1} + c_{n-1}) \cdot T_n + c_n = \cdots \tag{4.5}$$

oder als :

$$I(D) = \sum_{i=0}^{n} c_i \prod_{j=i+1}^{n} T_j \tag{4.6}$$

wobei $T_i \approx e^{-k(s_i)\cdot\delta}$ und $c_i \approx q(s_i) \cdot \delta$.

Die Integralannäherung durch eine Riemannsche Summe kann man auch durch Anwendung der *Trapezregel*, der *Simpsonregel* oder einer anderen *Newton-Coates-Formel* verbessern, allerdings erhält man dadurch keine Verbesserung an unstetigen Stellen, wie z.B. an der Grenze zwischen verschiedenen Materialschichten.[23]

Komposition

Für die Berechnung der Gleichung (4.6) gibt es zwei Möglichkeiten : Entweder man lässt den Index i von 0 bis n laufen[24] oder rückwärts von n bis 0[25]. Die zweite Variante bietet mehr Vorteile, da man die Berechnung frühzeitig abbrechen kann, falls der summierte Transparenzwert klein genug ist.

Bei der *Front-To-Back-Komposition* kann man die Gleichung (4.6) effizienter berechnen, wenn man das Ergebnis der inneren Produktformel in jedem Schritt in einer Zwischenvariable speichert. Pro Schritt würden sich folgende Aktualisierungen ergeben:

$$\begin{aligned} \hat{C}_i &= \hat{C}_{i+1} + \hat{T}_{i+1} \cdot C_i \\ \hat{T}_i &= \hat{T}_{i+1} \cdot T_i \end{aligned} \tag{4.7}$$

mit $\hat{C}_n := C_n$ und $\hat{T}_n := T_n$

Setzt man $C_{dst} := C_j$ (mit $j = i, i+1$), $C_{src} := C_i, \alpha_{dst} := 1 - \hat{T}_j$ und $\alpha_{src} := 1 - T_i$, erhält man die finale Kompositionsformel :

$$\boxed{\begin{aligned} C_{dst} &\leftarrow C_{dst} + (1 - \alpha_{dst}) \cdot C_{src} \\ \alpha_{dst} &\leftarrow \alpha_{dst} + (1 - \alpha_{dst}) \cdot \alpha_{src} \end{aligned}} \tag{4.8}$$

[23] Engel et al.: *Real-time Volume Graphics*, 2006, S.12-13.
[24] *Back-To-Front-Komposition*.
[25] *Front-To-Back-Komposition*, vom Betrachter zum Volumen.

Das Volumenrenderingintegral kann also für einen Strahl berechnet werden, indem man in jedem Schritt wiederholt die Variablen C_{dst} und α_{dst} mittels der Kompositionsformel (4.8) aktualisiert. $\hat{C}_0 = C_{dst}$ liefert im letzten Schritt den finalen sichtbaren Farbwert.

4.2.2 Ray Casting

Ausgehend vom diskretisierten Volumenrenderingintegral (4.6) bietet sich ***Ray Casting*** als geeignetste Technik für Volumenrendering an.
Ray Casting existiert CPU-basiert bereits seit den 80er Jahren[26]. Implementierungen für die Grafikkarte gibt es allerdings erst seit 2003 (*GPU-Ray-Casting*)[27,28].

Die Hauptidee besteht darin, für jeden Pixel das Volumenrenderingintegral entlang der Strahlen ausgehend von der Kamera zu berechnen.[29]
Die Volumendaten werden an diskreten Positionen entlang eines Strahls abgetastet, optische Eigenschaften wie Emission und Absorption mittels einer Transferfunktion berechnet.
Vorzugsweise wird die *Front-To-Back*-Komposition benutzt (4.8), mit der ein früher Abbruch der Berechnungen erzielt werden kann[30].

Die Vorteile dieser Technik sind, dass sie eine parallele Berechnung zulässt[31] und dass sie sehr flexibel ist. Weiterhin bietet sie viele Möglichkeiten Berechnungen zu optimieren und früh abzubrechen[32].

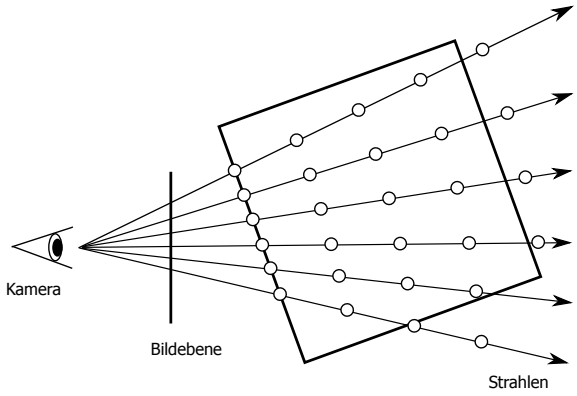


Abbildung 4.1: Grundprinzip vom Ray Casting. Für jeden Pixel auf der Bildebene wird entlang eines Sichtstrahls das Volumenrenderingintegral berechnet. Dabei wird jeder Strahl ausgehend vom Volumeneintrittspunkt an diskreten Positionen abgetastet.

[26] Levoy: *Display of Surfaces From Volume Data*, 1988.
[27] Roettger et al.: *Smart hardware-accelerated volume rendering*, 2003.
[28] Kruger/Westermann: *Acceleration Techniques for GPU-based Volume Rendering*, 2003.
[29] Ähnlich wie beim *Ray Tracing*.
[30] Falls $\alpha_{dst} \geq 1 - \epsilon$.
[31] Somit gut geeignet für aktuelle GPU-Hardware, die mehrere parallele Recheneinheiten besitzt
[32] z.B. mittels *Early-Ray-Termination*

Algorithmus

Der Hauptalgorithmus für *Ray Casting* ist folgendermaßen aufgebaut :

1. Strahl-Initialisierung Für jeden dargestellten Pixel werden die Parameter des zugehörigen Sichtstrahles festgelegt. Dazu werden abhängig von der Kameraposition und der Kameraausrichtung die Eintrittspunkte der Strahlen im Volumenquader sowie die Richtungen der Strahlen berechnet. Die Strahlrichtung ergibt sich aus den Koordinaten der Kamera, der Position des dargestellten Pixels in der Bildebene sowie dem horizontalen und vertikalen Bildwinkel der Kamera. Die Eintrittspunkte im Volumenquader entsprechen den Koordinaten, an denen die Strahlen ausgehend von der Kamera und abhängig von der Strahlrichtung zum ersten Mal die Oberfläche des Volumenquaders passieren.

2. Hauptschleife Im Hauptteil des Algorithmus wird entlang des Strahls das jeweilige Volumenrenderingintegral berechnet. Dazu wird der Strahl in diskrete Positionen unterteilt und von der Hauptschleife an diesen abgetastet. Mit den ermittelten Werten wird der zu bestimmende Pixelwert Schritt für Schritt anhand einer Kompositionsformel (4.8) aktualisiert.

2.1 Datenzugriff An der aktuellen Strahlposition wird auf den Datenbereich zugriffen. Bei nicht diskreten Positionen im Volumenquader muss für die Rekonstruktion der Daten ein Interpolationsverfahren angewendet werden. Um Farb- und Transparenzwerte aus den skalaren Daten zu erhalten (Klassifikation) wird eine Transferfunktion angewendet. Die Reihenfolge von Klassifikation und Interpolation spielt eine wichtige Rolle für die Qualität der 3D-Rekonstruktion[33].

2.2 Komposition Aktualisierung der momentanen Farb- und Transparenzvariablen mittels Gleichung (4.8).

2.3 Strahlposition fortsetzen Nächste Position entlang des Strahls berechnen.

2.4 Strahlabbruch Die Hauptschleife ist beendet, wenn der Strahl den Volumenquader verlässt. An dieser Stelle wird berechnet, ob sich die aktuelle Strahlposition noch innerhalb des Volumens befindet und mit der Iteration der Hauptschleife fortgefahren werden kann. Alternativ kann die Berechnung frühzeitig abgebrochen werden, wenn der Wert der aktuellen Transparenz nahe bei 0 liegt. In diesem Fall haben die nachfolgend zu ermittelten Werte keinen Einfluss mehr auf die Darstellung des momentanen Pixels.[34]

[33] siehe Kapitel 5.2.2

[34] $\alpha_{dst} \sim 1$, Early-Ray-Termination.

Kapitel 5

Implementierung

Zunächst werde ich vorstellen, wie ich dabei vorgegangen bin, die oberflächenbasierte Visualisierung und Volumenrendering mit Hilfe von VTK in Viskit umzusetzen. Dazu habe ich jeweils eine kleine Testanwendung entwickelt, mit der die Visualisierungen betrachtet und die wichtigsten Parameter über eine GUI gesteuert werden können (Abbildungen 9.1 und 9.2). Hauptaufgabe war es dabei, eine geeignete Schnittstelle zwischen der Java-Bibliothek Viskit und der C++-Bibliothek VTK zu schaffen. Weiterhin mussten geeignete Algorithmen und Module in VTK ausgewählt und zu einer Renderpipeline zusammengesetzt werden.

5.1 Oberflächenbasierte Visualisierung mit VTK

Für die Erstellung von oberflächenbasierten Modellen (kurz *Isofläche*) stehen in *VTK* zwei verschiedene Funktionen zur Verfügung. Während `vtkContourFilter` generell Konturen für 1-, 2- oder 3-dimensionale Datenfelder erzeugt, ist `vtkMarchingCubes` speziell für 3D-Bilddaten gedacht und sollte in diesem Fall aufgrund einer besseren Performance bevorzugt werden.[1]

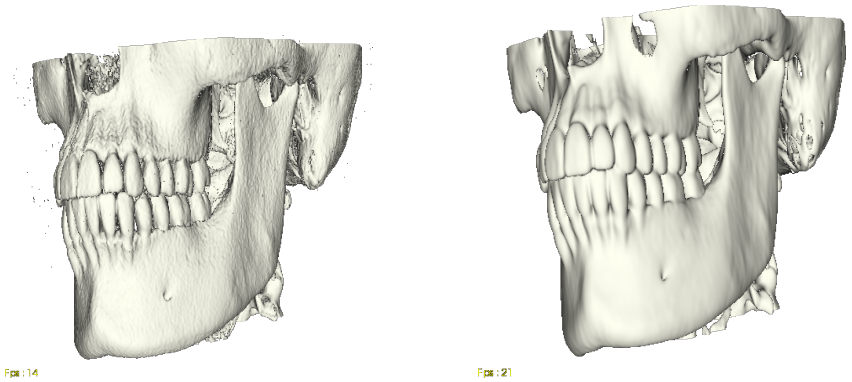
`vtkMarchingCubes` erwartet als Eingabe ein `vtkImageData`-Objekt, welches erst aus der Liste der *DICOM-Objekte* erzeugt werden muss. Nachfolgend wird genauer erläutert, wie ich bei der Generierung und Anzeige der Isoflächen in VTK vorgegangen bin.

5.1.1 Renderpipeline

1. **Konvertierung der Daten** Die Liste der DICOM-Objekte wird zuerst in ein (16-Bit)-`short`-Array konvertiert. Es wird davon ausgegangen, dass die Werte aller DICOM-Bilder im 16-Bit-Format vorliegen (Standard bei MRT und CT).[2] Das Java-Plugin für VTK bietet dabei die Möglichkeit, mittels des Daten-Wrapper-Objektes `vtkShortArray` ein `vtkImageData`-Objekt effizient aus einem Java-`short`-Array heraus zu erzeugen. Bei der Erzeugung des `vtkImageData`-Objektes muss auch darauf geachtet werden, die Pixel- und Schichtabstände[3] aus den *DICOM-Headern* zu übergeben, damit die Proportionen der Kantenlängen des Volumenquaders erhalten bleiben.

[1] Schroeder/Martin/Lorensen: *The Visualization Toolkit*, 2004, S.178-179.
[2] Auch interne 12-Bit-Werte werden extern als `short` geliefert.
[3] `spacing`

2. Weichzeichnerfilter Als einer der wichtigsten Schritte wird an dieser Stelle ein *Gaußscher Weichzeichnerfilter* (`vtkImageGaussianSmooth()`) auf die Bilddaten angewendet, der später für eine glattere zusammenhängende Oberfläche sorgt, ansonsten wirkt das erzeugte Modell etwas kantig und unnatürlich[4]. `vtkImageGaussianSmooth()` verwendet einen 3x3x3-Filterkernel mit einer Standardabweichung von 2.



(a) Original (b) Mit Weichzeichnerfilter

Abbildung 5.1: Anwendung des Weichzeichnerfilters

3. Resampling Um die Laufzeit der nachfolgenden Operationen zu verkürzen, wird nun der Volumenquader mittels `vtkImageResample()` verkleinert. Das *Resampling* wird erst nach dem *Weichzeichnerfilter* angewendet, um eine höhere Qualität zu erzielen[5]. Die Mindestlänge des Bildquaders kann vom Anwender eingestellt werden.[6]

4. Umwandlung in Polygondaten Aus dem geglätteten und verkleinerten Volumenquader werden mit Hilfe des Marching-Cubes-Algorithmus die Polygone einer (frei wählbaren) Isofläche erzeugt. Da bei diesem Algorithmus mitunter auch Quader entstehen, werden diese für die nachfolgenden Algorithmen mittels `vtkTriangleFilter()` in Dreiecke zerlegt.

5. Dreiecksdezimierung Die Anzahl der generierten Dreiecke lässt sich wesentlich[7] ohne nennenswerte Qualitätseinbußen reduzieren. Hierzu wird der Filter `vtkDecimatePro()` angewendet.[8] Die *Dreiecksdezimierung* kostet etwas Laufzeit bei der Generierung, jedoch wirkt sich eine reduzierte Anzahl der Polygone positiv auf die *Framerate* und damit auf die *Render-Laufzeit* aus.

[4] Als Folge des Marching-Cubes-Algorithmus.

[5] Für eine bessere Performance könnten beide Schritte theoretisch auch vertauscht werden.

[6] Der optimale Wert sollte ein trade-off zwischen Performance und Qualität sein. Auf einem Bürorechner (Testsystem T1, siehe Kapitel 6.1) lag dieser für einen Volumenquader mit einer Kantenbreite von 512 Pixeln bei circa 150 Pixeln.

[7] Bei einem maximal zugelassen Fehler von 0.001% auf bis zu 25% der ursprünglichen Dreiecksanzahl, siehe Tabelle 6.1.

[8] Dieser verwendet den leicht abgewandelten Dezimierungs-Algorithmus, der auf der SIGGRAPH '92 vorgestellt wurde. Schroeder/Zarge/Lorensen: *Decimation of triangle meshes*, 1992

6. **Mesh-Smoothing** Optional kann mittels `vtkSmoothPolyDataFilter()` eine Glättung des Polygonnetzes erzielt werden, jedoch scheint *Mesh-Smoothing* im Vergleich zum Weichzeichnerfilter keine signifikanten Qualitätsverbesserungen zu bringen.

7. **Rendering** Im letzten Schritt wird die Liste der erzeugten Dreiecke an den Renderer von VTK übergeben, welcher für eine automatische Anzeige sorgt. Wahlweise könnte die Dreiecksliste auch benutzt werden, um das Modell mit einem eigenen Renderer (z.B. OpenGL/JOGL) darzustellen.

Umwandlung und Rendering finden zusammen in der Klasse `ContourView`[9] statt.

Der gesamte Konvertierungsprozess kann interaktiv beeinflusst werden. Weichzeichnerfilter, Dreiecksdezimierung und Mesh-Smoothing können beliebig aktiviert oder deaktiviert werden[10]. Weiterhin lässt sich beim Resampling die Mindestlänge des Volumenquaders und der Konturwert für die Erzeugung der Isofläche setzen.

[9] Im Package `de.sofd.viskit.image3D.vtk.view`, ContourView ist abgeleitet von `vtkPanel` und kann als normales Panel in *AWT* oder *Swing* eingebunden werden.

[10] In diesem Fall wird die Renderpipeline entsprechend angepasst

5.1.2 Software-Architektur

Folgendes Klassen-Diagramm zeigt die einzelnen Komponenten des ersten Programms (`ContourViewer`) und wie diese in Zusammenhang stehen.

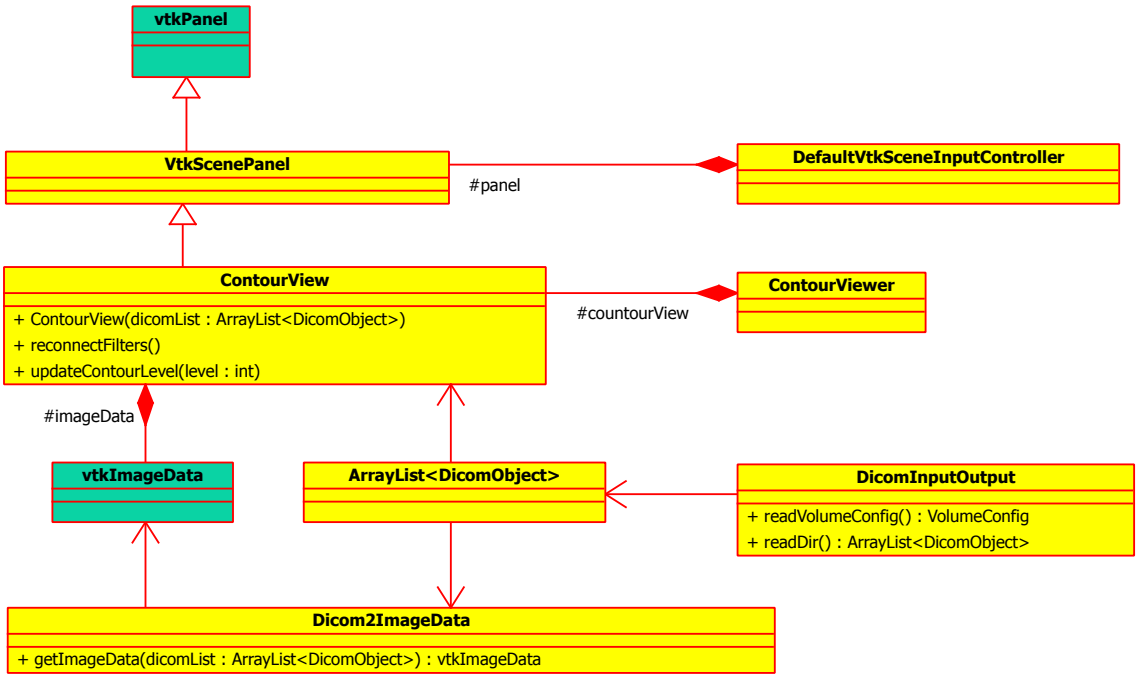


Abbildung 5.2: Klassen-Diagramm `ContourViewer` (vereinfacht), gelb: eigene Komponenten, türkis: vtk-Komponenten

Als Basis dient die Klasse `vtkPanel`, welche zum VTK Java Wrapping Paket gehört[11]. Im `vtkPanel` findet das eigentliche Rendering statt. Die Klasse ist von `java.awt.Canvas` abgeleitet und kann in eine beliebige AWT- oder Swing-Anwendung eingebunden werden.
Darüber hinaus habe ich in `viskit` eine Unterklasse `vtkScenePanel` entwickelt, die bestimmte Hilfsmethoden zum Betrachten einer Renderszene zur Verfügung stellt. Dazu gehört z.B. eine Kamerasteuerung (translate, zoom, rotate) und deren Verknüpfung mit Eingabecontrollern (Maus, Keyboard).
Die eigentliche Steuerung findet in `DefaultVtkSceneInputController` statt, in der das Verhalten von Maus und Keyboard genau definiert wird.

Von `vtkScenePanel` ist die Klasse `ContourView` abgeleitet, in der die *Renderpipeline* aus 5.1.1 erstellt wird. Der Konstruktor dieser Klasse erwartet eine Liste mit DICOM-Objekten, die über die Hilfsklasse `Dicom2ImageData` in ein `vtkImageData`-Objekt umgewandelt wird. `Dicom2ImageData` liest dazu aus dem DICOM-Header die Anzahl der Zeilen und Spalten, sowie die Pixel- und Schichtabstände. Dann wird als Zwischenschritt ein `ShortBuffer` erstellt, in dem die Pixeldaten jedes einzelnen Bildes gesammelt werden. Optional hat man die Möglichkeit die Daten über die *DICOM-Tags RescaleIntercept* und *RescaleSlope* zu normieren.

[11] `http://www.cmake.org/Wiki/VTK/Java_Wrapping`

Dazu ist zu sagen, dass die Bilddaten häufig nicht direkt abgespeichert, sondern vorher mittels einer linearen Transformation auf einen bestimmten Wertebereich gebracht werden. Hat man z.B. CT-Daten im Bereich von -1000 bis 2000, so werden diese häufig in den Bereich 0 bis 4096 transformiert (12 Bit, *unsigned*). Die genaue Formel für die Transformation lautet :

$$y = rescaleSlope \cdot x + rescaleIntercept$$

Aus dem erstellten `ShortBuffer` wird dann ein `vtkShortArray`[12] erzeugt und daraus schließlich das `vtkImageData`-Objekt.

Nach Erstellung der Renderpipeline und dem ersten Rendering wird `ContourView` durch die Hauptklasse `ContourViewer` kontrolliert, welches die entsprechenden Steuerelemente z.B. zum Setzen der Konturstufe enthält. Bei einigen Aktionen, wie der Aktivierung des Weichzeichnerfilters, ist es nötig, die Renderpipeline neu zusammenzusetzen, da der nachfolgende Schritt, wie z.B. das Resampling, seine Eingabe von `vtkImageGaussianSmooth` oder direkt von `vtkImageData` erhält. Das Zusammensetzen der Renderpipeline findet in der Methode `reconnectFilters` der Klasse `ContourView` statt.

[12] Dieses stammt ebenfalls aus dem Java-Wrapping-Paket

5.2 Volumenrendering mit VTK

Für das *Volumenrendering* stehen in VTK insgesamt vier Funktionen zur Verfügung[13], die allesamt auf dem Ray-Casting-Algorithmus basieren[14].
Dazu gehören:

1. **vtkVolumeRayCastMapper** Diese Funktion ist eine CPU-basierte Implementierung des Ray-Casting-Algorithmus. Sie bietet von allen vier Volumenrenderern die höchste Qualität, benötigt allerdings auch die meiste Rechenzeit. Leider werden nur die Datentypen `unsigned short`[15] und `unsigned char` unterstützt, für medizinische Bilder wird aber mindestens eine Unterstützung von `signed short`[16] benötigt.

2. **vtkFixedPointVolumeRayCastMapper** Eine Erweiterung von `vtkVolumeRayCastMapper`, die auch andere primitive Datentypen (u.a. `short`) unterstützt. Zusätzlich sind noch Performance-Optimierungen wie z.B. *Space leaping* und Early-Ray-Termination eingebaut. Trotzdem scheint diese Funktion für ein interaktives Volumenrendering zu langsam zu sein. Ein Vorteil ist allerdings, das softwarebasiertes Ray Casting in VTK multiple Prozessoren unterstützt[17].

3. **vtkVolumeTextureMapper2D** Hardwarebasiertes Ray Casting, bei dem die Daten im `vtkImageData`-Objekt in 2D-*Textur*schichten unterteilt wird. Leider werden auch hier nur die Datentypen `unsigned short` und `unsigned char` unterstützt, zudem gibt es nur eine *bilineare Interpolation*[18].

4. **vtkVolumeTextureMapper3D** Entspricht auch hardwarebasiertem Ray Casting. Das `vtkImageData`-Objekt wird komplett als 3D-Textur auf der Grafikkarte gespeichert und es gibt somit auch eine *trilineare Interpolation*. Einziger Nachteil ist, dass die Maximalgrösse des Texturwürfels auf 256x256x128 Voxel beschränkt ist (unabhängig wieviel Speicher in der Grafikkarte wirklich vorhanden ist). Hat das `vtkImageData`-Objekt größere Ausmaße, findet eine implizite Verkleinerung statt.

Da alle Daten im (`signed`) `short`-Format vorliegen, kommen nur die Methoden 2. und 4. in Betracht. Da sowohl Interaktivität als auch eine hohe Ausgabequalität wichtig sind, scheint kein Renderer optimal zu sein, jeder hat seine Vor- und Nachteile. `vtkFixedPointVolumeRayCastMapper` eignet sich gut für die finale Ausgabe, aber bei weniger als 1 *FPS* (Bilder pro Sekunde) nicht für ein interaktives Rendering. `vtkVolumeTextureMapper3D` läuft unter einer wesentlich besseren Framerate (10-30). Die meisten medizinischen Bilder haben jedoch eine Auflösung von 256 bis 1024 Pixel, d.h. bei der eingeschränkten Texturgröße würde es unweigerlich zu Qualitätsverlusten kommen, die durch eine Interpolation nicht unbedingt wieder ausgeglichen werden kann. Am besten wäre es wahrscheinlich beide Render-Methoden zu kombinieren, indem man den performanten `vtkVolumeTextureMapper3D` für Interaktionen und den höher qualitativen `vtkFixedPointVolumeRayCastMapper` für die statische Ansicht verwendet.[19] Bei dieser Lösung muss allerdings ein erhöhter Speicherbedarf in Kauf genommen werden, sowohl für den Haupt- als auch für den Grafikspeicher.

[13] KITWARE: *VTK User's Guide*, 2006, S.123-146.
[14] siehe Abschnitt 4.2.2
[15] Wertebereich von -32768 bis 32767.
[16] Wertebereich von 0 bis 65535.
[17] KITWARE: *VTK User's Guide*, 2006, S.140.
[18] nur zweidimensional, keine Interpolation zwischen den Texturschichten
[19] siehe dazu auch Kapitel 5.2.2

5.2.1 Problem mit Datenformat

Warum lässt sich die Methode `vtkVolumeRayCastMapper` nicht nutzen, obwohl das unterstützte Format `unsigned short` doch die gleiche Auflösung wie das Format `signed short` hat, in dem die Daten vorliegen? Können die Daten nicht einfach konvertiert werden? Dazu muss man wissen, wie die Daten an vtkVolumeRayCastMapper übergeben werden. Alle Mapper erwarten als Eingabe ein `vtkImageData`-Objekt, das die eigentlichen Rohdaten kapselt. Über `vtkImageData.GetPointData().SetScalars()` werden die Rohdaten übergeben, wobei `SetScalars()` als einzigen Parameter ein Objekt vom Typ `vtkDataArray` erwartet, wiederum eine Kapselung der primitiven Java-Datatypen. Von `vtkDataArray` existieren verschiedene abgeleitete Klassen, z.B. auch `vtkShortArray` und `vtkUnsignedShortArray`, mit denen man theoretisch Daten im `signed` oder `unsigned short`-Format übergeben könnte, was letztendlich mit der Methode `SetJavaArray()` getan wird. Übergibt man nun bei `vtkUnsignedShortArray` ein Java-`short[]`-Array (im `signed`-Format), wird allerdings nicht, wie man vielleicht erwarten könnte, eine Konvertierung vom Wertebereich -32768 bis 32767 in den Wertebereich 0 bis 65535 vorgenommen, sondern der Wertebereich einfach auf 0 bis 32767 verkürzt, es gehen also wichtige Daten verloren. Leider ist es bei vtk nicht möglich, diese Konvertierung von außen zu beeinflussen.

5.2.2 Renderpipeline

Wie beim Marching-Cubes-Algorithmus findet auch hier zuerst eine Datenkonvertierung und die Anwendung eines Weichzeichnerfilters auf dem resultierenden `vtkImageData`-Objekt statt, jedoch kein *Downsampling*, da die Bildqualität beim Volumenrendering hoch bleiben soll. Anschließend wird eine, für das Volumenrendering wichtige, Transferfunktion gesetzt. Diese legt fest, wie die Materialeigenschaften Absorption (*opacity*) und Emission (*color*) von den Skalarwerten abhängen. In der medizinischen Bildgebung werden diese Funktionen üblicherweise durch die *Fensterung* (windowing) und eine Farb-*Lookup-Table (LUT)* festgelegt.[20]

Die Fensterung ist eine dreiteilige, stückweise lineare Funktion, die von den beiden Parametern `windowWidth` (Fensterbreite) und `windowCenter` (Fensterzentrum) abhängt. Sie wird folgendermaßen definiert :

$$f(x) = \begin{cases} 0 & x \leq windowCenter - \frac{windowWidth}{2} \\ 1 & x \geq windowCenter + \frac{windowWidth}{2} \\ \frac{x - windowCenter}{windowWidth} + \frac{1}{2} & sonst \end{cases} \tag{5.1}$$

Entscheidend für die Ausgabequalität ist, an welcher Stelle die Transferfunktion angewendet wird. Findet die Klassifikation vor der Interpolation direkt auf den Daten statt, werden im anschließenden Schritt nicht die skalaren Werte, sondern die umgerechneten Transparenz- und Farbwerte interpoliert. Der Nachteil dabei ist, dass bei einer nichtlinearen Transferfunktion Informationen verloren gehen[21], was sich negativ auf die Renderqualität auswirkt. Wird hingegen vor Verwendung der Transferfunktion interpoliert, können auch die interpolierten skalaren Werte klassifiziert werden, was allerdings nicht immer wünschenswert sein muss. Hat man z.B. eine Grenze zwischen zwei Objekten mit sehr unterschiedlichen Dichtewerten (z.B. Luft und Zahnschmelz), kann es auf diese Weise passieren, dass der Raum dazwischen z.B. als Zahnkrone sichtbar wird. Der Nutzer sollte daher besser selbst entscheiden, an welcher Stelle die Transferfunktion angewendet werden

[20] Hinweis: In den folgenden Kapiteln ist mit dem Begriff „Transferfunktion“ gelegentlich auch nur die Farb-LUT gemeint.

[21] Der Wertebereich der Transferfunktion ist meist kleiner als der Definitionsbereich.

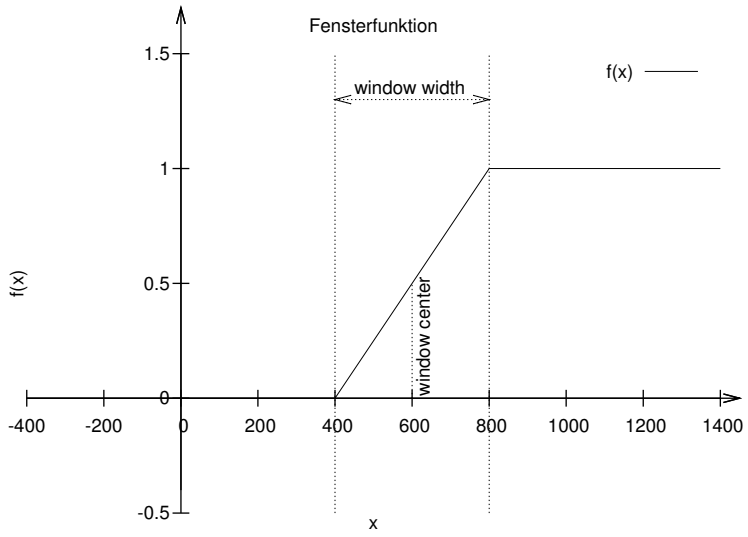


Abbildung 5.3: Graph der Fensterfunktion

soll.[22] Fortgeschrittene Anwendungen interpolieren erst und setzen dann im Anschluss eine multidimensionale Transferfunktion ein, die neben den skalaren Werten noch die *Gradientenlänge* mit berücksichtigt[23].

5.2.3 Software-Architektur

Folgendes Klassen-Diagramm zeigt die einzelnen Komponenten des (`VolumeViewer`)-Programms und wie diese in Verbindung stehen.
Die Klassenarchitektur ähnelt stark derjenigen vom `ContourViewer`. Als einziger Unterschied wurde ein neuer Controller (`VolumeInputController`) von `DefaultVtkSceneInputController` abgeleitet, der die Methoden `mousePressed` und `mouseReleased` überschreibt. D.h. bei Beginn einer Interaktion wird `VolumeView` über die Methoden `setFinalRendering(false)` und `updateMapper()` signalisiert, mit der performanteren `vtkVolumeTextureMapper3D`-Klasse zu rendern. Ist die Interaktion beendet, wird wieder zurück auf den besseren Software-Renderer `vtkFixedPointVolumeRayCastMapper` gewechselt. Ein ähnliches Verhalten ist in der Klasse `VolumeViewer` eingebaut, falls man dort das Windowing oder die Sampling-Distanz ändert.

[22] In VTK kann leider nur in der Klasse `vtkVolumeRayCastMapper` die Anwendung der Transferfunktion beeinflusst werden, alle anderen Renderer interpolieren immer vor der Transferfunktion.
[23] Engel et al.: *Real-time Volume Graphics*, 2006, S.249-273.

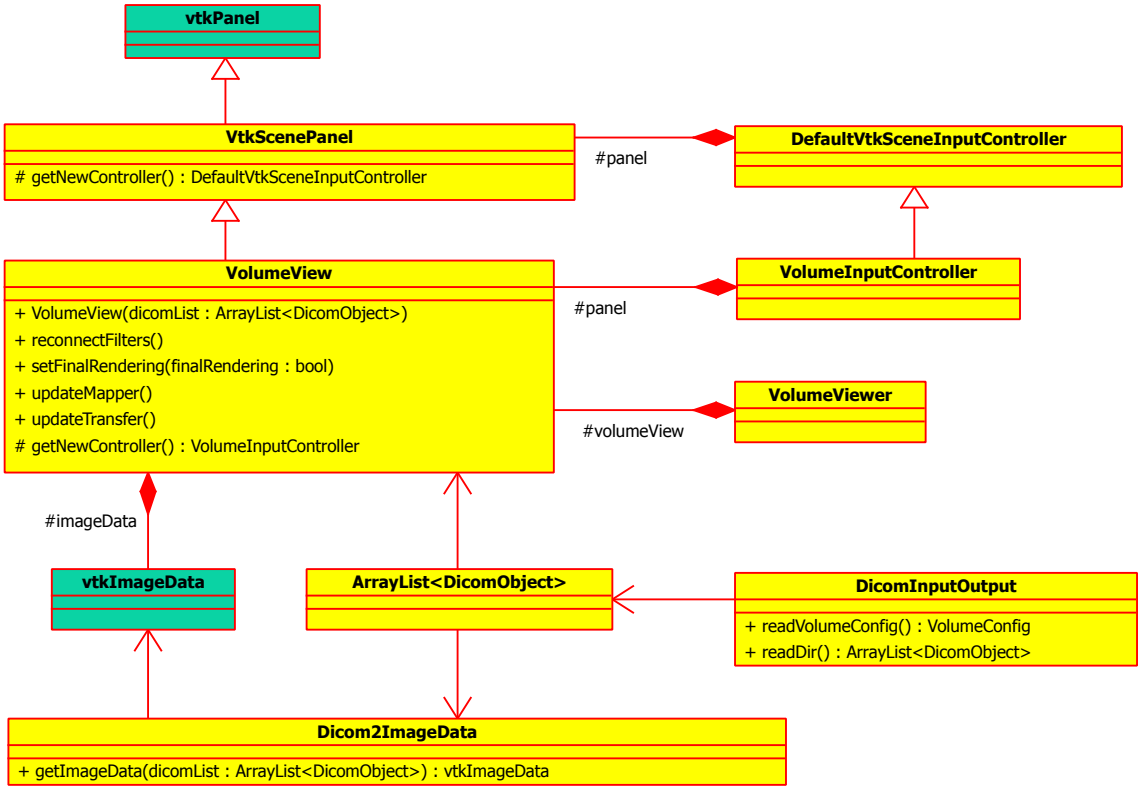


Abbildung 5.4: Klassen-Diagramm `VolumeViewer` (vereinfacht), gelb: eigene Komponenten, türkis: vtk-Komponenten

5.3 Volumenrendering mit JOGL

5.3.1 Motivation

Obwohl VTK 3D-Rekonstruktionen gut unterstützt und die Erstellung einer Anwendung stark erleichtert, bietet die Bibliothek auch einige Nachteile. Zum einen mangelt es bei in der Studienarbeit verwendeten Version (5.8)[24] an Unterstützung von aktueller Grafikhardware und zum anderen bietet die API nur begrenzte Zugriffsmöglichkeiten und es ist damit schwer die vorgefertigten Algorithmen nach eigenem Wunsch zu beeinflussen und zu optimieren. Als Beispiel dafür sind die festkodierten Texturgrenzen in der Klasse `vtkVolumeTextureMapper3D` zu nennen, die man leider nicht von außen setzen kann.
Von daher möchte ich versuchen, auf Basis der theoretischen Grundlagen von Kapitel 3.2 und einigen Techniken aus der Literatur[25], eine eigene Implementierung des Ray-Casting-Algorithmus für *Volumenrendering* in OpenGL zu erstellen, die die Fähigkeiten aktueller Grafikhardware so gut wie möglich ausnutzt. OpenGL scheint dafür hervorragend geeignet zu sein, da es zum einen die Standard-API für 3D-Programmierung ist, die auch in VTK eingesetzt wird, und zum anderen scheint es mit *JOGL* auch eine gute Anbindung an Java zu geben. Vor allem mit Hilfe der seit einigen Jahren entwickelten *Shader*-Technik ist es möglich, eigene Algorithmen direkt auf der Grafikhardware umzusetzen.

5.3.2 Datenkonvertierung

Der erste Schritt für das Rendering in OpenGL ist die Umwandlung der Daten in eine *3D-Textur*. Solange der Grafikspeicher groß genug ist, bietet eine 3D-Textur die größten Vorteile. Zum einen ist sie direkt auf der Hardware und damit schnell und direkt fürs Rendering verfügbar, andererseits steht für sie im Gegensatz zu einer alternativen Liste mit 2D-Texturen eine eingebaute trilineare Filterung zur Verfügung. Ähnlich wie bei der Implementierung in VTK habe ich die Liste der eingelesenen DICOM-Objekte zunächst in eine Liste von `ShortBuffer`-Objekten umgewandelt und anschließend als OpenGL-3D-Textur abgespeichert :

[24] Zum Zeitpunkt der Erstellung der Studienarbeit war diese aktuell. Mittlerweile wird auch unter Ubuntu Version 6.0 unterstützt.

[25] Engel et al.: *Real-time Volume Graphics*, 2006.

```
1  int[] texId = new int[1];
2  gl.glEnable(GL_TEXTURE_3D);
3  ...
4  gl.glGenTextures(1, texId, 0);
5  gl.glBindTexture(GL_TEXTURE_3D, texId[0]);
6  gl.glTexImage3D(GL_TEXTURE_3D, 0, GL_LUMINANCE16F, width, height, depth
       , 0, GL_LUMINANCE, GL_SHORT, null);
7  int zOffset = 0;
8  for (ShortBuffer dataBuf : dataBufList) {
9      gl.glTexSubImage3D(GL_TEXTURE_3D, 0, 0, 0, zOffset, width, height,
           1, GL_LUMINANCE, GL_SHORT, dataBuf);
10     zOffset++;
11 }
12 ...
13 gl.glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
14 gl.glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
15 ...
16 return texId[0];
```

Listing 5.1: Erzeugung einer 3D-Textur in OpenGL

Erläuterung:
Zuerst wird ein Textur-Objekt erzeugt und als aktuelles Ziel gebunden (1-5). Dann wird zunächst Speicher für eine leere Textur mit der entsprechenden Breite, Höhe und Tiefe reserviert, wobei als internes Speicherformat `GL_LUMINANCE16F` gewählt wird, dass heißt die Werte werden in einer Komponente mit 16-Bit-Genauigkeit abgelegt. Insgesamt werden $width \cdot height \cdot depth \cdot 2$ Byte reserviert (6). Im darauf folgenden Schritt (7-11) werden die Daten mit dem Befehl `glTexSubImage3D` an die Grafikkarte übermittelt und an das aktuelle Textur-Objekt gebunden. Die beiden vorletzten Parameter geben das Format der Quelldaten an. In den Zeilen 13 und 14 wird die trilineare Filterung aktiviert.

5.3.3 Basis-Rendering

Erster Renderpass

Der Algorithmus für das Ray Casting (Kapitel 4.2.2) benötigt zuerst die genauen Volumeneintritts- oder Volumenaustrittspunkte der Strahlen und die Strahlrichtung. Dazu wird in einem ersten Renderpass mit Hilfe eines Shaders die Rückseite des Volumenquaders mit den transformierten Texturkoordinaten als Farbwert in eine Textur gerendert. Die RGB-Tripel dieser Textur repräsentieren die Austrittspunkte der Strahlen und dienen als Eingabe für den zweiten Renderpass.

```
1 varying vec3 texCoord;
2 void main() {
3     gl_FragColor = vec4(texCoord, 1.0f);
4 }
```

Listing 5.2: Fragmentshader für ersten Renderpass

Zweiter Renderpass

Im zweiten Renderpass findet das eigentliche Ray Casting statt (Listing 5.2). Gerendert wird derselbe Quader wie aus dem ersten Pass, diesmal allerdings nur mit der Vorderseite. Als Texturen werden die persistente 3D-Textur mit den Daten `volTex` und die Hilfstextur aus dem ersten Pass mit den Strahlaustrittspunkten `backTex` gebunden. Zusätzlich werden Texturen mit den Windowing-Parametern `winTex` und der (vorintegrierten) Farb-LUT `transferTex` übergeben[26]. Als weiteren Eingabeparameter erhält der Shader die Texturkoordinate der Vorderseite `texCoord`, welche gleichzeitig den Eintrittspunkt des Strahls in den Quader darstellt. Aus dem Austritts- und Eintrittspunkt lässt sich leicht die Richtung `dir` des Sichtstrahls berechnen (Zeile 20-25).

Die Hauptschleife für das Ray Casting befindet sich in den Zeilen 32 bis 42.[27] Zuerst wird für den Texturwert der aktuellen Strahlposition `rayPos` die Windowingfunktion und die Farb-LUT angewendet (33-34). Dann wird entsprechend der Kompositionsformel 4.8 das Volumenrenderingintegral berechnet (36-37). Mit Hilfe des Parameters `alpha` (`decay`) kann zusätzlich die Transparenz des Materials beeinflusst werden. Je kleiner `decay` ist, umso durchscheinender wird das Volumen.
Ist der aufaddierte Deckwert `sumColor.a` in Zeile 38 nahe 1, kann die Berechnung frühzeitig abgebrochen werden, da `1-sumColor.a` dann nahezu 0 ist und sich keine Änderungen mehr in der Formel ergeben (Early-Ray-Termination), ansonsten wird die nächste Strahlposition berechnet und mit der Integrationsberechnung fortgefahren (40).

Zum Schluss wird das Ergebnis in den Framebuffer geschrieben (44-45).

[26] Näheres dazu in dem nächsten Kapitel
[27] vergleiche mit Kapitel 4.2.2

```
#version 130

uniform sampler3D volTex;        /* Volume texture */
uniform sampler2D backTex;       /* Ray escape points */
uniform sampler2D winTex;        /* Windowing texture */
uniform sampler2D transferTex;   /* Preintegrated transfer function (
    color LUT) */
in vec3 texCoord;                /* Ray entry point */
uniform int screenWidth;         /* Screen width */
uniform int screenHeight;        /* Screen height */
uniform float sliceStep;         /* Sampling distance */
uniform float alpha;             /* Alpha correction */
out vec4 gl_FragColor;           /* Final render color */
...
void main() {
    if (gl_FrontFacing) discard;

    vec3 rayStart=texCoord.xyz; /* Start point */
    vec3 rayPos=rayStart;       /* Current ray position */

    vec2 tc=vec2(gl_FragCoord.x/screenWidth,
                 gl_FragCoord.y/screenHeight);
    vec3 dir=texture(backTex, tc).rgb-rayStart.xyz;
    float dirLength=length(dir);
    float steps=floor(dirLength/sliceStep);
    vec3 moveDir=normalize(dir)*sliceStep;

    vec4 sumColor=vec4(0.0f);   /* Summed output color */
    vec2 tfCoord=vec2(0.0f);    /* Position in transfer texture */
    vec4 tfColor=vec4(0.0f);    /* Color of transfer texture */
    float decay=alpha*sliceStep*600;

    for (int i=0; i<steps; ++i) {
        tfCoord.y=getWindowed(rayPos);
        tfColor=texture2D(transferTex, tfCoord);

        sumColor.rgb=sumColor.rgb+(1-sumColor.a)*tfColor.rgb*decay;
        sumColor.a=sumColor.a+(1-sumColor.a)*tfColor.a*decay;
        if (sumColor.a>=0.95) break;

        rayPos+=moveDir;
        tfCoord.x=tfCoord.y;
    }

    gl_FragColor=sumColor;
    gl_FragColor.a=1.0f;
}
```

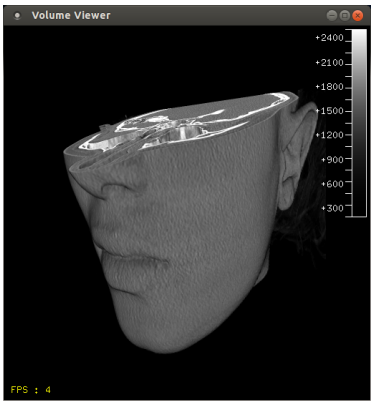
Listing 5.3: Fragmentshader für zweiten Renderpass (Basis-Algorithmus)

5.3.4 Windowing und Transferfunktion

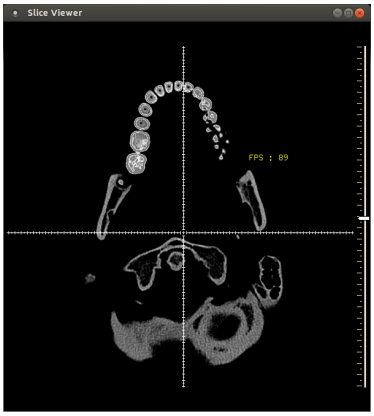
Das *Windowing* dient dazu, die in einem großen Wertebereich erstellten Daten (meist 16 Bit) auf einen kleineren Darstellungsbereich abzubilden, etwa auf 8 Bit bei normalen Monitoren oder 12 Bit bei besseren, speziell für den Einsatz im medizinischen Bereich vorgesehenen Bildschirmen. Die Abbildung wird durch eine stückweis lineare Windowingfunktion festgelegt, die eindeutig durch die Parameter `windowCenter` und `windowWidth` (Position und Breite des sichtbaren Bereiches) festgelegt wird.[28]

Je nach Einstellung können z.B. bei CT-Daten durch das Windowing unterschiedliche Körperregionen sichtbar gemacht werden, etwas Haut, Muskeln oder Knochen.

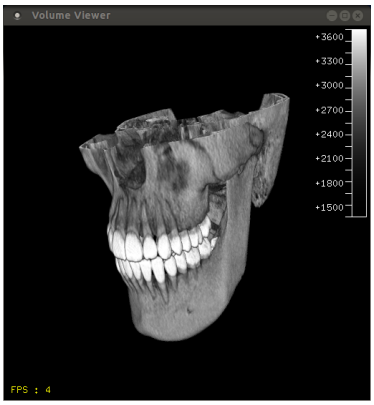
(a) Orthogonalsicht Haut



(b) 3D-Ansicht Haut



(c) Orthogonalsicht Knochen



(d) 3D-Ansicht Knochen

Abbildung 5.5: eingestelltes Windowing für die Ansicht von Haut, Knochen und Zähnen

[28] Näheres dazu im Kapitel 5.2.2

Zur Realisierung des Windowings wird für jedes Einzelbild eine eigene Windowingfunktion angewendet. Dazu werden in einer Textur für jedes Bild jeweils die Parameter `windowCenter` und `windowWidth` abgespeichert und mit an den Shader für den zweiten Renderpass übergeben (`winTex`). Die Parameter sind dabei, wie die Werte in der 3D-Textur, als 16 Bit-`short` abgespeichert. Die Textur wird anfangs mit den Original-Windowing-Werten aus den DICOM-Headern initialisiert, i.A. müssen die Bilder nicht unbedingt dieselben Windowing-Parameter besitzen. In der Anwendung hat der Nutzer die Möglichkeit, das Windowing auf ein einzelnes oder auf alle Bilder anzuwenden und auf die Originalwerte zurückzusetzen.

```
1  float getWindowed(in vec3 texPos) {
2      float windowCenter = texture2D(winTex, vec2(0, texPos.z)).a * 2.0f;
3      float windowWidth = texture2D(winTex, vec2(1, texPos.z)).a * 2.0f;
4
5      float value = (0.5f + (texture(volTex, texPos).r - windowCenter)
6                     / windowWidth);
7
8      if (value < 0.0f) value = 0.0f;
9      if (value > 1.0f) value = 1.0f;
10
11     return value;
12 }
```

Listing 5.4: Anwendung von Windowing im Fragmentshader

Im Shader werden anhand der Z-Koordinate[29] der aktuellen Texturposition `texPos` die Windowing-Werte bestimmt. Die Windowing-Textur bietet darüber hinaus den Vorteil, dass Windowing-Werte benachbarter Slices automatisch interpoliert werden. Die Fensterwerte werden dann anhand der Gleichung 5.1 auf den aktuellen Wert in der Volumentextur angewendet und das Ergebnis weitergereicht (5-8).

Vorintegrierte Transferfunktion

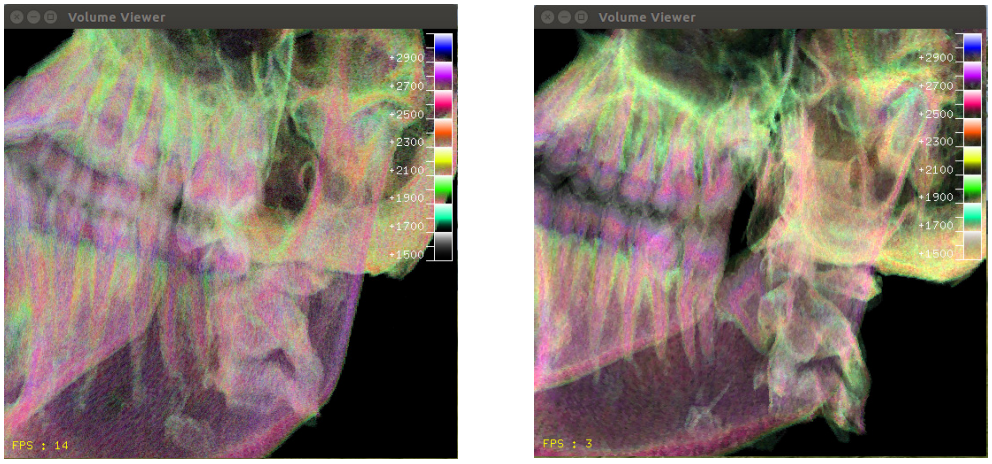
Als nächstes folgt die Anwendung der ***Transferfunktion***, die die Transparenz und den Farbwert und damit letztendlich die Ausgabe bestimmt. Wichtig für die Qualität der Ausgabe ist, dass die Transferfunktion *post-interpolativ* ist, d.h. dass sie nach der Interpolation der 3D-Textur stattfindet.[30] Eine weitere Qualitätsteigerung kann erreicht werden, indem man die Transferfunktion vorintegriert. Vor allem wenn die Transferfunktion eine hohe Frequenz besitzt, kann dies zu unerwünschten Alias-Effekten führen, die mit der Vorintegration verringert werden, ohne dass man die Sampling-Distanz verkleinern muss.

Die Idee bei der Vorintegration ist es, beim Volumenrenderingintegral die Integration der Funktionen für die Skalarwerte und für die Absorptions- und Emissionswerte zu trennen. Da letztere unabhängig von den skalaren Werten sind, kann die Integration dieser schon im Vorhinein berechnet werden. Wichtig dabei ist, dass die Frequenz der Absorptions- und Emissionsfunktion unabhängig von der Frequenz der Skalarwertfunktion wird.

[29] momentaner Slice (Texturscheibe)
[30] siehe Kapitel 5.2.2

(a) postinterpolativ (b) vorintegriert

Abbildung 5.6: Unterschied bei Anwendung einer niederfrequenten Transferfunktion (ohne Alpha-Blending, Sampling-Distanz 1.28)



(a) postinterpolativ (b) vorintegriert

Abbildung 5.7: Unterschied bei Anwendung einer hochfrequenten Transferfunktion (mit Alpha-Blending, Sampling-Distanz 1.71)

Die Vorintegration der Transferfunktion wurde genau wie von Klaus Engel[31] beschrieben implementiert. Die Grundidee ist, für die eindimensionale Transferfunktion (eine Farb-LUT mit Alpha-Kanal für die Absorptionsfunktion) das Integral für alle möglichen Wertepaare zu berechnen und dieses in einer 2D-Textur abzuspeichern. Beim Rendering wird dann für das vorberechnete Integral nur ein Texturzugriff benötigt. Als Lookup-Parameter dienen der aktuelle und der im vorherigen Raytracing-Schritt berechnete Windowing-Wert.

[31] Engel et al.: *Real-time Volume Graphics*, 2006, S.96-100.

Es hat sich gezeigt, dass eine vorintegrierte Transferfunktion meist bessere Bildergebnisse liefert, d.h. dass man auch bei größeren Sampling-Distanzen ohne Alias-Effekte auskommt.

5.3.5 Gauß-Filter

Wie auch beim Rendering mit VTK, dient die Anwendung eines *Weichzeichnerfilters* auf die Originaldaten zur Erzeugung von „glatteren“ Oberflächen und führt besonders bei der Darstellung von Knochenmaterial oder bei Bildserien mit geringer Auflösung zu einer besseren Ausgabequalität. Ein Weichzeichnerfilter ist auch gut bei der Berechnung von Gradienten geeignet, die in einer Beleuchtungsszene (und damit für eine plastische Darstellung) benötigt werden. Der Nachteil beim Weichzeichnerfilter ist, dass Details verloren gehen können, die gerade bei wissenschaftlichen Auswertungen von Interesse sind[32], von daher sollte es die Möglichkeit geben, den Weichzeichnerfilter bei der Darstellung auf Wunsch zu deaktivieren.

Der Weichzeichnerfilter kann auf zwei verschiedene Arten angewendet werden. Einmal „statisch“, d.h. dass man berechnet aus dem originalen Bildquader einmalig einen zweiten Bildquader, der zusätzlich im Speicher abgelegt wird. Der originale Bildquader kann dabei nicht überschrieben werden, da möglicherweise weiterhin auf die Originalwerte zugegriffen werden muss. Eine andere Möglichkeit wäre eine Berechnung „on-the-fly“, d.h. eine Neuberechnung bei jedem Rendervorgang, was jedoch zu rechenintensiv und daher nicht umsetzbar ist.

(a) ohne (b) mit Gauß-Filterung

Abbildung 5.8: Anwendung des Gauß-Filters

Weiterhin muss beachtet werden, an welcher Stelle das Windowing eingesetzt wird. Korrekterweise müsste es noch vor dem Weichzeichnerfilter auf den Originaldaten angewendet werden, eine Änderung der Windowing-Werte hätte dann allerdings stets eine Neuberechnung des Weichzeichnerfilters zur Folge und würde sich nachteilig auf die Interaktivität auswirken. Von daher müsste man den Anwender selbst entscheiden lassen, in welcher Reihenfolge Windowing und Weichzeichnerfilter angewendet sollen und welche Einbußen er dafür in Kauf nehmen will.

[32] z.B. eingeschlossene Luftlöcher in einer Zahnfüllung

Die Implementierung erfolgt mit Hilfe eines „Volumenbuffers“ und versucht damit, die schnellen (parallelen) Rechenfähigkeiten der Grafikkarte auszunutzen. Dazu wird das Ausgabe-Volumen anhand eines Fragment-Shaders scheibenweise in ein Framebuffer-Objekt geschrieben und als 3D-Textur abgelegt.

Für den Filterkernel $F_{3\times3\times3}$ mit $i, j, k \in \{-1, 1\}$ wurden folgende Koeffizienten gewählt:

$$F_{0,0,0} = \frac{1}{64} \cdot 8 \qquad F_{i,j,k} = \frac{1}{64} \cdot 1$$
$$F_{i,0,0} = \frac{1}{64} \cdot 4 \qquad F_{i,j,0} = \frac{1}{64} \cdot 2$$
$$F_{0,j,0} = \frac{1}{64} \cdot 4 \qquad F_{i,0,k} = \frac{1}{64} \cdot 2$$
$$F_{0,0,k} = \frac{1}{64} \cdot 4 \qquad F_{0,j,k} = \frac{1}{64} \cdot 2$$

```
1  ...
2  uniform sampler3D volTex;   /* originale Volumentextur */
3  //uniform sampler2D winTex; /* Textur mit Windowing-Werten */
4
5  uniform float xStep;        /* Schrittweiten (normal 1 Pixel) */
6  uniform float yStep;
7  uniform float zStep;
8
9  in vec3 tc;                 /* aktuelle Texturkoordinate */
10
11 out vec4 gl_FragColor;      /* Ausgabewert */
12 ...
13 float getValue(in vec3 texPos) {
14     return texture(volTex, texPos).r;
15 }
16
17 void main() {
18     float value=0.0f;
19
20     value+=0.5*getValue(tc); /* Gewichtung der Texturwerte
21                                 (nach Gauss) */
22     value+=0.25*getValue(vec3(tc.x, tc.y,tc.z-zStep));
23     ...
24     value+=0.125*getValue(vec3(tc.x,tc.y-yStep,tc.z-zStep));
25     ...
26     value+=0.0625*getValue(vec3(tc.x-xStep,tc.y-yStep,tc.z-zStep));
27     ...
28     value/=4;                /* Normierung */
29     //gl_FragColor.r = getWindowed(tc, value); /* Windowing */
30     gl_FragColor.r = value;
31 }
```

Listing 5.5: Anwendung des Gauß-Filters

Erläuterung:
Die Anwendung des Gauß-Filters wird in einem eigenen Fragmentshader vorgenommen, der als Eingabe die originale 3D-Textur erhält (2). Der Shader erhält als Eingabe auch die variablen Schrittweiten in x-, y-, und z-Richtung (5-7), die normalerweise einen Pixel betragen. Die Hilfsfunktion `getValue` liefert für die aktuelle Texturposition `texPos` den entsprechenden Wert in der 3D-Textur (13-15). Im Hauptprogramm werden die benachbarten Texturwerte entsprechend der Gauß-Filterung gewichtet (20-26), aufaddiert und normiert (28). Alternativ kann auf den gefilterten Gauß-Wert das Windowing angewendet werden (29).

5.3.6 Gradienten und Beleuchtung

3D-Objekte wirken erst realistisch, wenn sie einer Beleuchtung ausgesetzt werden und man Licht und Schatten an ihnen wahrnimmt. Eine Kugel würde z.B. ohne Beleuchtung nur wie ein flacher Kreis wirken, da man keine Tiefe mehr wahrnehmen würde. Andererseits kann man anhand der Beleuchtung auch Rückschlüsse auf die Materialeigenschaften ziehen. So hat ein Plastikball mit einer sehr glatten Oberfläche immer einen typischen hellen Punkt, der die Lichtquelle reflektiert und als Glanzlicht bezeichnet wird.

Gradientenberechnung

Wichtige Voraussetzung für die Erstellung einer Beleuchtungsszene ist die Erzeugung von *Normalen*. Dabei handelt es sich um normierte Vektoren, die genau senkrecht zur Oberfläche eines Objektes stehen. Anhand der Richtung einer Normalen und der Richtung, aus der die Lichtquelle stammt, kann bestimmt werden, ob ein Oberflächenpunkt angestrahlt wird oder sich auf der lichtabgewandten Seite befindet.

In Volumenobjekten werden Normalen mit Hilfe von *Gradienten* berechnet. Die Grundidee dabei ist, die Differenz benachbarter Voxel zu betrachten und daraus einen Einheitsvektor zusammenzusetzen. Je nach Qualität können dabei nur drei senkrecht stehende Voxel (z.B. oben, vorne, rechts) oder, indem man auch die Kanten und Ecken berücksichtigt, alle 26 Nachbarvoxel in die Berechnung einbezogen werden. Hat man die Möglichkeit, das Volumen zu interpolieren, kann man sogar beliebige Umgebungswerte[33] benutzen. Bei der Normierung muss beachtet werden, dass durch Rechenungenauigkeiten auch Vektorlängen von Null oder nahe Null existieren. In dem Fall handelt es sich vielleicht gar nicht um einen Oberflächenpunkt.

Steht genügend Speicher zur Verfügung, lassen sich die Gradienten in einer eigenen 3D-Textur abspeichern, wobei beachtet werden muss, dass man drei Komponenten pro Voxel und eine ausreichende Speichergenauigkeit benötigt. Alternativ kann der Gradient in Polarkoordinaten umgerechnet werden, die man nur noch in zwei Komponenten abspeichert. Allerdings müssen diese später wieder zurückgerechnet werden, was zusätzliche Rechenzeit in Anspruch nimmt. Eine andere Möglichkeit wäre eine Berechnung „on-the-fly“, also pro Rendervorgang. Dabei müssen nicht alle Gradienten, sondern nur die sichtbarer Voxel bestimmt werden. Ein Vorteil wäre, dass Gradienten für interpolierte Volumenwerte berechnet werden könnten, während man sonst die vorberechneten Gradienten interpoliert, was ungenauer ist. Ein weiterer Nachteil der ersten Variante wäre, neben dem höheren Speicherbedarf, eine ständige Neuberechnung der Gradiententextur nach Änderung der Windowing-Werte, da die Gradientenberechnung nur nach dem Windowing sinnvoll ist.

[33] mit beliebiger Entfernung zum Ausgangsvoxel

```
1  vec4 getNormal(in vec3 texPos) {
2      vec4 result;                        /* Berechnete Normale */
3
4      vec3 sample1; vec3 sample2;         /* Speicher fuer die 6
5                                             orthogonalen Nachbarvoxel */
6      sample1.x=getWindowed(vec3(texPos.x+xStep,texPos.y,texPos.z));
7      sample1.y=getWindowed(vec3(texPos.x,texPos.y-yStep,texPos.z));
8      sample1.z=getWindowed(vec3(texPos.x,texPos.y,texPos.z+zStep));
9
10     sample2.x=getWindowed(vec3(texPos.x-xStep,texPos.y,texPos.z));
11     sample2.y=getWindowed(vec3(texPos.x,texPos.y+yStep,texPos.z));
12     sample2.z=getWindowed(vec3(texPos.x,texPos.y,texPos.z-zStep));
13
14     result.xyz=sample2-sample1;              /* Differenz */
15     result.a=length(result.xyz)/sqrt(3);     /* Skalierung */
16     if (result.a>=gradientLimit*0.2)         /* Laenge pruefen */
17         result.xyz = normalize(result.xyz); /* Normierung */
18     else
19         result=vec4(0.0f);                   /* Nullvektor */
20
21     return result;
22 }
```

Listing 5.6: Shaderfunktion zur Gradientberechnung „on-the-fly“ nach dem Windowing

Erläuterung:
Im Fragmentshader wird mit Hilfe der Funktion `getNormal()` aus der aktuellen Texturposition `texPos` der Gradient berechnet. Da die Differenz der sechs orthogonalen Nachbarvoxel betrachtet wird, werden zur Speicherung zwei Vektoren mit je drei Komponenten benötigt (4). Auf die Werte der Nachbarvoxel wird erst das Windowing angewendet (6-12).[34] Die Differenz wird in einem neuen Vektor gespeichert (14). Danach erfolgt die Prüfung der Gradientenlänge (15-19).[35] Ist der Gradient zu kurz (19), findet keine Normierung statt, stattdessen wird der Nullvektor zurückgegeben.

Beleuchtung

Für die Beleuchtung in 3D-Szenen existieren verschiedene Beleuchtungsmodelle. In OpenGL wurde bisher als einziges das Blinn-Phong-Modell (1977) unterstützt, dass sich qualitativ nur wenig von dem Phong-Modell (1975) unterscheidet, sich dafür aber schneller berechnen lässt. Beim Blinn-Phong-Modell werden ambiente, diffuse und Glanzlichtanteile berechnet, die allgemeine mathematische Gleichung lautet:

$$C_{out} = C_{amb} \cdot L_{amb} + C_{diff} \cdot L_{diff} \cdot max(< N, L >, 0) + C_{spec} \cdot L_{spec} \cdot max(< N, H >, 0)^{\alpha} \quad (5.2)$$

mit:

- $H = \dfrac{L + E}{\|L + E\|}$

[34] siehe auch Kapitel 5.3.4

[35] In Zeile 15 wird `result.a` auf den Bereich [0,1] skaliert. `getWindowed()` liefert Werte zwischen 0 und 1, d.h. die Länge des Vektors `result.xyz` beträgt maximal $\sqrt{3}$.

- C_{out} = Ausgabewert
- $C_{amb}, C_{diff}, C_{spec}$ = Materialfarbe (ambienter, diffuser und Glanzlichtanteil), Wert des Voxels nach Anwendung der Transferfunktion (wobei $C_{amb} = C_{diff} = C_{spec}$)
- $L_{amb}, L_{diff}, L_{spec}$ = Lichtfarbe (ambienter, diffuser und Glanzlichtanteil)
- N = Normale (Gradient)
- L = Lichtvektor, Vektor von Lichtquelle zu Objekt
- E = „Eyevektor“, Vektor vom Betrachter zum Objekt
- α = Glanzfaktor, beeinflusst Radius und Stärke des Glanzlichtes

Wegen der Vereinfachung des Beleuchtungsmodells werden einige physikalische Gesetze nicht beachtet, z.B. dass verdeckte Stellen des Volumenobjektes nicht vom Lichtstrahl erreicht werden können (Schattenwirkung) und dass das Licht innerhalb des Volumens abgelenkt wird oder einige seiner Eigenschaften verliert.

Es ist zu beachten, dass in der festen OpenGL-Rendering-Pipeline immer nur eine Per-Vertex-Berechnung unterstützt wurde, welche in Gitternetzmodellen mit wenig Eckpunkten für die Berechnung von Glanzlichtern unzureichend ist. Mit Einführung der Shader-Technik ist es nun möglich, eine Per-Pixel-Berechnung für die Glanzlichtkomponenten durchzuführen und auch alternative Beleuchtungsmodelle zu implementieren.

(a) ohne (b) mit Beleuchtung

Abbildung 5.9: Beleuchtung mit berechneten Gradienten

Beleuchtungsberechnungen bei der 3-D-Visualisierung medizinischer Bilder machen vor allem bei der Darstellung von Hautflächen, Knochen oder Zähnen Sinn (siehe Abbildung 9.3). Für eine plastische Darstellung genügen bereits die ambienten und diffusen Anteile, deren Berechnung nicht so aufwendig ist wie bei der Glanzlichtkomponente, wo vor allem das Potenzieren mit dem Glanzfaktor viel Rechenzeit benötigt.

5.3.7 Optimierungen

Das Ziel bei Optimierungen im Volumenrendering ist es, Effizienzverbesserungen ohne große Qualitätsverluste zu erreichen. Bestimmte Optimierungsverfahren, wie „Early-Ray-Termination“ und Vorintegration der Transferfunktion, wurden bereits weiter oben besprochen. Einen großen Teil der Rechenzeit nimmt der Ray-Casting-Algorithmus ein, dessen Ausführungsgeschwindigkeit insbesondere von der Anzahl und Auflösung der Bilder, sowie von der Auflösung der Ausgabe abhängig ist. Aber auch die Struktur der Aufnahme (Größe und Dichte des Objektes) und die gewünschte Darstellungsform (Windowing, zusätzliche Effekte wie Beleuchtung) spielen eine gewisse Rolle. Da es sich um die Darstellung eines starren 3D-Objektes handelt (ohne Animationen), sollten vor allem flüssige Interaktionen sichergestellt werden. Abhängig von der Art der Interaktion müssen meist nur bestimmte Berechnungen neu ausgeführt werden. Dabei sind Kameraoperationen (Drehung, Translation und Zooming), die die Sichtperspektive verändern und Parametereinstellungen, die nur die Art der Darstellung beeinflussen, zu unterscheiden.

Beispielsweise muss die statische Gauß-Filterung, die direkt auf das Volumenobjekt angewendet wird, im Gegensatz zum „on-the-fly“-Smoothing bei Navigationsoperationen nicht neu berechnet werden, sondern nur, wenn sich das Windowing ändert. Eine Vorberechnung der Gauß-Filterung erfordert allerdings zusätzliche Grafikspeicherkapazitäten.

Ein wichtige Feststellung für die Verbesserung der Effizienz ist, dass die finale Renderqualität nicht unbedingt mit der interaktiven Renderqualität übereinstimmen muss. Bei interaktiven Aktionen bzw. bei der Navigation ist es meist nur wichtig, die räumliche Orientierung zu bewahren. Details und zusätzliche Effekte brauchen erst im letzten Rendervorgang angezeigt werden, also wenn die Interaktion beendet ist. Bei besserer Grafikhardware sollte der Anwender aber auch immer selbst über die Ausgabequalität entscheiden können, im Idealfall stimmen finales und interaktives Rendering möglicherweise überein.

Für die Umsetzung muss das Rendering dazu auf zwei Shader aufgeteilt werden. Auch wird nicht mehr direkt auf den Bildschirm gerendert, sondern erst in einem Zwischenschritt „Render-To-Texture“ angewendet, also in eine Textur gezeichnet. Der Vorteil bei diesem Verfahren ist, dass auf eine beliebige Auflösung ausgegeben werden kann, die dann im zweiten Schritt auf die Auflösung der Anzeige skaliert wird. So kann man z.B. bei einer Ausgabeauflösung von 512x512 Pixel beim interaktiven Rendern erst in eine 256x256 Pixel große Textur rendern und diese dann auf die doppelte Größe hochskalieren. Damit müssen zwar gewisse Qualitätseinbußen hingenommen werden, die aber durch (hardwarebeschleunigte) bi- oder trilineare Texturfilterung teilweise wieder ausgeglichen werden können. Das Bild wird dadurch etwas unschärfer, lässt jedoch die räumliche Struktur noch erahnen, was für Navigationszwecke ausreichend ist. Im Gegenzug kann beim finalen Rendering z.B. auf die doppelte Größe, also in eine 1024x1024-Textur gezeichnet werden.

Aber nicht nur die Ausgabeauflösung, auch die Qualität einzelner Renderschritte kann in beiden Shadern unterschiedlich implementiert werden. So kann z.B. bei „Gradienten-on-the-fly“ und Gauß-Filterung eine unterschiedliche Anzahl von Nachbar-Voxel in die Berechnung einbezogen werden, oder einzelne Effekte wie Beleuchtung beim interaktiven Rendern ganz ausgeschaltet werden.

5.3.8 Software-Architektur

Das Programm ist in insgesamt vier unabhängige Fenster aufgeteilt: zwei für die Ausgabe und zwei für Parametereinstellungen (Abbildung 9.3). Das erste Ausgabefenster (`SliceViewer`) ist in vier gleichgroße Unterfenster aufgeteilt, von denen drei für die orthogonale Sicht vorgesehen sind. Dort hat man die Möglichkeit, die Bilder schichtweise von vorne, von der Seite oder von oben zu betrachten. Die Darstellung selbst ist mit OpenGL implementiert, was für eine flüssige Schichtnavigation der als 3D-Textur abgespeicherten Bilder sorgt. Ein Doppelklick auf einem der Sichtfenster sorgt für eine Vergrößerung auf das ganze Fenster. Die Navigation wird mit Hilfe eines Schiebereglers unterhalb des jeweiligen Unterfensters vorgenommen. Eine andere Möglichkeit zur Navigation ist ein sogenanntes 3D-Fadenkreuz. Dieses wird in jedem orthogonalen Sichtfenster als 2D-Fadenkreuz dargestellt. Wählt man damit in einem Sichtfenster einen bestimmten Punkt aus, hat das direkte Auswirkungen auf die Auswahl der Schichten in den anderen beiden Unterfenstern. Mit dem 3D-Fadenkreuz ist es möglich, einen bestimmten Punkt im 3D-Quader auszuwählen und z.B. den Datenwert abzufragen. Kleine Unterteilungen des Fadenkreuzes lassen zusätzliche Messungen zu. An den Seiten jedes Sichtfensters befinden sich darüber hinaus sogenannte „Cutter“, mit denen das Volumen eingeschränkt werden kann. Die Auswirkungen des Cutters sind im zweiten großen Ausgabefenster, dem `VolumeViewer` sichtbar.

Der `VolumeViewer` ist das eigentliche Hauptfenster. Es zeigt das gerenderte Volumen und benutzt dieselbe 3D-Textur wie der `SliceViewer`. Dazu müssen sich `SliceViewer` und `VolumeViewer` denselben OpenGL-Kontext teilen. Mithilfe der Maus kann das Volumen rotiert und gezoomt werden. Auf der rechten Seite befindet sich eine Skala, die das aktuelle Windowing und die Transferfunktion anzeigt. Alle wichtigen Informationen über das Volumen finden sich in der Klasse `VolumeObject`. Dazu gehören einerseits die Werte des aktuellen Windowings, sowie die Textur der angewendeten Transferfunktion und die Positionen des Cutters (constraint). Andererseits findet man auch die verschiedenen `VolumenBuffer`, die für die Vorberechnung des Gauß-Filters und der Gradienten notwendig sind. Die `VolumenBuffer` selbst sind nichts anderes als 3D-Texturen, die als Renderziel dienen, wobei in einem Durchlauf jede Schicht einzeln gerendert wird.

Nun gibt es einige Parametereinstellungen, die sowohl Auswirkungen auf den `SliceViewer` als auch auf den `VolumeViewer` haben, wie beispielsweise das Windowing und die Transferfunktion. Andere dagegen, wie Beleuchtungsparameter oder Qualitätseinstellungen für finales und interaktives Rendering, haben nur Auswirkungen auf den `VolumeViewer`. Von daher hatte ich mich dafür entschieden, die Parametereinstellungen auf zwei unterschiedliche Fenster aufzuteilen (`TransferController` und `VolumeController`). Somit ist auch möglich z.B. den `SliceViewer` unabhängig vom `VolumeViewer` und dessen Parametereinstellungen zu starten.

Bevor überhaupt das Programm mit seinen vier Fenstern startet, gibt es noch ein Konfigurationsfenster (`VolumeViewStartController`). Damit ist es möglich Grundinformationen über die einzulesenden Bilddaten zu erhalten (Anzahl und Auflösung) und den benötigten Speicherverbrauch zu berechnen. Auch sind hier alle Einstellungen möglich, die Auswirkungen auf den Speicherverbrauch haben, wie etwa internes Datenformat der 3D-Textur oder Vorberechnung von Weichzeichnerfilter, Gradienten und Windowing.

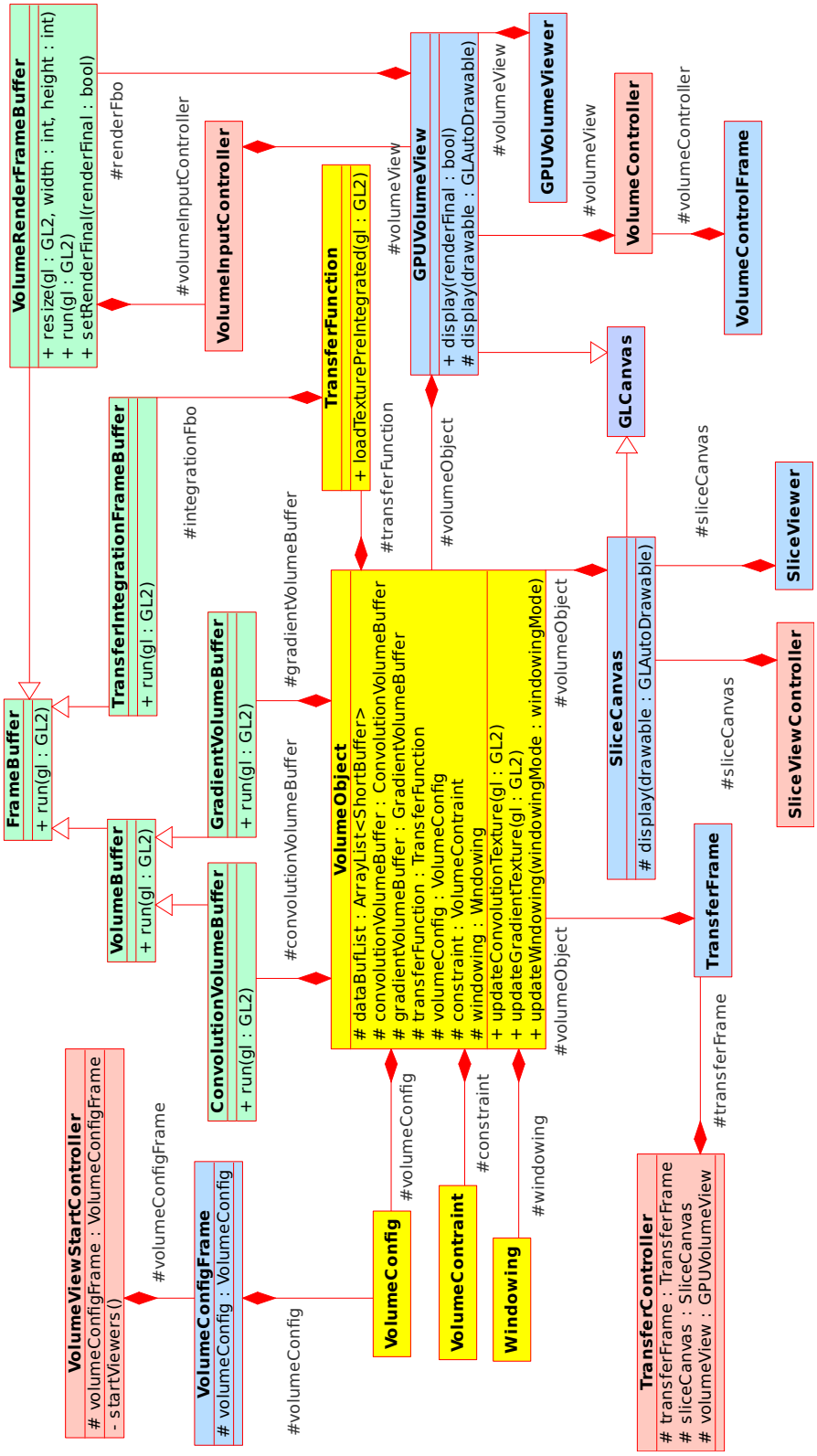


Abbildung 5.10: Klassen-Diagramm `SliceViewer` und `GPUVolumeViewer` (vereinfacht), alle Komponenten bis auf `GLCanvas` sind selbst entwickelt

Kapitel 6

Auswertung

6.1 Vergleich oberflächenbasierte Visualisierung/Volumenrendering

Es soll nun anhand der entwickelten Demoprogramme und praktischer Beispiele verglichen werden, welches der vorgestellten 3D-Rekonstruktionsverfahren in VTK am besten geeignet ist. Als ausschlaggebende Kriterien sollen dabei vor allem die gelieferte Bildqualitität, die Effizienz und Auswirkungen auf das Interaktionsverhalten berücksichtigt werden.

6.1.1 Testsysteme

Getestet wird auf zwei unterschiedlichen Testsystemen, die sich in erster Linie im Grafikprozessor unterscheiden. Das erste Testsystem besitzt einen Onboard-Grafikprozessor, das zweite eine leistungsfähigere 3D-Grafikkarte.

Testsystem T1: Prozessor(AMD Athlon LE-1640, 2,6 GHz), Hauptspeicher (1,7 GB), Grafik (GeForce 7050 PV / nForce 630a Onboard, 425 MHz, 512 MB shared memory)

Testsystem T2: Prozessor(Intel Core Duo CPU E8400, 3,00 GHz), Hauptspeicher (3,9 GB), Grafik (Quadro FX 1800, 768 MB, 275 MHZ, Memory Clock 300 MHz)

Testserie: Als Testbildserie diente der CT-Scan eines Kopfes, Auflösung 512x512 Pixel, 16Bit Genauigkeit, 166 Bilder, 83 MB Gesamtspeicher

6.1.2 Testergebnisse

Kriterium	**Isoflächen**	**Volumenrendering**
Startzeit	16 s	31 s
Anzeige Haut		
Konturstufe	500	-
Windowing	-	Center 500, Width 600
Framerate	27 FPS	7-8 FPS
Dreiecke	ca. 260.000	-
Sampledistanz	-	0,25
Weichzeichnerfilter	ja	ja
Bildqualität	glatte Oberfläche, detaillierte Ohren, Lippen, Nase, „Treppeneffekte“ kaum wahrnehmbar	etwas detaillierter, keine Sampleeffekte wahrnehmbar
nach Decimation	76 FPS, 64.930 Dreiecke, kaum sichtbare Unterschiede, Treppeneffekte weg, dafür andere kleine Artefakte	-
Latenz	2-3 Sekunden bei Änderung der Konturstufe, 10 Sekunden mit Dreiecksdezimierung	¡1s, Änderung sofort sichtbar, aber noch nicht sehr flüssig
Anzeige Knochen		
Konturstufe	1500	-
Windowing	-	Center 1500, Width 600
Framerate	16 FPS	7-8 FPS
Dreiecke	435.788	-
Sampledistanz	-	0,25
Weichzeichnerfilter	ja	ja
Bildqualität	Unterkiefer gut, Zähne zu rund und verwachsen, Sägezahneffekt bei oberen Knochenrändern	bei finalem Rendering Zähne deutlich detaillierter, Schädel wirkt insgesamt etwas realistischer, interaktives Rendering schlecht mit vielen Artefakten
optimale Anzeige Knochen		
Konturstufe	2000	-
Windowing	-	Center 2000, Width 1800
Framerate	4 FPS	6 FPS
Dreiecke	ca. 3.400.000	-
Sampledistanz	-	0,25
Weichzeichnerfilter	nein	nein
Bildqualität	optimal	optimal, äußerst realistisch
Latenz	sehr schlecht	gleichbleibend

Tabelle 6.1: Vergleich oberflächenbasierte Visualisierung/Volumenrendering auf T1

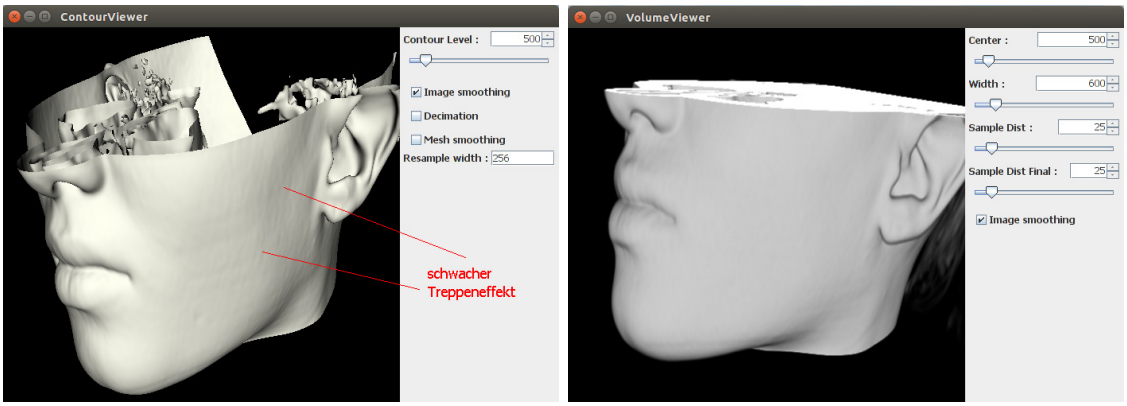
Kriterium	**Isoflächen (VTK)**	**Volumenrendering (VTK)**	**Volumenrendering (JOGL)**
Startzeit	7 s	12 s	4 s
Anzeige Haut			
Konturstufe	500	-	-
Windowing	-	Center 500, Width 600	Center 500, Width 600
Framerate	930 FPS	130 FPS	60 FPS
Sampledistanz	-	1,00	1,00
Interactive Quality	-	-	50%
Latenz	¡1s	$\sim 0s$	$\sim 0s$
Anzeige Knochen			
Konturstufe	2000	-	-
Windowing	-	Center 2000, Width 600	Center 2000, Width 600
Framerate	86 FPS	70 FPS	40 FPS
Sampledistanz	-	1,00	1,00
Interactive Quality	-	-	50%
Latenz	¡1s	$\sim 0s$	$\sim 0s$
optimale Anzeige Knochen			
Windowing	-	Center 2000, Width 2000	Center 2000, Width 2000
Framerate	-	50 FPS	15 FPS
Sampledistanz	-	0,25	0,25
Interactive Quality	-	-	100%
Latenz	¡1s	$\sim 0s$	$\sim 0s$

Tabelle 6.2: Vergleich oberflächenbasierte Visualisierung/Volumenrendering VTK/JOGL auf T2 (mit selben Parametern wie bei T1)

6.1.3 Interpretation

Testsystem T1

Der Start dauert mit 16 bzw. 31 s bei beiden VTK-Programmen relativ lange, wobei ein Großteil der Startzeit für das Einlesen der Bilder benötigt wird. Bei der oberflächenbasierten Visualisierung müssen erst die Dreiecke generiert, beim Volumenrendering die 3D-Textur erzeugt werden. Zusätzlich wird beim Volumenrendering ein Algorithmus zur Weichzeichnerfilterung angewendet, der ungefähr 8 s der Prozessorzeit benötigt. Für die Anzeige der Hautstufe ist die Framerate mit 27 FPS bei der oberflächenbasierten Visualisierung noch akzeptabel. 7 bis 8 FPS beim Volumenrendering reichen noch für die Mausnavigation, sorgen allerdings nicht mehr für ein ganz so flüssiges Rendering. Die optimale Renderqualtität wird bei beiden Verfahren nur mit dem Weichzeichnerfilter erreicht. Man erkennt die Haut als glatte Oberfläche. Ohren, Lippen und Nase sind im Detail erkennbar, beim Volumenrendering etwas detaillierter.



(a) Isofläche
(ca. 260.000 Dreiecke)

(b) Volumenrendering

Abbildung 6.1: Anzeige der Hautstufe

Bei der Isofläche ist ein ganz schwacher „Treppeneffekt“ erkennbar, welcher nach der Dreiecksdezimierung aber verschwindet.

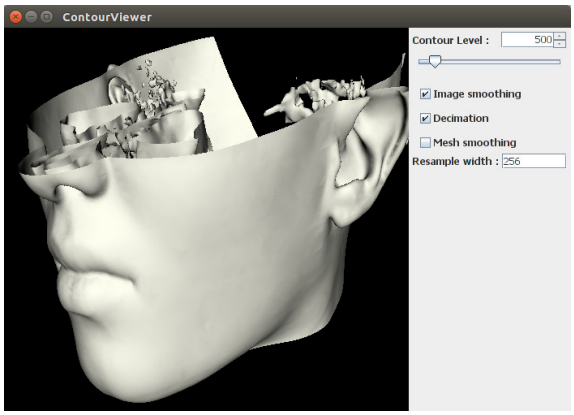
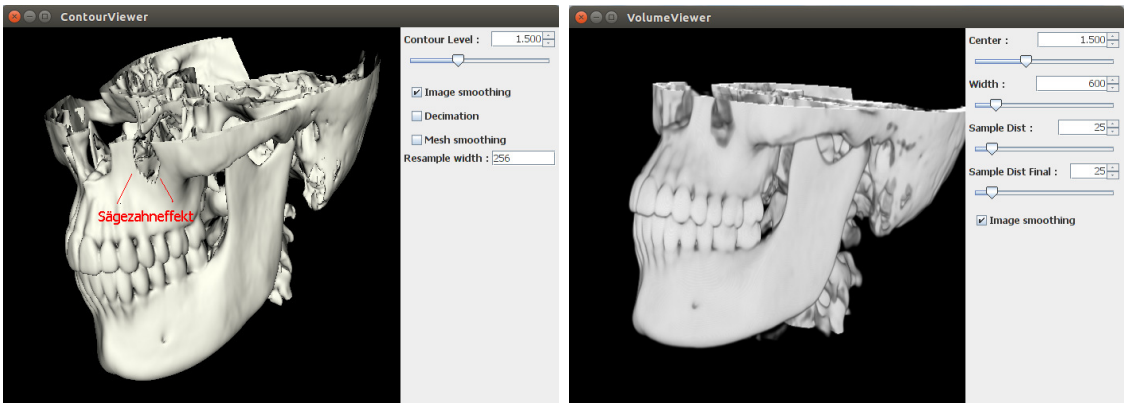


Abbildung 6.2: Isofläche nach Dreiecksdezimierung
(64.930 Dreiecke)

Die Dreiecksdezimierung sorgt für eine Verdreifachung der Framerate, ohne sich jedoch merklich negativ auf die Qualität des 3D-Modells auszuwirken. Die Verzögerungszeit beim Ändern der Konturstufe ist mit 2 bis 3 bzw. 10 s dafür relativ langsam. Sofern man die gewünschte Konturstufe vorher kennt, ist das kein Problem, fürs Ausprobieren allerdings nicht gut geeignet. Beim Volumenrendering sind Parameteränderungen deutlich schneller sichtbar, auch wenn sie noch nicht ganz flüssig spürbar sind. Die Anzeige der Knochenstufe (Konturlevel 1500) bringt kaum große Veränderungen. Da die Knochen eine komplexere Struktur als die Haut haben, steigt bei der Isofläche natürlich auch die benötigte Dreieckszahl, was zu einem Absinken der Framerate führt. Der Unterkiefer ist gut geglättet und sehr gut dargestellt, allerdings erscheinen die Zähne etwas zu rund und sind miteinander verwachsen, was wahrscheinlich auf die geringe Scanqualität zurückzuführen ist. An den oberen Knochenrändern ist ein unschöner Sägezahneffekt zu erkennen.

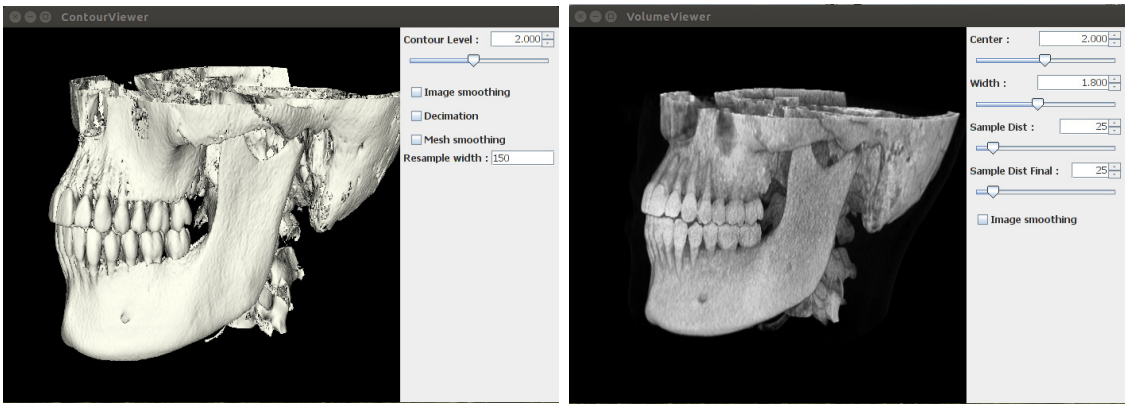


(a) Isofläche (435.788 Dreiecke)

(b) Volumenrendering

Abbildung 6.3: Anzeige der Knochenstufe

Beim Volumenrendering sind die Zähne etwas detaillierter, der Schädel wirkt deutlich realistischer. Allerdings kann die optimale Qualität nur beim finalen Rendering erreicht werden, das interaktive Rendering bleibt noch hinter den Erwartungen zurück.[1]



(a) Isofläche (ca. 3.400.000 Dreiecke)

(b) Volumenrendering

Abbildung 6.4: Knochen bei Konturstufe 2000

Die optimale Konturstufe für die Anzeige von Knochen und Zähnen scheint bei 2000 zu liegen (mit einer Fensterbreite zwischen 1600 und 2000 beim Volumerendering). Sowohl bei der Isofläche als auch beim Volumenrendering sind nun alle Details erkennbar, sofern der Weichzeichnerfilter ausgeschaltet bleibt. Bei aktiviertem Weichzeichnerfilter ist die Renderqualität der Isofläche allerdings schlechter als bei Konturstufe 1500. Die Framerate bleibt beim Volumenrendering mit 6 bis 7 FPS relativ stabil, während sie bei der Isofläche aufgrund der hohen Dreieckszahl (3.400.000) auf 4 FPS heruntergeht. Eine Dreiecksdezimierung sorgt leider für eine schlechtere

[1] da vtkVolumeTextureMapper3D die Bildserie runterskaliert, siehe Abschnitt 5.2

Modell-Qualität, während sich die Latenz dabei nicht wesentlich verbessert.

Testsystem T2

Die Startzeit hat sich bei beiden Programmen im Vergleich zu T1 mehr als halbiert, was auf die bessere Prozessorleistung zurückzuführen sein dürfte. Die Framerate für die Anzeige der Hautstufe hat sich mit 930 FPS bzw. 130 FPS merklich verbessert, was jedoch bei der Umstellung von Onboard-Chip auf 3D-Grafikkarte nicht weiter verwundert. Auch bei anderen Konturstufen fällt die Framerate nie unter 50 FPS. Die Latenzzeit für die Änderung des Windowings beträgt nun weniger als eine Sekunde, wobei bei der oberflächenbasierten Visualisierung im Gegensatz zum Volumenrendering noch eine leichte Verzögerung zu bemerken ist. Insgesamt scheinen aber beide Programme für den Anwender akzeptabel bedienbar zu sein.

Fazit

Insgesamt sind beide Verfahren für die 3D-Rekonstruktion anwendbar. Das Rendering mit Isoflächen eignet sich vor allem gut für die Darstellung glatter Oberflächen und sorgt auch bei schlechterer Grafikhardware für akzeptable Frameraten. Isoflächen eignen sich darüber hinaus gut für Effekte wie z.B. Beleuchtung und Schattierung, jedoch weniger gut, wenn es um detaillierte Darstellungen geht. Volumenrendering ist hingegen von Vorteil, wenn häufige Parameteränderungen (z.B. Windowing) oder Querschnitte erforderlich sind, da hier nicht immer wieder ein Dreiecksgitter neu berechnet werden muss. Dafür erfordert Volumenrendering mehr Grafikspeicher und ist, wenn es ums flüssige Rendern geht, eher für modernere Grafikkarten gut geeignet.

6.2 Grenzen von VTK

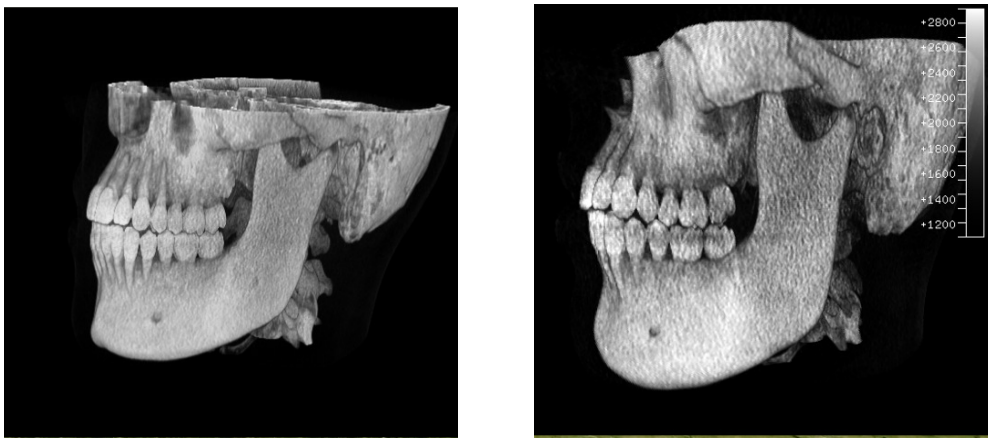
VTK wurde für die Visualisierung wissenschaftlicher Daten konzipiert und ist, wie die obigen Testergebnisse gezeigt haben, insbesondere auch für die Darstellung medizinischer Bilddaten gut geeignet. Die Bibliothek unterstützt die beiden wichtigsten 3D-Rekonstruktionsverfahren, oberflächenbasierte Visualisierung und Volumenrendering, und bietet die Möglichkeit - sofern man das Programmiermodell erlernt und verstanden hat - Visualisierungs-Anwendungen schnell und einfach zu erstellen.

Trotzdem gibt es einige Schwächen und Nachteile, die bei der Integration in die Java-Bibliothek "viskit" berücksichtigt werden sollten. Zum einen handelt es sich im Kern um eine C++-Bibliothek, d.h. auch wenn eine entsprechende Java-Anbindung zur Verfügung gestellt wird, muss man immer damit rechnen, dass der Java-Wrapper nicht auf dem selben Stand ist wie die Hauptbibliothek und einige Versionen zurückhängen kann. Des weiteren ist zu berücksichtigen, dass die Architektur von VTK Anfang der 1990er in Hinblick auf die damaligen Hardware-Gegebenheiten entworfen wurde und sich seitdem auch nicht mehr wesentlich verändert hat. Zwar soll die Bibliothek für die parallele Datenverarbeitung gut geeignet sein, doch werden die rechenintensiven Algorithmen immer noch auf dem Hauptprozessor ausgeführt und Daten im Hauptspeicher gehalten - die Möglichkeit Operationen auf der heutzutage extrem leistungsfähigen Grafikhardware auszulagern wird nicht genutzt. Die Architektur von VTK müsste, um diese Möglichkeiten optimal auszunutzen, wahrscheinlich grundlegend überarbeitet werden, was in naher Zukunft jedoch nicht zu erwarten ist.

Weiterhin ist zu beachten, dass die Klassen-Bibliotheken von VTK meist nur Grundfunktionalitäten bieten, d.h. man ist auf das beschränkt, was die API vorgibt und nicht in der Lage, diese um eigene neue Funktionalitäten zu erweitern. Als Beispiele seien hier die Klassen `vtkVolumeRayCastMapper` und `vtkVolumeTextureMapper2D` genannt, die nicht den wichtigen Datentyp `signed short` unterstützen oder die Klasse `vtkVolumeTextureMapper3D`, bei der die Maximalgröße des Texturwürfels auf 256x256x128 Voxel unveränderlich festgelegt ist.

6.3 Vergleich Volumenrendering VTK/JOGL

Als Testsystem für den Vergleich des Volumenrenderns zwischen VTK und JOGL dient das System T2, dass auch beim Vergleich von oberflächenbasierter Visualisierung und Volumenrendering verwendet wurde. Die Startzeit ist beim JOGL-Programm mit 4 s im Vergleich zu 12 s beim VTK-Programm überraschend niedrig. Dies dürfte daran liegen, dass die Daten größtenteils auf der schnellen Grafikkarte verarbeitet werden, während bei VTK noch Vorberechnungen (Verkleinerung des Datenvolumens) auf der Haupt-CPU stattfinden. Dafür ist die Framerate höchstens halb so groß, aber mit mindestens 15 FPS noch im akzeptablen Bereich. Die niedrigere Framerate dürfte auf die aufwendigen Berechnungen zurückzuführen sein, die im Shader durchgeführt werden. Dafür schneidet die Qualität der Ausgabe beim JOGL-Programm deutlich besser ab, sowohl die interaktive als auch die finale Renderqualität. Die Latenz bei Änderung der Windowing-Werte spielt bei beiden Programmen im Vergleich zur oberflächenbasierten Visualisierung kaum eine Rolle.



(a) VTK (b) JOGL

Abbildung 6.5: Vergleich Volumerendering (Konturstufe 2000)

Kapitel 7

Aussichten

Der in JOGL entwickelte Volumenrenderer bietet bereits die wichtigsten Basis-Funktionalitäten, die für eine 3D-Rekonstruktion medizinischer Bilddaten erforderlich sind. Dazu gehört das Anwenden von Windowing, die freie Festlegung der Sichtposition, Zooming und die Anwendung einer (vorgegebenen) Transferfunktion. Wünschenswert wären vielleicht noch Werkzeuge zur Messung und Auswertung der Bilddaten, etwa über dreidimensionale ROIs (Regions Of Interest). Auch die Implementierung von Segmentierungsverfahren, etwa zur Identifizierung eines Tumors, sind denkbar. Bisher lassen sich bestimmte Bereiche des gescannten Körpers ansatzweise mit einer ensprechenden einfachen Transferfunktion klassifizieren, bessere Ergebnisse würden sogenannte „multidimensionale Transferfunktionen“[1] bieten.

Ein realistischeres 3D-Modell lässt sich bisher dank der Implementierung eines einfachen Beleuchtungsmodells erzeugen. Um ein noch höheres Maß an Realismus zu erreichen, wäre die Erzeugung von Schatten oder Ambient Occlusion erforderlich. Allerdings müssten wegen des höheren Berechnungsaufwandes Einbußen bei der Laufzeit hingenommen werden und es ist fraglich, ob mehr Effekte und ein höherer Realismus unbedingt für die Auswertung der Bilddaten im Zuge medizinischer Studien erforderlich sind.

Über ein zusätzliches Fenster mit der klassischen 3-Seiten-Ansicht (MPR) ist es bereits möglich, orthogale Schnitte des Volumenquaders zu erzeugen. Aus Anwendersicht wäre eine freie Auswahl der Schnittebenen (d.h. beliebige Lage und Drehung) sicherlich wünschenswert.

Im Bereich Performance-Verbesserung wären Verfahren wie etwa *Empty-Space Leaping* wahrscheinlich noch sehr interessant.

[1] Engel et al.: *Real-time Volume Graphics*, 2006, S.249-273.

Kapitel 8

Zusammenfassung

Das Thema dieser Studienarbeit war die dreidimensionale Visualisierung von DICOM-Bildern mit Hilfe der C++-Klassenbibliothek VTK innerhalb der medizinischen Java-Visualisierungsbibliothek viskit. Dazu sollte erörtert werden, welche Möglichkeiten VTK bietet und welche Vor- und Nachteile existieren. Es konnte gezeigt werden, dass die Anbindung von VTK an viskit prinzipiell möglich ist und dass sich Serien von DICOM-Bilder damit dreidimensional rekonstruieren lassen. Es wurden zwei Testanwendungen entwickelt, mit denen sich die beiden wichtigsten Rekonstruktionsarten (Oberflächenbasierte Visualisierung und Volumenrendering) demonstrieren lassen. Darüber hinaus wurden aber auch die Grenzen der C++-Bibliothek aufgezeigt. Eine alternative Implementierung mit Hilfe des Java-OpenGL-Frameworks JOGL nutzt die Möglichkeiten moderner Grafikhardware deutlich besser aus und sorgt damit sowohl für eine bessere Performance als auch Renderqualität.

Kapitel 9

Abbildungen

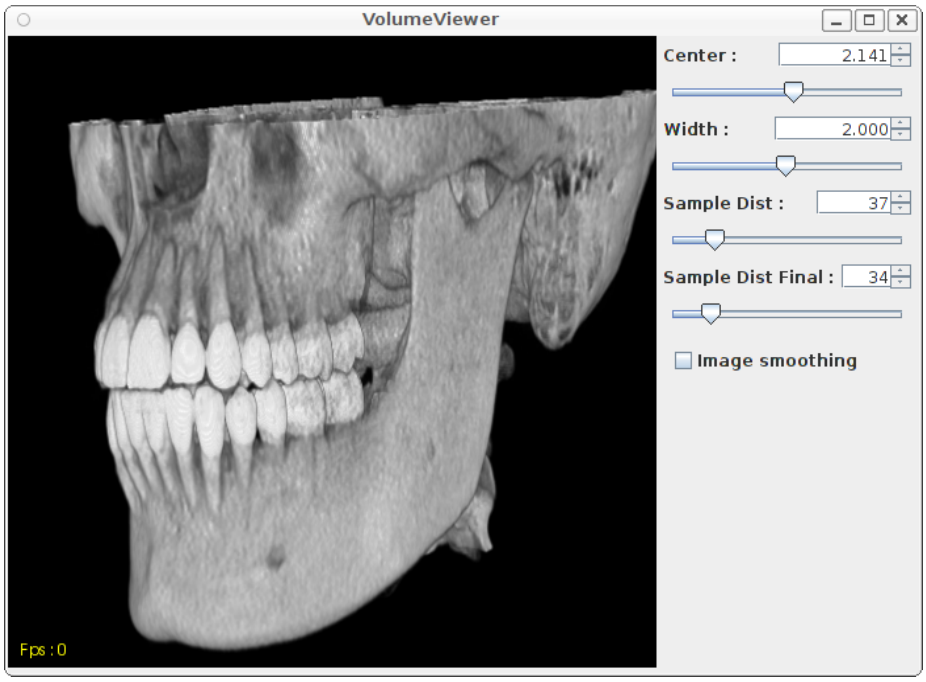


Abbildung 9.1: Demo-Anwendung mit Einstellmöglichkeiten für Windowing, Sample-Distanz und Weichzeichnerfilter. Bei interaktiven Aktionen wird der hardwarebeschleunigte Volumenrenderer (`vtkVolumeTextureMapper3D`)) aktiviert, der mit einer Framerate von 5 bis 30 FPS läuft. Bei Beendigung wird wieder zurück auf den softwarebasierten Ray Caster (`vtkFixedPointVolumeRayCastMapper`) geschaltet.

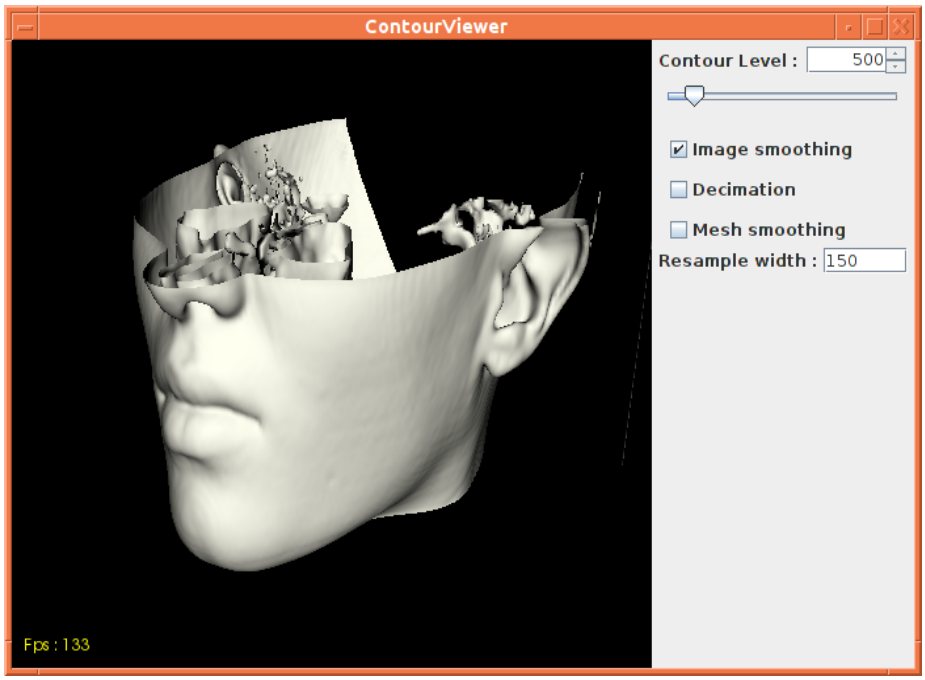


Abbildung 9.2: Demo-Anwendung für 3D-Rekonstruktionen auf Grundlage von Isoflächen. Angezeigt wird derselbe Datensatz mit Konturstufe 500 (Haut).

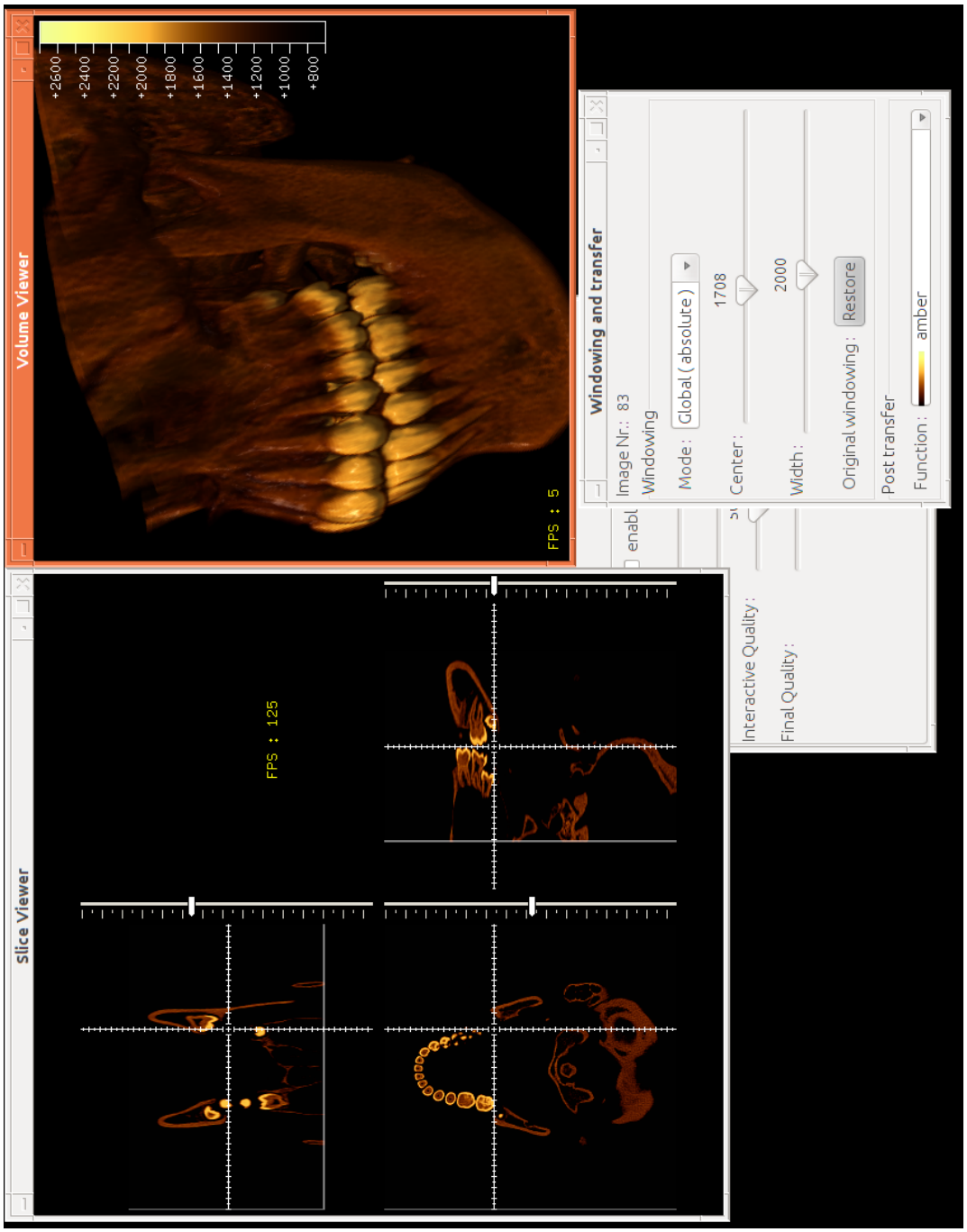


Abbildung 9.3: JOGL-Implementierung des Volume-Viewers. Die Testanwendung besteht aus insgesamt vier Fenstern (Volume-Viewer, orthogonale 3-Seitenansicht, Konfigurations-Fenster). Als Transferfunktion wurde eine 24-Bit-Colormap ausgewählt. Zusätzlich werden die Gradienten und die Beleuchtung berechnet.

Kapitel 10

Sonstiges

Glossar

3D-Rekonstruktion
Erstellung eines 3D-Modells aus einer Serie von 2D-Schichtbildern. 4, 10

3D-Scan
Tomographische Aufnahme. 6

3D-Textur
Siehe Textur. 9, 26

Back-To-Front-Komposition
Berechnung des Volumenintegrals in Richtung des Betrachters. 14

Bildgebendes Verfahren
Verfahren, bei dem aus Messgrößen Bilder erzeugt werden. 7

Bilineare Interpolation
Verfahren zur Interpolation eines 2D-Bildes oder einer Textur. 22

Blinded Reading
Auswertung von Bildserien durch Fachspezialisten unabhängig von weiteren Informationen. 6

dcm4che
Java-Bibliothek zum Laden und Speichern von DICOM-Bildern. 6

DICOM
Standard zur Repräsentation medizinischer Bilder und ihrer Zusatzinformationen. 6

DICOM-Datei
Datei im DICOM-Format. 6

DICOM-Format
Standardisiertes Bildformat für medizinische Bilder. Umfasst die reinen Bilddaten (komprimiert oder unkomprimiert) und Metadaten wie z.B. Bilderformat, Patientendaten, Geräteparameter usw. 4

DICOM-Header
Enthält Metadaten eines DICOM-Bildes. 17

DICOM-Objekt
siehe DICOM-Datei. 17

DICOM-Reader
Programm zum Einlesen von Bildern im DICOM-Format. 6

DICOM-Tag
Enthält einzelnen Parameterwert eines DICOM-Headers. Jedes Tag besitzt eine eindeutige achtstellige Id, die durch den DICOM-Standard festgelegt ist. 20

Downsampling
Operation, um Volumendaten in ein kleineres Format zu überführen. 23

Dreiecksdezimierung
Verringerung der Dreiecke in einem 3D-Modell. 18

Early-Ray-Termination
Früher Abbruch bei Front-To-Back-Komposition, falls Sichtstrahl auf lichtundurchlässiges Material stößt. 15

Emissions-Absorptions-Modell
Vereinfachtes physikalisches optisches Modell für Volume-Rendering. 13

Fensterung
Kontrasteinstellung bei einem medizinischem Bild. Durch die Fensterung kann ein bestimmter Wertebereich hervorgehoben und uninteressante Bereiche ausgeblendet werden. 23

Framerate
Anzahl der Bilder, die pro Sekunde von einer Grafikkarte auf dem Bildschirm angezeigt werden. 18

Front-To-Back-Komposition
Berechnung des Volumenintegrals vom Betrachter in Blickrichtung. 14

Gaußscher Weichzeichnerfilter
Filter, der in der Bildverarbeitung eingesetzt wird, um Signalrauschen zu unterdrücken. 18

GPU-Ray-Casting
Berechnung von Ray-Casting auf der Grafikkarte. 15

Gradient
Vektor der ersten Ableitung eines Punktes in einem skalaren Feld. Kann in einem Volumen dazu benutzt werden, die Normale einer Oberfläche zu bestimmen. 24, 36

Hounsfield-Einheit
Einheitliche Absorptionswerte für CT-Bilder, die relativ zum Absorptionswert von Wasser berechnet werden. 5, 11

Isofläche
Fläche, die einen konstanten Wert innerhalb eines Volumens repräsentiert. Wird mit Hilfe eines Polygonnetzes approximiert. 17

Kontrastmittel
Wird dem Körper beim CT- oder MRT-Verfahren hinzugegeben, um die Gewebedifferenzierung zu verbessern. 6

Lichtabsorption
Lichtdurchlässigkeit eines Materials. 12

Lichtemission
Farbe oder Leuchtkraft des Materials. 12

Marching-Cubes-Algorithmus
Algorithmus zur Erzeugung von Isosurfaces. 11

Mesh-Smoothing
Glättung eines Polygonnetzes. 19

Modalität
Gerät, mit dem Tomographie erstellt wird. 10

MPR
Multiplanare Reformatierung, Erzeugung alternativer planarer Schnittebenen einer 3D-Bildserie. 11

Newton-Coates-Formel
Numerische Quadraturformel zur Annäherung eines Integrals. 14

OpenGL
Programmierschnittstelle für zwei- und dreidimensionale Grafikprogrammierung. 6, 9

Optische Tiefe
Gibt an, wieviel Licht zwischen zwei Schichten eines 3D-Scans absorbiert wird. 13

Ray Casting
Diskrete Berechnung des Volumen-Integrals entlang eines Sichtstrahls. 15, 16

Ray Tracing
Aufwendige Berechnung zur bestmöglichen realitätsnahen Darstellung einer 3D-Szene. 15

Relaxationszeit
Zeit, nach der sich das induzierte Magnetfeld eines Atomkerns beim MRT-Verfahren wieder abgebaut hat. 6

Render-Laufzeit
Zeit, die die Grafikkarte benötigt, um ein Bild zu erzeugen. 18

Renderpipeline
Weg vom Laden der Daten in VTK bis zur Anzeige auf dem Bildschirm. 20

Resampling
Operation, um Volumendaten in ein anderes Format zu überführen (Verkleinerung oder Vergrößerung des Volumenquaders). 18

Riemannsche Summe
Vereinfachte Gleichung zur Berechnung eines Integrals durch die Diskretisierung des Wertebereichs. 13

Schichtabstand
Abstand zwischen zwei benachbarten tomographischen Bildern. 6

Shader
Kleines Programm für die freie Verarbeitung von Geometrie- und Pixeldaten. Auf modernen Grafikkarten ist es seit einigen Jahren möglich, mehr Einfluss auf die Ausgabe eines Grafikprogramm auszuüben. Davor waren die Funktionen zur Ausgabe fest verdrahtet und nur auf bestimmte Anwendungsfälle beschränkt. 9, 26

Simpsonregel
Annäherung des Integrals einer Funktion durch Parabeln. 14

Space leaping
Technik zur Verbesserung der Performance beim Volume-Rendering. 22

Swing
GUI-Bibliothek für Java. 9, 19

Textur
Bezeichnung für ein Bild im Grafikspeicher. Texturen können eins-, zwei-, oder dreidimensional sein und bis zu vier Komponenten enthalten (RGBA). 22

Tomographie
Aufnahme mehrerer 2D-Schichtbilder eines Objektes entlang einer Achse. 4

Transferfunktion
Funktion zur Umwandlung tomographischer Datenwerte in sichtbare Helligkeits- oder Farbwerte. Wird beim Volume-Rendering häufig mit einer Lookup-Table (LUT) realisiert. 13, 31

Trapezregel
Annäherung des Integrals einer Funktion durch Trapeze. 14

Trilineare Interpolation
Verfahren zur Interpolation einer 3D-Textur. 22

Volumenrendering
Grafische Darstellung eines Volume-Datensatzes. 6, 12, 22, 26

Volumenrenderingintegral
Gleichung zur Berechnung der optischen Eigenschaften eines Volumens entlang eines Sichtstrahls, der parallel zur Blickrichtung des Betrachters verläuft. 13

VTK
C++-Bibliothek zur Visualisierung von wissenschaftlichen Daten. 7, 17

Weichzeichnerfilter
Weichzeichnen eines zwei- oder dreidimensionalen Bildfeldes, z.B. mittels eines Gaußschen Weichzeichnerfilters. 18, 33

Windowing
siehe Fensterung. 6, 30

Acronyms

ACR	American College of Radiology. 6
AWT	Abstract Window Toolkit. 19
CT	Computertomographie. 4
DICOM	Digital Imaging and Communications in Medicine. 6
FPS	Frames Per Second. 22
HU	Hounsfield-Unit. 5
JOGL	Java Bindings for OpenGL. 9, 26
LUT	Lookup-Table. 23
MRT	Magnetresonanztomographie. 4, 5
NEMA	National Electrical Manufacturers Association. 6
ROI	Region of Interest. 6
VISKIT	VISualisation toolKIT. 6

Literaturverzeichnis

(1) **Bemmel, Jan H. van/Musen, Mark A.:** Handbook of Medical Informatics. 2. Auflage. Springer, Berlin 10 1997, 628 Seiten, ISBN 9783540633518.

(2) **Castleman, Kenneth R.:** Digital Image Processing. 2. Auflage. Prentice Hall 1 1996, 667 Seiten, ISBN 9780132114677.

(3) **Dugas, Martin/Schmidt, Karin:** Medizinische Informatik und Bioinformatik: Ein Kompendium für Studium und Praxis (Springer-Lehrbuch). 1. Auflage. Springer, Berlin 10 2002, 229 Seiten, ISBN 9783540425687.

(4) **Engel, Klaus et al.:** Real-time Volume Graphics. 1. Auflage. A K Peters 7 2006, 497 Seiten, ISBN 9781568812663.

(5) **Handels, Heinz:** Medizinische Bildverarbeitung: Bildanalyse, Mustererkennung und Visualisierung für die computergestützte ärztliche Diagnostik und Therapie. 2. Auflage. Vieweg+Teubner 3 2009, 432 Seiten, ISBN 9783835100770.

(6) **Hansen, Charles D./Johnson, Chris R.:** Visualization Handbook. 1. Auflage. Academic Press 12 2004, 984 Seiten, ISBN 9780123875822.

(7) **Kitware, Inc.:** VTK User's Guide Version 5. 5. Auflage. Kitware, Inc. 9 2006, 382 Seiten, ISBN 9781930934184.

(8) **Kruger, J./Westermann, R.:** Acceleration Techniques for GPU-based Volume Rendering. In: VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03). Washington, DC, USA: IEEE Computer Society 2003, ISBN 0–7695–2030–8, 38.

(9) **Levoy, Marc:** Display of Surfaces From Volume Data. In: IEEE Computer Graphics and Applications. IEEE Computer Society 1988, ISSN 0272–1716, 29–37.

(10) **Lipinski, Hans-Gerd:** Einführung in die medizintechnische Informatik. Oldenbourg 1999, 355 Seiten, ISBN 9783486238792.

(11) **Preim, Bernhard/Bartz, Dirk:** Visualization in Medicine: Theory, Algorithms, and Applications (The Morgan Kaufmann Series in Computer Graphics). 1. Auflage. Morgan Kaufmann 7 2007, 680 Seiten, ISBN 9780123705969.

(12) **Roettger, Stefan et al.:** Smart hardware-accelerated volume rendering. In: VISSYM '03: Proceedings of the symposium on Data visualisation 2003. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association 2003, ISBN 1–58113–698–6, 231–238.

(13) **Schroeder, Will/Martin, Ken/Lorensen, Bill:** The Visualization Toolkit, Third Edition. 3. Auflage. Kitware Inc. 8 2004, 504 Seiten, ISBN 9781930934122.

(14) **Schroeder, William J./Martin, Kenneth M./Lorensen, William E.:** The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In: VIS '96: Proceedings of the 7th conference on Visualization '96. Los Alamitos, CA, USA: IEEE Computer Society Press 1996, ISBN 0–89791–864–9, 93–ff..

(15) **Schroeder, William J./Zarge, Jonathan A./Lorensen, William E.:** Decimation of triangle meshes. In: SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques. New York, NY, USA: ACM 1992, ISBN 0–89791–479–1, 65–70.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel *"Integration von 3D-Algorithmen der C++-Klassenbibliothek VTK in eine medizinische Java-Visualisierungsbibliothek (viskit) für die dreidimensionale Visualisierung von DICOM-Bildern"* selbständig und ohne unerlaubte Hilfe verfasst habe.

Berlin, den 15. Dezember 2014

Oliver Sanftleben

Einverständniserklärung

Ich bin damit einverstanden, dass die vorliegende Arbeit mit dem Titel *"Integration von 3D-Algorithmen der C++-Klassenbibliothek VTK in eine medizinische Java-Visualisierungsbibliothek (viskit) für die dreidimensionale Visualisierung von DICOM-Bildern"* in der Bibliothek ausgelegt wird.

Berlin, den 15. Dezember 2014

Oliver Sanftleben