UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Interaktives Ray-Casting von Volumendaten

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Raphael Philipp Menges

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter: Gerrit Lochmann, M.Sc.
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im März 2014

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....
(Ort, Datum) (Unterschrift)

Abstract

This thesis covers the mathematical background of ray-casting as well as an exemplary implementation on graphics processing units, using a modern programming interface. The implementation is embedded within an editor, which enables the user to activate optimizations of the algorithm. Techniques like transfer functions and local illumination are available for a more realistic visualization of materials. Moreover, the user interface gives access to features like importing volumes, let one define a custom transfer function, holds controls to adjust parameters of rendering and allows to activate further techniques, which are also subject of discussion in this thesis. Benefit of all shown techniques is measured, whether it is expected to be visual or on the part of performance.

Zusammenfassung

Diese Arbeit vermittelt die mathematischen Grundlagen des Ray-Casting Algorithmus und bespricht eine interaktive Umsetzung auf Grafikkarten mit Hilfe einer modernen Schnittstelle. Die Implementation erfolgt im Rahmen eines umfassenden Programmes, welches weitere Techniken und Verbesserungen des Algorithmus für den Nutzer anwählbar macht. Unter anderem wird von Transferfunktionen und lokaler Beleuchtung Gebrauch gemacht, um realistische Materialien darstellen zu können. Die Benutzeroberfläche bietet die Möglichkeit, Volumina zu importieren, Transferfunktionen zu definieren, Parameter der Darstellung einzustellen und weitere Techniken zu aktivieren, deren Grundlagen und Umsetzung ebenfalls in dieser Arbeit dargelegt werden. Der Nutzen der anwählbaren Optionen wird je nach Fall auf optische Qualität oder Vorteil in der Performance hin untersucht.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In der modernen Medizin sind dreidimensionale Aufnahmen des Inneren von Körpern nicht mehr wegzudenken. Mithilfe von bildgebenden Verfahren wie Computer- oder Magnetresonanztomographie können genaue Volumendaten erfasst und damit projizierte Bilder generiert werden. Die Daten sind dabei räumlich immer hochauflösender geworden und können Blutbahnen und Organe genau abbilden. Diese enormen Datenmengen müssen mithilfe einer zielgerichteten Visualisierung für den Beobachter aufgearbeitet werden, um den Blick auf die wesentlichen Bestandteile reduzieren zu können und gleichzeitig eine hochwertige Darstellung zu gewährleisten. Dazu gehört das Einfärben von Regionen, die Simulation von Licht und die Wählbarkeit der sichtbaren Bereiche aus dem Volumendatensatz. Aktuelle Grafikkarten sollen als eine geeignete Plattform hochwertige Visualisierungen in Echtzeit ermöglichen. Sowohl aufgrund ihrer stark auf Parallelität ausgerichteten Architektur, als auch wegen des immer größeren Videospeichers, der extrem schnellen Zugriff gewährt und eine einfache Implementierung ohne Pufferung und Datenaustäusche ermöglicht, sind Grafikkarten für die Darstellung von Volumen die erste Wahl. Moderne Schnittstellen, wie das in den Beispielen zu dieser Arbeit genutzten OpenGL 3.3, vereinfachen die Programmierung weiterhin, da der gesamte Ray-Casting Algorithmus mit einem einzigen Shader umgesetzt werden kann.

1.2 Zielsetzung

Ziel dieser Arbeit ist es ein Verständnis über die Abbildung von Volumendaten in den zweidimensionalen Raum des Bildschirms zu vermitteln und mithilfe von Ray-Casting in Rahmen eines Programmes zu realisieren, das für die Visualisierung die Grafikkarte in großem Umfang nutzt. Dabei kommen verbreitete Ansätze wie Transferfunktionen und lokale Beleuchtung für die Aufbereitung der Daten zum Einsatz und werden sowohl theoretisch dargelegt, als auch praktisch umgesetzt. Verbesserung der Performance mithilfe von Early Ray Termination oder adaptivem Sampling werden hinsichtlich ihres Nutzens untersucht und bei Verfahren zur optischen Verbesserung die visuellen Ergebnisse aufgezeigt. Alle Techniken sollen in diesem einen Programm anwählbar sein und unvermittelt auf einen geladenen Volumendatensatz angewendet werden. Des Weiteren sollen XML-basierte Formate für die Speicherung von Transferfunktionen und der Volumendaten definiert und benutzt werden. Die Umsetzung erfolgt mit C++ und OpenGL 3.3. Für die Lehre werden später ausgewählte Techniken ausgegliedert und in minimalen Programmen umgesetzt.

2 Grundlagen

Hier werden die Grundlagen für den Ray-Casting Algorithmus vorgestellt. Beschreibungen zu einzelnen Techniken und deren Umsetzung folgt im nächsten Kapitel.

2.1 Datenstruktur

Volumendatensätze beinhalten zumeist skalare Werte, welche z.B. die Dichte des Objektes an der jeweiligen Stelle im Raum beschreiben können. Es gibt unterschiedliche Datenstrukturen um diese Werte zu speichern und dabei verschiedene Vor- und Nachteile bieten. Allgemein ist es aber immer das Ziel, aus einer beliebigen Stelle im Volumen den zugehörigen Wert zu extrahieren und ihn als Ausgangswert für weitere Transformationen und Kompositionen zu verwenden.

Uniformes Netz

Oft liegen Volumendaten als uniformes Netz oder Gitter vor, bei dem die Werte gleichförmig verteilt sind. Einen Punkt im Volumen nennt man analog zu Bildern anstatt Pixel (Picture Element) einen *Voxel* (Volume Element). Dabei gibt es zwei unterschiedliche Auffassung von diesem Begriff: Zum einen kann man darunter eine Ansammlung von gleichartigen Würfeln verstehen, in denen jeweils ein einziger Wert vorherrscht. Im Bezug auf Volume Rendering passt aber eher eine andere Auffassung von Voxeln. In dieser wird ein Voxel als Punkt im Raum angesehen und zwischen den Punkten werden Werte interpoliert, um glatte Oberflächen und fließende Übergänge zu ermöglichen. Diese uniformen Netze sind zwar nicht sehr flexibel, aber mit der Hardware schnell zu verarbeiten und einfach zu speichern. Wo früher noch auf eine intelligente Reihenfolge der Werte im Speicher geachtet werden musste, z.B. mit *swizzled* 3D Texturen [HKRs+06, Kapitel 8.1], gewährleistet heutzutage die Hardware und deren Treiber einen schnellen Zugriff. Deshalb können Volumendaten als Datenblock[1] in den Videospeicher geschrieben werden.

Simplex Netz

Es gibt noch andere und flexiblere Formen der Volumendatenspeicherung, z.B. Netze aus frei wählbaren konvexen Zellen. Das uniforme Netz ist eine Variante dieser sogenannten *Simplex Netze*, da man es als gleichförmige Würfel auffassen kann, die wiederum konvexe Zellen darstellen. Man spricht bei einer solchen Zelle von n-simplex, wobei 0-simplex einen Punkt, ein 1-simplex eine Linie, ein 2-simplex ein Dreieck und so weiter beschreibt.

[1] Auslesen mit: `value(x, y, z) = block[x + y*xDim + z*xDim*yDim]`

In “Real-Time Volume Graphics” [HKRs+06, Kapitel 7.5] sind Renderingtechniken für tetraedrische Netze beschrieben. Diese Arbeit wird im Folgenden ausschließlich uniforme Netze voraussetzen, um den Fokus auf Ray-Casting legen zu können.

Datenquellen

Verfahren zu Erfassung von Volumendaten unterscheiden sich in der Art der Strahlung, mit welcher die Eigenschaften innerhalb des Volumens gemessen werden. Bei *Computertomographie* (CT) wird aus einer Röntgenquelle Strahlung emittiert, diese durch das abzubildende Objekt je nach Eintrittswinkel abgeschwächt und auf der gegenüberliegenden Seite mit einem Detektor eingefangen. Aus den gemessenen Werte können dann zum Beispiel mit gefilterter Rückprojektion die Dichten innerhalb des Volumen rekonstruiert werden. Dieses Verfahren ist sehr schnell und liefert aufgrund der Röntgenstrahlung vor allem bei der Abbildung von Knochen gute Ergebnisse. Gewebe, wie zum Beispiel Blutbahnen und Organe, können ohne Einsatz von Kontrastmitteln jedoch nicht in ihrer Struktur erfasst werden. Dafür eignet sich eher *Magnetresonanztomographie* (MRI), bei welcher mit magnetischen Wechselfeldern die im Körper vorhandenen Moleküle durch Induktion anregt und zur Emission von Strahlung veranlasst werden. Aufnahmen dauern mit einigen Minuten länger als bei der Computertomographie, da die Strahlung weniger stark ist und mehr gestreut wird. Im Gegensatz zur Röntgenstrahlung gilt dieses Verfahren aber als unbedenklich für die Gesundheit. Diese beiden Verfahren werden hauptsächlich als Quelle für Volumendaten genutzt und es kommen im weiteren Verlauf sowohl CT- als auch MRI-Datensätze in Beispielen zum Einsatz. Für das Rendering besteht zwischen beiden Quellen kein Unterschied, man sollte sich bei der Wahl der Transferfunktion aber der Art der abgebildeten Daten bewusste sein. Bei einer Aufnahme aus einem CT existiert keine Transferfunktion, um die Windungen des Gehirns wiederzugeben, da die Informationen schlicht nicht im Volumendatensatz vorhanden sind.

2.2 Volume-Rendering Integral

Die Formeln und Erklärungen im Folgenden sind stark an das Kapitel “Theoretical Background and Basic Approaches” aus “Real-Time Volume Graphics” [HKRs+06, Kapitel 1] angelehnt. Einige für die Herleitung des *Volume-Rendering Integrals* und dessen Annäherung weniger wichtige Passagen werden nicht behandelt und können nachgeschlagen werden. Um den Fokus auf die eigentliche Herleitung zu legen, wird bei Licht und ähnlichem vereinfacht von ‘Strahlung’ gesprochen. Außerdem wird davon ausgegangen, dass bei Farben die drei Kanäle für Rot, Grün und Blau unabhängig betrachtet werden können und auch keine Brechung von Licht stattfindet.

Analytische Form

Das verbreitete optische Modell für die Darstellung von Volumen ist das *Emission-Absorption Modell*. Dieses besagt, dass das enthaltene Gas bzw. die Materie Licht aussenden und absorbieren kann. Formen von Lichtstreuung oder globale Beleuchtungsansätze werden vernachlässigt. Die folgende Gleichung beschreibt das Modell für einen Punkt auf einem einzelnen Strahl durch das Volumen und wird in einer allgemeineren Form als *Volume-Rendering Gleichung* beschrieben:

$$\frac{\mathrm{d}I(s)}{\mathrm{d}s} = -\kappa(s) * I(s) + q(s). \tag{1}$$

Die Position auf dem Strahl wird mit s bezeichnet und $I(s)$ ist die bisher auf dem Strahl eingesammelte Strahlung. Die linke Seite der Gleichung $\frac{\mathrm{d}I(s)}{\mathrm{d}s}$ ist eine Ableitung von $I(s)$ und steht für den Wert der Strahlung genau an der Position s auf dem Strahl. Wenn die eingesammelte Strahlung entlang des Strahles als Ergebnis der Funktion $I(s)$ betrachtet wird, so ist der Wert der Ableitung dieser Funktion an einer Position des Strahles die Strahlung an der Stelle s. Diese Strahlung pro Abtastpunkt wird auf der rechten Seite der Gleichung in die einzelnen Komponenten der *Absorption* und *Emission* aufgeteilt. Die Absorption, bei der Licht in Wärme umgewandelt wird, ist mit der Funktion $\kappa(s)$ gegeben und schwächt die an diesem Punkt einfallende Strahlung ab. Die Emission wird mit $q(s)$ abgebildet und auf die nach der Absorption übrigen Strahlung addiert. Die Parameter beider Funktionen können innerhalb des Volumens unterschiedliche Werte annehmen und sind technisch durch die sogenannte Transferfunktion umgesetzt.

Jene Berechnung der *Volume-Rendering Gleichung* für einen Punkt soll nun auf einem kompletten Strahl durch das Volumen ausgeführt werden. Dies wird mit folgendem Integral ausgedrückt, dass als *Volume-Rendering Integral* bezeichnet wird:

$$I(D) = I_0 * e^{-\int_{s_0}^{D} \kappa(t)\ \mathrm{d}t} + \int_{s_0}^{D} q(s) * e^{-\int_{s}^{D} \kappa(t)\ \mathrm{d}t}\ \mathrm{d}s. \tag{2}$$

Es wird in Richtung des Strahles von der Startposition $s = s_0$ bis zur Endposition $s = D$ integriert. Der rechte Term der Summe beschreibt in Form eines Integrals den Weg durch das Volumen hin zur Kamera und sammelt die Emission von Strahlung ein. Strahlung, die von hinten in das Volumen an der Startposition s_0 eintritt, wird mit I_0 im linken Term bedacht. Das Ergebnis $I(D)$ ist die Strahlung, welche letztendlich in die Kamera eintritt. Die Abschwächung von Strahlung über den Weg durch das Volumen hinweg wird jeweils mit einem exponentiellen Term abgebildet. Diese Abschwächung kann folgendermaßen dargestellt werden, was auch als *Optische Tiefe* bekannt ist:

$$\tau(s_1, s_2) = \int_{s_1}^{s_2} \kappa(t)\ \mathrm{d}t. \tag{3}$$

Die Absorption des Lichts zwischen s_1 und s_2 ist dabei von der Funktion $\kappa(t)$ abhängig, die wiederum als Eingabeparameter die Position auf dem Strahl, sprich die Koordinate im Volumen, aufnimmt. Kleine Werte von $\kappa(t)$ bedeuten wenig Abschwächung und transparente Regionen, große Werte viel Abschwächung der Strahlung und damit Opazität. Mithilfe der optischen Tiefe lässt sich die Transparenz eines Materials zwischen s_1 und s_2 wie folgt ausdrücken:

$$T(s_1, s_2) = e^{-\tau(s_1, s_2)}. \tag{4}$$

Damit kann man das *Volume-Rendering Integral* folgendermaßen umschreiben:

$$I(D) = I_0 * T(s_0, D) + \int_{s_0}^{D} q(s) * T(s, D)\,\mathrm{d}s. \tag{5}$$

Der Strahlungswert $I(D)$ am Ende des Strahles, an der Kamera, ist gleich der Summe aus Strahlung hinter dem Volumen, exponentiell abgeschwächt, und der Emission entlang des Strahles, ebenfalls exponentiell abgeschwächt.

Lokale Beleuchtung

Obwohl dieses Modell der Absorption und Emission keine Lichtstreuung beinhaltet, ist es aufgrund des geringen Berechnungsaufwandes populär. Einzelne Streustrahlen aus der Umgebung können aber primitiv simuliert werden, indem Ablenkung oder Verdeckung durch Inhalte des Volumens nicht beachtet werden. Jeder Punkt im Volumen wird für sich alleine lokal beleuchtet und dazu auf Beleuchtungsmodelle nach *Phong* [Pho75] oder *Blinn* [Bli77] zurückgegriffen. Die für die Berechnung erforderliche Normale wird über den Gradienten in der nahem Umgebung des zu beleuchtenden Punktes innerhalb des Volumens bestimmt. Die zusätzliche Strahlung $i(s)$ durch die Lichtquelle kann auf den vorhandenen Term von Strahlung aus Absorption und Emission aufaddiert werden, wobei in diesem Term noch Materialeigenschaften wie Farbe bedacht werden müssen:

$$I_{illum}(D) = I(D) + \int_{s_0}^{D} i(s) * T(s, D)\,\mathrm{d}s. \tag{6}$$

Die lokale Beleuchtung besitzt diffuse, spekulare und ambiente Anteile, welche mit einfachen mathematischen Operationen zusammengerechnet werden können. Dieser Ansatz der Beleuchtung hat nahezu keinen Einfluss auf den Berechnungsaufwand, sofern der Gradient entweder vorberechnet vorliegt oder anderweitig sowieso benötigt wird, z.B. für Fresnelterme, Gradienten-Magnituden abhängige Opazität oder Spherical Reflection Maps. Die visuellen Ergebnisse können durch die Beleuchtung in den meisten Fällen deutlich verbessert werden. Schon bei einzelnen direktionalen Lichtquellen sind die räumlichen Verhältnisse von Volumeninhalten zueinander deutlicher, wie in Kapitel 3.12 zu sehen ist.

Diskretisierung

Um das Integral in Echtzeit bei justierbaren Parametern berechnen zu können, wird die Umwandlung in eine endliche Summe vorgenommen. Dabei entsteht eine Annäherung, deren Fehler gegenüber der analytischen Methode möglichst zu minimieren ist. Dazu wird zuerst das Integral in

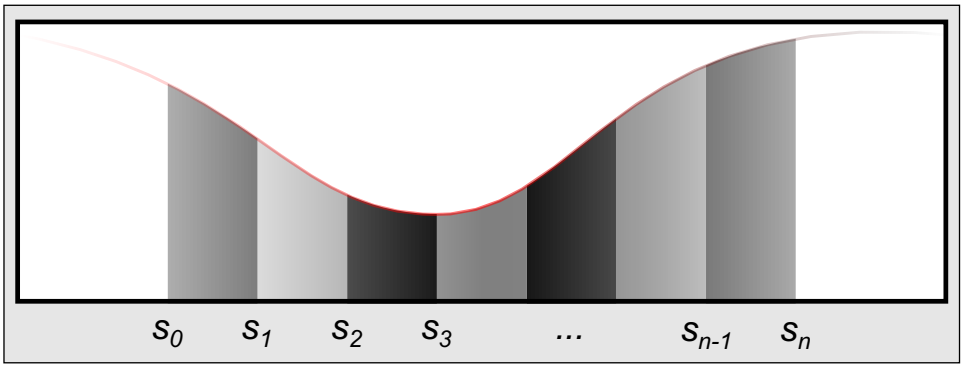

Abbildung 1: Gliederung des Integrals in einzelne Intervalle.

kleinere Teilintegrale aufgeteilt und jedes Intervall unabhängig für sich betrachtet. Es ist nicht garantiert, dass alle diese Unterteilungen eine gleiche Länge vorweisen. Diese Intervalle werden durch die Positionen $s_0 < s_1 < \cdots < s_{n-1} < s_n$ ausgedrückt, wobei s_0 der Startpunkt und s_n den Endpunkt bzw. D darstellt. Die Strahlung an einem Punkt s_i auf dem Strahl in Richtung der Kamera kann folgendermaßen ausgedrückt werden:

$$I(s_i) = I(s_{i-1}) * T(s_{i-1}, s_i) + \int_{s_{i-1}}^{s_i} q(s) * T(s, s_i)\,\mathrm{d}s. \tag{7}$$

Es handelt sich um das Volume-Rendering Integral, beschränkt auf ein Teilstück des Strahles in Form eines Intervalls. Nun wird für eine Vereinfachung eine neue Notation für die Transparenz T und den Farbanteil c eingeführt:

$$T_i = T(s_{i-1}, s_i),\ c_i = \int_{s_{i-1}}^{s_i} q(s) * T(s, s_i)\,\mathrm{d}s. \tag{8}$$

Durch das Einsetzen dieser Gleichungen in die Gleichung 7 für ein Intervall lässt sich die Strahlung an $I(D)$ rekursiv aufschreiben:

$$I(D) = I(s_n) = I(s_{n-1}) * T_n + c_n = (I(s_{n-2}) * T_{n-1} + c_{n-1}) * T_n + c_n = \ldots$$

Was folgendermaßen umgeschrieben werden kann:

$$I(D) = \sum_{i=0}^{n} (c_i * \prod_{j=i+1}^{n} T_j),\ mit\ c_0 = I(s_0). \tag{9}$$

Damit kann die rekursive Einsetzung als eine Summe mit beinhaltetem Produkt dargestellt werden. Mit c_i wird die Farbe als Kombination aus Rot, Grün und Blau beschrieben, welche vorher nur abstrakt als Strahlung betrachtet wurde. Statt der Transparenz wird oft die Opazität $\alpha_i = 1 - T_i$ in Transferfunktionen angegeben, da dies eher der Erwartung der Nutzer entspricht. In Bildern mit durchsichtigen Anteilen wird die Durchsichtigkeit auch über einen α-Wert angegeben. Je höher dieser, desto undurchsichtiger das Bild. Die Umwandlung in eine diskrete Form ist mit diesen

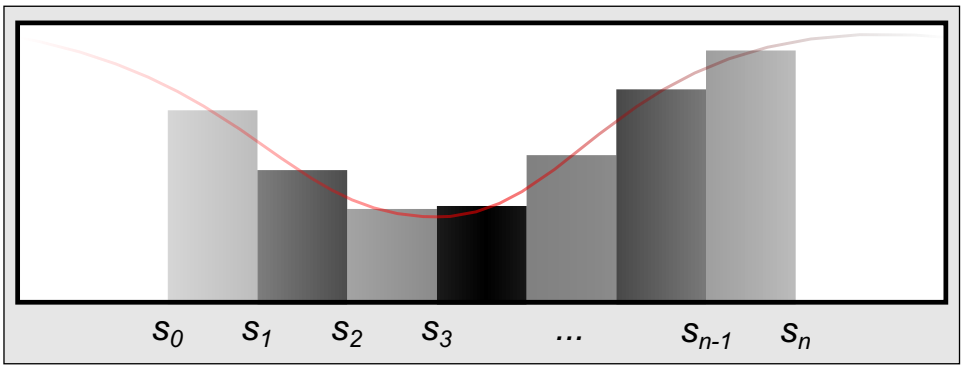

Abbildung 2: Annäherung der Integrale mithilfe der Riemann Summen.

Umformungen aber nicht vollbracht, da innerhalb der Intervalle in c_i jeweils noch ein Integral steckt. Bisher handelt es sich also noch nicht um Annäherung, sondern um eine Umformung der analytischen Gleichung in aufzusummierende Teilintegrale. Diese müssen nun für eine numerische Berechnung geschickt angenähert werden. Den einfachsten Fall dieser Annäherung bilden stückweit konstante Funktionen, die pro Abschnitt des Integrals einen Wert für Transparenz und Farbe liefern. Diese Vorgehensweise ist den sogenannten *Riemann Summen* nachempfunden, welche eine Schätzung über den Wert von Integralen darstellen. Mit dieser Annäherung ist die Transparenz für das Segment s_i mit der Länge Δx:

$$T_i \approx e^{-\kappa(s_i)*\Delta x}. \tag{10}$$

Der Farbbeitrag eines Segments ist:

$$c_i \approx q(s_i) * \Delta x. \tag{11}$$

Da die Annäherung mit konstanten Werte für ein Integral grob ist, sind für eine gute Darstellungsqualität kleine Abtastschritte, also kurze Intervalle, notwendig. Eine Beschleunigung kann durch *adaptives Sampling* erreicht werden, bei dem die Varianz der Volumendaten in die Berechnung der Abtastweiten einbezogen wird und unterschiedliche Längen von Intervallen ermöglicht. Diese Technik wird genauer in Kapitel 3.9 behandelt.

Eine weitere Möglichkeit der Minimierung des Annäherungsfehlers bilden Tabellen mit *Pre-Integration*, bei denen vor der Darstellung zwischen den möglichen, durch die Transferfunktion transformierten, Kombinationen aus Werten an s_{i-1} und s_i schrittweise abgetastet, aufsummiert und normiert wird. Dadurch wird zumindest der Fehler auf Seiten der Transferfunktion minimiert, nicht aber der bei der Abtastung der Volumendaten selbst. Damit lassen sich jedoch schon gute Ergebnisse erzielen, die einer Auswertung der analytischen Form näher kommt als das punktweise Abtasten der Volumendaten und Abfragen der Transferfunktion. Auf diese Technik wird in Kapitel 3.8 eingegangen.

Komposition

Für die Berechnung der diskreten Annäherung des Volume-Rendering Integrals wird die in Gleichung 9 aufgestellte Formel in einfacher zu berechnende Terme aufgelöst, die nacheinander zum Zielwert zusammengerechnet werden. Dafür gibt es zwei Konzepte: *Front-To-Back* Komposition und

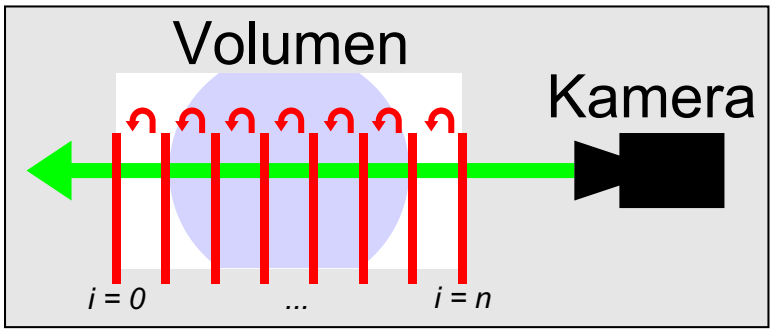

Abbildung 3: Front-To-Back Komposition analog zu Gleichung 12.

Back-To-Front Komposition. Im Folgenden wird die Front-To-Back Komposition behandelt, da sie sich direkt mit Ray-Castings umsetzen lässt. Die zugehörigen Gleichungen sind:

$$\begin{aligned} \hat{C}_i &= \hat{C}_{i+1} + \hat{T}_{i+1} * C_i, \\ \hat{T}_i &= \hat{T}_{i+1} * (1 - \alpha_i). \end{aligned} \tag{12}$$

Mit den Initialisierungen:

$$\begin{aligned} \hat{C}_n &= C_n, \\ \hat{T}_n &= 1 - \alpha_n. \end{aligned} \tag{13}$$

Die Farbe C wird iterativ über den Strahl in $\hat{C}$ gesammelt, wobei der Abstieg von $i+1$ zu i die Richtung weg von der Kamera bedeutet. Die Iteration startet nämlich mit $i = n$ und endet bei $i = 0$. Es ist zu beachten, dass die

Denkweise anders herum ist als z.B. in Gleichung 7, die den Strahl von hinten durch das Volumen zur Kamera hin beschreibt. Die Transferfunktion liefert jeweils C und α für die Berechnungen. Analog zur Farbe wird die Transparenz in $\hat{T}$ gespeichert. Durch eine Umbenennung der Variablen und eine leicht veränderte Schreibweise entsteht die allgemeine Beschreibung der Front-To-Back Komposition:

$$\begin{aligned} C_{dst} &\leftarrow C_{dst} + (1 - \alpha_{dst}) * C_{src}, \\ \alpha_{dst} &\leftarrow \alpha_{dst} + (1 - \alpha_{dst}) * \alpha_{src}. \end{aligned} \tag{14}$$

Hierbei beinhaltet C_{src} und α_{src} immer den Wert, den die Transferfunktion für den aktuell abgetasteten Punkt liefert. C_{dst} und α_{dst} dienen zur Zusammenrechnung und werden pro Iteration aktualisiert. Bei jeder Iteration wird die abgetastete Position also ein Stück weiter weg auf dem Strahl bewegt, die beiden Werte C_{src} und α_{src} mithilfe der Transferfunktion befüllt und dann mit den bisherigen Ergebnissen aus C_{dst} und α_{dst} verrechnet. Die Ergebnisse bilden jeweils mit C_{dst} bzw. α_{dst} die Ausgangswerte für den nächsten Schritt in der Iteration. Dadurch wird auch deutlich, dass die Operationen der Komposition nicht kommutativ sind, da sich die Reihenfolge der Abtastung des Strahles direkt auf das Ergebnis auswirkt.

Eine Alternative zu der vorgestellten Komposition für das Emission-Absorption Modell bildet unter anderem die *Maximum Intensity Projection* (MIP), welche die Abbildung durch Röntgengeräte simuliert. Analog formuliert zu der Front-To-Back Komposition lautet die Gleichung:

$$C_{dst} \leftarrow max(C_{dst}, C_{src}). \tag{15}$$

Die mathematische *max*-Funktion liefert pro Strahl den maximalen Farbwert aus der Transferfunktion, der von einem im Volumen geschnittenen Punkt, bzw. dessen skalaren Wert, erzeugt wurde. Im Gegensatz zur Front-To-Back Komposition handelt es sich um ein Verfahren, dass unabhängig von der Reihenfolge der Abtastungen des Strahles ist.

Assoziative Farben

In Transferfunktionen werden zumeist keine pre-multiplied Farben gespeichert. Diese Farben sind nicht mit ihrem korrespondieren α-Wert multipliziert worden und können daher zu dem sogenannten Effekts des *Color Bleedings* führen, siehe dazu ein Beispiel in Abbildung 4. Um dieses mögliche Problem zu lösen, genügt eine kleine Anpassung der Front-To-Back Komposition:

$$C_{dst} \leftarrow C_{dst} + (1 - \alpha_{dst}) * C_{src} * a_{src}. \tag{16}$$

Im weiteren Verlauf der Arbeit wird ausschließlich diese Variante der Front-To-Back Komposition für den Farbanteil benutzt.

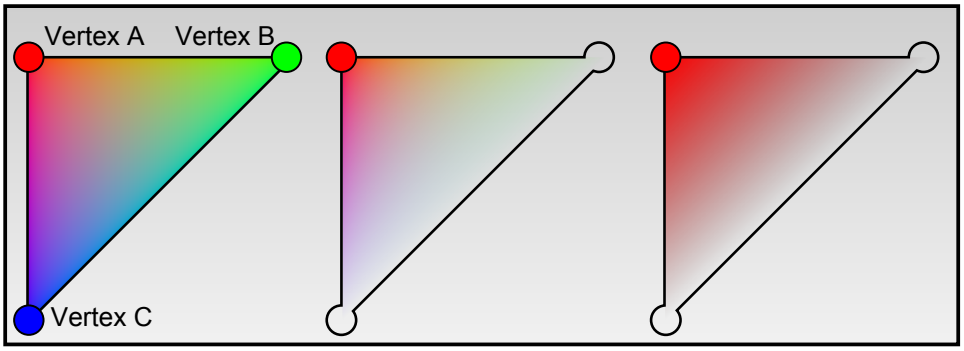

Abbildung 4: Links eine Interpolation zwischen drei Vertices ohne Transparenz. In der Mitte sind Vertex B und C volltransparent gesetzt und dann die Farbwert für das aufgespannte Dreieck interpoliert worden. Dabei wird Color Bleeding sichtbar. Rechts wurden die Farbwerte mit den Alphawerten gewichtet und Color Bleeding wird verhindert.

Korrektur der Opazität

Um eine Anpassung der Abtastweite durch den Nutzer oder durch adaptives Sampling zu ermöglichen, müssen die Weiten bei der Komposition bedacht werden. Die Herleitung ist in "Real-Time Volume Graphics" [HKRs+06, Kapitel 1.4.3] zu finden:

$$\begin{aligned} \tilde{T} &= T^{(\frac{\Delta\tilde{x}}{\Delta x})}, \\ \tilde{c} &= c * (\frac{\Delta\tilde{x}}{\Delta x}). \end{aligned} \tag{17}$$

Hierbei steht Δx für die normale Abtastweite und $\Delta\tilde{x}$ ist die adaptive.

2.3 Transferfunktion

Prinzipiell könnten in der Volumendatenstruktur gleich die physikalischen Eigenschaften der Emission und Absorption eines jeden Punktes gespeichert werden. Abgesehen von den riesigen Ausmaßen der gespeicherten Dateien muss man aber im Hinterkopf behalten, woher die meisten Volumendaten stammen. Computertomographen und Kernspins liefern nämlich skalare Werte, die abhängig von der eingesetzten Technik unterschiedliche Bedeutung haben, aber auf keinen Fall die wahrnehmbare Farbe von Objekten darstellen. Daher ist es sinnvoll in einem Volumen nur diese skalaren Werte mit einer geräteabhängigen Auflösungstiefe von meist 8 oder 16-Bit zu speichern. Schon mit diesen Wertebereichen ist die Speichergröße eines Volumens immens und fordert ab gewissen dreidimensionalen Ausmaßen selbst moderne Grafikkarten und deren Videospeicher.

Praktisch kann man sich ein Volumen als eine Sammlung von Grauwerten in einem Gitter vorstellen, die je nach Auflösungstiefe unterschied-

lich viele Abstufungen aufweisen. Um diese Werte anschaulich darstellen zu können, werden sie als Eingabe für eine Funktion verwendet, welche die skalaren Eingabewerte zu frei wählbaren Ausgabewerten transformiert. Siehe Abbildung 5 für ein Beispiel einer solchen Transformation durch eine Transferfunktion. Es kann sich je nach Szenario um Farbe, Transparenz, Emission oder Glanzstärke handeln. Diese werden pro Abtastpunkt aus den Volumendaten an der aktuellen Stelle mit der Transferfunktion errechnet oder ausgelesen und für die Komposition eingesetzt. Dazu gibt es verschiedene Verfahren mit Vor- und Nachteilen, die in Kapitel 3.4 genauer vorgestellt werden. Im begleitenden Programm ist ein Editor zur manuellen Erstellung von eindimensionalen Transferfunktion mithilfe einer Bezierkurve umgesetzt. Eine automatische Erstellung solcher

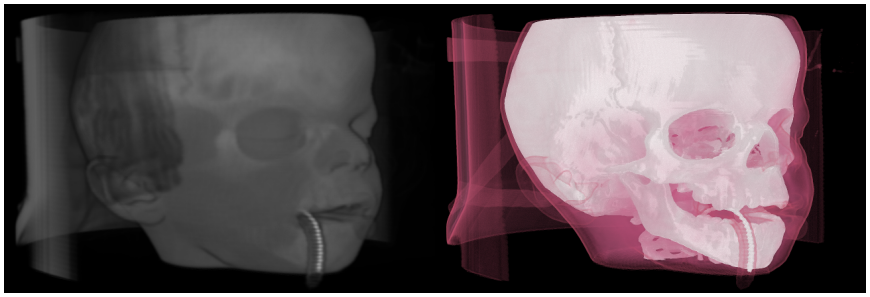

Abbildung 5: Beispiel einer Anwendung der Transferfunktion. Sowohl links als auch rechts handelt es sich um den selben Volumendatensatz [1], jedoch mit zwei unterschiedliche Transferfunktionen dargestellt.

Transferfunktionen ist problematisch, da es sich um eine Vielzahl von Parametern handelt und die Transferfunktion stark vom Einsatzszenario abhängig ist. In Bereichen wie der Medizin, bei denen es viele Szenarien gibt, die immerzu gleichartige Volumen liefern, machen halbautomatische Verfahren mit vorgegebenen Funktionen Sinn. Es müssen nur wenige Parameter geändert werden und es bleibt mehr Zeit für eine Beurteilung der Daten.

Bei der Wahl der Abtastrate sollte abgesehen von der räumlichen Auflösung der Volumendaten auch die Beschaffenheit der Transferfunktion bedacht werden. Da die Transferfunktion z.B. nur auf die abgetasteten Punkte angewendet wird, benötigen Unstetigkeiten in der Transferfunktion eine hohe Abtastrate des Volumens um korrekt abgebildet zu werden. Dieses Problem kann mit Pre-Integration aus Kapitel 3.8 behoben werden.

2.4 Ray-Casting

Um die vorhandenen Volumendaten auf eine zweidimensionale Ebene, wie den Bildschirm, zu projizieren gibt es verschiedene Möglichkeiten. Bei dem *bild-basierten* Verfahren des *Ray-Castings* wird pro Bildpunkt auf dem Bildschirm in Blickrichtung der Kamera ein Strahl durch die Volumendaten

geschickt und diese in vorgegebenen Abständen abgetastet. Die Technik ist konzeptionell nahe an der mathematischen Grundlage der diskreten Annäherung des *Volume-Rendering Integrals* und ermöglicht eine direkte Umsetzung der Front-To-Back Komposition. Heutzutage erlauben Grafikkarten eine unkomplizierte und sehr schnelle Realisierung dieser Technik, weshalb der Fokus dieser Arbeit auf der Umsetzung jener liegt. Das komplette Kapitel 3 widmet sich diesem Thema.

2.5 Slice Plane Rendering

Eine andere Technik für die Darstellung von Volumendaten ist aufgrund der Beschaffenheit früherer Grafikkarten zu verbreitetem Einsatz gekommen. Diese waren grundsätzlich auf das Rendering von Geometrie und Texturen spezialisiert, weshalb es sinnvoll gewesen ist das Volumen auf einer endlichen Anzahl von geometrischen Flächen scheibchenweise darzustellen und mit dem von der Hardware bereitgestelltem Alpha-Blending analog zur Back-To-Front Komposition zu verrechnen. Der simpelste Ver-

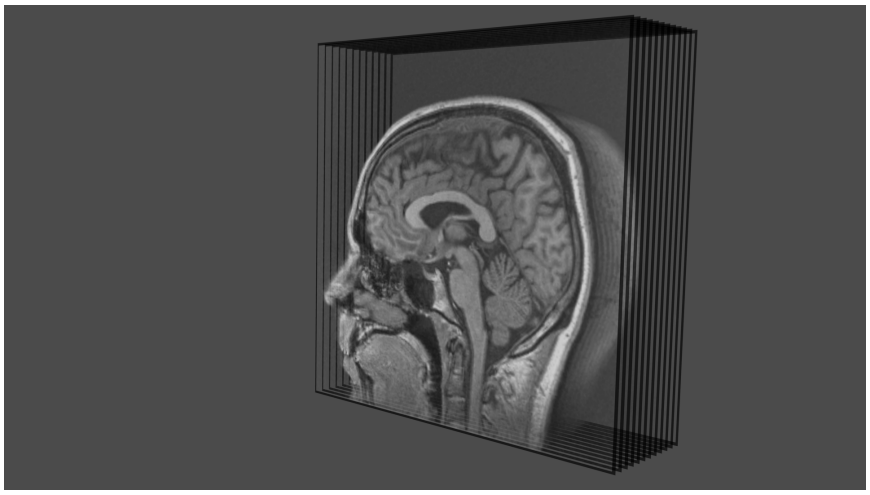

Abbildung 6: Slice Plane Rendering beispielhaft mit einem Volumendatensatz [2] dargestellt.

treter dieser *objekt-basierten* Technik benötigt drei Sets an Texturen, welche das Volumen auf den drei Hauptachsen x, y und z des Object-Space als Stapel von Bildern darstellen. Diese sogenannten *Slices* werden dann auf planaren Flächen gerendert, auch *Proxy-Geometrie* genannt. Abhängig vom Sichtwinkel der Kamera kommt eines der drei Sets zum Einsatz. Bei manchen Kamerabewegungen treten ruckartigen Bewegungen auf, da zwischen zwei Slice-Sets gewechselt werden muss und sich die Geometrie an einer anderen Hauptachse ausrichtet. Für eine Verbesserung der Darstellungsqualität gibt es verschiedene Techniken, welche ausführlich in “Real-

Time Volumes Graphics" [HKRs+06, Kapitel 3] dargelegt werden. So ist es möglich, extra Slices via Interpolation der vorhandenen Werte und dem Einsatz von mehr Geometrie zu erzeugen.

Sofern Volumendaten in Form einer 3D-Textur vorhanden sind und von der Grafikkarte verarbeitet werden können, kann planare Proxy-Geometrie in Richtung der Kamera gedreht werden. Dies wird zum Beispiel in einem Kapitel von "GPU Gems" [MI04] aufgezeigt. Durch den Einsatz des *View-Aligned Slicing* werden ruckartige Veränderungen der Geometrie unterbunden und ermöglicht so bei Kamerabewegungen um das Volumen eine flüssige Darstellung. Die Abtastweiten des Strahles durch die planaren Flächen sind jedoch bei den nun vorgestellten Techniken vom Winkel zur Blickrichtung hin abhängig, was zu verfälschten Darstellungsergebnissen zum Bildschirmrand hin führt.

Dieses Problem wird von einer Variante mit dynamischer Proxy-Geometrie umgangen. Anstatt planaren Flächen wird hier von halbkugelförmigen Formen Gebrauch gemacht, welche sich je nach Position und Blickrichtung der Kamera anpassen. Sie gewährleisten äquidistante Abtastabstände, sind aber aufwendig in der Implementation und benötigen ebenfalls Volumendaten in Form einer 3D-Textur, keine Sets von Slices.

2.6 Volume Rendering Pipeline

Sowohl Ray-Casting als auch Slice Plane basiertes Rendering gleichen sich in ihrer grundlegenden Pipeline. Diese kann, wie in "Real-Time Volume Graphics" [HKRs+06, Kapitel 1.6] aufgezeigt, beschrieben werden:

- **Datentraversierung**
  Abtasten der Werte im Volumen. Bei Slice Plane Rendering hängt die Abtastrate direkt von der Anzahl der benutzen Planes ab, bei Ray-Casting von der Abtastweite des Strahles. Siehe Kapitel 3.3.

- **Interpolation**
  Da meist nicht genau eine im uniformen Netz vorliegende Position getroffen wird, muss aus den am nächsten liegenden Positionen bzw. deren Werten ein neuer Zwischenwert interpoliert werden. Dazu wird oft auf trilineare Interpolation zurückgegriffen, da sie automatisch bei einem Texture-Look-Up im OpenGL Shader stattfindet und ausreichende Ergebnisse bei hoher Geschwindigkeit erzielt. Es soll aber nicht unerwähnt bleiben, dass es bessere Interpolationsmethoden z.B. mit Splines gibt, welche aber nicht in der Hardware abgebildet sind.

- **Berechnung des Gradienten**
  Der Gradient wird mithilfe des Umfeldes der Abtastposition berechnet, um in einem späteren Schritt Beleuchtung und andere gradien-

tenbasierte Techniken umsetzen zu können. Eine beliebte Methode zur Ermittlung des Gradienten ist eine Berechnung anhand von Zentraldifferenzen. Siehe Kapitel 3.12.

- **Klassifizierung**
  Hier werde die ausgelesenen bzw. interpolierten Volumenwerte mithilfe einer Transferfunktion zu Farben, Opazitäten und weiter Materialeigenschaften transformiert. Siehe Kapitel 3.4.

- **Beleuchtung und Shading**
  Die Beleuchtung kann zum Emissionswert des abgetasteten Punktes addiert werden, um eine bessere Tiefeneinschätzung und realistische Darstellung zu ermöglichen. Für die Berechnung ist zum einen der Gradient notwendig, zum anderen die Materialeigenschaften aus der Klassifizierung. Im begleitenden Programm sind in der Transferfunktion unter anderem Werte zur Farbe, Glanz und Emission angegeben. Abgesehen von der reinen Beleuchtung können auch weitere Shading-Effekte eingesetzt werden, wie z.B. den Alphawert abhängig von einem Fresnelterm zu machen oder die Magnitude des Gradienten für die Festlegung von Transparenz in homogenen Regionen zu nutzen. Siehe Kapitel 3.12.

- **Komposition**
  Jeder berechnete Punkt im Volumen muss mit den anderen Punkten auf dem Strahl verrechnet werden. Es gibt zwei Richtungen, in denen der Strahl das Volumen abtasten kann: Entweder Back-To-Front, also zur Kamera hin, oder Front-To-Back, weg von der Kamera. Die Ergebnisse sind gleichwertig, jedoch kann Front-To-Back bei Ray-Casting Optimierungen unterzogen werden, weshalb ausschließlich diese Richtung im weiteren Verlauf benutzt wird. Siehe Kapitel 3.3 und 3.5.

Die Reihenfolge wird sich so im Ray-Casting Shader wiederfinden und bildet den Kern des ganzen Algorithmus. Die Pipeline wird für jeden abgetasteten Punkt durchlaufen und stellt den Hauptaufwand beim Rendering dar. Deswegen wirkt sich jeder zusätzliche Schritt direkt auf die Performance aus und es sollte sichergestellt werden, dass nur notwendige Berechnung vollzogen werden. Wenn z.B. keine Shading stattfindet, wird der Gradient nicht benötigt und muss nicht berechnet werden.

3 Techniken und deren Implementierung

Im Folgenden wird die Umsetzung einer Ray-Casting basierten Darstellung von Volumendaten und mögliche Verbesserungen aufgezeigt. Es wird dabei auf OpenGL Shader in GLSL 3.3 gesetzt, welche auch im begleitenden Programm namens Voraca (**Vo**lume **Ra**y-**Ca**ster) eingesetzt wurden.

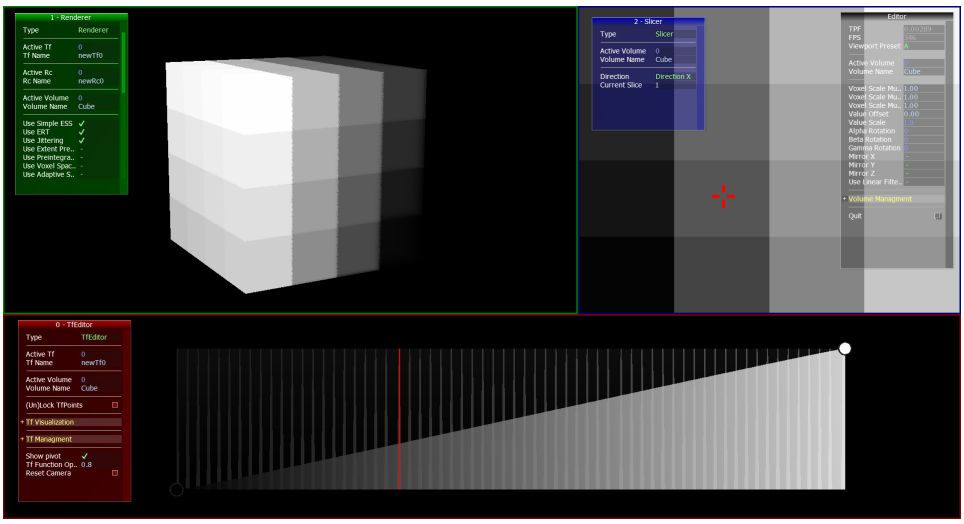

Abbildung 7: Voraca mit den Standardeinstellungen.

3.1 Aufbau des Programmes

Anforderungen an Voraca sind die Echtzeitanpassung der Transferfunktion und die Anwahlmöglichkeit der vorgestellten Techniken. Außerdem sollte es grundsätzlich plattformunabhängig programmiert sein, um es ohne weiteren Aufwand portieren zu können. Aufgrund dieser Anforderungen sind externe Bibliotheken benutzt worden, die für alle wichtigen Systeme verfügbar sind:

- **GLFW**[2] (OpenGL Fenstermanager)
- **GLEW**[3] (OpenGL Extensionmanager)
- **GLM**[4] (OpenGL Mathematikbibliothek)
- **RapidXML**[5] (XML Dateien lesen und schreiben)
- **AntTweakBar**[6] (Parametersteuerung via Bar)

[2] http://www.glfw.org/
[3] http://glew.sourceforge.net/
[4] http://glm.g-truc.net/
[5] http://rapidxml.sourceforge.net/
[6] http://anttweakbar.sourceforge.net/

- **stblib**[7] (Import von Texturen)

Des Weiteren ist ein Viewportsystem geschrieben worden, um verschiedene Kombinationen von Renderern, Transferfunktionseditoren und weiteren Typen von Viewports zu ermöglichen. Zum Zeitpunkt der Ausarbeitung ist der einzige weitere verfügbare Viewporttyp, neben dem Renderer und dem Editor für Transferfunktionen, ein sogenannter Slicer, mit welchem man durch die Rohdaten eines Volumen schichtweise navigieren kann. Jedes Viewport besitzt eine AntTweakBar, die Zugriff auf die Parameter des Viewports oder die Parameter der im Viewport zu bearbeitenden Daten gewährt. Außerdem besitzt das Editor Objekt selbst auch eine solche Bar und zeigt mit ihr unter anderem die Framerate an und dient zur Verwaltung der Volumen. Die Verwaltung der Transferfunktionen übernimmt der zugehörige Editor, die Möglichkeit zur Verwaltung der Raycaster Objekte wird von Renderer Viewports bereitgestellt. In Abbildung 8

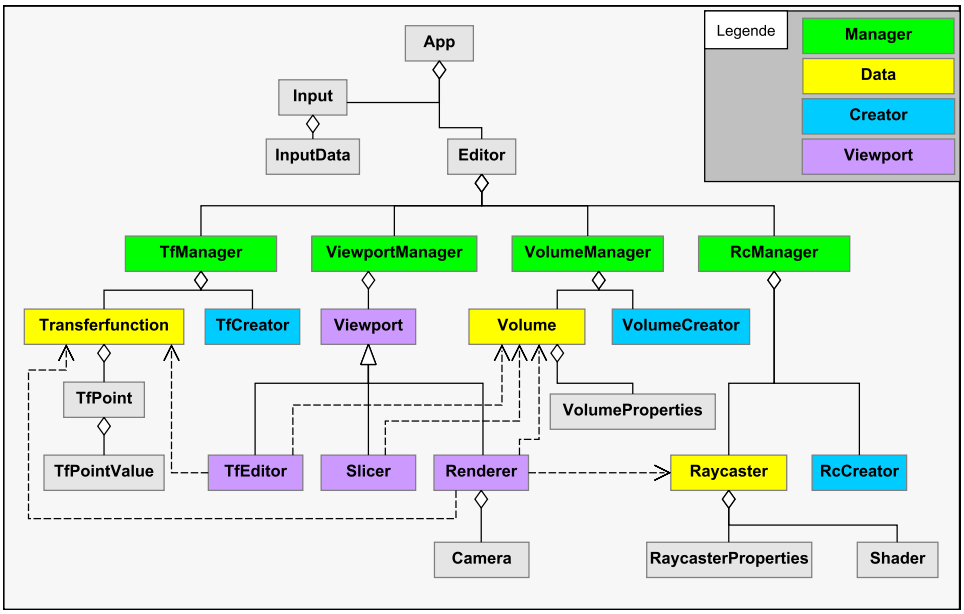

Abbildung 8: Klassenstruktur von Voraca.

ist die grobe Klassenstruktur des Programmes abgebildet. Es gibt jeweils eine Klassen für die Daten der Volumen, Transferfunktionen und Raycaster, welche jeweils über eine korrespondierende '-Creator' Klasse mit den initialen Werten gefüllt werden. Die Verwaltung von Instanzen dieser Klassen wird jeweils von einem Manager Objekt vorgenommen, das mithilfe von Handles Zugriff auf die Instanzen gewährt und intern eine `std::map` für die Speicherung nutzt. Diese Handles dienen in der Benutzeroberfläche auch dazu, in jedem Viewport das Volumen, die Transferfunktion oder

[7] http://code.google.com/p/stblib/

den Raycaster zur Anzeige oder Bearbeitung auszuwählen. Dies ermöglicht verschiedenen Viewports Daten aus einer gemeinsamen Basis anzuzeigen und unnötige Speicherauslastung zu vermeiden. Die AntTweakBars der Viewports ändern Werte nicht direkt in den Objekten, was zu Doppelstrukturen und einem gewissen Verwaltungsaufwand geführt hat. In der Retroperspektive wäre es vorteilhafter gewesen, die Adressen zu den Werten in den Objekten direkt an die Bars zu geben. Für die Messung der Performance können die zusätzlich entstehenden Kosten für die Verwaltung inklusive der AntTweakBars deaktiviert werden, um eine unverfälschte Messung zu erhalten.

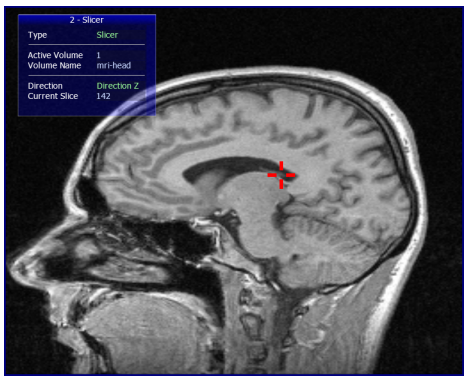

Abbildung 9: Volumen [2] in Slicer.

Für das Rendering mithilfe des Ray-Casting Algorithmus wird der Vertex- und Fragment-Shader der OpenGL Shader Pipeline benutzt, auf Geometry-Shader oder andere kann verzichtet werden. Da Shader nicht nur für den Ray-Casting Algorithmus sondern z.B. auch für den Editor der Transferfunktion benötigt werden, ist die Grundfunktionalität in eine Shader Klasse gekapselt worden. Instanzen dieser Klasse können ausschließlich nur Vertex- und Fragment-Shader aufnehmen und verarbeiten. Der Vertex-Shader nimmt dabei alle Eckpunkte eines Objektes als Eingabewert, bearbeitet diese mit dem gleichen Programm und übergibt sie dem Rasterisierer, der die Flächen zwischen den Eckpunkten interpoliert. Für jeden sichtbaren Bildpunkt der Flächen wird dann der Fragment-Shader aufgerufen, in welchem das eigentliche Ray-Casting stattfindet. Diese sehr vereinfachte Erklärung soll nur als Grundlage für die folgenden Sektionen dienen, weitere Informationen zu OpenGL und Shadern gibt es zum Beispiel in der OpenGL 3.3 Spezifikation[8] oder in Fachliteratur.

Da die im Folgenden vorgestellten Techniken in Voraca frei miteinander kombinierbar sein sollen, sind alle Techniken zusammen in einen einzigen Vertex- bzw. Fragment-Shader umgesetzt worden. Um einzelne davon zu aktivieren, werden nach dem Einlesen der Rohdateien bestimmte `#define`-Befehle vor den restlichen Inhalt geschrieben und erst dann kompiliert. Das hat zu einem über 600 Zeilen langen Fragment-Shader geführt, welcher nicht übersichtlich dargestellt werden kann. Für die Vorstellung der Techniken werden in den Listings ausschließlich Ausschnitte oder Entwicklungsvorstufen des Shaderpaares aus Voraca benutzt.

[8] https://www.opengl.org/registry/doc/glspec33.core.20100311.pdf, abgerufen am 10. März 2014

3.2 Datenstruktur

Als Struktur der Volumendaten werden uniforme Netze von beliebigen Ausmaßen vorausgesetzt. Diese lassen sich als 3D-Textur mit OpenGL in den Videospeicher laden und können mit `sampler3D` im Shader ausgelesen werden. Voraca ist in der Lage PVM-Dateien zu importieren, welche aus der Volumenbibliothek von Stefan Röttger[9] stammen. Die Daten liegen online jedoch komprimiert vor, weshalb sie zuerst mit Hilfe eines kleinen Tools namens 'pvmdds' des Versatile Volume Viewer[10] Projekts in unkomprimierte PVM-Dateien umgewandelt werden müssen. Das Format PVM3 hat laut der Beschreibung von Paul Bourke[11] den folgenden Aufbau:

- Format (im Moment wird von Voraca nur PVM3 unterstützt)
- Breite, Höhe und Tiefe des Volumens
- Relative Größe eines Voxels
- Anzahl der Bytes pro Voxel
- Binäre Volumendaten

Außerdem können noch DAT-Dateien importiert werden, wie vom "Institute of Computer Graphics and Algorithms" von der Technischen Universität Wien[12] bereitgestellt. Da es sich im Prinzip bei Volumendatensätzen immer um einen Header mit Metadaten und um eine Folge von unsigned byte (8-Bit) oder unsigned short (16-Bit) für die skalaren Werte der Voxel handelt, können andere Formate ohne großen Aufwand eingepflegt werden. Probleme stellen Kompressionsverfahren dar, für welche erst die passenden Bibliotheken gefunden und eingebunden werden müssen. Daher wurde auf einen Import von DICOM-Daten verzichtet und bei einer Veröffentlichung wird wahrscheinlich eine DAT-Datei als Beispieldatensatz genutzt, da es sich dort um unkomprimierte Rohdaten handelt.

Voraca bietet des Weiteren noch die Möglichkeit, importierte Volumendatensätze mit Parametern anzupassen und diese in Form einer XML zu speichern. Die Rohdaten des Volumens werden dabei ebenfalls gespeichert, indem sie in eine RAW-Datei ausgelagert werden. Die Anpassungen umfassen die Skalierung der Voxel für das Rendering, eine Verschiebung und Skalierung der Werte des Volumens, Rotation und Spiegelung des gesamten Volumes und eine Auswahl, ob linear- oder nearest-Filtering für die Interpolation zwischen Voxeln eingesetzt werden soll.

[9] http://www9.informatik.uni-erlangen.de/External/vollib/
[10] http://code.google.com/p/vvv/
[11] http://paulbourke.net/dataformats/pvm/
[12] http://www.cg.tuwien.ac.at/research/publications/2005/dataset-stagbeetle/

Implementierung

Das Auslesen und Interpretieren der Formate findet in der Klasse VolumeCreator statt. Diese befüllt eine Volume Objekt auf dem Heap mit den eingelesenen Daten und gibt es an den VolumeManager zurück, welcher es weiterhin verwaltet. Hier werden die Schritte beim Import einer PVM-Datei mit dem Format PVM3 und 8-Bit Tiefe nun skizziert.

Einlesen einer PVM Zuerst wird ein `std::ifstream` Objekt erzeugt um die Datei zu lesen. Falls die Datei erfolgreich eingelesen werden konnte, wird angefangen den Header auszulesen. Dabei ist zu beachten, dass es sich bei diesem Format um eine Mischung aus ASCII Zeichen am Anfang für die Metainformationen im Header und einen Block an binären Rohdaten für die Volumenwerte handelt. Daher wird die Datei mit der Flag `std::ios::binary` geöffnet und der Anfang des binären Flusses als ASCII-Zeichen interpretiert. In diesem Header liegen Informationen wie z.B. die Auflösung des Volumens vor. Hier wird beispielhaft das Format ausgelesen:

```
1  // Set up for reading
2  std::ifstream in("Dateiname.pvm", std::ios::in|std::ios::binary);
3  in.seekg(0, std::ios::beg);
4  GLchar buffer[5];
5
6  // Read file format
7  in.read(buffer, 4*sizeof(GLchar));
8  buffer[4] = 0;
9  std::string format(buffer);
10 if(format != "PVM3")
11 {
12   LogWarning("Cannot␣import␣anything␣else␣than␣PVM3");
13   return NULL;
14 }
15
16 // Read further header and save values to variables...
```

Listing 1: Einlesen der ersten Zeile des PVM-Headers.

Da die Informationen im Header eine fest vorgegebene Reihenfolge und Anzahl besitzen, ist bekannt ab wann die binären Rohdaten des Volumens im `ifstream` auftreten. Diese werden dann als kompletter Block in einen reservierten Speicherbereich geladen:

```
1  // Reservate memory for volume data
2  GLuint voxelCount = xResolution * yResolution * zResolution;
3  GLubyte* volumeData = (GLubyte*)calloc(voxelCount, sizeof(GLubyte));
4
5  // Read volume data
6  in.read((char*)volumeData, voxelCount * sizeof(GLubyte));
```

Listing 2: Volumendaten in reservierten Speicher laden.

Daraufhin wird eine 3D-Textur mit den Rohdaten befüllt und an ein Texture-Handle gebunden. Die Informationen aus dem Header, die Rohdaten und dieses Handle werden dann an ein neues Volume Objekt übergeben. Die Rohdaten werden nicht direkt gelöscht, da sie noch für Vorberechnungen zum adaptiven Sampling und das Speichern des Volumens ohne erneutes Einlesen benötigt werden. Außerdem wird noch ein Handle eingetragen, mit dem das Volume Objekt im Programm identifiziert werden kann:

```
// Generate, bind, set up, fill and unbind texture
GLuint textureHandle;
glGenTextures(1, &textureHandle);
glBindTexture(GL_TEXTURE_3D, textureHandle);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_CLAMP);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage3D(GL_TEXTURE_3D, 0, GL_LUMINANCE, xResolution, yResolution,
    zResolution, 0, GL_LUMINANCE, GL_UNSIGNED_BYTE, volumeData);
glBindTexture(GL_TEXTURE_3D, 0);

// Create volume object
Volume* pVolume = new Volume();

// Initialize volume
pVolume->init(handle, name, textureHandle, resolution, scale,
    valueResolution, volumeData);

// Volume raw data is freed by volume object at destruction
return pVolume;
```

Listing 3: OpenGL Textur mit Volumendaten füllen.

Für das Rendering wird dann der passende Shader gesetzt und mit den Daten aus dem Volume Objekt in Form von uniformen Variablen versorgt. Um die erzeugte 3D-Textur im Shader auslesen zu können, sind nur wenige Befehle notwendig:

```
#version 330 core
out vec4 fragmentColor;

uniform sampler3D uniformVolume;

void main()
{
  // Some position between (0,0,0) and (1,1,1)
  vec3 pos = vec3(0.2, 0.2, 0.2);

  // Get value from volume
  float val = texture(uniformVolume, pos).r;

  // Use it for coloring fragment
  fragmentColor = vec4(val.rrr, 1);
}
```

Listing 4: Fragment-Shader um einen Wert aus dem Volumen darzustellen.

Der gesamte Vorgang verteilt sich auf die Klassen VolumeCreator, Volume, VolumeManager und Renderer. Die Daten von einem Volumen sind in der Klasse Volume gekapselt, die anderen Klassen dienen zur Erstellung, Verwaltung und Darstellung der Daten.

3.3 Ray-Casting

Der Schritt von der 3D-Textur im Shader zur Umsetzung des Ray-Casting Algorithmus ist nun keine große Hürde. Der Aufbau einer Szene wird schematisch in Abbildung 10 dargestellt und bildet die Ausgangssituation für die folgenden Beschreibungen. Es wird angenommen, dass die Ausmaße des Volumens auf allen drei Hauptachsen gleich sind und das Volumen daher in einen Würfel ohne Stauchung oder Dehnung passt. Ein solcher Würfel wird dann als Proxy-Geometrie benutzt, um das Volumen mithilfe von Ray-Casting darstellen zu können. Technisch gesehen handelt es sich bei der Umsetzung mit Shadern nämlich nur um einen etwas spezielleren Effekt für die Darstellung der Oberfläche des Würfels. Anstatt nun z.B. mit Normal-Mapping Unebenheiten auf dem Würfel zu simulieren, werden hier iterativ die Werte aus der 3D-Textur mit den Volumendaten abgefragt und eine Komposition der Werte außen auf dem Würfel angezeigt. Dies hat den Nachteil, dass man die Kamera nicht in das Volumen bewegen kann. Man sieht von dort nicht die Außenseite des Würfels und daher auch keine Darstellung des Volumens. Dies kann mithilfe von komplexerer Proxy-Geometrie gelöst werden. Der Ray-Casting Algorithmus, der pro Punkt auf dem Bildschirm ausgeführt wird, ist im Folgenden mit Hilfe von Pseudocode umgesetzt. Es ist zu beachten, dass sich alle Positionen im gleichen Koordinatensystem befinden müssen:

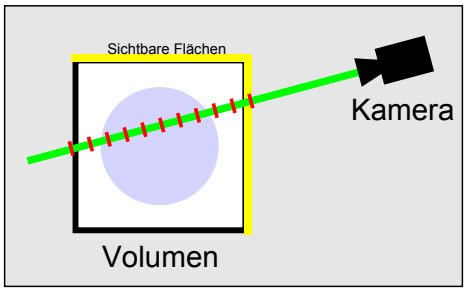

Abbildung 10: Einzelner Sichtstrahl von Kamera durch Volumen.

```
Richtung = PositionDesPixels - PositionDerKamera;
Position = PositionDesPixels;

Destination = 0;

for(i = 0; i < Iterationen; i++)
{
  Source = WertAusVolumen(Position);
  Destination = Komposition(Destination, Source);

  Position = Position + Richtung * Abtastweite;
```

```
    if(Position nicht mehr innerhalb von Würfel)
    {
      Abbruch;
    }
  }

  Fragmentfarbe = Destination;
```

Listing 5: Pseudocode zur Umsetzung des Ray-Casting Algorithmus.

Hier wird der Vereinfachung halber noch keine Transferfunktion eingesetzt, sondern die Daten aus dem Volumen als Grauwerte interpretiert und dargestellt. Zur Berechnung sind folgende Eingabewerte notwendig:

- Volumen in Form einer 3D-Textur
- Position des jeweiligen Pixels
- Position der Kamera

Der Algorithmus soll im Folgenden auf GLSL abgebildet und die Berechnungen auf einen Vertex- und Fragment-Shader verteilt werden. Dazu sind noch einige weitere Eingaben wie Matrizen und Vertex-Daten notwendig, damit der Algorithmus mithilfe von OpenGL respektive GLSL umgesetzt werden kann.

Implementierung

Aufseiten von C++ müssen die gleich vorgestellten Shader passend mit den eben aufgezeigten Werten gefüllt und weitere Matrizen für die Darstellung berechnet und eingesetzt werden. In den Vertex-Buffer wird ein einfacher Würfel geladen, dessen Ausmaße von 0 bis 1 auf allen drei Achsen reichen. Dies gewährleistet, dass der Model-Space gleich dem Texture-Space der 3D-Textur mit den Volumendaten ist. Dies vereinfacht Zugriffe auf diese enorm und ermöglicht später auch die Implementation von achsenparallelem Clipping ohne zusätzlichen Aufwand. Der Vertex-Shader übernimmt wie gewohnt die Transformation der eingespeisten Vertices und hat zusätzlich noch die Aufgabe, die Richtung des Kamerastrahles, der durch den jeweiligen Vertex führt, zu berechnen und dem Fragment-Shader weiterzureichen. Diese Berechnung findet direkt in Model-Space ergo Texture-Space statt:

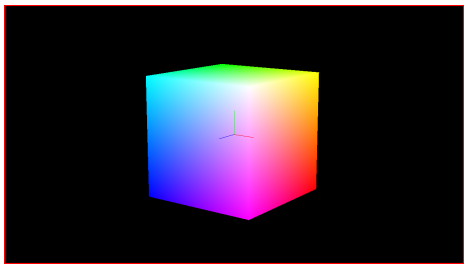

Abbildung 11: Würfel mit farblich kodierten Model-Space Positionen der von ihm gefüllten Bildpunkte.

```
1  #version 330 core
2  in vec4 positionAttribute;
3  out vec3 position;
4  out vec3 direction;
5
6  uniform mat4 uniformModel;
7  uniform mat4 uniformView;
8  uniform mat4 uniformProjection;
9
10 uniform vec4 uniformCameraPos;
11
12 void main()
13 {
14   // Vertex position for rasterizer
15   gl_Position = uniformProjection * uniformView * uniformModel *
         positionAttribute;
16
17   // Start position for rays (already model space)
18   position = positionAttribute.xyz;
19
20   // Direction in model space
21   direction = position - (inverse(uniformModel) * uniformCameraPos).xyz;
22 }
```

Listing 6: Vertex-Shader des Ray-Casting Programms.

Die Berechnung der Richtung des Kamerastrahles wäre zwar auch erst im Fragment-Shader möglich, aber teurer. Der Vertex-Shader wird pro Vertex ausgeführt, in diesem Fall 24-mal, und die Werte nach einer Interpolation an den Fragment-Shader weitergereicht. Dieser wird pro Bildpunkt des Objektes auf dem Bildschirm aufgerufen, was bei einer 10x10 großen Darstellung schon hundert Aufrufe ergibt. Da beide Vorgehensweisen das gleiche Ergebnis liefern, wird die Berechnung im Vertex-Shader ausgeführt. An Matrizen kommen keine Besonderheiten zum Einsatz, nur die bekannten Model-/View-/Projection-Matrizen für die Platzierung und Abbildung des Objektes im Raum. Die Model-Matrix kann im gegebenen Fall sogar mit einer Einheitsmatrix gefüllt werden, da keine Transformation im Raum stattfindet. Da die Position der Kamera in World-Space vorliegt, wird sie in Zeile 19 mit der inversen Model-Space Matrix transformiert. Dies gelingt aber nur, solange keine Translation stattfindet.

Nun wird nach der Berechnung des Vertex-Shaders und der Rasterisierung der Fragment-Shader aufgerufen, welcher das eigentlich Ray-Casting umsetzt und die berechneten Werte nach dem Durchlauf auf der Außenseite des Würfels anzeigt:

```
1  #version 330 core
2  in vec3 position;
3  in vec3 direction;
4
5  out vec4 fragmentColor;
6
7  uniform sampler3D uniformVolume;
8
```

```
9   const float STEP_SIZE = 0.008;
10  const float ITERATIONS = 1000;
11
12  void main()
13  {
14    // Use input from vertex shader
15    vec3 dir = normalize(direction);
16    vec3 pos = position;
17
18    // Variables for composition
19    float src;
20    float dst = 0;
21
22    // Do raycasting
23    for(int i = 0; i < ITERATIONS; i++)
24    {
25      // Get value from volume
26      src = texture(uniformVolume, pos).r;
27
28      // Front-To-Back composition (only alpha part)
29      dst += (1.0-dst) * src;
30
31      // Prepare for next sample
32      pos += dir*STEP_SIZE;
33
34      // Check whether still in volume
35      if(pos.x > 1 || pos.y > 1 || pos.z > 1 ||
36        pos.x < 0 || pos.y < 0 || pos.z < 0)
37      {
38        break;
39      }
40    }
41
42    // Output
43    fragmentColor = vec4(dst.rrr, 1);
44  }
```

Listing 7: Fragment-Shader des Ray-Casting Programms.

Eine Optimierung wert wäre unter anderem die Abfrage in Zeile 35f. darüber, ob die aktuelle Position innerhalb des Volumens liegt. Diese könnte in eine zusätzliche äußere Schleife ausgelagert werden. Dann kann der Compiler die innere Schleife mit festen x-Iterationen auffalten und es würde nur alle x-Iterationen eine if-Verzweigung auftreten, welche bei Shadern Leistungseinbußen mit sich bringen. Noch weitergehend könnte initial die Strahllänge ausgerechnet werden, die nötig ist das gesamte Volumen zu durchdringen und damit weitere Operationen eingespart werden. In den folgenden Beispielen werden solche Optimierungen der Übersichtlichkeit wegen aber größtenteils vernachlässigt. In Voraca gibt es einige Parameter, um den Ray-Casting Shader zu steuern, z.B. die inneren und äußeren Iterationen oder die Abtastweite. Auch die im weiteren Verlauf vorgestellten Techniken benötigen einige Parameter, welche sich im Programm über die AntTweakBars anpassen lassen. Alle dies Parameter und Einstellungen über die Aktivierung von Techniken werden in einem Raycaster Objekt gekapselt und können in Form einer XML abgespeichert werden.

Abbildung 12: Rendering eines Volumens [3] mit den vorgestellten Shadern.

3.4 Transferfunktion

Um aus den skalaren Werten innerhalb des Volumens natürliche Darstellungen zu generieren, wird auf die Technik der Transferfunktion zur Klassifikation der Volumendaten zurückgegriffen. Diesen wird der skalare Wert aus dem Volumen an der abgetasteten Stelle als Parameter übergeben und man bekommt daraufhin je nach Funktion Werte der Materialeigenschaften wie z.B. Farbe, Opazität, Reflexivität oder Emission zurückgegeben. Die erhaltenen Werte werden dann im Ray-Casting Algorithmus für die Komposition eingesetzt. Diese Transformation von Volumenwerten zu Materialeigenschaften wird mit Hilfe einer Look-Up-Tabelle, oder im Fall einer Implementation mit OpenGL mit einer 1D-Textur, realisiert. Der Eingabeparameter wird als Schlüssel interpretiert und dieser je nach Implementation auf einen vorhandenen Schlüssel gerundet oder die beiden benachbarten ausgelesen und zwischen deren Werten gemittelt.

Es wird hier davon ausgegangen, dass alle Volumenwerte zwischen 0 und 1 liegen und die Transferfunktionen immer zwischen 0 und 1 definiert sind. Dabei ist es nicht unbedingt notwendig, dass für ein 16-Bit Volumendatensatz auch eine für 16-Bit definierte Transferfunktion benutzt wird. Handelt es sich um einen Datensatz mit 8-Bit Tiefe, ist der Wertebereich genauso umfangreich wie bei einem Datensatz mit 16-Bit Tiefe, nur die Anzahl der Unterteilungen ist geringer. Damit kann auch ohne Probleme eine Transferfunktion mit einer Auflösung von 256 Werten (8-Bit) für einen 16-Bit Datensatz benutzt werden. Es werden dann viele Werte aus dem Datensatz auf die selben Schlüssel der Transferfunktion abgebildet und ergeben dadurch gleiche bzw. interpolierte Materialeigenschaften. Abgesehen von

dieser eindimensionalen Variante mit einem Parameter können beliebig weitere Eingabewerte hinzugefügt werden, wie z.B. bei einer zweidimensionalen Transferfunktion oft die Magnitudenstärke des Gradienten an der aktuellen Position im Volumen. Damit können dann Übergänge von Werte herausfiltert werden. In dieser Ausarbeitung wird auf eindimensionale Transferfunktionen eingegangen, weiteres zu zwei- oder mehrdimensionalen Transferfunktionen ist in "Real-Time Volume Graphics" [HKRs+06, Kapitel 10.3] zu finden. Auf eine Diskussion von Methoden zur automatischen Generierung einer Transferfunktion wird im Folgenden verzichtet und der Fokus ausschließlich auf eine manuelle Erzeugung gerichtet.

Integration in den Ray-Casting Algorithmus Es gibt es nun zwei sinnvolle Gelegenheiten, um die Transferfunktion in den bestehenden Algorithmus einzubringen. Bei der *pre-interpolativen* Anwendung der Transferfunktion werden die Werte im Volumen vor dem Durchgang des Ray-Casting Algorithmus transformiert und in einem oder mehreren Volumina abgelegt, je nachdem wie viele verschiedene Materialeigenschaften die Transferfunktion ausgibt. Für eine Abbildung von Farbe und Transparenz ist z.B. eine 3D-Textur mit jeweils einem Kanal für Rot, Grün, Blau und Alpha (RGBA) ausreichend, die aber schon viermal so viel Speicher benötigt wie das Ausgangsvolumen. Ein anderer Ansatz ist die *post-interpolative* Vorgehensweise, bei der keine Vorberechnung stattfindet. In jeder Iteration des Ray-Casting Algorithmus wird der interpolierte skalare Volumenwert in die Transferfunktion eingegeben und mit den resultierenden Werten der Materialeigenschaften fortgefahren. Das macht pro Iteration auf jedem Strahl durch das Volumen einen zusätzlichen Texture-Look-Up im Shader. Dadurch ergeben sich schwerwiegende Unterschiede zwischen beiden Herangehensweisen:

Pre-interpolative Anwendung ermöglicht ein schnelleres Rendering als die post-interpolative, benötigt aber mehr Speicher als das Ausgangsvolumen. In Abbildung 13 wird eine weiterer Unterschied deutlich, der sogar rechtfertigt die pre-interpolative Anwendung der Transferfunktion nicht im begleitenden Programm umzusetzen. In dieser Abbildung sind zwei Voxel, welche den maximalen Werteunterschied besitzen, schematisch von der Seite abgebildet. Geht nun der Kamerastrahl durch diese Voxel und tastet die interpolierten Volumenwerte ab, liefern pre- und post-interpolative Vorgehensweise zwei verschiedene Ergebnisse von unterschiedlicher Qualität. Es ist zu beachten, dass bei der Abtastung der Volumenwerte zwischen den beteiligten Voxeln gewichtet wird, um einen interpolierten Volumenwert zu erhalten. Bei der pre-interpolativen Variante stellen die vorhandenen Voxel schon den transformierten Wert dar und es werden die Materialeigenschaften, hier z.B. die Farbe, interpoliert. Da die Werte der Materialeigenschaften nur für die zwei Voxel vorliegen, sind nur zwei Pro-

ben aus der Transferfunktion vorhanden. Im dargestellten Fall fällt der blaue Anteil, welcher in der Transferfunktion um den skalaren Volumenwert von 0.5 vorliegt, komplett unter den Tisch. Bei der post-interpolativen Variante hingegen werden die skalaren Volumenwerte pro Abtastschritt interpoliert und daraufhin eine Probe aus der Transferfunktion genommen. Dadurch findet auch der blaue Anteil Einfluss in die Komposition. Ganz praktisch kann man sich Haut vorstellen, welche zwischen den beiden Volumenwerten für Fleisch und Luft vorliegt und nicht immer direkt durch die Voxel im Volumen abgebildet wird. Falls solche Werte bei Übergängen nicht beabsichtigt sind, kann auf eine zweidimensionale Transferfunktion mit Einbindung der Magnitudenstärke des Gradienten als zweiter Parameter zurückgegriffen werden.

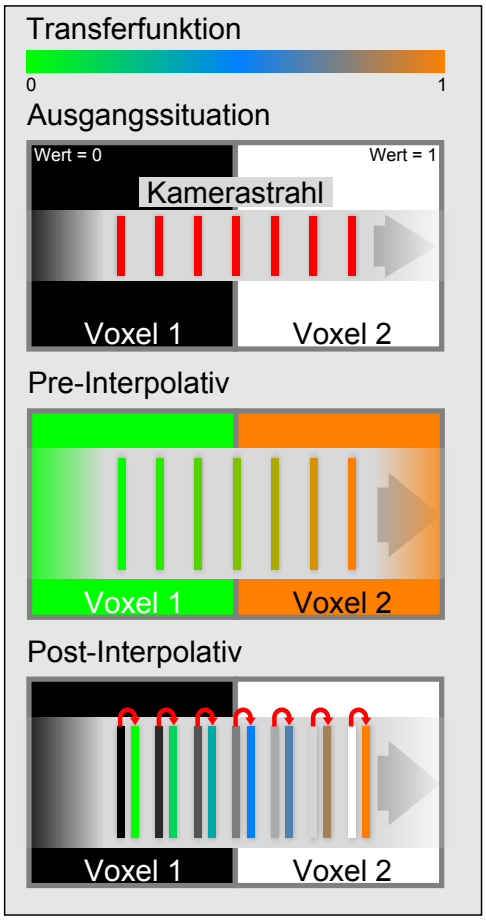

Abbildung 13: Unterschied zwischen pre- und post-interpolativen Anwendung der Transferfunktion.

Dieser Nachteil der pre-interpolativen Vorgehensweise kann auch nicht durch kleinere Abtastraten behoben werden, es müsste schlicht die Auflösung der transformierten Volumen erhöht werden. Das wiederum würde zu immensen Anforderungen an den Speicher führen. Die Abbildungsqualität der Transferfunktion bei der post-interpolativen Anwendung ist hingegen direkt von der Abtastrate abhängig, da Unstetigkeiten in der Transferfunktion nur angenähert dargestellt werden können. Das bedeutet, dass sowohl die Abtastung der Volumendaten selbst also auch die Abtastung der Transferfunktion den Gesetzen des Sampling unterliegt. Nur weil in einem Volumen in bestimmten Abschnitten nur geringe Differenzen bei den Werten der Volumendaten vorliegen, können in der Transferfunktion große Wertunterschiede bei den Materialeigenschaften sein, die bei weiten Abtastschritten nicht gut abgebildet würden. Das wird vor allem zu einem Problem, wenn adaptives Sampling eingesetzt wird, welches die Varianz der Werte im Volumen als

Gewichtung für die Abtastweiten nimmt. Ein Lösung stellen pre-integrierte Transferfunktionen dar, denen Kapitel 3.8 gewidmet und eine Implementation aufgezeigt wird.

Implementierung

Es gibt in Voraca einen Editor für eindimensionale Transferfunktionen, ein Editor für zweidimensionale Funktionen ist für die Vermittlung der Grundlagen nicht zwingend und sehr aufwendig zu implementieren. Da es sich dort um eine zweidimensionale Fläche handelt, ähneln Editoren eher Pixel- oder Vektorgrafikprogrammen, weil freie Formen und Farbverläufe notwendig sind. Wie sich in dieser Sektion zeigen wird, ist schon die Implementation der eindimensionalen Variante nicht trivial.

Punkte der Transferfunktion Transferfunktionen bestehen in Voraca aus manipulierbaren Punkten der Klasse TfPoint, welche die Werte der Materialeigenschaften beinhalten. Zwischen diesen Punkten wird interpoliert, um eine durchgehende Funktion zu erhalten. Folgende Materialeigenschaften können pro Punkt jeweils mit 32-Bit Genauigkeit eingestellt werden:

- **Position des Punktes**
  Hierbei handelt es sich um eine zweidimensionale Koordinate, wobei der x-Werte als Schlüssel dient, um Volumenwerten die Materialeigenschaften zuzuordnen. Der y-Wert wird als Opazität interpretiert, welche für den mit x angegebenen Volumenwert in die Komposition eingebracht wird. Praktisch gesehen gibt man bei Abfrage der Transferfunktion den x-Wert an, dessen zugeordnete Materialeigenschaften man erhalten möchte.

- **Position der Kontrollpunkte**
  Anstatt einer linearen Interpolation zwischen den Punkten wird auf kubische Bezierkurven gesetzt, die jeweils einen linken und rechten Kontrollpunkt benötigen. Von diesen wird nur die Position gespeichert.

- **Farbe**
  Es wird vereinfacht angenommen, dass Regionen im Volumen eine Farbe besitzen und diese sowohl für diffuse als auch direkte Reflektion und Emission von Licht gilt. Für die Speicherung wird das RGB-Format mit einer 32-Bit Genauigkeit pro Kanal benutzt.

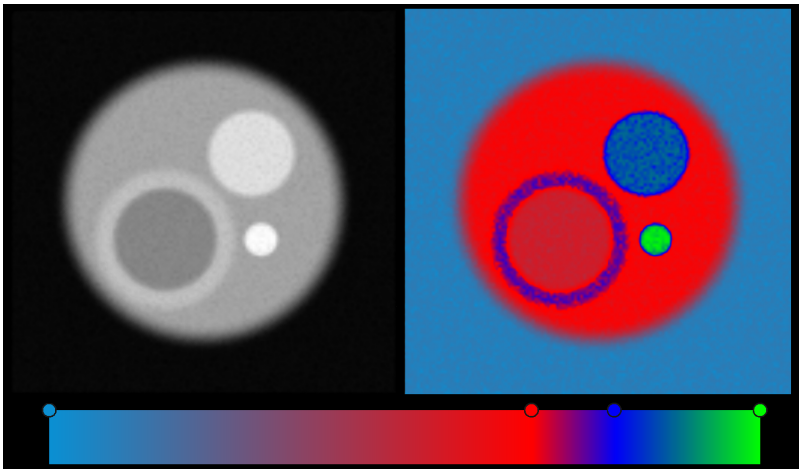

Abbildung 14: Anwendung des Farbwertes aus der Transferfunktion. Links die skalaren Volumenwerte als Grauwerte interpretiert. Rechts ist das selbe Volumen [4] abgebildet, jedoch dienen die Volumenwerte hier als Eingabeparameter für die unten abgebildete Transferfunktion.

- **Ambienter Multiplikator**
  Da keine globale Beleuchtung vorhanden ist, wird uniformes ambientes Umgebungslicht aufaddiert. Um Kontrolle über die Empfänglichkeit für solches Licht zu haben, wird bei jeder Komposition entlang des Strahles ein Wert aus der Transferfunktion mit dem ambienten Licht der Szene multipliziert. Dadurch können vereinfacht lichtdurchlässige und weniger durchlässige Materialien simuliert werden.

- **Spekularer Multiplikator**
  Direkte Reflektion von Licht aus der direktionalen Lichtquelle wird mit einem spekularen Term simuliert, dessen Ergebnis auf die vorhandene direkte und ambiente Beleuchtung aufaddiert wird. Um die Stärke des Glanzes zu steuern gibt es ebenfalls einen Multiplikator.

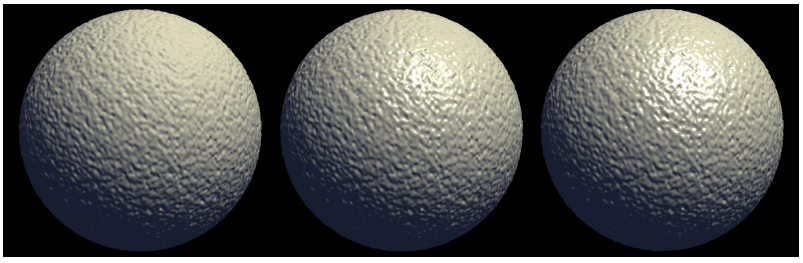

Abbildung 15: Effekt des spekularen Multiplikators auf Darstellung eines Volumens [4]. Dessen Wert ist links 0, in der Mitte 0.5 und rechts 1.

- **Spekulare Sättigung**
  Anstatt für den Glanz eine eigene Farbe zu definieren, was drei Werte

für eine Darstellung mit RGB benötigen würde, wird in Voraca davon ausgegangen, dass die Glanzfarbe direkt von der diffusen Farbe abhängt. Die spekulare Sättigung gibt an, wie viel Sättigung der diffusen Farbe in die Farbe des Glanzes einfließt.

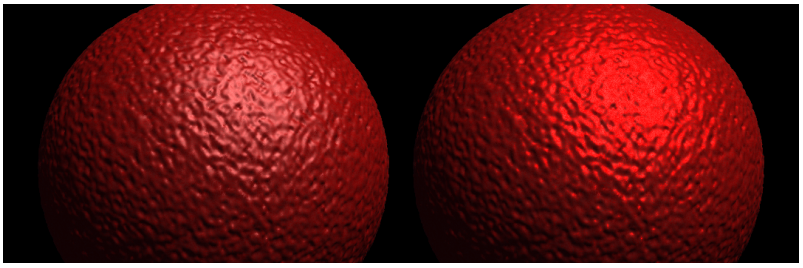

Abbildung 16: Effekt der spekularen Sättigung auf ein Volumen [4]. Deren Wert ist links 0 und rechts 1.

- **Spekularer Exponent**
  Die Größe des Glanzpunktes durch die spekulare Reflektion kann über einen Exponenten gesteuert werden. Hohe Werte resultieren dabei in kleinen Glanzpunkten, kleine Werte in großen. Mehr dazu in der Beschreibung des Phong-Beleuchtungsmodell in Kapitel 3.12.

- **Gradient-Alpha Multiplikator**
  Obwohl es sich nur um eine eindimensionale Transferfunktion handelt, welche als Eingabwert ausschließlich den Volumenwert benutzt, kann trotzdem die Magnitudenstärke des Gradienten in die Komposition Einfluss nehmen. In diesem Fall wird der Wert mit der vorhandenen Opazität bei einer einstellbaren Gewichtung multipliziert, was zu einer Darstellung ähnlich zu Schalen führt. So etwas wäre sonst nur mit einer zweidimensionalen Transferfunktion möglich.

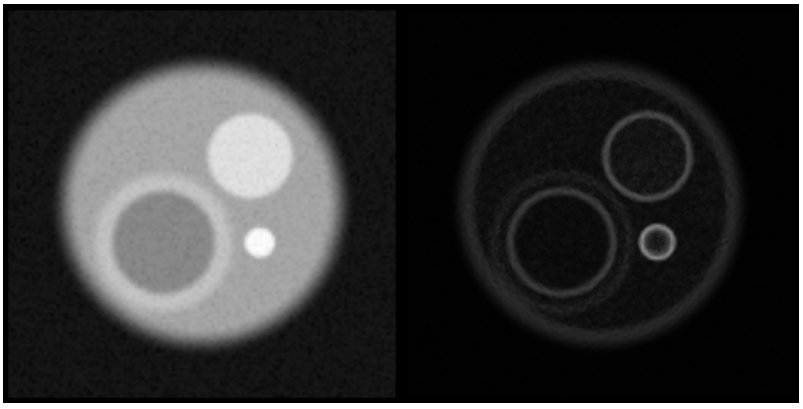

Abbildung 17: Das gleiche Volumen [4] wie in den oberen Abbildungen, ein Schnitt durch Kugeln in Kugeln. Rechts wird die Opazität mit der Magnitudenstärke des Gradienten gewichtet, was zu einen Effekt ähnlich zu Schalen führt.

- **Fresnel-Alpha Multiplikator**
  Falls gewünscht, wir die Opazität mit einem Fresnelterm multipliziert. Diese Materialeigenschaft aus der Transferfunktion dient zur Gewichtung der Operation.

- **Reflektion-Farbe Multiplikator**
  Um metallische Materialien zu simulieren, wird auf sogenannte Spherical Environment Maps zurückgegriffen. Der Wert aus dieser Map, die technisch gesehen ein normales Bild ist, wird mit dem Farbwert multipliziert. Die Operation wird mithilfe dieser Eigenschaft gewichtet.

- **Emission-Farbe Multiplikator**
  Leuchtende Materialien sind aufgrund der ausschließlich lokalen Beleuchtung nur unzureichend darstellbar. In Voraca wird zumindest eine Gewichtung eingesetzt, welche angibt wie viel der Farbe des Materials nach Anwendung der Beleuchtung auf die Farbe addiert wird. Dadurch können Materialien geschaffen werden, welche ihre Farbe zu einem gewissen Grad unabhängig von direktem und ambientem Licht aus der Umgebung wiedergeben.

Die gewählten Werte pro Punkt bzw. Kanäle der Transferfunktion entsprechen den persönlichen Geschmack und Erwartung an ein solches Programm. Natürlich könnten noch beliebig weitere Werte hinzugefügt und innerhalb der Komposition verwendet werden.

Erstellung der Transferfunktion Bei der Änderung eines Wertes oder wenn ein Punkt hinzugefügt, entfernt oder dupliziert worden ist, wird die Transferfunktion in Voraca neu berechnet. Dazu wird zwischen den Punkten auf eine Interpolation durch Bezierkurven zurückgegriffen. Es sind mehrere Schritte notwendig, um aus den gesetzten Punkten eine Transferfunktion zu erzeugen. Im Folgenden wird zwischen dem Begriff 'Punkt', als Bezeichnung für die vom Nutzer gesetzten Punkte der Transferfunktion und dem Begriff 'Bezierpunkt', der einen Punkt aus der errechneten Bezierkurve zwischen den Punkten der Transferfunktion beschreibt, unterschieden:

- **Abschätzen der Schrittzahl der diskreten Bezierkurve**
  Zuerst soll eine diskrete, kubische Bezierkurve aus den gesetzten Punkten der Transferfunktion und deren Kontrollpunkten erzeugt werden. Da die Punkte unterschiedlich weit voneinander entfernt sind, muss z.B. anhand des Abstandes auf der x-Achse geschätzt werden, wie viele Berechnungen der Bezierkurve zwischen jeweils zwei aufeinanderfolgenden Punkte notwendig sind um eine gesetzte Gesamtanzahl von Abtastungen nicht zu unterschreiten. Diese Schrittanzahl zum nächsten Punkt wird pro Punkt in einem `std::vector` gespeichert.

Die Schätzung funktioniert aber nur angemessen, wenn die Bezierkurve durch die Kontrollpunkte nicht extrem verzerrt wird. Die Position der Kontrollpunkte finden in diese Schätzung nämlich keinen Eingang .

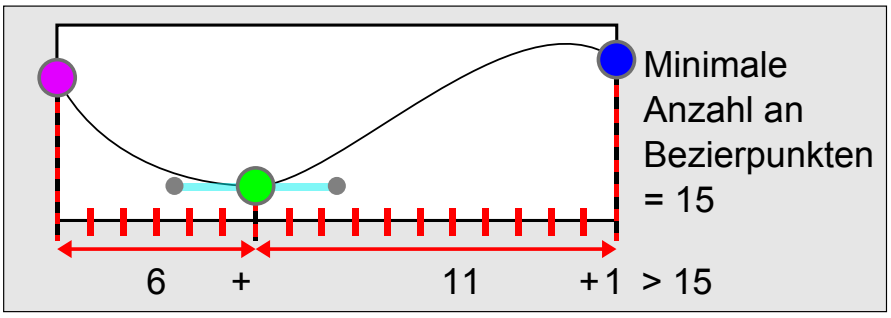

Abbildung 18: Grobes Abschätzen der Schrittzahlen für die Bezierkurve zwischen zwei Punkten über deren Abstand auf der x-Achse.

- **Berechnung der diskreten Bezierkurve**
  Für jeden Abschnitt zwischen zwei Punkten wird anhand der Positionen und denen der Kontrollpunkten ein Punkt in der Bezierkurve berechnet und folgende drei Werte gemerkt: x-Position, y-Position und Gleitkomma-Index. Dieser Index gibt an, wie viel Einfluss jeweils die beiden benachbarten Punkte der Transferfunktion auf dem Punkt der Bezierkurve haben. Wenn der Index 1.25 ist, hat der Punkt der Transferfunktion mit dem Index 1 einen Einfluss von 75% und der Punkt mit dem Index 2 einen Einfluss von 25%. Mit diesem Indikator lassen sich im nächsten Schritt die Materialeigenschaften gewichten und so interpolieren. Die drei Werte werden im weiteren Verlauf zusammengefasst als Bezierpunkt bezeichnet und als `glm::vec3` in einem `std::vector` gespeichert.

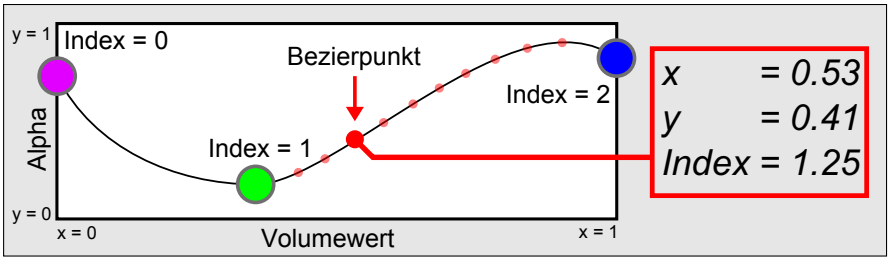

Abbildung 19: Darstellung eines Bezierpunktes und seiner Werte.

- **Transformation der Bezierkurve zur Transferfunktion**
  Nun wird über die Transferfunktion, bzw. derer diskreten Repräsentation durch `std::vector`, iteriert. Dies entspricht einer Iteration über die x-Achse, weshalb im `std::vector` mit den Bezierpunkten immer der Eintrag mit der nächstgrößeren x-Koordinate gesucht

wird. Dann wird ebenfalls der Eintrag des vorangehenden Bezierpunktes ausgelesen und zwischen beider y-Koordinaten und Gleitkomma-Indices gemittelt. Die gemittelte y-Koordinate dient als Alphawert der Transferfunktion an dieser Stelle. Der gemittelte Gleitkomma-Index wird benutzt, um alle anderen Materialeigeschaften beider beteiligten Punkte der Transferfunktion zu gewichten und in der diskreten Repräsentation dieser abzulegen.

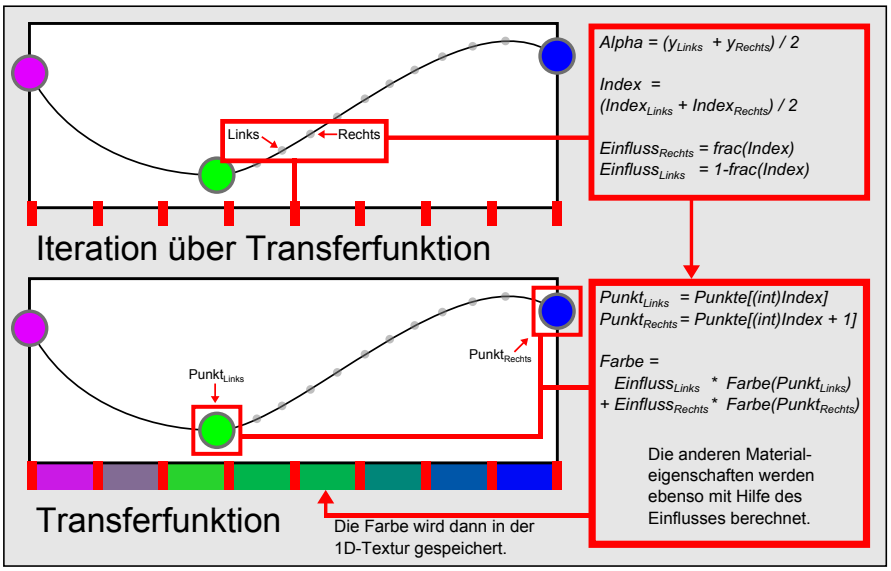

Abbildung 20: Aus den Bezierpunkten wird eine Transferfunktion.

- **Speichern als Textur**
  Im Fall von Voraca kommen für die Speicherung der diskreten Transferfunktion drei `std::vector` mit `glm::vec4` zum Einsatz. Diese können für das Befüllen von 1D-Texturen benutzt werden, welche im Shader ausgelesen werden sollen.

Es handelt sich beim Schritt von der Bezierkurve zur Transferfunktion um eine potentiell verlustbehaftete Abbildung, da eine Bezierkurve für einen gleichen x-Werte mehrere y-Werte erlaubt. Außerdem ist es unschön, dass erst eine diskrete Bezierkurve erstellt wird und diese dann nochmal für die Erstellung der Transferfunktion abgetastet werden muss. Die Ergebnisse sind jedoch überzeugend und bieten mehr Freiheiten als eine rein lineare Interpolation. Der Programmcode wird aus Gründen der Übersichtlichkeit hier nicht aufgelistet, kann aber innerhalb der Funktion `updateFunction()` der Klasse Transferfunction gefunden werden.

Integration in den Shader Die Transferfunktion wird dem Ray-Casting Fragment-Shader in Form mehrerer eindimensionaler Texturen übergeben, die anhand des Wertes aus dem Volumen ausgelesen werden. Hier wird

beispielhaft die 1D-Textur mit Farb- und Alphawerten aus der Transferfunktion in den Shader eingepflegt:

```
1  #version 330 core
2  in vec3 position;
3  in vec3 direction;
4
5  out vec4 fragmentColor;
6
7  uniform sampler3D uniformVolume;
8  uniform sampler1D uniformTransferfunction;
9
10 const float STEP_SIZE = 0.008;
11 const float ITERATIONS = 1000;
12
13 void main()
14 {
15   // Use input from vertex shader
16   vec3 dir = normalize(direction);
17   vec3 pos = position;
18
19   // Variables for composition
20   vec4 src;
21   vec4 dst = vec4(0,0,0,0);
22   float value;
23
24   // Do raycasting
25   for(int i = 0; i < ITERATIONS; i++)
26   {
27     // Get value from volume
28     value = texture(uniformVolume, pos).r;
29
30     // Use value as coodinate in transferfunction
31     src = texture(uniformTransferfunction, value).rgba;
32
33     // Front-To-Back composition
34     dst.rgb += (1.0-dst.a) * src.rgb * src.a;
35     dst.a += (1.0-dst.a) * src.a;
36
37     // Prepare for next sample
38     pos += dir*STEP_SIZE;
39
40     // Check whether still in volume
41     if(pos.x > 1 || pos.y > 1 || pos.z > 1 ||
42       pos.x < 0 || pos.y < 0 || pos.z < 0)
43     {
44       break;
45     }
46   }
47
48   // Output
49   fragmentColor = vec4(dst.rgb, 1);
50 }
```

Listing 8: Kompletter Fragment-Shader mit Abfrage der Transferfunktion für die Farbe und Opazität.

Es handelt sich hier um eine post-interpolative Anwendung der Transferfunktion, da beim Texture-Look-Up in Zeile 28 ein Volumenwert interpoliert wird und dieser als Eingabe für die Transferfunktion dient. Der inter-

polierte Wert aus dem Volumen ist nun die Eingabe für die Transferfunktion, die einen vierdimensionalen Vektor zurückgibt. Dieser Vektor wird in diesem Fall dann als Farb- und Alphawert interpretiert. In Voraca kommen noch zwei weitere Texturen zum Einsatz, welche die anderen acht Materialeigenschaften der Transferfunktion repräsentieren. Auch diese Texturen nehmen den Wert aus dem Volumen als Eingabe.

Editor in Voraca Pro gesetzten Punkt der Transferfunktion sind alle Werte der Materialeigenschaften gespeichert und können in Echtzeit über die AntTweakBar des Editors angepasst werden. Des Weiteren sind noch direkte Manipulationsmöglichkeiten innerhalb des Viewports mit der Maus umgesetzt worden. Mit Hilfe der gedrückten linken Maustaste lassen sich die Punkte und deren Kontrollpunkte verschieben. Die mittlere Maustaste dient der Navigation und das gedrückte Halten von Steuerung ermöglicht eine Mehrfachselektion von Punkten. Weitere Tastenbelegungen sind der beiliegenden Anleitung zu entnehmen. Im Hintergrund der Transferfunktion wird außerdem das Histogramm und der aktuelle Pivot eines gewählten Volumens angezeigt. Der Pivot wird für ein Volumen im Slicer Viewport ausgewählt. Die Transferfunktionen kann, wie die Einstellungen von Volumen, in Form einer XML abgespeichert und wieder eingelesen werden. In diese XML werden ausschließlich die Punkte, deren ein oder zwei Kontrollpunkte und die Materialeigenschaften geschrieben. Die Erzeugung der Transferfunktion geschieht dann automatisch nach dem Laden.

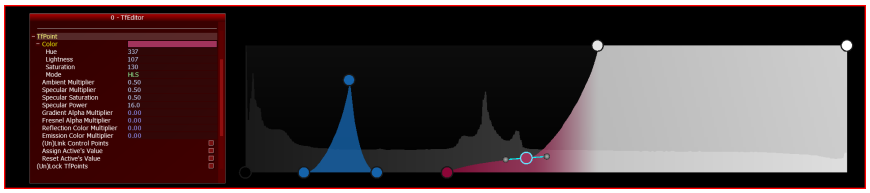

Abbildung 21: Editor für Transferfunktionen mit AntTweakBar in Voraca.

3.5 Early Ray Termination

Die Verwendung von Ray-Casting bietet viele Möglichkeiten der Verbesserung, sowohl in visueller Hinsicht als auch vonseiten der Performance. Da es sich um Front-To-Back Komposition handelt, kann durch *Early Ray Termination* (ERT) der Algorithmus gestoppt werden, sobald die Opazität einen festgelegten Schwellwert überschritten hat. Da alle Strahlen unabhängig berechnet werden, können einzelne Strahlen bzw. Fragment-Shader verfrüht ihr Ergebnis in den Framebuffer schreiben und auf der Berechnungseinheit Platz für die nächste Aufgabe machen. Diese Technik der Early Ray Termination ist mit dem sogenannten *Occlusion Culling* vergleichbar,

welches verhindert, dass verdeckte Geometrie unnötigerweise gerendert wird. Bei Back-To-Front Komposition wäre dieses Verfahren nicht umsetzbar. Man kann sich bei dieser Reihenfolge nicht sicher sein, ob zur Kamera hin noch Regionen im Volumen auftreten, welche die bisher durchdrungenen Anteile überdecken würden.

Implementierung

Es wird zuerst die Integration in den Fragment-Shader aus dem Listing 8 gezeigt und dann der Zugewinn an Performance gemessen.

Integration in Shader Einen Abbruch der Iterationen bei Übertretung eines gesetzten Schwellwertes für die Opazität ist leicht umzusetzen. Zu dem schon bestehenden Abbruchkriterium bei Austritt des Strahles aus dem Volumen wird das neue Kriterium parallel eingefügt und ebenfalls bei jeder Iteration geprüft:

```
#version 330 core
...
const float ALPHA_THRESHOLD = 0.95;

void main()
{
  ...

  // Do raycasting
  for(int i = 0; i < ITERATIONS; i++)
  {
    ...

    // Front-To-Back composition
    dst.rgb += (1.0-dst.a) * src.rgb * src.a;
    dst.a += (1.0-dst.a) * src.a;
    ...

    // Early Ray Termination
    if(dst.a > ALPHA_THRESHOLD)
    {
      break;
    }
  }
  ...
}
```

Listing 9: Early Ray Termination im Fragment-Shader.

In Voraca sind beide Abbruchkriterien in einer äußeren Schleife ausgelagert, damit sie nicht, wie im Beispiel oben, bei jeder Iteration geprüft werden und dadurch einen erheblichen Aufwand pro Iteration ausmachen.

Performance Die Performance wird nicht mit dem vorgestellten Shader gemessen, sondern mit dem Ray-Casting Shader aus Voraca. Dieser führt

Abbildung 22: Testaufbau für die Messung von ERT in Voraca. Es werden zwei verschiedene Transferfunktionen auf dem selben Volumen [5] benutzt, die jeweils die Haut bzw. Knochen visualisieren.

mehr Operationen pro Iteration durch, was zu einem aussagekräftigeren Ergebnis führt. Der Testaufbau ähnelt dem aus "Real-Time Volume Graphics" [HKRs+06, Kapitel 8.4] und ermöglicht weitgehendere Schlussfolgerungen. Wie in Abbildung 22 zu sehen, wird auf lokale Beleuchtung und weitere Techniken zur visuellen Aufbereitung zurückgegriffen. Bei einer Messung kommt es zu folgenden Ergebnissen:

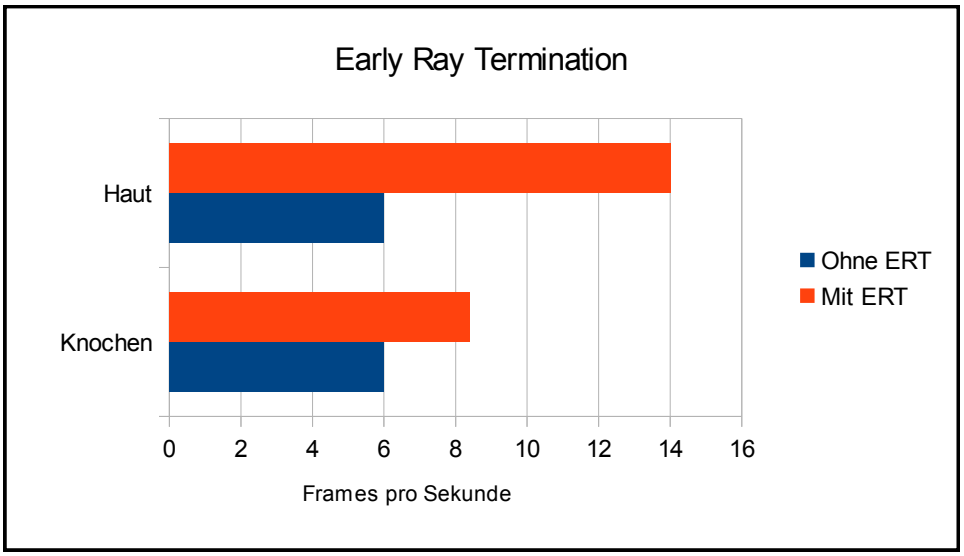

Abbildung 23: Messung der Performance von Early Ray Termination.

Hier fällt besonders bei der Darstellung der Haut ein bemerkenswerter Gewinn an Performance auf. Die Iteration im Shader kann nämlich stoppen,

sobald der Kamerastrahl die Haut erreicht und die Opazität über den gesetzten Schwellwert kommt, welcher bei dieser Messung 0.95 ist. Bei der Darstellung der Knochen des Fußes fällt diese Optimierungstechnik dagegen nicht sonderlich ins Gewicht, da mehr durchsichtige Volumenanteile vorhanden sind, welche komplett abgetastet werden und pro Iteration alle Berechnungen stattfinden. Dies kann mit *Empty Space Skipping*, das in Kapitel 3.6 vorgestellt wird, enorm verbessert werden. Der Schwellwert für das Early Ray Termination sollte nicht zu klein gewählt werden (< 0.9), da es sonst zu einem verfrühten Abbruch kommt. Dadurch können eigentlich sichtbare Anteile des Volumens verschluckt werden. Ein hoher Schwellwert von nahezu 1 ist ebenfalls nicht empfehlenswert, da die Opazität aufgrund der Komposition gegen 1 konvergiert. Diese Schranke wird daher nicht unbedingt oder sehr spät aufgrund von Ungenauigkeiten erreicht.

3.6 Empty Space Skipping

Die eigentlich interessanten Volumendaten sind meist von niedrigen, gleichförmigen Werten umgeben, die oft in der Transferfunktion als unsichtbare Region klassifiziert werden. Nun macht es in einer unsichtbaren Region keinen Sinn, Gradienten zu ermitteln, zu beleuchten und weitere aufwändige Berechnungen durchzuführen. Die Komposition gewichtet die erhaltenen Werte nämlich mit 0, weshalb sie keinen Einfluss auf den finalen Farbwert des Bildpunktes haben. Deshalb wäre es besser, so wenig Aufwand wie möglich in unsichtbare Regionen zu verwenden.

Bei [HKRs+06, Kapitel 8.4] wird ein Verfahren namens *Empty Space Leaping* vorgestellt, bei welchem das Volumen mithilfe von mehreren, kleineren und gleichförmigen Würfel-Proxy-Geometrien in Gitteranordnung dargestellt wird. Für jeden dieser Würfel wird initial der minimale und maximale skalare Volumenwert herausgefunden, den er vom Volumendatensatz beinhaltet. Bei jeder Änderung der Transferfunktion wird dann mit einem gesetzten Schwellwert geprüft, ob der Inhalt des jeweiligen Würfels überhaupt sichtbar ist und in die Komposition einfließen würde. Falls dies nicht der Fall ist, wird der ganze Würfel verworfen und es entsteht im Fragment-Shader keinerlei Aufwand. Statt dem Emtpy Space Leaping kommt in Voraca eine stark vereinfachte Variante zum Einsatz, die trotzdem Berechnungen einsparen kann und dabei sehr übersichtlich zu implementieren ist. Sie wird hier als *Empty Space Skipping* (ESS) bezeichnet, da sie alle unnötigen Berechnungen überspringt und die Abtastung des nächsten Punktes auf dem Kamerastrahl anstößt.

Implementierung

Analog zur Implementation von Early Ray Termination wird zuerst die Integration in den Fragment-Shader mit dem Ray-Casting Algorithmus ge-

zeigt und dann am Beispiel des Testaufbaus aus Abbildung 22 eine Messung der Performance durchgeführt.

Integration in Shader Die Abfrage des Schwellwertes geschieht direkt nach Anwendung der Transferfunktion und vor den anderen Berechnung und der Komposition. Ist der Schwellwert vom erhaltenen Alphawert unterschritten, wird die Position auf dem Strahl für die nächste Iteration erweitert und dann via `continue`-Befehl eine neue Iteration ausgelöst, ohne die weiteren Operationen der aktuellen Iteration durchzuführen:

```
#version 330 core
...
const float EMPTY_SPACE_SKIPPING_THRESHOLD = 0.001;

void main()
{
  ...

  // Do raycasting
  for(int i = 0; i < ITERATIONS; i++)
  {
    // Get value from volume
    value = texture(uniformVolume, pos).r;

    // Use value as coodinate in transferfunction
    src = texture(uniformTransferfunction, value).rgba;

    // Empty Space Skipping
    if(src.a < EMPTY_SPACE_SKIPPING_THRESHOLD)
    {
      // Prepare for next sample and continue
      pos += dir*STEP_SIZE;
      continue;
    }
    ...

    // Operations like shading etc.
    ...

    // Front-To-Back composition
    dst.rgb += (1.0-dst.a) * src.rgb * src.a;
    dst.a += (1.0-dst.a) * src.a;
    ...
  }
  ...
}
```

Listing 10: Empty Space Skipping im Fragment-Shader.

Für die Umsetzung der Technik ist, wie in Listing 10 zu sehen, nur eine kleine Änderung in der Schleife notwendig. In ihr muss ein konstanter Wert als Schwellwert gesetzt werden. Im Allgemeinen ist es ausreichend, diesen Wert direkt im Shader zu setzen und ihn nicht über die Benutzeroberfläche manipulierbar zu machen. Sobald der Nutzer der Transferfunktion einen Alphawert größer Null zuweist, wird auch eine Darstellung erwartet. Nur

wenn der Wert sehr nahe Null liegt, kann angenommen werden, dass keine Darstellung erwartet wird.

Performance Der Testaufbau aus Abbildung 22 bildet auch bei dieser Technik einen aussagekräftigen Rahmen. Eine Messung führt zu folgenden Ergebnissen:

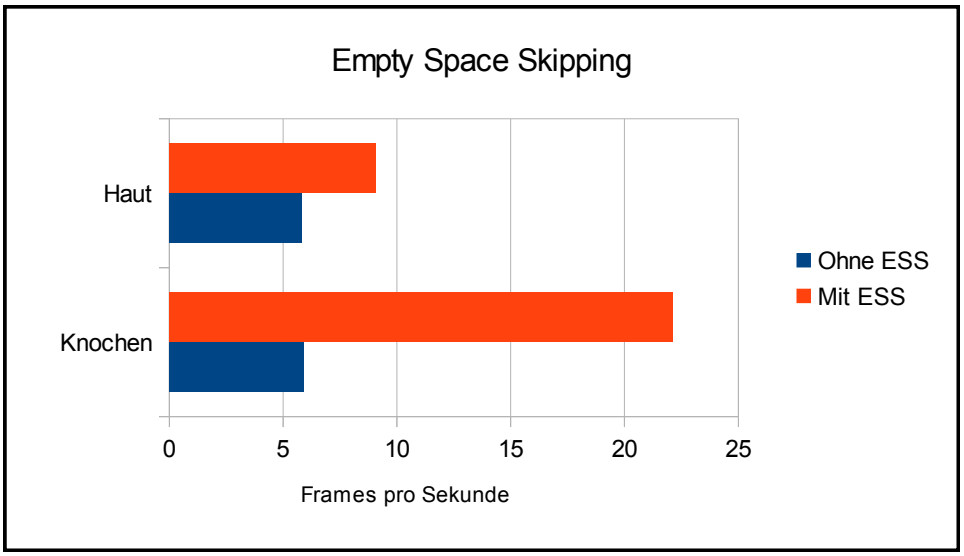

Abbildung 24: Messung der Performance von Empty Space Skipping am Testaufbau aus Abbildung 22.

In beiden Fällen bringt Empty Space Skipping einen Vorteil in der Bildrate. Besonders bei der Darstellung der Knochen fällt die verbesserte Performance auf, welche dank der Technik erzielt werden kann. Wenn man diese Messung mit der der Early Ray Termination in Abbildung 23 vergleicht, fällt ein gegensätzliches Verhalten auf. Early Ray Termination profitiert von massiven Regionen, wogegen Empty Space Skipping vor allem bei durchsichtigen Regionen einen großen Vorteil in der Performance aufweist. Beide können kombiniert eingesetzt werden, was in Kapitel 4 auch vorgenommen wird. Ein Problem stellen aber weiterhin halbtransparente Regionen dar, weil sie von beiden Verfahren unoptimiert bleiben. Wirklich viel kann man dort in Sachen Performance auch nicht tun, es gilt eher mit so wenig Abtastpunkten wie möglich ein angemessenes Ergebnis zu erzielen.

3.7 Stochastic Jittering

Wenn die vorgestellte Umsetzung des Ray-Casting Algorithmus als Eingabe eine Transferfunktion bekommt, die eine scharfe Kante zwischen Luft und Material abbildet, kommt es zu unübersehbaren Artefakten. Diese Ar-

tefakte sind ähnlich einer Holzmaserung und rühren von der Verletzung des *Nyquist-Shannon Theorem* her. Dieses Theorem besagt, dass die Abtastfrequenz mindestens doppelt so hoch sein sollte wie die maximale, im abgetasteten Raum vorkommende Frequenz. In einem Volumendatensatz ist die kleinste Distanz die zwischen zwei Voxeln, da zwischen diesen ein maximaler Werteunterschied vorkommen kann, sowohl in Form der skalaren Volumenwerte als auch durch die Transformation einer Transferfunktion. Die Abtastweite müsste mindestens kleiner als die Hälfte der räumlichen Distanz sein. Nur so werden zumindest die Volumenwerte perfekt abgetastet, die Transferfunktion jedoch noch nicht unbedingt. Dies ist selbst mit moderner Hardware bei den meisten Datensätzen nicht in interaktiver Form umsetzbar. Stattdessen wendet man einige Optimierungen an, von denen *Stochastikc Jittering* eine der effektivsten ist. Um eine zu niedrige

Abbildung 25: Typische Artefakte bei einer zu niedrigen Abtastrate eines Volumens [6].

Abtastrate zu verschleiern und die ungewollte Holzmaserung auf Oberflächen zu unterbinden, wird die Abtastposition initial um einen zufälligen Wert expandiert. Dadurch kommt es zwar zu einem Rauschen, dieses ist für das menschliche Auge aber weniger störend als die gleichmäßigen Strukturen, die bei niedriger Abtastrate auftreten. Durch diese Technik wird die Abtastrate nicht erhöht, das Ergebnis auf dem Bildschirm ist also keineswegs genauer. Daher ist der Aufwand für dieses Verfahren pro Frame sehr gering, da er zwar pro Bildpunkt, aber außerhalb der Iterationen entsteht.

Implementierung

Die Umsetzung in "Real-Time Volume Graphics" [HKRs+06, Kapitel 9.1.4] benutzt ein gleichverteiltes Rauschen, welches in Form einer 2D-Textur in den Shader gegeben wird. In Voraca wird stattdessen ein normalverteiltes Rauschen verwendet, was einen leicht höheren Aufwand bei der Erzeugung der Textur bedeutet, jedoch potentiell bessere Ergebnisse liefert. Das Rauschen wird in eine 2D-Textur geschrieben und im Fragment-Shader mit

dem Ray-Casting Algorithmus als Ersatz für einen Zufallszahlengenerator eingesetzt. Eine Texturkoordinate zum Auslesen des Zufallswertes wird von der Position des Fragments bzw. Bildpunktes im Framebuffer geliefert, die sich im Shader mit der eingebauten Variable `gl_FragCoord` auslesen lässt. Der einigermaßen zufällige Wert aus der Textur wird mit dem 3D-Vektor der Richtung des Kamerastrahles und der Abtastweite multipliziert. Das Ergebnis wird dann auf die initiale Position der Abtastung vor Start der Iterationen aufaddiert:

```
#version 330 core
...
uniform sampler2D uniformNoise;

// Resolution of noise texture
const float NOISE_RES = 64;

void main()
{
  ...

  // Read random value from noise texture
  float jitter = texture(uniformNoise, gl_FragCoord.xy/NOISE_RES).r;

  // Expand ray before start of iterations
  pos += dir * STEP_SIZE * jitter;

  // Do raycasting
  for(int i = 0; i < ITERATIONS; i++)
  {
    ...
  }
  ...
}
```

Listing 11: Stochastic Jittering im Fragment-Shader.

Bevor das Ergebnis hier gezeigt wird, soll noch eine kleine Erweiterung der Technik, die ebenfalls in Voraca Einsatz findet, grob beschrieben werden. Bei Tests während der Programmierung ist aufgefallen, dass das Stochastic Jittering an den Kanten der Proxy-Geometrie sehr auffällige, rauschende Artefakte erzeugt. Da im Ray-Casting Shader von Voraca pro Iteration die Information vorhanden ist, wie weit der Strahl schon durch das Volumen vorangekommen ist und wie weit es noch bis zum Austritt aus der Proxy-Geometrie ist, kann damit die Verschiebung des Abtastpunktes durch das Jittering vom Rand der Geometrie an eingeblendet werden. Dadurch kommt Stochastic Jittering erst innerhalb der Proxy-Geometrie zum Einsatz und Schnittkanten bleiben scharf:

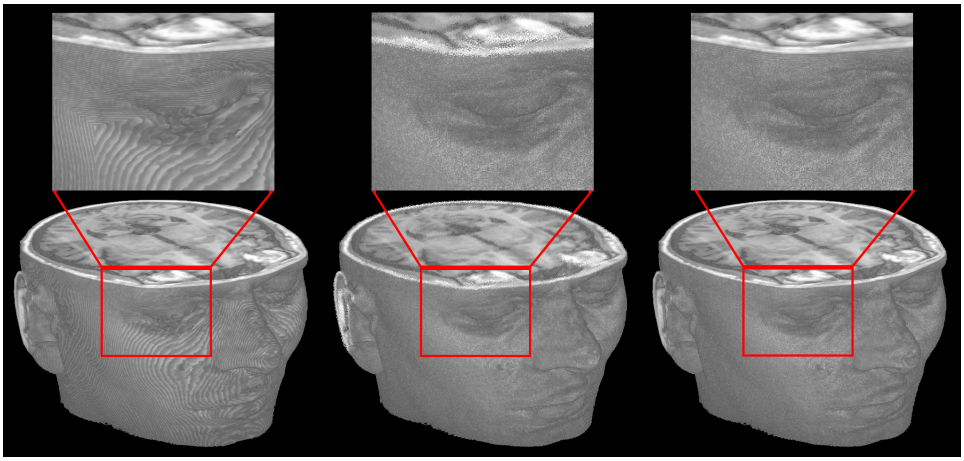

Abbildung 26: Darstellung eines Volumens [2] links ohne und in der Mitte mit Stochastic Jittering. Rechts kommt dazu noch Extent Preserving Jittering zum Einsatz.

Dieses hier benannte *Extent Preserving Jittering* funktioniert auch, wenn durch Clipping nur einen Ausschnitt des Volumendatensatzes mit Hilfe einer verkleinerten Proxy-Geometrie dargestellt wird, wie in Abbildung 26 zu sehen. Mehr zu Clipping in Kapitel 3.14.

3.8 Pre-Integration der Transferfunktion

Bei der post-interpolativen Anwendung der Transferfunktion ist die Qualität der Darstellung direkt von der Abtastrate der Volumendaten abhängig. Die Technik der *Pre-Integration* trennt die Abtastung der Transferfunktion von der der Volumendaten und ermöglicht eine sowohl korrektere als auch potentiell performantere Darstellung, weil die Abtastrate der Volumendaten reduziert werden kann. Der mathematische Hintergrund und eine vollkommen korrekte Berechnung wird in "'Real-Time Volume Graphics" [HKRs+06, Kapitel 9.5] beschrieben, hier wird die Idee dargestellt und im darauffolgenden Abschnitt eine angepasste Umsetzung gezeigt.

Anstatt wie bei der post-interpolativen Anwendung jeweils pro Abtastung des Volumens die Transferfunktion abzufragen, wird sie hier im Vorfeld mit einer numerischen Integration Wert für Wert abgetastet, addiert, normiert und die Ergebnisse in einer zusätzlichen Look-Up-Tabelle abgelegt. Diese Integration der Werte stellt dabei nicht die Transformation eines einzelnen Wertes aus dem Volumen dar, sondern die normierte Zusammenfassung der Transferfunktion, welche zwischen zwei Volumenwerten auftreten kann. Von diesen Werten wird angenommen, dass sie räumlich hintereinander auf dem Kamerastrahl liegen und der Abstand zwischen ihnen gering ist. Für die Kombination aus dem skalaren Werte der vorderen Abtastposition s_f (Front) und dem skalaren Wertes der hinteren Abtastpo-

sition s_b (Back) liegt nun ein integriertes Ergebnis vor, was eine Tabelle mit Ausmaßen gleich dem Quadrat der Auflösung der diskreten Transferfunktion macht. Die räumliche Distanz zwischen beiden Werten wird bei Voraca vernachlässigt, da sie bei kleinen Abständen einen minimalen Einfluss auf das Ergebnis hat. Die Datenmenge er Look-Up-Tabelle kann reduziert werden, wenn man folgende Regel der Integralrechnung einsetzt:

$$\int_a^b f(x)\,\mathrm{d}x = \int_0^b f(x)\,\mathrm{d}x - \int_0^a f(x)\,\mathrm{d}x. \tag{18}$$

Das bedeutet für die Pre-Integration, dass man nicht für alle möglichen Kombinationen aus s_f und s_b einen Eintrag hinterlegen muss. Es sind le-

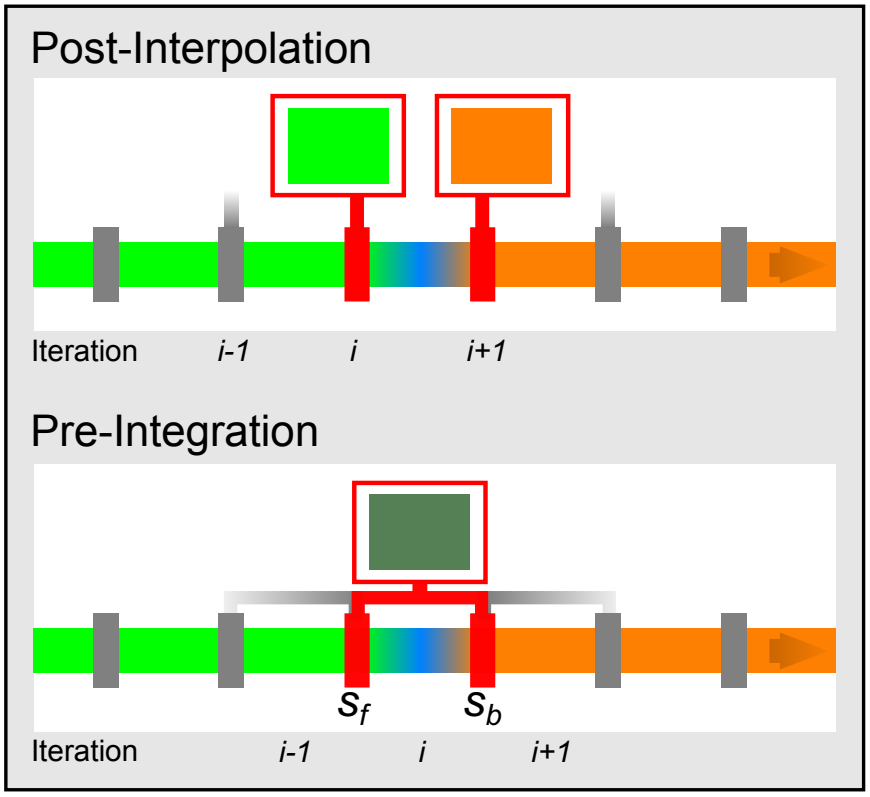

Abbildung 27: Schematische Darstellung des Unterschiedes zwischen Post-Interpolation und Pre-Integration.

diglich die Abfragen zweier vorher integrierter Werte notwendig, in denen alle Werte der Transferfunktion von jeweils 0 bis zu einem s_b aufsummiert wurden, wobei ein solches Integral für alle möglichen s_b vorliegen muss. Bei der Implementierung wird aber deutlich, dass es doch Sinn macht eine größere Datenmenge zu speichern und dafür Berechnungen während der Darstellung einzusparen. Eine weitere Optimierung im Shader ist möglich,

da der aktuelle s_b bei der Front-To-Back das nächste s_f sein wird und damit der skalare Wert von s_b in die nächste Iteration mitgenommen werden kann. Die Selbstverdeckung innerhalb des Intervalls von s_f und s_b wird bei der Berechnung hier vernachlässigt, was bei kleinen Abständen aber keinen großen Einfluss auf die Darstellung hat. Die Technik der Pre-Integration verbessert also die Darstellungsqualität, ist unabhängiger von der Abtastrate des Volumendatensatzes, erzeugt aber mehr Berechnungsaufwand bei einer Änderung der Transferfunktion.

Implementierung

In Voraca wird die Pre-Integration nach einer kleinen Wartezeit bei jeglicher Änderung der Transferfunktion berechnet. Die Berechnungszeit ist auf dem Testsystem nicht bemerkenswert. Die Look-Up-Tabelle für die Pre-Integration Werte wird als eine 2D-Textur an den Shader gegeben.

Die Funktion `updatePreintegrationHelper` der Klasse Transferfunction nimmt als Eingabe einen Anteil der Transferfunktion (`function`) in Form eines `std::vector<glm::vec4>` und einen Handle, dass zu einem OpenGL Texturobjekt zeigt (`GLuint& textureHandle`). In Voraca sind die 12 Werte, ergo Materialeigenschaften, der Transferfunktion in drei `std::vector<glm::vec4>` auf der C++ Seite und in drei RGBA-Texturen aufseiten von OpenGL verteilt. Daher ist eine allgemeine Funktion für die Berechnung der Pre-Integration von Vorteil und lässt ohne Aufwand auch noch mehr Materialeigenschaften zu. Zur Vereinfachung wird im Weiteren nur der Teil der Transferfunktion beachtet, der die Farb- und Alphawerte für die Komposition liefert. Folgender Abschnitt speichert die von 0 aufsummierten Werte für jedes mögliche s_b:

```
1  // Some preparation
2  std::vector<glm::vec4> preintegration(TRANSFERFUNCTION_TEXTURES_RES);
3
4  // First element can be simply read out
5  preintegration[0] = function[0];
6
7  // Now calculate other values
8  for(GLuint i = 1; i < TRANSFERFUNCTION_TEXTURES_RES; i++)
9  {
10   preintegration[i] = preintegration[i-1] + function[i];
11 }
```

Listing 12: Numerische Integration der Transferfunktion.

Nun liegen in dem lokalen `std::vector` mit dem Namen `preintegration` die summierten, bzw. numerisch integrierten, Werte der Transferfunktion vor. Um die Berechnungen im Shader zu minimieren, wird aus diesen Werten eine 2D-Textur erstellt, bei welcher die beiden Koordinaten zum Auslesen des Texels s_f und s_b entsprechen. Man könnte sich diesen Schritt auch sparen, jedoch müsste die Berechnung dann pro Iteration im Shader des

Ray-Casting Algorithmus ausgeführt werden. Da moderne Grafikkarten über üppigen Speicher verfügen, kann man die Werte einmalig vorberechnen und dann als 2D-Textur in den Speicher laden.

Bei der Erstellung wird von der Regel der Integralrechnung aus Gleichung 18 Gebrauch gemacht, um die Werte der Textur aus dem Ergebnis des vorherigen Listing zu berechnen. Des Weiteren wird noch die Normierung der Ergebnisse durchgeführt, indem das Ergebnis durch die Anzahl der eingeflossenen Werte aus der Transferfunktion dividiert wird:

```
1  // Create 2D texture of preintegration
2  std::vector<glm::vec4> textureData(TRANSFERFUNCTION_TEXTURES_RES *
       TRANSFERFUNCTION_TEXTURES_RES);
3  glm::vec4 tmp;
4
5  for(GLuint x = 0; x < TRANSFERFUNCTION_TEXTURES_RES; x++)
6  {
7    for(GLuint y = 0; y < TRANSFERFUNCTION_TEXTURES_RES; y++)
8    {
9      if(x > y)
10     {
11       tmp = preintegration[x] - preintegration[y];
12       tmp *= (1.0f / static_cast<GLfloat>(x-y));
13       textureData[x + TRANSFERFUNCTION_TEXTURES_RES*y] = tmp;
14     }
15     if(x < y)
16     {
17       tmp = preintegration[y] - preintegration[x];
18       tmp *= (1.0f / static_cast<GLfloat>(y-x));
19       textureData[x + TRANSFERFUNCTION_TEXTURES_RES*y] = tmp;
20     }
21     else
22     {
23       textureData[x + TRANSFERFUNCTION_TEXTURES_RES*y] = function[x];
24     }
25   }
26 }
27
28 // Fill texture
29 glBindTexture(GL_TEXTURE_2D, textureHandle);
30 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, TRANSFERFUNCTION_TEXTURES_RES
       , TRANSFERFUNCTION_TEXTURES_RES, 0, GL_RGBA, GL_FLOAT,
       reinterpret_cast<GLfloat*> (&(textureData[0])));
31 glBindTexture(GL_TEXTURE_2D, 0);
```

Listing 13: Vorbereitung der Pre-Integration für den Einsatz im Ray-Casting Shader.

Im korrekten Ansatz der Pre-Integration werden die Farbwerte noch während der Integration mit Alphawerten gewichtet, um Selbstverdeckung zu simulieren. Das hätte im Fragment-Shader jedoch zu Komplikationen geführt, da von ungewichteten Farbwerten ausgegangen wird. Die erzeugte 2D-Textur wird dem Fragment-Shader übergeben, in welchem sie anstatt der post-interpolativen Verwendung der Transferfunktion eingesetzt wird. `prevValue` entspricht dabei s_f und `currValue` ist s_b:

```
#version 330 core
in vec3 position;
in vec3 direction;

out vec4 fragmentColor;

uniform sampler3D uniformVolume;
uniform sampler2D uniformPreintegratedTransferfunction;

const float STEP_SIZE = 0.008;
const float ITERATIONS = 1000;

void main()
{
  // Use input from vertex shader
  vec3 dir = normalize(direction);
  vec3 pos = position;

  // Variables for composition
  vec4 src;
  vec4 dst = vec4(0,0,0,0);
  float prevValue;
  float currValue;

  // Prepare for pre-integration
  float prevValue = texture(uniformVolume, pos).r;
  pos += dir*STEP_SIZE;

  // Do raycasting
  for(int i = 0; i < ITERATIONS; i++)
  {
    // Get value from volume
    currValue = texture(uniformVolume, pos).r;

    // Use values as coordinate in pre-integrated transferfunction
    src = texture(uniformPreintegratedTransferfunction, vec2(prevValue,
        currValue)).rgba;

    // Front-To-Back composition
    dst.rgb += (1.0-dst.a) * src.rgb * src.a;
    dst.a += (1.0-dst.a) * src.a;

    // Prepare for next sample
    pos += dir*STEP_SIZE;
    prevValue = currValue;

    // Check whether still in volume
    if(pos.x > 1 || pos.y > 1 || pos.z > 1 ||
      pos.x < 0 || pos.y < 0 || pos.z < 0)
    {
      break;
    }
  }

  // Output
  fragmentColor = vec4(dst.rgb, 1);
}
```

Listing 14: Kompletter Fragment-Shader mit Implementation einer vereinfachten Variante von Pre-Integration.

Die Ergebnisse dieser Technik sind vor allem bei Transferfunktionen mit hochfrequenten Anteilen denen mit der post-interpolativen Anwendung überlegen. Eine solche Transferfunktion kommt ziemlich oft vor, z.B. bei der Darstellung von Haut. Diese soll meist von der umliegenden Luft ohne Übergang getrennt werden, siehe Abbildung 28 für ein Beispiel.

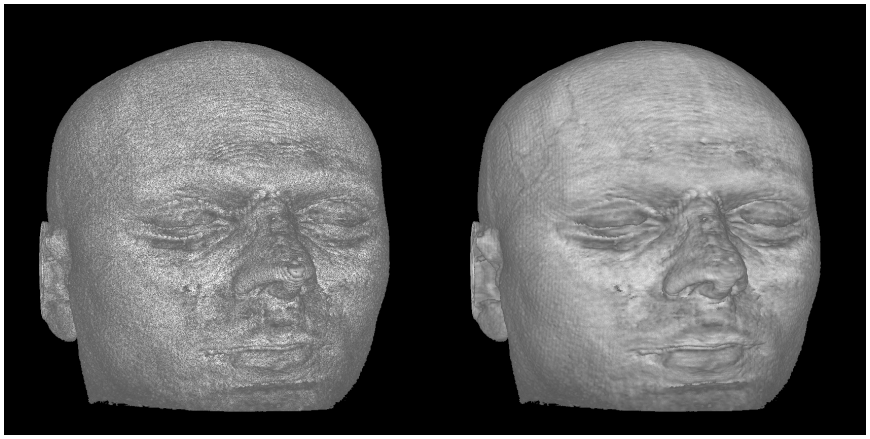

Abbildung 28: Darstellungen des selben Volumendatensatzes [2] und der selben Transferfunktion. Links mit Post-Interpolation und rechts mit Pre-Integration.

3.9 Adaptive Sampling

Eine weit verbreitete Technik zur Optimierung des Ray-Casting Algorithmus ist *adaptives Sampling*. In den Volumendaten gibt es Regionen, in denen die Werte mehr variieren als in anderen Regionen. Wenn man die Abtastung der Transferfunktion außer Acht lässt, macht es Sinn in Regionen mit wenig oder vielleicht gar keiner Varianz der Werte auch weniger Abtastpunkte in die Komposition mit einzubeziehen und dann mehr Präzision in die stark variierenden Regionen zu legen. Da die Transferfunktion nicht beachtet wird, macht adaptives Sampling vor Allem in Kombination mit Pre-Integration Sinn, da damit die Abtastung der Volumendaten und der Transferfunktion unabhängig voneinander stattfinden. Für die Umsetzung des adaptiven Sampling sind einige Vorberechnung notwendig:

Es wird für jeden Volumendatensatz eine weiteres, niedriger aufgelöstes *Importance Volume* erstellt, dass mit jedem Voxel die Abtastrate innerhalb der vom Voxel abgebildeten Region des Ausgangsvolumen beschreibt. Praktisch umsetzen lässt sich dies mit der Berechnung der Varianz der Werte in kubische Regionen des Ausgangsvolumen und dem Speichern der Ergebnisse in dem zusätzlichen Volumen. Beide Volumina stehen dem Algorithmus zur Verfügung und die Abtastweite wird adaptiv pro Abtastpunkt für die Weite bis zum nächsten ausgelesen.

Implementierung

Zuerst soll aufgezeigt werden, wie das Importance Volume erzeugt wird. Danach folgt die Intergration in den Ray-Casting Fragment-Shader und zum Schluss wird die Performance gegenüber einer konstanten Abtastrate bei vergleichbarer Darstellung gemessen.

Generierung des Importance Volumes Wie oben umschrieben soll der vorhandene Datensatz genommen und in kubische Regionen unterteilt werden. Innerhalb dieser Regionen wird die Varianz gemessen und in einem Importance Volume hinterlegt. Die Varianz[13] kann folgendermaßen berechnet werden:

$$\sigma_I^2 = (\frac{1}{|Loc|} * \sum_{p \in Loc} I^2(p)) - \mu_I^2. \tag{19}$$

In Voraca wird direkt nach dem Import bzw. Laden eines Volumens die Generierung des Importance Volume vorgenommen und macht einen Großteil der Wartezeiten bei dieser Aktion aus. Die Umsetzung ist in der Klasse Volume innerhalb der Methode `createImportanceVolume()` zu finden. Die Implementierung ist jedoch nicht vollkommen, da bei einem Ausgangsvolumen mit Ausmaßen, die nicht ohne Rest durch den Down-Sampling-Faktor des Importance Volume teilbar sind, einige Bereiche nicht in die Berechnung mit einbezogen werden. Ansonsten ist die Umsetzung recht einfach aber unhandlich, weshalb sie hier nicht gelistet wird. Es handelt sich um eine dreifache For-Schleife, welche über die Voxel des Importance Volume iteriert. Pro Voxel wird in einer weiteren, inneren und ebenfalls dreifachen For-Schleife über die Region im Ausgangsvolumen iteriert, von welcher die Varianz gemessen und im Voxel des Importance Volume gespeichert werden soll. Am Ende wird der Inhalt des Importance Volume noch anhand der größten auftretenden Varianz normiert.

Integration in Shader Als Grundlage wird der Fragment-Shader aus Kapitel 3.8 genommen, da in diesem Pre-Integration schon umgesetzt worden ist. Als weitere Eingabe ist das Importance Volume in Form eines `sampler3D` und zwei konstante Werte für die minimale und maximale Abtastweite notwendig, zwischen denen der aktuelle Wert aus dem Importance Volume an der jeweiligen Position gewichtet. Das Importance Volume wird, genauso wie das Ausgangsvolumen, linear gefiltert. Dadurch verändert sich sich die Abtastrate nicht abrupt sondern fließend, womit Artefakte vermieden werden.

Eine weitere Bedingung für adaptive Abtastweiten ist die Anpassung der Alphawerte bei jeden Komposition. Dazu kommt die Erkenntnis aus

[13] Die hier gezeigte Formel zur Berechnung der Varianz stammt aus der Vorlesung "Bildverarbeitung 1" an der Universität Koblenz-Landau im Wintersemester 2012/2013.

Kapitel 2.2 zum Einsatz, welche vor der Komposition umgesetzt wird. Sie sorgt für eine adaptive Anpassung der Opazität:

```
1  #version 330 core
2  ...
3  uniform sampler3D uniformImportanceVolume;
4
5  const float MIN_STEP_SIZE = 0.004;
6  const float MX_STEP_SIZE = 0.01;
7
8  void main()
9  { ...
10   // Average step size is needed for alpha correction
11   avgStepSize = (MAX_STEP_SIZE + MIN_STEP_SIZE)/2;
12
13   // Calculate step size
14   float importance = texture(uniformImportanceVolume, pos).r;
15   stepSize = importance * MIN_STEP_SIZE + (1-importance) *
         MAX_STEP_SIZE;
16
17   // Prepare for pre-integration
18   float prevValue = texture(uniformVolume, pos).r;
19   pos += dir*stepSize;
20
21   // Do raycasting
22   for(int i = 0; i < ITERATIONS; i++)
23   { ...
24     // Adjust alpha value of source
25     src.a = 1-pow(1-src.a, stepSize/avgStepSize);
26
27     // Front-To-Back composition
28     dst.rgb += (1.0-dst.a) * src.rgb * src.a;
29     dst.a += (1.0-dst.a) * src.a;
30
31     // Calculate new step size for expansion
32     importance = texture(uniformImportanceVolume, pos).r;
33     stepSize = importance * MIN_STEP_SIZE + (1-importance) *
           MAX_STEP_SIZE;
34     ...
35 }
```

Listing 15: Erweiterung des Fragment-Shaders aus Listing 14 um adaptives Sampling.

Damit entsteht pro Iteration ein weiterer Texture-Look-Up, welcher mehr Aufwand bedeutet. Auch die Berechnung der adaptiven Abtastweite in Zeile 33 hat aufgrund der vielen Iterationen einen Einfluss auf die Performance. Das Importance Volume kann in Voraca angezeigt werden, die skalaren Varianzwerte werden mit dem in Kapitel 2.2 vorgestellten Verfahren der Maximum Intensity Projection dargestellt. Siehe hierfür Abbildung 29.

Performance Ein Vergleich der Performance von adaptivem Sampling gegenüber dem konstanten Sampling ist nicht auf alle Situationen anwendbar. In vielen Fällen ist der Gewinn von Performance durch adaptives Samp-

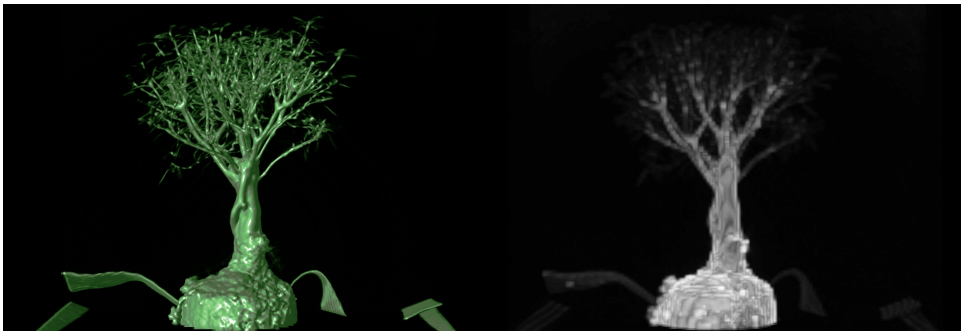

Abbildung 29: Darstellung eines Importance Volumes auf der rechten Seite, links das Ausgangsvolumen [7].

ling zumindest in Voraca nicht bemerkbar oder es wird für eine qualitativ gleichwertige Darstellung sogar eine längere Berechnungszeit pro Frame benötigt. Eine mögliche Situation ist dennoch in Abbildung 30 darge-

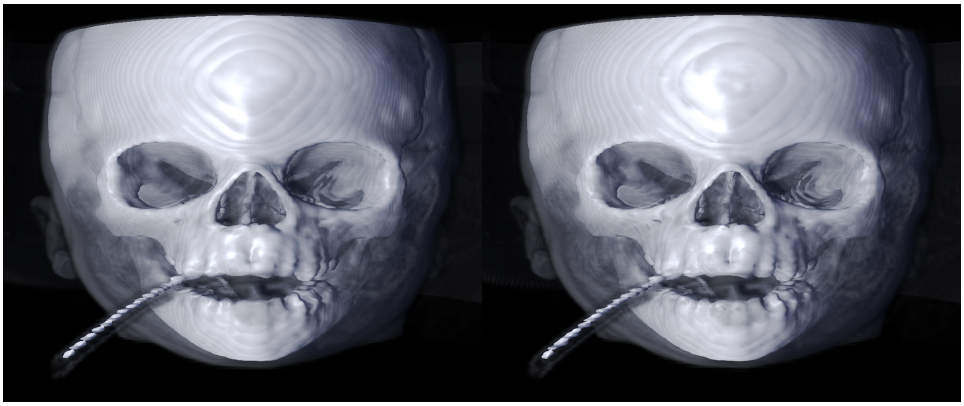

Abbildung 30: Darstellung eines Schädels. Links mit einer konstanten Abtastweite, rechts mit adaptiver Abtastweite. Diese kann sich anhand der Variation der Volumendaten [1] in ihrer Schrittweite anpassen.

stellt. Es handelt sich um den CT-Scan des Schädel eines Säuglings, wobei die Transferfunktion das Gewebe halbtransparent und den Knochen undurchsichtig abbildet. Die konstante Abtastweite beträgt 0.008, die minimale Weite der adaptiven 0.002 und die maximale 0.011 Einheiten im Model-Space. Die visuellen Ergebnisse sind in ihrer Qualität miteinander vergleichbar, wie in der Abbildung 30 zu sehen ist. Bei den Messungen sind andere Optimierungen wie Empty Space Skipping oder Early Ray Termination deaktiviert worden:

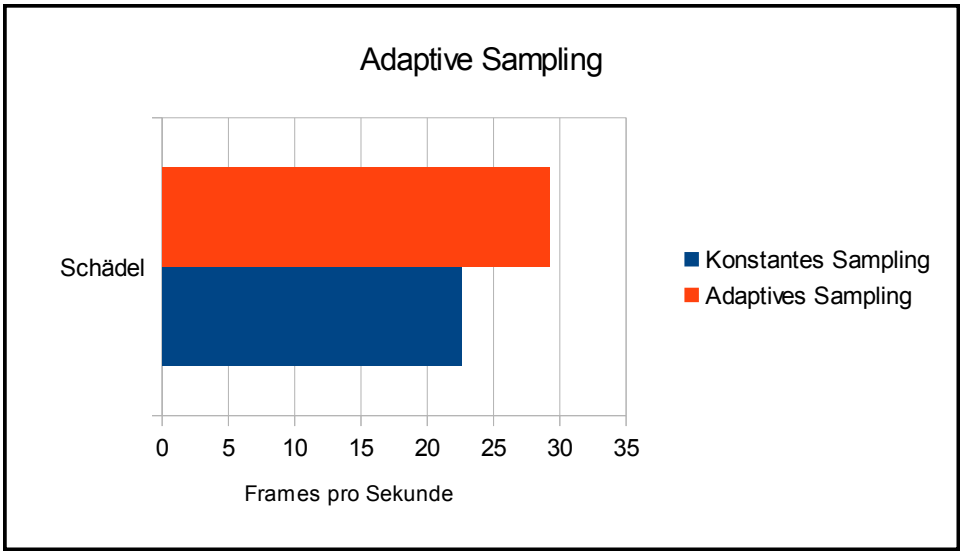

Abbildung 31: Messung der Performance von Adaptive Sampling am Testaufbau aus Abbildung 30.

Der Zugewinn von sechs Frames pro Sekunde in der Messung aus Abbildung 31 sieht bei diesen niedrigen Raten nach einem großen Gewinn aus, ist aber direkt von den eingestellten Abtastraten abhängig und es wird keine zuverlässige Anhebung der Performance gewährleistet. Wenn die Werte geschickt eingestellt werden, können konstante Abtastraten schneller in der Berechnung sein als adaptive, und das bei visuell vergleichbaren Ergebnissen. Eventuell hängt es mit einer unzureichenden Implementierung zusammen, da das eigentliche Konzept sinnig ist und theoretisch mehr Performance bei gleicher Darstellungsqualität liefern könnte. Es kann aber auch sein, dass der zusätzliche Aufwand pro Iteration den möglichen Gewinn stark minimiert.

3.10 Voxel Spaced Sampling

Bei *Voxel Spaced Sampling* handelt es sich um eine kleine Erweiterung, welche im Rahmen dieser Arbeit entstanden ist. Wahrscheinlich ist sie aber unter anderem Namen irgendwo in der Literatur zu finden. Es handelt sich nicht um eine umfangreiche Optimierung, sondern eine Variation in der Angabe der Abtastweite. In dieser Variante ist die Abtastweite abhängig von den Dimensionen des Volumens, das dargestellt werden soll. Siehe Abbildung 32 für eine schematische Darstellung. Anstatt die absolute Weite in Model-Space über die Benutzeroberfläche zu steuern, gibt man nun einen Multiplikator an. Mit dem Multiplikator wird die Abtastweite, welche im Voxel-Space vorliegt, skaliert.

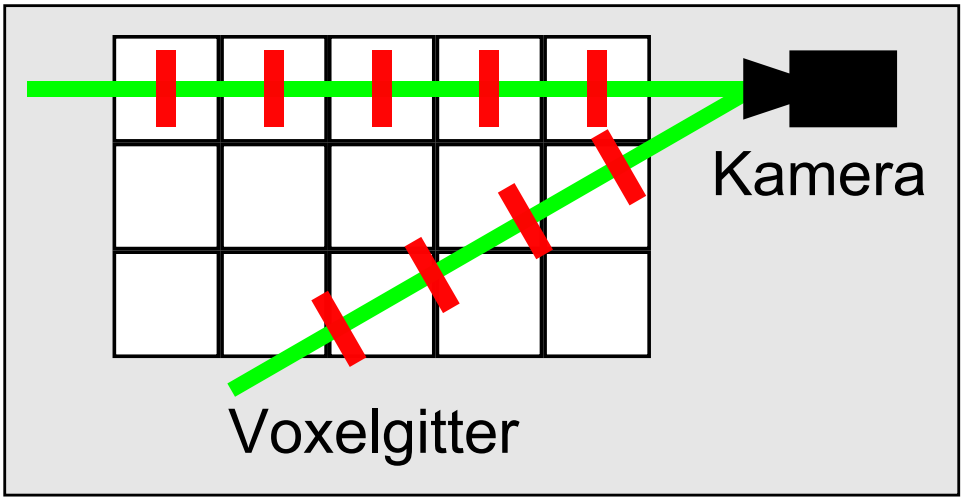

Abbildung 32: Schematische Darstellung von Voxel Spaced Sampling.

Implementierung

Eine Implementation kann durch eine zusätzliche Funktion im Shader umgesetzt werden. Diese Funktion multipliziert die normierten Richtung des Kamerastrahles mit den Model-Space Ausmaßen eines der uniformen Voxel. Als globale Variablen wird mit `vec3 uniformVolumeResolution` die Auflösung des Volumendatensatzes benötigt und als Eingabe nimmt die Funktion die normierte Richtung `vec3 dir` des Kamerastrahles. Diese Funktion wird in jeder Instanz des Fragment-Shaders, sprich für jeden zu berechnenden Kamerastrahl, einmal aufgerufen:

```
1  float voxelSpacedStepSize(vec3 dir)
2  {
3    vec3 voxelSize = 1.0/uniformVolumeResolution;
4    vec3 result = voxelSize * dir;
5    return length(result);
6  }
```

Listing 16: Funktion zur Berechnung der Abtastweite in Voxel-Space.

Das Ergebnis wird mit einem Multiplikator gewichtet und als Abtastweite während den Iteration benutzt. Die Performance wird nicht beeinflusst, bei vergleichbarer optischer Qualität ist die Framerate identisch. Die Einstellung der Abtastweite ist aber intuitiver, da beim Multiplikator mit Werten zwischen 0.1 und 2 gearbeitet wird. Des Weiteren ist die Abtastweite nun von den Volumendaten abhängig, wodurch meist auf Anhieb eine für das Volumen angemessene Weite benutzt wird.

3.11 Downsampling

Die einfachste Technik um die Berechnung eines Frames zu beschleunigen, ist die Anzahl der zu berechnenden Bildpunkte, und damit Aufrufe des Fragment-Shader, zu reduzieren. Das Bild wird mit weniger Bildpunkten berechnet, z.B. statt in 1080p nur in 720p, und dann zur gewünschte Darstellungsgröße auf dem Bildschirm hochskaliert. Dabei kann von Filtern Gebrauch gemacht werden, um unter anderem die Schärfe auf dem skalieren Bild nachträglich zu stärken.

Implementierung

Bei einer Implementation in Voracas Viewport-System könnte von Framebuffer-Objekten Gebrauch gemacht werden. Das Bild des Renderer würde in einen eigenen Framebuffer geschrieben, der dann als Eingabetextur eines Viewport-Filling-Quad dient. Damit könnte man noch weiteres Post-Processing betreiben, wie z.B. Tone-Mapping. Die Umsetzung der Transferfunktion in Voraca wäre für High Dynamic Range Rendering dank 32-Bit pro Kanal schon bereit, es müssten jedoch einige Anpassungen und Erweiterungen in der Renderer Klasse geschrieben werden.

3.12 Lokale Beleuchtung

Lokale Beleuchtung ist bisher in fast allen Abbildungen benutzt, aber noch nicht als Thema behandelt worden. Beleuchtung gibt einer zweidimensionalen Abbildung zusätzliche Informationen über die Tiefe und Form des abgebildeten Objektes. Besonders Glanzpunkte können die Beschaffenheit der Oberfläche eines Objektes gut wiedergeben. In diesem Abschnitt wird nun die lokale Beleuchtung der Volumendaten mit Hilfe von Gradienten aufgezeigt und implementiert.

Der Zusatz 'lokal' bedeutet im Falle der Beleuchtung, dass die Beleuchtung unabhängig von der Umgebung berechnet wird. Es wird nicht die Beleuchtung von Stellen um den aktuellen Abtastpunkt herum abgefragt, weshalb auch keine indirekte Beleuchtung durch abgelenkte Lichtstrahlen oder durch emittierende Regionen im Volumen möglich ist. Einzig die lokale Information des Abtastpunktes und die Informationen über die globale Lichtquelle werden für die Berechnung benutzt, wodurch sich der Aufwand in Grenzen hält. Dies ist auch notwendig, da diese Berechnung pro Abtastpunkt auf jedem Stahl vorgenommen werden muss.

Beleuchtungsmodell Es wird die physikalisch nicht korrekte Annäherung durch das Phong-Beleuchtungsmodell [Pho75] gewählt, da sie schnell zu berechnen ist, visuell überzeugt und die erforderlichen Eingaben ohne zu großen Aufwand bereitgestellt werden können. Das Modell kann durch

drei Terme beschrieben werden, welche an jedem Abtastpunkt gewisse Materialeigenschaften benötigen. Verkürzt wird die Lichtquelle als L und die Materialeigenschaften als M bezeichnet:

$$I = I_{ambient} + I_{diffus} + I_{spekular} \tag{20}$$

$$I_{ambient} = L_{ambient} * M_{ambient} \tag{21}$$

$$I_{diffuse} = L_{diffus} * M_{diffus} * (\vec{L} \cdot \vec{N}) \tag{22}$$

$$I_{spekular} = L_{spekular} * M_{spekular} * (\vec{R} \cdot \vec{V})^{M_{exponent}} \tag{23}$$

Im Term für $I_{spekular}$, welcher den Glanz berechnet, beschreibt $\vec{V}$ die Blickrichtung und $\vec{R}$ steht für den anhand der Normalen $\vec{N}$ der Volumendaten am Abtastpunkt reflektierten Lichtstahles $\vec{L}$. Die Paare L_{diffus} und $L_{spekular}$ bzw. M_{diffus} und $M_{spekular}$ werden meist als gleich angenommen, um weniger Einstellungen vornehmen zu müssen. Die Implementation in Voraca wird in einem späteren Abschnitt aufgezeigt. Zuerst werden noch einige Werte benötigt, die nicht durch die Transferfunktion abgedeckt sind. Diese liefert mit den Materialeigenschaften nur die Werte für die M Variablen. Die Werte von L werden in einer Szene bzw. dem Renderer festgelegt und sind für alle Abtastungen gleich, sofern von einem direktionalen Licht ausgegangen wird. Des Weiteren wird von dem Licht die Richtung $\vec{L}$ benötigt, welche bei einen direktionalen Licht ebenfalls in der kompletten Szene gleich ist. Die Blickrichtung $\vec{V}$ kann berechnet werden, wenn die Position der Kamera bekannt ist.

Zentraldifferenzen Der Reflektionsvektor $\vec{R}$ des Lichtes ist direkt von der Normalen $\vec{N}$ an der abgetasteten Stelle abhängig. Da die Normale auch bei dem Skalarprodukt für die Berechnung der diffusen Beleuchtung notwendig ist, ist sie unbedingt erforderlich. Jedoch sind die Volumendaten nur ein Feld aus skalaren Werten, es gibt keine Oberflächen die eine echte Normale haben könnten. Stattdessen benutzt man den Gradienten, welcher an den jeweiligen Positionen aus den Unterschieden der Volumenwerte ermittelt wird. Eine Variante ist die Berechnung mit Hilfe der *Zentraldifferenzen* $\nabla f(x, y, z)$, welche in "Real-Time Volume Graphics" [HKRs+06, Kapitel 5.3] vorgestellt werden. Dabei werden auf jeder der drei Raumachsen des Model-Space rechts und links des Abtastpunktes zwei Proben der Volumenwerte genommen und die Differenz gebildet, siehe folgende Gleichung:

$$\nabla f(x, y, z) \approx \frac{1}{2h} * \begin{pmatrix} f(x+h, y, z) - f(x-h, y, z) \\ f(x, y+h, z) - f(x, y-h, z) \\ f(x, y, z+h) - f(x, y, z-h) \end{pmatrix}. \tag{24}$$

Der Abstand der Proben vom Abtastpunkt wird dort mit h beschrieben. Die drei Differenzen bilden normiert die drei Dimensionen einer Normalen

der Volumendaten an diesem Abtastpunkt. Problematisch sind homogene Regionen, bei welchen keine Varianz in den Volumendaten vorliegt. Dieses Problem wird im Folgenden ignoriert, es kann aber anhand der Stärke der Magnitude eines Gradienten eine Gegenmaßnahme wie die Abschwächung der diffusen und spekularen Anteile des Lichtes an dieser Stelle bewirkt werden.

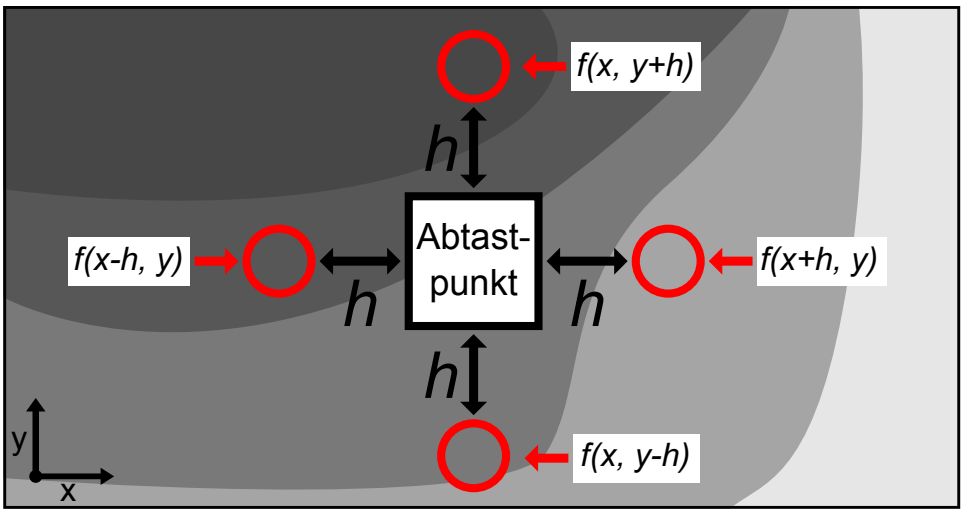

Abbildung 33: Zentraldifferenzen im 2D-Raum.

Die Gradienten können entweder im Vorfeld berechnet und in einem zusätzlichen Volumen mit drei Werten für die x-, y- und z-Komponente der Normale abgelegt oder bei jeder Abtastung aufs neue kalkuliert werden. Die erste Methode hat den Nachteil, dass zum einen ein weiterer, großer Datensatz erzeugt wird, welcher potentiell dreimal so viel Speicher benötigt wie das Ausgangsvolumen. Zum anderen tritt ein ähnliches Problem wie bei der pre-interpolativen Anwendung der Transferfunktion auf. Ausschließlich jene Normalen stehen zur Verfügung, welche vorher berechnet worden sind. Wenn an einer Zwischenstelle im Volumen eine Normale benötigt wird, werden vorhandene Normalen linear interpoliert und zu einer neuen gewichtet. Es ist aber korrekter, die Normale an jeder Position aus den vorhandenen Volumendaten zu berechnen. Dies ist nur bei einer *On-The-Fly* Umsetzung möglich. Außerdem ermöglicht die Kalkulation pro Abtastung auch den Einbezug der Transferfunktion, sodass nicht die rohen Volumendaten als Dichten interpretiert werden, sondern die Opazität der transformierten Daten. All dies erzeugt zwar anstatt einer immensen Speicherauslastung einen beachtlichen Rechenaufwand pro Iteration, ist auf modernen Grafikkarten aber interaktiv umsetzbar und hat Vorteile gegenüber einer vorberechnete Normalen bzw. Gradienten. Anhand der nun verfügbaren Normalen $\vec{N}$ und der Richtung des Lichts $\vec{L}$ kann die direkte

Beleuchtung und der reflektierte Lichtstrahl $\vec{R}$ ausgerechnet und für den spekularen Term verwendet werden.

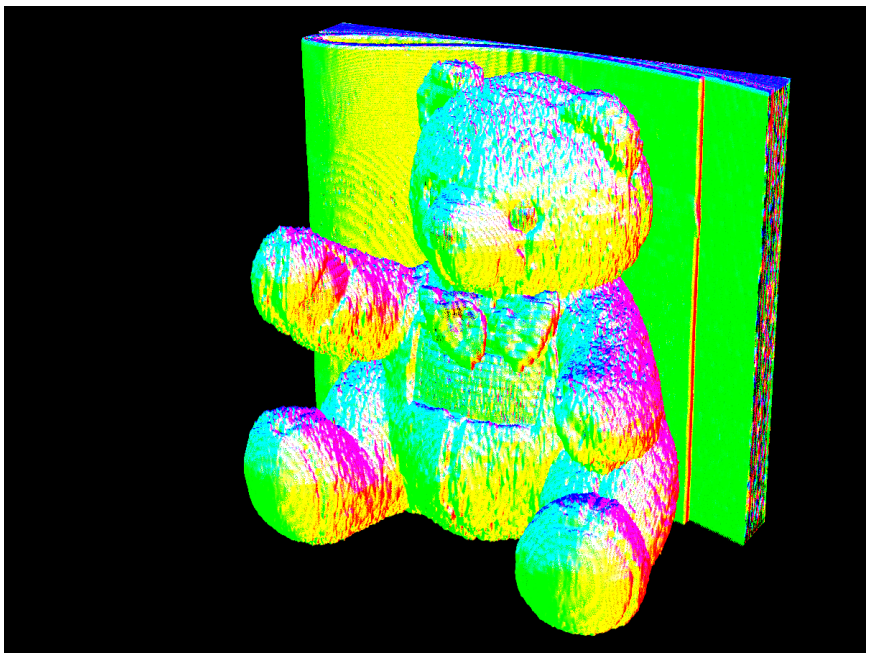

Abbildung 34: Gradienten der sichtbaren Anteile eines Volumens [8], hier kodiert mit RGB-Farben. Es ist zu beachten, dass die Normalen nicht in World-Space vorliegen.

Implementierung

Zuerst wird die Berechnung der Normalen im Fragment-Shader angegeben, dann die Beleuchtung selbst und zum Schluss eine Messung der Performance ausgeführt.

Berechnung der Normalen im Shader Bevor an der abgetasteten Position im Volumen beleuchtet werden kann, muss die Normale aus den vorhandenen Daten generiert werden. Dazu benutzt der Shader in Voraca eine ausgelagerte Funktion. In dieser wird mit der eben vorgestellten Zentraldifferenz der Gradient berechnet, aus welchem dann mithilfe einer Normalisierung die Normale an der Stelle extrahiert wird. Die Länge des Gradienten wird weiterhin noch in die vierte Komponente des Ausgabevektors geschrieben, da man diese Magnitude anderweitig benutzen kann. Als Eingaben dient nur die Abtastposition `pos` und der Abstand für die Zentraldifferenzen `offset`, der in der Gleichung 24 mit h beschrieben worden ist:

```
/** Returns vec4(normalized normal, magnitude) in model-space */
vec4 normalOnRawData(vec3 pos, float offset)
{
  vec3 nrm = vec3(0,0,0);
  float x1, x2, y1, y2, z1, z2;

  // Get values from volume
  x1 = texture(uniformVolume, vec3(pos.x + offset, pos.y, pos.z)).r;
  x2 = texture(uniformVolume, vec3(pos.x - offset, pos.y, pos.z)).r;
  y1 = texture(uniformVolume, vec3(pos.x, pos.y + offset, pos.z)).r;
  y2 = texture(uniformVolume, vec3(pos.x, pos.y - offset, pos.z)).r;
  z1 = texture(uniformVolume, vec3(pos.x, pos.y, pos.z + offset)).r;
  z2 = texture(uniformVolume, vec3(pos.x, pos.y, pos.z - offset)).r;

  // Gradient
  nrm.x = x1 - x2;
  nrm.y = y1 - y2;
  nrm.z = z1 - z2;

  // Use length as magnitude, divided through maximal length of sqrt(3)
  float mag = length(nrm) / 1.7320508;

  // Transformate with uniformModelScale
  nrm = (uniformModelScale * vec4(nrm,0)).rgb;

  nrm = normalize(nrm);
  return vec4(nrm, mag);
}
```

Listing 17: Berechnung der Normalen im Fragment-Shader.

Die Anfrage nach der Normalen bedeutet sechs Texture-Look-Ups und damit einen hohen Aufwand pro Iteration in der For-Schleife des Fragment-Shaders. Da die Normale aber auch für andere Berechnungen, wie z.B. den Fresnelterm, benötigt wird, ist es ein lohnendes Unterfangen.

Beleuchtung im Shader Da nun die Normale bereitsteht, kann in einer weiteren Methode die Beleuchtung stattfinden. Dazu sind einige Eingabeparameter zu füllen, welche teilweise aus der Transferfunktion entnommen werden können. Sowohl die Farbe `col` als auch die Beleuchtungseigenschaften `mat` aus der Rückgabe der Transferfunktion an der abgetasteten Stelle fließen in die Berechnung mit ein. Unter dem Begriff der Materialeigenschaft werden in dieser Arbeit alle Werte der Transferfunktion beschrieben. In dieser Funktion ist `mat` aber nur die Sammlung der Werte, welche exklusiv für die Beleuchtung benutzt werden. Genauer gesagt ist im Rotkanal der ambiente Multiplikator, im Grünkanal der spekulare Multiplikator, im Blaukanal die spekulare Sättigung und im Alphakanal der spekulare Exponent. Siehe jeweils Kapitel 3.4 für eine Beschreibung. Natürlich wird auch die Normale mit `nrm` übergeben, genau wie die Richtung des Kamerastrahles `dir` als Blickrichtung. Die Farbe `sunColor` bzw. `ambientColor` und die Richtung `sunDirection` des Lichts liegen als uniforme Werte vor und werden nicht explizit mitgegeben:

```
1  /** Returns shaded color */
2  vec3 calcLighting(vec3 col, vec3 nrm, vec4 mat, vec3 dir)
3  {
4    // Lighting (Phong)
5    vec3 light = mat.x * ambientColor + sunColor.rgb * max(dot(
         sunDirection, nrm),0) * sunColor.a;
6    float spec = pow(max(0.0, dot(reflect(-sunDirection, nrm), dir)), mat.
         w) * brightness;
7
8    // Combine lighting and color
9    vec3 result = col * light;
10
11   // Add specular lighting
12   result += ((col * mat.z + (1-mat.z) * (col.r+col.g+col.b)/3 ) * mat.y
         * spec);
13
14   // Some clamping
15   result = min(result, 1);
16
17   return result;
18 }
```

Listing 18: Funktion zur Beleuchtung an einem Abtastpunkt.

Als Beleuchtungsmodell wird wie erwähnt Phong umgesetzt. Interessant ist Zeile 12, da dort die spekulare Sättigung berechnet wird. Es handelt sich um eine Variation des Modells, die mit weniger Materialwerten auskommt und trotzdem sowohl ungesättigte Glanzpunkte als auch Glanzpunkte in der Farbe der transformierten Abtaststelle zulässt. Der von der Funktion

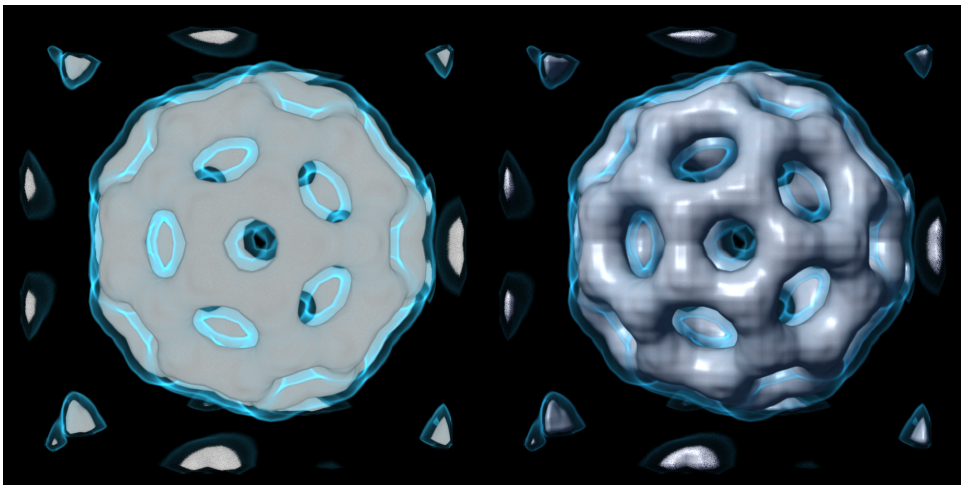

Abbildung 35: Darstellung eines Volumens [9] einmal links ohne und rechts mit lokaler Beleuchtung durch eine direktionale Lichtquelle.

zurückgegebene Wert wird anstatt dem Farbwert aus der Transferfunktion für die Komposition verwendet. Es kann auf den Rückgabewert vor der Komposition noch Farbe als emittierter Anteil aufaddiert werden. Wie

Abbildung 35 zeigt, profitiert vor Allem die Tiefenwahrnehmung von der hier aufgezeigten Beleuchtung. Dank der Glanzpunkte kann auch bei einer gleichfarbigen Fläche die Oberflächenkrümmung visualisiert werden.

Performance Die Performance wird deutlich von der lokalen Beleuchtung beeinflusst, wie die Messung in Abbildung 36 zeigt. Der Messung liegt die Szene aus Abbildung 35 zugrunde. Für die Messung wurde sowohl Empty Space Skipping als auch Early Ray Termination aktiviert:

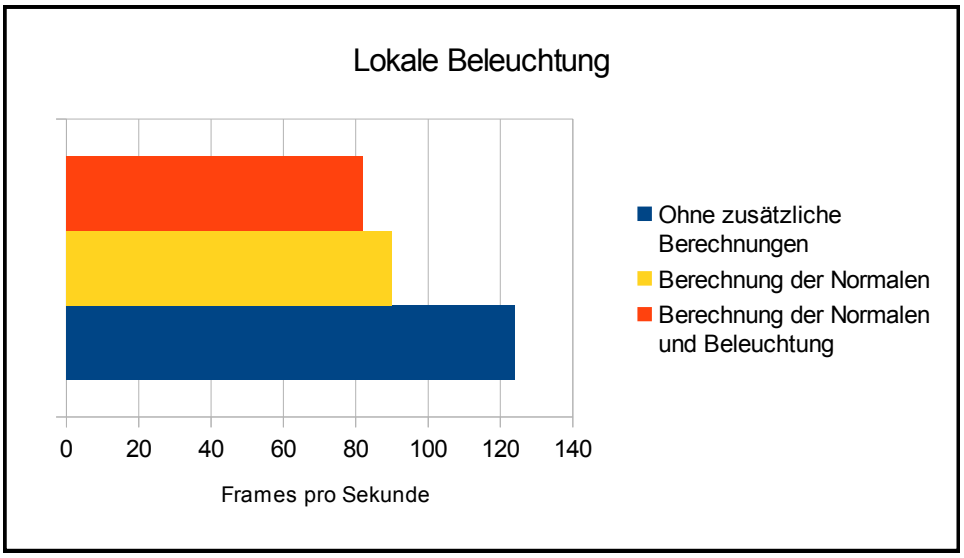

Abbildung 36: Messung der Performance bei lokaler Beleuchtung.

Damit wird auch klar, dass zumindest eine primitive Implementierung von globaler Beleuchtung nicht interaktiv darstellbar ist. Es könnte z.B. die globale Beleuchtung Frame für Frame bis zu einer Änderung der Lichtquelle oder Transferfunktion iterativ erweitert und einem zusätzlichen Volumen zwischengespeichert werden.

Erweiterungen dieser Technik sind auf vielfältige Weise möglich. In Voraca sind aufseiten der Normalen zwei Verbesserungen umgesetzt worden. *Normals On Classified Data* bedeutet, dass der Gradient nicht auf den Rohdaten des Volumens gebildet wird, sondern die Daten mit der Transferfunktion transformiert und dann auf den resultierenden Alphawerten die Zentraldifferenzen gebildet werden. Eine weitere Verbesserung sind die *Extent Aware Normals*, bei welcher berücksichtigt wird, ob bei der Bildung der Zentraldifferenzen die Abfrage einen Wert außerhalb des Volumens trifft. Von diesem wird dann der Wert Null angenommen und so glatte Flächen an Clipping-Kanten ermöglicht. Siehe dazu Abbildung 37.

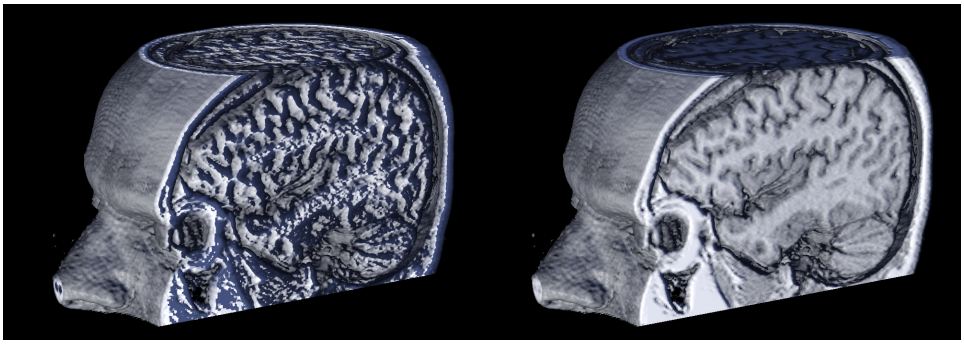

Abbildung 37: Links Volume Clipping ohne Extent Aware Normals, rechts eine Darstellung des Volumens [2] mit dieser Verbesserung.

3.13 Direkte Schatten

Eine Erweiterung der lokalen Beleuchtung bilden *direkte Schatten*, bei welchen pro Abtastpunkt ein Schattenfühler in Richtung der Lichtquelle ausgesandt wird. Dieser tastet ähnlich dem Kamerastrahl das Volumen ab und

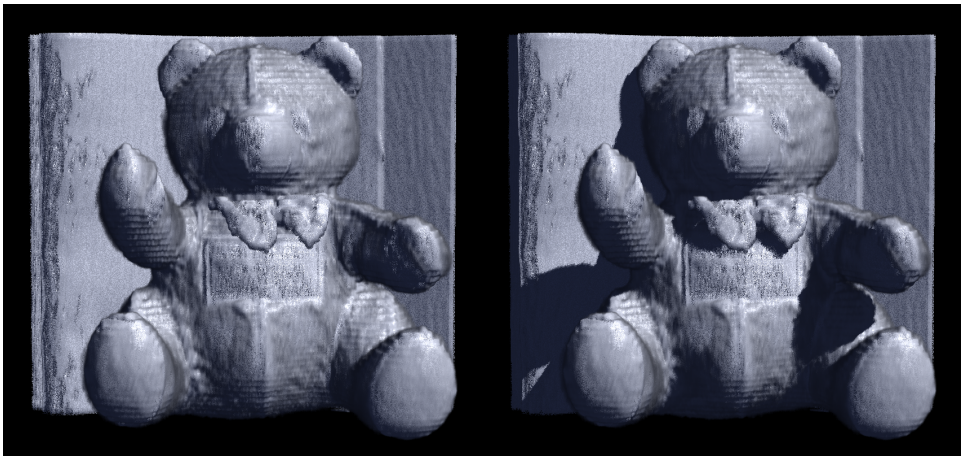

Abbildung 38: Links ist für die Darstellung des Volumens [8] nur lokale Beleuchtung aktiviert, rechts werden direkte Schatten berechnet.

sammelt die Alphawerte transformierter Volumendaten ein. Auch hier kann Early Ray Termination eingesetzt werden, um bei einer vollen Beschattung durch undurchsichtige Regionen im Volumen die Abtastung vorzeitig zu beenden. Die Beschattung wird daraufhin in der Berechnung der beleuchteten Farbe aus Listing 18 eingebracht, sodass an einer beschatteten Stelle ausschließliche ambientes Licht Einfluss hat. Die Berechnung der Schatten ist extrem aufwendig, da pro Abtastpunkt ein Schattenfühler ausgesendet wird und dieser selbst eine Abtastung der Volumendaten auf einem Strahl darstellt.

Implementierung

Die Implementierung in Voraca ist weder vollständig noch vollkommen korrekt. Es sind nur wenige Optimierungen wie Early Ray Termination und die Unterstützung von Pre-Integration umgesetzt worden. Daher wird auch kein Listing dazu aufgeführt. Das Ergebnis in Abbildung 38 ist zwar überzeugend, aber extrem aufwändig. Dies zeigt folgende Messung der Performance der abgebildeten Szene mit den verfügbaren Optimierungen:

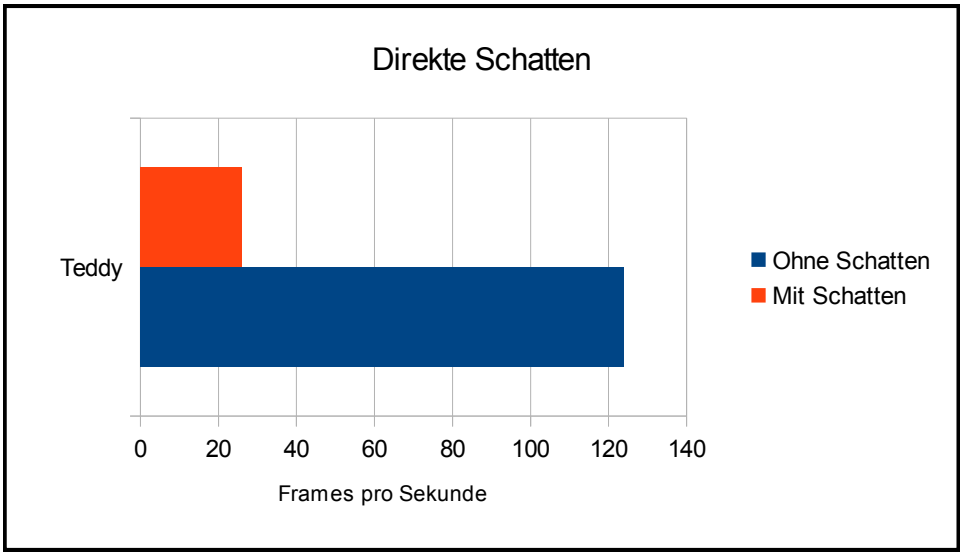

Abbildung 39: Messung der Performance bei Beleuchtung ohne und mit direkten Schatten.

Bei einigen Volumina und vor Allem bei Transferfunktion mit halbtransparenten Anteilen stürzt der Grafikkartentreiber bei Aktivierung der Schatten ab, weil der Berechnungsaufwand zu groß wird. Man muss sich vorstellen, dass hundert Abtastpunkte auch hundert Schattenfühler bedeuten, welche wiederum einige Abtastungen vornehmen. Auch wenn die Abtastungen des Schattenfühlers weniger Aufwand bedeuten als die Abtastungen durch den Kamerastrahl, entstehen sehr viele Kalkulationen und Texture-Look-Ups.

3.14 Volume Clipping

Um Einblicke in das Innere der Volumen zu gewähren, können unterschiedliche Aufwände betrieben werden. Eine Möglichkeit ist es, die Proxy-Geometrie zu beschneiden oder zu skalieren. Dieses *Volume Clipping* kann im einfachsten Fall über eine achsenparalelle Skalierung des Würfels erreicht werden, auf den der Ray-Casting Shader angewendet wird. Weitergehende Tech-

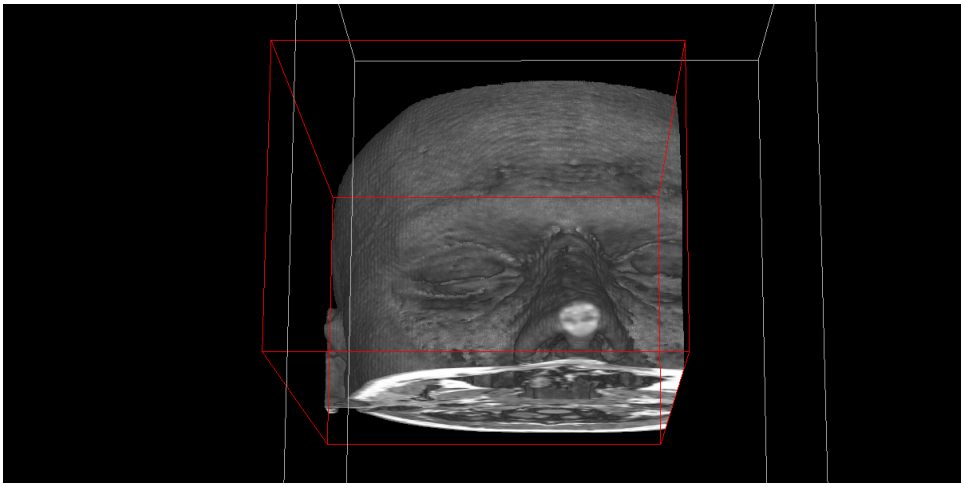

Abbildung 40: Visualierung von Clipping eines Volumens [2] in Voraca.

niken verändern die Proxy-Geometrie in ihrer Struktur und können unter anderem von der Near-Plane der Kamera beeinflusst werden, sodass man mit der Kamera in das Volumen fliegen und es von innen betrachten kann.

Implementierung

Eine Veränderung der Proxy-Geometrie hätte einen erheblichen Aufwand in der Implementation verursacht und würde zu Seiteneffekte mit den bereits vorgestellten Techniken führen. So müsste Extent Preserving Jittering aus Kapitel 3.7 und Extent Aware Normals aus Kapitel 3.12 angepasst werden und könnten danach mehr Berechnungsaufwand verursachen. Deswegen ist nur achsenparalleles Clipping umgesetzt worden, welches mit der AntTweakBar des Renderes gesteuert werden kann. Im Endeffekt werden dem Vertex-Shader eine `uniform mat4 uniformVolumeExtent` für die Skalierung und ein `uniform vec3 uniformVolumeExtentOffset` für die Verschiebung des Clipping-Quaders übergeben. Achsenparalleles Clipping ist nichts anderes als den Ausgangswürfel mit einem Quader zur schneiden und die Schnittmenge darzustellen. Siehe Abbildung 40 für die Visualisierung in Voraca. Nun folgt ein Ausschnitt aus dem Vertex-Shader, der die Umsetzung aufzeigt. Es ist zu beachten, dass die Proxy-Geometrie in Voraca ihren Schwerpunkt nicht wie bisher angenommen bei (0.5,0.5,0.5) hat, sondern im Ursprung des Koordinatensystemes. Daher ist eine Verschiebung notwendig:

```
1  // Calculate vertex position afer volume clipping
2  vec4 vertexPosition = (uniformVolumeExtent * positionAttribute) + vec4(
       uniformVolumeExtentOffset,0);
3  vertexPosition =  clamp(vertexPosition, vec4(-0.5f,-0.5f,-0.5f,0), vec4
       (0.5f,0.5f,0.5f,1));
4
5  // Output for rasterization
6  gl_Position = uniformProjection * uniformView * uniformModelRotation *
       uniformModelScale * vertexPosition;
7
8  // Start position for rays inclusive same clamping as above
9  position = clamp((uniformVolumeExtent * positionAttribute).xyz +
       uniformVolumeExtentOffset, vec3(-0.5f,-0.5f,-0.5f), vec3(0.5f,0.5f
       ,0.5f));
```

Listing 19: Vertex-Shader mit Volume Clipping.

Es handelt sich also nur um eine Skalierung und Verschiebung der Proxy-Geometrie. Zusätzlich muss noch das Abbruchkriterium aus dem Listing 7 erweitert werden. Dieses lässt einen Abbruch zu, sobald der abtastende Strahl außerhalb des Volumens gelangt. In Voraca ist dies nicht direkt umgesetzt, da dieses Kriterium so nicht existiert. Stattdessen wird initial die Länge des Strahles durch das gesamte bzw. geclippte Volumen berechnet und bei jeder Abtastung anhand dieses Wertes geprüft, ob er sich noch im Volumen befindet.

4 Ergebnisse

Die Implementation nahezu aller vorgestellten Techniken ist in Voraca umgesetzt worden und steht dem Nutzer bereit zum Ausprobieren. Man kann Volumina der Formate PVM und DAT importieren, eine eindimensionale Transferfunktion auf Basis einer Bezierkurve definieren und im Renderer lokale Beleuchtung und Volume Clipping aktivieren. Abbildung 41 zeigt einen typischen Screenshot aus Voraca. Der Volumendatensatz 'baby.pvm' aus der Volume Library von Stefan Röttger ist importiert und eine passende Transferfunktion erstellt worden. Die Anwendung der Transferfunktion

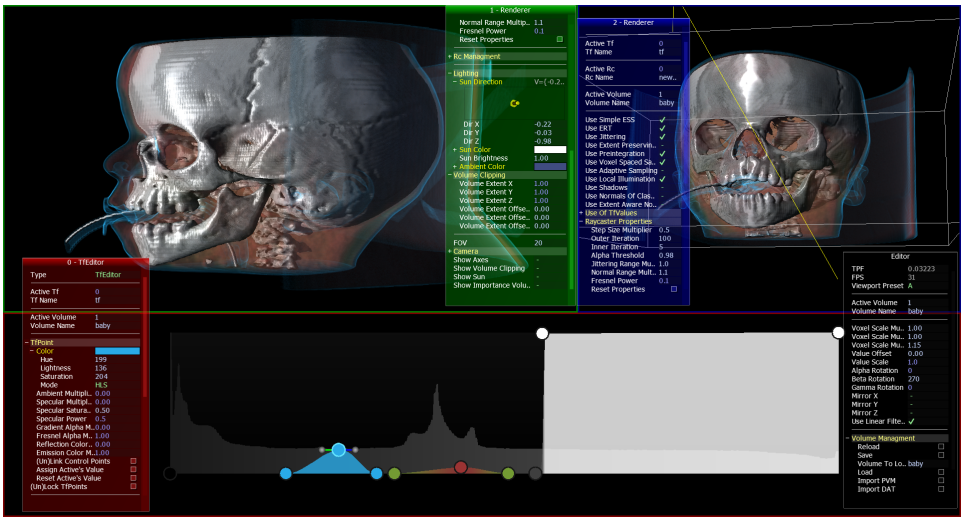

Abbildung 41: Screenshot aus Voraca bei der Darstellung eines Volumens [1].

findet in beiden Renderer-Viewports statt. Natürlich lässt sich die Transferfunktion interaktiv anpassen, im Moment ist der mittlere blaue Punkt ausgewählt und könnte mit der Maus verschoben werden. Seine Materialeigenschaften lassen sich über die AntTweakBar des Viewports ändern. Die AntTweakBar ist in rot dargestellt, passend zur Umrandung des korrespondierenden Viewports. Verschiedene Materialtypen können über die Einstellung der Transferfunktion erzeugt werden. Die Vielfalt reicht von Knochen, über Metall, über Glas bis hin zu glühendem Gas. Im Renderer sind die verschiedene Techniken an- oder abgewählt. Auf dem Screenshot ist eine Kombination aus Pre-Integration, Jittering, Voxel Spaced Sampling und lokaler Beleuchtung zu sehen. Wie folgende Messung zeigt, ist es immer sinnvoll sowohl Early Ray Termination als auch Empty Space Skipping zu aktivieren. Beide Verfahren ergänzen sich, wie schon in Kapitel 3.6 erwähnt worden ist:

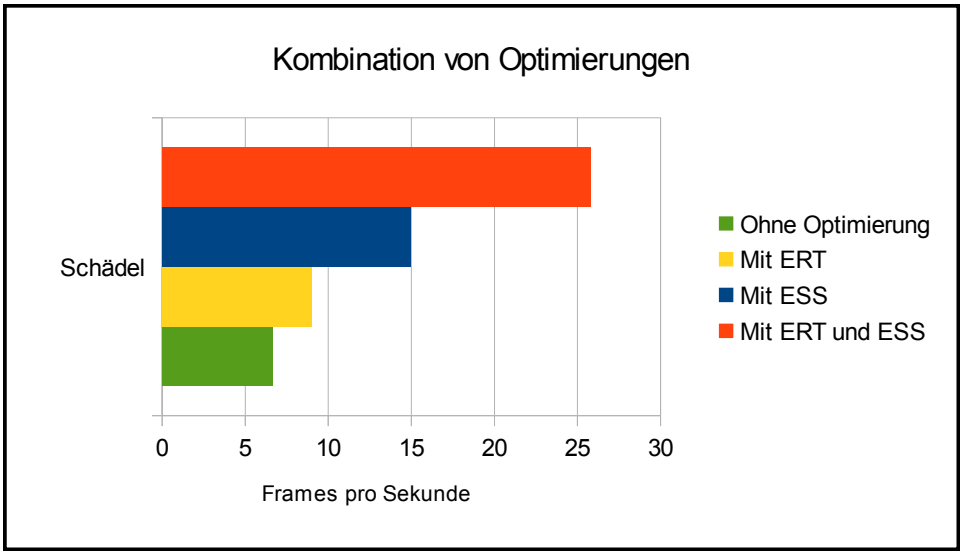

Abbildung 42: Messung der Performance bei der Aktivierung sowohl von Early Ray Termination als auch Empty Space Skipping.

Die Messung ist bei einem Vollbild-Viewport bei den in Abbildung 41 zu sehenden Einstellung vorgenommen worden. Es zeigt sich, dass beide Optimierungen eine immense Bedeutung für die Performance haben und keinerlei visuellen Beeinträchtigungen vorweisen. Andere Optimierungen wie adaptive Sampling bringen nicht immer eine Verbesserung, siehe dazu die Einschätzung in Kapitel 3.9.

Die Darstellung fordert auch aktuelle Grafikkarten enorm, wie die in allen Tests eingesetzte AMD Radeon 7950. Daher kann bei hohen Auflösungen nicht immer eine interaktive Framerate erreicht werden. Wenn die Einstellungen aber mit Bedacht gewählt werden, kann dem gewünschten Ergebnis bei angemessener Framerate sehr nahe gekommen werden.

5 Fazit und Ausblick

Diese Arbeit zeigt, dass moderne Grafikkarten hervorragend für die Darstellung von Volumendaten geeignet sind. Sie ermöglichen in Kombination mit aktuellen Schnittstellen wie OpenGL 3.3 eine Umsetzung des Ray-Casting Algorithmus ohne Umwege und erlaubt viele Verbesserungen sowohl hinsichtlich der optischen Qualität (Pre-Integration, Lokale Beleuchtung…) also auch in Hinblick auf die Performance (Early Ray Termination, Empty Space Skipping…). Eine große Erleichterung bilden vor Allem Schleifen und Kontrollflüsse in den Shadern, welche eine direkte Abbildung des Algorithmus mit Front-To-Back Komposition ermöglichen. Vor wenigen Jahren war es für das Rendering noch notwendig, mehrere Buffer zu benutzen und die Volumendaten aufgrund von begrenzen Videospeicher intelligent aufzuteilen und geschickt auf die Grafikkarte zu laden. Solche Probleme sind heute zwar immer noch bei hochaufgelösten oder zeitlichen Abfolgen von Volumendaten eine Herausforderung, die meisten statischen Volumina können aber ohne Probleme als ein Block in den Videospeicher geladen werden.

Das Rendering selbst ist aufgrund der stark genutzten Parallelität in Form von einem Aufruf des Fragment-Shader pro Kamerastrahl ebenfalls sehr gut auf der Grafikkarte umsetzbar und GPGPU-Sprachen wie Cuda, OpenCL oder der Compute-Shader versprechen zumindest bei diesem Aspekt keine Verbesserungen. Eventuell ergeben sich aber Vorteile bei großen Volumendaten, welche nicht komplett in den Videospeicher passen und stückweise geladen werden müssen. GPGPU-Sprachen bieten mehr Freiheiten in der Speicherverwaltung auf einer tieferen Ebene, was in diesem Fall eine Optimierung erlauben würde. Wovon die Umsetzung allerdings profitiert ist bessere Hardware. Videospeicher und Rechentakt der Grafikprozessoren haben direkten Einfluss auf die Ausmaße der interaktiv darstellbaren Volumendaten und Qualität der Visualisierung. Die Umsetzung in Voraca hat zwei Richtungen aufgezeigt, welche auf aktueller Hardware noch Raum für Verbesserungen lassen: Zum einen könnte globale Beleuchtung als iterativer Prozess umgesetzt werden und so eine realistischere Darstellung gewährleisten. Zum anderen könnte Post-Processing für den Renderer umgesetzt werden, sodass High Dynamic Range Rendering und Tone Mapping ermöglicht würden. Dafür wäre das Rendering in einen Framebuffer notwendig, was sich mit ein wenig Aufwand umsetzen ließe.

Zusammenfassend bleibt zu sagen, dass Ray-Casting heutzutage auf Desktop- und Notebooksystemen interaktiv umsetzbar ist und ältere Verfahren wie Slice Plane Rendering dort verdrängen wird. Es ist spannend zu verfolgen, wie sich mobile Plattformen entwickeln und ob sie ebenfalls bald eine interaktive Umsetzung erlauben.

Anhang

Testsystem

- CPU: Intel Core i7 920
- GPU: AMD Radeon 7950
- GPU-Treiber: Catalyst 13.12
- RAM: 6GB DDR3
- OS: Windows 8.1

Alle Messungen sind ungefähr bei eine Auflösung von 1900x1200 gemacht worden. Bei vergleichenden Messungen wurde auf eine exakt gleiche Auflösung geachtet und die gleichen Einstellungen für die Kamera vorgenommen.

Volumina in Abbildungen

1. "Baby Head", PVM, http://lgdv.cs.fau.de/External/vollib/
2. "MRI Head", PVM, http://lgdv.cs.fau.de/External/vollib/
3. "Stag beetle", DAT, http://www.cg.tuwien.ac.at/research/publications/2005/dataset-stagbeetle/
4. "Test Spheres", PVM, http://lgdv.cs.fau.de/External/vollib/
5. "Foot", PVM, http://lgdv.cs.fau.de/External/vollib/
6. "Lobster", PVM, http://lgdv.cs.fau.de/External/vollib/
7. "Bonsai #1 linear quantized version", PVM, http://lgdv.cs.fau.de/External/vollib/
8. "Teddy Bear", PVM, http://lgdv.cs.fau.de/External/vollib/
9. "Bucky Ball", PVM, http://lgdv.cs.fau.de/External/vollib/

Die Hyperlinks in dieser Auflistung wurden am 12. März 2014 auf ihren Inhalt überprüft.

Hyperlinks

Sofern nicht anders angegeben, sind Hyperlinks am 15. März 2014 auf ihren Inhalt geprüft worden.

Abbildungsverzeichnis

Listings

Literatur

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977.

[HKRs+06] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[MI04] Aaron Lefohn Charles Hansen Milan Ikits, Joe Kniss. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 39. Volume Rendering Techniques. Pearson Higher Education, 2004.

[Pho75] Bui-Tuong Phong. Illumination for Computer Generated Pictures. 18(6):311–317, 1975.