

# Ein Volume-Rendering-Framework für OpenSG

Manfred Weiler, Thomas Ertl

Abteilung Visualisierung und Interaktive Systeme  
Universität Stuttgart\*

## Überblick

Wir beschreiben ein auf OpenSG basierendes Volume-Rendering-Framework, das volumetrische Objekte definiert, die sich an beliebiger Stelle im Szenengraph einfügen lassen und zusammen mit polygonalen Objekten des Szenengraphen gezeichnet werden. Das Rendering verwendet Textur-basierte Volumenvisualisierung mit 2D- oder 3D-Texturen. Anwendungshintergrund ist primär die wissenschaftliche Visualisierung von Volumendaten aus Sensormessungen oder Simulationen. Daneben sind auch andere volumetrische Effekte wie Nebel oder Wolken möglich. Das Framework besteht aus wenigen, abstrakten Objekten, die eine einfache Schnittstelle für den Anwendungsentwickler bieten und gleichzeitig durch ein Shader-Konzept die flexible Erweiterbarkeit gewährleisten. Mit Hilfe der Shader können neue Visualisierungsalgorithmen, die auch zukünftige Features noch kommender Graphikchips voraussetzen, einfach integriert werden.

## 1 Einleitung

In den letzten Jahren hat die Volumenvisualisierung einen immer größeren Stellenwert eingenommen. Das liegt zum einen an der fortschreitenden Technologie, die in verschiedenen Gebieten die Aufzeichnung von Volumendaten ermöglicht, wie zum Beispiel in der Medizin (Computertomographie, Kernspintomographie etc.), der Strömungsmechanik oder der Akustiksimulation im Fahrzeugbau. Zum anderen hat sich die Graphik-Hardware dahingehend weiterentwickelt, dass 3D-Textur-Hardware als Basis für effiziente Volumenvisualisierung wohl bald auch zur Standardfunktionalität von PC-Graphikkarten gehört.

Den wachsenden Einsatzfeldern wurden die diversen Szenengraph-APIs bisher nicht gerecht, da sie lediglich flächige Daten in Form von polygonalen Netzen oder parametrischen Flächen repräsentierten. Die fehlende Unterstützung durch Programmierbibliotheken erschwerte die Realisierung von Anwendungen zur Volumenvisualisierung, insbesondere da interaktive Volumenvisualisierung, die für eine effiziente Analyse der Volumendaten unbedingt erforderlich ist, in hohem Maße von der Graphikhardware Gebrauch machen muss. Dies erfordert für den Programmierer sehr detaillierte Kenntnisse der Zielplattform und führt zu erheblichen Problemen bei der Portierung solcher Anwendungen.

Erst langsam halten volumetrische Objekte Einzug in freie und kommerzielle Szenengraph-APIs [1, 2]. Von einer flexiblen, erweiterbaren und plattformübergreifenden Lösung sind diese Ansätze jedoch noch weit entfernt. Einen ersten Vorschlag in dieser Richtung hat die Firma SGI bereits 1998 mit OpenGL Volumizer [4] gemacht. Er erlaubt die Integration von volumetrischen Objekten in beliebige OpenGL-Umgebungen. Die Anwendungsimplementierung kann dabei auf einer höheren Abstraktionsebene erfolgen, was die Programme auf den verschiedenen Plattformen

des Herstellers lauffähig macht. In der Version 2.0 des APIs hat SGI die Schnittstelle, die anfangs noch sehr low-level und nahe an OpenGL orientiert war, weiter abstrahiert, dabei aber die Plattformunabhängigkeit aufgegeben.

Mit dem Open-Source Projekt OpenSG steht nun ein neues, freies und portables Szenengraphsystem zur Verfügung. Es erhebt den Anspruch eines universellen Echtzeit-Rendering-Systems insbesondere auch für Anwendungen der Virtuellen Realität. Im Rahmen des BMBF-Projektes OpenSG PLUS soll dieses Szenengraphsystem um die Funktionalität der Volumendarstellung erweitert werden. Das vorliegende Dokument beschreibt ein auf OpenSG basierendes Volume-Rendering-Framework. Abschnitt 2 geht zunächst auf die Grundlagen und Mechanismen der Textur-basierten Volumenvisualisierung ein. Abschnitt 3 stellt einige grundlegende Konzepte von OpenGL Volumizer vor, welche das Design des Frameworks beeinflusst haben. Abschnitt 4 beschreibt Volume-Shader als Kernidee dieses Frameworks für plattformübergreifende und erweiterbare Volumeneffekte. Es schließen sich eine Beschreibung des Anwendungsinterface in Abschnitt 5 und einige Aspekte der Implementierung an.

## 2 Textur-basierte Volumenvisualisierung

Neben den zahlreichen Verfahren zur direkten Volumenvisualisierung wie Raycasting [9], Splatting [15] oder Shear-Warp [8] hat sich in den letzten Jahren die Textur-basierte direkte Volumenvisualisierung [3] als wichtiger Algorithmus für das Rendering von Volumendaten auf regulären Gittern herauskristallisiert, da er unter Ausnutzung der 3D-Texturhardware moderner Graphikchips vergleichsweise hohe Frameraten erreicht. Wie Abbildung 1 zeigt, wird das Volumen als 3D-Textur definiert. Innerhalb der Boundingbox des Volumens werden äquidistante Schnittebenen, sogenannte Slices, parallel zur Bildebene betrachtet. Die durch Schneiden der Slices mit der Volumen-Boundingbox entstehenden Schnittpolygone werden dann beginnend mit dem entferntesten mit aktivierter 3D-Textur gezeichnet und mit Alphablending in den Framebuffer geschrieben. Das Ergebnis entspricht unter gewissen Näherungsannahmen der Auswertung des Volumenintegrals in einem selbstleuchtenden Gas, analog zum Raycasting. Im Unterschied zum Raycasting erfolgt die Berechnung hier jedoch für alle Sehstrahlen parallel.

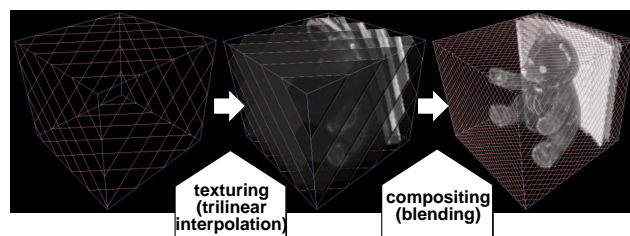


Abbildung 1: Direkte Volumenvisualisierung mit 3D-Texturen und Schichtpolygonen parallel zur Bildebene.

\*Universität Stuttgart, IfI, Abt. VIS, Breitwiesenstr. 20-22, 70565 Stuttgart, Germany; E-mail: {Manfred.Weiler | Thomas.Ertl}@informatik.uni-stuttgart.de.

Stehen auf einer Hardware nur 2D-Texturen zur Verfügung, so ist eine leichte Modifikation des Algorithmus aus Abbildung 2 erforderlich. Das Volumen wird als Stapel von 2D-Texturen definiert. Anstelle vom Schnittebenen parallel zur Bildebene werden nun Ebenen parallel zu einer der Hauptachsen der Boundingbox gerendert. Es wird immer diejenige Achse gewählt, die mit der Blickrichtung den kleinsten Winkel bildet, was eine Verdreifung des Texturspeichers bewirkt, da für jede Achse ein eigener Texturstapel erforderlich ist. Die Bildqualität ist dabei etwas schlechter, da nur eine bilineare Interpolation der Volumendaten erfolgt und es zu visuellen Artefakten aufgrund der Texturumschaltung kommt. Außerdem ist die Anzahl der Schichten durch die Anzahl der Texturen beschränkt. Rezk et al. [11] haben jedoch gezeigt, dass durch Einsatz von Multitexturen (ab OpenGL 1.2) die Bildqualität durch zusätzliche Zwischenschichten erhöht werden und auch die Textur-Replikation vermieden werden kann.

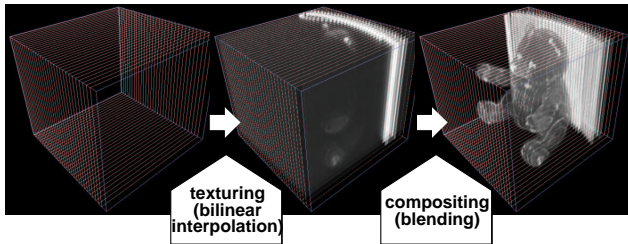


Abbildung 2: Direkte Volumenvisualisierung mit 2D-Texturen und Schichtpolygenen parallel zu den Volumenhauptachsen.

Eines der wesentlichen Probleme bei der Volumenvisualisierung mit 3D-Texturen ist die Größe des verfügbaren Texturspeichers, da Datensätze oftmals die Kapazität des Texturspeichers übersteigen. Dies erfordert die Unterteilung des Volumens in eine Anzahl sogenannter *Tiles* oder *Bricks*, die jeweils klein genug für den zur Verfügung stehenden Texturspeicher sind. Die Bricks müssen überlappen, um eine nahtlose Interpolation zu gewährleisten. Unter Verwendung der Bricks lassen sich die einzelnen Schritte zum 3D-Textur-basierten Volume-Rendering durch den in Abbildung 3 folgenden Pseudocode beschreiben:

```
Sortiere Bricks back-to-front
for (alle Bricks) {
    Lade Brick-Textur in den Texturspeicher
    for (alle Slices)
        Zeichne alle Polygone des Slice
        innerhalb des Bricks back-to-front
}
```

Abbildung 3: Pseudocode der Textur-basierten Volumenvisualisierung mit Bricking.

### 3 OpenGL Volumizer

Um die Konzepte hinter unserem Volume-Rendering-Framework besser zu verstehen, lohnt sich zunächst ein Blick auf bereits existierende Volumenvisualisierungs-APIs. Im Jahre 1998 wurde unter dem Namen OpenGL Volumizer erstmals eine allgemeine Programmierschnittstelle für Anwendungen zur direkten Volumenvisualisierung vorgestellt. Sie stellt dem Anwendungsprogrammierer Mechanismen für das Textur-Handling inklusive Bricking und

die Berechnung der Schnittpolygone zur Verfügung. Darüberhinaus existieren eine gewisse Menge von Hardware-Abstraktionen wie Farbtabelle, die mit vergleichsweise wenig Aufwand Anwendungen ermöglichen, die auf allen Plattformen der Firma SGI lauffähig sind. Während der Weiterentwicklung der Bibliothek wurde die Plattformunabhängigkeit leider aufgegeben. Zu den wichtigsten Konzepten von OpenGL Volumizer zählt die Trennung von Geometrie und Erscheinung (Appearance).

#### 3.1 Geometrie und Erscheinung

OpenGL Volumizer erweitert das Konzept von Geometrie und Textur, das schon seit Jahren erfolgreich für polygonale Oberflächen eingesetzt wird, konsistent für Volumengraphik. Genauso wie flächige Objekte ihr endgültiges Aussehen durch Kombination einer geometrischen Beschreibung mit einer Beschreibung der Erscheinungsbildes in Form einer Textur erhalten, werden volumetrische Objekte durch die geometrische Beschreibung ihrer Ausdehnung und die Beschreibung ihrer Erscheinung, der Volumendaten, definiert.

Neben der Konsistenz mit der Flächengraphik bietet diese konzeptionelle Trennung zusätzliche Flexibilität beim Rendering in Form von unterschiedlichen Zuordnungen zwischen Geometrie und Erscheinung. Die Geometrie kann sehr nahe an der Erscheinung orientiert sein wie bei der Textur eines Gemäldes, die auf ein quadratisches Polygon gemapped wird, oder der 3D-Textur, die auf einen Quader gleicher Größe abgebildet wird. Dies muss jedoch nicht so sein, wie im Fall der Textur, die auf eine Kugeloberfläche gelegt wird. Wie Abbildung 4 demonstriert, kann diese Flexibilität in der Volumenvisualisierung sinnvoll genutzt werden. Wird die Geometrie nur auf Teile des Volumens abgebildet, so ergibt sich die Einschränkung auf eine Region-of-Interest. Wird das Volumen als Ganzes auf ein irreguläres Polyeder abgebildet, entsteht eine Deformation des Volumens, die sich zur Registrierung medizinischer Datensätze einsetzen lässt [14].

Um die geometrische Ausdehnung des Volumens modellieren zu können, werden *Volumetrische Primitive* analog zu Dreiecken, Vierecken, Tristrips, Quadstrips usw. definiert. Basiselement des Volumens ist der Tetraeder. Um die Komplexität von Volumizer klein zu halten, werden höhere Primitive letztlich in Tetraeder zerlegt. Alle Operationen erfolgen auf den tessellierten Tetraedern. Ein Volumenprimitiv besitzt Eigenschaften in Form von Farbe, Texturkoordinaten oder Normale, die linear im Inneren des Volumenprimitives interpoliert werden.

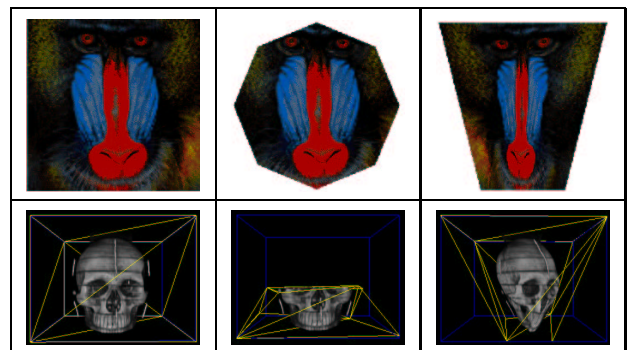


Abbildung 4: Unterschiedliche Möglichkeiten der Zuordnung von Geometrie und Volumen, analog zu der Zuordnung von Texturen zu Polygonen, erlaubt in natürlicher Weise die Volumendeformation oder Eingrenzung auf eine beliebige Region-of-Interest.

## 3.2 Programmierschnittstelle

Das Konzept der Volumenprimitive ist sehr allgemein und mit OpenGL Volumizer sollte ein flexibles API zum Rendern dieser Primitive geschaffen werden. Als Ergebnis blieb die Programmierschnittstelle auf niedriger Abstraktionsstufe relativ nahe an der Hardware. OpenGL Volumizer bietet Methoden für das Unterteilen einer 3D-Textur in mehrere Bricks, für das Sortieren der Bricks und für die Berechnung der Schnittpolygone je Tetraeder und Brick. Aus diesen Komponenten muss der Anwendungsprogrammierer seine Anwendung zusammenstellen. Dies erfordert noch immer erheblichen Programmieraufwand, weil der gesamte Grundalgorithmus erst implementiert werden muss. Portable Anwendungen, zumindest zwischen den unterstützten Plattformen, sind zwar möglich, erfordern aber explizite Verzweigungen im Programmcode für Plattformen mit 2D- bzw. 3D-Textur-Hardware und Plattformen mit Pre-Interpolation- bzw. Post-Interpolation-Farbtabelle. Weiterhin bleibt das Textur-Handling auf niedriger Abstraktionsstufe. Der Anwendungsprogrammierer muss sich insbesondere mit geeigneten OpenGL-Texturformaten auseinandersetzen.

Die Flexibilität zwingt zu Kompromissen bezüglich der Performanz. So wird auch für den einfachen Fall, dass als Geometrie für das Volumen dessen Boundingbox gewählt wird, eine Zerlegung in Tetraeder durchgeführt, wodurch Volumizer weniger performant ist als spezialisierte Programme für die Visualisierung kartesischer Volumendaten.

Ein weiteres Problem tritt bei der Berechnung der Slices auf. Die Slice-Polygone werden erst komplett, d.h. für alle Tetraeder und für alle Bricks gemeinsam, berechnet und in eine große Datenstruktur im Speicher geschrieben. Die Renderroutine liest diese Polygone wieder aus und übergibt sie an die OpenGL-Pipeline. Das dadurch erforderliche mehrfache Kopieren geht auf Kosten der Performanz. Darüberhinaus werden die Daten in anderer Reihenfolge in den Speicher geschrieben als sie gelesen werden, was zur Folge hat, dass mindestens ein Zugriff eine schlechte Cachekohärenz aufweist, was angesichts der Größe des Datenfeldes ein Problem darstellt.

Die Version 2.0 von OpenGL Volumizer führt ein deutlich abstrakteres szenengraphähnliches Interface ein, gibt dabei jedoch die Plattformunabhängigkeit vollständig auf und unterstützt nur noch SGI InfiniteReality3-Graphiksysteme mit 3D-Textur-Hardware. Es existiert ein Volume-Shape, das eine Appearance und eine Geometrie enthält. Die Appearance speichert die Parameter des Volumens und der Visualisierung wie z.B. die 3D-Textur, eine Farbtabelle oder einen Lichtvektor für beleuchtete Volumen. Es stehen verschiedene Shader-Objekte zur Verfügung, die mit dem Volumen verbunden werden und den Visualisierungsalgorithmus bestimmen. Volumizer 2.0 liefert eine Reihe von vordefinierten Shadern mit, wie zum Beispiel einen Primitiv-Shader, der nur texturierte Polygone zeichnet, oder einen Shader, der zusätzlich eine Transferfunktion anwenden kann. Der Anwendungsprogrammierer definiert lediglich die einzelnen Komponenten und steckt sie ähnlich einem Baukastensystem zusammen. Einblick in den eigentlichen Rendering-Algorithmus hat er nicht mehr. Er braucht sich auch nicht um das Textur-Management zu kümmern. Nachteil dieser abstrakten Schnittstelle ist jedoch, dass es keine Möglichkeit gibt, eigene Shader zu implementieren ohne das komplette Slicing und Textur-Management selbst nachzuprogrammieren. Auch Hardware-Abstraktionen bleiben unberücksichtigt.

## 4 Volume-Shader

Wesentliche Designziele für dieses OpenSG-Framework waren einfache Programmierung für den Anwender bei gleichzeitiger flexibler Erweiterbarkeit im Hinblick auf neue Hardware und neue Algorithmen zur Volumenvisualisierung. Der Begriff Algorithmus

meint in diesem Zusammenhang Variationen der Textur-basierten Volumenvisualisierung mittels Slicing. Ein neuer Algorithmus sollte dabei jedoch nicht die komplette Reimplementierung von Slicing und Textur-Management beinhalten. Nachdem, wie in Abschnitt 2 gezeigt, alle Algorithmen zur Textur-basierten Volumenvisualisierung ein gemeinsames Grundgerüst haben, bietet sich ein Callback-Mechanismus an. Als Einsprungpunkt für diese Callbacks dient ein Shader-Objekt ähnliche wie in Volumizer 2.0. Dieser Mechanismus erlaubt sowohl die Integration neuer Visualisierungseffekte als auch die einfache Unterstützung unterschiedlicher Graphikhardware in Form von angepassten oder neuen Shader-Objekten.

### 4.1 Hardware-Abstraktion

Unterschiede zwischen den einzelnen Algorithmen bestehen im OpenGL-Setup je Brick bzw. je Slice und in den Texturen, die zum Rendering verwendet werden. Darüberhinaus können Unterschiede abhängig von den Fähigkeiten der Hardwareplattform, auf der die Anwendung läuft, bestehen. Einige Graphikchips unterstützen beispielsweise ein Farbmapping nach dem Texture-Lookup womit sich Transferfunktionen für die Visualisierung realisieren lassen, andere Graphik-Adapter benötigen für dasselbe Ergebnis eine Dependent-Texture und spezielle Per-Pixel-Operationen.

Eine Hardwareabstraktion für Transferfunktionen alleine wäre noch verhältnismäßig einfach zu realisieren. Probleme entstehen dann, wenn unterschiedliche Funktionen kombiniert werden. Sollen beispielsweise beleuchtete Volumen mit gleichzeitig angewendeter Transferfunktion gerendert werden, so kann der Fall auftreten, dass beide Funktionen Per-Pixel-Operationen verwenden müssten. In diesem Fall muss die Pixel-Pipeline mit der kombinierten Operation programmiert werden. Die allgemeine Lösung dieses Problems würde die Verwendung einer Shading-Language erfordern. Jede Teil-Funktionalität, wie Transferfunktionen oder Beleuchtungsrechnung, könnte dann als Funktion ähnlich einer C-Funktion in dieser Sprache formuliert werden. Aufbauend darauf lassen sich durch Kombination der bereits existierenden Funktionen, analog zu C-Funktionsaufrufen, neue Funktionen definieren und damit zusätzliche Effekte erreichen. Proudfoot et al. [10] haben ein solches System vorgestellt. Es besteht aus einer Shader-Sprache und einem Compiler, der die Syntax der Sprache in eine Zwischenrepräsentation überführt und in der Lage ist daraus, mit Hilfe verschiedener Back-Ends Hardware-Setups für unterschiedliche Plattformen zu generieren.

Das vorgestellte Framework erlaubt zwar problemlos die Verwendung von Shader-Languages wie sie für OpenGL 2.0 diskutiert werden. Zum jetzigen Zeitpunkt aber wäre eine Integration in OpenSG zu aufwändig. Daher stellen wir stattdessen verschiedene Shader, mit sinnvollen Funktionen für die Volumenvisualisierung für aktuelle Hardwareplattformen zur Verfügung und überlassen die plattformübergreifende Implementierung neuer Algorithmen dem Anwender. Das Framework erleichtert diese Arbeit jedoch erheblich, da ausschließlich einige wenige Callbacks der Shader-Objekte plattformabhängige Implementierungen besitzen müssen.

### 4.2 Shader-Schnittstellen

Um den Anforderungen unterschiedlicher Visualisierungsalgorithmen gerecht zu werden, wurden mehrere Module für das Framework entworfen. Abbildung 5 stellt die einzelnen Komponenten und ihr Zusammenwirken im Detail dar.

Das Framework besteht aus vier Komponenten, einem Renderer, einem Texture-Manager, einem Slicer und dem Shader. Die zentrale Steuerinstanz ist der Renderer, der die übrigen Komponenten daher auch initialisiert. Der Shader entscheidet bei der Initialisierung abhängig vom gewünschten Visualisierungseffekt wie viele Texturen und welche Art von Texturen er benötigt. So kann er beispiels-

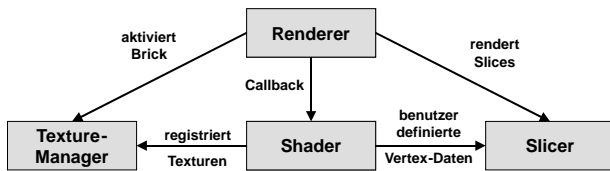


Abbildung 5: Im Zusammenspiel zwischen Shader, Textur-Management und Slicer lassen sich unterschiedliche Visualisierungseffekte mit Textur-basierter Volumenvisualisierung erreichen.

weise für einfaches Slicing eine Luminanz-Textur mit den Volumenskalarkwerten definieren, für mit Slicing gerenderte beleuchtete Isoflächen, wie in [13] vorgeschlagen, können RGBA-Texturen mit Gradienteninformation in RGB und dem Volumenskalarkwert im Alpha-Kanal definiert werden. Der Shader kann auch mehrere Texturen beim Texture-Manager registrieren. Da der Texture-Manager Multitexturen unterstützt, sind interpolierte Zwischenschichten zwischen 2D-Texturen entsprechend [11] ebenso möglich wie Dependent-Texturen, die beispielsweise Engel und Kraus [5] verwenden, um mit Hilfe von vorintegrierten Strahlstücken des Volumenintegrals eine neue Qualität des Renderings zu erreichen. Eine weitere Anwendung von Multitexturen sind Tag-Texturen, oder Stencil-Texturen, die mit Hilfe des OpenGL-Alpha-tests das Ausblenden bestimmter Voxel des Volumens durch Multiplikation mit einem binären Klassifikationswert erlauben. Für alle benötigten Texturen wählt der Shader auch geeignete OpenGL-Texturformate und übergibt die Texturen an den Texture-Manager.

Der Texture-Manager realisiert das Ressourcen-Management für die Texturen inklusive Bricking. Er untersucht die Hardware in Hinblick auf verfügbaren Texturspeicher und lässt den Shader nur so viele Texturen registrieren, wie Texturereinheiten verfügbar sind. Schließlich erfolgt die Zerlegung in einzelne Bricks. Dabei unterscheidet der Manager Texturen, die nicht zerlegt werden dürfen, wie Dependent-Texturen zur Realisierung einer Transferfunktion oder zum Nachschlagen im Rahmen des Pre-Integrated Volume-Rending. Diese werden zunächst vom verfügbaren Texturspeicher abgezogen. Anschließend werden die übrigen Texturen, die Datenwerte oder beispielsweise Gradienten je Voxel beschreiben, so unterteilt, dass jeweils ein Brick von jeder Textur gleichzeitig in den Texturspeicher geladen werden kann.

Nach der Initialisierung steuert der Renderer den eigentlichen Slicing-Algorithmus wie in Abbildung 3 angedeutet. Er sortiert die Bricks back-to-front und aktiviert über den Texture-Manager alle jeweiligen Brick-Texturen. Für jeden Brick ruft er einen entsprechenden Callback des Shaders auf, der den korrekten OpenGL-State initialisiert. Danach rendert er über den Slicer alle Slice-Polygone des Bricks. Auf diese Weise entfällt eine aufwändige Zwischenspeicherung der Slice-Polygone, die stattdessen direkt als OpenGL-Polygone definiert werden können.

Für manche Visualisierungseffekte kann es erforderlich sein, zusätzliche Parameter je Vertex zu definieren. Für Deformation des Volumens sind beispielsweise explizite Texturkoordinaten erforderlich, weil hierbei die Texturkoordinate bei Verschieben der Vertexposition konstant bleibt. Im Allgemeinen lässt sich keine Aussage darüber machen, welche Parameter zukünftige Shader erfordern werden. Um zu vermeiden, den Algorithmus für die Slice-Berechnung jeweils an unterschiedliche benutzerdefinierte Knotendaten anpassen zu müssen, besitzt der Shader die Möglichkeit beliebige Daten pro Vertex zu definieren, die bei der Berechnung der Slice-Polygone linear auf den Kanten der Geometrie interpoliert werden. Anstelle unterschiedliche Knotenformate zu definieren, wie in Volumizer realisiert, schlagen wir ein generisches Sam-

melformat vor. Als User-Data je Knoten kann dadurch eine beliebige Anzahl von Float-Werten definiert werden.

Dies erfordert jedoch eine eigene Routine zum Rendering der Slice-Polygone durch den Shader. Sie muss die benutzerdefinierten Knotenparameter in geeigneter Weise interpretieren und in die richtigen OpenGL-Kommandos umsetzen. Shader-spezifische Render-Methoden werden ebenfalls über einen Callback angesteuert, der als Parameter das jeweilige Slice-Polygon mit allen interpolierten Knotendaten erhält, was wiederum einen Overhead durch Schreiben, Lesen und Kopieren der Daten bewirkt. Daher ist dies nicht der Standardmechanismus, sondern wird nur verwendet, wenn der Shader explizit eine eigene Render-Methode beim Renderer registriert.

## 5 Volumen-Modellierung mit OpenSG

Das Konzept der Volume-Shader erlaubt eine flexible Anpassung des Volume-Rending-Frameworks an unterschiedliche Hardware-Funktionalität und die einfache Realisierung neuer Algorithmen. Für den Anwendungsentwickler, der Volumeneffekte oder Visualisierungen in seine OpenSG-Szene aufnehmen will, ist daneben aber auch eine möglichst einfache und übersichtliche Struktur der Szenengraphobjekte erforderlich. Sie wurde basierend auf OpenSG-Nodecore und OpenSG-Feldcontainern entsprechend der Darstellung in Abbildung 6 realisiert.

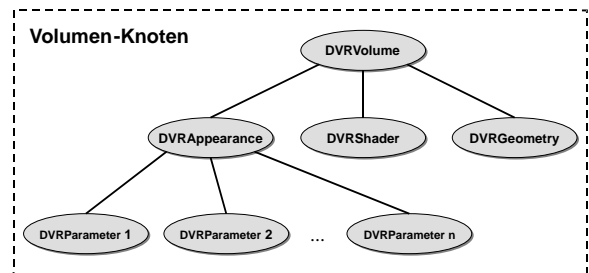


Abbildung 6: Aufbau des OpenSG-Volumenknotens aus einzelnen Feldcontainern.

Das Framework enthält nur einen einzigen neuen Nodecore, DVRVolume, der in den Szenengraph eingehängt werden muss. Diese Struktur wurde gewählt, weil eine Hierarchie von Knoten, die entsprechend den Konzepten des Szenengraph-API traversiert wird, zu unflexibel ist und außerdem die Gefahr birgt, dass der Anwendungsprogrammierer unerwartete Knoten in den Volumenteilbaum einhängt, was zu Problemen führen kann. Alle anderen dargestellten Objekte sind OpenSG-Feldcontainer. Nachdem OpenSG eine transparente Datenverteilung und -replikation für mehrere Threads auf der Basis der Feldcontainer realisiert, gewährleistet dies die Funktionsfähigkeit des Volumenknotens auch in Multi-Threading- und Cluster-Umgebungen. Einem Einsatz des Volumenknotens in VR-Umgebungen steht demnach nichts im Wege.

Der Volumenknoten enthält eine Appearance mit unterschiedlichen Parametern, eine Geometry, welche die geometrische Ausdehnung des Volumens beschreibt sowie einen Shader, der in Abschnitt 4 bereits dargestellt wurde.

### 5.1 DVRAppearance

Die Appearance wird abgeleitet von der Klasse `osg::Material` und beschreibt das Erscheinungsbild, also die visuellen Attribute, des Volumens. In erster Linie sind dies die Volumendaten, die Skalarwerte in den Voxeln. Sie werden als 3D-Bild

(`osg::Image`) gespeichert, wodurch die allgemeine Ladefunktionalität von OpenSG ausgenutzt werden kann. Hinzu kommen weitere Parameter, die das Erscheinungsbild der Visualisierung beeinflussen: Eine Transferfunktion bildet die Skalarwerte auf Farbe und Transparenz ab, wobei Transferfunktionen, wie in [7] vorgeschlagen, mehr als eine Dimension besitzen können. Ambiente und spekulare Farbe beschreiben die Beleuchtungsparameter für beleuchtete Isoflächen. Weitere Parameter wie Tag-Volumen oder Environment-Maps sind denkbar. Diese Volumenparameter verwendet der Shader um die erforderlichen Texturen aufzubauen und den passenden OpenGL-State zu setzen.

Jeder Shader kann einen unterschiedlichen Satz an Parametern benötigen. Diese müssen als Member in der Appearance enthalten sein. Im Sinne einer Erweiterung des Volumenvisualisierungs-APIs soll es möglich sein, neue Shader zu implementieren. Dies erfordert die nachträgliche Erweiterung der Appearance-Klasse. Eine Ableitung der Basisklasse zur Erweiterung wäre jedoch ein zu großer Overhead und führt zu Typ-Problemen sowie zur Notwendigkeit der Verwendung von Mehrfachvererbungen. Deshalb wird der Attachment-Mechanismus von OpenSG verwendet, der prinzipiell eine beliebige Anzahl benutzerdefinierter Daten im Sinne von User-Data-Zeigern realisiert. Alle Parameter in der Appearance sind von der Klasse `osg::Attachment` abgeleitet. Werden für neue Shader weitere Parameter erforderlich, so sind diese ebenfalls von `osg::Attachment` abzuleiten. Jedes Attachment erhält einen eindeutigen Namen, das seine Identifizierung und damit die Referenzierung durch die Shader erlaubt. Bei der Initialisierung überprüft der Shader, ob die Appearance die erforderlichen Attachments enthält, und erzeugt gegebenenfalls einen Fehler.

## 5.2 DVRGeometry

Die Geometrie beschreibt die geometrische Ausdehnung des Volumens und ist als eine Spezialisierung von `osg::Geometry` realisiert. Sie kann als beliebiges Dreiecksnetz beschrieben sein und somit innerhalb der Datenstrukturen von `osg::Geometry` gespeichert werden. Als Einschränkung gilt jedoch, dass es sich um ein geschlossenes Dreiecksnetz handeln muss. Aus diesem Dreiecksnetzes erfolgt die Berechnung der Slices durch Methoden der Klasse `DVRGeometry`. Dazu wird ein iterativer Algorithmus verwendet, der den zuletzt berechneten Slice heranzieht, um den nächsten Slice sehr schnell zu ermitteln. Das Slice-Polygon wird als geschlossener Linienzug zurückgeliefert, was nachträglich ein Clipping an den Brick-Grenzen oder einer zusätzlich verwendeten Clip-Geometrie entsprechend Sutherland-Hodgman [12] ermöglicht. Neben diesem allgemeinen Dreiecksnetz ist auch ein `DVRBlock` vorgesehen, der eine einfache Würfelgeometrie repräsentiert. Eine Sonderbehandlung dieser Geometrie erachten wir als gerechtfertigt, da sie den häufigsten Fall darstellen wird und sich die Schnittpolygone in einem Block im Vergleich zum Dreiecksnetz sehr effizient berechnen lassen.

## 6 Volumen-Rendering mit OpenSG

Das bisherige Design von OpenSG weist bezüglich der hier vorgestellte Volumenvisualisierung einige Schwächen auf. So ist beispielsweise ein `osg::Geometry`-Objekt eine relativ umfangreiche Datenstruktur, was in der enormen Flexibilität begründet liegt. OpenSG-Geometrieobjekte erlauben das gemeinsame Speichern gemischter Primitive wie Dreiecke oder Quadstrips als indiziertes Knotenfeld mit verschiedenen Knotenparametern – Farbe, Texturkoordinaten, Normalen usw. Jeder Knoten kann dabei sogar mehrere Parameter des gleichen Typs wie beispielsweise mehrere Normalenvektoren besitzen. Naturgemäß ist die Erzeugung von OpenGL-Befehlen aus dieser komplexen Datenstruktur für einfache Polygone weniger effizient. In der Tat führt das in der momen-

tanen Implementierung dazu, dass bei Verwendung von OpenGL-Displaylisten, welche eine einmal erzeugte Geometrie cachen, eine OpenSG-Anwendung eine deutlich höhere Performanz erreicht als ohne. Für sich dynamisch ändernde Geometrie, wie im Falle von an der Bildebene ausgerichteten Slice-Polygonen, ist dies sehr ungünstig.

Ein weiteres Problem entsteht aus der starren Zuordnung zwischen Material und Geometrie in OpenSG. Jede Geometrie kann nur ein Material besitzen. Für die hier vorgestellte Anwendung der Textur-basierten Volumenvisualisierung bedeutet dies, dass beispielsweise bei Verwendung von 2D-Texturen jeder Slice in einem eigenen Geometrieobjekt enthalten sein muss. Jeder Slice benötigt in diesem Fall seine eigene Textur, also sein eigenes Material. Für eine große Anzahl von Slices stellt dies einen unverhältnismäßigen Overhead dar. Als Lösung für dieses Dilemma schlagen wir Multipass-Materialien vor.

### 6.1 Multipass-Materialien

Ein Material abstrahiert nicht nur einen einzigen Satz von OpenGL-States, mit dem eine Geometrie gezeichnet wird, sondern mehrere. Das Rendern der Geometrie erfolgt dementsprechend in mehreren Durchläufen. Zunächst wird der erste Satz von OpenGL-States aktiviert und die Geometrie gezeichnet, dann der zweite Satz mit derselben Geometrie usw. Auf diese Weise entsteht ein einfache Möglichkeit für Multipass-Rendering und es lassen sich sehr flexible „Materialien“ realisieren [6, 10]. Im Rahmen einer Erweiterung von OpenSG wird zur Zeit an einem solchen Konzept der Material-Abstraktion gearbeitet. Die Idee der Multipass-Materialien überträgt sich in natürlicher Weise auf die Textur-basierte Volumenvisualisierung. Das Multipass-Material speichert die einzelnen Bricks als States und ruft mit jedem aktivierten Brick die Geometrie auf, welche die entsprechenden Slice-Polygone dynamisch generiert. Zur Realisierung dieses Konzepts muss allerdings auch die Szenengraph-Traversierung in OpenSG leicht angepasst werden.

### 6.2 Traversierung und Rendering Backend

Die Traversierung des Szenengraphen in OpenSG erfolgt in einer Reihenfolge, welche die OpenGL-State-Changes minimiert um unnötige Stillstände der OpenGL-Pipeline zu vermeiden. Realisiert wird dies durch ein sogenanntes Rendering Backend, das aus einem weiteren gerichteten Graphen, parallel zum Szenengraph besteht, in den Referenzen auf die sichtbaren Knoten in der Reihenfolge des Rendering einsortiert werden. Dabei dienen die Materialknoten, welche die Abstraktion für den OpenGL-State bilden, mit dem eine Geometrie gezeichnet werden soll, als innere Knoten und die Geometrien als Blätter. Alle Geometrien, die mit demselben OpenGL-State gezeichnet werden, erscheinen als Kinder unterhalb eines gemeinsamen Materialknotens.

Unter Verwendung des Rendering Backend erfolgt das Rendern in OpenSG in zwei Durchläufen. Der erste Pass läuft über den eigentlichen Szenengraphen und baut den Rendering Graph auf, den sogenannten *draw tree*. In einem zweiten Pass, dem eigentlichen Rendering Pass, wird der Graph des Rendering Backends in einer Depth-First-Strategie traversiert. Die Struktur des draw tree reduziert dabei die State-Changes automatisch.

Für die Unterstützung von Multipass-Materialien müsste lediglich der OpenGL-State, für den OpenSG bereits Abstraktionen in Form der Klasse `osg::State` vorsieht, als Sortierkriterium verwendet werden und die inneren Knoten des draw tree müssten auf State-Objekte statt auf die Materialknoten verweisen. Außerdem müssten die Geometrienknoten wissen, welches State-Objekt gerade aktiv ist, um unterschiedliche Geometrien erzeugen zu können. Dies kann auf einfache Weise als Member der

`osg::DrawAction` geschehen, die der eigentlichen Geometrie-Zeichenmethode als Parameter mitgegeben wird. Die Volumengeometrie kann somit über den State erfahren, welcher Brick des Volumens gerade aktiv ist, und die entsprechenden Slice-Polygone für OpenGL generieren.

## 7 Zusammenfassung

In diesem Dokument wurde ein auf OpenSG basierendes Framework für Textur-basierte Volumenvisualisierung mit 2D- oder 3D-Texturen vorgestellt. Das Framework lässt sich einfach verwenden und erlaubt die Integration von volumetrischen Objekten in beliebige mit OpenSG modellierte Szenen. Es wurde versucht, die vorteilhaften Konzepte bereits existierender APIs aufzugreifen, deren Schwächen jedoch zu vermeiden. Um die unterschiedlichen Plattformen von OpenSG unterstützen zu können und für Weiterentwicklungen der Hardware gerüstet zu sein, wurde ein Hardware-Abstraktionkonzept basierend auf der Idee der Volume-Shader vorgestellt, das durch Ableitung eigener Shader ebenso eine einfache Integration neuer Visualisierungsalgorithmen auf der Grundlage texturierter Slice-Polygone erlaubt. Die Implementierung dieses Frameworks in OpenSG ist zwar möglich, erfordert jedoch für eine effiziente Umsetzung im Rahmen dieses APIs kleinere Modifikationen.

## Referenzen

- [1] <http://openrm.sourceforge.net/>.
- [2] <http://www.tgs.com/>.
- [3] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *1994 Symposium on Volume Visualization*, pages 91–98, October 1994.
- [4] George Eckel. *OpenGL Volumizer Programmer's Guide*. Silicon Graphics Computer Systems, Mountain View, CA, USA, 1998.
- [5] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01*, Annual Conference Series, pages 9–16. Addison-Wesley Publishing Company, Inc., 2001.
- [6] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In Alyn Rockwood, editor, *Proceedings of the Conference on Computer Graphics (Siggraph99)*, pages 171–178, N.Y., August 8–13 1999.
- [7] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *IEEE Visualization '01*, pages 255–262. IEEE CS Press, October 2001.
- [8] Philippe Lacroute and Marc Levoy. Fast volume rendering using shear-warp factorization of the viewing transformation. In Andrew Glassner, editor, *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 451–458, July 1994.
- [9] Mark Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(3):29–37, 1988.
- [10] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings, Annual Conference Series*, pages 159–170. ACM Press / ACM SIGGRAPH, 2001.
- [11] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '00*, pages 109–118, 147. Addison-Wesley Publishing Company, Inc., 2000.
- [12] B. Sutherland and G.W. Hodgman. Reentrant Polygon Clipping. *CACM 17(1)*, pages 32–42, January 1974.
- [13] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (SIGGRAPH 98 Proceedings)*, pages 169–177, 1998.
- [14] R. Westermann and C. Rezk-Salama. Real-Time volume deformations. In *Eurographics '01*, pages 443–451, 2001.
- [15] Lee Westover. Footprint evaluation for volume rendering. *Computer Graphics*, 24(4):367–376, August 1990. ACM Siggraph '90 Conference Proceedings, ACM Press, 1990.