UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

MeVis RESEARCH

Ein modulares Shaderframework für Volumenrendering

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von
Stephan Palmer

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
Dipl.-Inf. Matthias Raspe
Institut für Computervisualistik, AG Computergrafik

Zweitgutachter: Dr.-Ing. Felix Ritter
Dipl.-Inf. Alexander Köhn
MeVis Research GmbH

Koblenz, im Januar 2008

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)

Danksagung

Großen Dank möchte ich meinen beiden Betreuern Dipl.-Inf. Alexander Köhn und Dipl.-Inf. Matthias Raspe aussprechen, die mir während des gesamten Bearbeitungszeitraums eine hervorragende Betreuung zukommen ließen. Neben fachlichen Ratschlägen war auch ihre moralische Unterstützung äußerst wertvoll. Mein Dank gilt darüber hinaus Felix Ritter und Florian Link von MeVis Research, die immer ein offenes Ohr für Fragen und Bitten hatten und in vielen Gesprächen Anteil an meiner Diplomarbeit genommen haben. Besonderer Dank gilt meinen fleißigen Korrekturlesern, allen voran Christian Schumann, Herbert Palmer, Rodja Trappe, Timo Dickscheid und Andreas Langs, die mit ihren wertvollen Ratschlägen zum Gelingen dieser Arbeit beitrugen. Nicht zuletzt möchte ich mich bei meinen Eltern bedanken, die mich bei allem, was ich je tat, uneingeschränkt unterstützt haben.

Stephan Palmer, Januar 2008

Inhaltsverzeichnis

1 Einleitung 1
1.1 Problemstellung . . . 1
1.2 Zielsetzung . . . 2
1.3 Kontext . . . 2
1.4 Aufbau der Arbeit . . . 3

2 Grundlagen des GPU-basierten Volumenrenderings 4
2.1 Grundlagen der Grafikhardware . . . 4
2.1.1 Geometrische Primitive und Texturen . . . 4
2.1.2 Die Grafikpipeline . . . 5
2.1.3 Programmierbarkeit . . . 7
2.2 Volumenrendering . . . 10
2.2.1 Theoretische Grundlagen . . . 10
2.2.2 Die Volumenrendering-Pipeline . . . 13
2.2.3 Umsetzung auf der GPU . . . 20

3 Shadererstellung 24
3.1 Überblick . . . 24
3.2 General Purpose GPU . . . 26
3.3 Datenflussbasierte Beschreibung von Shadern . . . 27
3.3.1 Datenflussprogrammierung . . . 27
3.3.2 Visuelle Datenflussprogrammierung . . . 29
3.3.3 Shadetrees . . . 32
3.3.4 Visuelle Shaderprogrammierung . . . 36
3.4 Metaprogrammierung mit C++ . . . 38
3.5 Kombinierbare Shader . . . 41
3.6 Einsatzzweck und Eignung . . . 44

4 Entwicklungswerkzeuge bei MeVis Research 45
4.1 Das Entwicklungsframework MeVisLab . . . 45
4.1.1 Aufbau . . . 46
4.1.2 Der Giga Voxel Renderer (GVR) . . . 47
4.2 Volumenrendering bei MeVis: Eine Bestandsaufnahme . . . 50
4.2.1 Überblick und Motivation . . . 50
4.2.2 Auswertung . . . 51
4.2.3 Zusammenfassung . . . 55
4.2.4 Bewertung . . . 56

5 Entwurf eines modularen Shaderframeworks 58
5.1 Analyse . . . 58
5.1.1 Einordnung in ein Volumenrendering-Programm . . . 58
5.1.2 Anforderungen . . . 61

5.1.3 Bewertung möglicher Ansätze 62
5.1.4 Auswahl eines geeigneten Ansatzes 64
5.2 Konzept 65
5.2.1 Grundstruktur der Shaderrepräsentation 65
5.2.2 Verarbeitung 74
5.2.3 Erweiterung für Schleifen 77
5.2.4 Einsatz im Volumenrendering 81

6 Praktische Umsetzung 85
6.1 Shader Utility Library (SUL) 85
6.1.1 Beschreibung und Speicherung 86
6.1.2 Erstellung und Verarbeitung 89
6.2 Integration der SUL in den GVR 90
6.2.1 Verwaltung des Shadergraphs 90
6.2.2 Schnittstellen 91
6.2.3 Beschleunigung 92
6.3 Grafische Benutzungsschnittstelle 94
6.3.1 Auswahl eines MeVisLab-Modultyps 95
6.3.2 Realisierung 96

7 Ergebnisse 103
7.1 Exemplarische Anwendungen 103
7.1.1 Preintegriertes Volumenrendering 103
7.1.2 Jittering 107
7.1.3 Lit Sphereshading 108
7.1.4 Raycasting 108
7.2 Leistungsvergleich 112
7.2.1 Optimierung durch die Compiler 112
7.2.2 Benchmarks 114

8 Diskussion 118
8.1 Bewertung 118
8.2 Ausblick 120

Anhang **122**

A Programmcode **122**

B Fragebogen zur Umfrage **130**

C Inhalt der beigefügten CD **132**

D Verzeichnisse **133**
Glossar 133
Abkürzungsverzeichnis 134
Abbildungsverzeichnis 135
Tabellen- und Algorithmusverzeichnis 136
Literaturverzeichnis 137

Notation

Im Text werden Auszeichnungen für Wörter besonderer Bedeutung verwendet. Eine gesonderte Kennzeichnung englischer Wörter wird nicht vorgenommen. Englische Fachbegriffe sind im Vokabular deutschsprachiger Informatik verbreitet. Diese Arbeit verzichtet auf die zusätzliche Übersetzung üblicher englischer Begriffe. Das Glossar ab Seite 133 enthält Fachbegriffe, die im Grundlagenteil der Arbeit nicht behandelt werden.

Bedeutung	**Erläuterung**
Betonung	Einführung neuer Begriffe und allgemeine Betonungen.
`Programmcode`	Klassen- oder Funktionsnamen im Fließtext.
EIGENNAMEN	Firmen-, Projekt- und Programmnamen.

1 Einleitung

Computer werden seit mehr als zwei Jahrzehnten zur Erzeugung von Bildern eingesetzt. Ihre Leistungsfähigkeit steigt auch heute noch stetig an, was sich auch in der Wirklichkeitstreue und Komplexität der erstellten Bilder widerspiegelt. Computererzeugte Bilder finden sich in vielen verschiedenen Anwendungen. So sind z. B. Computeranimationen für Filme nicht mehr wegzudenken. Vielleicht das bekannteste Einsatzfeld sind Computerspiele, deren fortwährender Erfolg die treibende Kraft hinter der besonders rasanten Entwicklung spezieller Hardware zur Bilderzeugung ist. Wenige andere Komponenten von Computern konnten in den vergangenen Jahren mit der Leistungssteigerung der Grafikkarten mithalten.

Ein weiteres Anwendungsgebiet ist die Darstellung von Volumendaten. Diese dreidimensionalen Bilddaten beschreiben ein Abbild nicht nur in Höhe und Breite, wie dies z. B. Fotografien oder Röntgenaufnahmen tun. Volumendaten fügen die Tiefe als dritte Dimension hinzu und können als Stapel vieler zweidimensionaler Einzelbilder aufgefasst werden. Typische Volumendatensätze entstehen durch bildgebende Aufnahmeverfahren der Tomographie. Sie fertigen volumetrische Abbildungen realer Gegenstände an, indem viele Schichten durch das Objekt abgetastet werden. Tomographieverfahren werden in vielen Industriebereichen eingesetzt, um das Innere von Werkstoffen sichtbar zu machen. Aus der Medizin sind diese Aufnahmetechniken nicht mehr wegzudenken. Sie erlauben den Blick in den Körper eines Patienten und damit einzigartige diagnostische Möglichkeiten.

Verfahren des Volumenrenderings ermöglichen die Darstellung von Volumen auf dem Computerbildschirm. Die Möglichkeit, die Funktionalität moderner Grafikkarten durch sogenannte Shaderprogramme anpassen zu können, erlaubt ihren Einsatz zur Volumenvisualisierung.

1.1 Problemstellung

Schlüsseleigenschaft der Grafikhardware, die ihren Einsatz für Volumenrendering attraktiv macht, ist die hochoptimierte Verarbeitung von Bilddaten in Form von Texturen. Volumen werden als dreidimensionale Texturen auf die Grafikkarte übertragen. Shaderprogramme übernehmen die für Volumenrendering spezifischen Aufgaben bei der Erstellung des Bilds, und bestimmen maßgeblich die Ergebnisse des Volumenrenderings. Um die Darstellung zu verändern, muss meist ein angepasstes Shaderprogramm erstellt und aktiviert werden. Da Shaderprogramme auf der Grafikkarte jederzeit ausgetauscht werden können, ist hier prinzipiell *Rapid Prototyping* einsetzbar.

Der Grafikkarte können nur komplette Shaderprogramme zur Ausführung übergeben werden. Veränderte Shaderprogramme für Volumenrendering müssen daher alle notwendigen Bestandteile enthalten. Shaderprogramme für Volumenrendering verwenden viele gemeinsame, oft genutzte Komponenten. Es ist wünschenswert, nur neue Komponenten spezifizieren zu müssen, und existierende Komponenten einfach weiterverwenden zu können. Für die Realisierung müssen zwei Aufgaben bewältigt werden: Existierender Code muss bei der Erstellung der Shaderprogramme einfach zugänglich sein und der Volumenrenderer muss Schnittstellen zur Veränderung seiner Shaderprogramme zur Verfügung stellen. Diese Arbeit untersucht Möglichkeiten und Grenzen für einen solchen flexiblen Einsatz von Shaderprogrammen im Volumenrendering.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, ein Konzept für den flexiblen Umgang mit Shaderprogrammen eines Volumenrenderers zu entwickeln. Es soll möglich sein, Shader modular zu beschreiben, um oft genutzte Komponenten wiederverwenden zu können. Shader sollen nicht mehr als Ganzes ausgetauscht werden müssen, sondern nur relevante Teile. Dabei sollen die Schnittstellen zum Renderer-Hauptprogramm flexibel sein, so dass kein Eingriff in dessen Programmcode notwendig ist. Das erstellte Konzept soll exemplarisch in die bei MEVIS RESEARCH vorhandenen Software integriert werden.

Bei der Entwicklung soll der Bezug zur Praxis hergestellt werden. Dazu wird eine Bestandsaufnahme der Anforderungen und Verwendungen des Volumenrenderings bei MEVIS erstellt. Weiterhin sollen exemplarische Anwendungen mit dem erstellten Framework umgesetzt werden.

1.3 Kontext

Die Diplomarbeit wurde mit gemeinsamer Betreuung von MEVIS RESEARCH und der Arbeitsgruppe Computergrafik der Universität Koblenz-Landau, Abteilung Koblenz, verfasst. Sie wurde am MEVIS RESEARCH Forschungszentrum für medizinische Bildverarbeitung und Visualisierung in Bremen erstellt. MEVIS RESEARCH ist eine gemeinnützige Einrichtung und forscht im Bereich der Computerunterstützung in der bildbasierten Diagnostik und Therapie. Dabei wird ein praxisorientierter Ansatz verfolgt, in dem klinisch besonders relevante Krankheitsbilder einzelner anatomischer Bereiche im Zentrum der Forschung stehen.

Neben dem Anwendungsgebiet Medizin wird die grafische Darstellung von dreidimensionalen Daten auch in Bereichen wie wissenschaftlicher Simulationen und industrieller Werkstoffforschung eingesetzt. Die Ausführungen dieser Arbeit zum Volumenrendering sind unabhängig vom Anwendungsfall.

1.4 Aufbau der Arbeit

Kapitel 2: In diesem Kapitel werden der Aufbau der Grafikhardware und die Grundlagen ihrer Programmierbarkeit vorgestellt. Es werden die für diese Arbeit relevanten Grundlagen des Volumenrenderings dargestellt.

Kapitel 3: Ziel dieses Kapitels ist es, einen Überblick über den Stand der Forschung im Bereich modularer Shaderentwicklung zu geben, und so die für die weitere Arbeit notwendigen Grundlagen zu schaffen.

Kapitel 4: Zunächst wird die bei MEVIS RESEARCH genutzte Entwicklungsumgebung MEVISLAB und der enthaltene Volumenrenderer GVR vorgestellt, die auch in dieser Arbeit genutzt werden. Im Anschluss werden Anforderungen an Volumenrendering aus der Praxis dargestellt, die in Gesprächen mit Anwendern festgestellt wurden. Für diese Arbeit relevante Schlussfolgerungen werden herausgearbeitet.

Kapitel 5: In diesem Kapitel werden die Problemstellung dieser Arbeit detailliert betrachtet und relevante Bereiche abgegrenzt. Die Eignung möglicher Lösungsansätze wird diskutiert. Daraufhin wird ein Konzept eines modularen Shaderframeworks für Volumenrendering vorgestellt.

Kapitel 6: In diesem Kapitel wird die praktische Umsetzung beschrieben. Dazu werden Implementierung und Integration in die bei MEVIS vorhandene Umgebung dargestellt.

Kapitel 7: Die Funktionsweise des erstellten Frameworks wird in diesem Kapitel gezeigt. Dazu werden ausgewählte, bisher im vorhandenen Renderer nicht integrierte Verfahren des Volumenrenderings mit seiner Hilfe umgesetzt. Die Performance der Umsetzung wird betrachtet.

Kapitel 8: Dieses abschließende Kapitel fasst die Ergebnisse der Arbeit zusammen und bewertet Stärken und Schwächen der erarbeiteten Lösung zur Modularisierung von Shadern im Kontext Volumenrendering. Es wird ein Ausblick auf mögliche weitere Entwicklungen und Anschlussrbeiten gegeben.

2 Grundlagen des GPU-basierten Volumenrenderings

Verfahren zum Volumenrendering werden seit langem zur Visualisierung dreidimensionaler Daten eingesetzt. Seit deren Entwicklung verwenden Algorithmen zur Volumendarstellung auch programmierbare Graphikhardware. Gerade die immer weiter ausgebaute Programmierbarkeit moderner GPUs macht solche Umsetzungen möglich. Kern dieser Arbeit bildet die Erstellung von Programmen für die Graphikkarte, sogenannter Shadern, zur Volumenvisualisierung.

2.1 Grundlagen der Grafikhardware

Bei *Graphics Processing Units (GPUs)* handelt es sich um spezialisierte Hardware zur Erzeugung von zweidimensionalen *Rastergrafiken* aus geometrischen Primitiven wie Punkten und Dreiecken im dreidimensionalen Raum. Rastergrafiken unterteilen einen im Allgemeinen rechteckigen Bereich in eine endliche Anzahl von Pixel genannten Bereichen, denen ein konstanter Farbwert zugeordnet wird.

GPUs werden als Bestandteil von Grafikkarten als *Koprozessoren* neben dem Hauptprozessor des Computers (*Central Processing Unit (CPU)*) eingesetzt, um die beschriebene Bilderzeugung (*Rendering*) möglichst schnell durchzuführen. Die *Grafikpipeline* unterteilt diese Bilderzeugung in einzelne, unabhängige Teilschritte. Grafikkarten setzen die Teilschritte der Pipeline direkt als Schaltkreise der Hardware um.

Für die Bilderzeugung mit Grafikkarten gibt es zum Zeitpunkt dieser Arbeit zwei verbreitete Standards: OpenGL [SWND05] und DirectX [1] [2]. Die in beiden Standards beschriebenen Pipelines unterscheiden sich nur geringfügig (vgl. [SWND05, Kapitel 1] und [Mica]). Im den folgenden Abschnitten wird die Grafikpipeline am Beispiel von OpenGL beschrieben.

2.1.1 Geometrische Primitive und Texturen

Die Grafikpipeline verarbeitet *Primitive* in Form von Vielecken. Diese *Polygone* werden durch Punkte (engl. *Vertices*; Einzahl: *Vertex*) im dreidimensionalen Raum beschrieben und durch Nachbarschaftsinformationen zu Vielecken gruppiert. Benachbarte Vertices werden durch Kanten (engl. *Edges*) verbunden. Teilen mehrere Primitive Kanten, so entstehen komplexe Oberflächen, welche vielfältige Objekte darstellen können (*Mesh*). Jedem Vertex kann neben seiner Position eine Farbe, eine *Normale* und *Texturkoordinaten* für bis zu acht Texturen als weitere Informationen mitgegeben werden.

[1] http://www.microsoft.com/directx/. Zugriff: 15.01.2008

[2] Der OpenGL-Standard wird vom *OpenGL Architecture Review Board (ARB)*, einem Zusammenschluss von Firmen aus dem Umfeld der 3D-Computergrafik festgelegt. DirectX ist ein Standard von der *Microsoft Corporation*.

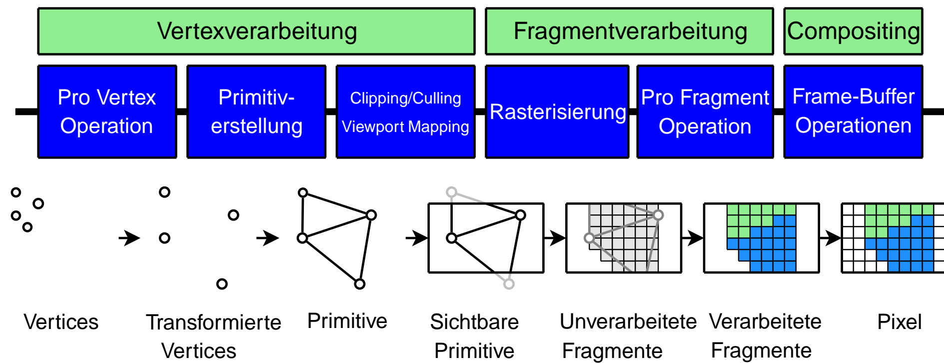


Abbildung 1: Überblick über die Grafikpipeline. Vgl. [HKRs+06]

Diese Informationen werden verwendet, um das Aussehen der Oberfläche des Primitivs zu beschreiben.

Eine Normale beschreibt die Orientierung einer Oberfläche an einem Punkt, und wird bei der Auswertung von Beleuchtungsmodellen verwendet (siehe Abschnitt 3.1). Mit Hilfe von *Texturen* können komplexe Farbverläufe auf der Oberfläche von Primitiven dargestellt werden (*Texturemapping*). Statt Eckpunktefarben wird die Farbe einer Position innerhalb einer Rastergrafik verwendet. Texturen können unterschiedliche Dimensionalität haben (1D-,2D-,3D-Texturen). Mit Hilfe der *Texturfilterung* werden Farbwerte zu Positionen berechnet, die nicht genau den Mittelpunkt eines Pixels in der Rastergrafik beschreiben. Texturemapping ist ein umfangreiches Thema, mehr Informationen gibt z. B. [SWND05].

2.1.2 Die Grafikpipeline

Die Grafikpipeline unterteilt den durchzuführenden Berechnungsprozess bei der Bilderzeugung (Abbildung 1). Ausgabedaten eines Schritts werden zu den Eingaben des nächsten Schritts. Die Berechnungen eines Teilschritts sind ausschließlich abhängig von den Eingabedaten und den globalen Einstellungen. Dies stellt sicher, dass für unterschiedliche Eingabedaten derselbe Pipelineschritt mehrmals gleichzeitig ausgeführt werden kann. Diese Parallelität ermöglicht den GPUs ihre Geschwindigkeit. Sie nutzen mehrere parallel arbeitende Hardwareeinheiten für die Bearbeitung eines Pipelineschritts. Die Schritte der Pipeline sind in mehrere Gruppen unterteilt.

2.1.2.1 Vertexverarbeitung

Die Aufgabe dieses Teils der Pipeline ist die Verarbeitung der grafischen Primitive. Dazu müssen die Vertices positioniert und den Primitiven zugeordnet werden. Mit Hilfe von *Kamera-* und *Projektionseinstellungen* werden nicht sichtbare Primitive entfernt.

Pro Vertex Operation: In diesem Schritt werden die Vertices einzeln und unabhängig von ihrer Zuordnung zu den geometrischen Primitiven verarbeitet. Zentrale Aufgabe ist es zu bestimmen, welche der gegebenen Punkte im den sichtbaren Bereich definierenden *kanonischen Einheitswürfel* liegen. Dazu werden verschiedene Transformationen und Projektionen auf einen Punkt angewendet (für weitere Informationen siehe z. B. [Shi02]). Die Projektionsart bestimmt den Tiefeneindruck im Bild. *Perspektivischen Projektion* erzeugt einen Tiefeneindruck, *orthographische Projektion* erzeugt keinerlei Tiefe im Bild.

Primitiverstellung: In diesem Schritt werden die Vertices den einzelnen Primitiven zugeordnet. Somit sind die Nachbarschaftsinformationen für den nächsten Verarbeitungsschritt vorhanden.

Clipping/Culling/Viewport Mapping: In diesem Schritt werden Primitive entfernt, die komplett außerhalb des sichtbaren Einheitsvolumens liegen (*Culling*). Primitive, die teilweise außerhalb liegen, werden an den Rändern des sichtbaren Volumens geschnitten (*Clipping*). Der *Viewport* beschreibt die Bildebene des erstellten Bildes. Beim *Viewport Mapping* werden die x und y Koordinaten der Punkte der Primitive an die Dimensionen des Viewports angepasst, der die Tiefe darstellende z-Wert bleibt unverändert.

2.1.2.2 Fragmentverarbeitung

Die Aufgabe dieses Teils der Pipeline ist die Erstellung und Verarbeitung der *Fragmente*. Jedes Fragment entspricht der Position eines Pixels im erstellten Bild, und geht ganz, teilweise oder gar nicht in dessen Endzustand mit ein.

Rasterisierung: Die Rasterisierung zerlegt die geometrischen Primitve in entsprechende Fragmente, abhängig davon, welche Pixel von ihnen überdeckt werden. Dabei werden die Position, Farbe und Texturkoordinaten der Vertices *interpoliert*, und jedem Fragment der interpolierte Wert mitgegeben. Perspektivische Verzerrungen werden ggf. bei der Interpolation korrigiert. Dies gilt insbesondere für den z-Wert der Eckpunkte.

Pro Fragment Operation: In diesem Schritt werden die Fragmente unabhängig voneinander verarbeitet. Mit Hilfe der interpolierten Vertexattribute und ggf. ausgelesenen Texturen werden die Eigenschaften des Fragments bestimmt: Seine *Farbe*, der für Verdeckungstests verwendete *Tiefenwert*, und der meist für Transparenzberechnungen verwendete *Alphawert*.

2.1.2.3 Compositing

Als Ziel des Rendervorgangs (*Rendertarget*) beschreibt der *Framebuffer* die Attribute der Pixel des erstellten Bildes. Er besteht aus mehreren einzelnen *Buffern*. Die Wichtigsten sind der *Colorbuffer*, der Farbe und Alphawert eines Pixels enthält, und der *Depthbuffer* für den den Pixeln zugeordneten Tiefenwert. Auf neueren Grafikkarten ist es möglich, den Colorbuffer durch eine Textur zu ersetzen und direkt in diese zu rendern (*Render to Texture*). Dies ist nützlich, wenn das erstellte Bild nicht direkt dargestellt, sondern ausgelesen oder in einem weiteren Rendervorgang (*Multipass-Rendering*) verwendet werden soll.

Ob und wie ein Fragment in das Aussehen eines Pixels eingeht entscheiden die *Framebuffer-Operationen*:

Depthtest Primitive können sich überdecken, und somit auch die aus ihnen generierten Fragmente. Der Tiefentest bestimmt abhängig vom Tiefenwert eines Fragments, ob es in den Framebuffer eingetragen wird. Er wird klassischerweise angewendet um sicherzustellen, dass die vordersten Teile der Primitve sichtbar sind. Ein Fragment wird nur in den Framebuffer eingetragen, wenn sein Tiefenwert kleiner ist als der sich aktuell im Depthbuffer befindende Wert. Andere Entscheidungskriterien (*Depthfunction*) sind möglich.

Alphablending Abhängig vom Alphawert eines neu eingetragenen Fragments und des entsprechenden Pixels im Framebuffer, sowie einer *Blendingfunction*, wird die neue Pixelfarbe als Mischung von alter Pixelfarbe und Fragmentfarbe bestimmt. Dies erlaubt z. B. Transparenzeffekte.

Es existieren weitere Operationen, die für diese Arbeit keine direkte Bedeutung haben (siehe z. B. [SWND05]).

2.1.3 Programmierbarkeit

Alle Stufen der im vorigen Abschnitt vorgestellten Pipeline lassen sich durch entsprechende Einstellungen steuern. Um die Funktionalität darüber hinaus zu verändern, können auf Grafikkarten einzelne Schritte der Pipeline durch eigene Programme ersetzt werden. Die Fähigkeiten der programmierbaren Einheiten einer Grafikkarte hängen davon ab, welches *Shadermodel* [Micd] sie unterstützt. Aktuell ist das Shadermodel 4.0.

Austauschbar sind die pro Vertex und pro Fragment ausgeführten Operationen. An ihre Stelle treten sogenannte *Shaderprogramme* (zur Benennung siehe Abschnitt 3.1). Das *Vertexprogramm* wird für jeden Vertex, und das *Fragmentprogramm* für jedes Fragment ausgeführt. Soll die Pipeline ihre Funktionalität beibehalten, müssen die Aufgaben der ersetzen Stufen vom

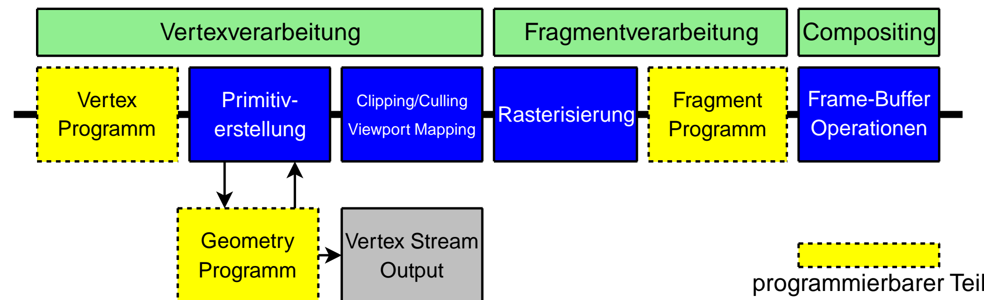


Abbildung 2: Die programmierbare Grafikpipeline. Vgl. [HKRs+06]

Programm übernommen werden. Grafikkarten ab Shadermodel 4.0 haben die Möglichkeit, für jedes erstelle Primitiv ein *Geometrieprogramm* auszuführen [Nvi]. Es ist ein Zusatz und nicht Teil der normalen Pipeline. Das Geometrieprogramm erlaubt die Erzeugung neuer Primitive. Diese fließen in die Pipeline zurück und werden dargestellt, oder können ausgelesen werden. Eine typische Anwendung ist die Verfeinerung eines Meshs direkt während dessen Darstellung. Abbildung 2 zeigt die Pipeline und ihre programmierbaren Teile.

Zur Beschreibung der Shaderprogramme wird ein auf Grafikanwendungen zugeschnittener Befehlssatz zur Verfügung gestellt [Micd]. Dieser enthält unter anderem Funktionen für Vektor- und Matrixoperationen und den Zugriff auf Texturen. Grafikkarten neuerer Shadermodels unterstützen auch Kontrollflusskonstrukte (vgl. Abschnitt 3.3.1) wie bedingte Anweisungen und Schleifen. Ab Shadermodel 4.0 werden sie vollständig unterstützt. Hängen bedingte Anweisungen nur von konstanten Eingabewerten des Shaderprogramms ab, so erlaubt dies *Static Branching*. Der Kontrollpfad durch das Programm wird vor der Ausführung des Shaders bestimmt, und nur dieser Pfad ausgeführt. Dies führt zu Geschwindigkeitsvorteilen, da die Programme sehr oft ausgeführt werden. Sind bedingte Anweisungen von Berechnungen oder Texturzugriffen abhängig, so muss diese Entscheidung zur Laufzeit ausgeführt werden (*Dynamic Branching*).

Es gibt verschiedene Möglichkeiten, die Eingaben der Shaderprogramme festzulegen. Jeder Vertex kann mit *Attributen* versehen werden, welche die Applikation dem Vertexprogramm mitteilt. *Konstanten* können in allen Programmen deklariert werden. *Varying*-Variablen sind Ausgaben des Vertexprogramms, welche bei der Rasterisierung für jedes Fragment interpoliert und dem Fragmentprogramm mitgeteilt werden. Sie repräsentieren Werte, die sich über die Oberfläche eines Primitivs verändern. *Uniforme* Variablen sind innerhalb eines Primitivs konstant und können in allen Programmen ausgelesen werden. Alle Programmarten haben Zugriff auf die Werte des OpenGL-States. Bei Karten neuerer Generationen trifft dies auch

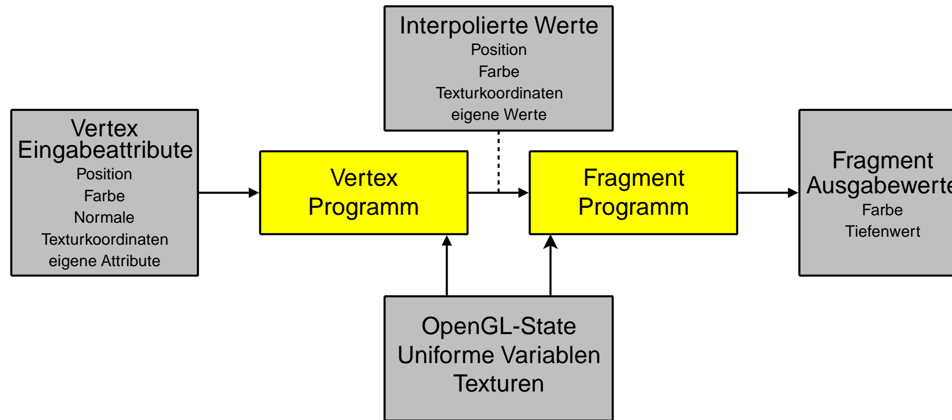


Abbildung 3: Vertex- und Fragmentprogramm sowie ihre Ein- und Ausgaben. Vgl. [MGB07]

auf Texturen zu, bei älteren Modellen haben nur die Fragmentshader Zugriff auf Texturen. Abbildung 3 zeigt den Informationsfluss für Vertex- und Fragmentprogramm.

Erstellt werden Shaderprogramme mit spezialisierten Programmiersprachen (*Shadinglanguages*). Diese erlauben es, den Befehlssatz der Grafikkarten durch eine *Highlevel API* zu nutzen. Dafür stellen sie eine *Standardbibliothek* von Datentypen und Operationen zur Verfügung. Vordefinierte Variablen (*Built-in Variables*) erlauben den Zugriff auf Werte des OpenGL-States und auf Standardattribute von Vertices und Fragmenten. Verbreitete Sprachen sind die High Level Shading Language (HLSL) für DirectX [Micb], die OpenGL Shading Language (GLSL) für OpenGL [Ros06], sowie das für beide Bibliotheken einsetzbare C for Graphics (Cg) von Nvidia [FK03].

Für diese Arbeit wurde GLSL als Hochsprache für Shader gewählt. Umsetzung und Beispiele ab Kapitel 5 verwenden diese Sprache. Sie verwendet eine C-ähnliche Syntax. Jedes Vertex- und Fragmentprogramm muss eine `main`-Methode besitzen, die bei seiner Ausführung aufgerufen wird. Die Schnittstelle eines Programms wird über Variablen (uniform, varying, ...) definiert, die außerhalb der `main`-Methode stehen. Datenstrukturen und weitere Funktionen können ebenfalls vor der Hauptfunktion angegeben werden. In dieser Arbeit wird der Bereich als `main`-Methode als *Rumpf* bzw. *Body* des Shaders bezeichnet. Alle anderen Teile stehen vor dieser Methode, und werden hier als *Header* bezeichnet. Programmcode 5 in Anhang A zeigt ein Beispiel einfacher Shaderprogramme in GLSL.

Ausgeführt werden die Programme von Teilen der GPU. GPU-Generationen bis einschließlich Shadermodel 3 hatten spezialisierte, parallel arbeitende *Vertex*- und *Fragmentprozessoren*. Mit den *Unified Shadern* wurde

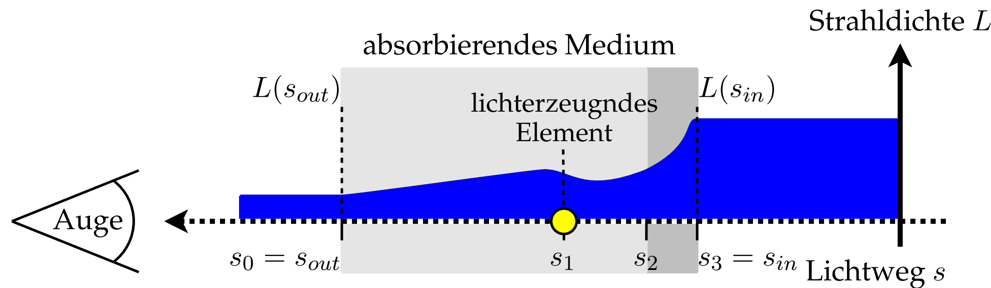


Abbildung 4: Verlauf der Strahldichtefunktion entlang eines Lichtwegs durch ein absorbierendes Medium. Der Strahl tritt bei s_{in} in das Medium ein und bei s_{out} aus. Das Medium absorbiert zwischen s_2 und s_3 stärker als zwischen s_0 und s_2. Bei s_1 enthält es ein lichterzeugendes Element.

das Konzept spezialisierter Hardwareeinheiten beim Shadermodel 4 aufgegeben. Es gibt nur noch einen Prozessortyp, der Vertex-, Fragment- und Geometrieprogramme ausführen kann. Die Prozessoren können damit abhängig vom Arbeitsaufkommen dynamisch auf die unterschiedlichen Aufgaben verteilt werden.

2.2 Volumenrendering

Volumenrendering ist der Prozess der Erstellung eines zweidimensionalen Bilds aus dreidimensionalen Bilddaten (*Volumen*). Es wird berechnet, wie das Volumen Licht auf dem Weg zum Betrachter verändert, bzw. was der Betrachter beim Blick durch das Volumen wahrnimmt. Mit Hilfe moderner Grafikhardware ist es möglich, dies in Echtzeit (zum Begriff der Echtzeit siehe Abschnitt 3.1) durchzuführen. Eine umfangreiche Einführung in das Thema echtzeitfähigen Volumenrenderings gibt [HKRs+06]. Abbildung 27 auf Seite 48 zeigt ein Beispiel für das Ergebnis eines Volumenrenderings.

2.2.1 Theoretische Grundlagen

Im Zentrum jedes Renderingalgorithmus steht die Beschreibung von Licht. Es wird berechnet, wieviel Licht (*Helligkeit*) welcher Wellenlänge (*Farbe*) den Betrachter erreicht. Dazu werden von einem virtuellen Auge aus *Lichteffekte* entlang einer Schar von *Sichtstrahlen* nachvollzogen. Beim Volumenrendering wird berücksichtigt, wie Licht auf dem Weg durch ein Medium (*Participating Medium*) verändert wird. Oberflächenrendering beschreibt hingegen das Verhalten des Lichts nur an den Grenzen zwischen zwei verschiedenen Medien. Folgende Effekte können auftreten:

Lichtschwächende Effekte Licht wird entlang des Sichtstrahls vom Medium abgeschwächt. Licht kann „geschluckt“ (*Absorbtion*) oder in eine

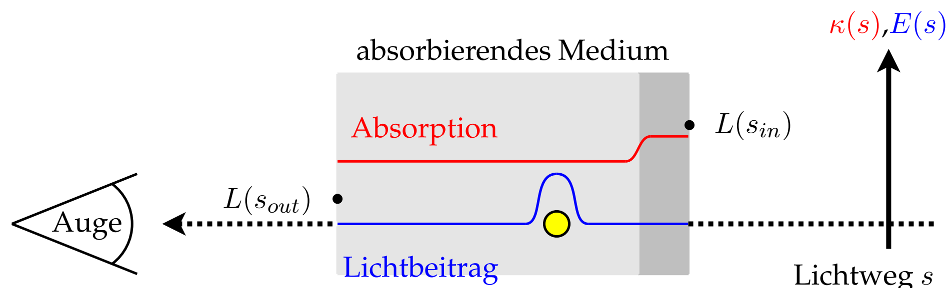


Abbildung 5: Absorptionsfunktion $\kappa(s)$ und Lichtbeitragsfunktion $E(s)$, wie sie im Volumenrendering-Integral verwendet werden.

andere Richtung umgelenkt werden (*Outscattering*).

Lichtverstärkende Effekte Licht wird entlang des Sichtstrahls durch das Medium verstärkt. Licht kann aus einer anderen Richtung in Richtung des Auges umgelenkt werden (*Inscattering*), oder das Medium strahlt selbst Licht aus (*Emission*).

Die Menge des Lichts wird dabei durch die *Strahldichte* L dargestellt. Sie beschreibt die Strahlungsmenge, die in einer Zeiteinheit von einer Fläche in eine bestimmte Richtung abgegeben wird. Ein Überblick über die physikalischen Grundlagen des Lichts bieten [HKRs+06, PH04].

Ziel des Volumenrenderings ist es, die entlang eines Lichtwegs aus dem Medium austretende Strahldichte $L(s_{out})$ zu berechnen (Abbildung 4). Das *Volumenrendering-Integral* beschreibt ein Modell für diese Berechnung. Sie besteht aus zwei Teilen L^1_{out} und L^2_{out}:

$$L(s_{out}) = \underbrace{L(s_{in}) \cdot T(s_{in}, s_{out})}_{L^1_{out}} + \underbrace{\int_{s_{in}}^{s_{out}} E(s) \cdot T(s, s_{out}) ds}_{L^2_{out}} \tag{1}$$

Formelquelle: [HKRs+06]

Der Lichtweg wird mit Längenparameter s beschrieben. Bei s_{in} handelt es sich um den Eintrittspunkt in das Medium, bei s_{out} um den Austrittspunkt. L^1_{out} beschreibt den Anteil des eintreffenden Lichts, der beim Austritt noch vorhanden ist. Die *Transparenzfunktion* $T(s_0, s_1)$ beschreibt den zurückbleibenden Lichtanteil nach dem Passieren der Strecke zwischen t_0 und t_1. L^2_{out} beschreibt, wieviel Licht vom Medium noch hinzugefügt wird. Es wird über den Strahldichtebeitrag jedes einzelnen Punkts auf dem Weg integriert. Für jeden Punkt wird betrachtet:

1. $E(s)$: Wieviel Licht wird dem Lichtweg an dieser Stelle hinzugefügt?

2. $T(s, s_{out})$: Welcher Anteil dieses Lichts geht auf dem verbleibenden Weg zum Austrittspunkt verloren?

Die Lichtbeitragsfunktion $E(s)$ wird im einfachsten Fall nur über die Emission des Punkts auf dem Lichtweg modelliert. Der Einfluss von Lichtquellen in der Umgebung kann ebenfalls berücksichtigt werden (siehe Abschnitt 2.2.2.4). Dies entspricht Inscattering von Licht aus Lichtquellen außerhalb des Lichtwegs. Die Transparenzfunktion wird über ein *Absorptionsmodell* realisiert:

$$T(s_0, s_1) = e^{-\int_{s_0}^{s_1} \kappa(s)ds} \tag{2}$$

Formelquelle: [HKRs+06]

Die Funktion $\kappa(s)$ beschreibt für jeden Punkt des Wegs den *Absorptionskoeffizient*, der den Anteil des absorbierten Lichts beschreibt. Die Transparenzfunktion bestimmt sich als Integral. Abbildung 5 veranschaulicht, über welche Funktionen das Volumenrendering-Integral integriert.

Um das Volumenrendering-Integral im Computer lösen zu können, wird es durch eine Approximation mit Hilfe einer stückweise konstanten Funktion ersetzt. Zunächst wird der Integrationsbereich in mehrere Intervalle aufgeteilt, und das Volumenrendering-Integral für jedes Intervall separat gelöst:

$$L(s_i) = L(s_{i-1}) \underbrace{T(s_{i-1}, s_i)}_{T_i} + \underbrace{\int_{s_{i-1}}^{s_i} L(s)T(s, s_i)ds}_{C_i} \tag{3}$$

Formelquelle: [HKRs+06]

T_i bezeichnet den Transparenzbeitrag und C_i den Beitrag an Lichtmenge des Intervalls. Jedes Teilintegral greift auf das Ergebnis des vorher gelegenen Intervalls zurück. Das erste Integral nach dem Eintrittspunkt verwendet stattdessen die einfallende Strahldichte $L(s_{in})$. Damit lässt sich das Volumenrendering-Integral als Summe schreiben:

$$L(s_{out}) = L(s_n) = L(s_{n-1})T_n + c_n = (L(s_{n-2})T_{n-1} + c_{n-1})T_n + c_n = \cdots$$

$$L(s_{out}) = \sum_{i=0}^{n} C_i \prod_{j=i+1}^{n} T_j, \qquad \text{mit } C_0 = L(s_{in}) \tag{4}$$

Formelquelle: [HKRs+06]

Die Anzahl n der Schritte der Summe ist dabei abhängig von der Breite $\Delta x = |s_i - s_{i-1}|$ der Intervalle. Im letzten Schritt werden die Teilintegrale T_i und c_i durch konstante Werte vom Rand des entsprechenden Intervalls angenähert, und durch die Länge des Intervalls in der Summe gewichtet:

$$T_i \approx e^{-\kappa(s_i)\Delta x}$$
$$C_i \approx I(s_i)\Delta x \tag{5}$$

Formelquelle: [HKRs+06]

Je kleiner die Intervallbreite, desto genauer ist die Approximierung des Volumenrendering-Integrals.

2.2.2 Die Volumenrendering-Pipeline

Verfahren zur Umsetzung des optischen Modells des Volumenrenderings können in mehrere Schritte aufgeteilt werden. Die *Volumenrendering-Pipeline* beschreibt typischerweise vorhandene Teile. Die Schritte übernehmen sowohl die Zuordnung optischer Eigenschaften zu den Volumenwerten, als auch die Berechnung des diskretisierten Volumenrendering-Integrals. Abbildung 6 zeigt die Pipeline.

2.2.2.1 Volumendaten

Das darzustellende Volumen wird mathematisch als *dreidimensionales Feld* f beschrieben. Seine Werte sind meist skalar:

$$f : \mathbb{R}^3 \rightarrow \mathbb{R} \tag{6}$$

Formelquelle: [HKRs+06]

Punkten im dreidimensionalen Raum wird ein einfacher Wert zugeordnet. Zur Darstellung von Gasen und Wolken repräsentiert *photorealistisches Volumenrendering* diese Funktion über Simulationen oder prozedurale Modelle, die an jedem Punkt im Raum ausgewertet werden können. In der Praxis liegen oft nur Daten einer Messung vor, die ein Volumen an diskreten Stellen abtastet. Typische Beispiele für solche Datensätze sind Aufnahmen

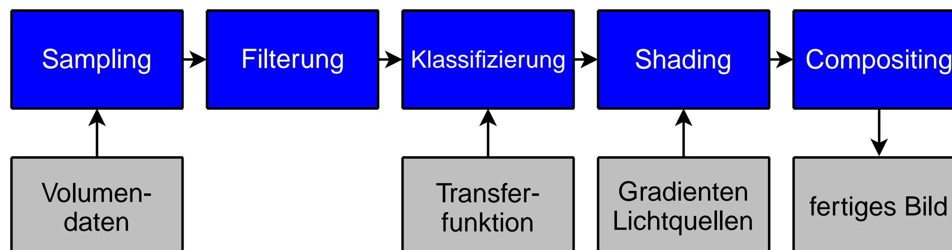


Abbildung 6: Die Volumenrendering-Pipeline. Die Schritte der Pipeline sind oben dargestellt, verwendete und erstellte Daten unten. Vgl. [MGB07]

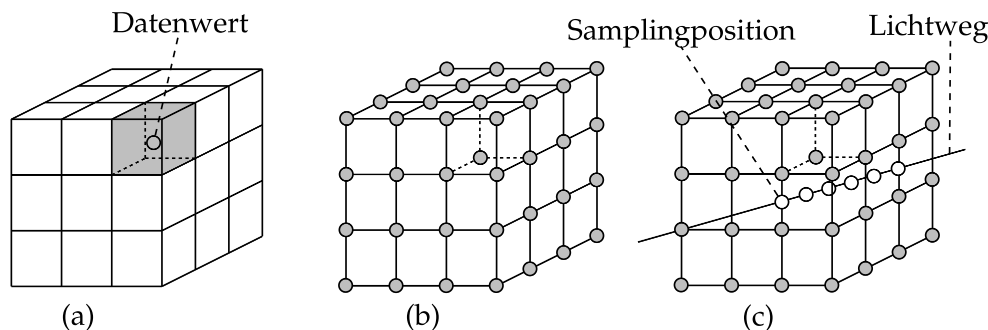


Abbildung 7: Geometrische Modelle für diskrete Volumendaten. (a) Voxelvolumen (b) Regelmäßiges Gitter (c) Sampling entlang eines Lichtwegs durch ein regelmäßiges Gitter. Vgl. [MGB07]

der medizinischen Bildgebung. Computertomographie (CT) und Magnetresonanztomographie (MRT) ordnen Punkten im aufgenommenen Volumen Dichtewerte bzw. Signalintensitäten zu. *Direktes Volumenrendering* beschäftigt sich mit der Darstellung dieser Volumenrepräsentation, und ist der Schwerpunkt dieser Arbeit.

Diskrete Volumendaten werden als dreidimensionales Array von Werten gespeichert. Die Zuordnung von Wert zu Position geschieht mit Hilfe von *geometrischen Modellen*. Ein *Voxelvolumen* ordnet jeden Wert einem kleinen Teilwürfel einer größeren Region zu (siehe Abbildung 7a). Ein *regelmäßiges Gitter* ordnet die Werte Knotenpunkten innerhalb des Volumens zu, die gleichmäßig verteilt sind (siehe Abbildung 7b). Die so gebildeten rechteckigen Zellen haben selbst keinen zugeordneten Wert. Eine Übersicht über diese und andere Modelle mit flexibleren Zelltypen gibt [SM00, HKRs+06]. Im Folgenden wird ein regelmäßiges als geometrisches Modell angenommen.

2.2.2.2 Sampling und Rekonstruktion

Beim *Sampling* (dt. Abtastung) werden innerhalb des dargestellten Volumens Lichtwege ausgewählt. Darauf werden *Samplingpositionen* ausgewählt, welche die Intervallgrenzen des diskretisierten Volumenrendering-Integrals (Gleichung 4) bestimmen (Abbildung 7c). Je höher die *Samplingrate*, desto mehr Samplingpositionen werden ausgewählt. Für jede Position wird der entsprechende Wert der Volumenrepräsentation bestimmt. Konkrete Ansätze zum Sampling stellt Abschnitt 2.2.3.2 vor.

Beim Sampling des Lichtwegs wird auf Positionen des Originalvolumens zugegriffen, für die kein diskreter Wert vorhanden ist. Für diese Positionen muss auf Basis der vorhandenen Informationen ein Wert approximiert werden. Diesen Vorgang nennt man *Filterung* oder *Rekonstruktion*. Ziel ist es, die ursprüngliche Funktion bestmöglich wiederherzustellen. Die

Qualität der Rekonstruktion wird begrenzt durch die Genauigkeit der Aufnahme des realen Volumens. Das Samplingtheorem von *Nyquist und Shannon* beschreibt, dass bei hohen Frequenzen im Wertebereich eine hohe Aufnahmefrequenz für die korrekte Rekonstruktion notwendig ist. Die Abtastfrequenz entspricht der Auflösung des regelmäßigen Gitters im Vergleich zum aufgenommenen Objekt. Eine Einführung zur Samplingtheorem von Nyquist und Shannon gibt [Gri05], eine Betrachtung im Hinblick auf Volumenrendering gibt [HKRs+06].

Die Qualität der Rekonstruktion wird innerhalb der gegebenen Grenzen von der verwendeten *Rekonstruktionsfunktion* bestimmt. Sie bestimmt die Anzahl und Auswahl vorhandener Werte, die zur Bestimmung des Werts einer beliebigen Position hinzugezogen werden und wie diese verrechnet werden. Im Kontext Volumenrendering wird oft die schnell berechenbare *trilineare Interpolation* eingesetzt [HKRs+06]. Aufwändigere Filterungsfunktionen erzielen bessere visuelle Ergebnisse auf Kosten erhöhten Rechenaufwands.

2.2.2.3 Klassifizierung

Skalare Volumendaten beschreiben ein Volumen durch Dichte-, Anregungs- oder andere Werte an einer Position. Die für die Darstellung notwendigen Informationen des Lichtbeitrags und der Transparenz sind im Datensatz nicht gegeben. Unter *Klassifizierung* versteht man die Bestimmung dieser optischen Eigenschaften für ein Volumen. Für das Anwendungsfeld interessante Stellen (*Region of Interest*) sollen im Volumen sichtbar gemacht werden.

Die Klassifizierung geschieht mit Hilfe einer *Transferfunktion*. Sie weißt jedem Wert des Volumens einen Transparenzwert T und einen Lichtbeitrag C zu:

$$t : \mathbb{R} \rightarrow (C, T) \tag{7}$$

Formel nach: [HKRs+06]

Die Festlegung der Transferfunktion ist eine meist manuelle gelöste Aufgabe. Der Lichtbeitrag wird im Allgemeinen als *Farbwert* interpretiert und auch im Folgenden so bezeichnet. In der Praxis ist der Wertebereich der Volumendaten diskret. Transferfunktionen werden daher meist als *Lookuptable* realisiert, die jedem Datenwert die entsprechenden optischen Eigenschaften zuordnet. Die Farbe wird als RGB-Wert festgelegt, und die Transparenz als Wert zwischen 0 und 1. Beide Werte können in einem RGBA-Wert kombiniert werden. Anstelle der Transparenz kann auch der reziproke Wert der *Opazität* verwendet werden, welcher die Lichtundurchlässigkeit beschreibt.

Die Qualität der Darstellung hängt davon ab, wann die Transferfunktion angewendet wird. Abbildung 8 veranschaulicht die im Folgenden be-

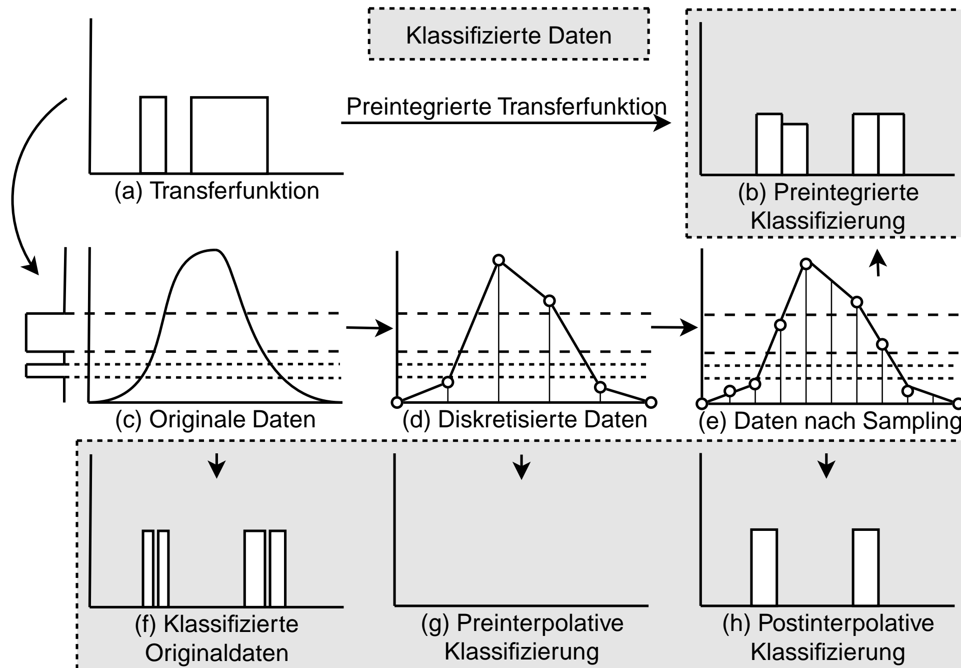


Abbildung 8: Verschiedene Stellen für die Anwendung der Transferfunktion und ihr Einfluss auf die Qualität der Klassifizierung. (a) Transferfunktion (b) Preintegrative Interpolation (c)-(e) Weg von Originaldaten zu Samplingpositionen (f) Ideale Klassifizierung. (g) Preinterpolative Klassifizierung (h) Postinterpolative Klassifizierung. Vgl. [HKRs+06]

schriebenen Zusammenhänge. Die *preinterpolative* Klassifizierung wendet die Transferfunktion auf die Werte des Datenvolumens an. Sampling und Rekonstruktion werden in diesem Fall auf die Farb- und Transparenzwerte angewendet, und nicht auf die Werte des Datenvolumens. Die *postinterpolative* Klassifizierung wendet die Transferfunktion nach Sampling und Rekonstruktion an. Für jeden interpolierten Wert des Datenvolumens wird die Transferfunktion ausgewertet.

Die postinterpolative Klassifizierung erzielt die besseren visuellen Ergebnisse. Sie approximiert die optimale Klassifizierung (Anwendung der Transferfunktion auf die Originaldaten) besser. Der Grund liegt in der Genauigkeit, mit der die Transferfunktion abgetastet wird. Die Transferfunktion kann selbst hohe Frequenzen im Wertebereich enthalten. In der Praxis enthält die Transferfunktion oft hohe Frequenzen, um aneinandergrenzende Bereiche unterschiedlicher Wichtigkeit zu unterscheiden. Für eine hochwertige Darstellung ist es in diesem Fall notwendig, die Transferfunktion

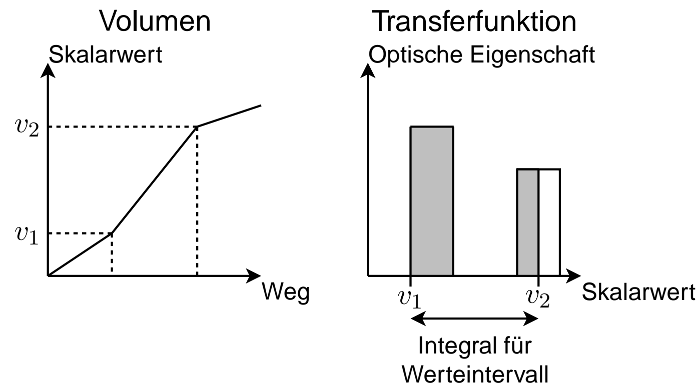


Abbildung 9: Vorberechenbarkeit der Anwendung der Transferfunktion. Vgl. [HKRs+06]

mit hoher Frequenz abzutasten[3]. Um bei der postinterpolativen Klassifizierung die Abtastfrequenz der Transferfunktion zu erhöhen, muss die Samplingrate für das Volumen erhöht werden. Dies führt zu Einbußen in der Rechenperformance.

Preintegriertes Volumenrendering [EKE01] erlaubt es, die Abtastung der Transferfunktion zu verfeinern, ohne die Samplingrate des Volumenrenderings selbst zu verändern. Dazu wird das Sampling des Volumens von der Auswertung der Transferfunktion getrennt. Jeweils zwei Samplingpositionen bilden ein Intervall des Volumenrendering-Integrals. Postinterpolative Klassifikation wertet die Transferfunktion nur für diese Positionen aus. Preintegriertes Volumenrendering integriert für alle möglichen Kombinationen diskreter Randwerte über den Verlauf der Transferfunktion, und bestimmt so die optischen Eigenschaften des Intervalls (siehe Abbildung 9). Dabei kann die Transferfunktion mit hoher Frequenz abgetastet werden. Die Verarbeitung der Transferfunktion geschieht in einem Vorverarbeitungsschritt. Dies ist ein Nachteil des preintegrierten Volumenrenderings, denn bei Veränderung der Transferfunktion muss der Vorverarbeitungsschritt wiederholt werden. Beim Rendering selbst wird anhand der Intervallgrenzwerte auf die vorberechneten optischen Eigenschaften zurückgegriffen. Abbildung 10 zeigt einen Vergleich der vorgestellten Klassifikationsarten.

2.2.2.4 Shading

Beim *Shading* wird die Beleuchtung der Volumenelemente durch Lichtquellen außerhalb des Volumens berechnet. Der Farbwert der einzelnen Positionen $\boldsymbol{p}$ wird durch die Anwendung von Beleuchtungsmodellen verändert (zu Shading und Beleuchtungsmodellen siehe auch Abschnitt 3.1). Beim

[3] Aufgrund des Samplingtheorems von Nyquist und Shannon.

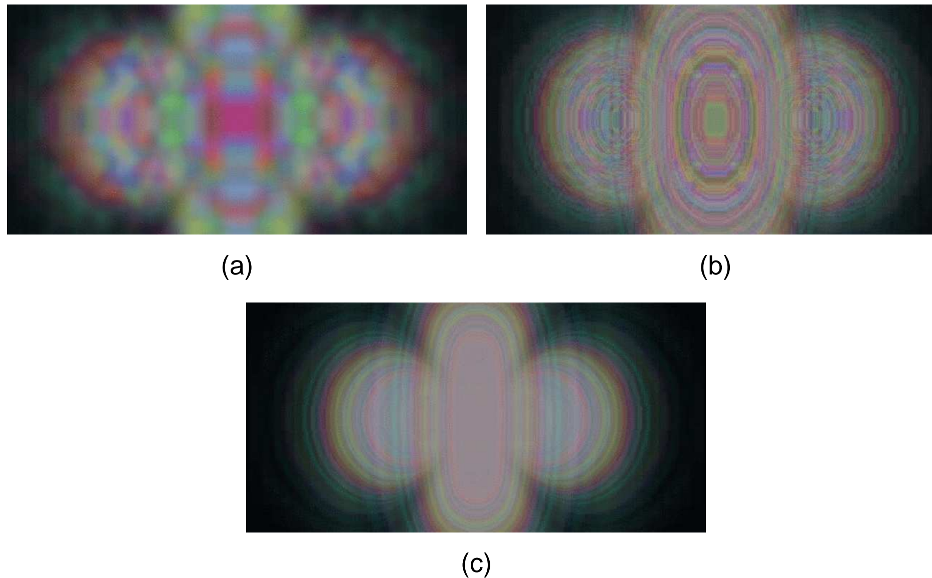


Abbildung 10: Qualität der Darstellung hoher Frequenzen verschiedener Klassifikationsarten. (a) preinterpolativ (b) postinterpolativ (c) preintegriert. Bildquelle: [EE02]

Oberflächenrendering bestimmt die Normale die Orientierung einer Oberfläche an einem Punkt. Sie steht im rechten Winkel zur Oberfläche und wird für die Entscheidung verwendet, in welchem Winkel das Licht auf die Oberfläche trifft. Für Volumenrendering wird angenommen, dass Licht von Oberflächen aus Punkten gleichen Intensitätswerts reflektiert wird (*Isooberflächen*). Der Gradient des skalaren Felds $f(\boldsymbol{x})$ ist definiert als:

$$\nabla f(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial f(\boldsymbol{x})}{\partial x} \\ \frac{\partial f(\boldsymbol{x})}{\partial y} \\ \frac{\partial f(\boldsymbol{x})}{\partial z} \end{pmatrix} \tag{8}$$

Formelquelle: [HKRs+06]

Gehört der Punkt $\boldsymbol{x}$ zu einer Isooberfläche, so ist der Gradient an dieser Stelle senkrecht zu dieser Oberfläche. Er kann daher äquivalent zu einer Normale eingesetzt werden. Für diskrete Volumendaten kann der Gradient über *zentrale Differenzen* geschätzt werden. Dabei wird der Wert der Ableitung an einem Punkt mit Hilfe der Differenz benachbarter Funktionswerte geschätzt. Damit ergibt sich für die Gradienten des diskreten Volumens:

$$\nabla f(\boldsymbol{x}) \approx \frac{1}{2h} \begin{pmatrix} f(x+h,y,z) - f(x-h,y,z) \\ f(x,y+h,z) - f(x,y-h,z) \\ f(x,y,z+h) - f(x,y,z-h) \end{pmatrix} \tag{9}$$

Formelquelle: [HKRs+06]

Diese Approximation ist schnell zu berechnen. Aufwändigere Verfahren erzielen genauere Ergebnisse und werden z. B. in [HKRs+06] vorgestellt. Gradienten in uniformen Bereichen haben oft die Länge null. Sie sind dann nicht für die Beleuchtungsberechnung verwendbar.

2.2.2.5 Compositing

Unter *Compositing* wird die iterative Berechnung von Gleichung 4 verstanden. Der gesuchte Wert $L(s_{out})$ wird dabei repräsentiert durch einen einzigen RGBA-Wert (*Destination*). Er setzt sich zusammen aus der RGB-Farbe C_{dst} und der Opazität α_{dst}. Die Iteration geht über alle Samples entlang des Sichtstrahls. Dabei wird der Zielwert mit dem Farbwert C_{src} und Opazitätswert α_{src} jedes Samples verrechnet. Für das Compositing ist von Bedeutung, ob die Farbwerte mit der Opazität gewichtet gespeichert werden (*assoziierte Farben*) oder nicht. In diesem Abschnitt werden die Formeln für nicht assoziierte Farben kurz vorgestellt. Einen umfangreichen Überblick, auch zu assoziierten Farben, gibt [HKRs+06].

Beim *Front-to-Back Compositing* wird der Lichtweg vom Auge durch das Volumen verfolgt. Begonnen wird mit dem Samplingpunkt, der am nähsten zur Kamera ist. Beim Weg durch das Volumen wird die Opazität der passierten Samplingwerte aufgesammelt. Je größer die schon aufgesammelte Opazität, desto weniger geht ein Sample ins Endergebnis mit ein. Das Ergebnis jedes Iterationsschritts errechnet sich durch:

$$\begin{aligned} C_{dst} &\leftarrow C_{dst} + (1 - \alpha_{dst})\alpha_{src} C_{src} \\ \alpha_{dst} &\leftarrow \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src} \end{aligned} \tag{10}$$

Formelquelle: [HKRs+06]

Back-to-Front Compositing geht den umgekehrten Weg, und fängt mit dem Sample an, welches am weitesten von der Kamera entfernt ist. Beim Weg durch das Volumen wird berechnet, wie stark eine Position die vorherigen Positionen verdeckt. Das Ergebnis jedes Iterationsschritts errechnet sich durch:

$$C_{dst} \leftarrow (1 - \alpha_{src}) C_{dst} + \alpha_{src} C_{src} \tag{11}$$

Formelquelle: [HKRs+06]

Eine Anpassung des Opazitätswerts α_{dst} ist nicht notwendig, da der Wert nicht in die Berechnung einfließt.

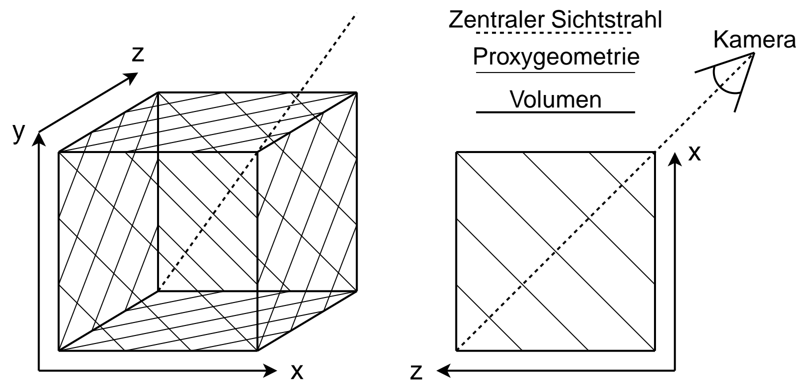


Abbildung 11: Erstellung der Proxygeometrie für View-aligned Slicing.

Front-to-Back und Back-to-Front Compositing führen die Berechung des diskreten Volumenrendering Intergrals aus. Eine andere Compositing-Variante ist ebenfalls gebräuchlich. Bei der *Maximum Intensity Projection (MIP)* wird jeweils der hellere Wert weiterverwendet:

$$C_{dst} \leftarrow \max(C_{dst}, C_{src}) \tag{12}$$

Formelquelle: [HKRs+06]

Der hellste Wert bleibt am Ende übrig. MIP ist keine Approximation des Volumenrendering-Integrals. Das Verfahren findet oft im Feld der medizinischen Visualisierung Anwendung, z. B. zur Gefäßdarstellung.

2.2.3 Umsetzung auf der GPU

Moderne programmierbare Grafikkarten können zur Volumenvisualisierung in Echtzeit eingesetzt werden. Dabei ist entscheidend, dass Veränderungen an den Darstellungsparametern, wie Betrachterstandpunkt oder Transferfunktion, mit minimal möglicher Verzögerung dargestellt werden. Anwender können somit interaktiv mit dem Volumendatensatz arbeiten.

2.2.3.1 Volumendaten auf der GPU

Die diskretisierten Volumendaten werden als dreidimensionale Textur auf die Grafikkarte geladen. Die Texturverarbeitung der GPU ist spezialisiert auf RGBA-wertige Texturen. Das Volumen wird in dieser Form abgelegt. Handelt es sich um skalare Volumenwerte, so wird nur ein Kanal der Textur verwendet.

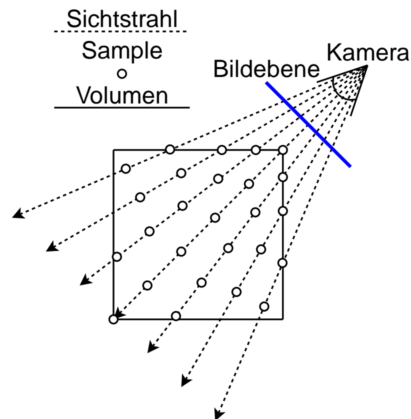


Abbildung 12: Raycasting verfolgt Sichtstrahlen durch die Bildebene und das Volumen. Innerhalb des Volumens werden Samples genommen.

2.2.3.2 Sampling

Es gibt zwei verbreitete Verfahren, die auf der Grafikkarte zur Bestimmung der Samplingpositionen genutzt werden: Raycasting und Textureslicing (siehe [HKRs+06, Seite 27ff]). Für beide Ansätze existieren Beschleunigungsverfahren, welche die große Menge an notwendigen Berechnungen einzuschränken versuchen.

Beim *Textureslicing* wird eine *Proxygeometrie* aus planaren Polygonen erzeugt. Deren Schnittpunkte mit dem Volumen stellen die Sampingpositionen dar. Der Geometrie wird dabei geeignet positioniert und mit Texturkoordinaten versehen zur Grafikkarte gesendet. Bei der Rasterisierung entsprechen die interpolierten Texturkoordinaten dann der Position im dreidimensionalen Volumen. Die Proxygeometrie wird durch *View-aligned Slicing* erstellt. Zur zentralen Blickrichtung der Kamera senkrechte Ebenen werden mit dem Volumenwürfel geschnitten. Betrachtet werden alle Ebenen, die das Volumen schneiden und um eins Vielfaches der Samplingdistanz von der Kamera entfernt sind. Abbildung 11 zeigt das Prinzip von Slicing mit Ebenen. Eine Möglichkeit zur Beschleunigung Slicing-basierten Volumenrenderings ist die Verwendung eines *Octrees* für die Texturdaten [LHJ99] (siehe auch Abschnitt 4.1.2).

Raycasting verfolgt explizit Sichtstrahlen von der Kamera durch die Bildebene und das Volumen. Auf diesen Strahlen werden Samplingpositionen ausgewählt (Abbildung 12). Beim *Raysetup* werden die Sichtstrahlen erzeugt. Dies geschieht mit Hilfe einer Proxygeometrie, die meist dem Äußeren des Volumens entspricht. In der *Raycasting-Loop* werden die einzelnen Schritte der Volumenrendering-Pipeline für jeden Strahl und jede Samplingposition auf dem Strahl durchgegangen. Grafikkarten neuerer Shadermodels erlauben die Verwendung von Schleifen in Shaderprogrammen.

Die Raycasting-Loop wird dann als Schleife im Fragmentprogramm realisiert (*Singlepass GPU Raycasting*). *Multipass GPU Raycasting* realisiert die Verfolgung von Strahlen durch mehrere Rendervorgänge und ohne Schleifen [RGW+03, KW03]. Für Grafikkarten mit der Fähigkeit Schleifen zu verarbeiten sind diese Ansätze unnötig komplex.

Beschleunigungsverfahren für Raycasting versuchen, die Anzahl der Schleifeniterationen zu minimieren. *Early Raytermination* bricht die Verfolgung eines Strahls ab, wenn die akkumulierte Opazität nahe an einhundert Prozent ist. Weitere Samples sind komplett verdeckt. *Adaptive Sampling* verwendet eine dynamische Schrittgröße, die abhängig vom Inhalt des Volumens gewählt wird. Leere oder uniforme Bereiche können so mit großen Schrittweiten passiert werden. *Empty Space Skipping* besimmt in einem Vorverarbeitungsschritt, inwieweit ein Sichtstrahl sichtbare Voxel trifft. Nur diese Strahlen werden mit hoher Samplingrate verarbeitet. [HKRs+06] gibt einen detaillierten Überblick über Beschleunigungsverfahren für Raycasting.

Bei variablen Samplingraten verändert sich der Abstand zwischen benachbarten Samples. In diesem Fall muss die Opazität der Samples angepasst werden, da sein Gewicht von der Schrittweite abhängt (Gleichung 5). Variable Abstände zwischen Samples treten auch beim Textureslicing auf, wenn perspektivische Projektion verwendet wird. Die Sichtstrahlen schneiden die parallelen Polygone der Proxygeometrie in unterschiedlichen Winkeln. Der Abstand der Schnittpunkte hängt vom Winkel ab.

2.2.3.3 Schritte der Pipeline

Im Folgenden wird die Umsetzung der übrigen Schritte der Volumenrendering-Pipeline auf der GPU dargestellt.

Sampling: Das Sampling auf der GPU wurde in Abschnitt 2.2.3.2 behandelt.

Filterung: Für die Rekonstruktion kann die trilineare Filterung von 3D-Texturen der Grafikkarte weiterverwendet werden. Aufwändigere Filterungsalgorithmen können im Fragmentprogramm implementiert werden, da es Zugriff auf alle Elemente der 3D-Textur hat.

Klassifizierung: Die Lookuptable der Transferfunktion wird als Textur realisiert. Der Wertbereich der Textur entspricht den optischen Eigenschaften. Zur Auswertung der Transferfunktion wird die Volumentextur ausgelesen und die Lookuptextur mit dem erhaltenen Wert indexiert. Die Transferfunktion wird im einfachsten Fall ausgewertet, indem der Wert der Volumentextur ausgelesen und die Lookuptextur mit ihm indexiert wird (*Dependant Textureread*). Die Lookuptextur ist in diesem Fall eine eindimensionale RGBA-Textur, deren Größe der Wertemenge des diskreten Volumens entspricht. Opazität und Farbe

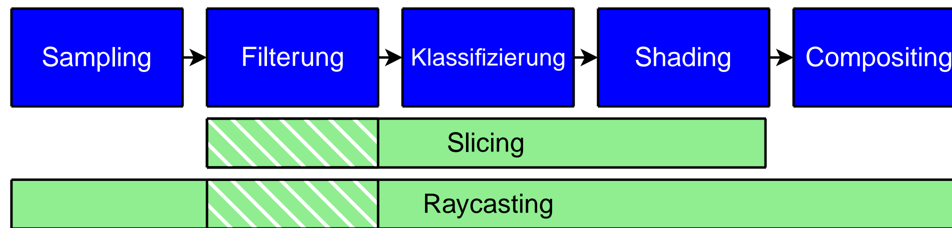


Abbildung 13: Aufgaben des Fragmentprogramms beim Volumenrendering. Die Filterung ist gestrichelt dargestellt, da hier auch auf die Hardwarefilterung der GPU zurückgegriffen werden kann.

werden über einen RGBA-Wert festgelegt. Komplexere Lookuptexturen mit mehr Dimensionen können Transferfunktionen kodieren, die mehr als nur den Intensitätswert des Volumens für die Bestimmung der optischen Eigenschaften heranziehen.

Shading: Die Beleuchtung eines Volumenelements wird im Fragmentprogramm durchgeführt. Dazu werden entsprechende Beleuchtungsmodelle im Shadercode implementiert. Die als Normalenersatz benötigten Gradienten können dabei im Fragmentprogramm berechnet werden (*On-the-fly Gradients*). Reicht die Leistungefähigkeit der Fragmentprozessoren nicht aus, so können die Gradienten in einem Vorverarbeitungsschritt erzeugt werden (*Precomputed Gradients*). Sie werden als weitere 3D-Textur auf die Grafikkarte geladen.

Compositing: Raycasting implementiert Compositing innerhalb der Raycasting-Loop im Fragmentshader. Slicing setzt Compositing mit Alphablending im Framebuffer um. Durch den Einsatz entsprechende Blending-Funktionen können die vorgestellten Compositing-Verfahren realisiert werden.

2.2.3.4 Shaderprogramme

Shaderprogramme werden für GPU-basiertes Volumenrendering intensiv eingesetzt. Der Hauptteil der Arbeit fällt dabei im Fragmentprogramm an. Abbildung 13 zeigt die Aufgaben des Fragmentprogramms. Das Vertexprogramm übernimmt die Weiterreichung der entsprechenden Koordinateninformationen. Es kann darüber hinaus auch für die Positionierung der Proxygeometrie verwendet werden. Die neuen Geometryshader haben zur Zeit noch keine Bedeutung im Kontext des Volumenrenderings. Ein Einsatz für die Erzeugung von Proxygeometrie ist jedoch denkbar. Programmcode 5 in Anhang A zeigt beispielhafte einfache Vertex- und Fragementprogramme für Volumenrendering auf Basis von Textureslicing (mit 1D-Lookuptabelle und Phong-Beleuchtung).

3 Shadererstellung

Im Folgenden werden Ansätze zur *Metaprogrammierung* von Shaderprogrammen vorgestellt. Sie stellen erweiterte Strukturierungsmöglichkeiten wie Objektorientierung zur Verfügung, die in der Syntax der Shadinglanguages oft nicht vorhanden sind. Dazu wird der Shadercode nicht mehr ausschließlich in den Shadinglanguages erstellt, sondern mit Hilfe zusätzlicher Werkzeuge wie der Programmiersprache C++. Ziel der Ansätze ist es oft, den Überblick über die Shaderprogramme zu verbessern, und häufig genutzte Komponenten flexibel einsetzen und wiederverwenden zu können. Zunächst wird der Begriff des Shaders genauer betrachtet. Daraufhin werden einzelne Klassen von Ansätzen zur Shaderbeschreibung betrachtet, und beispielhafte Umsetzungen vorgestellt.

3.1 Überblick

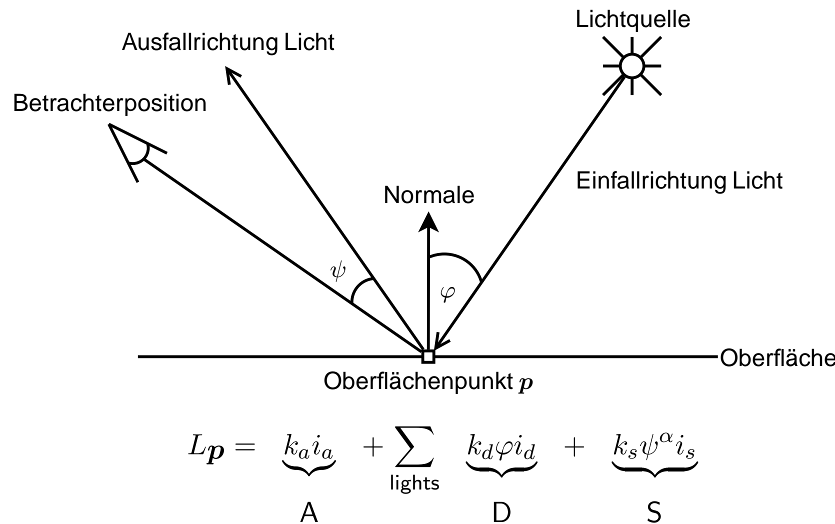
Die Erstellung eines computergenerierten Bildes (*Rendering*) kann im weitesten Sinne als die Berechnung dessen verstanden werden, was von einer gewählten Betrachterposition aus zu sehen ist. Dazu müssen mit geeigneten Verfahren zwei Dinge bestimmt werden:

1. Die von der Betrachterposition sichtbaren Punkte eines Objekts (z. B. sichtbare Oberflächepunkte oder sichtbare Volumenelemente).

2. Die Farbe bzw. das Aussehen dieser Punkte.

Für die Bestimmung der sichtbaren Punkte ist die Vertexverarbeitung und Rasterisierung auf der GPU nur eine Möglichkeit (siehe [Shi02, HKRs+06]). In diesem Zusammenhang bezeichnet *Shading* die Berechnung des Aussehens eines Punkts unter Einbeziehung der Betrachterposition, und *Shader* eine diese Berechung ausführende Einheit. Ein Shader beschreibt, wie eine Oberfläche oder ein Volumen auf Licht reagiert, und welches Licht von einem Punkt des Objekts beim Betrachter ankommt. Dabei sind die Menge des Lichts, welche die *Helligkeit* bestimmt, und die Art des Lichts, welche die *Farbe* bestimmt, zu unterscheiden. Es lässt sich also eine Doppelverwendung des Begriffs *Shader* beobachten. Einerseits sind allgemein Einheiten gemeint, welche eine Shadingberechnung durchführt. Andererseits sind explizit Programme für Grafikkarten gemeint.

Grundsätzlich lassen sich zwei Arten von Shadern unterscheiden: *Parametrische Beleuchtungsmodelle* und *prozedurale Shader*. Parametrische Beleuchtungsmodelle sind mathematische Formeln, die aus einer festen Anzahl einfacher Parameter wie Lichtquellenposition oder Oberflächennormale die gewünschte Information berechnen. Beleuchtungsmodelle sind aufgrund ihres Aufbaus in der Vielfalt der Effekte eingeschränkt. Ein Beispiel für ein parametrisches Beleuchtungsmodell ist das Modell nach *Phong*



$$L_{\boldsymbol{p}} = \underbrace{k_a i_a}_{\text{A}} + \sum_{\text{lights}} \underbrace{k_d \varphi i_d}_{\text{D}} + \underbrace{k_s \psi^\alpha i_s}_{\text{S}}$$

Abbildung 14: Das Phong-Beleuchtungsmodell.

[Pho75]. Es beschreibt die Farbe bzw. das Licht $L_{\boldsymbol{p}}$, welches von gegebener Betrachterposition vom Obeflächenpunkt $\boldsymbol{p}$ wahrgenommen wird. Mehrere Lichtterme werden unterschieden: Stets vorhandenes Licht (*ambienter* Term A), vom Objekt in alle Richtungen reflektiertes Licht (*diffuser* Term D) und vom Objekt in die Betrachterrichtung reflektiertes Licht (*spekularer* Term S). k beschreibt die jeweiligen Reflektionseigenschaften des Objekts, und i die jeweiligen Lichteigenschaften. Abbildung 14 veranschaulicht die Berechnung.

Prozedurale Shader abstrahieren den Berechnungsprozess als eine Funktion, welche die entsprechende Shadinginformation zurückliefert. Eingabewerte und interne Berechnungen sind weitgehend frei wählbar. Ein prozeduraler Shader kann ein Beleuchtungsmodell implementieren, ist aber nicht darauf beschränkt. Prozedurale Shader werden meist in eigenen Sprachen geschrieben, und erlauben z. B. die Integration von *Texturemapping*, und vielen anderen zusätzlichen Effekten.

Abhängig vom Berechnungsansatz werden Shadingverfahren als *photorealistisch*, *phänomenologisch* (wie das Phong-Modell) oder *physikalisch basiert* bezeichnet. Einen Überblick gibt [MH02]. Abhängig von den Ansprüchen, die an die Geschwindigkeit der Shadingberechnung gestellt wird, unterscheidet man echtzeitfähiges *Realtimeshading* und *Offlineshading*. Echtzeitfähiges Shading muss schnell genug sein, um eine gewisse Anzahl von Bildern pro Sekunde berechnen zu können. Meist werden Raten von 15 und mehr Bildern pro Sekunde als *echtzeitfähig* bezeichnet. Darunter liegende Raten, die immer noch mehrere Bilder pro Sekunde haben, werden als *interaktive* Raten bezeichnet, da die Darstellung damit immer noch

schnell auf Benutzereingaben reagieren kann. An Offlineshading werden keine prinzipiellen Geschwindigkeitsansprüche gestellt, auch wenn natürlich pragmatische Einschränkungen gelten. Für Offlineshading gibt es umfangreiche Programmiersprachen, die z. B. für die Erstellung computeranimierter Filme verwendet werden. Eine bekannteste Spezifikation einer Offlineshading-API ist RENDERMAN [HL90, Pix], ein Überblick über verschiedene Sprachen gibt [MH02].

Für die Erstellung von Bildern in Echtzeit werden heute meist Grafikkarten verwendet. So beschäftigen sich auch die meisten Ansätze und Veröffentlichungen zu Echtzeitshading heute mit grafikkartenbasierten Shadern. Die fortschreitende Entwicklung verwandter Gebiete wie *Echtzeit-Raytracing* lässt aber vermuten, dass echtzeitfähiges Shading auch in diesen Bereichen bald eine wichtige Rolle spielen wird [AGM06].

GPUs bieten die Möglichkeit, prozedurales Shading einzusetzen. Mit dem Vertex- und Fragmentprogramm lassen sich zwei Teile der Grafikpipeline austauschen, und für entsprechende Berechnungen nutzen. Obwohl zu den Aufgaben beider Programme, insbesondere des Vertexprogramms, noch mehr Aufgaben als Shading gehören, hat sich für beide der Begriff *Shaderprogramm* etabliert.

3.2 General Purpose GPU

Shaderprogramme profitieren von der hohen Geschwindigkeit aktueller Grafikkarten bei der Durchführung von Berechnungsoperationen. Grafikkarten werden daher auch zur Ausführungen von Programmen verwendet, die nichts mit Rendering und Shading zu tun haben. Dieser Einsatz von Grafikkarten als Hilfsprozessor für allgemeine Berechnungen wird unter dem Begriff *General Purpose Computation on GPU* (GPGPU) zusammengefasst [OLG+07].

Ansätze zur Verwendung der Grafikkarte für beliebige Berechnungen abstrahieren oft stark von der Hardwarearchitektur. Der Anwender wird mit den Eigenheiten der GPU nicht mehr konfroniert. BROOK FOR GPUS [BFH+04] nutzt dazu die Eigenschaften der Grafikkarte als *Streamingprozessor*, die Ströme gleichförmiger Daten (Vertices, Primitive, Fragmente) verarbeitet. BROOK erlaubt die Verarbeitung vieler weiterer Datentypen als Ströme auf der GPU. Auch SH (siehe Abschnitt 3.4) integriert Streams [MTP+04]. Das Streamkonzept abstrahiert von der Grafikhardware, wurde aber dennoch zur Realisierung grafikrelevanter Probleme verwendet [ODK+00, PBMH02].

Die Grafikkartenhersteller selbst ermöglichen mit eigenen Entwicklungswerkzeugen GPGPU-Programmierung auf den jeweiligen Grafikkarten. CUDA [4] und CTM [AMD06, SP06] erlauben die Nutzung der Grafikkarte

[4] http://developer.nvidia.com/object/cuda.html. Zugriff: 15.01.2008

als allgemeiner Koprozessor mit umfangreichen Fähigkeiten für parallele Datenverarbeitung.

GPGPU-Ansätze lassen sich, ebenso wie jede klassische CPU, zur Berechnung von Grafikalgorithmen verwenden [OLG+07]. Diese Algorithmen nutzen dann GPGPU-Techniken, die von der Grafikhardware abstrahieren, zur Berechnung von Bildern. Sind die Algorithmen dem jeweiligen Ansatz angepasst, sind performante Umsetzungen möglich. Renderingalgorithmen mit viel Geometrie (Slicing) sind für die Standardpipeline der GPU konzipiert, und eignen sich schlecht für GPGPU-Umsetzungen. Andere Ansätze wie Raycasting nutzen wenige Teil der GPU-Pipeline. Sie bieten sich daher an zur Realisierung mit GPGPU-Ansätzen oder auf der CPU. Diese Arbeit konzentriert sich auf Ansätze mit Verwendung der Standardpipeline (nicht auf GPGPU-Ansätze).

3.3 Datenflussbasierte Beschreibung von Shadern

Eine Klasse von Ansätzen für die Erstellung von Shaderprogrammen modelliert die Bestandteile von Shadern über Operationen, zwischen deren Eingabe- und Ausgabewerten Datenabhängigkeiten festgelegt werden. Sie basieren auf der *Datenflussprogrammierung*; entsprechende Programme haben die Struktur eines Graphen. Ein Überblick über Graphen und entsprechende Algorithmen gibt [Sed02]. Im Folgenden wird zunächst ein Überblick über Datenflussprogrammierung und ihre grafische Anwendung gegeben. Daraufhin werden auf diesem Konzept aufbauende Ansätze zur Shadererstellung vorgestellt.

3.3.1 Datenflussprogrammierung

[Sha92] gibt folgende Definitionen zur Unterscheidung von *Datenflussprogrammierung* zu herkömmlicher *Kontrollflussprogrammierung*:

- In einer *Datenflussberechnung* werden die Operationen in einer Reihenfolge ausgeführt, die von den Abhängigkeiten der verwendeten Daten untereinander bestimmt wird. Der Programmierer legt die Datenabhängigkeiten fest.
- In einer *Kontrollflussberechnung* werden Operationen in einer Reihenfolge ausgeführt, die vom Programmierer festgelegt wird. Dazu nutzt er festgelegte Regeln, z. B. die Syntax einer Programmiersprache.

Ein Datenflussprogramm kann durch einen *gerichteten Graph* dargestellt werden. Die grundlegenden Elemente eines solchen Programms sind *Knoten* und *Kanten*. Knoten repräsentieren Operationen bzw. Funktionen, oder stellen als *Konstantengenerator* Werte zur Verfügung. Kanten stellen die Datenabhängigkeiten zwischen Knoten dar. Ein Beispiel für ein Datenflussprogramm, sowie ein entsprechendes Kontrollflussprogramm, ist in Ab-

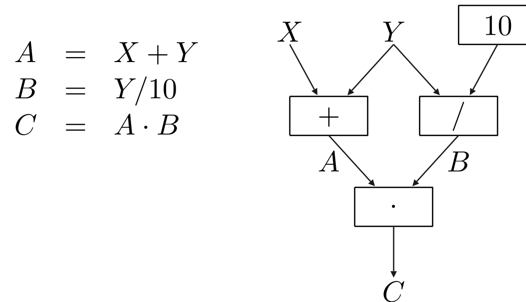


Abbildung 15: Einfache sequentielle Berechnung und entsprechendes Datenflussberechnung.

bildung 15 gezeigt. Datenflussprogramme haben aufgrund ihres Aufbaus verschiedene Eigenschaften, von denen die wichtigsten das *Nichtvorhandensein von Seiteneffekten* und die *einmalige Zuweisung von Variablen* sind. Da die Programmausführung von Datenabhängigkeiten bestimmt wird, dürfen Variablen nicht einfach ihre Werte ändern. Man kann nicht sicher sein, dass alle abhängigen Berechnungen schon ausgeführt wurden.

Ein solches Programm kann auf zwei Arten ausgeführt werden. Geschieht die Ausführung *data-driven*, so wird die Reihenfolge der Operationen von der Verfügbarkeit der Daten bestimmt. Eine Operation wird ausgeführt, sobald alle Eingabewerte zur Verfügung stehen. Die Abarbeitung des Graphen beginnt bei den Eingabewerten. Bei der *demand-driven* Abarbeitung wird die Reihenfolge der Operationen bestimmt davon, welche Werte benötigt werden. Benötigt eine Berechnung einen Eingabewert, so wird die dem Eingabewert entsprechende Operation ausgeführt. Die Abarbeitung des Graphen beginnt bei den gesuchten Ausgabewerten. Wird die Berechnung von der Verwendung der Daten gesteuert, so werden also keine unnötigen Berechnungen gemacht. Teilgraphen, die nicht zur Berechnung eines gewünschten Wertes beitragen, werden nicht verarbeitet.

Die ursprüngliche Motivation für die Entwicklung des Datenfluss-Berechnungsmodells war die Ermöglichung von *paralleler Ausführung der Operationen*. So sind dem Konzept nach für die Berechnung des sequentiellen Programms aus Abbildung 15 drei Zeitschritte notwendig, da Operationen nur hintereinander ausgeführt werden können. Beim Datenflussprogramm können die ersten beiden Operationen gleichzeitig ausgeführt werden. Somit sind nur zwei Zeitschritte notwendig. Dieses Potential für Parallelität kann allerdings nur begrenzt ausgenutzt werden, da zwischen Knoten, die auf einem *Pfad* im Graphen liegen, eine sequentielle Abhängigkeit vorliegt (*Sequential Codesegment*, [Bic92]). Die Knoten eines Datenflussgraphs stellen Operationen dar, welche sehr fein strukturiert sind (*Fine-grained Dataflow*). Der Aufwand für die Verarbeitung des Graphen überwiegt daher meist die

Gewinne durch die Parallelität. Ansätze mit geringerer Granularität (*Hybrid Dataflow*) versuchen, diese Probleme zu umgehen. Ein Knoten kann dann für mehr als eine Operation stehen, z. B. eine Funktion, die in einer sequentiellen Programmiersprache geschrieben ist.

Kontrollfluss- und Datenflussparadigma können als zwei Extreme verstanden werden, deren Vorteile im hybriden Datenfluss verbunden werden. Die Datenflussbeschreibung bietet den Vorteil in natürlicher Weise parallelisierbar zu sein. Auch bietet die Datenflussbeschreibung für viele Probleme eine intuitive Möglichkeit, Lösungen zu strukturieren [Sch98]. Sequentielle Programme hingegen haben bei der Abarbeitung einzelner Operationen weniger Mehraufwand.

Es gibt viele weitere Aspekte, die für die Beschreibung von Datenflusssprachen relevant sind. Einen guten Überblick über vermittelt [JHM04].

3.3.2 Visuelle Datenflussprogrammierung

Datenflussgraphen spielen eine wichtige Rolle im Feld der *visuellen Programmierung*. Eine umfangreiche Auseinandersetzung mit diesem Thema, die auch eine genaue und differenzierte Definition des Begriffs der visuellen Programmierung enthält, liefert [Sch98]. Im Folgenden soll zunächst kurz der Begriff der visuellen Programmierung charakterisiert werden, und danach auf wichtige Aspekte visueller, datenflussbasierter Programmiersprachen eingegangen werden.

Ziel der visuellen Programmierung ist es, Programme zu entwerfen und zu implementieren. Nach [Sch98] nutzt sie Programmiersprachen, deren syntaktische und semantische Elemente visueller bzw. graphischer Natur sind. [Mye86] merkt an, dass diese Elemente zwei- oder mehrdimensional sind. Im Gegensatz dazu stehen textuellen Programmiersprachen, die als eindimensionale Ströme verarbeitet werden.

Erst *integrierte Werkzeuge* erlauben die Arbeit mit der visuellen Programmiersprache. Analog zum Texteditor bei textueller Programmierung sind Werkzeuge notwendig, die das Manipulieren der visuellen Implementierung zulassen. Im Vergleich zu textbasierten Sprachen sind visuelle Sprachen stärker auf die zugehörige Entwicklungsumgebung angewiesen. Visuelle Programmiersprachen verwischen die Unterscheidung zwischen Programmiersprache und Entwicklungsumgebung [JHM04]. Auch die Phasen Entwurf und Implementation werden vermischt. Danach ist es gerade diese Vermischung verschiedener Aufgaben und Phasen, die ein Rapid Protoyping ermöglicht.

Bereits in den ersten Veröffentlichungen zur Datenflussprogrammierung wurden grafische Repräsentationen verwendet, um die vorgestellten Ideen zu transportieren und diskutieren [Bic92]. Grundlegende Elemente einer graphischen Notation zeigt Abbildung 16. Dargestellt sind Daten- und Operationsknoten, Verbindungen, bedingte Anweisungen (*Gates*) und

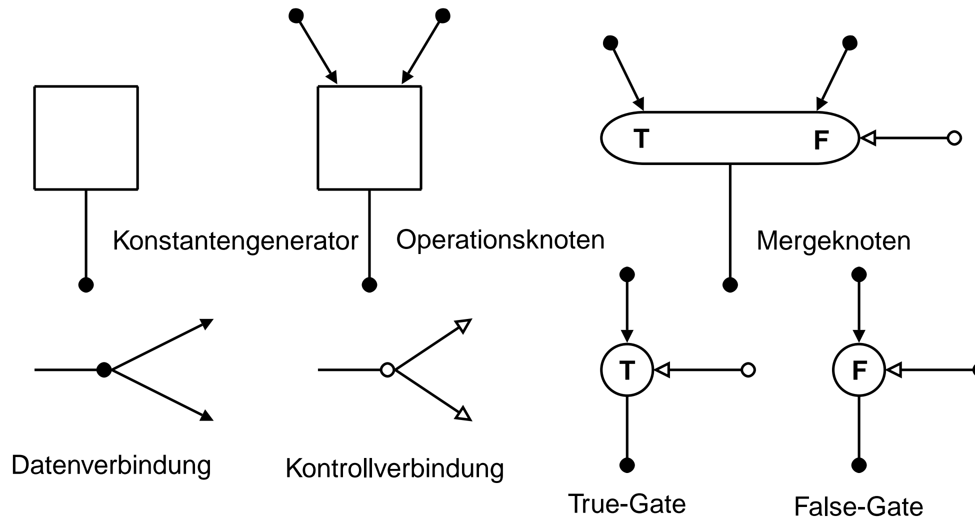


Abbildung 16: Beispielhafte, grafische Notation für Datenflussgraphen. Diese Notation wurde [Sha92] entnommen, geht aber auf [Den74] zurück.

ein Auswahlknoten (*Mergeknoten*). Die Datenfluss-Programme selbst wurden zunächst in textuellen Sprachen geschrieben [JHM04]. Um die Vorteile grafischer Notation auch zur Programmspezifikation einsetzen zu können, wurden visuelle Sprachen zur Beschreibung von Datenflussprogrammen (*Dataflow Visual Programming Languages, DFVPLs*) entwickelt. Die Diskussion der Vor- sowie Nachteile visueller Repräsentation von Datenflussprogrammen ist umfangreich. Eine zentrale Annahme ist, dass Problemlösungen sich oft grafisch oder in Form eines Diagramms besonders gut vermitteln und finden lassen. Für eine tiefergehende Diskussion visueller Datenflussprogrammierung sei auf [Sch98, MP00, JHM04] verwiesen. In diesen Veröffentlichungen ist auch eine umfangreiche Sammlung an Beispielen von visuellen Datenflussprogrammiersprachen zu finden.

Die Bedeutung der visuellen Datenflussprogrammierung liegt nicht primär darin, Datenflussprogramme zu spezifizieren. Vielmehr liegt sie in ihrer guten Unterstützung des generellen Entwurfsprozesses von Algorithmen und Software [JHM04] sowie Rapid Prototyping [MP00]. Datenflusssprachen eignen sich gut als *Koordinationssprache* [GC92]. Dabei sind es vor allem die Entwicklungsumgebungen, in denen die visuellen Elemente nach Datenflusskonzepten geordnet werden. Oft sind die darunter liegenden Schichten klassische, kontrollflussorientierte Programmiersprachen. Ein Beispiel dafür ist das in Abschnitt 4.1 vorgestellte Framework, welches in C++ geschriebene Knoten verwendet, und diese im visuellen Editor nach Datenabhängigkeiten strukturiert. Hier kommt somit ein *Hybrid Dataflow* Ansatz zum Einsatz. Ein weiteres Beispiel einer visuellen Entwicklungsumgebung mit Datenflussgrundlage ist die MICROSOFT VISUAL

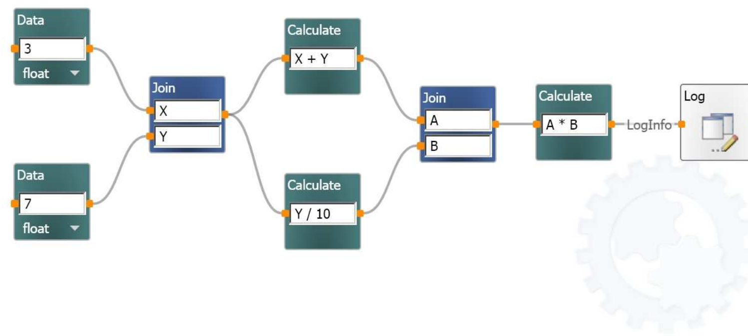


Abbildung 17: Berechnung aus Abbildung 15, umgesetzt in der MICROSOFT VISUAL PROGRAMMING LANGUAGE.

PROGRAMMING LANGUAGE [Micc]. Abbildung 17 zeigt die Berechnung aus Abbildung 15 in dieser Entwicklungsumgebung. Abbildung 19 zeigt eine einfache Schleife in derselben Entwicklungsumgebung.

In Datenflusssprachen wurden schon von Anfang an zusätzliche Kontrollstrukturen wie Schleifen oder bedingte Ausführung integriert [Sha92] (siehe Abbildung 18). [MP00] stellt fest, dass solche aus der kontrollflussorientierten Programmierung bekannten Strukturen für die Lösung vieler nicht-trivialer Probleme notwendig sind. Die Autoren untersuchen umfangreich, wie man Kontrollstrukturen in visuelle Datenflusssprachen integrieren kann[5]. Im Hinblick auf die Ausführungen in Abschnitt 5 und 6 ist es sinnvoll, die grundlegenden Aussagen über Schleifen in Datenflussgraphen wiederzugeben:

- Integriert man iterative Konstrukte in einen Datenflussgraph, entstehen *Zyklen*.
- Zyklische Subgraphen, und alle direkt oder indirekt von seinen Knoten abhängenden Knoten, sind Bestandteil des Rumpfs einer Iteration.
- Iterative Subgraphen benötigen Schnittstellen, um initiale Werte zu erhalten und Ergebniswerte weiterzugeben.

Diese Zusammenhänge sind in Abbildung 18 zu sehen. Alle Knoten, die zum Schleifenrumpf gehören können, sind visuell im Kasten *Schleifenrumpf* zusammengefasst. Der *Mergeknoten* sowie die *Gates* stellen die Schnittstellen nach außen dar. Abbildung 19 zeigt die Realisierung einer Schleife in

[5] Die Autoren untersuchen hier auch die Korrektheit im Bezug auf Datenflussparadigmen, wie den Verzicht auf veränderbare Variablen. Dieser Aspekt hat für diese Arbeit jedoch keine Relevanz.

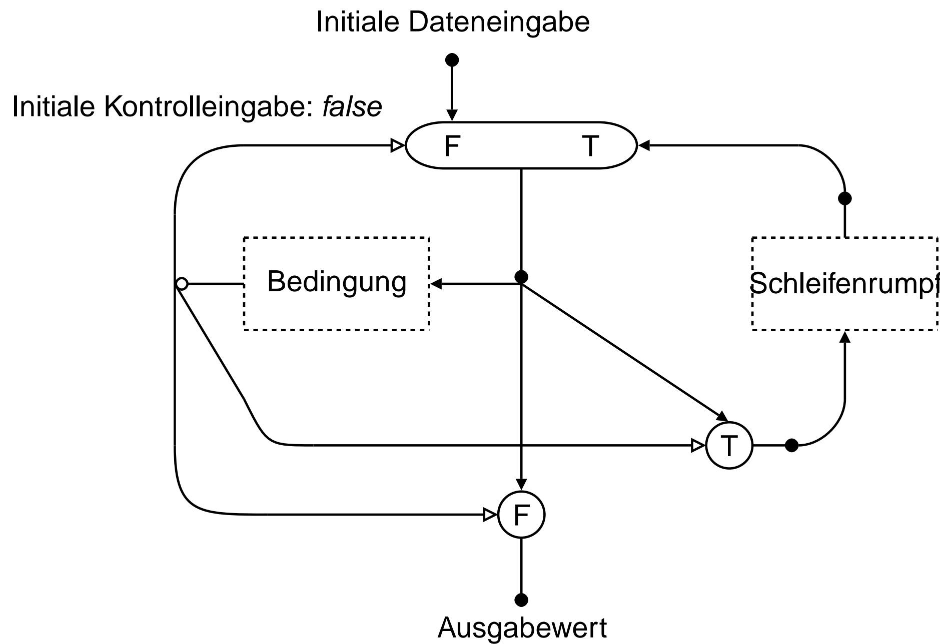


Abbildung 18: While-Schleife in der Notation aus Abbildung 16. Gestrichelt umrandete Bereiche stellen Subgraphen dar. Vgl. [Sha92]

der *Microsoft Visual Programming Language*. Zur Realisierung der Schleife werden Variablen eingesetzt, der Wert im Schleifenrumpf angepasst wird. Dies ist kein reiner Datenflussansatz. Dennoch sind auch hier der Zyklus und die Schnittstelle nach außen, die hier der Mergeknoten bildet, zu erkennen.

Bei komplexeren iterativen Konstrukten nimmt die Übersichtlichkeit und intuitive Begreifbarkeit der Programme ab (Abbildung 20). Größere Projekte benötigen daher weitere Mechanismen wie Gruppierungen in mehrere Abstraktionsebenen, um die Übersichtlichkeit zu verbessern.

3.3.3 Shadetrees

Bereits vor 20 Jahren stellte [Coo84] mit den *Shadetrees* einen flexiblen Ansatz zur Erzeugung von Shadern vor, der auf Datenflussprogrammierung und ihrer guten, visuellen Darstellbarkeit aufbaut. Der Ansatz beschreibt eine grundlegende Herangehensweise an die Beschreibung von Shadern, und bildet auch heute noch die Grundlage für neue Arbeiten im GPU-Umfeld. Die vorgestellte Grundannahme besagt, dass kein Beleuchtungsmodell allein für die Beschreibung aller möglichen Objekte ausreicht. Shadetrees sind modular aufgebaut und erlauben es, verschiedene Techniken zur Beleuchtung und Texturierung zu kombinieren.

Bei Shadetrees handelt es sich um einen *Hybrid Dataflow* Ansatz zur Beschreibung von Shadern. Einzelne Codeteile, die in einer sequentiellen

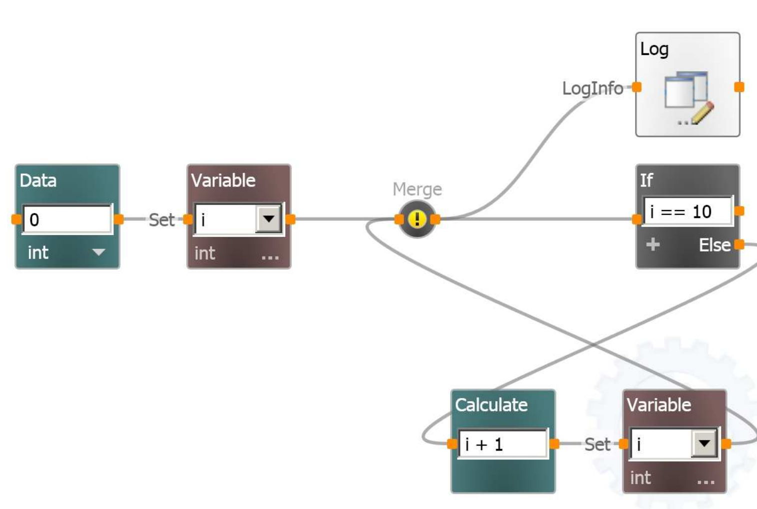


Abbildung 19: Einfache Schleife in der MICROSOFT VISUAL PROGRAMMING LANGUAGE.

Sprache geschrieben werden, werden durch Datenflüsse verbunden. Die Elemente eines Shadetree bestehen aus *Darstellungsparametern* und *Operationen*. Die Elemente werden in einem gerichteten, nichtzyklischen Graph (*Directed Acyclic Graph*, DAG) nach Datenabhängigkeiten strukturiert. Die *Blätter* des Graphen sind Darstellungsparameter, die nicht durch Operationen erzeugt werden, sondern globale Eingaben des Shaders darstellen (Konstantengeneratoren). Dabei kann es sich um Oberflächennormalen, Farbwerte, Texturen und vieles mehr handeln. Alle anderen Knoten sind Operationen. Operationen haben eine beliebige Anzahl an Darstellungsparametern als Eingabe, und berechnen ein oder mehrere Parameter als Ausgabe. Bei den Berechnungen kann es sich um mathematische Operationen, um die Auswertung eines Beleuchtungsmodells, oder beliebige andere Berechnungen handeln. Operation nutzen die Ausgabewerte ihrer Kindknoten als Eingabewerte. Die Verknüpfung zweier Parameter stellt somit die Vorgänger-Nachfolger-Beziehung zweier Knoten her, und damit eine Datenabhängigkeit. Die Ausgaben einer Operation können von mehreren anderen Operationen weiterverwendet werden. Somit kann ein Knoten mehrere direkte Vorgänger haben. Es handelt sich bei Shadetrees daher nicht um *Bäume* im Sinne der klassischen Datenstruktur (siehe [Sed02]). Berechnet ein Shadetree die Farbe eines Objektpunkts, so ist der Ausgabeparameter des Wurzelknotens ein Farbwert. Beispiele für Shadetrees zeigt Abbildung 21.

Die Berechnungen des Shadetrees werden ausgeführt, indem der Graph mit *Tiefensuche* traversiert wird. Die Operationen des Shaders werden dabei *topologisch* sortiert (siehe [Sed02]). Da bei Tiefensuche die Nachfolge-

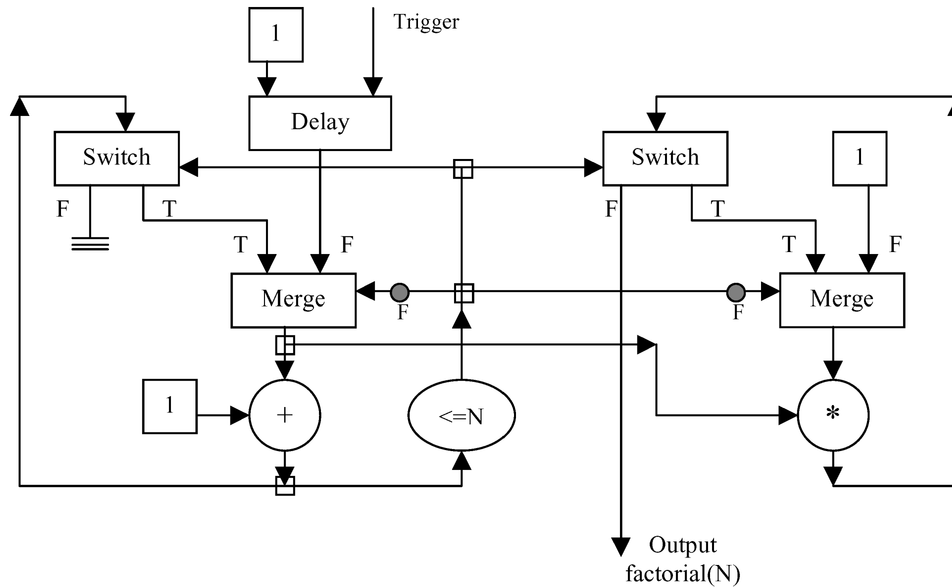


Abbildung 20: Datenflussgraph einer Fakultätsberechnung. Bildquelle: [JHM04]

Knoten vor ihren Vorgängern verarbeitet werden, ist sichergestellt, dass alle Operationen die nötigen Eingabewerte vorfinden. Diese sequentielle Abhängigkeit wird in Abschnitt 3.3.1 weiter erläutert. Erstellung und Traversierung des Graphen wird einmalig vor der eigentlichen Ausführung des Shaders durchgeführt. Beim Traversieren wird eine Liste von sequentiell abarbeitbarem Code erstellt. Die Mehrarbeit durch die Auswertung des Shadetrees ist aus diesem Grund minimal.

Zu betonen ist, dass die inhärente Möglichkeit zur parallelen Abarbeitung von Programmteilen, die durch die Datenflussbeschreibung oft gegeben ist, durch die Kompilierung in jeweils ein komplett sequentielles Programm immer verloren geht. Dies ist bei allen in dieser Arbeit vorgestellten Shadetree-Ansätzen der Fall. Datenflussprogrammierung wird zur Strukturierung des Programms genutzt, aber nicht zu dessen Ausführung (siehe Abschnitt 3.3.2) Bei der Umwandlung in Programme für Offlineshading ist es aber denkbar, auch die Parallelität zu berücksichtigen. GPUs hingegen unterstützen im Moment keine Parallelität innerhalb der Shaderprogramme.

Die *Buildingblock-Shader* [AW90] entwickeln die Idee der Shadetrees weiter. Der Ansatz wird um Abhängigkeiten zwischen Knoten erweitert, die nicht auf Datenfluss beruhen. Der Programmierer kann so auch Codeteile ausführen, die keinen Wert zurückliefern und ihren Wirkung z. B. über Seiteneffekte entfalten. Ein Beispiel zeigt Abbildung 21b.

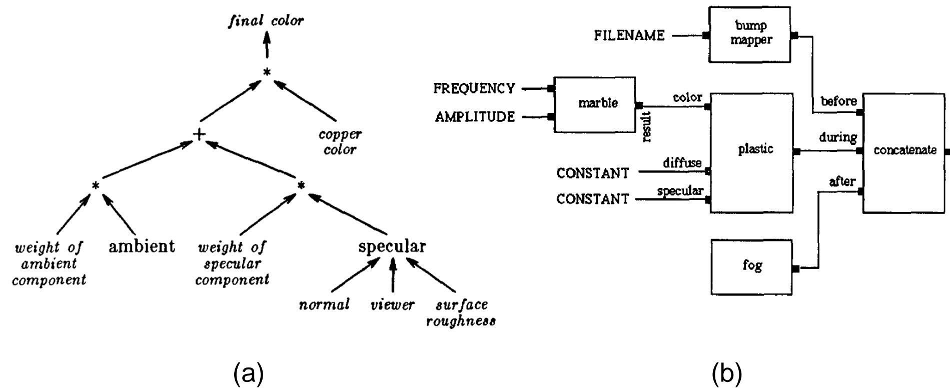


Abbildung 21: Beispielhafte Shadetrees. Bildquelle (a): [Coo84]; Bildquelle (b): [AW90]

Ein weiterer Aspekt der Shadetrees ist, dass Ein- und Ausgabeparametern *Datentypen* zugeordnet sind. Diese Typen müssen bei verbundenen Parametern übereinstimmen. Es gibt also die Möglichkeit der *Nichtübereinstimmung von Typen*. Die bisher vorgestellten Umsetzungen der Shadetrees überlassen dem Anwender die Sicherstellung der Typübereinstimmung. Die Betrachtung der Building Block Shader aus [AW90] erwähnt die Möglichkeit *automatischer Typkonvertierung*. [MSPK06] stellt *Abstract Shadetrees* vor, welche eine solche automatische Typanpassung realisieren. Hier werden nicht mehr die einzelnen Parameter von Berechnungsknoten verknüpft. Zwei Knoten werden durch eine einzige Kante verbunden, wenn zwischen ihnen eine Datenabhängigkeit besteht. Hier werden die Parameterverbindungen *abstrakt* dargestellt. Ein als *Weaver* bezeichnetes Programm übernimmt die Verarbeitung eines Abstract Shadetrees. Dabei wird dieser zu einem konkreten Shadetree übersetzt, indem die Knotenparameter verknüpft wurden. Um die Umwandlung des abstrakten in den konkreten Shadetree zu ermöglichen, werden Parametern nicht nur Datentypen, sondern auch *semantische Typen* zugeordnet. Beispielhafte Typen sind:

- **Space**: Tangent, Object, World, Screen
- **Interpretation**: Color, Texture Coordinate, Surface Normal, Direction, Point

Diese zusätzliche semantische Information erlaubt es den Programmieren der einzelnen Knoten, Aussagen über den Bestimmungszweck einzelner Parameter zu machen. Der Weaver verknüpft zwei Parameter nur dann, wenn Datentyp und semantischer Typ exakt übereinstimmen. Ist dies nicht der Fall, wird eine umfangreiche Sammlung von *Konvertierungsregeln* durchsucht, um über Zwischenschritte diese Übereinstimmung herzustellen. Ei-

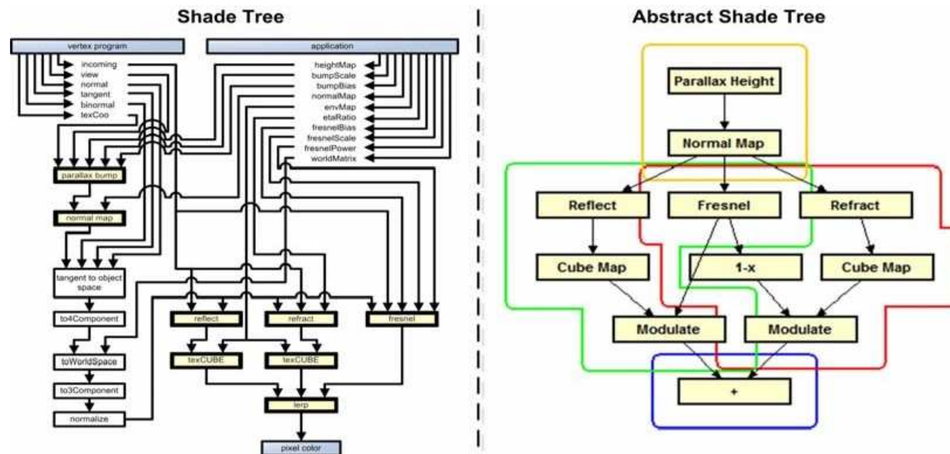


Abbildung 22: Abstrakter Shadetree mit zugehörigem konkreten Shadetree. Bildquelle: [MSPK06]

ne solche Konvertierung kann z. B. die Umrechnung von Welt- in Objektkoordinaten sein.

Im Unterschied zu den bisher vorgestellten Ansätzen ist es das Ziel der Abstract Shadetrees, Shadercode für die Grafikkarte zu erstellen. Als Basissprache kommt GLSL zum Einsatz. Ein Beispiel für einen Abstract Shadetree zeigt Abbildung 22.

Der Ansatz der Abstract Shadetrees möchte weitere Probleme lösen. Shadetrees werden durch die große Anzahl an Parameterverknüpfungen schnell sehr unübersichtlich (vgl. Abbildung 22), was durch die einzelne abstrakte Verknüpfung behoben wird. Da die Parameterverknüpfungen abhängig von der semantischen Information über die Parameter automatisch gemacht werden, kann sich sogar das Interface eines Moduls ändern, ohne es in einer bereits existierenden Verwendung unbrauchbar zu machen. Zuletzt erlaubt es die Abstraktion, einfachere Graphen für ein Problem zu erstellen, da die automatische Typkonvertierung viele Elemente klassischer Shadetrees überflüssig werden lässt. Klassische Shadetrees sind dem durch sie repräsentierten Programmcode oft sehr ähnlich, da sie z. B. viele Konvertierungsknoten enthalten. Somit bieten abstrakte Shadetrees nach Aussage von [MSPK06] für Nutzer ohne Programmierkenntnisse (z. B. Künstler) einen erleichterten Zugang.

3.3.4 Visuelle Shaderprogrammierung

Auf Shadetrees basierende Ansätze zur Entwicklung von Shaderprogrammen sind datenflussbasiert, und eignen sich daher gut für die visuelle Programmierung. Während die ursprünglichen Shadetrees aus [Coo84] noch

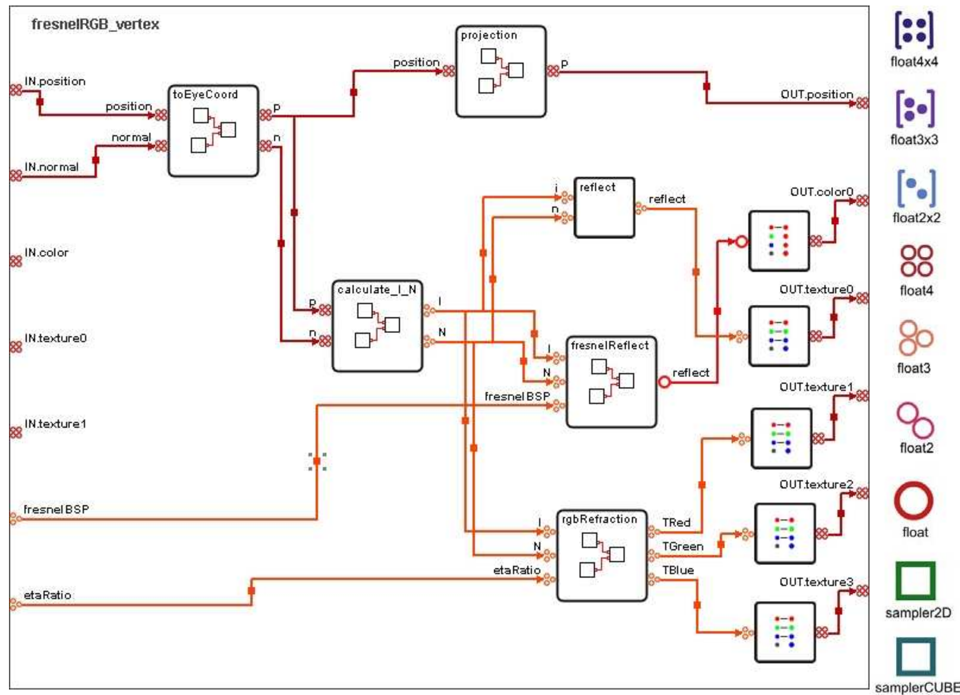


Abbildung 23: Ein Vertexprogramm im VISUAL SHADITOR, mit beigefügter Legende für die visuelle Repräsentation von Datentypen. Bildquelle: [GBD04]

rein textuell erstellt wurden, beschreiben darauf aufbauende Arbeiten jeweils auch passende, grafische Editoren [AW90, MSPK06]. Diese Entwicklungsumgebungen erlauben es, effektiv Shaderprogramme zu erstellen, und die Zusammenhänge visuell zu erfassen.

[GBD04] stellt eine visuelle Programmiersprache für die Entwicklung von Shadern für die Grafikkarte vor, der in der späteren Veröffentlichung [GD06] als VISUAL SHADITOR bezeichnet wird. Es kommt der klassische Shadetree Ansatz zum Einsatz. Als Basissprache für den aus dem Shadetree erzeugten Code wird CG genutzt. Für jeden Programmtyp (Vertex, Fragment) wird ein eigener Graph erstellt. Programme haben selbst Eingabe und Ausgabeparameter, mit denen Knoten verbunden werden können. Dabei handelt es sich um die pro Vertex oder Fragment vom entsprechenden Programm erhaltenen bzw. berechneten Informationen, wie Normalen- oder Texturinformationen. Im Unterschied zu den bisher vorgestellten grafischen Editoren, in denen Parameter und ihre Datentypen textuell (Building Block Shader) oder gar nicht (Abstract Shadetrees) dargestellt werden, werden die Parameter hier grafisch visualisiert. So werden Datentypen visuell unterschieden und Parameter können zudem benannt

werden. Abbildung 23 zeigt ein Vertexprogramm in dieser Notation. Um die Arbeit auch bei komplexen Programmen überschaubar zu halten, gibt es im VISUAL SHADITOR verschiedene Knotentypen. Sie ordnen die Knoten verschiedenen Aufgaben zu, und erlauben deren Gruppierung in wiederverwendbare Einheiten. Auf diese Weise ist das Arbeiten auf mehreren Abstraktionsebenen möglich.

[PHF07] stellen einen Editor zur visuellen Erstellung von Shadetrees für den Einsatz im Volumenrendering vor. Knoten und Parameter werden visuell dargestellt, Parametertypen in textueller Form innerhalb der Knoten. Es kommt der klassische Shadetree-Ansatz zum Einsatz. Zur Anpassung an das Volumenrendering werden wichtige Elemente, wie die darzustellenden Volumen, direkt im Editor angezeigt und können mit den Knoten des Shadetrees verbunden werden.

Alle in diesem Kapitel vorgestellten Shadetree-Ansätze erlauben die Erstellung nicht-zyklischer Shadertrees. Dies erlaubt die Erstellung linearer Datenabhängigkeiten zwischen den Knoten. Schleifen sind auf Ebene des Shadetrees nicht realisierbar.

Während sich Arbeiten wie [GBD04, MSPK06] allein mit der Erstellung von Shadern für GPUs beschäftigen, gibt es auch für andere Anwendungsgebiete Editoren zur Bearbeitung von Shadingeffekten auf Grundlage visueller Datenflussprogrammierung. Sie sind Bestandteil komplexer Programme wie Computer-Animationssoftware oder Gameengine-Entwicklungspaketen. Beispiele sind der *Visual Material Editor* der UNREAL ENGINE [6] oder der Editor für *Shading Networks* von AUTODESK MAYA [7] . MENTAL MILL [8] schließlich erlaubt die visuelle Programmierung von Shadern mit Hilfe eines Graphkonzepts, und kann Shadercode sowohl für Offlinerenderer als auch für die Grafikkarte erzeugen.

3.4 Metaprogrammierung mit C++

Shaderprogramme sind neben anderen Prozessen, wie der Festlegung der grafischen Primitive oder dem Management der Texturen, nur ein Bestandteil einer Grafikanwendung. Das Shaderprogramm wird als separates Programm verwaltet, und ist in einer spezialisierten Shadersprache wie GLSL geschrieben. Die restlichen Befehle für de Grafik-API (z. B. OpenGL) befinden sich im Hauptprogramm, welches in einer nicht-spezialisierten Programmiersprache wie C++ geschrieben ist. Die Trennung zwischen Shadern und den anderen Programmteilen der grafikrelevanten Implementation birgt ein Schnittstellenproblem. Informationen, die bereits im Haupt-

[6] http://www.unrealtechnology.com/features.php?ref=editor. Zugriff: 15.01.2008

[7] http://www.autodesk.de/maya. Zugriff: 15.01.2008

[8] http://www.mentalimages.com/2_4_mentalmill/index.html. Zugriff: 15.01.2008

ShInputNormal3f	Vertex-Normale
ShInputPosition3f	Vertex-Position
ShInputColor3f	Vertex-Farbe
ShInputTexCoord2f	Vertex-Texturkoordinaten
ShOutputPosition3f	Transformierte Position
ShOutputColor3f	Veränderte Farbe
ShOutputTexCoord2f	Veränderte Texturkoordinaten

Tabelle 1: Liste semantischer Variablen eines Vertexprogramms in SH.

programm vorliegen, müssen dem Shader in einem gesonderten Schritt bekannt gemacht werden. Im Folgenden werden Ansätze vorgestellt, die dieses Problem lösen möchten. Sie stellen ein Interface zur Shaderprogrammierung direkt im Hauptprogramm zu Verfügung. Hier können alle Fähigkeiten der eingesetzten, herkömmlichen Programmiersprache verwendet werden (z. B. Objektorientierung, generische Programmierung). Durch einen Übersetzungsmechanismus werden im Hauptprogramm festgelegte Shader in eine Darstellung für die externen Shadersprachen übersetzt. Werte von für den Shader bedeutsamen Variablen werden dabei automatisch weitergegeben.

[MQP02] stellt SH vor, eine API zur Metaprogrammierung von Shadern in C++. Diese Bibliothek stellt eine umfangreiche Sammlung an Datentypen und dazugehörigen Operationen zur Verfügung. Es existieren allgemeine Datentypen wie Fließkommatypen, Vektoren und Matrizen, aber auch spezielle Datentypen, die z. B. für die Ein- und Ausgabewerte eines Shaderprogramms zuständig sind. Diese speziellen Variablentypen werden als *semantische Typen* bezeichnet, eine beispielhafte Liste für die semantischen Variablen eines Vertexprogramms zeigt Tabelle 1. SH ist auf Basis von C++ realisiert. Datentypen werden als Klassen realisiert, die Operationen durch entsprechend überladene Operatoren. Der Anwender kann Objekte der entsprechenden Klassen im C++-Code verwenden, und Berechnungen damit durchführen. Um Bereiche mit Shadercode zu markieren, schließt SH sie in die Funktionsaufrufe `shBeginShader` und `shEndShader` ein. Programmcode 1 in Anhang A zeigt eine beispielhafte Implementierung eines SH Shaders aus [MTP+04]. Dort ist der Einsatz von Objektorientierung und Templates zu erkennen, sowie die semantischen Variablen der einzelnen Shaderprogramme.

Zur Laufzeit sammelt SH alle Shaderbestandteile und übersetzt sie mit einem separaten Compiler[9]. Es gibt *Backends* für verschiedene Zielsysteme. Es existierend unter anderem Backends für die GPU (GLSL) und CPU-

[9] SH arbeitet in zwei Modi. Im *immediate* Modus führt das Hauptprogramm auch den Code innerhalb von Shaderbereichen aus. Im *retained* Modus wird dieser Code den Backends übergeben.

basierte Lösungen. Das Zielsystem führt die Berechnungen an Stelle des Hauptprogramms aus. SH trennt den ggf. objektorientierten Code des C++-Interface klar vom erzeugten Code des Shadersprache. Im Shadercode sind keine objektorientierten Strukturen mehr zu finden.

Die Operatoren und Datenklassen von SH sind so implementiert, dass sie gleichzeitig im unmittelbaren und aufgenommenen Modus verwendet werden können. Somit können Elemente gleichzeitig im Hauptprogramm als auch in der Shaderdeklaration verwendet werden. Dies erlaubt es dem Programmierer, alle Elemente der Programmiersprache C++ zu verwenden, um Shader zu schreiben. So können Objektorientierung, Vererbung, Templates usw. eingesetzt werden.

Abschnitt 3.5 kommt nochmals auf SH zurück, um eine weitere Eigenschaft dieser Bibliothek zu beschreiben. Weitere Informationen über SH sind in [MTP+04, MT04] zu finden. Die anfangs quelloffene Entwicklung [10] von SH wurde mittlerweile eingestellt. Die Firma RAPIDMIND [11] entwickelt das Framework als Teil ihrer Produkte weiter.

[Kuc07] stellt einen weiteren Ansatz zum objektorientierten Design von Shadern vor, der C++ zur Erzeugung von GLSL-Code nutzt. Ziel ist es hierbei, auch in der Ausführung des GLSL-Programms noch eine objektorientierte Kontrolle zu behalten. Es wird angestrebt, Shadercode in Klassen zu kapseln, und mehrere Objekte dieser Klassen anlegen zu können. Einzelne Objekte einer Klasse unterscheiden sich nur in den Werten Ihrer Membervariablen.

Zur Realisierung wird für jede in C++ angelegte Klasse und jedes davon instanziierte Objekt eine Zuordnung von C++-Code zu GLSL-Code hergestellt. Dazu sind folgende Dinge zu leisten:

1. Jeder Funktion einer Klasse muss eine äquivalente GLSL-Funktion zugeordnet werden.

2. Für jede Instanz einer Klasse sind alle Membervariablen als eindeutig benannte Variablen im GLSL-Code anzulegen.

3. Beim Aufruf einer Funktion in GLSL muss bekannt sein, für welches Objekt die Funktion ausgeführt werden soll. Die Funktion ist mit den richtigen, dem Objekt zugeordneten Daten auszuführen.

Um eine Funktion in GLSL zu einer Funktion einer (abgeleiteten) Klasse zuzuordnen, wird ein im ganzen Shadercode eindeutiger Namen vergeben. Objekte erhalten eine Id, wodurch eine eindeutige Benennung der den Membervariablen entsprechenden GLSL-Variablen möglich ist. Um die Zuordnung von Funktionsaufrufen zu den entsprechenden Objektdaten zu

[10] http://libsh.org/. Zugriff: 15.01.2008
[11] http://www.rapidmind.net/. Zugriff: 15.01.2008

ermöglichen, wird ein einfacher *Referenzmechanismus* implementiert. Funktionen werden dabei nicht direkt aufgerufen, sondern über einen *Dispatcherfunktion*. Die Dispatcherfunktion wird auch als *Kontrollshader* bezeichnet. Jedem Objekt ist eine einfache Integer-Zahl zugeordnet, welche somit als Referenz dient. Führt man in C++ eine Funktion aus, wird die entsprechende Dispatcherfunktion mit der Referenznummer des Objekts aufgerufen. Die Dispatcherfunktion ruft die eigentliche Funktion mit den dem Objekt entsprechenden Daten als Parameter auf. Dies wird über eine Liste bedingter Anweisungen realisiert. Deren Ausführung verursacht durch Ausnutzung von Static Branching nur geringen Aufwand. Ein entsprechendes Codebeispiel gibt Programmcode 2 in Anhang A.

3.5 Kombinierbare Shader

Eine weitere Klasse von Ansätzen zur Erstellung von Shaderprogrammen beschäftigt sich mit der *Kombination* von Shadern. Hierbei sind schon die einzelnen Teile vollwertige Shader, deren Effekte kombiniert werden sollen.

[MTP+04] stellt auf Basis von SH das Konzept der *Shaderalgebra* vor. Dazu wird für jeden Bereich mit Shadercode ein Objekt angelegt, dem Eingabe- und Ausgabeparameter des dargestellten Shaders zugeordnet sind. Es handelt sich um die Eingaben und Ausgaben des Shaderprogramms. Anzahl, Reihenfolge und Typ der Parameter bestimmen die *Signatur* eines Shaderprogramms. Shaderalgebra behandelt Shaderprogramme als Elemente, auf denen Operationen ausgeführt werden können. Operationen verknüpfen zwei Programme, und erzeugen so ein neues Programm. Programme und Operationen bilden eine *Algebra*.

Für SH werden zwei Verknüpfungsoperationen definiert. Der *Verbindungsoperator* verknüpft zwei Programme, sodass sie hintereinander ausgeführt werden. Die Ausgabewerte des ersten Programms sind dabei die Eingabewerte des zweiten Programms. Anzahl und Reihenfolge dieser Werte müssen übereinstimmen, und eine Kompatibilität muss zwischen den Datentypen existieren.

Der *Kombinationsoperator* verknüpft zwei Programme so, dass sie parallel ausgeführt werden. Zwischen diesen Programmen existieren keine Abhängigkeiten, und bei der Verknüpfung wird durch entsprechende Einrichtung von *Scopes* eine gegenseitige Beeinflussung verhindert.

Um zwei Programme mit dem *Verbindungsoperator* verknüpfen zu können, müssen sie exakt zusammenpassen. Um diese starke Einschränkung abzuschwächen, stellt SH für die Shaderalgebra eine Menge von *Manipulatoren* zur Verfügung. Diese erlauben es, die Anzahl und Reihenfolge der Ein- und Ausgabeparameter eines Shaderprogramms zu verändern. Auch können mit ihrer Hilfe elementare Programme, z. B. für einen Texturzugriff, erstellt werden. Durch Hinzufügen, Entfernen und Verändern der Signatur

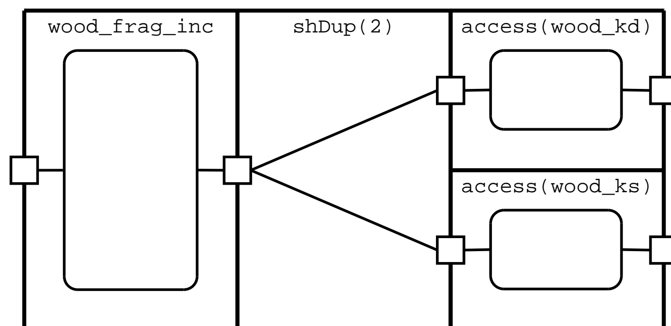


Abbildung 24: Grafische Repräsentation für das Beispiel der Shaderalgebra aus Abbildung 3. Vgl. [MTP+04]

eines Programms, und der Verknüpfung verschiedener Programme, können Effekte kombiniert und verfeinert werden.

Shaderalgebra modelliert die Operatoren über Datenabhängigkeit von Parametern. Dabei werden die Parameter des resultierenden Programms mit denen der Eingabeprogramme, sowie die Parameter der Eingabeprogramme untereinander in Verbindung gesetzt. Shaderalgebra realisiert also eine Funktionalität, die der datenflussbasierten Beschreibung von Shadern als Graphen sehr ähnlich ist. Diese Funktionalität ist aber auf C++-Programmebene nutzbar, und verzichtet zugunsten der algebraischen Verknüpfungssichtweise auf visuelle Programmierung. Durch die rigiden Regeln der Verknüpfungsoperatoren, und das Fehlen eines visuellen Editors, geht jedoch der Vorteil der intuitiven und guten Übersicht über den erzeugen Shader verloren. Andererseits sind für den Einsatz von Shaderalgebra auch keine zusätzlichen Entwicklungswerkzeuge notwendig.

Ein Beispiel für die Anwendung von Shaderalgebra in SH zeigt Programmcode 3 in Anhang A. Der dort gezeigte Vorgang wird in Abbildung 24 grafisch dargestellt. Es ist zu erkennen, dass der Vorgang der Verknüpfung von Shaderprogrammen auf Codeebene komplex ist, und durch eine visuelle Beschreibung wesentlich einfacher zu erfassen ist. Beim gesamten Prozess der Kombination von Shadern mit Hilfe der Shaderalgebra sind viele Details zu beachten. [MTP+04] gibt einen umfassenden Überblick.

Einen andersartigen Weg zur Kombination von Shaderprogrammen gehen [Har04, McG05]. Hier wird die Grundannahme getroffen, dass es eine endliche, bekannte Menge an Codeteilen für Shaderprogramme gibt. Jeder Codeteil steht für eine Berechnung oder einen Effekt. Durch Kombinationen der Fragmente lassen sich verschiedene Programme erstellen. Um die Kombination zur Laufzeit des Shaderprogramms zu ermöglichen, wird ein *SuperShader* genannter Kontrollshader erstellt[12]. Dieses enthält alle Frag-

[12]Die Technik des Kontrollshaders wurde bereits in Abschnitt 3.4 erwähnt

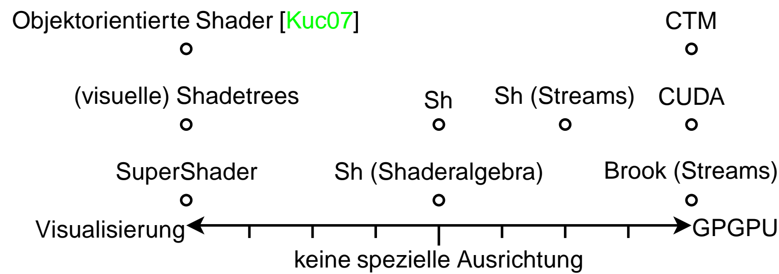


Abbildung 25: Ausrichtung der vorgestellten Ansätze zwischen Visualisierung und allgemeinen Berechnungen.

mente als Funktionen, und eine Zuordnung der Fragmente zu den einzelnen Programmen in Form einer Tabelle boolscher Variablen. Bei der Ausführung des Kontrollshaders kann gewählt werden, welches der Programme ausgeführt werden soll. Das aktive Shaderprogramm muss dazu nicht ausgetauscht werden, was zu Geschwindigkeitsverlusten führen würde. Durch Ausnutzung von Static Branching sind die Laufzeitkosten für den Kontrollshader minimal. Zur Realisierung des SuperShader-Ansatzes werden zwei zentrale Ideen verwendet:

1. Es wird die Grundannahme getroffen, dass ein Shader in eine lineare Abfolge typischer Operationen aufgeteilt werden kann (Operationen wie Beleuchtung, Transformation, ...).

2. Fragmente kommunizieren über eine globale Datenstruktur, die jedem Code-Fragment bei der Ausführung übergeben wird. Eingabewerte werden aus dieser Struktur gelesen, Ausgabewerte hineingeschrieben.

[FW04, TD07] erweitert die Grundidee um ein *Shader-Managementsystem*, welches die Funktionen zu den Fragmenten, den Kontrollshader und die Steuertabelle automatisch erstellt. [TD07] ordnet Codeteile zudem typischen Operationen zu (*Prototypen*). Programmcode 4 in Anhang A zeigt ein Beispiel für einen SuperShader.

Die Grundannahme der SuperShader, das Shaderprogramme aus einer endlichen Menge bekannter Fragmente bestehen, trifft vor allem für Beleuchtungsshader zu. Die genannten Veröffentlichungen zu SuperShadern kommen dementsprechend aus dem Umfeld der Visualisierung und Spieleentwicklung.

3.6 Einsatzzweck und Eignung

Ansätze zur Beschreibung von Shadern haben unterschiedliche Ausrichtungen. Einige zielen auf die Erstellung von Shadern im Visualisierungskontext, andere auf allgemeine Berechnungen im Sinne der GPGPU. Ansätze wie SH beschränken sich nicht auf einen Kontext, und haben keinen speziellen Anspruch. Typisch für GPGPU-Ansätze ist hoher Abstraktionsgrad von der Grafikhardware, sowie umfangreiche Unterstützung beim Transfer der Daten auf und von der Grafikkarte. Bei Visualisierungseinsatz ist der Abstraktionsgrad geringer, der Nutzer hat Kontrolle über die Eigenheiten der Grafikhardware. In diesem Fall ist das primäre Ziel, eine für die Beschreibung von Geometrie- und Beleuchtungsalgorithmen passende Strukturierung des Programmcodes bereitzustellen. Abbildung 25 zeigt eine Einordnung der vorgestellten Ansätze nach der in den Veröffentlichungen genannten Ausrichtung. Die Stream-Ansätze von SH und BROOK werden unterschiedlich bewertet, weil BROOK komplett von der Grafikhardware abstrahiert, SH jedoch nicht. Es bleibt anzumerken, dass sich für die vorgestellten Ansätze mit großer Wahrscheinlichkeit auch Anwendungsfälle außerhalb des primären Anwendungsfelds finden lassen.

Shaderprogramme für ältere Grafikkarten (vor Shader Model 4.0) sind in der Zahl ausführbarer Operationen beschränkt [Micd]. Durch den Metaprogrammierungsansatz und die damit verbundene automatische Code-Generierung können schnell Shader entstehen, die die maximale Länge überschreiten. [CNS+02, FHH04, Hei05] stellen Ansätze zur automatischen Aufteilung solcher Shaderprogramme auf mehrere Renderdurchgänge vor (*Shaderpartitioning*).

4 Entwicklungswerkzeuge bei MeVis Research

Dieses Kapitel gliedert sich in zwei Teile. Zunächst wird MEVISLAB vorgestellt, die bei MEVIS RESEARCH eingesetzte Entwicklungsplattform, sowie der darin enthaltene Volumenrenderer. Im Anschluss werden die Ergebnisse einer Umfrage unter MEVIS-Mitarbeitern im Hinblick auf die gestellten Anforderungen an Volumenrendering dargestellt.

4.1 Das Entwicklungsframework MeVisLab

Im Forschungskontext ist es wünschenswert, Algorithmen und Prototypen in kurzer Zeit entwickeln und testen zu können. MEVISLAB ist eine Plattform für Rapid Prototyping, die für Aufgaben der Bildverarbeitung und Visualisierung umfangreiche Werkzeuge anbietet. Besonderer Fokus ist dabei der medizinische Kontext. MEVISLAB wird von der MEVIS RESEARCH GMBH entwickelt. Wichtige Teile der in dieser Arbeit vorgestellten Software-Entwicklung nutzen MEVISLAB. Im Folgenden wird die Grundstruktur der Entwicklungsumgebung vorgestellt. Besondere Aufmerksamkeit wird dem integrierten Volumenrenderer gewidmet.

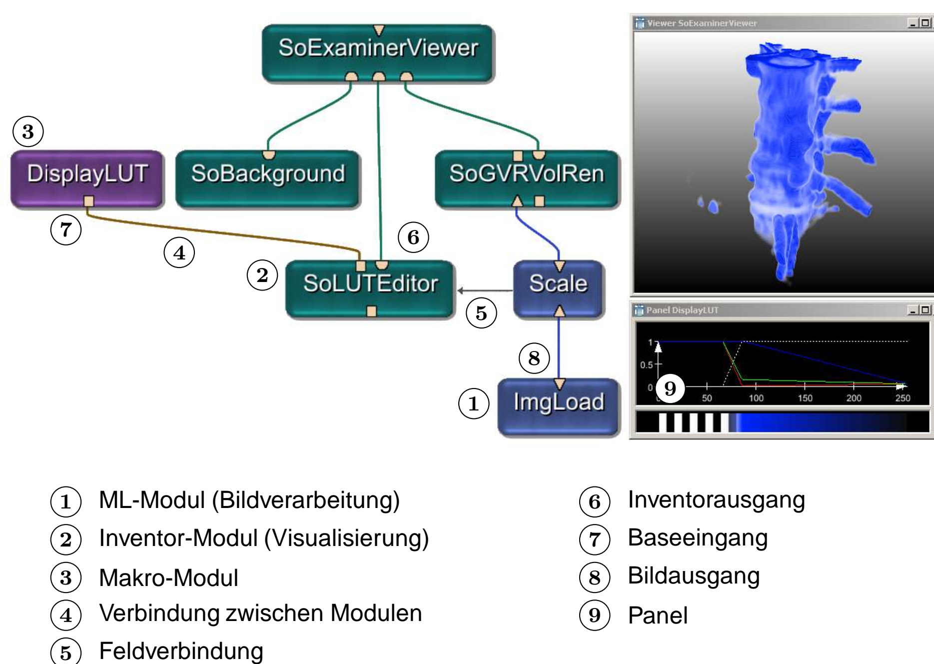


Abbildung 26: Bestandteile eines MEVISLAB-Netzwerks.

4.1.1 Aufbau

MEVISLAB stellt eine große Sammlung grundlegender Algorithmen und Datenstrukturen der medizinischen Bildverarbeitung und Visualisierung zu Verfügung. Sie sind unabhängig voneinander realisiert und dienen als Grundelemente zur Erstellung komplexer Anwendungen mit Hilfe visueller Programmierung. Eine Anwendung wird strukturiert durch:

Netzwerk: Ein Netzwerk kapselt die Grundelemente einer mit MEVISLAB erstellten Anwendung: *Module* und *Verbindungen*. Innerhalb eines Netzwerks sind Instanzen von Modulen eindeutig benannt.

Modul: Ein Modul kapselt eine Berechnung oder Datenstruktur. Es werden grundlegende Modultypen zur *Bildverarbeitung* und zur *Visualisierung* unterschieden. Die unterschiedlichen Modultypen werden weiter unter in diesem Abschnitt beschrieben. Ein Modul besitzt *Ein- und Ausgänge* zum Datenaustausch, über die es mit anderen Modulen zu größeren Einheiten zusammengeschlossen werden kann. Die Modultypen haben spezifische Ein- und Ausgangstypen. Das Verhalten eines Moduls kann über *Felder* genannte Parameter gesteuert werden. Felder kapseln Variablen eines bestimmten Typs, und erweitern sie um weitere Funktionalität wie Speicherbarkeit (*Persistenz*) und Verknüpfbarkeit. Vorhandene Feldtypen in MEVISLAB sind vielfältig, es können unter anderem numerische Werte, Vektoren und Strings verwendet werden. Inhaltlich zusammengehörende Module werden in *Projekten* gruppiert.

MEVISLAB-Module werden in C++ implementiert. MEVISLAB erlaubt die Integration eigener Module durch die Implementation entsprechender Klassen auf C++-Ebene.

Innerhalb eines Netzwerks kann ein Modul durch mehrere Instanzen vertreten sein. Für jede Instanz gibt es eine grafische Repräsentation. Diese zeigt den Namen des Moduls und einen eindeutigen Instanznamen an. Ein- und Ausgänge sind ebenfalls sichtbar, sowie ggf. vorhandene Verbindungen. Auf die Felder kann über ein eigenes Fenster (*Panel*) zugegriffen werden.

Verbindungen: Zusammenhänge zwischen Modulen und ihren Parametern werden über Verbindungen zwischen Ein- und Ausgängen identischen Typs hergestellt. *Feldverbindungen* verknüpfen zwei Felder miteinander.

Für verschiedene Einsatzzwecke gibt es Modultypen, deren Verhalten sich grundlegend voneinander unterscheidet:

Bildverarbeitungsmodule/ML-Module: Für die Verarbeitung von Bilddaten wird die MEVIS IMAGE PROCESSING LIBRARY (ML) verwendet

[Spi]. Ein-/Ausgänge tragen Bilddaten und werden als Dreieck dargestellt. Der weitere Ein-/Ausgangstyp *Base* erlaubt die Weitergabe beliebiger Daten. Bildverarbeitungsmodule arbeiten nach dem Datenflussprinzip. Die Verarbeitung erfolgt demand-driven: Nur wenn das Ergebnis verlangt wird, fängt ein ML-Modul an zu arbeiten. MEVISLAB ist also eine Plattform zur visuellen Datenflussprogrammierung. Die Berechnungen werden vom sequentiellen C++-Code ausgeführt, so dass der Datenfluss nur zur Strukturierung der Berechnungen eingesetzt wird (Hybrid Dataflow).

Visualisierungsmodule/Inventormodule: Zur Datenvisualisierung wird OPEN INVENTOR eingesetzt, eine quelloffene Bibliothek zur Erzeugung von 3D-Grafik [Wer94a, Wer94b]. Es basiert auf OpenGL und strukturiert Geometrie, Texturen und weitere Ressourcen in einem *Szenengraph*. Der Szenengraph wird zur Darstellung mit Tiefensuche traversiert, und die Knoten in entsprechender Reihenfolge angezeigt. Ein Inventormodul in MEVISLAB entspricht einem Knoten im Szenengraph von Open Inventor. Mit Ein- und Ausgängen für Inventorknoten (dargestellt als Halbkreise) können Module dieses Typs zu baumartigen Strukturen verknüpft werden, wodurch der Szenengraph definiert wird. Inventormodule können ebenfalls Ein-/ausgänge vom Typ Base enthalten.

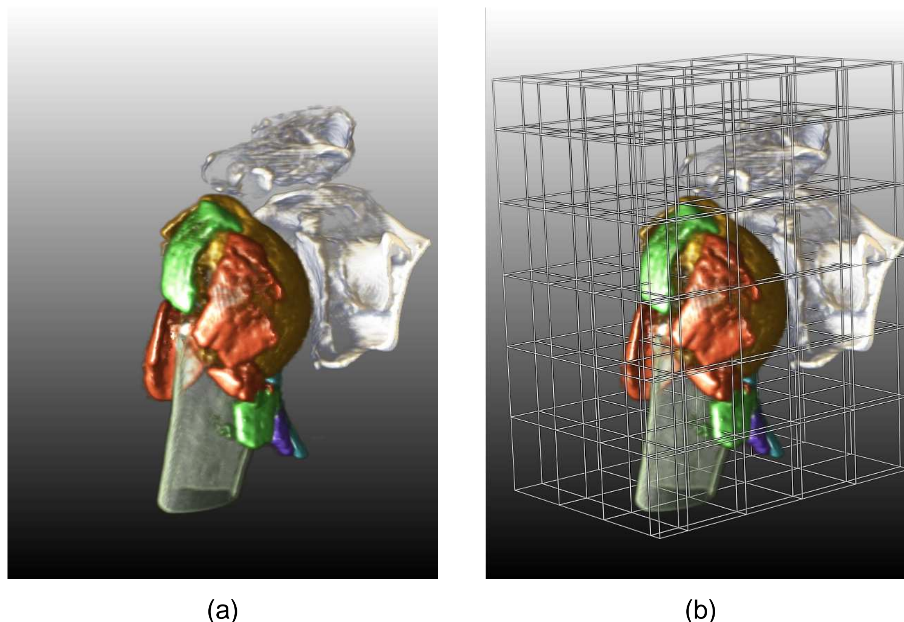
Makromodule: Ganze Netzwerke lassen sich durch Makromodule repräsentieren. Somit ist eine Strukturierung auf mehreren Ebenen möglich, und die Übersicht wird verbessert.

Abbildung 26 zeigt Beispiele für die vorgestellten Elemente. Makromodule und die grafische Oberfläche der Module können über die MEVISLAB MODULE DEFINITION LANGUAGE (MDL) [MeV] festgelegt werden. Die Skriptsprache PYTHON [13] erlaubt die Steuerung der Module auf Netzwerkebene.

4.1.2 Der Giga Voxel Renderer (GVR)

Für die Darstellung von Volumendaten steht in MEVISLAB der GIGA VOXEL RENDERER (GVR) zur Verfügung. Der Renderer basiert auf einem Textureslicing-Algorithmus. Er ermöglicht die Darstellung hoch aufgelöster Volumen (512^3 und mehr) auf programmierbarer Grafikhardware. Der GVR nutzt mehrere Auflösungsstufen des Datensatzes, die in einem Octree organisiert sind. Geringere Stufen erfordern zur Darstellung weniger Rechenleistung und Speicherplatz auf der Grafikkarte. Abhängig von der Entfernung der Kamera, dem darzustellenden Bereich des Volumens (*Volume of*

[13] http://www.python.org/. Zugriff: Zugriff: 15.01.2008



(a) (b)

Abbildung 27: Volumenrendering mit dem GVR. (a) CT-Aufnahme einer Schulter mit Per-Object Shading. (b) Selbe Darstellung mit Einblendung des Octrees. Beide Bilder wurden mit Hilfe eines Beispielnetzwerks von MEVISLAB erstellt.

Interest) und den Ressourcen des System wird eine Auflösungsstufe gewählt. Dieser *Multiresolution* Ansatz wird in [LHJ99] vorgestellt.

Der GVR besteht aus zwei Teilen. Die Darstellungsfunktionalität ist als externe Bibliothek unabhängig von MEVISLAB realisiert (Projekt GVRLIB). In MEVISLAB steht eine grafische Benutzerschnittstelle aus Inventor-Modulen zur Verfügung (Projekt SOGVR). Die GVRlib ist auf Basis von OpenGL erstellt und nutzt auf aktuellen Grafikkarten GLSL-Programme. Diese werden abhängig von den Einstellungen beim Rendern jedes Frames neu erstellt und auf die Grafikkarte geladen.

Der GVR stellt eine Vielzahl von Darstellungsfunktionen zur Verfügung. Eine Auswahl wird im Folgenden vorgestellt:

Shading: Neben der Beleuchtung mit einem Blinn-Phong Beleuchtungsmodell sind erweiterte Effekte möglich. *Toneshading* verbessert die Tiefenwahrnehmung, *Silhouette-Enhancement* und *Boundary-Enhancement* verbessern den Sichteindruck von Konturen teilweise transparenter Objekte.

Beschleunigung: Der GVR arbeitet in zwei Modi. Bei Bewegung des Volumens wird die Qualität zugunsten der Darstellungsgeschwindigkeit

verringert (*interaktiver* Modus). Bei unveränderter Darstellung wird die Qualität erhöht (*statischer* Modus). Octreeknoten werden nicht dargestellt, wenn sie keine sichtbaren Elemente enthalten (*Empty Space Skipping*) oder sich außerhalb des sichtbaren Bereichs befinden (*Culling*).

Per-Object Shading: Liegt eine Segmentierung des Datensatzes in verschiedene Bereiche vor, so kann das Gewicht der Beleuchtungseffekte pro Objekt angepasst werden (*Tagged Rendering*). Die Gewichte werden in einer Textur übergeben [MK06].

Maskenvolumen: Abhängig von einem zweiten Volumen (*Maske*) kann die Darstellung des Hauptvolumens verändert werden.

Slabrendering: Es ist möglich, dünne Schichten beliebiger Orientierung (*Slabs*) aus einem Datensatzes zu rendern.

Flexibles Compositing: Der GVR unterstützt verschiedene Compositingverfahren (Back-to-front, Front-To-Back, MIP, . . .).

Gradienten: Es werden vorberechnete Gradienten und On-the-fly Gradienten unterstützt.

Lookuptabellen (LUTs): Es können mehrdimensionale Lookup-Tabellen für die Transferfunktion verwendet werden.

Picking: Dargestellte Objekte können selektiert werden.

Clipping: Das Volumen kann durch *Clipebenen* beschnitten werden.

Debug und Diagnosis: Interne Einstellungen des GVRs können durch entsprechende GUI-Module eingesehen und verändert werden. Ein eigenes, vollständiges Shaderprogramm kann eingestellt werden, welches den internen Shader überschreibt.

Eigenes Datenformat: Der vom GVR erstellte Octree kann abgespeichert und geladen werden. Dies ermöglicht es, die Vorverarbeitung zur Erstellung des Octrees nur einmal ausführen zu müssen.

Depthpeeling: Erlaubt die Darstellung transparenter Geometrie innerhalb des GVR-Volumens mit korrekter Sortierung der transparenten Flächen [NK03].

Darstellung von Bildfolgen: Ein Bildatensatz in MeVisLab kann zeitlich aufeinanderfolgende Bilder enthalten. Der GVR unterstützt die Darstellung dieser Bilder durch das Erzeugen eines Octrees pro Einzelbild.

4.2 Volumenrendering bei MeVis: Eine Bestandsaufnahme

Im Folgenden werden die Ergebnisse einer Umfrage zum Thema „Verwendung von Volumenrendering bei MEVIS“ beschrieben.

4.2.1 Überblick und Motivation

Die Erstellung dieser Arbeit bei MEVIS RESEARCH hat ermöglicht, den Einsatz von Volumenrendering in einem typischen Anwendungsumfeld zu untersuchen. Anforderungen und Probleme ergeben sich aus dem Kontext echter Projekte. Es wurde versucht, einen Gesamteindruck der gegebenen Verhältnisse zu erarbeiten, wobei Shaderprogrammierung nur einen Teilbereich der gestellten Fragen ausmachte. Damit wurde sichergestellt, dass eine Entscheidung über die Relevanz für diese Arbeit erst nach Erhebung aller Einflussfaktoren getroffen wurde.

Die Bestandsaufnahme hat zwei zentrale Anliegen. Zum einen soll die Verwendung von Volumenrendering, welche bei MEVIS RESEARCH der Verwendung des GVR entspricht, möglichst breit dargestellt werden. Zum anderen soll eine Einordnung und Bewertung dieser Arbeit in die bei MEVIS vorliegende Umgebung erreicht werden. Anforderungen an das erarbeitete Konzept sollen identifiziert werden. Die Sammlung empirischen Daten zur Motivation dieser Arbeit wurde nicht angestrebt.

Zur Erstellung des angestrebten Profils wurden 17 Mitarbeitern von MEVIS RESEARCH und ein Mitarbeiter von MEVIS TECHNOLOGY befragt[14]. Dafür wurde ein Fragebogen erstellt, der sich nach folgenden Schwerpunkten gliedern:

- Einsatzgebiet des GVR
- Verwendung und Probleme vorhandener Features
- Generelle Anforderungen und gewünschte Erweiterungen
- Computergrafikkenntnisse der GVR-Anwender/Interviewpartner
- Workflow mit dem GVR
- Sonstiges

Der Fragebogen ist in Anhang B beigefügt. Während der Erstellung dieser Diplomarbeit bei MEVIS RESEARCH wurde eine umfangreichere Auswertung der Interviews verfasst. Diese kann, in anonymisierter Form, auf der beigelegten CD eingesehen werden. Dort sind auch alle Mitschriften zu den Gesprächen bereitgestellt.

[14] Die Firma MEVIS TECHNOLOGY wurde nach der Fertigstellung der Interviews umbenannt in MEVIS MEDICAL SOLUTIONS.

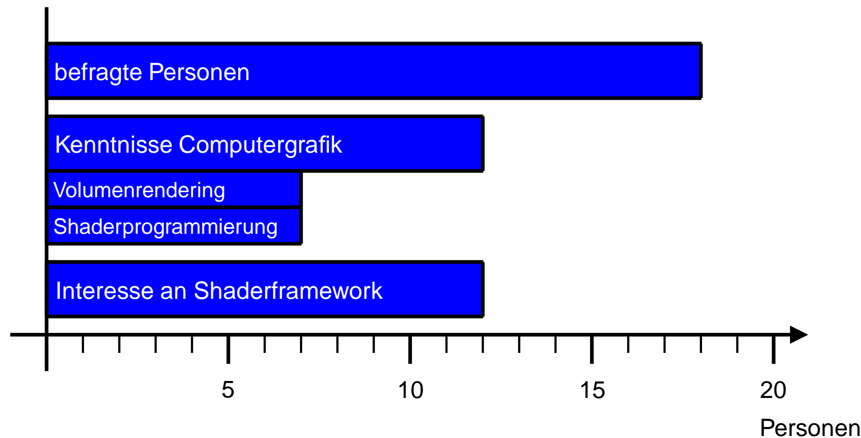


Abbildung 28: Anteil der befragten Personen mit Computergrafik-Kenntnissen.

Unter den Befragten waren 14 Informatiker, 2 Physiker und 2 Mathematiker. Abbildung 28 zeigt den Anteil der Personen, welche Kenntnisse aus dem Bereich der Computergrafik besitzen. Zwei Drittel der Befragten besitzen entsprechende Kenntnisse, gut die Hälfte davon in den Bereichen Volumenrendering und Shaderprogrammierung. Zu den täglichen Aufgaben gehört Computergrafik, Volumenrendering und Shaderprogrammierung aber nur bei wenigen Befragten. Ebenfalls zwei Drittel der Befragten zeigten Interesse, mit einem Shaderframework für Volumenrendering zu arbeiten. Diese Gruppe überschneidet sich fast vollständig mit der Gruppe der Personen mit Computergrafikkenntnissen.

4.2.2 Auswertung

Dieser Abschnitt fasst die Ergebnisse der Gespräche nach den Themenschwerpunkten geordnet zusammen.

4.2.2.1 Einsatzzweck des GVR

Volumenrendering mit dem GVR wird in den meisten Projekten eingesetzt. Einsatzzweck und Wichtigkeit sind dabei sehr unterschiedlich. Der folgende Abschnitt gibt einen Überblick über die Bandbreite der Verwendungen.

Darstellungsarten: Der GVR wurde zur Erzeugung von direkten Volumendarstellungen entwickelt. Er findet aber auch Anwendung in der Erzeugung von 2D-Overlays, z. B. für die Erstellung von zweidimensionalen Schnittbildern durch ein Volumen (*Multi Planar Reformatting (MPR)*, eine weiter Möglichkeit, ein Volumen visuell zu explorieren). Er bietet hier Geschwindigkeitsvorteile, da die teure Reformatierung der Datensätze auf

diese Weise enorm beschleunigt wird. Ein Drittel der Befragten verwendet den GVR zur direkten Darstellung von Volumen, ein weiteres Drittel zur Erzeugung von MPR-Ansichten. Für das letzte Drittel ist der GVR nicht essentiell für die Arbeit.

Direktes Volumenrendering der medizinischen Datensätze und die Darstellung in MPR-Ansicht ergänzen sich im Einsatz bei MEVIS. Beide Darstellungsarten haben spezifische Vorteile. Die Orientierung im Datensatz fällt im Volumenrendering leichter, die genaue Bearbeitung von Befunden ist in der MPR besser zu leisten. Im gemeinsamen Einsatz fällt der Koordination beider Darstellungsarten besonderes Gewicht zu. So soll z. B. eine Selektion in 2D auch in 3D angezeigt werden, und umgekehrt. Im realen Umfeld der Kliniken ist die Verwendung von direktem Volumenrendering noch nicht weit verbreitet, doch in den Forschungsprojekten bei MEVIS wird intensiv an klinisch sinnvollen Einsatzmöglichkeiten geforscht.

Dargestellte Informationen: In den meisten Fällen werden mit dem GVR medizinisch wichtige Informationen dargestellt. Die Identifizierung und Hervorhebung von Befunden ist ein zentrales Anliegen der Projekte. Die Befunde sollen zur Operationsplanung und Operationsdurchführung leicht zugänglich gemacht werden. Auch die Darstellung von Risiko- und Sicherheitsbereichen oder die Positionierung von OP-Werkzeugen sind häufig auftretende Visualisierungsaufgaben. Der Volumenrenderer wird weiterhin als Kontrollwerkzeug eingesetzt, um synthetische Daten oder Bildverarbeitungs-Ergebnisse darzustellen.

Mit dem GVR werden praktisch alle bei MEVIS eingesetzten Datentypen angezeigt. Diese unterscheiden sich im Hinblick auf Aufnahmeart (CT, MRT, . . .), Anzahl der Kanäle (Grauwertbilder, Farbbilder), Auflösung, und dargestellte Informationen (anatomische oder funktionale Daten). Die mögliche Auflösung steigt immer weiter an, da die verfügbaren Aufnahmegeräte immer besser werden. Zudem werden häufig mehrere Datensätze gleichzeitig verwendet, z. B. zur Darstellung verschiedenartiger Aufnahmen desselben Objekts (*multimodaler* Daten). In Kombination kann so eine sehr große Datenmenge anfallen. Hier sind die Beschleunigungsmechanismen des GVR für Haupt- und Grafikkartenspeicher von großer Bedeutung.

4.2.2.2 Einsatz der Funktionen des GVR

Der GIGA VOXEL RENDERER besitzt eine Vielzahl an Features und Einstellungsmöglichkeiten (siehe Abschnitt 4.1.2). Dabei werden bei MEVIS praktisch alle vorhandenen Features eingesetzt, wobei ausgefallenere Fähigkeiten seltener Verwendung finden als Grundlegende. Gravierende Probleme mit vorhandener Funktionalität sind selten.

Verwendung: Viele Projekte verwenden nur grundlegende Features des GVR, andere setzen fast alle Fähigkeiten ein. Es lassen sich zentrale Featu-

Feature	Nennungen
LUT	16
Geschwindigkeitsmodi	10
Beleuchtung	10
Per-Object Shading	8
Boundary-Enhancement	8
Slabs	8
Toneshading	6
Silhouette-Enhancement	6
Maskenvolumen	6

Feature	Nennungen
Debug-Modul	5
Clipping	4
MIP	4
Diagnosis-Modul	3
Eigener Shader	3
Picking	3
Depthpeeling	2
GVR Datenformat	2
Bildfolgen	1

Tabelle 2: Häufigkeit des Einsatzes von GVR Features nach Nennungen in den Interviews.

res identifizieren (siehe Tabelle 2). Die größte Bedeutung kommt den Lookuptables zu, die immer Verwendung finden.[15]. Weiterhin oft eingesetzt werden Beleuchtungseffekte und die verschiedenen Geschwindigkeitsmodi. Für die Erzeugung von 2D-Overlays wird oft das Slabrendering verwendet.

Die Gesprächspartner haben einen guten Überblick über die vorhandenen Features, und nehmen sie größtenteils als intuitiv wahr. Die Dokumentation des GVR wird meist als ausreichend empfunden. Welche Fähigkeiten des GVRs Anwendung finden, hängt hauptsächlich von den Notwendigkeiten der Problemstellung ab. Eine Ausnahme ist hier die Depthpeeling-Funktion. Eine solche Funktion zum korrekten Darstellen von teilweise transparenter Geometrie innerhalb des GVR Volumenrenderings wird oft benötigt. Die Performanceeinbuße beim Einsatz von Depthpeeling ist jedoch meist zu groß, sodass Depthpeeling selten eingesetzt wird (siehe Tabelle 2).

Probleme: Beim Einsatz des GVR kommt es sehr selten zu gravierenden Problemen. Der GVR funktioniert stabil, nur beim Einsatz des GVRs in sehr komplexen MeVisLab-Netzwerken kommt es zu Problemen. In diesem Fall werden beispielsweise Einstellungen für die Geschwindigkeitsmodi nicht korrekt übernommen, oder einzelne Octreeknoten eines Volumens werden nicht richtig dargestellt. Dies tritt insbesondere auf, wenn mehrere GVRs verwendet werden. Umfangreichere Probleme sind oft auf nicht vorhandene Funktionalität zurückzuführen. Jeweils ein Drittel der befragten gibt keine, geringfügige oder Probleme durch fehlende Funktionalität mit dem GVR an.

[15]Dies ist erwartungskonform, da die LUT die Transferfunktion beschreibt und die visuellen Eigenschaften zu den gemessenen Signalen definiert.

4.2.2.3 Anforderungen an den GVR

In diesem Abschnitt wird untersucht, welche Anforderungen an den GVR existieren, welche neuen Funktionen die Gesprächspartner sich für den GVR wünschen, und welche Ansprüche an die Geschwindigkeit und Darstellungsqualität gemacht werden.

Wünschenswerte Features: In den Gesprächen ließen sich vier zentrale Anforderungen an den GVR identifizieren:

1. Die Unterstützung einer flexiblen Anzahl von Volumen und LUTs. Es soll möglich sein, mehrere Masken-, Tag-, Datenvolumen oder LUTs zu verwenden.

2. Einfachere und stabilere LUTs. Sie sollen intuitiver generiert werden können. Dies trifft insbesondere auf die 2D-LUTs für das Tagged Volumerendering zu. Weiterhin sollen LUTs auch datensatzübergreifend verwendbar bleiben (z. B. adaptive LUTs).

3. Unterstützung und Bereitstellung von gesammelten Voreinstellungen für alle Einstellungsparameter (*Presets*). Obwohl einzelne Features intuitiv bedienbar sind, wird die Menge der Einstellungsmöglichkeiten des GVR insgesamt meist als zu groß empfunden.

   Features werden oft nur dann als wirklich gut empfunden, wenn man sie nicht pro Datensatz einstellen muss. Es existiert der Wunsch nach datensatzübergreifend funktionierenden Vorbelegungen der gesamten GVR-Einstellungen (inkl. LUT). Die Presets sollen dabei eine Darstellung ermöglichen, die für den Anwendungsfall wichtigen Informationen geeignet darstellt. Weiterhin soll die Darstellung „gut aussehen". Diese Anforderung ist eng verwandt mit der LUT-Anforderung.

4. Schnelle und korrekte Darstellung von transparenter Geometrie innerhalb des GVR Volumens.

Geschwindigkeit und Qualität: Die Performance des GVR bei der Darstellung wird allgemein als gut empfunden, nur die oft lange Dauer der Vorverarbeitung beim Ladevorgang (Erstellung des Octrees) stellt oft ein Zeitproblem dar[16]. Oft genannt wurde die relativ unscharfe Anforderung, dass Bilder in ausreichender Geschwindigkeit mit ausreichender Qualität erstellbar sein sollen. Viele der Befragten haben eher Geschwindigkeitsprobleme (z. B. aufgrund des Ladens der Daten) als Qualitätsprobleme.

Die meisten Befragten setzen das Feature der verschiedenen Geschwindigkeitsmodi ein, um interaktive Geschwindigkeiten zu ermöglichen, ohne

[16]Ein weiteres Problem mit dem Einsatz des GVR in dieser Hinsicht ist, dass viele Kliniken oft keine genügend leistungsfähige Hardware besitzen.

auf hochqualitative Bilder zu verzichten. Es gibt aber auch Anwendungsfälle, bei denen sich durch die Bewegung die Darstellungsqualität nicht ändern soll[17]. Die generelle Möglichkeit hochqualitativer Bilder wird als unverzichtbar eingestuft[18].

4.2.2.4 Workflow

Der folgende Abschnitt behandelt den Weg von einer Problemstellung der Visualisierung zur Lösung mit Hilfe des GVR. Im Zentrum des Interesses steht, inwiefern die Suche nach einer optimalen Lösung von vorhandenen Features des GVR beeinflusst wird, und wie häufig Weiterentwicklungen eine Möglichkeit darstellen.

Entwicklungsprozess: Die Befragten versuchen stets, unabhängig von vorhandener Funktionalität eine optimale Lösung zu identifizieren. Auf Basis der Ressourcen und des Zeitplans des Projekts wird ein Kompromiss zwischen Neuentwicklung und Nutzung vorhandener Features gesucht. Dabei wird auch die Zielvorstellung entsprechend angepasst. Oft kommt es vor, dass die Ressourcen in den Projekten keine Zeit für Eigenimplementationen lässt, so dass nur die vorhandenen Werkzeugfähigkeiten ausgeschöpft werden können[19].

Neue Darstellungseffekte für den GVR: In dieser Arbeit wird ein Shaderframework entwickelt, mit dem der GVR mit neuen Darstellungseffekten versehen werden kann. Zwei Drittel der Befragten zeigten Interesse an einem solchen Framework für den GVR, und könnten sich vorstellen, damit zu experimentieren (siehe Abbildung 28). Die Befragten haben jedoch meist in der Projektarbeit keine Ressourcen, um den GVR mit Hilfe eines Shaderframeworks zu erweitern. Die meisten Befragten wenden sich mit Wünschen für neue Effekte an die Programmierer des GVR, und geben diese als *Featurerequest* weiter. In den Gesprächen entsteht der Eindruck, dass ein Framework zur Manipulation der Grafikkartenshader des GVR ein Tool ist, welches hauptsächlich für GVR- und Computergrafik-Experten interessant wäre. Diese hätten ein weiteres Tool, mit dem GVR zu arbeiten.

4.2.3 Zusammenfassung

Der GVR wird als Werkzeug für Visualisierungsaufgaben bei MEVIS in fast allen Projekten eingesetzt. Direkte Volumenvisualisierung und Darstellung

[17] Ein Beispiel ist die manuelle *Registrierung* [HHH01].

[18] Es gibt Anwendungen, für die höchste Qualität notwendig ist, z. B. Gefäßdarstellung oder Werberenderings.

[19] Visualisierung ist oft nicht der Schwerpunkt der Projekte, und andere Probleme sind zentraler. Der beschriebe Workflow trifft auch auf andere Problemstellungen zu, wie z. B. Bildverarbeitungsthemen.

einzelner 2D-Schichten sind die zwei großen Einsatzgebiete. Die Visualisierung selbst hat häufig keine hohe Priorität, obwohl sie ein unverzichtbarer Bestandteil der Arbeit von MEVIS ist. Insgesamt ist der GVR zu einem unverzichtbaren Werkzeug geworden.

Der GVR bietet eine umfangreiche Sammlung von Features, die auch alle genutzt werden. Es lassen sich zentrale Funktionen ausmachen, mit denen der GVR meistens eingesetzt wird. Hierzu gehören vor allem Lookuptables, aber auch die Geschwindigkeitsmodi und Beleuchtungseffekte.

Als problematisch wird vor allem die Arbeit mit den Lookuptables und die Menge der Einstellungen empfunden. Hier besteht der Wunsch, datensatzübergreifend funktionierende Voreinstellungen (Presets) verwenden zu können. Das Festlegen von Einstellungen und LUTs soll intuitiver werden. Weitere Probleme bestehen in der teils langen Vorverarbeitungszeit beim Laden von Bildern, und Fehlern bei Verwendung des GVRs in sehr großen Netzwerken. Neben der Verbesserung dieser Dinge wünschen sich die Nutzer die Möglichkeit, mehrere Volumen und LUTs gleichzeitig verwenden zu können. Schnelle und stabile Darstellung von teilweise transparenten Objekten im GVR-Volumen ist eine weitere Anforderung, die häufig auftritt.

Ein Shaderframework für den GVR würde nach Einschätzung der Befragten nur in den Projekten eine Bereicherung darstellen, in denen ein sehr starker Visualisierungsanteil vorhanden ist, und in denen die entsprechenden Ressourcen für eine Beschäftigung mit eigenen Erweiterungen am GVR gegeben sind. Ein solches Framework wäre immer ein gutes Hilfsmittel für Veränderungen und Erweiterungen am GVR durch die GVR-Experten bei MEVIS.

4.2.4 Bewertung

Ein im Rahmen dieser Arbeit entwickeltes Shaderframework zur Manipulation der GVR-Shader erscheint als wertvolles Tool, um die schwerpunktmäßig gewünschten neuen Features wie mehrere Volumen oder LUTs flexibel einsetzen zu können. Neu hinzugefügte Ressourcen gehen meist mit einer Änderung am Shadercode einher. Der flexible Zugriff auf Shaderprogramme und entsprechende Ressourcen erleichtert somit die Integration neuer Features. Viele der in der Umfrage schwerpunktmäßig gewünschten Fähigkeiten erfordern jedoch weitere Neuerungen. So benötigt z. B. ein flexibler Umgang mit LUTs entsprechende Editoren. Ein Shaderframework allein reicht nicht immer aus.

Zielgruppe eines Shaderframeworks sind die GVR- und Computergrafik-Experten bei Mevis. Aus der Umfrage kann geschlossen werden, dass ein solches Framework für die schwerpunktmäßig mit dem GVR arbeitenden Entwickler ein hilfreiches Werkzeug darstellen würde. Anspruch eines solchen Frameworks sollte es nach den Ergebnissen der Umfrage nicht

sein, die Shader des GVRs für jeden einfach zugänglich zu machen. Es ist anzunehmen, dass ein solches Ziel nur durch eine geringe, feste Anzahl an Einflussmöglichkeiten auf den Shader zu erreichen ist. Dies würde für die Wünsche der GVR-Experten nicht ausreichen. Diese sind es jedoch, die bei MeVis schwerpunktmäßig mit dem Volumenrender arbeiten.

5 Entwurf eines modularen Shaderframeworks

Dieses Kapitel behandelt die Konzeption eines modularen Shaderframeworks für Volumenrendering. Zu Beginn werden die gestellten Anforderungen analysiert und bisher vorgestellte Strategien zur Shadererstellung im Hinblick auf die Problemstellung dieser Arbeit bewertet.

5.1 Analyse

Der folgende Abschnitt untersucht die gegebene Aufgabenstellung mit dem Ziel, einen geeigneten Ansatz für ihre Lösung auszuwählen. Dazu wird die Aufgabe eines Shaderframeworks zunächst in den gesamten Darstellungsprozess des Volumenrenderings eingeordnet, um daraufhin relevante Anforderungen herausarbeiten zu können.

5.1.1 Einordnung in ein Volumenrendering-Programm

Die bisherigen Betrachtungen dieser Arbeit ordnen die Bestandteile eines Renderers nach der Zugehörigkeit zu den Aufgabenbereichen der Volumenrendering-Pipeline. Für die weitere Diskussion ist es sinnvoll, eine weitere Sichtweise auf einen Volumenrenderer darzustellen. Die erstellten Shader repräsentieren selbst einen großen Teil der Pipeline (siehe Abschnitt 2.2.3.4), müssen aber dennoch korrekt mit den restlichen Teilen des Programms zusammenarbeiten. Diese übrigen Teile des *Darstellungsvorgangs* eines Volumenrendering-Programms werden zunächst kurz vorgestellt, und die Verwaltung der Shaderprogramme von ihnen abgegrenzt. Auf Basis dieser Betrachtung wird die Aufgabe eines Shaderframeworks skizziert.

5.1.1.1 Abgrenzung anderer Teile des Darstellungsvorgangs

Abbildung 29 gibt einen Überblick über die Bestandteile eines Volumenrendering-Programms. Zu den Aufgaben eines Volumenrendering-Programms gehört die Verwaltung der Parameter des umgesetzten Algorithmus, wie

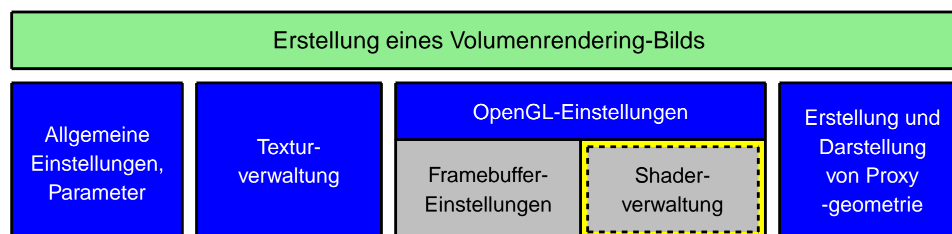


Abbildung 29: Bestandteile eines Programms für Volumenrendering. Diese Arbeit stellt ein Konzept zur Verwaltung der Shaderprogramme vor.

Samplingrate und Transferfunktion. Die Parameter müssen dem Shaderprogramm ggf. mitgeteilt werden. Weiterhin müssen die darzustellenden Volumendaten als Texturen auf die Grafikkarte geladen werden. Bei realistischen Datensätzen, z. B. aus medizinischen Aufnahmeverfahren, fallen große Datenmengen an. Für echtzeitfähige Renderingergebnisse ist daher ein umfangreiches Texturmanagement auf der Grafikkarte notwendig. Aufgabe der Texturverwaltung ist es auch, weitere Datenstrukturen als Texturen auf der Grafikkarte zur Verfügung zu stellen. Dabei kann es sich z. B. um eine Lookuptable mit Werten der Transferfunktion oder Hilfsdaten zum Volumen (wie Histogramminformationen) handeln. Auch die gewählte Funktionalität der Framebuffer-Operationen wie Tiefentest und Alphablending bestimmen maßgeblich das Aussehen des Ergebnisses. Das aktive Shaderprogramm bestimmt das Verhalten der Pipeline und übernimmt die Ausführung zentraler Teile des Renderingalgorithmus. Nachdem alle Texturen geladen und Einstellungen gemacht sind, wird der Renderingvorgang durch die Erzeugung von Proxygeometrie gestartet.

Alle Teile eines Gesamtsystems zur Volumenvisualisierung müssen für die erfolgreiche Darstellung zusammenpassen. Sind diese Einstellungen nicht so wie vom Fragmentprogramm erwartet, wird kein korrektes Bild erzeugt. Shaderprogramme bilden die zentrale Schnittstelle. Sie verbindet Geometrie, Texturen und Bilderzeugung im Framebuffer. Da alle Bestandteile des Darstellungsprozesses eng verzahnt sind, und erst zusammen die Umsetzung eines Algorithmus bilden, müssen bei der Änderung von Shaderprogrammen oft auch die Proxygeometrie oder Framebuffer-Einstellungen verändert werden. Deren Modularisierung ist jedoch die Aufgabe eines modularen Gesamtansatzes für Volumenrendering, wie es z. B. in [RLP07] beschrieben wird. Diese Arbeit konzentriert sich auf die Konzeption eines modularen Shaderframeworks zur Erstellung und Verwaltung der Shaderprogramme selbst.

5.1.1.2 Aufgabe eines Shaderframeworks

Shaderprogramme für Volumenrendering bestehen aus Abschnitten, die den Teilen der Volumenrendering-Pipeline zugeordnet werden können (siehe Programmcode 5 in Anhang A). Der Großteil der Berechnungen wird im Fragmentprogramm ausgeführt. Um die Visualisierung in nur einem Aspekt der Pipeline zu verändern, z. B. der Veränderung der Klassifizierungsberechnung, ist nur der entsprechende Abschnitt im Shadercode zu bearbeiten. Der Rest bleibt unberührt. Shaderprogramme können auf der Grafikkarte nur als Ganzes ersetzt werden. Die Veränderung eines Codeteils erfordert die Neuübersetzung des gesamten Programms.

Die Zusammensetzung der Shader eines Volumenrenderers hängen von seinen Einstellungen ab. Volumenrenderer wie der GVR pflegen daher oft eine Liste *interner Shader* oder Shaderbestandteile, die abhängig von den

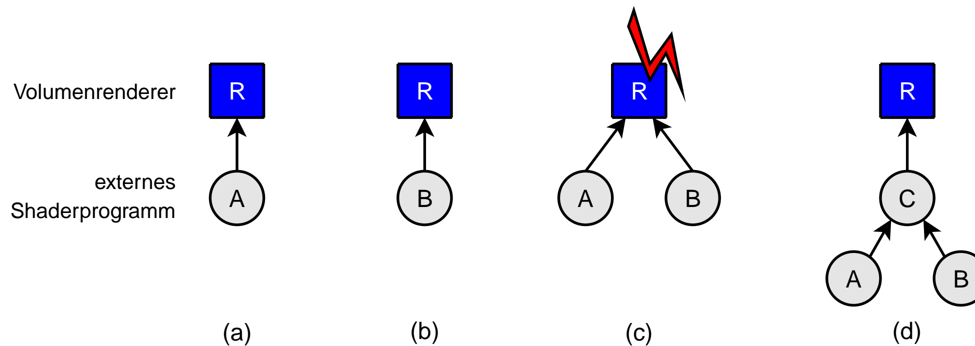


Abbildung 30: Shaderprogramme eines Volumenrenderers können als Ganzes ersetzt werden (a,b). Sollen Bestandteile mehrerer Programme verwendet werden, müssen diese erst kombiniert werden (c,d).

Einstellungen zusammengefügt und aktiviert werden. Bieten die existierenden Shader gewünschte Funktionalität nicht, so ist es mit Hilfe der Grafikbibliothek einfach möglich, die Shaderprogramme des Renderers als Ganzes durch eigene Programme zu ersetzen. Der GVR bot zu Beginn dieser Arbeit die Möglichkeit, ein eigenes, komplettes Shaderprogramm statt des internen Shaders zu verwenden, und so angepasste Berechnungen im Shader auszuführen.

Dieser komplette Austausch des Shaderprogramms ist problematisch. Einstellungen des Renderers, die zur Aktivierung eines veränderten internen Shaders führen, werden nicht mehr berücksichtigt. Sollen mehrere existierende Anpassungen zusammen verwendet werden, so muss der Code per Hand zusammengefügt werden (Abbildung 30). Die inhärente Aufteilung der Shader nach der Volumenrendering-Pipeline lässt die komplette Ersetzung des internen Shadercodes unnötig aufwändig erscheinen. Eine entsprechend erweiterte Darstellung für Shader könnte diese Probleme beseitigen.

Diese Arbeit möchte dazu beitragen, dass der Anwender des Renderers die Grafikkartenprogramme flexibel verändern kann. Anpassungen sollen für den Anwender möglich sein, ohne den gesamten Programmcode ersetzen oder kennen zu müssen. Es ist anzumerken, dass die Integration neuen Shadercodes immer auch die Neuübersetzung eines Shaderprogramms erfordert, da die Schnittstellen der Grafikkarten es nicht zulassen, dass ein Shaderprogramm partiell ersetzt wird. Die Neuübersetzung ist mit Hilfe des Grafikkartentreibers zur Laufzeit problemlos möglich. Die Aufgabe eines Shaderframeworks ist es, diese Neuübersetzung zu kapseln. Der Anwender muss sich darum nicht kümmern und somit auch nicht den gesamten Shadercode kennen. Der folgende Abschnitt geht genau auf die Anforderungen an ein modulares Shaderframework für Volumenrendering ein.

5.1.2 Anforderungen

Grundlegende Anforderungen an das zu entwickelnde Framework ergeben sich aus der Aufgabenstellung und dem Kontext der Shaderprogrammierung. Weitere Anforderungen leiten sich aus dem Anwendungsfeld des Volumenrenderings, sowie dem zu erwartenden Einsatzprofil des Frameworks ab. Berücksichtigt werden weiterhin die Ergebnisse der Gespräche mit Volumenrendering-Anwendern bei MEVIS.

Das Framework dient zur Erstellung von Vertex- und Fragmentshadern für die Grafikkarte. Die erstellten Programme müssen vollwertige Shader darstellen. Sie müssen in eine Form gebracht werden, welche auf der Grafikkarte ausführbar ist. Der Karte kann über geeignete Schnittstellen, wie sie z. B. von OpenGL oder Cg definiert werden, kompilierter Shadercode übergeben werden.

Die Erstellung eines Shaders sollte flexibel und schnell möglich sein. Dazu ist eine Wiederverwendbarkeit existierenden Codes gewünscht. Durch einen entsprechenden, modularen Aufbau der Programme ist so die zügige Neugestaltung oder Veränderung von Shaderprogrammen zu ermöglichen. Teile der Programme sollen austauschbar sein, ohne den gesamten Programmcode der restlichen Teile kennen oder angeben zu müssen. Die Aktivierung der Programme auf der Grafikkarte soll entsprechend ohne große Verzögerung möglich sein, um Veränderungen direkt durch Kontrolle der erstellten Visualisierung bewerten zu können (Rapid Prototyping).

Die erstellte Beschreibung für Shaderprogramme soll für den Einsatz im GPU-basierten Volumenrendering geeignet sein. Algorithmen wie Texturеslicing abstrahieren die Grafikpipeline nicht. Daher sollen Eigenheiten der Shaderprogrammierung, wie die Unterscheidung von Vertex- und Fragmentberechungen, und die Interpolation von Parameterwerten zwischen diesen Schritten, berücksichtigt und gesteuert werden können. Erweiterte Fähigkeiten der Hardware-Verarbeitungseinheiten, z. B. abhängige Texturzugriffe, müssen verwendbar sein.

Die geführten Gespräche mit Volumenrendering-Anwendern bei MEVIS legen nahe, dass vor allem Volumenrendering- und Grafikexperten mit einem Framework dieser Art arbeiten würden. Für Experten sind umfangreiche und detailreiche Einflussmöglichkeiten auf den erstellen Code wünschenswert. In den Interviews wurde weiterhin der Wunsch deutlich, im erstellten Shadercode auch auf selbst angelegte Texturen und uniforme Variablen zugreifen zu können. Dafür ist eine entsprechende Schnittstelle im Framework notwendig, die den Zugriff auf solche externe Ressourcen verwaltet.

Unterschiede zwischen verschiedenen Ansätzen zum Volumenrendering finden sich im Shadercode wieder. Ein Shaderframework für Volumenrendering sollte nicht auf einen Algorithmustyp beschränkt sein. Codemodule sollten in Implementierungen unterschiedlichen Typs wieder-

verwendbar sein. Bei Textureslicing wird pro Ausführung des Shaderprogramms jeder Schritt der Pipeline einmal durchgeführt. Raycasting verarbeitet pro Aufruf des Fragmentprogramms einen ganzen Strahl, und führt die Schritte der Pipeline in einer Schleife aus. Iterativer Kontrollfluss spielt somit für Volumenrendering eine Rolle, und sollte entsprechend berücksichtigt werden.

Der modulare Ansatz muss die Schritte der Volumenrendering-Pipeline angemessen darstellen. Es soll möglich sein, einzelne Teile der Pipeline auszutauschen ohne andere zu beeinflussen. Würden z. B. nur Vertex- und Fragmentprogramme unterschieden (keine nennenswerte, zusätzliche Modularisierung im Vergleich zu Standard-Shadern), wäre dies nicht möglich.

5.1.3 Bewertung möglicher Ansätze

In Kapitel 3 wurden Ansätze zur Beschreibung von Shadern vorgestellt, die über die Verwendung von Shaderhochsprachen hinausgehen. Im Folgenden werden diese kurz im Hinblick auf die vorgestellten Anforderungen bewertet.

5.1.3.1 Shadetrees

Shadetrees und darauf aufbauende Ansätze beschreiben Shader durch Module, zwischen deren Ein- und Ausgabeparameter Abhängigkeiten modelliert werden. Eine hohe Wiederverwendbarkeit der einzelnen Module ist gewährleistet. Die Module stellen sich nach außen als *Blackbox* dar. Wichtig für Ihre Verwendung ist die Signatur an erwarteten und produzierten Parametern. Ihr interner Code ist auf Ebene des Shadetrees unwichtig. Somit können Teile des Shadetrees ersetzt werden, ohne den Code anderer Teile beachten zu müssen. Lediglich auf die Signatur ist zu achten. Shadetrees können zur Laufzeit verändert und in Programmcode einer Shaderhochsprache übersetzt werden [GBD04, MSPK06]. Sie eignen sich somit grundsätzlich für Rapid Prototyping. Auch lineare Strukturen wie eine Pipeline lassen sich gut mit Shadetrees darstellen. Im diesem Falle hat der erzeugte Baum kaum oder keine Verzweigungen. Shadetrees eignen sich insgesamt gut für die Anforderungen dieser Arbeit.

Shadetrees bieten ein hohes Potential für die Anwendung visueller Programmierung. Grafische Werkzeuge können auf ihrer Basis leicht erstellt werden. Dies ist gleichzeitig ein Nachteil: Eine rein textuelle Erstellung wird schnell unübersichtlich, und es besteht die Notwendigkeit grafischer Werkzeuge, die oft großen Implementationsaufwand bedeuten.

Allgemein bedarf die Beschreibung und Verwaltung von Shadetrees umfangreicher Datenstrukturen, mit entsprechend hohem Implementierungsaufwand. Ein weiterer Nachteil der Shadetrees ist die Darstellung von Schleifen, die in Datenflussgraphen schwierig umsetzbar ist (siehe Abschnitt 3.3.2).

5.1.3.2 Metaprogrammierung mit C++

Ansätze wie SH zur Metaprogrammierung von Shadern auf Basis von C++ erlauben es, die mächtigen Konzepte der Programmiersprache zur Strukturierung von Shadercode einzusetzen. Diese Ansätze eignen sich sehr gut zur Erstellung modularen Codes. Sie sind jedoch auf einen C++-Compiler angewiesen, und Shadercode sowie Applikation teilen dieselbe Programmiersprache. Die Aktivierung von neu erstelltem oder verändertem Shadercode bedeutet die Neuübersetzung von Teilen, oder sogar der gesamten Applikation. Obwohl Code flexibel weiterverwendet werden kann, können Veränderungen nicht schnell aktiviert werden. Ein Vorgehen im Sinne von Rapid Prototyping sind daher nur schwierig realisierbar. Ansätze zur Metaprogrammierung von Shadern mit C++ eignen sich daher wenig für die in dieser Arbeit gestellten Anforderungen.

5.1.3.3 Kombinierbare Shader

Diese Ansätze erlauben die Kombination mehrerer Shader zu einem Gesamtshader. Eine Shaderalgebra erlaubt die Kombination von elementaren Shaderprogrammen mit Hilfe von Eingabe- und Ausgabeparametern und Verknüpfungsoperationen. Sie ähneln dahingehend den Shadetrees, und erlauben eine gute Modularisierung. Die Verwendbarkeit der existierenden Implementation von Shaderalgebra in SH ist jedoch eingeschränkt, da sie auf der Metaprogrammierung in C++ beruht.

Auf der Idee eines SuperShaders basierende Ansätze stellen Codeteile von Shaderprogrammen auf der Grafikkarte zur Verfügung. Zur Laufzeit kann eine Untermenge dieser Codeteile ausgewählt und linear hintereinander ausgeführt werden. Verschiedene, logische Shaderprogramme werden auf diese Weise repräsentiert. Die lineare Hintereinanderausführung von Teilen eines Shaders scheint gut geeignet für die Darstellung der Volumenrendering-Pipeline. Sie könnte vom Kontrollshader dargestellt werden. Durch die gegebene Unterteilung eines Shaders in Fragmente lassen sich auch neue Codeteile einfach integrieren.

Nachteil des Ansatzes eines Kontrollshaders für Codeteile ist die Art, wie die einzelnen Fragmente kommunizieren. Daten werden über eine globale Datenstruktur weitergegeben. Jeder Codeteil nutzt dieselbe Struktur. Alle Codeteile haben daher dieselbe Signatur, abgesehen vom eigenen Namen. Welche Werte von einem Fragment erwartet bzw. erzeugt werden, ist ohne den Quellcode des Fragments nicht ersichtlich.

Bei modularen Shadern mit Hilfe eines Kontrollshaders handelt es sich um einen schlanken Ansatz (en. *lightweight*). Er kann komplett mit Hilfe der Standardfunktionalität einer Shaderhochsprache umgesetzt werden. Insgesamt hat der Ansatz sowohl Vor- als auch Nachteile im Bezug auf die Anforderungen dieser Arbeit.

5.1.3.4 General Purpose GPU

Ansätze zur Verwendung der Grafikkarte als allgemeinen Koprozessor abstrahieren stark von der Grafikpipeline. Eine solche Abstraktion ist nicht für alle Algorithmen des Volumenrenderings geeignet (vgl. Abschnitt 3.2). Die Ergebnisse dieser Arbeit sollen auch für hardwarenahe Algorithmen wie Slicing geeignet sein. GPGPU-Ansätze sind daher für das zu erstellende Framework nicht geeignet.

5.1.4 Auswahl eines geeigneten Ansatzes

Die vorhergehenden Betrachtungen identifizieren Shadetrees und *Super-Shader*-basierte Ansätze als mögliche Grundlagen für die Entwicklungen dieser Arbeit. Das in Abschnitt 5.2 vorgestellte Konzept basiert auf Shadetrees. Der folgende Abschnitt diskutiert diese Entscheidung.

Die Volumenrendering-Pipeline kann gut auf das Konzept der Super-Shader übertragen werden. Ihre Einzelschritte könnten als wiederverwendbare und leicht austauschbare Funktionen vom Kontrollshader aufgerufen werden. Für diesen Ansatz spricht auch, dass er mit den Mitteln einer Shaderhochsprache wie GLSL komplett umsetzbar ist. Zusätzliche Hilfsmittel, wie z. B. eine GUI für die Erstellung des Kontrollshaders oder die Zuordnung der Funktionen zur Volumenrendering-Pipeline, sind hilfreich, aber nicht zwingend notwendig.

Ein Nachteil des SuperShader-Ansatzes liegt in der identischen Signatur aller Funktionen. Es ist somit ohne die Betrachtung des Quellcodes der Funktion nicht erkennbar, welche Werte erwartet oder geliefert werden. Gerade dies ist aber im Kontext Volumenrendering von Bedeutung. Von einem Schritt zum Nächsten werden Werte weitergegeben, z. B. Farb- und Alphawerte von der Klassifizierung zur Beleuchtung. In den Anforderungen wurde aufgezeigt, dass ein Teil des Programmcodes ohne genaue Kenntnisse anderer Teile austauschbar sein soll. Um die Abhängigkeiten der einzelnen Funktionen untereinander zu erkennen, muss jedoch deren Programmcode betrachtet werden.

Der Ansatz eines SuperShaders ist schlank. Er ist direkt in einer Hochsprache für Shader umsetzbar. Aus diesem Grund sind auch iterative Kontrollstrukturen einfach integrierbar. Eine Raycasting-Loop könnte gut auf Ebene des Kontrollshaders integriert werden.

Auch Shadetrees erlauben es, die Volumenrendering-Pipeline geeignet darzustellen. Zwischen verbundenen Knoten im Graph besteht eine zeitliche Abhängigkeit. Die Berechnungen von Nachfolgerknoten werden vor ihren Vorgängern ausgeführt. Ordnet man Knoten einem Einzelschritt zu, kann der gesamte Graph in die Form der Pipeline gebracht werden. Bei Shadetrees ist die Signatur der Knoten aussagekräftig. Ein- und Ausgabewerte sind bekannt, ohne den internen Code betrachten zu müssen. Ein einzelner Teil des Graphs kann sehr einfach ausgetauscht werden, da die

Schnittstellen der verbundenen Teile des Graphs bekannt sind. Dies ist ein Vorteil gegenüber einem SuperShader-basierten Ansatz. Ein weiterer Vorteil der Shadetrees ist ihre gute Visualisierbarkeit. Graphen eignen sich hervorragend zum Transport von Ideen. Es ist möglich, auf abstrakter Ebene über einen Shadetree zu diskutieren, ohne die Details der Programmierung kennen zu müssen.

Der intuitiven Bedienbarkeit der Shadetrees steht ein erhöhter Aufwand zur Umsetzung gegenüber. Ein Graph wird erzeugt, in dem Knoten angelegt und ihre Parameter verknüpft werden. So werden Abhängigkeiten zwischen den Knoten definiert. Dies entspricht dem Datenflusskonzept, auf welchem Shadetrees beruhen (siehe Abschnitt 3.3.1). Sie können somit nicht direkt in einer Programmiersprache für Shader realisiert werden. Eine eigenständige, zusätzliche Repräsentation muss geschaffen werden, die auch iterative Strukturen ermöglicht. Für die Erstellung von Shadern für die Grafikkarte müssen die Shadetrees erst in passenden Programmcode transformiert werden. Die Spezifikation von Schleifen sind in Shadetrees mit vergleichsweise hohem Aufwand verbunden.

Shadetrees sind im Vergleich zu SuperShader-basierten Beschreibungen intuitiver, denn die Schnittstellen zwischen den Teilen eines Programms sind klarer definiert. Die Strukturen eines Shadetrees sind gut visuell darstellbar. Dafür ist ihre Umsetzung umfangreicher, und die Integration von iterativen Strukturen ist aufwändiger. Die Vorteile der Shadetrees rechtfertigen den Mehraufwand im gegebenen Anwendungsfall. Es wird erwartet, dass klare Signaturen für flexible und schnelle Veränderungen am Shader von großer Hilfe sind.

5.2 Konzept

In den folgenden Abschnitten wird ein auf Shadetrees basierendes Konzept eines modularen Shaderframeworks für Volumenrendering beschrieben. Der Basisansatz wurde durch die Integration iterativer Konstrukte und eine Strukturierung des Graphen auf Basis der Volumenrendering-Pipeline erweitert. Die Beschreibung geschieht unabhängig von Implementierungsdetails, sodass Umsetzungen des vorgestellte Konzepts mit verschiedenen Volumenrenderern oder Shader-Programmiersprachen denkbar sind. Kapitel 6 beschreibt die in dieser Arbeit erstellte exemplarische Realisierung.

5.2.1 Grundstruktur der Shaderrepräsentation

Bei der Beschreibung von Shadern mit entsprechenden Hochsprachen gibt es zwei Ebenen. Der Programmierer erstellt und verändert die Beschreibung des Shaders in der Hochsprache. Auf Ebene der Grafikkarte wird die kompilierte Version des Programms ausgeführt. Das im Weiteren vorgestellte Konzept fügt eine dritte Ebene hinzu (Abbildung 31). Der Program-

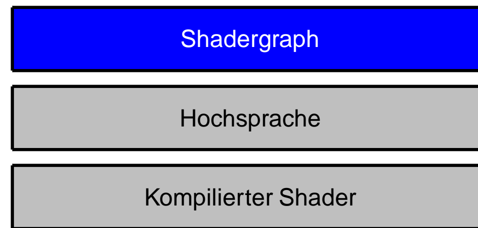


Abbildung 31: Die Darstellung von Shaderprogrammen durch Hochsprache und kompiliertes Programm wird um eine weitere Ebene erweitert.

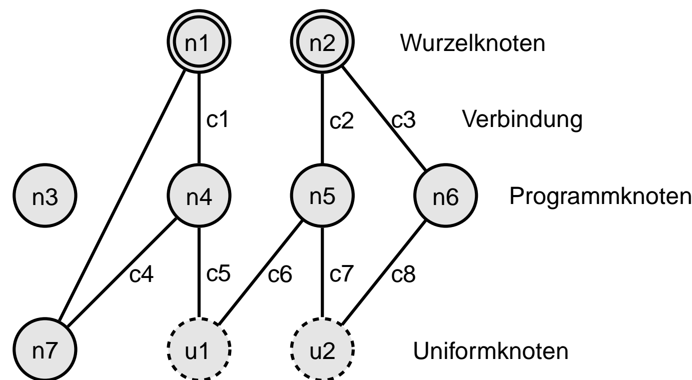


Abbildung 32: Bestandteile eines Shadergraphen.

mierer verwaltet das Programm nicht mehr in der Hochsprache selbst, sondern auf der übergeordneten Ebene eines *Shadergraphen* (Abschnitt 5.2.1.1 gibt die genaue Definition). Auf dieser Ebene ist es möglich, Funktionalität in unabhängigen, wiederverwendbaren Modulen zu entwickeln. Auch die weiteren Anforderungen, wie die Strukturierungsmöglichkeit anhand der Volumenrendering-Pipeline, werden berücksichtigt. Die im Shadergraph erstellte Beschreibung wird später in Programmcode in einer Hochsprache für Shader überführt, bevor das Programm auf der Grafikkarte aktiviert wird. Somit sind alle Fähigkeiten der Hochsprache, wie z. B. abhängige Texturzugriffe, nutzbar. Die in Kapitel 6 beschriebene Umsetzung verwendet GLSL. Die Beispiele dieses Kapitels werden ebenfalls in dieser Sprache gegeben. Es ist denkbar, eine Implementierung des Konzepts mit Hilfe einer anderen Hochsprache wie Cg oder HLSL umzusetzen.

5.2.1.1 Shadergraph

Ein *Shadergraph* ist ein ungerichteter, nicht-zyklischer Graph und beschreibt Shaderprogramme für die GPU. Wie bei Shadetrees, auf deren Konzept sie

beruhen, handelt es sich bei Shadergraphen nicht um Bäume[20]. Dennoch wird in den folgenden Betrachtungen an geeigneter Stelle auf die Nomenklatur von Bäumen zurückgegriffen (Wurzel, Blätter), da sie für die dargestellten Zusammenhänge geeignet erscheint. Kanten in Shadergraphen werden in beide Richtungen verfolgt (siehe Abschnitte 5.2.2 und 5.2.3), und sind daher ungerichtet.

Ein Shadergraph beschreibt gleichzeitig ein Vertex- und ein Fragmentprogramm. Beide Programme werden zusammen in einem Graph integriert, da gleichzeitig aktive Vertex- und Fragmentprogramme in jedem Fall aufeinander abgestimmt werden müssen. Weiterhin kann so die Repräsentation von uniformen Parametern im Graph von beiden Programmen geteilt werden. Dasselbe trifft auch auf Variablen zu, die vom Vertex- zum Fragmentprogramm interpoliert werden. Ein Shadergraph S besteht aus vier endlichen Mengen:

$$S = (N, U, C, R) \tag{13}$$

Die Mengen der Uniformknoten (*Uniforms*) U und Programmknoten (*Nodes*) N bilden die Knoten des Graphen S. Sie werden über ungerichtete Kanten (Verbindungen bzw. *Connections*) C verknüpft. Uniformknoten können nur als Blätter im Graphen auftreten, Programmknoten sowohl als Blätter als auch als innere Knoten. Die Menge der Wurzelknoten (*Roots*) R ist eine Teilmenge von N:

$$R \subseteq N \tag{14}$$

Die Knoten r der Menge R sind Wurzeln von Teilgraphen des Shadergraphen. Diese Teilgraphen werden unabhängig voneinander verarbeitet. Knoten, welche über keinen Pfad mit einem der Wurzelknoten verbunden sind, werden bei der Verarbeitung nicht berücksichtigt. Eine Wurzel des gesamten Shadergraphen existiert nicht immer. Abbildung 32 zeigt die Grundstruktur eines Shadergraphen.

5.2.1.2 Knoten

Ein *Programmknoten* kapselt Programmcode sowohl für das Vertex- als auch für das Fragmentprogramm. Er repräsentiert jeweils eine vom Rest des Graphen unabhängige Berechnung. Die Berechnungen verarbeiten Eingabewerte des Knoten, und geben ihre Ergebnisse als dessen Ausgabewerte nach außen weiter. Ein Programmknoten p hat folgende Bestandteile:

$$p = (n, I, O, vb, fb) \tag{15}$$

Jeder Knoten hat einen Namen n, der im ganzen Shadergraph eindeutig sein muss. Der Name muss einem gültigen Variablennamen der Basissprache entsprechen, da er Einfluss auf den aus dem Graph erzeugten Programmcode hat. Die endlichen Mengen I der Eingabeparameter und O

[20] Die Namensgebung von Shadetrees ist im Hinblick auf diese Tatsache unglücklich.

Eigenschaft	mögl. Werte
Name	eindeutiger Variablenname der Basissprache
Datentyp	alle Datentypen der Basissprache
Parametertyp	input output uniform
Verhaltenstyp	dataflow varying loop
Programmtyp	vertex fragment any

Tabelle 3: Eigenschaften von uniformen Parametern und Knotenparametern.

der Ausgabeparameter bestimmen die Signatur des Programmknotens. Sie stellen seine Schnittstellen zu anderen Knoten dar. Der Programmcode für die dargestellten Berechnungen wird in jeweils einem String gespeichert. vb (*Vertex Body*) enthält Programmcode der Hochsprache für das Vertexprogramm, fb (*Fragment Body*) entsprechend für das Fragmentprogramm. Ein- und Ausgabeparameter haben einen im Knoten eindeutigen Namen. Diese können im Programmcode verwendet werden, um die Werte der Parameter zu lesen und zu schreiben.

Ein *Uniformknoten* kapselt eine einzige uniforme Variable. Dazu beinhaltet ein Uniformknoten u einen einzigen uniformen Parameter $p_{uniform}$:

$$u = (p_{uniform}) \tag{16}$$

Der Wert des Parameters wird dem Rest des Graphen zur Verfügung gestellt.

Programmknoten entsprechen im Shadergraph den Operationen eines Datenflussgraphs. Uniformknoten entsprechen den Konstantengeneratoren, deren Aufgabe die Bereitstellung eines Wertes ist.

5.2.1.3 Parameter

Parameter sind Bestandteile von Programm- und Uniformknoten. Programmknoten haben eine beliebige Anzahl an Parametern, Uniformknoten entsprechen einem einzigen Parameter. Jeder Parameter repräsentiert einen Wert eines bestimmten Datentyps. Ein Parameter wird definiert über mehrere Eigenschaften (vgl. Tabelle 3). Nicht alle Parameterkombinationen sind möglich.

Es werden drei *Parametertypen* unterschieden: Eingabe- (*input*), Ausgabe- (*output*) und uniforme Parameter (*uniform*). Ein- und Ausgabeparameter werden in Programmknoten verwendet, uniforme Parameter in Uniformknoten.

Der *Datentyp* eines Parameters bestimmt die Art der Daten, die durch ihn fließen können. Es können alle Datentypen der Basissprache eingesetzt werden. Dabei ist zu beachten, dass Datentypen, die Zugriff auf eine Textur erlauben (z. B. `sampler1D`, `sampler2D`, usw.), nur als Eingabeparameter oder uniforme Parameter auftreten können. Variablen solcher Datentypen können nicht innerhalb eines Shaderrumpfs angelegt werden.

Der *Name* eines Parameters muss ein gültiger GLSL-Variablenname sein. Uniforme Parameter müssen für den ganzen Shadergraph und Knotenparameter innerhalb eines Knotens eindeutig benannt sein. Eingabe- und Ausgabeparameter werden über den *Programmtyp* dem Vertex- oder Fragmentprogamm zugeordnet. Sie stehen einem Programmknoten im entsprechenden Programmcode als Variablen zur Verfügung. Uniforme Parameter sind keinem Programm zugeordnet.

Das Verhalten eines Parameters kann durch den *Verhaltenstyp* näher bestimmt werden. Er legt fest, wo der Wert eines Parameters zur Verfügung steht und wie ein Knoten einen Eingabeparameter verarbeitet. Im Normalfall erhalten Parameter den Verhaltenstyp *dataflow* (dt. Datenfluss). Ausgabeparameter dieses Typs stellen ihren Wert im entsprechenden Vertex- oder Fragmentprogramm zur Verfügung und sind im jeweils anderen Programm nicht zugänglich. Eingabeparameter dieses Typs stellen ihren Wert als Variable zur internen Verwendung im Programmknoten zur Verfügung. Uniforme Parameter sind immer vom Typ *dataflow*, und stellen ihren Wert in allen Programmen zur Verfügung. Mit ihrer Hilfe werden die Abhängigkeiten zwischen einzelnen Operationen modelliert.

Für Ausgabeparameter existiert zudem der Verhaltenstyp *varying*. Variablen dieses Typs stehen im Vertex- zum Fragmentprogramm zur Verfügung, ihr Wert wird zwischen den Programmen interpoliert. Typischerweise wird sie im Vertexteil geschrieben, und im Fragmentteil gelesen. Ein weitere Verhaltenstyp für Ein- und Ausgabeparameter ist *loop*. Er dient zur Realisierung von Schleifen im Shadergraph und wird in Abschnitt 5.2.3 näher erläutert.

5.2.1.4 Verbindungen

Ein Shadergraph besitzt *Verbindungen*, welche die Datenabhängigkeiten zwischen Parametern definieren. Sie entsprechen den Kanten des Graphen. Eine Verbindung c verbindet einen Eingabeparameter und jeweils entweder einen Ausgabeparameter oder einen uniformen Parameter.

$$c = \begin{cases} (p_{in}, p_{out}) \\ (p_{in}, p_{uniform}) \end{cases} \qquad (17)$$

Für eine gültige Verbindung müssen Daten- und Programmtyp der verbundenen Parameter übereinstimmen. Hat der Ausgabeparameter den Programmtyp *any*, so kann er an Eingabeparameter jeden Typs angeschlossen werden. Die Verhaltenstypen *dataflow* und *varying* sind kompatibel. Entsprechende interpolierte Ausgabeparameter können an beliebige Eingabeparameter angeschlossen werden. Es ist nicht möglich, Eingabe- und Ausgabeparameter desselben Programmknotens zu verbinden. Ein weiteres Kriterium für die Möglichkeit einer Verbindung sind bereits vorhandene Verbindungen. Eingabeparameter können nur in einer Verbindung auftreten, da sonst unklar wäre, welche der eintreffenden Daten zu verwenden sind. Ausgabewerte können beliebig oft in einer Verbindung verwendet werden, da die Daten zu den Eingabeparametern kopiert werden.

Verbindungen sind ungerichtet. Die Richtung des Datenflusses entlang einer Verbindung ergibt sich aus den verbundenen Parametern. Daten fließen immer in Richtung des Eingabeparameters. Kanten können bei der Verarbeitung des Shadergraphen in beide Richtungen traversiert werden. Die Verarbeitung des Shadergraphen wird in Abschnitt 5.2.2 beschrieben.

5.2.1.5 Beispiele und Grafische Notation

Dieser Abschnitt zeigt erste Beispiele für Shadergraphen und seine Elemente. Eine beispielhafte grafische Notation für Shadergraphen wird vorgestellt, die auch im Rest dieses Kapitels verwendet wird.

Abbildung 33 zeigt den beispielhaften Programmknoten `interpolateAndSample3DTexture`. Er enthält alle vorgestellten Parametertypen und stellt den Wert einer Textur zur Verfügung, die mit dem Wert einer zuvor erstellten Varyingvariable indexiert wird. Ebenfalls enthalten ist die grafische Notation für Knoten eines Shadergraphs (*Rechtecke mit runden Kanten*). Es werden neben den Knoten selbst auch ihre Parameter grafisch dargestellt (*Dreiecke*[21]). Die Leserichtung ist von unten nach oben: Ausgabeparameter werden oben und Eingabeparameter unten am Knoten eingezeichnet. Die Zuordnung zu Vertex- oder Fragmentprogramm wird über eine Teilung des Knotens gezeigt. Links sind Vertexparameter, rechts Fragmentparameter angeordnet. Eine Erweiterung auf weitere Programmtypen durch weitere Unterteilungen (wie Geometrieprogramme) ist denkbar. Die Unterteilungen sollten dann weiterhin von links nach rechts angeordnet werden, entsprechend der Position des zugeordneten Programmtyps in der Pipeline. Varyingparameter, die von beiden Programmen verwendet werden können, sind in einem geteilten Bereich angeordnet. Bei Uniformknoten fehlt die Unterteilung, da der enthaltene uniforme Parameter in allen Programmen verfügbar ist. Verbindungen werden durch Kurven zwischen der visuellen Darstellung zweier Parameter dargestellt.

Bei der Gestaltung einer grafischen Darstellung für gegebene Struktu-

[21]Nicht zu verwechseln mit den dreieckigen Bildein- und ausgängen von ML-Modulen.

Parameter				
Richtung	Programm	Verhaltenstyp	Datentyp	Name
input	vertex	dataflow	int	inTextureUnit
output	any	varying	vec3	texturePosition
input	fragment	dataflow	sampler3D	inTexture
output	fragment	dataflow	vec4	outColor

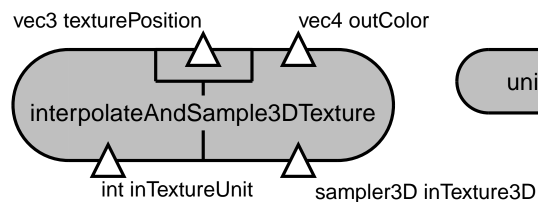
Codeteil Vertexprogramm:

```
1 texturePosition = vec3(gl_TextureMatrix[inTextureUnit] * gl_Vertex);
```

Codeteil Fragmentprogramm:

```
1 outColor = texture3D(inTexture,texturePosition);
```

(a)



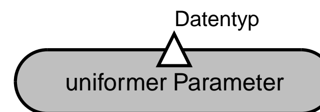


Abbildung 33: (a) Beispiel für Elemente eines Programmknotens `interpolateAndSample3DTexture`. (b) Beispielhafte grafische Notation des Programmknotens. (c) Grafische Notation für Uniformknoten.

ren stellt sich die Frage, welche Informationen eine visuelle Repräsentation enthalten soll. Oft ist es nicht sinnvoll, alle Informationen in der visuellen Entsprechung anzuzeigen. Grafiken sind schnell überladen, und verlieren ihre Fähigkeit, Zusammenhänge schnell und einfach zugänglich zu präsentieren (siehe [SM00, Kapitel 1]). Auch für die Repräsentation von Shadergraphen erscheint es nicht sinnvoll, alle relevanten Informationen gleichzeitig in der Grafik darzustellen. Detailinformationen wie Verhaltenstypen, Parameternamen oder die Codeteile für Vertex- und Fragmentprogramm sollten nicht immer dargestellt werden. Sie sind für den initialen Überblick über einen komplexen Shadergraph nicht zwingend notwendig. Zur Verdeutlichung einzelner Zusammenhänge können einzelne Knoten oder Subgraphen vergrößert und mit erweiterten Informationen dargestellt werden. Bei einer Umsetzung als grafische Benutzungsschnittstelle im Computer können Detailinformationen auch durch Auswahl eines Knotens oder Parameters sichtbar gemacht werden.

Abbildung 34 zeigt die grafische Notation eines kompletten Shadergraphs. Er stellt einfache Shader für Volumenrendering dar, die zu den Pro-

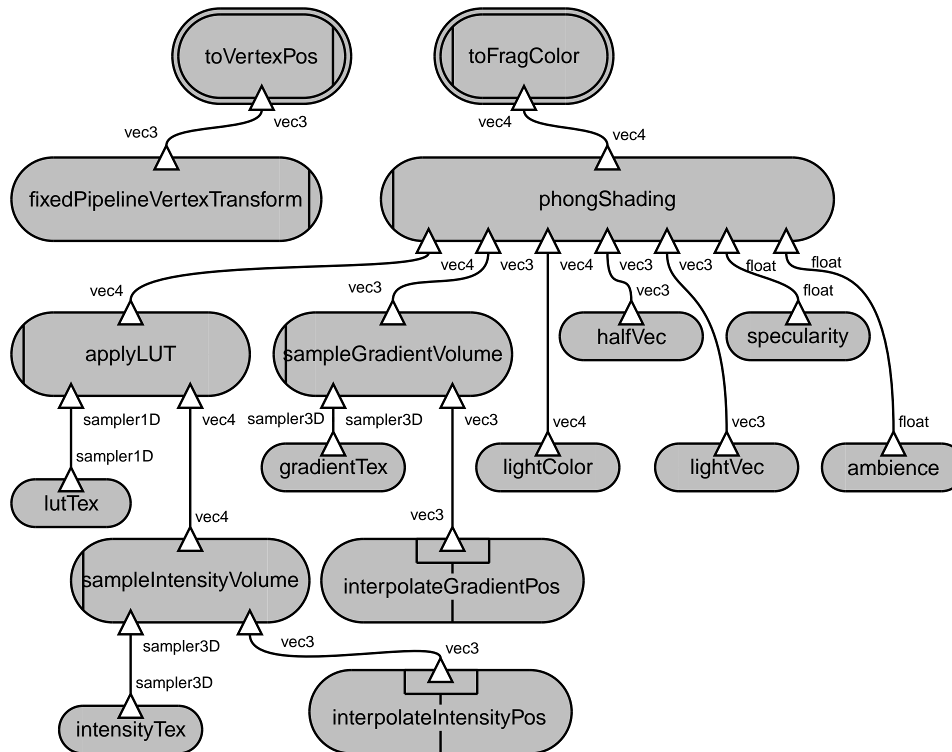


Abbildung 34: Kompletter Shadergraph. Er beschreibt ein einfaches Vertex- und Fragmentprogramm für Volumenrendering, und ist äquivalent zu den Programmen aus Programmcode 5.

grammen aus Programmcode 5 äquivalent sind. Der Shadergraph besteht aus den zwei Wurzeln `toVertexPos` und `toFragColor`, welche die Ergebnisse von Vertex- und Fragmentprogramm in die entsprechenden Variablen der Hochsprache schreiben. Der restliche Graph besteht aus Programmknoten zum Zugriff auf Intensitäts- und Gradientenvolumen, zur Anwendung der Transferfunktion, zur Berechnung des Phong-Beleuchtungsmodells sowie Uniformknoten für verschiedene Einstellungswerte.

5.2.1.6 Datenflusseigenschaften

Shadergraphen stellen einen Hybrid Dataflow Ansatz dar. Einzelne Operationen werden durch sequentiellen Shadercode beschrieben. Das Gesamtprogramm wird durch Datenabhängigkeiten modelliert. Der entstehende Shadergraph wird anschliessend in ein klassisches Shaderprogramm übersetzt. In diesem Programm ist der beschriebene Datenfluss noch implizit vorhanden. Ausgeführt wird das Programm aber nicht parallel, sondern sequentiell.

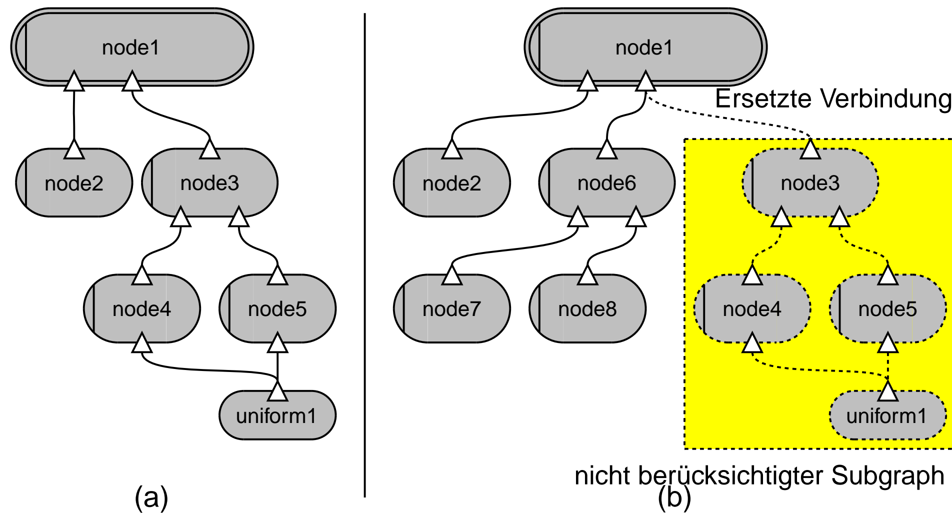


Abbildung 35: Der in (a) dargestellte Shadergraph wird in (b) durch neue Knoten und Verbindungen verändert.

5.2.1.7 Wiederverwendbarkeit und Anpassung

Die Festlegung von Parametern und Programmcode für einen Knoten kann auch als Beschreibung eines *Knotenmusters* verstanden werden. Knoten stellen in diesem Fall Instanzen eines Knotenmusters dar. Einmal definierte Knoten können so mehrmals im Graph verwendet werden. Knoten unterscheiden sich vom zugehörigen Knotenmuster dadurch, dass erst mit konkreten Knoten der Aufbau von Verbindungen mit seinen Parametern möglich ist. Durch die Hinterlegung eines Knotenmusters in geeigneter Form (siehe auch Abschnitt 6.1) lässt sich einmal erstellte Funktionalität weiterverwenden. Es ist hilfreich, den Knotenmustern ebenfalls eine Benennung zu geben. Durch einen aussagekräftigen Namen und die Signatur der Parameter kann somit die Funktionalität des Knotenmusters erkannt werden, ohne dass der interne Code betrachtet werden muss.

Ein existierender Shadergraph kann einfach erweitert oder verändert werden. Dazu wird ein neuer Subgraph erstellt, und durch eine entsprechende Verbindung an den Graphen angehängt. Durch das Ersetzen existierender Verbindungen können so Shadergraphen modifiziert werden (siehe Abbildung 35). Durch das Hinzufügen mehrerer Verbindungen zwischen neuem Subgraph und existierendem Graph können auch neue Programmknoten zwischen existierende platziert werden. Werden existierende Verbindungen entfernt, so ist der Subgraph unterhalb der Verbindung unter Umständen von den Wurzeln des Shadergraphen nicht mehr erreichbar. Er ist abgetrennt und wird nicht weiter berücksichtigt.

Das Verändern eines Shadergraphen basiert auf mehreren Operationen. Zwei Shadergraphen S_1 und S_2 können vereint werden:

$$\begin{aligned} S &= S_1 \oplus S_2 \\ &= (N_1 \cup N_2, U_1 \cup U_2, C_1 \cup C_2, R_1 \cup R_2) \end{aligned} \tag{18}$$

Danach gehören die Knoten und Verbindungen zum selben Shadergraphen, sind aber noch nicht durch neue Verbindungen zu einem einzigen Graph zusammengefügt. Neue Verbindungen können hinzugefügt werden, die ggf. existierende Verbindungen überschreiben:

$$\begin{aligned} S_{neu} &= S \oplus c \\ &= S \oplus (p_{in}, x) \\ &= (N, U, (C \setminus \{(p_{in}, y)\}) \cup \{(p_{in}, x)\}, R) \end{aligned} \tag{19}$$

Ebenfalls verändert werden können die Wurzeldefinitionen. Wurzeln können entfernt werden:

$$\begin{aligned} S_{neu} &= S \oplus r \\ &= (N, U, C, R \cup \{r\}) \end{aligned} \tag{20}$$

Wurzeln können hinzugefügt werden:

$$\begin{aligned} S_{neu} &= S \ominus r \\ &= (N, U, C, R \setminus \{r\}) \end{aligned} \tag{21}$$

Mit Hilfe beider Operationen ist es möglich, eine alte Wurzel durch eine neue zu ersetzen.

5.2.2 Verarbeitung

Aus einem Shadergraph werden zwei gültige GLSL-Programme für Vertex- und Fragmentshader generiert, um sie auf der Grafikkarte ausführen zu können. Sie stellen die Berechnungen des Shadergraphen durch äquivalenten, sequentiellen Programmcode dar. Die Übersetzung eines Shadergraphen in sequentiellen Programmcode erfolgt in zwei Schritten. Zunächst wird die durch die Datenabhängigkeiten zwischen den Programmknoten festgelegte Ausführungsreihenfolge der Berechnungen bestimmt. Diese Reihenfolge findet sich in den erstellten GLSL-Programmen wieder. Im zweiten Schritt wird für jeden Programmknoten ein Codeblock in das entsprechende Programm eingefügt. Ein- und Ausgabeparameter werden durch entsprechende Variablen repräsentiert. Variablen verbundener Parameter werden einander zugewiesen.

5.2.2.1 Topologische Sortierung

Daten fließen im Shadergraph von Aus- zu Eingabeparametern. Ein Knoten kann seine Ausgabedaten nur zur Verfügung stellen, wenn die entsprechenden Eingabedaten vorliegen, und seine Berechnungen ausgeführt wurden. Er ist somit direkt abhängig von den Werten der Ausgabeparameter seiner direkten Nachfolger. Der Knoten ist die Wurzel eines Subgraphs und somit indirekt abhängig von allen Knoten in diesem Graph.

Programmknoten werden abhängig von diesen Datenabhängigkeiten in eine lineare Liste einsortiert. Ein Knoten darf in die Liste erst eingefügt werden, wenn alle Knoten seines Subgraphs schon eingefügt wurden. Der Programmcodes der Knoten wird in der durch diese Liste gegebenen Reihenfolge ausgeführt. Dadurch ist für jeden Knoten sichergestellt, dass alle Programmteile der direkten und indirekten Vorgängerknoten schon ausgeführt wurden und alle notwendigen Eingabewerte vorliegen. Uniformknoten müssen bei der Sortierung nicht berücksichtigt werden, da sie keine Berechnungen beinhalten. Für sie ist nur entscheidend, ob sie überhaupt verwendet werden. Falls nicht, werden sie bei der Erstellung des Shadercodes ignoriert.

Bei der Sortierung handelt es sich um eine *topologische Sortierung*. Sie erfolgt unabhängig von der Zuordnung zu Vertex- und Fragmentprogramm. Für Bäume, deren Knoten nicht mehr als einen direkten Vorgänger haben, liefert die Traversierung des Graphen mit Hilfe der *Tiefensuche* eine topologische Sortierung, in der jeder Knoten nur einmal besucht wird.

Bei Shadergraphen können die Ausgabedaten eines Knotens von mehreren anderen Knoten verwendet werden, in welchem Fall er mehrere direkte Vorgänger hat. Ein solcher Knoten wird bei der Traversierung mit Tiefensuche mehrmals besucht. Ein entsprechendes Beispiel ist Knoten n7 in Abbildung 32. Einfache Tiefensuche liefert die sortierte Sequenz:

n7-n7-n4-n1-n5-n6-n2

Eine direkte Umsetzung der sortierten Sequenz würde den Programmcode des Knotens $n7$ mehrmals ausführen. Eine mehrmalige Ausführung des Programmcodes eines Knotens ist jedoch überflüssig. Während einer Ausführung eines Shaderprogramms ändern sich dessen Eingabewerte (Vertex- bzw. Fragmenteigenschaften, uniforme Parameter) nicht. Wurde der Code eines Knoten bereits zu einem früheren Zeitpunkt im Shaderprogramm berechnet, können die Ergebnisse weiterverwendet werden. Daher sind doppelte Eintragungen eines Knoten in der sortierten Liste zu entfernen. Zur direkten Erstellung einer topologischen Sortierung ohne doppelte Eintragungen kann bei der Tiefensuche für jeden Knoten gespeichert werden, ob er schon besucht wurde. Dies führt zu dem Ergebnis:

n7-n4-n1-n5-n6-n2

Die angepasste Tiefensuche sortiert so, dass Knoten eines Subgraphen nacheinander einsortiert werden. Abhängige Knoten stehen in der erstellten Sortierung nah beieinander.

Eine alternative Anpassung der Tiefensuche speichert die Tiefe des längsten Pfades von der Wurzel zu jedem Knoten. Diese werden dann nach dieser Tiefe absteigend sortiert, beginnend mit dem tiefsten Knoten. Die entstehende Sortierung ist eine gültige, topologische Sortierung. Abhängige Knoten können nie dieselbe Tiefe erhalten, da zwischen ihnen eine Kante vorhanden ist. Damit existiert zum Nachfolgerknoten auf jeden Fall ein Pfad, dessen Tiefe um eins größer ist als die des Vorgängerknotens. In der so entstehenden Sortierung stehen Knoten ähnlicher Tiefe im Graph nah beieinander. Der Graph aus Abbildung 32 würde mit diesem Ansatz wie folgt sortiert:

n7-n4-n5-n6-n1-n2

Für die Erstellung des sequentiellen Shadercodes spielt es keine Rolle, welche der vorgestellten, angepassten Tiefensuchverfahren eingesetzt wird. Es erscheint jedoch möglich, dass die Orientierung im entstehenden Code etwas leichter fällt, wenn Programmcode von Knoten gleicher Tiefe zusammen steht. Die Tiefe eines Knotens im Graph kann visuell gut erfasst werden. Mit dieser Information ist dann leicht auf die Position im entstehenden Code zu schließen. Ein weiterer Vorteil im Kontext Volumenrendering ist, dass Code für Knoten eines Pipelineschritts nahe beieinander stehen (siehe Abschnitt 5.2.4). Im Fall der Tiefensuche ohne doppelte Verarbeitung von Knoten hängt die Position nicht von der Tiefe im Graph ab, sondern von der Reihenfolge der Verarbeitung von Kindkoten (z. B. links nach rechts). Eine Schluss auf die Position im entstehenden Code ist damit schwieriger zu ziehen.

5.2.2.2 Erzeugung von Programmcode

Für jeden Knoten wird nach der topologischen Sortierung unabhängig von den anderen Knoten Programmcode erzeugt. Der erzeugte Code wird dann an die durch die Sortierung vorgegebene Stelle in den Rumpf des entsprechenden Shaderprogramms eingefügt. Früher auszuführende Berechnungen stehen weiter oben im Programm. Die Erzeugung der Programme erfolgt getrennt. Für Vertex- und Fragmentprogramm werden nur die jeweils relevanten Parameter und Codeteile eines Programmknotens berücksichtigt.

Die Erstellung eines Codeblocks besteht aus mehreren Schritten. Zunächst werden die im Knoten hinterlegten Programmzeilen durch geschweifte Klammern eingeschlossen, wodurch ein eigener *Scope* erstellt wird. Dies stellt sicher, dass im Codeteil deklarierte Variablen nicht mit Variablen anderer Knoten kollidieren. Danach wird außerhalb dieses eingeklammerten Bereichs jeweils eine Variable für die Ein- und Ausgabeparameter vom

Typ *dataflow* erzeugt. Ausnahme sind hier Eingabeparameter zum Zugriff auf Texturen (Sampler in GLSL). Diese können innerhalb des Shaderrumpfs nicht angelegt werden. Ausgabevariablen vom Typ *varying* werden dem Header hinzugefügt. Ebenfalls dem Header hinzugefügt werden Deklarationen der uniformen Variablen, die durch die Uniformknoten definiert werden.

Variablen erhalten den Namen des Parameters, der um den Namen des Knotens erweitert wurde. Da Knotennamen innerhalb des Graphen eindeutig sind, wird eine Namenskollision somit vermieden. Im Programmcode des Knotens werden die verwendeten Ein- und Ausgabevariablen ebenfalls entsprechend umbenannt. Innerhalb des Scopes sind die extern deklarierten Variablen sichtbar. Der Codeblock innerhalb des Scopes kann damit die Ausgabewerte am Ende der Berechnung in die entsprechenden Variablen schreiben. Die Eingabewerte stehen ihm ebenfalls zur Verfügung. Die Eingabevariablen werden bei der Deklaration abhängig vom angeschlossenen Parameter initialisiert. Der Variable wird entsprechend der Wert einer uniformen Variable, einer Varyingvariable oder der Ausgabevariable eines anderen Codeblocks zugewiesen. Für Eingabeparameter zum Zugriff auf Texturen (Sampler) gibt es keine Zuweisung. Auf die entsprechende uniforme Variable wird direkt im Codeblock zugegriffen.

Programmcode 6 in Anhang A gibt ein Beispiel für erstellte Codeblöcke zum Beispielknoten aus Abbildung 33. Dem Shadergraph aus Abbildung 34 entsprechende, vollständige Vertex- und Fragmentporgramme sind ebenfalls in Anhang A in Programmcode 7 und 8 zu finden.

5.2.3 Erweiterung für Schleifen

Neuere Grafikkarten erlauben den Einsatz von Kontrollflussstrukturen wie Schleifen und bedingte Anweisungen (siehe Abschnitt 2.1.3). Die Syntax der Shaderprogrammiersprachen stellt entsprechende Elemente zur Verfügung. Innerhalb eines Programmknotens können bedingte und iterative Strukturen eingesetzt werden. In der bisher vorgestellten Form ist es im Shadergraph jedoch nicht möglich, diese Strukturen knotenübergreifend einzusetzen. Mehrmaliges Ausführen eines Subgraphen innerhalb einer Schleife kann nicht realisiert werden. Im Kontext des Volumenrenderings wäre dies jedoch wünschenswert. Raycasting führt die Schritte der Pipeline innerhalb einer Schleife aus (siehe Abschnitt 2.2.3.2). Ziel des Shadergraphen ist es, die Volumenrendering-Pipeline durch modulare Codeteile darzustellen. Für einen Raycasting-Shader leitet sich daraus ab, dass die Inhalte des Schleifenrumpfs modular gestaltet werden müssen. Im Folgenden wird eine Erweiterung des Shadergraphen vorgestellt, die die Beschreibung von Schleifen erlaubt.

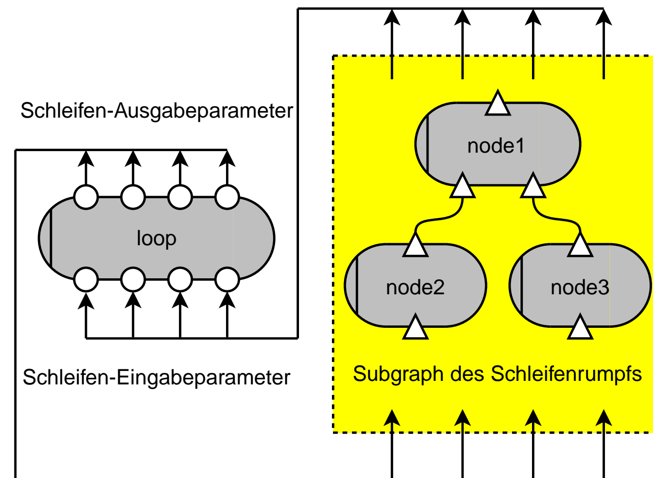


Abbildung 36: Kommunikation eines Schleifen-Programmknoten mit dem Schleifenrumpf über Schleifenparameter. Der Knoten loop kontrolliert die Schleifenausführung und wertet den Subgraph mehrmals aus.

Die Betrachtungen in Abschnitt 3.3.2 haben gezeigt, dass die Realisierung iterativer Konstrukte in visuellen Datenflusssprachen aufwändig ist. Die Darstellung von Schleifenrumpf, Laufvariablen und Abbruchbedingung durch einzelne Elemente führt zu unübersichtlichen Datenflussgraphen. Für die Beschreibung von Schleifen im Shadergraph wird daher ein einfacherer Ansatz gewählt. Laufvariablen und Abbruchbedingung werden nur auf Ebene des Programmcodes in einem speziellen Programmknoten für Schleifen gesteuert. Nur der Schleifenrumpf wird durch einzelne Elemente dargestellt.

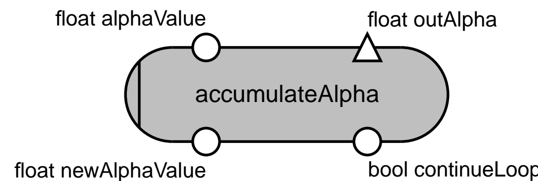
Schleifen im Shadergraph bestehen aus zwei Teilen: Einem *Schleifen-Programmknoten* und einem *Schleifenrumpf*. Der mehrmals ausgeführte Schleifenrumpf wird durch einen Subgraphen dargestellt. Er ist verbunden mit dem Schleifen-Programmknoten, der die Laufvariablen und Abbruchbedingung steuert. Die Ausführung der Schleife entspricht der mehrmaligen Auswertung der durch den Subgraph im Schleifenrumpf dargestellten Berechnungen. Der Kontrollknoten besitzt neben den bisher vorgestellten Parametern spezielle Eingabe- und Ausgabeparameter, welche die Laufvariablen repräsentieren. Diese Parameter werden mit dem Verhaltenstyp *loop* gekennzeichnet. *Schleifen-Ausgabeparameter* werden dem Schleifenrumpf bei jeder Ausführung übergeben. *Schleifen-Eingabeparameter* können von den Knoten des Subgraphs im Schleifeninneren in jedem Durchlauf mit neuen Werten versehen werden. Die Benennung geht bei der Richtung von der Sicht des Schleifen-Programmknotens aus. Seine Ausgabeparameter für Schleifen stellen Eingabeparameter für den Schleifenrumpf dar, seine Eingabeparameter werden entsprechend durch Ausgabeparameter des

Parameter				
Richtung	Programm	Verhaltenstyp	Datentyp	Name
input	fragment	loop	float	newAlphaValue
input	fragment	loop	bool	continueLoop
output	fragment	loop	float	alphaValue
output	fragment	dataflow	float	outAlpha

Codeteil Fragmentprogramm:

```
1 bool internContinueLoop = true;
2 while (internContinueLoop == true) {
3   @loopBody
4   alphaValue = newAlphaValue;
5   internContinueLoop = continueLoop;
6 }
7 outAlpha = newAlphaValue;
```

(a)



(b)

Abbildung 37: (a) Beispielhafter Schleifen-Programmknoten. `accumulateAlpha` (b) Grafische Notation für Schleifenknoten am selben Beispiel.

Rumpfs versorgt. Abbildung 36 veranschaulicht die Idee. Die Schleifenparameter stellen die Schnittstelle zum Schleifenrumpf dar (siehe Abschnitt 3.3.2).

Um eine klassische Laufvariable zu erstellen, die im Inneren der Schleife verändert wird, müssen zwei korrespondierende Ein- und Ausgabeparameter des Typs *loop* existieren. Der innerhalb der Schleife veränderte Wert des Eingabeparameters wird in den Ausgabeparameter geschrieben. Eine direkte Veränderung der Laufvariable ist nicht möglich. Eine solche Beschreibung von Laufvariablen ist in Datenflusssprachen üblich (siehe dazu [JHM04, Seite 24]).

Um einen Schleifen-Programmknoten zu realisieren, werden ihm Ein- und Ausgabeparameter vom Typ *loop* zugewiesen. Innerhalb des Programmcodes wird eine Schleife definiert. Sie setzt zu Beginn des Schleifenrumpfs die Werte der entsprechenden Schleifen-Ausgabeparameter. Am Ende werden die Schleifen-Eingabeparameter mit den Ausgabeparametern der innerhalb des Schleifenrumpfs ausgeführten Knoten versehen. Dazwischen wird die Stelle markiert, an der die Berechnungen der Knoten im Rumpf ausgeführt werden sollen. Dazu wird die Markierung `@loopBody` verwendet. Um Laufvariablen zu realisieren, muss nach jedem Schleifendurch-

lauf jeder Schleifen-Eingabeparameter in den korrespondierenden Schleifen-Ausgabeparameter geschrieben werden. So steht er im nächsten Schleifendurchlauf zur Verfügung. Diese Zuweisung kann vom Programmierer übernommen werden, falls mit den Schleifen-Eingabeparametern nach dem Schleifendurchlauf weitere Berechnungen durchgeführt werden sollen. Die Zuweisung könnte jedoch auch beim Erstellen des Programmcodes automatisch realisiert werden. Entsprechende Benennungskonventionen ermöglichen einen solchen Automatismus. Abbildung 37 zeigt einen beispielhaften Schleifen-Programmknoten sowie eine entsprechende grafische Notation. Parameter vom Typ *loop* werden durch ein eigenes Symbol (Kreis) dargestellt.

Die Erstellung der Shaderprogramme aus dem Shadergraph wird um die Berücksichtigung der Schleifenknoten erweitert. Dazu wird der Programmcode für den Schleifenrumpf separat erstellt, und an die Stelle der Markierung `@loopBody` gesetzt. Zunächst muss jedoch bestimmt werden, welche Knoten zum Schleifenrumpf gehören. Nur Knoten, die direkt oder indirekt von einem Ausgabeparameter des Verhaltenstyps *loop* abhängen, müssen innerhalb der Schleife ausgeführt werden (siehe Abschnitt 3.3.2). Alle anderen Knoten sind unabhängig von Schleifenparametern, und können einmalig vor der Schleife berechnet werden. Knoten im Schleifenrumpf können auf diese Weise normale Parameter von Knoten außerhalb der Schleife verwenden. Die Bestimmung der Ausführungsreihenfolge der Knoten bleibt unverändert. Schleifen-Programmknoten und Knoten im Subgraph des Rumpfs werden zusammen topologisch sortiert. Dies ist trotz der separaten Codeerstellung für Knoten im Schleifenrumpf möglich, da es nur auf die Reihenfolge der Sortierung ankommt, und nicht auf die Abstände zwischen den ermittelten Pfadtiefen pro Knoten.

Die Traversierung des Graphen wird erweitert um die Bestimmung der Zugehörigkeit von Knoten zu einem Schleifenrumpf. Für jeden Ausgabeparameter vom Verhaltenstyp *loop* wird dazu eine weitere Tiefensuche gestartet. Sie verfolgt die Verbindungen in umgekehrter Richtung. Der Schleifenknoten wird zur Wurzel eines neuen Graphen, der von unten nach oben wächst. Alle Knoten werden als zugehörig zur Schleife markiert. Mit diesem einfachen Mechanismus können Knoten nur zu einer Schleife gehören. Für verschachtelte Schleifen wäre ein komplexerer Algorithmus notwendig.

Die Ausgabeparameter von Knoten des Schleifensrumpfs müssen direkt oder indirekt zu Eingabeparametern vom Typ *loop* des Schleifenknotens führen. Es darf ausschließlich solche Pfade geben. Wird diese Bedingung verletzt, so kann die Schleife nicht korrekt konstruiert werden. Der Schleifenrumpf wäre dann nicht zwischen Ein- und Ausgabeparametern vom Typ *loop* eingeschlossen, wodurch der Rumpf nicht eindeutig identifiziert werden kann.

Durch Schleifen werden Zyklen im Graphen erzeugt (siehe auch Ab-

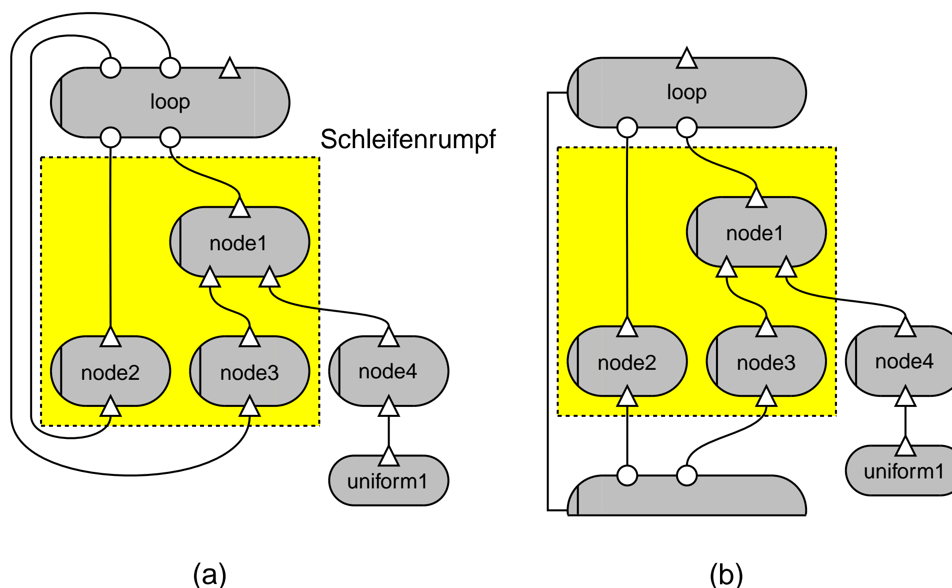


Abbildung 38: (a) Shadergraph mit Schleife. (b) Shadergraph in übersichtlicherer Notation, mit markiertem Schleifenrumpf.

schnitt 3.3.2). Damit handelt es sich bei Shadergraphen mit Schleifen um ungerichtete, zyklische Graphen. Bei der Traversierung muss sichergestellt werden, dass Schleifen nicht unendlich oft verfolgt werden. Abbildung 38 zeigt einen beispielhaften Teil eines Shadergraphs mit Schleife und markiertem Schleifenrumpf. In Abbildung 38a ist zu erkennen, die der Zyklus Kanten erzeugt, die sich über große Teile des Graphen erstrecken. Dies ist eine potentielle Quelle für unübersichtliche Graphen, z. B. wenn es viele Schleifenvariablen gibt oder der Schleifenrumpf sehr umfangreich ist. Abbildung 38b stellt eine Notation für Schleifenknoten mit verbesserter Übersicht vor. Die Ausgabeparameter der Schleife werden an einem eigenen *Pseudoknoten* unterhalb des Schleifenrumpfs dargestellt, der über eine einzige Kante mit dem Schleifenknoten verbunden ist.

5.2.4 Einsatz im Volumenrendering

Die Anforderungen an ein modulares Shaderframework für Volumenrendering lassen sich mit Hilfe der vorgestellten Funktionalität eines Shadergraphen erfüllen. Dazu verwaltet der Volumenrenderer seine internen Shaderprogramme in Form von Shadergraphen. Abhängig vom Umfang des Renderers sind verschiedene Möglichkeiten denkbar. Für kleine Programme zur Volumenvisualisierung ist der Einsatz mehrerer fest definierter Shadergraphen praktikabel. Komplexere Renderer können den internen Shadergraph abhängig von Einstellungswerten aufbauen. Aus einer festgelegten Anzahl von Knoten oder Subgraphen wir der gesamte Graph erst zur Laufzeit erstellt.

Die Erstellung des GLSL-Codes aus dem Shadergraph geschieht zur Laufzeit. Ändert sich der eingesetzte Shadergraph oder Teile davon, so wird er erneut verarbeitet und auf der Grafikkarte aktiviert. Änderungen am eingesetzten Shadergraph wirken sich direkt aus. Entsprechende visuelle Auswirkungen können direkt beobachtet werden. Somit wird ein Rapid Prototyping von Shaderprogrammen ermöglicht.

Der Renderer exportiert Informationen über den verwendeten Shadergraph. Der Anwender hat die Möglichkeit, auf die verwendete Datenbank von Knotenmustern zuzugreifen. Der aktuell verwendete Shadergraph ist ebenfalls vom Anwender einsehbar, sowie der daraus erstellte GLSL-Programmcode. Für den Shader verwendbare Einstellungen des Renderers, wie z. B. die Samplingrate, werden als Uniformknoten zur Verfügung gestellt. Der Anwender bekommt auf diese Weise umfangreiche Informationen über die aktuell eingesetzten Shadergraphen.

Der Renderer stellt zusätzlich eine Schnittstelle zur Verfügung, die es dem Anwender erlaubt, eigene Subgraphen und Knoten zu definieren und dem Graphen hinzuzufügen. Neue Wurzelknoten können definiert, und existierende entfernt werden. Durch die Definition von Verbindungen zu Knoten des internen Graphen kann dieser verändert werden. Dazu werden die in Abschnitt 5.2.1.7 beschriebenen Operationen verwendet. Der Anwender kennt die Struktur des internen Graphen und kann gezielt Verbindungen zu existierende Knoten erstellen. Die angebotenen Uniformknoten liefern wichtige Informationsquellen für die Berechnungen der eigenen Subgraphen. Externe Ressourcen wie vom Anwender erstellte Texturen oder uniforme Variablen können als uniforme Uniformknoten dem Shadergraph hinzugefügt werden. Sie stehen dann dem gesamten Graphen zur Verfügung. Es ist möglich, einen Großteil des existierenden, internen Shadergraphen weiterzuverwenden. Nur die vom Anwender angepassten Teile müssen ersetzt werden.

Der Renderer speichert die hinzugefügten Subgraphen, sowie die Operationen zu deren Verbindung mit dem internen Graph, separat ab. Wird der interne Shader verändert, so werden die gespeicherten Veränderungen erneut angewendet. Dadurch bleiben die Veränderungen durch den Anwender auch bei veränderten Einstellungen des Volumenrenderers erhalten. Der im Endeffekt entstandene Graph wird zur Erzeugung der GLSL-Shader verwendet. Die Verarbeitung des Shadergraphen zur Laufzeit ermöglicht es, die Effekte der Veränderungen direkt zu beobachten. Damit ist eine dynamische Entwicklung der Shader möglich.

In Abschnitt 5.1.1 wurde die Problematik der Koordination von interner Shaderbeschreibung und externer Veränderung beschrieben. Die komplette Ersetzung des Shaders führt dazu, dass Anpassungen des internen Shaders überschrieben werden. Dieses Problem besteht bei Einsatz des Shadergraphen nicht mehr. Alle Teile des Shadergraphen, die vom Anwender unberührt bleiben, unterliegen nach wie vor der Kontrolle des Volumen-

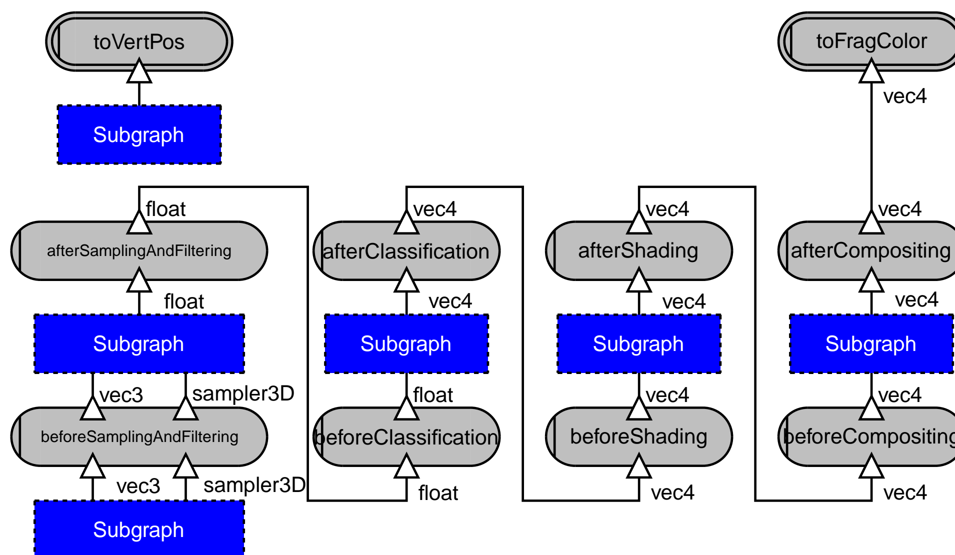


Abbildung 39: Knoten des Grundgerüsts eines Shadergraphen für Volumenrendering. Subgraphen sind austauschbar.

renderers. Dort können Veränderungen gemacht werden, die sich im visuellen Ergebnis niederschlagen. Einzig interne Veränderungen an Teilen des Shadergraphen, die später von Anpassungen des Anwenders ersetzt werden, wirken sich nicht mehr aus. Dieser Effekt ist gewünscht, denn die Veränderungen des Anwenders sollen sich immer durchsetzen.

Die Volumenrendering-Pipeline ist im Shadergraph aus Abbildung 34 implizit modelliert. Wird der Shadergraph zur Laufzeit erstellt, kommen abhängig von den Einstellungen bestimmte Knoten nur manchmal zum Einsatz. Veränderungen des Anwenders verlieren dadurch unter Umständen ihre Wirksamkeit. Dies ist der Fall, wenn der Anknüpfungsknoten durch veränderte Einstellungen aus dem internen Shadergraph verschwindet. Um diese Probleme zu umgehen, wird ein Grundgerüst für Shadergraphen im Volumenrendering definiert. Es handelt sich um eine Sammlung von Knoten, die im Shadergraph des Renderers immer vorhanden sind. Dieses *Grundgerüst* orientiert sich an der Volumenrendering-Pipeline. Abbildung 39 zeigt das Standardgerüst für einen Volumenrenderer für Intensitätsvolumen. An den Grenzen der einzelnen Schritte werden *Grenzknoten* eingefügt, die selbst keine Berechnungsaufgabe erfüllen. Für jeden Schritt der Pipeline gibt es einen Knoten, der ihn abschließt. Er trägt den Namen des Schritts, versehen mit dem Schlüsselwort *after*, z. B. *afterClassification*. Vor jedem Schritt gibt es ebenfalls einen Knoten, der den Namen der Pipelinestufe zusammen mit dem Schlüsselwort *before* trägt, z. B. *beforeClassification*. Für jeden Pipelineschritt gibt es einen eigenen Subgraphen, der durch ein solches

Knotenpaar eingeschlossen wird. Ausnahmen bilden die Schritte Sampling und Filterung. Sie werden im Shadergraph zu einem Schritt zusammengefasst. Die Zusammenfassung ist damit begründet, dass bei echtzeitfähigem Volumenrendering auf der GPU meist die Filterung der Grafikhardware verwendet wird. Zudem hängen Sampling und Filterung stark zusammen (siehe Abschnitt 2.2.2.2) Im Shadergraph gibt es dann keinen eigenen Teil für die Filterung. Die Grenzknoten geben die Eingabe- bzw. Ausgabewerte der jeweiligen Stufe weiter. Die Datentypen, die von den Grenzknoten weitergegeben werden, müssen bei Bedarf für einen konkreten Volumenrenderer noch angepasst werden (z. B. falls im Volumen Farbwerte statt Intensitätswerte stehen).

Der Subgraph eines Pipelineschrittes kann vom Anwender somit einfach ausgetauscht werden. Er definiert einen eigenen Subgraph, und verbindet ihn mit den entsprechenden Anfangs- und Endknoten des Gerüsts. Zusätzliche Schritte zwischen den Stufen lassen sich ebenfalls einfach einfügen, da zwischen zwei Pipelineschritten zwei Grenzknoten liegen.

Anpassungen des Anwenders sind immer wirksam, wenn sie auf diese stets vorhandenen Knoten zurückgreifen. Der Anwender braucht keine weiteren Kenntnisse über den internen Shadergraphen. Auf die exportierten Informationen über den internen Graphen muss er nur dann zurückgreifen, wenn er sehr detaillierte Anpassungen innerhalb eines Pipelineschrittes vornehmen will. Durch die Einführung eines Grundgerüsts repräsentieren Shadergraphen die Pipeline in expliziter Form. Der Anwender kann sowohl auf der Ebene der Pipeline, als auch direkt im Shadergraph flexible Änderungen vornehmen.

6 Praktische Umsetzung

Das im letzten Abschnitt vorgestellte Konzept wurde beispielhaft innerhalb des MEVISLAB-Frameworks umgesetzt. Dazu wurde die Bibliothek SHADER UTILITY LIBRARY (SUL) entwickelt, die die Erstellung und Verarbeitung von Shadergraphen erlaubt. Sie wurde in den Volumenrenderer GVR integriert, um dessen interne Shaderprogramme zu verwalten. Der Volumenrenderer ist unabhängig von MEVISLAB als Bibliothek GVRLIB realisiert. MEVISLAB fügt die *grafische Benutzungsoberfläche (GUI)* und ein Interface für MEVISLAB-Bilddaten hinzu. Durch die Integration der SUL zur Verwaltung der internen Shader wird der Volumenrenderer abhängig von der Shadergraph-Bibliothek. Bei der Erstellung der SUL konnte aus diesem Grund nicht auf MEVISLAB zurückgegriffen werden, da von Seiten der MEVIS-Gruppe die Anforderung einer MEVISLAB-unabhängigen GVR-Bibliothek besteht. Innerhalb von MEVISLAB wurde jedoch eine umfangreiche grafische Benutzungsoberfläche zur Arbeit mit den Erweiterungen erstellt. Dazu wurden entsprechende neue MEVISLAB-Module erzeugt und die GUI des GVR erweitert. Dieses Kapitel stellt wichtige Teile der Umsetzung vor und Abbildung 40 zeigt ihre Ebenen.

6.1 Shader Utility Library (SUL)

Die Bibliothek SUL ermöglicht es, Shadergraphen zu erzeugen und sie in entsprechende GLSL-Programme umzuwandeln. Sie ist in C++ realisiert, unter Verwendung einiger Hilfsbibliotheken. Ihre Klassen lassen sich in Gruppen unterschiedlicher Aufgaben aufteilen. Eine Sammlung von Datenstrukturen dient zur *Beschreibung und Speicherung* der Komponenten eines Shadergraphen und ihrer jeweiligen Eigenschaften. Aus der Beschreibung der Einzelkomponenten wird ein konkreter Shadergraph erzeugt (*Erstellung*). Diese konkrete Repräsentation enthält Topologieinformationen, da Parameter- und Verbindungsobjekte verknüpft sind, und kann traversiert werden. Die *Verarbeitung* des erstellten Graphen erzeugt die korrespondierenden GLSL-Programme. Da sich die Werte uniformer Variablen ändern können, ohne dass sich das Programm verändert, werden diese durch separate Klassen beschrieben. Abbildung 41 zeigt ein Klassendiagramm der wichtigsten Bestandteile der SUL. Die Klassen sind nach den

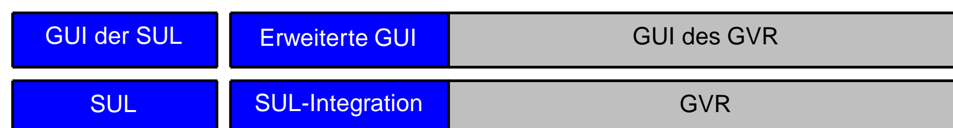


Abbildung 40: Bestandteile der Umsetzung des Shadergraph-Konzepts.

beschriebenen Gruppen aufgeteilt.

6.1.1 Beschreibung und Speicherung

Ein Shadergraph besteht aus Knoten, Parametern, Verbindungen und Wurzeldefinitionen. Diese Bestandteile haben jeweils eine Reihe von Eigenschaften. Zur Beschreibung dieser Eigenschaften stellt die SUL verschiedene Klassen zur Verfügung.

Typklassen beschreiben die in Abschnitt 5.2 vorgestellten Typen. Sie kapseln entsprechende *Enumerations*, die gültige Werte eines Typs definieren.

Beschreibungsklassen enthalten Informationen, die zur eindeutigen Beschreibung der Komponenten eines Shadergraphen notwendig sind. Dazu verwenden sie die vorgestellten Typklassen sowie Strings und Datenstrukturen der STANDARD TEMPLATE LIBRARY (STL). Beschreibungsklassen fungieren als Datenspeicher. Sie bestehen aus einem Datenverbund von außen zugänglicher Variablen (`public`-Variablen), sowie Hilfsfunktionen zur Verarbeitung der Daten.

Die Klasse `sulNodeDescription` beschreibt Programmknoten, und greift neben der Parameterbeschreibung `sulParameterDescription` auf die Klasse `sulModuleDescription` zurück. Diese Klasse beschreibt Knotenmuster, deren Instanzen Programmknoten sind. In Knotenmuster-Datenbanken (`sulModuleDatabase`) können existierende Muster mit eindeutigem Namen gespeichert werden. Sie stehen zur Erzeugung von Knoten als Informationsquelle zur Verfügung.

Beschreibungen von Komponenten eines Shadergraphen können zusammengefasst werden. Eine *Komponentenliste* (`sulComponentList`) besteht aus linearen Listen von Beschreibungen für Programmknoten, uniformen Parametern, Verbindungen und Wurzeln. Eine Komponentenliste kann als Sammlung von Befehlen verstanden werden, die das Anlegen von Knoten, Verbindungen und Wurzeln im Graphen verursachen. Diese Befehle werden bei der Erstellung des Graphen verarbeitet (Abschnitt 6.1.2), und erst dort auf Konsistenz überprüft. Wurzelbeschreibungen der Klasse `sulRootDefinition` können Wurzeln hinzufügen und entfernen. Verbindungsbeschreibungen der Klasse `sulConnectionId` können existierende Verbindungen überschreiben, wenn dies durch ein entsprechendes *Flag* gekennzeichnet ist. Durch überschreibende Verbindungen werden neue Subgraphen in den Graph integriert.

Mehrere Komponentenlisten werden zu *Programmbeschreibungen* zusammengefasst (Klasse `sulProgramDescription`). Eine Programmbeschreibung sammelt Komponenten eines einzigen Shadergraphs. Sie referenziert die Komponentenlisten. Eine Kopie wird nicht erstellt. Die referenzierten Komponentenlisten werden nacheinander verarbeitet. Durch das Hinzufügen einer vom Anwender erstellten Komponentenliste kann die Beschreibung eines existierenden Shadergraphen nachträglich verändert werden.

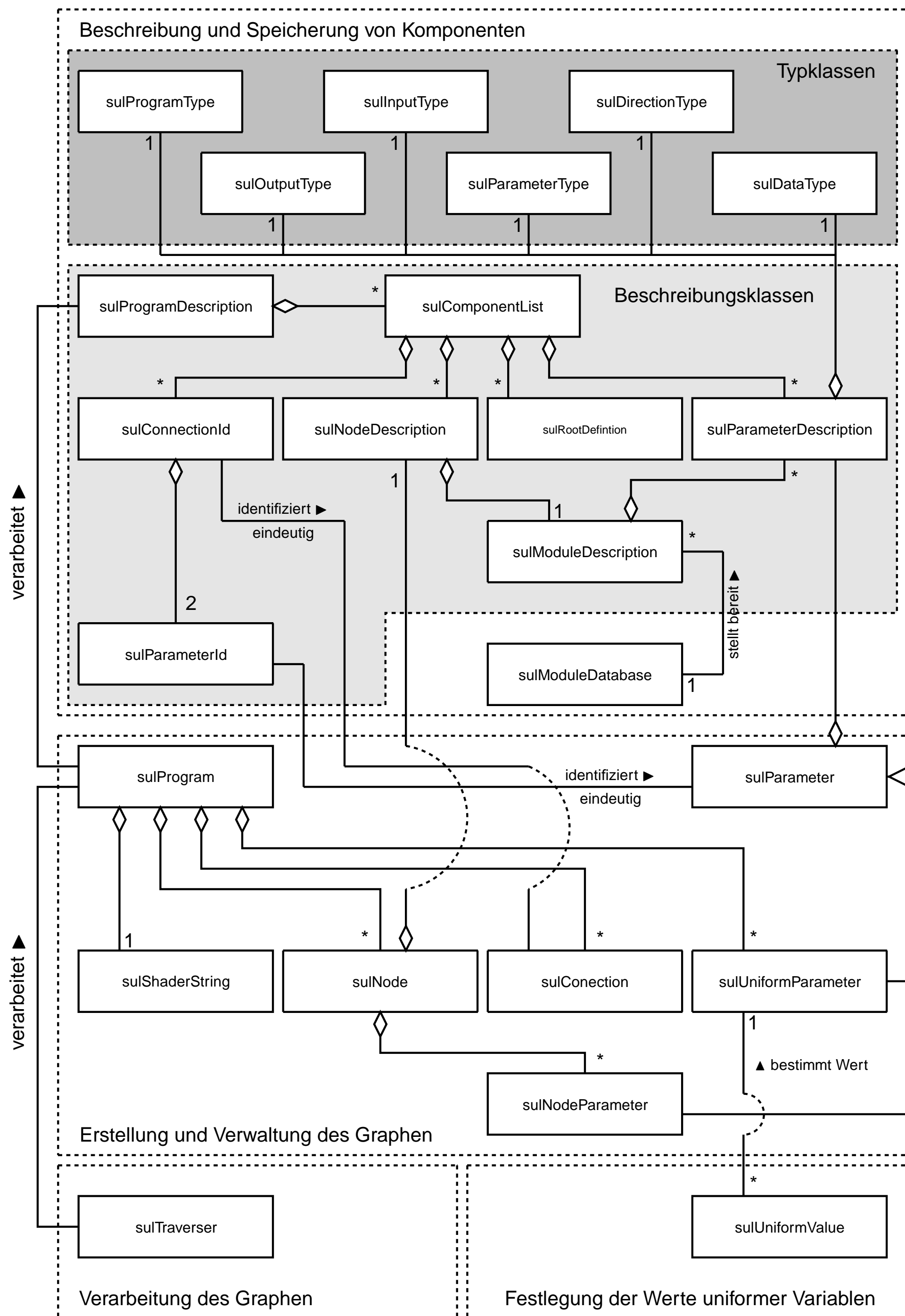


Abbildung 41: Klassendiagramm der wichtigsten Bestandteile der SUL.

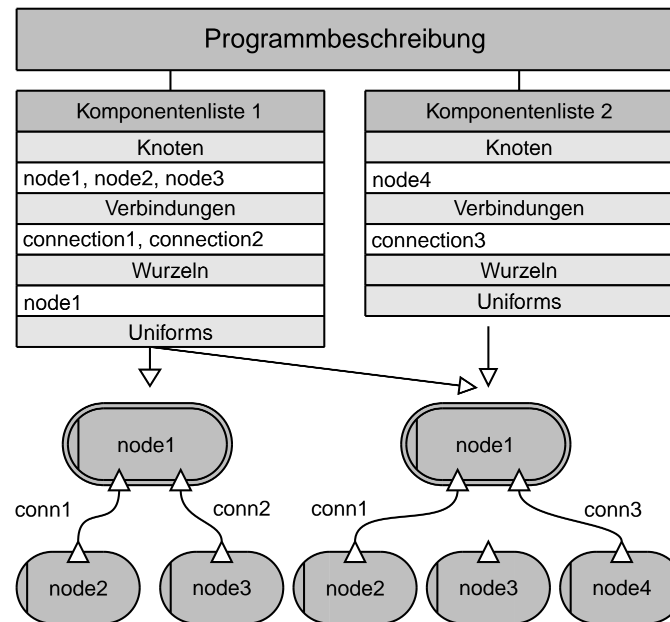


Abbildung 42: Programmbeschreibung aus zwei Komponentenlisten, sowie die nach Verarbeitung der jeweiligen Listen entstandenen Graphen. Die Verbindung *connection3* überschreibt *connection2*.

Abbildung 42 zeigt ein Beispiel für Komponentenlisten und Programmbeschreibungen.

Programmbeschreibungen und Komponentenlisten enthalten eine Vielzahl an Elementen. Eine Weitergabe durch Kopieren würde hohe Laufzeit- und Speicherkosten zur Folge haben. Aus diesem Grund existieren Verwaltungsobjekte, über die der Zugriff auf und die Erzeugung der Programmbeschreibungen und Komponentenlisten gesteuert werden. Verwaltungsobjekte werden mit dem *Singleton-Pattern* [GHJV95] global verfügbar gemacht. Jeder Programmbeschreibung und jeder Komponentenliste wird eine eindeutige Identifikationsnummer in Form einer `unsigned int` Variable zugeordnet. Statt einer Kopie kann dann diese Identifikationsnummer weitergegeben werden. Programmbeschreibungen nutzen die Identifikationsnummern von Komponentenlisten, um sie zu referenzieren.

Programmbeschreibungen, und damit die repräsentierten Shadergraphen, sind identisch, wenn sie dieselben Komponentenlisten in derselben Reihenfolge referenzieren. Komponentenlisten beinhalten eine Statusvariable vom Typ `unsigned int`. Sie wird bei Veränderungen der Liste inkrementiert. Programmbeschreibungen speichern diesen Wert, wenn eine Komponentenliste hinzugefügt wird. Somit kann durch den Vergleich von gespeichertem und aktuellem Statuswert überprüft werden, ob sich eine Komponentenliste seit dem Hinzufügen zur Programmbeschreibung ver-

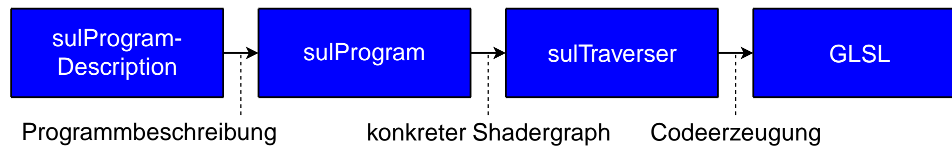


Abbildung 43: Verarbeitungsreihenfolge bei der Erzeugung von GLSL-Code aus einer Programmbeschreibung.

ändert hat. Eventuell vorher durchgeführte Verarbeitungsschritte sind dann ungültig.

Für die flexible Arbeit mit der SUL ist die Möglichkeit der Speicherung erzeugter Beschreibungen wichtig. Dazu wurde eine Serialisierung in XML mit Hilfe der TinyXML-Bibliothek realisiert [22] . Konvertierungsfunktionalität wird auf diese Weise in den entsprechenden Beschreibungs- und Typklassen zur Verfügung gestellt.

6.1.2 Erstellung und Verarbeitung

Aus einer Programmbeschreibung wird ein konkreter Shadergraph erzeugt, der dann zur Erzeugung des GLSL-Programmcodes verarbeitet wird. Abbildung 43 gibt einen Überblick über den Ablauf.

Die Repräsentation eines Shadergraphs ist ein Objekt der Klasse `sulProgram`. Es verarbeitet die Komponentenlisten, und erzeugt für jede der enthaltenen Komponentenbeschreibungen eine entsprechende Komponente im Graph. Bei der Verarbeitung führt es auch die Korrektheitsüberprüfung der angelegten Komponenten durch. Existiert ein Knotenname schon, oder ist eine Verbindung ungültig, so wird die Komponente nicht erzeugt. Der Shadergraph in `sulProgram` enthält Topologieinformationen: Die Objekte der Parameterklasse `sulParameter` und der Verbindungsklasse `sulConnection` werden über Zeiger miteinander verknüpft. Eine Traverserierung des erstellten Graphen ist somit direkt möglich. Kanten können von Vorgänger zu Nachfolger, und umgekehrt, verfolgt werden. Uniforme Parameter und Knotenparameter sind von der Basisklasse `sulParameter` abgeleitet und können miteinander verbunden werden.

Die Klasse `sulTraverser` kapselt die Erzeugung der GLSL-Programme aus dem Shadergraph. Sie stellt eine direkte Umsetzung des in Abschnitt 5.2.2 vorgestellten Verarbeitungskonzepts dar. Im ersten Schritt wird die topologische Sortierung und Zuordnung zu Schleifenrümpfen erzeugt. Dazu werden die in Abschnitt 5.2.2.1 beschriebenen Tiefensuchverfahren eingesetzt. Die topologische Sortierung wird so vorgenommen, dass Knoten gleicher Tiefe im durch den zweiten Schritt erzeugten Code nah beieinander stehen. Bei der Codeerstellung wird die STL für String-Operationen

[22] http://www.grinninglizard.com/tinyxml/. Zugriff: 15.01.2008

eingesetzt. Weiterhin wird umfangreich auf Textersetzung zurückgegriffen, z. B. zur Erweiterung von Variablennamen mit einem Knotennamen. Dazu werden *Regular Expressions* eingesetzt, die Textveränderung und -ersetzung mit flexiblen Mustern erlauben. Für den Einsatz von *Regular Expressions* in C++ wird die Bibliothek BOOST [23] eingesetzt. Rumpf und Header der erzeugten Shaderprogramme werden in jeweils einem String abgelegt, die in der Klasse `sulShaderString` gekapselt sind. Nach der Verarbeitung kann der erzeugte Text dem GLSL-Compiler übergeben und auf der Grafikkarte aktiviert werden.

Zur Festlegung der Werte uniformer Variablen gibt es eine Gruppe von Klassen, die von `sulUniformValue` abgeleitet sind. Dabei gibt es für jeden Typ uniformer Variablen eine Klasse. Objekte der Klassen werden mit Namen und Wert initialisiert. Zur Laufzeit der Shaderprogramme können diese Objekte den Wert der zugehörigen uniformen Variable setzen. Werte von uniformen Variablen können so unabhängig vom Shadergraph festgelegt werden. Die unabhängige Festlegung entspricht der Idee uniformer Variablen, deren Verwendung im Shadercode von der Bestimmung ihrer Werte getrennt ist.

6.2 Integration der SUL in den GVR

Die GVRLIB ist modular und objektorientiert aufgebaut. Zur Erstellung und Verwaltung des Programmcodes für Vertex- und Fragmentshader existiert eine Basisklasse. Konkrete Klassen zur Shadererzeugung sind von ihr abgeleitet und können daher leicht ausgetauscht werden. Der GVR wurde um eine derartige abgeleitete Klasse erweitert, die Shader mit Hilfe der SUL verwaltet. Alle Veränderungen am GVR sind so gestaltet, dass der GVR unverändert funktioniert, solange die Erweiterungen nicht explizit aktiviert werden.

6.2.1 Verwaltung des Shadergraphs

Für die Umsetzung der internen Shader mit einem Shadergraph wurde die Klasse `gvShaderIlluminatedSUL`[24] erstellt. Sie ist von der Basisklasse `gvShaderGLSL` des GVRs für GLSL-Shader abgeleitet. Sie verarbeitet einen Shadergraph und stellt mit Hilfe des Graphen erstellten GLSL-Programmcode zur Verfügung. Zur Aktivierung der GLSL-Shader auf der Grafikkarte konnte die Funktionalität der Basisklasse weiterverwendet werden. `gvShaderIlluminatedSUL` ersetzt die Klasse `gvShaderIlluminatedGLSL`, die im GVR für die Verwaltung der GLSL-Shaderprogramme

[23] `http://www.boost.org/`. Zugriff: 15.01.2008

[24] Die Benennung mit dem Zusatz `Illuminated` orientiert sich an den im GVR vorhandenen Klassen zur Shaderverwaltung.

zuständig ist. Für Vergleiche wird auf diese Klasse weiterhin unter der Bezeichnung *Standard-Shaderverwaltung* des GVR Bezug genommen.

Die Standardfunktionalität des Renderers, die durch seine Einstellungen und ohne Manipulation des Shadergraphen durch den Anwender genutzt werden können soll, wird auf C++-Ebene festgelegt. Der Entwickler des Renderers muss alle Variationen dieser Standardfunktionalität also einprogrammieren. Die erstellte Shaderklasse pflegt dazu eine Komponentenliste für den internen Shader. Knoten, Verbindungen und Wurzeldefinitionen werden dieser Liste abhängig von den Einstellungen des Renderers hinzugefügt. Komponenten werden anhand ihrer Aufgabe gruppiert. Zusammengehörende Komponenten, wie die Knoten und Verbindungen des Grundgerüsts, werden der Komponentenliste immer gemeinsam hinzugefügt. Für jede Gruppe existiert eine entsprechende C++-Funktion, welcher ein Zeiger auf die zu erstellende Komponentenliste übergeben wird. Beispiele für diese *Komponentenfunktionen* sind `addShaderSkeleton` oder `addDirectionalLighting`.

Die Liste der internen Komponenten wird einer Programmbeschreibung hinzugefügt. Daraus wird der Shadergraph erzeugt und verarbeitet. Der erzeugte Programmcode wird auf der Grafikkarte aktiviert. Neben der Verwaltung der Shadergraphen und -programme hat die Klasse `gvShaderIlluminatedSUL` eine weitere Aufgabe. Die Werte der uniformen Variablen müssen abhängig von den aktuellen Einstellungen der Renderers gesetzt werden. Dazu setzt die Klasse die entsprechenden Teile der SUL ein.

6.2.2 Schnittstellen

`gvShaderIlluminatedSUL` stellt dem Anwender die Programmbeschreibung des zuletzt verwendeten Shadergraphen zur Verfügung. Dazu exportiert die Klasse die entsprechende Identifikatiosnummer. Anwender erhalten mit dieser Identifikationsnummer über das Verwaltungsobjekt Zugriff auf die Beschreibung aller Komponenten des Shadergraphs. Dies ermöglicht die Planung von Veränderungen am Shadergraph.

`gvShaderIlluminatedSUL` definiert eine Schnittstelle, mit der der internen Programmbeschreibung weitere Komponentenlisten hinzugefügt werden können. Der Anwender kann Komponentenlisten extern definieren, und den Shadergraph des GVR dadurch verändern. Mehrere externe Komponentenlisten erlauben dabei die Kombination unabhängiger Veränderungen. Die Grenzknoten des Grundgerüsts ermöglichen es zudem, in Komponentenlisten verschiedene Teile der Pipeline auszutauschen. Veränderungen im selben Teil des Grundgerüsts überschreiben einander und es ist die Aufgabe des Anwenders, solche Kollisionen zu vermeiden. Die erste Komponentenliste in der Programmbeschreibung des GVR ist immer die intern erstellte Liste. Sie enthält das Standardgerüst und die intern er-

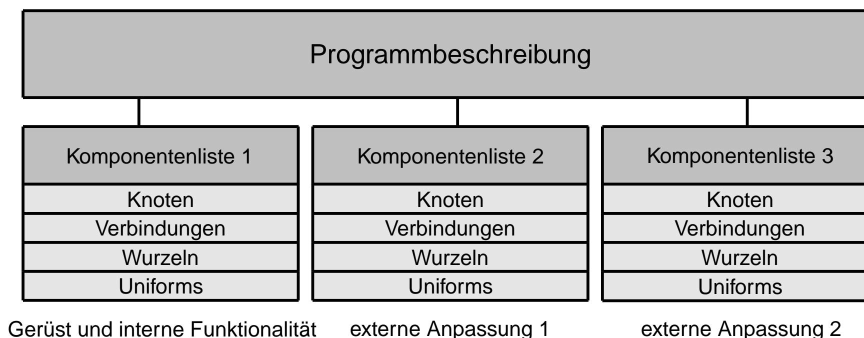


Abbildung 44: Programmbeschreibung, die von der erstellten GVR-Shaderverwaltung verwendet wird.

stellte Standardfunktionalität. Alle weiteren Komponentenlisten enthalten externe Anpassungen (siehe Abbildung 44). Durch das Entfernen der Standardwurzeln des GVR-Shadergraphs und das Hinzufügen eigener Wurzeln kann der intern verwendete Shadergraph komplett ersetzt werden.

6.2.3 Beschleunigung

Einstellungen des GVR können sich jederzeit ändern, auch ohne eine unmittelbare Benutzerinteraktion. In komplexeren Szenengraphen können sich die Einstellungen des GVRs von Rendervorgang zu Rendervorgang verändern. Ein Beispiel dafür ist eine einzige GVR-Instanz, die zur Erstellung mehrerer MPR-Ansichten unter Verwendung unterschiedlicher Einstellungen pro Ansicht eingesetzt wird. Der Shader des GVR ändert sich dann regelmässig.

Die Shaderklasse muss den Shader zu Beginn jedes Rendervorgangs an die aktuellen Einstellungen anpassen. Die Standard-Shaderverwaltung erzeugt den verwendeten Shader unter Berücksichtigung der Einstellungen bei jedem Darstellungsvorgang erneut, und greift dabei auf direkt im C++-Code hinterlegte Shaderteile zurück. Veränderte Einstellungen des Renderers werden in diesem Fall automatisch berücksichtigt. Die Kombination der Textfragmente benötigt im Vergleich zum Kompilieren des Shaders kaum Zeit. Ein Einbruch der Framerate wird vermieden, indem kompilierte Shaderprogramme gespeichert und wiederverwendet werden (*Caching*).

Erstellung und Verarbeitung einer Programmbeschreibung in `gvShaderIlluminatedSUL` ist im Vergleich zur Shadererstellung in `gvShaderIlluminatedGLSL` aufwendig. Die erneute Erstellung und Traversierung des Shadergraphs in jedem Frame lässt die Framerate stark zurückgehen (vgl. Abschnitt 7.2). Zur Vermeidung des Performanceverlusts wurde die Klasse `gvShaderIlluminatedSUL` ebenfalls um *Caching* erweitert. Shadergraphen werden nicht bei jedem Rendervorgang erzeugt,

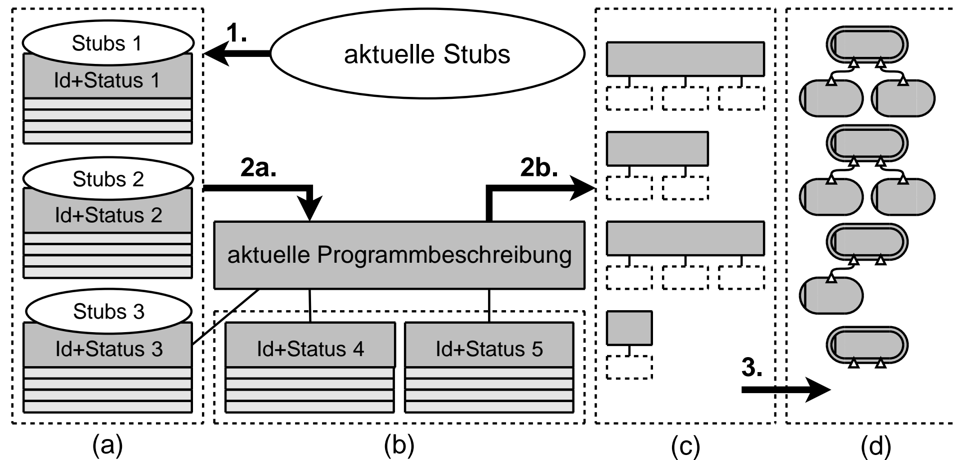


Abbildung 45: Caching in `gvShaderIlluminatedSUL`. Schritte 1-3 werden in Abschnitt 6.2.3 erläutert. (a) Gespeicherte Komponentenlisten. (b) Externe Komponentenlisten. (c) Gespeicherte Programmbeschreibungen. (d) Gespeicherte Shadergraphen inkl. erzeugten Programmcode.

sondern nur, wenn sich Einstellungen ändern. Dazu werden erstellte Komponentenlisten, Programmbeschreibungen, Shadergraphen und erstellter Programmcode gespeichert, auch wenn sie nicht mehr aktuell sind. Die Anzahl der auf diese Weise gleichzeitig gespeicherten Elemente ist einstellbar. Wird die Maximalzahl erreicht, wird das am längsten ungenutzte Element gelöscht. Bei jedem Rendervorgang wird überprüft, ob ein bereits verarbeiteter Shadergraph den aktuellen Einstellungen des GVRs entspricht. Ist dies der Fall, kann er wiederverwendet werden.

Für die Überprüfung werden Operationen verwendet, die weniger aufwändig sind als die Neuerstellung des Shadergraphen. Daher kann die Wiederverwendung gespeicherter Shadergraphen die Performance erhöhen. Die Überprüfung besteht aus mehreren Schritten (vgl. Abbildung 45):

1. Existiert eine Komponentenliste der internen Bestandteile des Shadergraphs, die weiterverwendet werden kann?

2. Existiert eine Programmbeschreibung, die der aktuell vorliegende Kombination aus interner Komponentenliste und extern verwendeten Komponentenlisten entspricht?

3. Wenn Komponentenliste und Programmbeschreibung vorhanden sind, finde den entsprechenden Shadergraph und Programmcode.

Intern erstellten Komponentenlisten wird ein eindeutiger Identifikationsstring zugeordnet. Der String kann unabhängig von der eigentlichen

Liste erzeugt werden. Am Anfang des Überprüfungsprozesses wird nur der Identifikationsstring erzeugt und die Existenz einer entsprechenden Liste geprüft. Diese wird ggf. weiterverwendet.

Zum Aufbau des Identifikationsstrings wird jeder Komponentenfunktion ein kurzer, für sie eindeutiger String hinzugefügt (*Stub*). Diese Stubs bilden zusammengefügt den Identifikationsstring. Werden die Funktionen ohne übergebene Komponentenliste aufgerufen, so werden nur die Stubs erzeugt. Es ist die Aufgabe des Entwicklers der Klasse `gvShaderIlluminatedSUL`, die Stubs für jede Funktion eindeutig zu gestalten. Weiterhin muss sichergestellt werden, dass eine Komponentenfunktion immer dieselben Komponenten zur Liste hinzufügt. Dann ist sichergestellt, dass ein Identifikationsstring eine Komponentenliste eindeutig identifiziert.

Ob die aktuelle Programmbeschreibung des Shadergraphs schon verarbeitet wurde, wird durch einen Vergleich mit den gespeicherten Programmbeschreibungen überprüft. Zwei Programmbeschreibungen sind identisch, wenn die Anzahl, Reihenfolge, Identifikationsnummern und Statusvariablen ihrer Komponentenlisten übereinstimmen. Veränderungen existierender Komponentenlisten bewirken eine Inkrementierung der Statusvariable und werden somit erkannt. Die Überprüfung wird nur durchgeführt, wenn schon eine gespeicherte und gültige interne Komponentenliste gefunden wurde. Nur dann kann eine gespeicherte Programmbeschreibung existieren.

Zwischen Programmbeschreibungen und Shadergraphen existiert eine direkte Zuordnung. Mit Hilfe der gespeicherten Programmbeschreibung kann der entsprechende Graph sofort identifiziert werden. Der ebenfalls noch vorhandene GLSL-Code kann weiterverwendet werden.

6.3 Grafische Benutzungsschnittstelle

Die entwickelte Bibliothek SUL und ihre Integration in den GVR wurde innerhalb der visuellen Programmierumgebung von MeVisLab verfügbar gemacht. Dazu wurden entsprechende MeVisLab-Module erstellt. Sie erfüllen zwei Aufgaben. Zum einen können Shadergraphen unabhängig vom Volumenrenderer grafisch erstellt werden. Zum anderen können vom GVR intern verwendete Shadergraphen dargestellt und manipuliert werden.

Die Grundidee der Integration von Shadergraphen in MeVisLab besteht aus deren Repräsentation durch die in Kapitel 4.1 vorgestellten Module und Verbindungen. Je ein Knoten eines Shadergraphs, und damit seine Codeteile, wird durch ein entsprechendes Modul im Netzwerk repräsentiert. Verbindungen zwischen Modulen definieren Verbindungen im Shadergraph. Die Verarbeitung des Netzwerks erzeugt eine Komponentenliste der Elemente im Shadergraph, aus der der konkrete Graph erzeugt wird (Instanz von `sulProgram`). Er wird nach der Verarbeitung des Netzwerks ausgewertet zu einem Shaderprogramm.

Die folgenden Abschnitte gehen auf die Idee hinter der realisierten grafischen Benutzungsschnittstelle ein. Beispiele und Screenshots der Elemente der GUI liefert auch das nächste Kapitel.

6.3.1 Auswahl eines MEVISLAB-Modultyps

Shadergraphen und MEVISLAB-Netzwerke teilen ähnliche Konzepte. Knoten und Module haben Parameter bzw. Ein- und Ausgänge sowie Felder, die verbunden werden können. Aufgrund dieser Ähnlichkeit erscheint eine direkte Realisierung von Knoten als Modul auf den ersten Blick möglich. Knotenparameter würden in diesem Fall durch Felder oder im Bezug auf den Datentyp flexible Ein- und Ausgänge vom Typ Base repräsentiert. Verbindungen könnten damit automatisch erzeugt werden.

Ein- und Ausgänge von MEVISLAB-Modulen müssen jedoch im Programmcode (C++ oder MDL) des Moduls definiert werden. Alle Instanzen eines Moduls haben die gleiche Art und Anzahl an Ein-/Ausgängen und Feldern. Sie können einer Instanz nicht zur Laufzeit hinzugefügt oder von ihr entfernt werden. Für die Repräsentation unterschiedlicher Knoten, deren Parameteranzahl sich unterscheiden kann, wäre dies notwendig. Statt der dynamischen Erzeugung von Feldern oder Ein-/Ausgängen ist die Erstellung einer größeren, festen Anzahl im Programmcode denkbar. Diesen könnten Parameter des repräsentierten Knotens zur Laufzeit zugewiesen werden. Dieser Ansatz erscheint jedoch unübersichtlich, da sich mit der Belegung auch der Datentyp der Felder und Ein-/Ausgänge ändern würde. Für viele Modulinstanzen würde auch eine große Anzahl „leerer“ Felder/Ein-/Ausgänge dargestellt werden.

Die vorhandenen MEVISLAB-Modultypen eignen sich daher nur bedingt für die Repräsentation von Knoten eines Shadergraphs. Die Erstellung eines neuen Modultyps mit geeigneter Funktionalität wurde als zu umfangreich für die Realisierung innerhalb dieser Diplomarbeit eingeschätzt. Ein solcher Modultyp könnte sich ggf. an der in Abschnitt 5.2 vorgestellten grafischen Notation orientieren. Der Typ sollte nicht auf Shadergraphen beschränkt sein. Vielmehr könnte er Module beschreiben, deren Felder und Ein-/Ausgänge zur Laufzeit veränderbar sind, und die sich zu traversierbaren Graphen zusammenschließen lassen.

In dieser Arbeit wurde daher eine Lösung auf Basis der vorhandenen Modultypen erstellt. Dabei werden Parameter und Verbindungen nicht über Felder bzw. Ein-/Ausgänge dargestellt. Die Realisierung wird im nächsten Abschnitt beschrieben. Zuvor werden die Vor- und Nachteile der Modultypen kurz dargestellt:

Inventormodule: Diese Module werden zu einem Szenengraph kombiniert, der gut traversiert werden kann. Der *Inventorstate* ist eine beim Traversieren gepflegte Sammlung von Variablen. Er ist für alle verarbei-

teten Module im Szenengraph zugänglich und erlaubt so die Weitergabe von Informationen beim Traversieren. Der Aufbau des Shadergraph aus dem Netzwerk von Inventormodulen ist mit diesen Werkzeugen einfach umsetzbar.

ML-Module: Bildverarbeitungsmodule werden zu Datenflussgraphen kombiniert. Die Datenweitergabe erfolgt lokal zwischen direkten Nachbarn im Graph. Datenflussgraphen werden nicht traversiert, und erlauben keinen globalen Austausch von Daten zwischen den Modulen. Die Verarbeitung eines Netzwerks aus Bildverarbeitungsmodulen zum Aufbau des Shadergraphen würde daher im Vergleich zu Inventormodulen erhöhten Implementierungsaufwand bedeuten.

Für die entwickelten Module in dieser Arbeit wurden Inventormodule gewählt, weil die bei Inventor-Szenengraphen durchgeführte Traversierung eine einfache Verarbeitung der Module erlaubt. Durch die Weitergabe von Informationen im Inventorstate ist es einfach möglich, aus allen SoSUL-Modulen eines Szenengraphen eine Komponentenliste der repräsentierten Shadergraph-Elemente zu erzeugen. Bildverarbeitungsmodule sind aufgrund ihres Datenflusskonzepts Shadergraphen zwar ähnlich, erlauben aber keine einfache Verarbeitung des gesamten Graphen. Die Erzeugung einer Komponentenliste aus Bildverarbeitungsmodulen würde großen zusätzlichen Implementierungsaufwand bedeuten.

6.3.2 Realisierung

Zur Integration von Shadergraphen in MeVisLab wurden Module in zwei Gruppen erstellt: Zum einen GVR-unabhängige Module, zum anderen GVR-spezifische Module. Shadergraphen können unabhängig vom GVR mit Modulen des Projekts SoSUL erstellt werden. Die MeVisLab-Module des GVR wurden um eine Schnittstelle für Shadergraphen erweitert (Projekt SoGVR).

Bei der Erstellung der Inventormodule wurde auf die Funktionalität von Open Inventor und auf die Funktionalität der MDL zur Beschreibung von Panels zurückgegriffen. Für die Kommunikation zwischen Modulpanels und den Funktionen in C++ wurden Felder eingesetzt. Deren Werte können von beiden Seiten verändert werden. Die SUL verwendet eigene zusammegesetzte Datentypen, die auch in den erstellten Modulen umfangreich eingesetzt werden. Für sie wurden eigene Feldklassen erstellt. Zusammen mit ihrer Serialisierbarkeit in XML werden diese Datentypen als Felder automatisch im Netzwerk von MeVisLab mit abgespeichert. Dadurch konnten die erstellten Module und ihr aktueller Zustand mit *Persistenz* versehen werden. Sie lassen sich im Netzwerk speichern und laden. Selbst erstellte Feldtypen können im Panel nicht direkt angezeigt werden,

Established parameter connections

Program	Datatype	Parameter		From Parameter	From Node
vertex	int	inIntensityUnit	<-	intensityUnit	-
fragment	sampler3D	inVolume	<-	intensityTexture	-
fragment	float	inSliceDistanceFrag	<-	sliceDistance	-
vertex	float	inSliceDistance	<-	sliceDistance	-
fragment	vec3	inTexCoord1	<-	outTextureCoord	volumePos
fragment	sampler2D	inPreIntegratedLUT	<-	preIntegratedLUT	-

Parameter connection definition

Node inputs

Program	Datatype	Flavour	Name
vertex	int	dataflow	inIntensityUnit
fragment	sampler2D	dataflow	inPreIntegratedLUT
vertex	float	dataflow	inSliceDistance
fragment	float	dataflow	inSliceDistanceFrag
fragment	vec3	dataflow	inTexCoord1
fragment	sampler3D	dataflow	inVolume

Matching child outputs

Conn.	Program	Datatype	Flavour	Name	Child
x	any	sampler2D	dataflow	preIntegratedLUT	-

Children

Connected Nodes

Type	Name
interpolateVolumePosition	volumePos

Child outputs

Program	Datatype	Flavour	Name	Child
any	vec3	varying	outTextureCoord	volumePos
any	sampler3D	dataflow	intensityTexture	-
any	int	dataflow	intensityUnit	-

Abbildung 46: Verbindungsdefinition im Panel der Module, die Knoten repräsentieren.

da sie in der MDL nicht bekannt sind. Aus internen Datenstrukturen können im C++-Code jedoch Strings erzeugt und mit Hilfe der MDL als Tabellen oder Textfelder im Panel angezeigt werden.

Funktionen der Module, welche die internen Datenstrukturen bearbeiten und aktualisieren, werden durch *Sensoren* ausgelöst. Diese Objekte werden von OPEN INVENTOR zur Verfügung gestellt, um Veränderungen an Feldern und der Topologie des Szenengraphen zu erkennen. Mit Hilfe von Sensoren wurde die Interaktion zwischen Panel, C++-Modul und Anwender realisiert.

Im Folgenden werden die erstellten Module vorgestellt. Beispiele und Abbildungen für Netzwerke mit den erstellten Modulen sind im Ergebniskapitel 7 dargestellt.

6.3.2.1 Knoten und Verbindungen

Knoten im Shadergraph werden jeweils durch ein Inventormodul repräsentiert. Werden zwei Module über eine Inventorverbindung zusammengefügt, wird eine Vorgänger-Nachfolger-Beziehung im Inventor-Szenengraph definiert. Diese Verbindung repräsentiert die *mögliche* Verbindung von Parametern der dargestellten Knoten. Eine konkrete Parameterverbindung wird durch die Inventorverbindung nicht festgelegt. Dies kann mit Verbindungen zwischen Knoten in abstrakten Shadetrees verglichen werden. Im Unterschied zu abstrakten Shadetrees müssen Verbindungen zwischen Parametern zusätzlich manuell festgelegt werden.

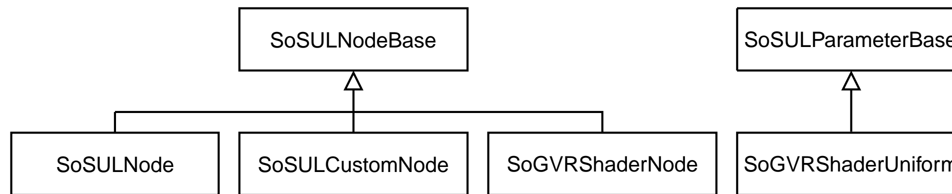


Abbildung 47: Wichtige Klassen des Projekts SoSUL.

Codeteile, Parameter und Verbindungen werden im Modul über interne Datenstrukturen verwaltet. Dazu werden die entsprechenden Datenstrukturen der SUL eingesetzt. Ein Modul besitzt eine Variable vom Typ `sulNodeDescription`, welche die Eigenschaften (Name, Parameter und Codeteile) des repräsentierten Knotens beschreibt. Jedes Modul kann Verbindungsbeschreibungen (Instanzen von `sulConnectionId`) für Verbindungen zu Nachfolgermodulen anlegen. Im Panel des Moduls werden Informationen zu Knotenbeschreibung und Verbindungsbeschreibungen angezeigt. Ebenfalls angezeigt werden Nachfolgermodule und ihre Ausgabeparameter. Diese Informationen werden aktualisiert, wenn ein Nachfolgermodul hinzugefügt oder entfernt wird. Im Panel können durch den Anwender Verbindungen definiert und entfernt werden (Abbildung 46).

Die gesamte Funktionalität zur Verwaltung von Knoten- und Verbindungsbeschreibungen sowie die entsprechenden Teile des Modulpanels wurden in der abstrakten Basisklasse `SoSULNodeBase` zusammengefasst. Davon abgeleitet sind mehrere Klassen, die konkrete Module beschreiben. Sie unterscheiden sich lediglich in der Quelle der Knotenbeschreibungen, also der Definition von Parametern und Codeteilen:

- Module der Klasse `SoSULNode` können Knotenbeschreibungen einer Knotenmusterdatenbank annehmen. Der Name des Knoten entspricht dem Modulnamen.

- Module der Klasse `SoSULCustomNode` erlauben dem Benutzer die Knotenbeschreibung völlig frei festzulegen. Durch entsprechende Teile des Panels können Parameter und Codeteile zur Laufzeit angelegt werden (Abbildung 48). Der Name des Knotens entspricht dem Modulnamen.

- Module der Klasse `SoGVRShaderNode` repräsentieren Knotenbeschreibungen eines internen Shaders des GVR. Die Aufgabe dieses Modultyps ist es, Anknüpfungen an den internen Teil eines Shadergraphen zu ermöglichen. Als Informationsquelle nutzen sie dazu die exportierte Programmbeschreibung des GVR. Ein Modul dieses Typs hat zwei Funktionsweisen. Die Knoten des Standardgerüsts können

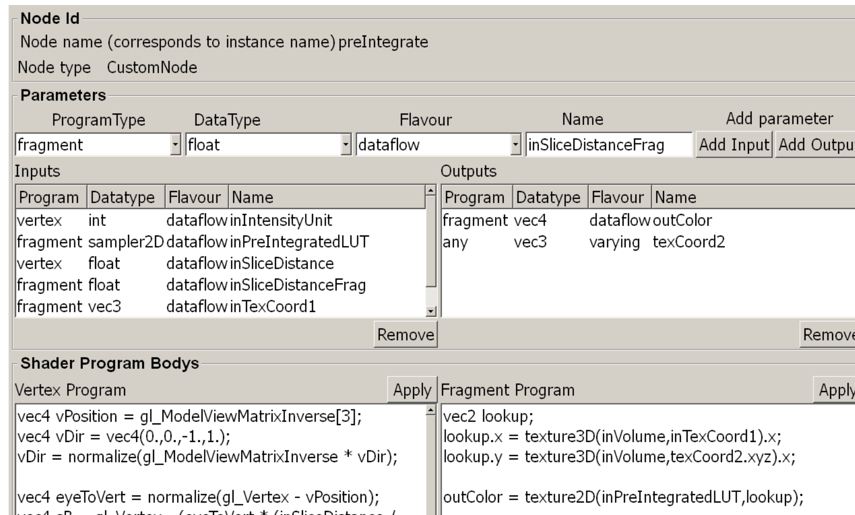


Abbildung 48: Festlegung der Knoteneigenschaften im Panel von `SoSULCustomNode`.

immer dargestellt werden. Hat das Modul Informationen über den internen Graph einer GVR-Instanz (Id der genutzten Programmbeschreibung), so können zusätzlich alle Knoten der internen Komponentenliste repräsentiert werden, die nicht zum Gerüst gehören. Für Anknüpfungen an Standardknoten ist also keine zusätzliche Information der konkreten GVR-Instanz notwendig.

Module der Klasse `SoGVRShaderNode` fügen beim Traversieren einer Komponentenliste nur Verbindungen und keine Knotenbeschreibungen hinzu. Ihr Einsatz ist daher nur im Zusammenhang mit dem GVR sinnvoll. Die Verbindungsbeschreibungen werden mit dem Flag zum Überschreiben existierender Verbindungen versehen.

Uniforme Parameterknoten des internen Shadergraphen des GVR können auf dieselbe Weise dargestellt werden. Dazu wird die Klasse `SoGVRShaderUniform` eingesetzt.

Abbildung 47 zeigt ein Klassendiagramm der wichtigsten Elemente des Projekts SoSUL.

Zur Erstellung neuer uniformer Parameterknoten wird das Projekt SoShader eingesetzt. Dieses Framework für monolithische Shaderprogramme ist Bestandteil von MeVisLab. Entsprechende Module für verschiedene uniforme Parameter stehen zur Verfügung. Dies schließt Module für Texturen mit ein, welche MeVisLab-Bilddaten auf der Grafikkarte verfügbar machen. Die Module des SoShader-Frameworks repräsentieren die uniformen Parameterknoten des Shadergraphen. Sie werden bei der Verbindungsdefinition und Traversierung des Szenengraphs berücksichtigt.

6.3.2.2 Verarbeitung

Ein Inventor-Szenengraph zur Definition eines Shadergraphen besteht aus von `SoSULNodeBase` und `SoSULParameterBase` abgeleiteten Modulen, sowie Modulen für uniforme Shaderparameter des Projekts SoShader. Die Wurzel des Inventor-Szenengraphen bildet ein Modul der Klasse `SoSULProgram`. An dieses Modul werden die Module angeschlossen, welche die Wurzeln des Shadergraphen repräsentieren. Wird der Inventor-Szenengraph bei seiner Darstellung traversiert, so erstellt `SoSULProgram` eine Komponentenliste und fügt der Liste die Namen seiner direkten Nachfolgermodule als Wurzeln hinzu. Danach wird die Liste durch einen Zeiger im Inventorstate bekannt gemacht. Alle Nachfolgermodule können bei der Traversierung auf diese Liste zugreifen. Sie fügen ihre Knotenbeschreibung und Verbindungsbeschreibungen der Komponentenliste hinzu. Jedes Modul beinhaltet Verbindungsdefinitionen zu seinen Nachfolgern, wodurch alle möglichen Verbindungen erzeugt werden können. Kommt die Tiefensuche der Traversierung wieder beim `SoSULProgram`-Knoten an, so ist die Komponentenliste gefüllt mit allen Bestandteilen des Shadergraphen. Sie wird ausgewertet, um den konkreten Shadergraph zu erstellen.

Die Inventormodule zur Beschreibung des Shadergraphen sind Bestandteil des Szenengraphen im MeVisLab-Netzwerk, der bei jedem dargestellten Einzelbild erneut ausgewertet wird. Er wird erneut traversiert, wenn sich sein Aufbau ändert. Dies schließt Änderungen an den Codeteilen für den Shadergraph mit ein. Jegliche für den Shadergraphen relevanten Änderungen werden erkannt und berücksichtigt. Aus diesem Grund ist *Rapid Prototyping* möglich.

Zur Beschleunigung sind die erstellten Module der Projekte SoSUL und SoGVR so gestaltet, dass sie nur dann ihre Arbeit ausführen, wenn dies wirklich nötig ist. Sobald der repräsentierte Shadergraph einmal erzeugt wurde, wird er nur nach einer Veränderung der Module durch den Nutzer erneut aufgebaut. Sind alle Module der Shadergraph-Beschreibung unverändert, funktionieren die Inventormodule wie einfache Gruppenmodule im Szenengraph (`SoGroup`). Sie leiten die Traversierung für den Fall weiter, das sich ein vom Shadergraph unabhängiger Szenengraphteil verändert hat, führen aber selbst keine Operationen aus.

6.3.2.3 Visuelle Datenflussprogrammierung mit der GUI

Zur Beschreibung von Shadergraphen in MeVisLab-Netzwerken werden Inventormodule eingesetzt, die in einem Szenengraph organisiert werden. Die Inventormodule werden eingesetzt, um den Datenfluss in einem Shadergraph festzulegen. Die Richtung des repräsentierten Datenflusses ist am Inventor-Szenengraph ablesbar. Er findet von unten nach oben statt. Mit Hilfe der erstellten Inventormodule wird visuelle Datenflussprogrammierung realisiert, auch wenn die Verarbeitung des Szenengraphen auf andere

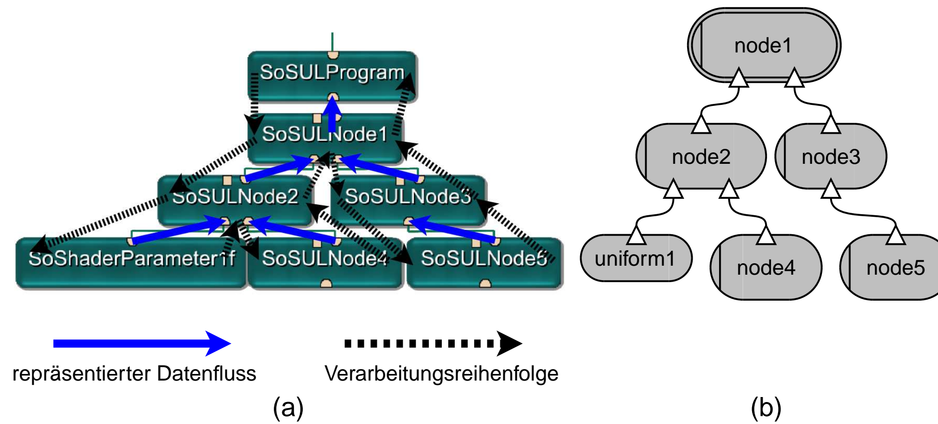


Abbildung 49: (a) Verarbeitungsreihenfolge und repräsentierter Datenfluss in einem Szenengraph mit SoSUL-Knoten. **(b)** Entsprechender Shadergraph.

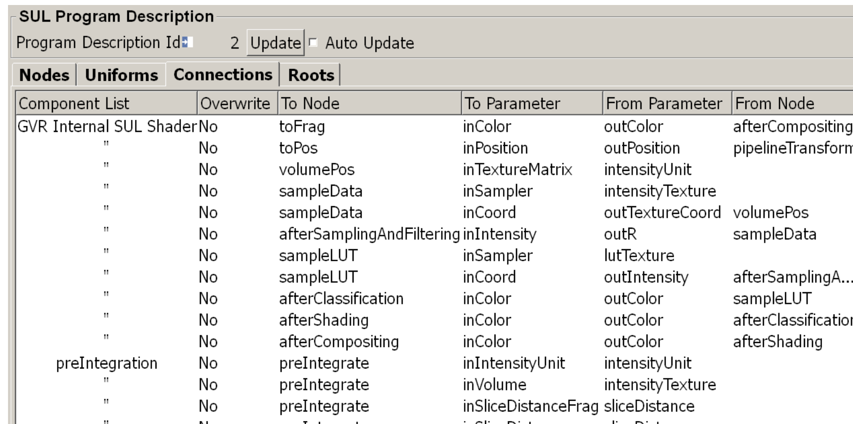
Weise geschieht (Abbildung 49). Datenflussprogrammierung kommt hier als Koordinationskonzept zum Einsatz, und die Inventormodule stellen die Werkzeuge dafür dar (siehe Absschnitt 3.3.2).

6.3.2.4 Hilfsmodule

Es existieren weitere Hilfsmodule zur übersichtlichen Arbeit mit der GUI. `SULModuleManager` erlaubt die Verwaltung einer Knotenmusterdatenbank. `SULProgramDescriptionInspector` ermöglicht die Ansicht der Komponentenlisten einer Programmbeschreibung. Diese werden in Textform als Listen dargestellt(Abbildung 50). Eine grafische Repräsentation wurde aufgrund des erwarteten Aufwands nicht erstellt, würde aber ein gutes Hilfsmittel darstellen. Die textuelle Repräsentation erlaubt jedoch ebenfalls einen guten Einblick in die Struktur eines bestehenden Shadergraphen bzw. seiner Definition in einer Programmeschreibung. Das Modul `SoSULLoopOutput` kann über eine Base-Verbindung an ein SoSUL-Modul angeschlossen werden. Es stellt dann dessen Ausgabeparameter für Schleifen dar, und enspricht einem Pseudoknoten (siehe Abschnitt 5.2.3). Damit kann eine übersichtlichere Darstellung entsprechend Abbildung 38b erreicht werden. Ein Beispiel zeigt Abbildung 55 auf Seite 110.

6.3.2.5 Anpassung des Shadergraphen des GVR

Neben den Modulen des Projekts SoSUL werden zwei weitere Module des Projekts SoGVR im Zusammenhang mit dem Shadergraph des GVR verwendet. Im Normalzustand verwendet der GVR weiterhin die Standard-Shaderverwaltung. Shadergraphen werden nicht verwendet. Der GVR ver-



Component List	Overwrite	To Node	To Parameter	From Parameter	From Node
GVR Internal SUL Shader	No	toFrag	inColor	outColor	afterCompositing
"	No	toPos	inPosition	outPosition	pipelineTransform
"	No	volumePos	inTextureMatrix	intensityUnit	
"	No	sampleData	inSampler	intensityTexture	
"	No	sampleData	inCoord	outTextureCoord	volumePos
"	No	afterSamplingAndFiltering	inIntensity	outR	sampleData
"	No	sampleLUT	inSampler	lutTexture	
"	No	sampleLUT	inCoord	outIntensity	afterSamplingA...
"	No	afterClassification	inColor	outColor	sampleLUT
"	No	afterShading	inColor	outColor	afterClassification
"	No	afterCompositing	inColor	outColor	afterShading
preIntegration	No	preIntegrate	inIntensityUnit	intensityUnit	
"	No	preIntegrate	inVolume	intensityTexture	
"	No	preIntegrate	inSliceDistanceFrag	sliceDistance	

Abbildung 50: Verbindungsdefinitionen in der Ansicht eines `SULProgramDescriptionInspector`.

hält sich dann unverändert und kann von den Anwendern bei MEVIS wie gewohnt eingesetzt werden. Zur Aktivierung des Modus mit Verwendung von Shadergraphen muss das Modul `SoGVRShaderExtension` dem Inventor-Szenengraphen hinzugefügt werden. Das Modul ist abgeleitet von der Basisklasse `SoGVRExtension`. Diese *GVR-Erweiterungen* werden vom GVR eingesammelt und können seine Einstellungen verändern. Sie müssen in der Traversierung vor dem Volumenrenderer erscheinen. `SoGVRShaderExtension` exportiert die Id der intern verwendeten Programmbeschreibung. Diese kann in den Modulen `SoGVRShaderNode` und `SULProgramDescriptionInspector` als Informationsquelle weiterverwendet werden. Der interne Shadergraph des GVRs kann auf diese Weise inspiziert werden.

Mit Hilfe der Modulklasse `SoGVRShaderComponents` lassen sich externe Komponentenlisten definieren und dem Shadergraph des GVRs hinzufügen. Für jedes Modul dieses Typs wird eine Komponentenliste erstellt. Das Modul selbst fungiert als Wurzel eines Subgraphs des Szenengraphen. Dieser Subgraph füllt die Komponentenliste des Moduls. Anknüpfungspunkte an den internen Shader werden mit Hilfe von Modulen der Klassen `SoGVRShaderNode` und `SoGVRShaderUniform` definiert. Hinzukommende Knoten lassen sich durch Module des Projekts SOSUL hinzufügen. In `SoGVRShaderComponentes` selbst lassen sich der Komponentenliste Befehle zum Erstellen oder Entfernen von Wurzeldefinitionen hinzufügen. Damit lassen sich alle Einflussmöglichkeiten externer Komponentenlisten auf den GVR-Shadergraph durch die in MEVISLAB erstellten Module ausüben. Externe Komponentenlisten werden grafisch erstellt, indem Teile von Shadergraphen spezifiziert werden.

7 Ergebnisse

Das erstellte Shaderframework für Volumenrendering wurde für verschiedene Erweiterungen am GVR eingesetzt. Es wurde ebenfalls ein einfacher Raycaster realisiert. Ausgewählte Beispielanwendungen werden im Folgenden vorgestellt. Im zweiten Teil dieses Kapitels wird die Performance des Frameworks bewertet. Dazu wurde die Geschwindigkeit der gesamten Darstellung gemessen sowie die Instruktionsanzahl kompilierter Shaderprogramme zwischen generiertem und manuell optimiertem Code verglichen.

7.1 Exemplarische Anwendungen

Der Volumenrenderer GVR wurde mit Hilfe der erstellten Frameworks um neue Funktionalität erweitert. Die Grundlage dafür bildet die Umsetzung grundlegender Darstellungseffekte des GVRs im internen Shadergraphen. Der existierende Funktionsumfang des GVR ist sehr groß, und dementsprechend auch die Menge der umzusetzenden Shaderfunktionalität. Aus Zeitgründen wurde nicht die gesamte bestehende Funktionalität des GVR im internen Shadergraph umgesetzt. Die Ergänzung der fehlenden Funktionalität sollte jedoch kein prinzipielles Problem darstellen. Folgende Fähigkeiten des GVRs wurden umgesetzt:

- Darstellung von Intensitätsvolumen
- Lookuptables (1D,2D,2D mit Tagvolumen)
- Blinn-Phong-Beleuchtung (bis zu drei Lichtquellen)
- Enhancement (Boundary und Silhouette)
- Tone Shading

Ausgehend von dieser internen Funktionalität wurden über die im letzten Kapitel beschriebenen Schnittstellen neue Funktionalität hinzugefügt. Diese exeeplarischen Erweiterungen werden im Folgenden mit den zugrundeliegenden Ideen kurz vorgestellt.

7.1.1 Preintegriertes Volumenrendering

Volumenrendering wird in der realen Anwendung oft mit großen Volumen und vielen Darstellungseffekten eingesetzt, sodass die Optimierung der Darstellungsgeschwindigkeit wichtig wird. Preintegratives Volumenrendering bietet die Möglichkeit, die Darstellungsqualität mit geringen Laufzeitkosten zu verbessern, und wurde beispielhaft in den GVR integriert.

Der Inventor-Szenengraph des in Abbildung 51a gezeigten Netzwerks verarbeitet ausgehend vom Modul `SoExaminerViewer` die Shaderanpassungen und danach das Modul `SoGVRVolRen`, welches die Anpassungen an den Volumenrenderer weitergibt und die Bilderzeugung steuert. Die Anpassungen bestehen aus einem eigenen Programmknoten (Modul `SoSULCustomNode`), der die Schritte Sampling und Klassifikation des internen Shadergraphen ersetzt. Er wird daher an den Gerüstknoten `afterClassification` angeschlossen. Der im Beispiel eingesetzte Shadercode wurde aus [HKRs+06, Seite 98-100] entnommen. Der Knoten erhält Informationen zum Intensitätsvolumen, die Textur mit den preintegrierten Werten der Lookuptable sowie den Abstand zwischen den einzelnen Samples (*Slicedistance*) als Eingabe. Der Abstand wird für den Zugriff auf die preintegrierte Lookuptextur benötigt.

Es wurden zwei zusätzliche MEVISLAB-Module erstellt. Preintegriertes Volumenrendering benötigt Blendingeinstellungen im Framebuffer, die vom GVR nicht unterstützt wurden. In diesem Fall war eine Anpassung des GVR-Quellcodes notwendig, da andere Teile des Volumenrenderers als der Shader betroffen waren (siehe Abschnitt 5.1.1.1). Das Modul `SoGVRFramebufferSettings` erlaubt das Setzen jeder möglichen OpenGL-Blendingeinstellung im GVR durch den Anwender, und macht Änderungen am GVR-Quellcode an dieser Stelle in Zukunft überflüssig. Zur Erzeugung der Textur mit den preintegrierten Werten wurde das Bildverarbeitungsmodul `PreIntegrateLUT` erstellt. Es basiert auf dem Programmcode aus [HKRs+06, Seite 97].

Problematisch bei der erstellten Lösung ist der Wert der Slicedistance. Er berechnet sich aus dem vom Nutzer eingestellten Wert der Samplingrate, wird aber danach durch Effekte wie Anisotropiekorrektur (Darstellungsverbesserung für Volumen, deren Voxel nicht würfelförmig sind) angepasst. Der entgültige Wert ist im GVR für die Shaderklassen nicht zugänglich, so dass er für den Shader und die Erstellung der preintegrierten Lookuptextur selbst berechnet werden muss. Um die Berechnung einfach zu halten, wurden Korrekturen der Slicedistance deaktiviert (Modul `SoGVRDebug`). Für den flexibleren Einsatz des preintegrierten Volumenrenderings sollte der GVR so angepasst werden, dass die Slicedistance als Information zur Verfügung steht.

Abbildung 51 zeigt die Anwendung für eine Isooberflächen-Darstellung. Die Transferfunktion (Abbildung 51b) enthält dabei nur für einen kleinen Ausschnitt des Intensitätsbereichs eine Opazität größer null. Die Darstellung mit durchschnittlicher Samplingrate und postinterpolativer Klassifikation zeigt deutlich sichtbare Artefakte (51c), wegen der hohen Frequenzen in der LUT. Preintegriertes Volumenrendering zeigt bei unveränderter Samplingrate keine Artefakte (51d). Preintegriertes Volumenrendering mit dem GVR wurde von Mitarbeitern von MEVIS RESEARCH als vielversprechend eingeschätzt und mit realen Daten getestet (Abbildung 52).

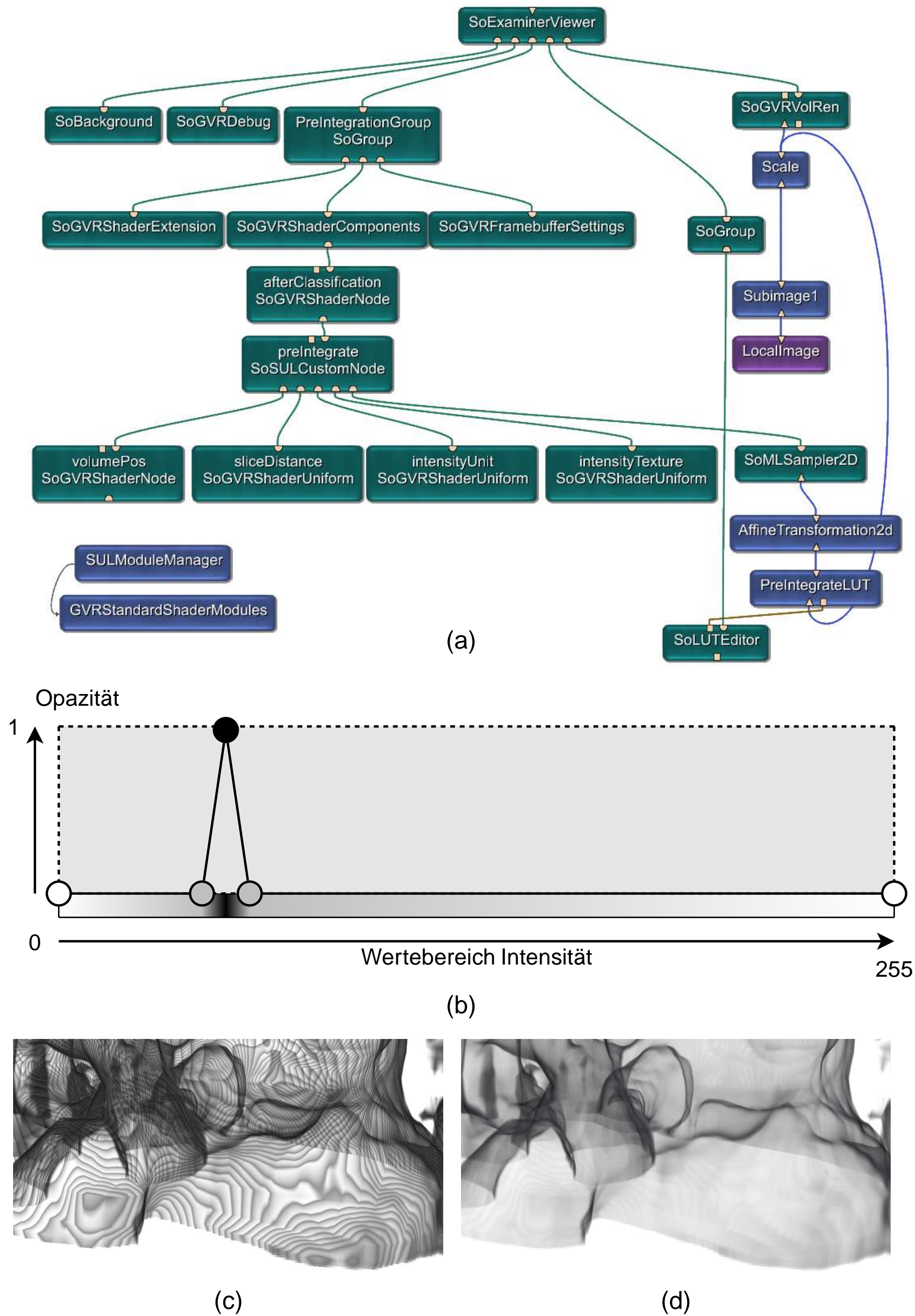


Abbildung 51: (a) MeVisLab-Netzwerk zur Erweiterung des GVR mit preintegrierter Klassifizierung. (b) Verwendete Lookup-Tabelle für Isooberflächen-Rendering. (c) Ergebnis mit postinterpolativer Klassifizierung. (d) Ergebnis mit preintegrativer Klassifizierung.

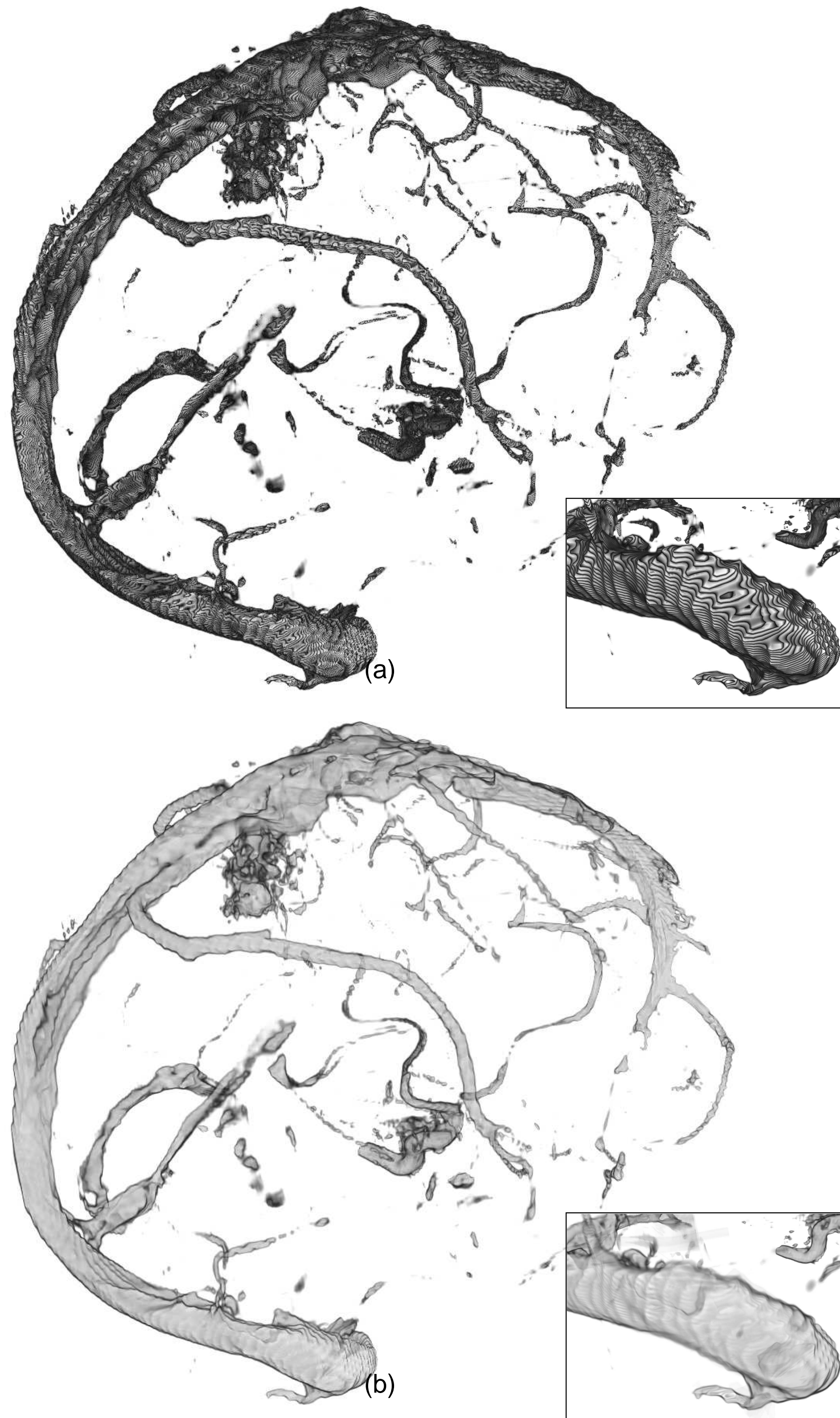


Abbildung 52: Isooberflächen-Rendering für Gehirngefäße. (a) Postinterpolative Klassifizierung. (b) Preintegrative Klassifizierung.

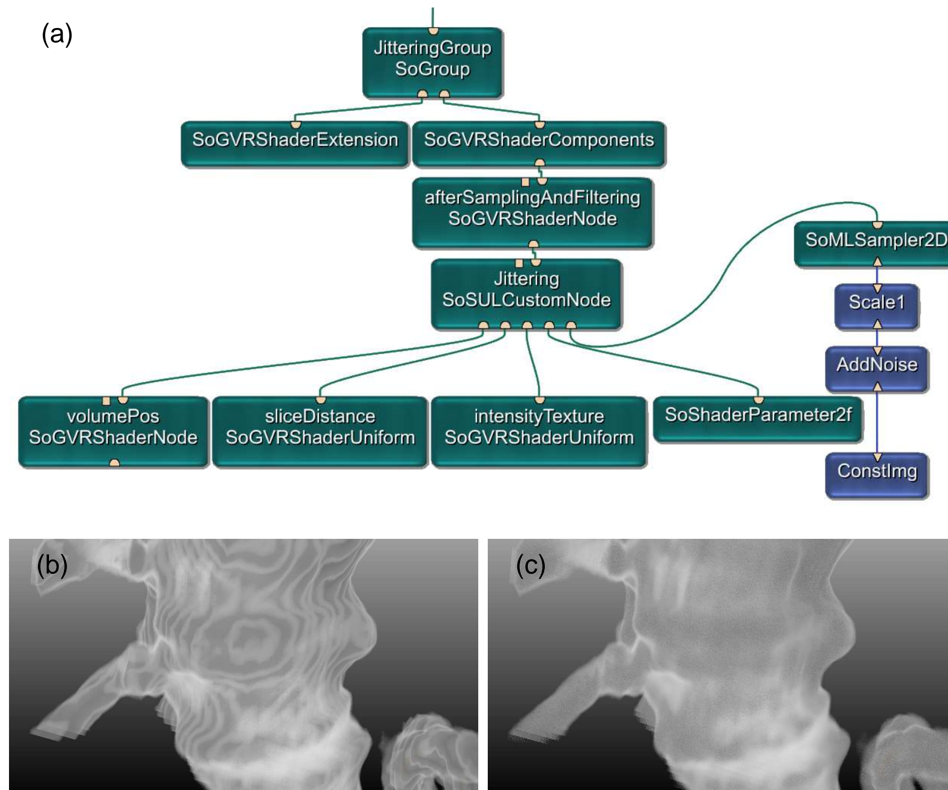


Abbildung 53: (a) Ausschnitt aus MEVISLAB-Netzwerk zur Erweiterung des GVR mit stochastischem Jittering. (b) Ergebnis ohne Jittering. (c) Ergebnis mit Jittering.

7.1.2 Jittering

Eine zu geringe Samplingrate führt auch bei Transferfunktionen mit niedrigen Frequenzen zu Artefakten (Abbildung 53b). Die Samples liegen für einen kontinuierlichen Bildeindruck zu weit auseinander. Artefakte dieser Art können ohne Erhöhung der Samplingrate durch *stochastisches Jittering* vermindert werden (Abbildung 53c). Dabei werden die Samplingpositionen entlang des Sichtstrahls um einen zufälligen Wert in der Größenordnung des Abstands benachbarter Samples verschoben. Das Volumen wird gleichmäßiger abgetastet.

Das Netzwerk zur Umsetzung von Jittering ersetzt den Samplingschritt des internen Shadergraphen. Für die Umsetzung musste nur ein einziger neuer Knoten geschrieben werden. Dies zeigt, ebenso wie das vorherige Beispiel, die gute Modularisierung durch den Shadergraph. Die Zufallswerte zur Verschiebung der Samplingposition werden durch eine Textur mit *Rauschen* bestimmt. Abbildung 53a zeigt die im Vergleich zum Netz-

werk in Abschnitt 7.1.1 veränderten Teile des Beispielnetzwerks. Weitere Informationen zu stochastischem Jittering, sowie der in der Umsetzung verwendete Programmcode, kann in [HKRs+06, Seite 220-223] gefunden werden.

7.1.3 Lit Sphereshading

Die Erstellung von Transferfunktionen für visuell ansprechende Darstellungsergebnisse wird oft als schwierig empfungen (siehe Abschnitt 4.2.2.3). *Style Transferfunctions* beschreiben eine Möglichkeit, Transferfunktionen und Beleuchtungseffekte visuell über mehrere Spheremaps festzulegen [BG07]. Beleuchtungseffekte werden bereits auf die Spheremap angewendet, und während des Volumenrenderings nicht mehr auf das Volumen ausgewertet (*Lit Sphereshading* [SMGG01]). Eine Spheremap wird mit dem Gradienten des Volumens im Kamerakoordinaten indexiert, um den Farbwert des Voxels zu bestimmen (Abbildung 54d).

Lit Sphereshading wurde exemplarisch in den GVR integriert, um detailreiche Änderungen innerhalb des Shadergraphen zu demonstrieren. Abbildung 54 zeigt den Teil des MeVisLab-Netzwerks mit Shaderanpassungen und Ergebnisbilder. Das Beispiel basiert auf einem Demonstrationsnetzwerk des GVRs. Eine Schulterfraktur wird dabei mit einer 2D-Lookuptable dargestellt. Abhängig von den über ein Tagvolumen bestimmten Segmenten des Volumens kommt ein anderer Teil der LUT zum Einsatz (Abbildung 54b). Danach wird für alle Tags ein Phong-Beleuchtungsmodell ausgewertet. Dem Shadergraph werden mit Hilfe von `SoMLSampler2D` mehrere Spheremaps hinzugefügt. Das Modul `LitSphereshading` fügt einen Knoten hinzu, der für eine Teilmenge der Volumensegmente den so bestimmten Farbwert ersetzt. Stattdessen wird Lit Sphereshading mit jeweils einer der Spheremaps durchgeführt (Abbildung 54c). Die restlichen Teile des Volumens bleiben unverändert.

Bei den Beispielen zu Jittering und preintegriertem Volumenrendering wurde das Sampling bzw. Sampling und Klassifikation, und damit der Anfang der Pipeline ersetzt. Im Shadergraph entspricht dies dem Austausch eines kompletten Subgraphen. In Fall des Beispiels zu Lit Sphereshading wird eine neue Berechnung zwischen die Pipelineschritte Shading und Compositing eingefügt. Auch Knoten, die nicht zum Gerüst gehören, werden verwendet (Gradientenknoten). Dies zeigt die Flexibilität bei den Veränderungen am Shadergraph.

7.1.4 Raycasting

Zur Demonstration von Schleifen in einem Shadergraphen wurde ein einfacher Raycaster in Form eines MeVisLab-Netzwerkes erstellt. Alle Bestandteile des Raycasters sind Module, es wurde kein zusätzlicher C++-

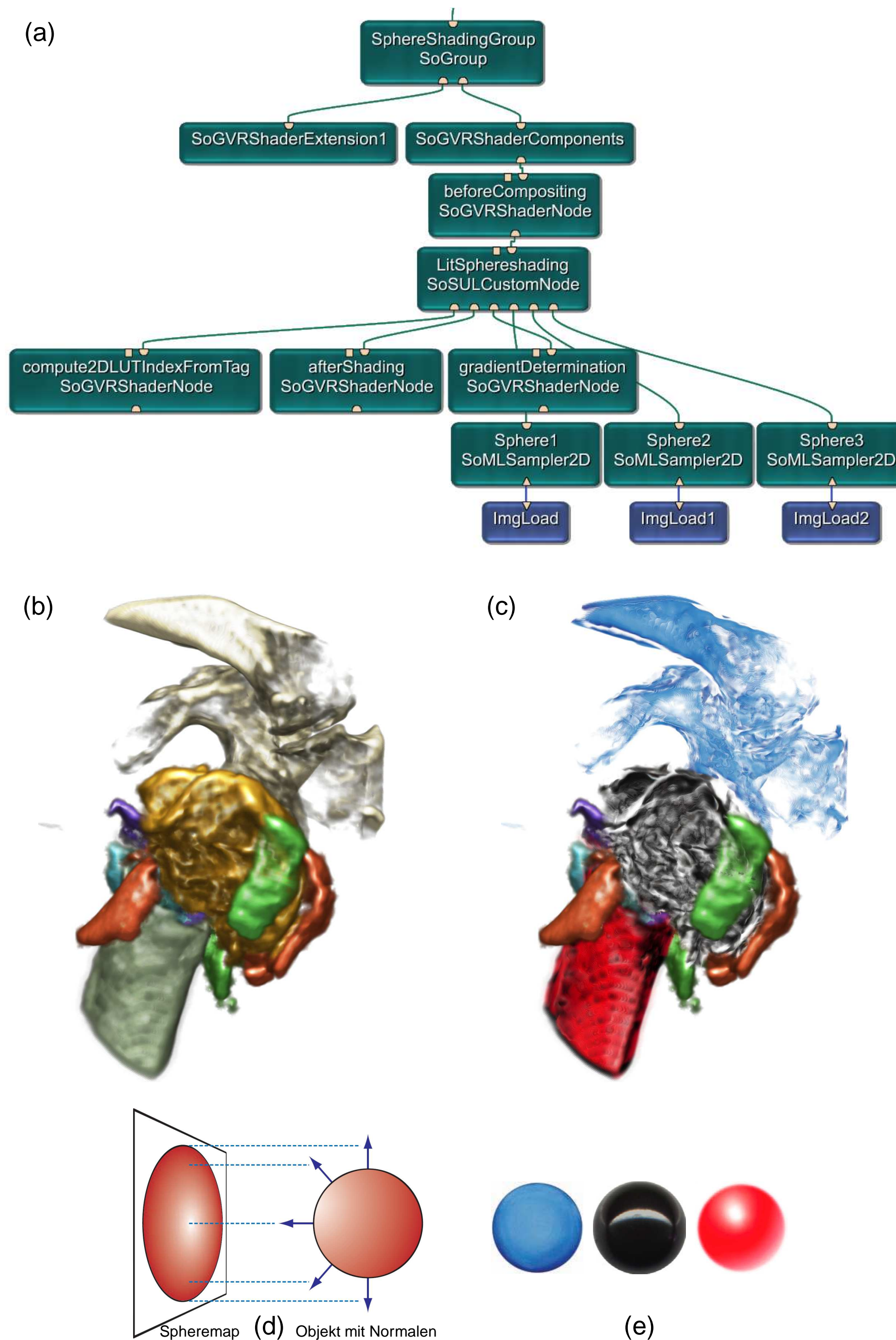


Abbildung 54: (a) Ausschnitt aus MEVISLAB-Netzwerk zur Erweiterung des GVR mit Lit Sphershading. (b) Darstellung einer Schulterfraktur mit 2D-Lookuptable. (c) Darstellung mit Lit Sphereshading für Teile des Volumens. (d) Grundidee des Lit Sphereshading. (e) Verwendete Spheremaps. Bild (d) nach/Bildquelle (e): [BG07].

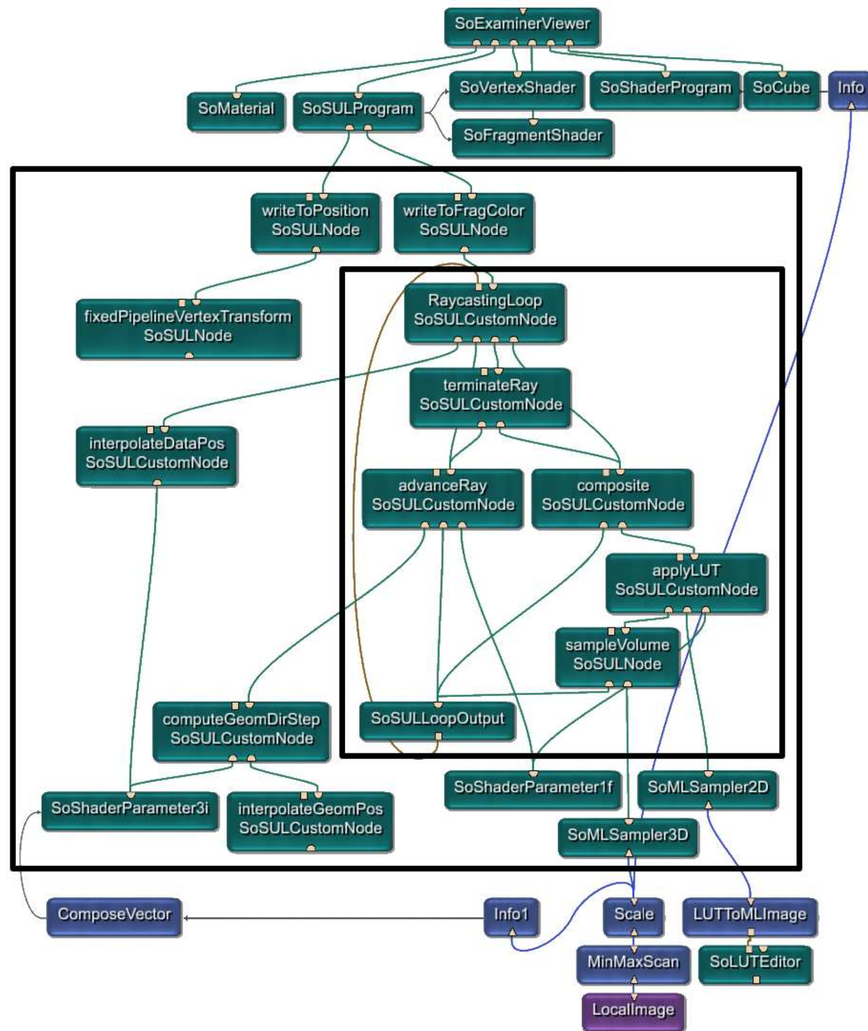


Abbildung 55: MEVISLAB-Netzwerk eines einfachen Raycasters. Das äußere Rechteck kennzeichnet Module zur Beschreibung des gesamten Shadergraphs, das innere Rechteck die Module der Raycasting-Loop.

Code erstellt. Der Raycaster basiert auf dem existierenden Netzwerk eines *GLSL Volume Raytracers*, welches das SoShader-Framework und manuell erstellten Code einsetzt [Rit07]. Es wurde mit Hilfe der SoSUL-Module um die Beschreibung der Shader durch einen Shadergraph erweitert. Die aus dem Graph entstehenden Programme werden mit Hilfe der Module `SoVertexShader`, `SoFragmentShader` und `SoShaderProgram` des Projekts SoShader kompiliert und aktiviert. Die Module `SoMLSampler2D` und `SoMLSampler3D` übertragen das Volumen und die Lookuptabelle als Texturen auf die Grafikkarte. Als Proxygeometrie wird ein Würfel in der Größe des Volumens gezeichnet. Aus der Position eines Vertex kann so einfach die entsprechende Texturkoordinate berechnet werden.

Das Netzwerk des Raycasters zeigt Abbildung 55. Module, welche den Shadergraphen beschreiben, sind markiert. Die Raycasting-Loop ist nochmals gesondert gekennzeichnet. Das Modul `RaycastingLoop` repräsentiert einen Schleifen-Programmknoten, und kennzeichnet zusammen mit dem Modul `SoSULLoopOutput` den Schleifenrumpf. Dazwischen liegende Module gehören zum Schleifenrumpf, und stehen für die Schritte innerhalb der Raycasting-Loop. Neben den Schritten der Pipeline (`sampleVolume`, `applyLUT`, `composite`) sind dies raycasting-spezifische Module zur Auswahl der nächsten Samplingposition (`advanceRay`) und zum Abbruch der Schleife (`terminateRay`). Die Position auf dem Strahl wird um eine feste Schrittweite nach vorne verschoben, und abgebrochen wenn das Volumen verlassen wird. Beschleunigungsverfahren wie Empty Space Skipping oder Early Termination können durch Austausch oder Erweiterung der raycasting-spezifischen Module in das Netzwerk integrierbar. Die übrigen Module beschreiben Teile des Shadergraphen außerhalb der Schleife, wie die Interpolation von Koordinaten zwischen Vertex- und Fragmentprogramm oder die Vorberechnung von Sichtstrahl und Schrittweite. Der Raycasting-Shader konnte somit modular beschrieben werden.

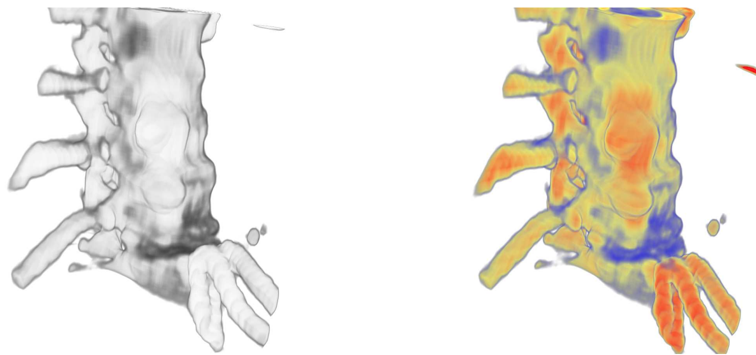


Abbildung 56: Einfaches Raycasting mit verschiedenen Lookuptables.

Fall	Shadereffekte	Volumen	Knoten*	Anmerkungen
1	1D-LUT, 1 Lichtquelle	64^3	8	
2	2D-LUT, Tagvolumen, 3 Lichtquellen, Tone Shading, Boundary & Silhouette Enhancement	$132 \times 94 \times 166$	17	
3	1D-LUT, Raycasting, keine Beleuchtung	64^3	16	Raycaster aus Abschnitt 7.1.4; unabhängig vom GVR
4	1D-LUT, preintegrierte Klassifikation, keine Beleuchtung	**a:** 64^3 **b:** $256^2 \times 132$	5	Anpassung des GVR aus Abschnitt 7.1.1

* Knotenanzahl des Shadergraphs ohne Grenzknoten.

Tabelle 4: Testfälle für den Leistungsvergleich.

Die Lookuptable kann über das Modul `SoLUTEditor` interaktiv eingestellt werden. Änderungen an Volumen oder Shaderbeschreibung werden sofort auf der Grafikkarte aktiviert. Abbildung 56 zeigt beispielhafte Darstellungen des Raycasters mit unterschiedlichen Lookuptables.

7.2 Leistungsvergleich

Das Geschwindigkeitsverhalten des erstellten Shaderframeworks wurde durch einen Vergleich mit der Standard-Shaderverwaltung des GVR untersucht. Shadergraphen verlassen sich bei der Codeerzeugung auf die Optimierungsfähigkeiten des Compilers, um die große Anzahl zusätzlicher Zuweisungen zu entfernen. Um den Erfolg von Compilern für GLSL bei dieser Aufgabe einzuschätzen, wurden die kompilierten Shader des Shadergraph-Frameworks mit „per Hand“ erstellten Shadern der Standard-Shaderverwaltung verglichen. Die Geschwindigkeit der gesamten Bilderzeugung beim Einsatz beider Verwaltungsmögichkeiten wurde ebenfalls verglichen. Es werden im Folgenden Ergebnisse zu mehreren Testfällen dargestellt (Tabelle 4).

7.2.1 Optimierung durch die Compiler

Der Erfolg von GLSL-Compilern bei der Optimierung wurde untersucht, indem die Anzahl der Instruktionen kompilierter Shader zwischen manuell (man) und vom Framework (sul) erstelltem Code verglichen wurde. Da-

Fall	Shader	Instruktionen							
		GF 6800GT		GF 7800GT		Radeon R500		Radeon R600	
		#Inst.	r-Regs	#Inst.	r-Regs	ALU	TEX	ALU	TEX
1	man	17	3	17	3	13	3	11	3
	sul	19	2	19	2	14	3	12	3
2	man	51	5	51	5	32	4	25	4
	sul	54	4	54	4	34	4	28	4
3*	man	-	-	-	-	25	2	29	2
	sul	-	-	-	-	27	2	32	3
4	man	4	1	4	1	4	3	4	3
	sul	4	1	4	1	4	3	4	3

NVIDIA GeForce (GF): Anzahl Instruktionen (#Inst.) und temporärer Register (r-Regs).
ATI Radeon: Anzahl der ALU-Instruktionen (ALU) und Texturzugriffe (TEX).
* NVShaderPerf konnte den Shader von **Testfall 3** nicht erfolgreich verarbeiten.

Tabelle 5: Instruktionsanzahl der kompilierten Shader der Testfälle. Verglichen werden manuell erstellte Shader (man) mit auf Shadergraph-Basis erstellten Shadern (sul) für verschiedene Grafikkarten-Generationen.

zu kamen die Programme NVShaderPerf 1.8 [25] von NVIDIA und GPU ShaderAnalyzer 1.40 [26] von AMD/ATI zum Einsatz. Sie ermitteln für GLSL-Code die Eigenschaften und Simulieren das Laufzeitverhalten des kompilierten Shaders auf verschiedenen Grafikkarten-Generationen des Herstellers. Die ermittelten Informationen sind dabei für beide Programme unterschiedlich. Für NVIDIA wurden die Generationen GeForce 6800GT und GeForce7800GT mit Treiberversion ForceWare 81.94 getestet. Eine Version von NVShaderPerf mit aktuellen Treibern und Unterstützung für GLSL und die neuesten Grafikkarten stand zum Zeitpunkt dieser Arbeit nicht zur Verfügung. Bestimmt wurden die *Instruktionsanzahl (#Inst)* des kompilierten Shaders und die Anzahl der genutzten *temporären Register (r-Regs)* (zu Registern siehe [Micd]). Für ATI wurden die Generationen Radeon R500 (Radeon X1***) und Radeon R600 (Radeon HD *) mit der Treiberversion Catalyst 7.12. Bestimmt wurden die Anzahl der Instruktionen für die *Arithmetic Logic Unit (ALU)* sowie die Anzahl der Texturzugriffe.

Es wurden die Fragmentshader aller Testfälle untersucht. Tabelle 5 zeigt die Ergebnisse. Für kurze Programme bzw. kleine Shadergraphen sind die Instruktionszahlen identisch (Testfall 4). Werden die Programme umfangreicher, so verwendet der kompilierte Shader des Shadergraphen mehr Instruktionen als der manuelle erstellte Shader (Testfall 1-3) und kann einen Zwischenwert weniger über temporäre Register weitergeben. Die Differenzen bewegen sich zwischen einer (Testfall 1 ATI) und drei zusätzlichen In-

[25] http://developer.nvidia.com/object/nvshaderperf_1_8_home.html. Zugriff: 15.01.2007
[26] http://ati.amd.com/developer/gpusa/index.html. Zugriff: 15.01.2008

Computer A	
CPU:	Pentium 4 2.5GHz (Singlecore) 1.5GB RAM
GPU:	GeForce 7600GT AGP4x 256MB (Treiber: ForceWare 169.21)

Computer B	
CPU:	Pentium 4 3.4GHz (Dualcore) 2GB RAM
GPU:	GeForce 8800GTS PCIe 640MB (Treiber: ForceWare 169.21)

Tabelle 6: Geschwindigkeitstests: Verwendete Computer.

struktionen (Testfall 2 NVIDIA). Die Anzahl der Texturzugriffe bleibt in allen Shadern unverändert[27].

Die Ergebnisse zeigen, dass die Anzahl zusätzlicher Instruktionen gering ist im Vergleich zur Anzahl zusätzlicher Befehle im Programmcode. Die Compiler sind in der Lage, den Großteil der Befehle zu entfernen. Dies scheint auch innerhalb von Schleifen zu funktionieren (Testfall 3). Dennoch sind Shaderprogramme auf Basis von Shadergraphen im Vergleich zu manuell erstellten Programmen geringfügig langsamer. Die Auswirkung auf die Geschwindigkeit der gesamten Bilderzeugung zeigt Abschnitt 7.2.2. Mit zunehmender Knotenanzahl im Shadergraphen wächst die Anzahl der zusätzlichen Instruktionen bei den Testfällen nur langsam. Es ist anzunehmen, dass sich dies auch bei weiter steigenden Knotenanzahlen fortsetzt. Die Art der zusätzlich hinzukommenden Befehle ist für jeden Knoten gleich, und die Compiler sollten sie ebenso entfernen können.

7.2.2 Benchmarks

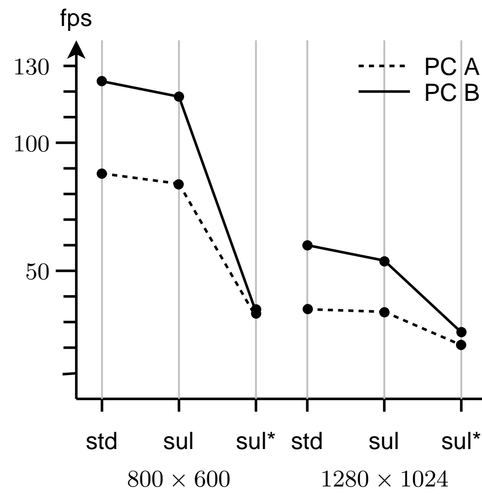
Das Laufzeitverhalten des Shaderframeworks wurde untersucht, indem für jeden Anwendungsfall die *Frames per Second (fps)* gemessen wurden. Für Testfälle mit dem GVR konnte weiterhin die Zeit gemessen werden, die die Shaderverwaltung zum Erstellen und Aktivieren der Shaderprogramme benötigt hat (*Shader-Setuptime*). Die Tests wurden auf zwei Computern unterschiedlicher Leistungsklasse durchgeführt (siehe Tabelle 6). *Computer A* hat einen Prozessor mit einem Kern und eine Grafikkarte der GEFORCE 7600-Generation, die das Shadermodel 3 unterstützt. *Computer B* hat einen Prozessor mit zwei Kernen und eine Grafikkarte der GEFORCE 8800-Generation, die das Shadermodel 4 unterstützt. Der Prozessor hat Einfluß auf die gemessene Geschwindigkeit, da er die Shaderverwaltung verarbeitet. Für jeden Testfall wurde die Performance für die Standard-Shaderverwaltung (std), für die Shadergraphen mit Caching (sul) und die Shadergraphen ohne Caching (sul*) auf beiden Geräten gemessen. Da Test-

[27] Dies ist erwartungskonform, da vom Shadergraph keine zusätzlichen Texturzugriffe eingefügt werden

fall 3 nicht den GVR verwendet, wurde hier eine manuell erstellte Shadervariante (man) mit der Verwaltung durch Shadergraphen verglichen.

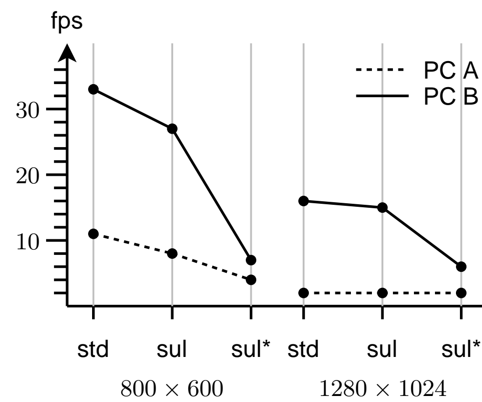
Die Ergebnisse der Geschwindigkeitstests zeigen Abbildung 57 und 58. Es ist zu erkennen, dass die Darstellung mit Hilfe der Shadergraphen und Caching im Vergleich zur Standard-Shaderverwaltung zu geringen Geschwindigkeitseinbußen führt. Interaktive oder echtzeitfähige Frameraten bleiben gewahrt. Die Shader-Setuptime mit der SUL ist von derselben Größenordnung wie die der Standardverwaltung, aber meist etwas höher. Nur im Testfall 2, indem der erstellte Shadercode umfangreich ist, sind die Setuptimes fast gleich. Die Auflösung hat keinen Einfluss auf die Setuptime[28]. Die geringfügig höhere Setuptime und Ausführungsgeschwindigkeit der Shaderprogramme durch die erhöhte Instruktionsanzahl bewirkt bei den Testfällen einen Geschwindigkeitsverlust zwischen 0% (Testfall 4a) und 28% (Testfall 2). Die Setuptime der Shadergraph-Verwaltung ohne Caching ist um zwei Größenordnungen höher als mit Caching. Für die meisten Testfälle führt dies zu großen Einbrüchen in der Geschwindigkeit (z. B. Testfall 3). Die erneute Erstellung und Verarbeitung des Shadergraphen in jedem Frame ist für interaktive Frameraten daher kaum einsatzbar. In manchen Anwendungsfällen hat jedoch selbst eine sehr hohe Setuptime keinen Einfluß. In Testfall 4b bei Auflösung 1280×1028 müssen aufgrund der hohen Auflösung von Volumen und erstelltem Bild so viele Fragmente verarbeitet werden, dass nur noch die Ausfrühungsgeschwindigkeit des Shaders der begrenzende Faktor der Geschwindigkeit ist.

[28] Dies ist erwartungskonform, da die Shaderverwaltung von der CPU verarbeitet wird.



Testfall 1

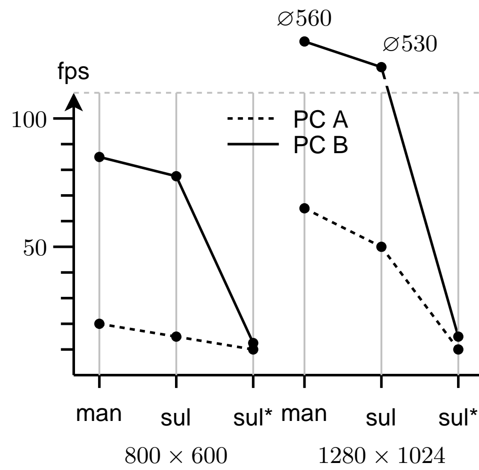
PC	Shader	Setuptime	
		800×600	1280×1024
A	std	0.244ms	0.341ms
	sul	0.569ms	0.486ms
	sul*	20.339ms	20.173ms
B	std	0.271ms	0.294ms
	sul	0.409ms	0.415ms
	sul*	23.669ms	23.191ms



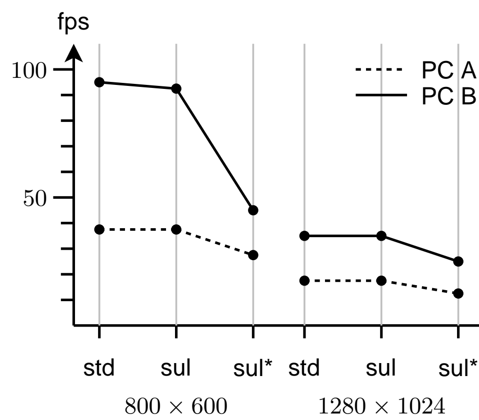
Testfall 2

PC	Shader	Setuptime	
		800×600	1280×1024
A	std	1.323ms	1.315ms
	sul	1.402ms	1.272ms
	sul*	136.278ms	136.474ms
B	std	0.846ms	0.854ms
	sul	0.984ms	0.833ms
	sul*	93.755ms	93.326ms

Abbildung 57: Frames per Second und Shader-Setuptime für **Testfall 1 und 2**.

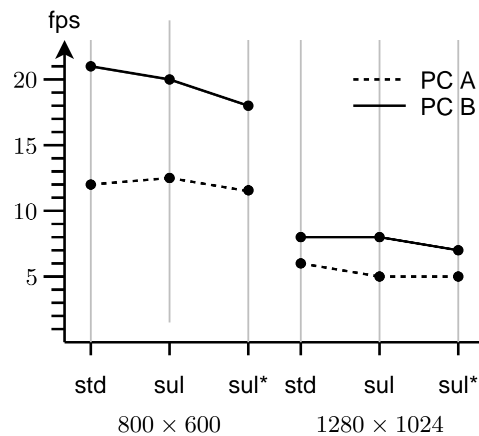


Testfall 3



Testfall 4a

PC	Shader	Setuptime	
		800×600	1280×1024
A	std	0.329ms	0.346ms
	sul	0.630ms	0.611ms
	sul*	17.704ms	17.603ms
B	std	0.248ms	0.311ms
	sul	0.385ms	0.381ms
	sul*	12.838ms	12.495ms



Testfall 4b

PC	Shader	Setuptime	
		800×600	1280×1024
A	std	0.333ms	0.345ms
	sul	0.632ms	0.560ms
	sul*	17.667ms	17.590ms
B	std	0.281ms	0.243ms
	sul	0.372ms	0.383ms
	sul*	12.180ms	12.407ms

Abbildung 58: Frames per Second für **Testfall 3**. Frames per Second und Shader-Setuptime für **Testfall 4**. Ein Vergleich der Setuptime ist für **Testfall 3** nicht anwendbar.

8 Diskussion

In dieser Arbeit wurden effektive und flexible Ansätze zur Erstellung von Shaderprogrammen für grafikkartenbasiertes Volumenrendering untersucht sowie ein Framework mit diesem Einsatzzweck konzipiert und prototypisch umgesetzt. Dazu wurden zunächst Ansätze zur *Metaprogrammierung* von Shadern vorgestellt, die Programmiersprachen für Shader um zusätzliche Möglichkeiten zur Kapselung von Code erweitern. Eine Umfrage zu Anforderungen an Volumenrendering bei MEVIS zeigte, dass dort ein Shaderframework für Volumenrendering ein Expertenwerkzeug wäre. Der Analyseteil stellte die zentralen Anforderungen an das zu erstellende Framework vor: Erstellung von Shadern des Volumenrenderers mittels *Rapid Prototyping*, modularer Aufbau der Shaderprogramme, Eignung für verschiedene Volumenrendering-Algorithmen und die Möglichkeit detaillierter Anpassungen am Volumenrendering-Shader, um für den Einsatz durch Experten geeignet zu sein.

Als Grundlage für die Entwicklungen dieser Arbeit wurden *datenflussbasierte Shadetrees* ausgewählt, da sie modulare und prototypische Programmierung erlauben und außerdem über große Übersichtlichkeit und klare Schnittstellen zwischen ihren Bestandteilen verfügen. Darauf aufbauend wurde das Konzept eines *Shadergraphs* vorgestellt, der Shadetrees um spezifische Erweiterungen für Volumenrendering ergänzt: Die Strukturierung des Graphs anhand der Volumenrendering-Pipeline und die Integration von Schleifen. Auf Basis von Shadergraphen wurde ein *modulares Shaderframework für Volumenrendering* vorgestellt, welches die Erstellung der Shader durch den Volumenrenderer und deren Anpassung durch den Anwender verwaltet. Das Konzept wurde beispielhaft im Volumenrenderer GVR und der Programmierumgebung MEVISLAB umgesetzt. Die Leistungsfähigkeit der Entwicklungen wurde an Beispielen überprüft und durch die modulare Integration neuer Darstellungseffekte in den GVR demonstriert. Es wurde festgestellt, dass das erstellte Framework zu geringfügigen Einbußen in der Darstellungsgeschwindigkeit führt, echtzeitfähige oder interaktive Frameraten aber gewahrt bleiben.

8.1 Bewertung

Das Ziel dieser Diplomarbeit, ein Framework zur Shaderentwicklung im Kontext des Volumenrendering zu konzipieren und exemplarisch umzusetzen, wurde vollständig erreicht. Die Anwendungsbeispiele haben gezeigt, dass mit dem entwickelten Framework die modulare Beschreibung von Shadern für verschiedene Volumenrendering-Ansätze möglich ist. Es wurde auch die Fähigkeit zum *Rapid Prototyping* demonstriert. Anpassungen an Shaderkomponenten können zur Laufzeit gemacht werden und wirken sich sofort im dargestellten Bild aus. Beispielhaft umgesetzte Erwei-

terungen am Volumenrenderer zeigten sich für die Praxis als vielversprechend. Die Ergebnisse dieser Arbeit demonstrieren, dass Shadergraphen insgesamt eine lohnenswerte Erweiterung für Volumenrenderer sind.

Die Vorteile des vorgestellten Konzepts liegen vor allem in umfangreichen Konfigurationsmöglichkeiten für den Anwender. Er hat komplette Kontrolle über die Shader und muss nur notwendige Teile austauschen. Die Arbeit des Entwicklers des Renderers wird verändert, aber nicht gemindert. So hat die beispielhaft erstellte Shaderverwaltung auf Shadergraph-Basis für den GVR ähnlichen Umfang wie dessen Standardverwaltung. Sie bietet zudem noch großes Potential zur Optimierung. Zur Laufzeit führt das Framework dennoch nur zu geringen Geschwindigkeitseinbußen im Vergleich zu den Standardshadern des GVR. Die Größenordnung der Verlangsamung skaliert gut mit der Anzahl der Knoten im Shadergraph. Interaktives Arbeiten wurde nicht beeinträchtigt. Mit dem Shaderframework können zudem alle Volumen und Datenmengen verarbeitet werden, die mit manuell erstelltem Code möglich sind, da diesbezüglich keine neuen Einschränkungen gemacht werden.

Der Ansatz stößt bei großen Graphen und komplexen Kontrollstrukturen an seine Grenzen. Die Übersichtlichkeit der Shadergraphen nimmt in diesem Fall schnell ab, wie das Raycasting-Beispiel in dieser Arbeit zeigte. Bedingte Anweisungen und verschachtelte Schleifen sind auf Ebene des Shadergraphen im Moment nicht möglich. Im Volumenrendering sind verschachtelte Schleifen denkbar, wenn innerhalb einer Raycasting-Loop pro Sample ein weiterer Strahl verfolgt wird (z. B. zu einer Lichtquelle). Dem Anwender sind hier also stärkere Grenzen gesetzt als bei der manuellen Erstellung von Programmcode, da nicht alle Kontrollstrukturen können modular dargestellt werden.

Weiterhin bleibt zu untersuchen, wie sich die Erstellung von Shadern mit Shadergraphen in der Praxis auf die Größe von kompilierten Shaderprogramme auswirkt. Die hohe Anzahl zusätzlicher Befehle im Shadercode wird größtenteils vom Compiler entfernt. Dennoch kann vermutet werden, dass die einfache Kombination großer Mengen von Programmcode mit Hilfe der visuellen Programmierung zu Shadern mit hoher Instruktionszahl führt. Für Grafikkarten mit Shadermodel 3.0 und niedriger könnte die maximale Instruktionsanzahl überschritten werden. In diesem Fall wäre der Einsatz existierender Ansätze zur automatischen Aufteilung des Shadercodes auf mehrere Renderdurchgänge denkbar [Hei05].

Auf Shadetrees basierende Ansätze wurden für verschiedeneste Einsatzzwecke genutzt. Es ist daher anzunehmen, dass auch das vorgestellte Framework viele weitere Einsatzmöglichkeiten hat. Shadergraphen und die umgesetzten Mechanismen zu ihrer Verwaltung sind umfangreiche Expertenwerkzeuge, die flexibel eingesetzt werden können.

8.2 Ausblick

Das vorgestellte Konzept für ein modulares Shaderframework für Volumenrendering bietet umfangreichen Raum für Optimierungen und Erweiterungen. Die Möglichkeit zur Angabe initialer Parameterwerte würde es erlauben, auch nicht verbundene Parameter sinnvoll zu nutzen, z. B. für optionale Parameter. Die Möglichkeit der Ergänzung der Header um eigenen Programmcode würde es dem Anwender erlauben, Funktionen zu definieren und Präprozessorbefehle zu nutzen. Benennungskonventionen für die Ein- und Ausgabeparameter sowie uniforme Parameter könnten den Überblick über Shadergraphen noch verbessern. Weitere Kontrollstrukturen wie bedingte Anweisungen könnten Einschränkungen der aktuellen Implementation mindern. Die Integration sehr komplexer Kontrollstrukturen wie verschachtelter Schleifen wird jedoch als nicht trivial eingeschätzt.

Insbesondere die grafische Erstellung von Shadergraphen scheint großes Potential für weitere Arbeiten zu beinhalten. Eine speziell für die Bedürfnisse von Shadergraphen optimierte GUI würde ihre visuelle Programmierung noch klarer gestalten. Die weitere Entwicklung von Shadergraphen scheint zudem auch von den Möglichkeiten und der Qualität dieser Werkzeuge abzuhängen, da ein großer Teil ihrer Flexibilität durch die visuelle Programmierung ermöglicht wird. Arbeiten zur übersichtlicheren Gestaltung großer Graphen und komplizierter Kontrollflussstrukturen könnten aus diesem Grund von großem Nutzen sein.

Die Zusammensetzung der Shader für Volumenrendering hängt von den Einstellungen des Renderers ab, weshalb die Shader oft intern im Renderer verwaltet werden. Ein weiterer Schritt zur Modularisierung der Shadererstellung könnte es sein, alle Einstellungen des Renderers extern zugänglich zu machen und auf interne Shader komplett zu verzichten. Zur Shadererstellung könnte ein dem Anwendungsfall entsprechendes Konzept zum Einsatz kommen.

In dieser Arbeit wurden GPGPU-Ansätze nicht schwerpunktmäßig behandelt. Bei Raycasting werden große Teile der Grafikpipeline kaum eingesetzt (Vertexverarbeitung, Rasterisierung). Eine effiziente Umsetzung von Raycasting mit GPGPU-Ansätzen wie CUDA scheint daher möglich. Ein Vergleich eines solchen Ansatzes mit dem Konzept dieser Arbeit im Hinblick auf Mächtigkeit und Performance könnte interessant sein.

Es ist anzunehmen, dass die Komplexität der Shaderprogramme in ähnlichem Umfang zunehmen wird wie die stetig steigende Leistungsfähigkeit der Grafikkarten. Die Entwicklungen der nächsten Jahre könnten daher zum Einzug weiterer Konzepte in Shaderprogrammiersprachen führen, z. B. zu objektorientierten Shadern. GPUs beziehen ihre enorme Leistungsfähigkeit jedoch vor allem aus ihrer Spezialisierung. Demgegenüber steht ein Trend zu immer allgemeinerer Nutzung von Grafikkarten (GPGPU). Inwieweit beide Anforderungen auf Dauer vereinbar sein werden,

wird die zukünftige Entwicklung zeigen. Diese Entwicklungen werden die zukünftigen Anforderungen an die Programmiersprachen für Shader mit prägen.

Sicher ist, dass die Programmierung von Grafikkarten auch weiterhin für Volumenrendering von großer Wichtigkeit sein wird. Prototypische Shadererweiterungen für Volumenrendering werden schon heute in der Forschung bei MEVIS RESEARCH eingesetzt. Shadergraphen haben sich in dieser Arbeit als gutes Werkzeug für diesen Einsatzzweck gezeigt und könnten hier zu einem hilfreichen Werkzeug werden.

Anhang

A Programmcode

```
1  template <int NLIGHTS> class BlinnPhong {
2    public:
3      ShTexture2D<ShColor3f> kd;
4      ShTexture2D<ShColor3f> ks;
5      ShAttrib1f spec_exp;
6      ShPoint3f light_position[NLIGHTS];
7      ShColor3f light_color[NLIGHTS];
8      template <ShVariableKind IO> struct VertFrag {
9        ShPoint<4,IO,float> pv; // position (VCS)
10       ShTexCoord<2,IO,float> u; // texture coordinate
11       ShNormal<3,IO,float> nv; // normal (VCS)
12       ShColor<3,IO,float> ec; // total irradiance
13     };
14     ShProgram vert, frag;
15     BlinnPhong (int res) : kd(res,res), ks(res,res) {
16       vert = SH_BEGIN_PROGRAM("gpu:vertex") {
17         ShInputNormal3f nm; // normal vector (MCS)
18         ShInputTexCoord2f u; // texture coordinate
19         ShInputPosition3f pm; // position (MCS)
20         VertFrag<SH_OUTPUT> vf;
21         ShOutputPosition4f pd; // position (HDCS)
22         vf.pv = modelview | pm;
23         vf.u = u;
24         vf.nv = normalize(modelview | nm );
25         pd = perspective | vf.pv;
26         for (int i=0; i<NLIGHTS; i++) {
27           ShVector3f lv =
28             normalize(light_position[i] - vf.pv(0,1,2));
29           vf.ec += light_color[i] * pos(vf.nv|lv);
30         }
31       } SH_END;
32       frag = SH_BEGIN_PROGRAM("gpu:fragment") {
33         VertFrag<SH_INPUT> vf;
34         ShOutputColor3f fc; // fragment color
35         ShVector3f vv = normalize(-vf.pv(0,1,2));
36         ShNormal3f nv = normalize(vf.nv);
37         fc = kd(vf.u) * vf.ec;
38         ShColor3f kst = ks(vf.u);
39         for (int i=0; i<NLIGHTS; i++) {
40           ShVector3f lv =
41             normalize(light_position[i] - vf.pv(0,1,2));
42           ShVector3f hv = normalize(lv + vv);
43           fc += kst * pow(pos(hv|nv),spec_exp) * light_color[i];
44         }
45       } SH_END;
46     }
47 };
```

Programmcode 1 : SH-Shader, der ein abgewandeltes Phong-Beleuchtungsmodell realisiert. Quelle: [MTP+04]

C++-Code:

```
 1 class TextureArray;
 2
 3 class HardwareTextureArray:
 4   public Shader<HardwareTextureArray ,TextureArray>{
 5   public:
 6     HardwareTextureArray(list<Image> tiles);
 7
 8     virtual vec3 texture2D(int index ,vec2 uv){
 9             return invoke<vec3>("HardwareTA_texture2D" );
10     };
11   private:
12     //data
13     sampler2DArray<uniform> texture;
14
15     DERIVED_DECL(HardwareTextureArray ,TextureArray)
16 };
17
18 CLASS_INIT(HardwareTextureArray ,"TextureArray.glsl",
19     (texture2D),(texture))
```

GLSL-Code:

```
 1 //reference and data defining an object instance
 2 #define HardwareTextureArray_SELF \
 3   OBJREF self , sampler2DArray texture
 4
 5 //function implementation
 6 vec3 HardwareTA_texture2D(HardwareTextureArray_SELF ,
 7     int index ,vec2 uv)
 8 {
 9   vec3 coord = vec3(uv,float(index));
10   return texture2DArray(texture,coord);
11 }
12
13
14
15 //data for each instance in c++
16 uniform sampler2DArray obj_0x1_texture;
17 uniform sampler2DArray obj_0x2_texture;
18
19 //dispatcher function
20 vec3 TextureArray_texture (OBJREF self,int arg1 ,vec2 arg2 ){
21   if (self == 1)
22     return HardwareTA_texture2D(self,obj_0x1_texture,
23         arg1,arg2);
24   else if (self == 2)
25     return HardwareTA_texture2D(self ,obj_0x2_texture,
26         arg1,arg2);
27   //Default return value
28   return vec3 (0.,0.,0.);
29 }
```

Programmcode 2 : Codebeispiel für objektorientiertes Design von Shadern. Nach: [Kuc07]

```
1  ShAttrib2f wood_freq;
2  ShAttrib1f wood_scale;
3  ShAttrib1f wood_noise_scale;
4  ShTexture1D<ShColor3f> wood_kd(256);
5  ShTexture1D<ShColor3f> wood_ks(256);
6
7  ShProgram wood_frag;
8  // define kernel to generate texture coords from noise
9  ShProgram wood_frag_inc = SH_BEGIN_PROGRAM("gpu:fragment") {
10   ShInputPoint3f x; // IN(0): model space position
11   ShOutputTexCoord1f u; // OUT(1): texture coordinate
12   ShPoint3f scaledX = x(0,1,2)*wood_scale*0.5;
13   ShAttrib1f noise = sturbulence(scaledX);
14   u = frac(wood_scale*(x(1,2)|x(1,2))+wood_noise_scale*noise);
15 } SH_END;
16 // make two copies of texture coords (one for ks, one for kd)
17 wood_frag = shDup(2) << wood_frag_inc;
18 // feed texture coords into texture lookups for ks and kd
19 wood_frag = (access(wood_kd) & access(wood_ks)) << wood_frag;
```

Programmcode 3 : Shaderalgebra in SH. Verbindungsoperator « und Kombinationsoperator & erlauben die Verknüpfung von Sh-Programmen. access erlaubt Texturzugriff. shDup ist ein Manipulator, der Parameter dupliziert. Die Zusammenhänge der Einzelprogramme sind in Abbildung 24 dargestellt. Nach: [MTP+04]

```
1  struct VertexContext {...} Context;
2
3  void main(void) {
4    if(vertexHandlerInvokerTable[0])
5      shaderAonInit(Context);
6    if(vertexHandlerInvokerTable[1])
7      shaderBonTransform(Context);
8    if(vertexHandlerInvokerTable[2])
9      shaderConTransform(Context);
10   if(vertexHandlerInvokerTable[3])
11     shaderBonLighting(Context);
12   if(vertexHandlerInvokerTable[4])
13     shaderConLighting(Context);
14   if(vertexHandlerInvokerTable[5])
15     shaderAonFinish(Context);
16 }
```

Programmcode 4 : Beispiel für einen Kontrollshader. Quelle: [TD07]

```
1 varying vec3 intensityTexCoord;
2 varying vec3 gradientTexCoord;
3
4 void main {
5   gl_Position = ftransform();
6   intensityTexCoord = vec3(gl_TextureMatrix[0] * gl_Vertex);
7   gradientTexCoord = vec3(gl_TextureMatrix[1] * gl_Vertex);
8 }
```

```
1 uniform sampler3D intensityTex;
2 uniform sampler3D gradientTex;
3 uniform sampler1D lutTex;
4 varying vec3 intensityTexCoord;
5 varying vec3 gradientTexCoord;
6
7 uniform vec3 lightVec;
8 uniform vec3 halfVec;
9 uniform vec4 premultipliedColor;//lightcolor*intensity*materialcolor
10 uniform float ambience;
11 uniform float specularity;
12
13 void main() {
14   //--Sampling and filtering--
15   //using hardware filtering
16   vec4 index = texture3D(intensityTex, intensityTexCoord);
17
18   //--Classification--
19   vec4 color = texture1D(lutTex, index.x);
20
21   //--Shading--
22   //precalculated gradient from texture
23   vec4 gradient;
24   gradient = texture3D(gradientTex, gradientTexCoord);
25   gradient.xyz = myGradient.xyz*2.0-1.0;
26   gradient.xyz = normalize(gradient.xyz);
27   //phong shading
28   vec4 shadedColor;
29   shadedColor.xyz = color.xyz * ambience;
30   float diffuse = max( dot( gradient.xyz, lightVec), 0.0);
31   shadedColor.xyz += diffuse * color.xyz * premultipliedColor.xyz;
32   float specular = max( dot( gradient.xyz, halfVec), 0.0);
33   specular = pow( specular, specularity );
34   shadedColor.xyz += specular;
35   shadedColor.a = color.a;
36
37   //--Compositing--
38   //using framebuffer operations
39
40   //--write fragment color--
41   gl_FragColor = shadedColor;
42 }
```

Programmcode 5 : Einfaches GLSL Vertex- und Fragmentprogramm für Volumenrendering mit einem Slicing-Algorithmus. Es orientiert sich an Shaderprogrammen des GVRs.

```
1  ...
2  varying vec3 mySampleTex_texturePosition;
3  ...
4
5  void main {
6  ...
7
8  //begin node mySampleTex
9  int mySampleTex_inTextureUnit = someIntUniform;
10 {
11 mySampleTex_texturePosition =
12   vec3(gl_TextureMatrix[mySampleTex_inTextureUnit] * gl_Vertex);
13 }
14 //end node mySampleTex
15
16 ...
17 }
```

```
1  ...
2  varying vec3 mySampleTex_texturePosition;
3  ...
4
5  void main {
6  ...
7
8  //begin node mySampleTex
9  sampler3D mySampleTex_inTexture = someSampler3D;
10 vec4 mySampleTex_outColor;
11 {
12 mySampleTex_outColor =
13   texture3D(mySampleTex_inTexture,mySampleTex_texturePosition);
14 }
15 //end node mySampleTex
16
17 ...
18 }
```

Programmcode 6 : Erstellte Codeblöcke für den Knoten `interpolate-AndSample3DTexture` aus Abbildung 33 (oben Vertexprogramm, unten Fragmentprogramm). Beginn und Ende der `main`-Methode wurden zur besseren Lesbarkeit ergänzt.

```
1  uniform int intensityUnit;
2  uniform int gradientUnit;
3  varying vec3 interpolateIntensityPos_outTextureCoord;
4  varying vec3 interpolateGradientPos_outTextureCoord;
5
6  void main() {
7
8  //start of node interpolateIntensityPos
9  int interpolateIntensityPos_inTextureMatrix=intensityUnit;
10 {
11 interpolateIntensityPos_outTextureCoord =
12   vec3(gl_TextureMatrix[interpolateIntensityPos_inTextureMatrix] *
13                              gl_Vertex);
14 }
15 //end of node interpolateIntensityPos
16
17 //start of node interpolateGradientPos
18 int interpolateGradientPos_inTextureMatrix=gradientUnit;
19 {
20 interpolateGradientPos_outTextureCoord =
21   vec3(gl_TextureMatrix[interpolateGradientPos_inTextureMatrix] *
22                        gl_Vertex);
23 }
24 //end of node interpolateGradientPos
25
26 //start of node pipelineTransform
27 vec4 pipelineTransform_outPosition;
28 {
29 pipelineTransform_outPosition = ftransform();
30 }
31 //end of node pipelineTransform
32
33 //start of node toVertexPos
34 vec4 toVertexPos_inPosition=pipelineTransform_outPosition;
35 {
36 gl_Position = toVertexPos_inPosition;
37 }
38 //end of node toVertexPos
39
40 }
```

Programmcode 7 : Vertexprogramm, welches dem Shadergraph aus Abbildung 34 entspricht.

```
 1 uniform float specularity;
 2 uniform bool renormalizeGradients;
 3 uniform sampler1D lutTexture;
 4 uniform vec3 lightVector0;
 5 uniform vec4 premultipliedColor0;
 6 uniform sampler3D intensityTexture;
 7 uniform vec3 halfVector0;
 8 uniform sampler3D gradientTexture;
 9 uniform float ambience;
10 varying vec3 interpolateIntensityPos_outTextureCoord;
11 varying vec3 interpolateGradientPos_outTextureCoord;
12
13 void main() {
14
15 //start of node sampleIntensityVolume
16 vec3 sampleIntensityVolume_inCoord=interpolateIntensityPos_outTextureCoord;
17 vec4 sampleIntensityVolume_outIntensity;
18 {
19 sampleIntensityVolume_outIntensity = texture3D(intensityTexture,
20                                               sampleIntensityVolume_inCoord).x;
21 }
22 //end of node sampleIntensityVolume
23
24 //start of node applyLUT
25 float applyLUT_inCoord=sampleIntensityVolume_outIntensity;
26 vec4 applyLUT_outColor;
27 {
28 applyLUT_outColor = texture1D(lutTexture, applyLUT_inCoord);
29 }
30 //end of node applyLUT
31
32 //start of node sampleGradientVolume
33 vec3 sampleGradientVolume_inCoord=interpolateGradientPos_outTextureCoord;
34 bool sampleGradientVolume_renormalizeFlag=renormalizeGradients;
35 vec3 sampleGradientVolume_outGradient;
36 {
37 sampleGradientVolume_outGradient = texture3D(gradientTexture,
38                                             sampleGradientVolume_inCoord).xyz;
39 sampleGradientVolume_outGradient = sampleGradientVolume_outGradient*2.-1.;
40 if (sampleGradientVolume_renormalizeFlag) {
41 sampleGradientVolume_outGradient = normalize(sampleGradientVolume_outGradient);
42 }
43 }
44 //end of node sampleGradientVolume
45
46 //start of node phongShading
47 vec4 phongShading_inColor=phongShading_outColor;
48 vec3 phongShading_inGradient=sampleGradientVolume_outGradient;
49 vec3 phongShading_inHalfVector=halfVector0;
50 vec3 phongShading_inLightVector=lightVector0;
51 vec4 phongShading_inPremultipliedLightColor=premultipliedColor0;
52 float phongShading_inSpecularity=specularity;
53 vec4 phongShading_inSurfaceColor=afterClassification_outColor;
54 float phongShading_inWeight=ambience;
55 vec4 phongShading_outColor;
56 {
57 phongShading_outColor.xyz = phongShading_inColor.xyz * phongShading_inWeight;
58 phongShading_outColor.a = phongShading_inColor.a;
59 float diffuse = max( dot( phongShading_inGradient, phongShading_inLightVector), 0.);
60 phongShading_outColor.xyz += diffuse * phongShading_inSurfaceColor.xyz *
61                              phongShading_inPremultipliedLightColor.xyz;
62 float specular = max( dot( phongShading_inGradient, phongShading_inHalfVector), 0. );
63 specular = pow( specular, phongShading_inSpecularity );
64 phongShading_outColor.xyz += specular;
65 }
66 //end of node phongShading
67
68 //start of node toFragColor
69 vec4 toFragColor_inColor=phongShading_outColor;
70 {
71 gl_FragColor = toFragColor_inColor;
72 }
73 //end of node toFragColor
74
75 }
```

Programmcode 8 : Fragmentprogramm, welches dem Shadergraph aus Abbildung 34 entspricht

B Fragebogen zur Umfrage

Für die Gespräche wurde ein Fragebogen ausgearbeitet, der als Leitfaden den Gesprächsverlauf gestützt hat. Die Fragen wurden im gemeinsamen Gespräch behandelt. Der Fragebogen wurde nicht direkt von den befragten Personen ausgefüllt. Die Antworten wurden während den Gesprächen vom Interviewleiter notiert.

Wofür wird der GVR eingesetzt?

1. In welchem Projekt bzw. Umfeld wird Volumenrendering/GVR bei Ihnen eingesetzt?
2. Können Sie mir ein Beispielnetzwerk zeigen? Wie ist der GVR im Netzwerk eingebettet?
3. Welche Daten werden dargestellt? Wie groß sind die Daten (Auflösung, Speicherbedarf)?
4. Welche Informationen sollen zugänglich gemacht werden? Was soll also visuell dargestellt werden bzw. "sichtbar" gemacht werden?
5. Wie wichtig ist Volumenrendering im Gesamtkontext? Essentiell? Oder eher "eine nette Ergänzung"?

Features des GVR

6. Welche Features des GVR werden verwendet? Sind ihnen außerdem noch weitere Features bekannt?

    - Maske
    - Tags
    - Depth Peel
    - Beleuchtung
    - Tone Shading
    - Silhouette Enhancement
    - Boundary Enhancement
    - LUT
    - Diagnosis
    - Debug
    - Slabs
    - Geschwindigkeits-Modi
    - ...

7. Gibt es Probleme mit bestehenden Features des GVR? Sind Bugs bekannt? Wie stabil läuft der GVR?
8. Gibt es Features, die nicht erwartungskonform sind? Also Features, die anders funktionieren als Sie es intuitiv vermutet hätten?

Anforderungen an den GVR

9. Konnte etwas mit dem GVR nicht realisiert werden? Wenn ja, was? Welche Features bzw. Effekte wären für Sie daher noch wünschenswert?
10. Ist Geschwindigkeit oder Qualität wichtiger? Setzen sie die unterschiedlichen Einstellungen für interaktives und statisches Rendering ein?
11. Sind Sie mit Dokumentation, Beispielnetzwerken und sonstigen verfügbaren Informationen zum GVR zufrieden? Was fehlt?

Zum Anwender des GVR-Moduls

12. Kennen Sie sich mit Computergrafik aus?
13. Kennen Sie sich mit Volumenrendering aus?
14. Ist Ihnen Shaderprogrammierung ein Begriff?
15. Gibt es in Ihrem Team jemanden, der solche Kenntnisse hat?

Workflow

16. Wie ist bei Ihnen der generelle Entwicklungsprozess mit dem GVR?
    - Möglichkeit 1: Erst überlegen, was dargestellt werden soll. Danach wird überprüft, was mit vorhandenen Features umsetzbar ist.
    - Möglichkeit 2: Darstellungsansatz wird anhand vorhandener Features ausgearbeitet.
    - Oder noch anders?
17. Würden Sie es sich zutrauen, in einem entsprechenden Framework, eigenen Shadercode zu schreiben, um die eigenen Vorstellungen der Darstellung umzusetzen? Wäre das zielführend?
18. Wäre es eine Möglichkeit, Effekte "in Auftrag" zu geben, und diese als Makromodul einzubinden?
19. Im Bezug auf die letzte Frage. Zurückblickend, hätten Sie in Ihrem Projekt (oder einem anderen) etwas anders gemacht, wenn Sie sich nicht auf vorhandene Features beschränken hätten müssen?

Sonstiges

20. Haben Sie noch allgemeines Feedback oder sonstige Anmerkungen? Fragen, die gefehlt haben?

C Inhalt der beigefügten CD

Auf der innen am Buchrücken befestigten CD-ROM befindet sich:

- Eine Videoaufnahme, welche den Umgang mit dem erstellten Framework in MEVISLAB zeigt. Die beispielhaften Umsetzungen werden ebenfalls demonstriert.
- Screenshots zu den beispielhaften Umsetzungen.
- Die Auswertung und Gesprächsmitschriften (in anonymisierter Form) zu der geführten Umfrage.
- Dieses Dokument in elektronischer Form.

D Verzeichnisse

Glossar

Benchmark: Vergleichende Analyse. Oft synonym für die Analyse der Geschwindigkeit eines Computers oder Programms verwendet.

Blackbox: Eine Einheit, für deren Betrachtung nur ihre Schnittstellen, und nicht die interne Funktionsweise von Bedeutung ist.

Frame: Einzelbild einer Bildfolge.

Framework: Ein Framework stellt den Rahmen zur Verfügung, in dem Programme erstellt werden. Dies kann Entwicklungsumgebungen, Entwurfmuster und anderes beinhalten.

Koprozessor: Prozessor, welcher neben dem Hauptprozessor (CPU) des Computers weitere Berechnungsaufgaben übernimmt.

Metaprogrammierung: Die Erstellung von Programmcode durch andere Programme oder Beschreibungen.

Pixel: Kunstwort aus *Picture Element*, welches die Elemente einer zweidimensionalen Rastergrafik beschreiben.

Rapid Prototyping: Die Überprüfung eines Konzepts durch schnelle und exemplarische Umsetzung mit Hilfe geeigneter Werkzeuge.

Rastergrafik: Rastergrafiken unterteilen einen im Allgemeinen rechteckigen Bereich in eine endliche Anzahl von Pixel genannten Bereichen, denen ein konstanter Farbwert zugeordnet wird.

Scope: Bereich innerhalb eines Progamms, der zu einem gewissen Grad unabhängig vom Rest des Programms ist.

Signatur: Die formale Schnittstelle einer Berechnungseinheit, bestehend aus Anzahl und Typ der Parameter.

String: Eine geordnete Folge von Buchstaben bzw. Symbolen.

Treiber: Programm oder Programmteil, welches die Steuerung eines an den Computer angeschlossenen Geräts übernimmt.

Tomographie: Erzeugung eines volumetrischen Abbilds eines Objekts durch die Aufnahme einer Reihe von Schnittbildern.

Voxel: Kunstwort aus *Volume Element*, welches im Zusammenhang mit Elementen von Volumendaten eingesetzt wird.

Abkürzungen

ALU (Arithmetic Logic Unit): Bestanteil eines Prozessors, der für arithmetische und logische Berechnungen zuständig ist.

APU (Application Programming Interface): Eine Schnittstelle auf Quelltextebene zur Anbindung eines Programms an ein System oder ein Bibliothek.

CPU (Central Processing Unit): Hauptprozessor eines Computers.

CT (Computertomographie): Bildgebendes Verfahren der Tomographie auf Basis von Röntgenstrahlen.

GPU (Graphics Processing Unit): Bezeichnung für den Prozessor auf modernen Grafikkarten.

GUI (Graphical User Interface): Grafische Benutzungsoberfläche zur Interaktion mit Programmen.

GVR (Giga Voxel Renderer): Volumenrenderer der Entwicklungsumgebung MEVISLAB.

MPR (Multi-Planar Reformatting): Die Darstellung von Volumendaten durch mehrere Schnittebenen.

MRT (Magnetresonanztomographie): Bildgebendes Verfahren der Tomographie auf Basis von Magnetfeldern.

RGB (Rot Grün Blau): Beschreibung von Farben durch die Menge der Anteile rot, grün und blau.

RGBA (Rot Grün Blau Alpha): RGB-Angabe, der ein zusätzlicher Alphawert zugeorndet ist. Dieser Wert wird oft als Opazität verstanden.

SUL (Shader Utility Library): Die in dieser Arbeit entwickelte Biblithek zur Erstellung von Shadern.

Abbildungsverzeichnis

1 Grafikpipeline . . . 5
2 Programmierbare Grafikpipeline . . . 8
3 Ein- und Ausgaben von Shaderprogrammen . . . 9
4 Strahldichteverlauf in einem Volumen . . . 10
5 Funktionen im Volumenrendering-Intergral . . . 11
6 Volumenrendering-Pipeline . . . 13
7 Geometrische Modelle für Volumendaten . . . 14
8 Vergleich von Klassifikationsmethoden . . . 16
9 Vorberechnung der Anwendung der Transferfunktion . . . 17
10 Qualität verschiedener Klassifikationsmethoden . . . 18
11 Proxygeometrie für View-aligned Slicing . . . 20
12 Sampling beim Raycasting . . . 21
13 Volumenrendering: Aufgaben des Fragmentprogramms . . . 23
14 Phong-Beleuchtungsmodell . . . 25
15 Vergleich zwischen Kontroll- und Datenflussberechung . . . 28
16 Grafische Notation für Datenflussgraphen . . . 30
17 Beispiel für Datenflussberechnung in der MSVP . . . 31
18 While-Schleife in Datenflussnotation . . . 32
19 Schleife in der MSVP . . . 33
20 Datenflussgraph einer Fakultätsberechnung . . . 34
21 Beispielhafte Shadetrees . . . 35
22 Abstrakter Shadetree . . . 36
23 Vertexprogramm im VISUAL SHADITOR . . . 37
24 Verknüpfung von Shadern mittels Shaderalgebra . . . 42
25 Ausrichtung von Ansätzen zur Shadererstellung . . . 43
26 Bestandteile eines MEVISLAB-Netzwerks. . . . 45
27 Volumenrendering im GVR . . . 48
28 Çomputergrafik-Kenntnisse der Umfrageteilnehmer . . . 51
29 Bestandteile Volumenrendering-Programm . . . 58
30 Ersetzung von Shadern eines Volumenrenderers . . . 60
31 Ebenen der Shaderbeschreibung . . . 66
32 Bestandteile eines Shadergraphen. . . . 66
33 Grafische Notation von Programm- und Uniformknoten . . . 71
34 Kompletter Shadergraph . . . 72
35 Veränderung eines Shadergraphen . . . 73
36 Idee der Schleifenparameter . . . 78
37 Grafische Notation eines Schleifen-Programmknotens . . . 79
38 Shadergraph mit Schleife . . . 81
39 Grundgerüst für Shadergraphen beim Volumenrendering . . . 83
40 Bestandteile der Umsetzung des Shadergraph-Konzepts. . . . 85
41 Klassendiagramm der SUL . . . 87
42 Verarbeitung von Komponentenlisten . . . 88

43 Verarbeitungsreihenfolge der Codeerzeugung 89
44 Programmbeschreibung der GVR-Shaderverwaltung 92
45 Caching in gvShaderIlluminatedSUL 93
46 Verbindungsdefinition in MEVISLAB-Panel 97
47 Wichtige Klassen des Projekts SOSUL. 98
48 Panel von SOSULCUSTOMNODE 99
49 Visuelle Datenflussprogrammierung mit SOSUL 101
50 Panel von SULProgramDescriptionInspector 102
51 Beispiel: Preintegriertes Volumenrendering 1 105
52 Beispiel: Preintegriertes Volumenrendering 2 106
53 Beispiel: Jittering 107
54 Beispiel: Lit Sphereshading 109
55 Beispiel: Raycasting-Netzwerk 110
56 Beispiel: Raycasting-Ergebnisse 111
57 Ergebnisse Geschwindigkeitstest 1 116
58 Ergebnisse Geschwindigkeitstest 2 117

Tabellenverzeichnis

1 Semantische Variablen in SH 39
2 Häufigkeit des Einsatzes von GVR-Features 53
3 Parametereigenschaften 68
4 Testfälle für den Leistungsvergleich. 112
5 Instruktionszwahl kompilierter Shader 113
6 Geschwindigkeitstests: Verwendete Computer. 114

Liste der Algorithmen

1 SH-Shader für 122
2 Objektorientierte Shader 123
3 Shaderalgebra in SH 124
4 Beispiel für Kontrollshader 124
5 GLSL-Shader für Volumenrendering 125
6 Codeblöcke für Programmknoten 126
7 Vertexprogramm aus einem Shadergraph 127
8 Fragmentprogramm aus einem Shadergraph 128

Literatur

[AGM06] Oliver Abert, Markus Geimer, and Stefan Müller. Direct and fast ray tracing of nurbs surfaces. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 2006.

[AMD06] AMD Inc. ATI CTM Guide. Technical Reference Manual. Version 1.01, 2006.

[AW90] Gregory D. Abram and Turner Whitted. Building block shaders. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, 1990.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, 2004.

[BG07] Stefan Bruckner and Meister Eduard Gröller. Style transfer functions for illustrative volume rendering. *Computer Graphics Forum*, 26(3):715–724, 2007.

[Bic92] Lubomir Bic. A process-oriented model for efficient execution of dataflow programs. In *Dataflow Computing: Theory and Practice*. 1992.

[CNS+02] Eric Chan, Ren Ng, Pradeep Sen, Kekoa Proudfoot, and Pat Hanrahan. Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2002.

[Coo84] Robert L. Cook. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 1984.

[Den74] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, 1974.

[EE02] K. Engel and T. Ertl. Interactive high-quality volume rendering with flexible consumer graphics hardware. In *Eurographics '02*, 2002.

[EKE01] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated

pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 2001.

[FHH04] Tim Foley, Mike Houston, and Pat Hanrahan. Efficient partitioning of fragment shaders for multiple-output hardware. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2004.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[FW04] Niklas Folkegård and Daniel Wesslén. Dynamic code generation for realtime shaders. In *Linköping Electronic Conference Proceedings*, 2004.

[GBD04] Frank Goetz, Ralf Borau, and Gitta Domik. An xml-based visual shading language for vertex and fragment shaders. In *Web3D '04: Proceedings of the ninth international conference on 3D Web technology*, 2004.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2), 1992.

[GD06] Frank Goetz and Gitta Domik. Visual shaditor: a seamless way to compose high-level shader programs. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Research posters*, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gri05] Rüdiger Grimm. *Digitale Komunikation*. Oldenburg Verlag, 2005.

[Har04] Shawn Hargraeves. *Generating Shaders from HLSL Fragments*, chapter 7.3. 2004.

[Hei05] Alan Heirich. Optimal automatic multi-pass shader partitioning by dynamic programming. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2005.

[HHH01] J.V. Hajnal, D.L.G. Hill, and D.J. Hawkes. *Medical Image Registration*. CRC Press, 2001.

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

[HL90] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, 1990.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1), 2004.

[Kuc07] Roland Kuck. Object-oriented shader design. In *Computer Graphics Forum*, 2007.

[KW03] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, 2003.

[LHJ99] Eric LaMar, Bernd Hamann, and Kenneth I. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *VIS '99: Proceedings of the conference on Visualization '99*, 1999.

[McG05] Morgan McGuire. *The SuperShader. ShaderX4, Kapitel 8.1*. Charles River Media, 2005.

[MeV] MeVis Research GmbH. MeVisLab Module Definition Language Reference. http://www.mevislab.de/fileadmin/docs/html/mdl/mevislabref.pdf. Zugriff: 15.01.2008.

[MGB07] Stefan Müller, Thorsten Grosch, and Matthias Biedermann. Computergrafik 2+3. Vorlesung Universität Koblenz-Landau, Campus Koblenz, Wintersemester 2006/2007.

[MH02] Michael McCool and Wolfgang Heidrich. *Real-Time Shading*. A. K. Peters, Ltd., 2002.

[Mica] Microsoft Corporation. Direct X 10 Pipeline Stages. http://msdn2.microsoft.com/en-us/library/bb205123.aspx. Zugriff: 15.01.2008.

[Micb] Microsoft Corporation. High Level Shading Language (HLSL). http://msdn2.microsoft.com/en-us/library/bb509561.aspx. Zugriff: 15.01.2008.

[Micc] Microsoft Corporation. Microsoft Visual Programming Language. Documentation. http://msdn2.microsoft.com/en-us/library/bb483127.aspx. Zugriff: 15.01.2008.

[Micd] Microsoft Corporation. Shader Models. http://msdn2.microsoft.com/en-us/library/bb509626.aspx. Zugriff: 15.01.2008.

[MK06] H.-O. Peitgen M. Koenig, F. Link. Combining multi-resolution volume rendering with per object shading. Unveröffentlich, 2006.

[MP00] M. Mosconi and M. Porta. Iteration constructs in data-flow visual programming languages. *Computer Languages*, 26(2–4), 2000.

[MQP02] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2002. Überarbeitete Version.

[MSPK06] Morgan McGuire, George Stathis, Hanspeter Pfister, and Shriram Krishnamurthi. Abstract shade trees. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, 2006.

[MT04] Michael McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh*. AK Peters Ltd, 2004.

[MTP+04] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, 2004.

[Mye86] B. A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, 1986.

[NK03] Zoltán Nagy and Reinhard Klein. Depth-peeling for texture-based volume rendering. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, 2003.

[Nvi] Nvidia Corporation. NVIDIA OpenGL Extension Specifications for the GeForce 8 Series Architecture. http://developer.download.nvidia.com/opengl/specs/g80specs.pdf. Zugriff: 15.01.2008.

[ODK+00] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon rendering on a stream architecture. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 2000.

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1), 2007.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3), 2002.

[PH04] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., 2004.

[PHF07] John Plate, Thorsten Holtkaemper, and Bernd Froehlich. A flexible multi-volume shader framework for arbitrarily intersecting multi-resolution datasets. *IEEE Transactions on Visualization and Computer Graphics*, 13(6), 2007.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[Pix] Pixar. The RenderMan Interface Specification. https://renderman.pixar.com/products/rispec/index.htm. Zugriff: 15.01.2008.

[RGW+03] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser. Smart hardware-accelerated volume rendering. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003*, 2003.

[Rit07] Felix Ritter. Visual Programming for Prototyping of Medical Imaging Applications. Turorial Material. http://www.mevislab.de/vis2007/, 2007. Zugriff: 11.01.2008.

[RLP07] Matthias Raspe, Guido Lorenz, and Stephan Palmer. Hierarchical and Object-Oriented GPU Programming. 2007. Eingereicht November 2007.

[Ros06] Randi J. Rost. *OpenGL Shading Language. Second Edition.* Addison-Wesley, 2006. Second Edition.

[Sch98] Stefan Schiffer. *Visuelle Programmierung*. Addison-Wesley, 1998.

[Sed02] Robert Sedgewick. *Algorithmen. Zweite Edition.* Pearson Studium, 2002.

[Sha92] John A. Sharp. A brief introduction to data flow. In *Dataflow Computing: Theory and Practice*. 1992.

[Shi02] Peter Shirley. *Fundamentals of Computer Graphics*. AK Peters, 2002.

[SM00] Heidrun Schumann and Wolfgang Müller. *Visualisierung. Grundlagen und allgemeine Methoden.* Springer-Verlag, 2000.

[SMGG01] Peter-Pike J. Sloan, William Martin, Amy Gooch, and Bruce Gooch. The lit sphere: a model for capturing npr shading from art. In *GRIN'01: No description on Graphics interface 2001*, 2001.

[SP06] Mark Segal and Mark Peercy. A performance-oriented data parallel virtual machine for gpus. Technical report, ATI Technologies, Inc., 2006.

[Spi] Wolf Spindler. The ML Programming Guide. http://www.mevislab.de/fileadmin/docs/html/mlguide/TheMLGuide.html. Zugriff: 15.01.2008.

[SWND05] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide. Second Edition.* Addison-Wesley, 2005.

[TD07] Matthias Trapp and Jürgen Döllner. Automated combination of real-time shader programs. In *Proccedings of Eurographics 2007*, 2007.

[Wer94a] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[Wer94b] Josie Wernecke. *The Inventor Toolmaker: Extending Open Inventor, Release 2*. Addison-Wesley Longman Publishing Co., Inc., 1994.