Programmierbare Graphikhardware

Ausarbeitungen des Seminars/Proseminars

Universität des Saarlandes
WS 2003/2004

Inhaltsverzeichnis

Daniel Fischer
Betreuer: Philipp Slusallek

Einführung in die Programmierbarkeit von Graphikkarten

Diese Einführung untersucht die Möglichkeiten heutiger programmierbarer Graphikkarten. Dabei werden die aktuellen Versionen 3.0 der Vertex Shader und Pixel Shader untersucht.

Dazu wird die traditionelle Graphik-Pipeline der neuen programmierbaren Pipeline gegenüber gestellt und auch kurz die bisherige Geschichte von Graphikkarten angerissen. Desweiteren werden die Unterschiede zwischen CPUs und den Graphikkarten-Prozessoren untersucht.

Den Abschluss bildet ein Ausblick auf zukünftige Entwicklungen.

1 Die traditionelle Graphik-Pipeline

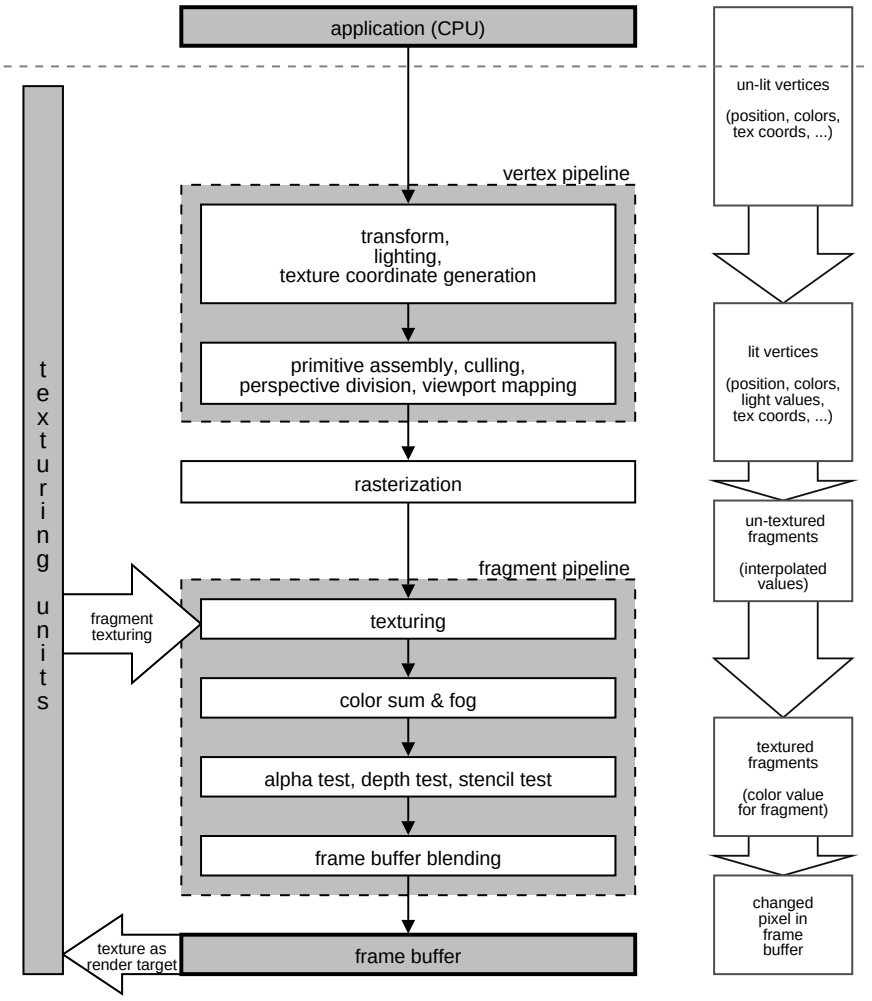
Die Aufgabe einer Graphik-Pipeline ist es, auf Basis ihrer Eingabe, die sich im Wesentlichen aus einer großen Anzahl von Vertices zusammensetzt, ein zweidimensionales Abbild (der durch die Vertices beschriebenen drei-dimensionalen Welt) zu berechnen.

Die traditionelle Graphik-Pipeline, die oft auch „fixed function pipeline“ genannt wird, lässt sich in eine Reihe von Pipeline-Stufen gliedern, von denen jede Stufe einen gewissen Teil dieser Arbeit übernimmt. Die Graphik-Pipeline lässt sich in drei Abschnitte aufteilen:

- die Vertex Pipeline,
- die Rasterisierung und
- die Fragment Pipeline.

In den Abschnitt Vertex-Pipeline fallen mehrere Pipeline-Stufen, die sich ausschließlich mit Operationen auf Vertices befassen. Die Rasterisierung ist die Pipeline-Stufe, die die Schnittstelle zwischen Vertex Pipeline und Fragment Pipeline darstellt. Die Pipeline-Stufen im Abschnitt Fragment Pipeline arbeitet dann schließlich auf Fragmenten.

Abbildung 1 traditionelle Graphik-Pipeline



1.1 Vertex Pipeline

Die Eingabe der Graphik-Pipeline und somit auch direkt der Vertex Pipeline sind die unbeleuchteten Vertices[1], die die Anwendung über das Graphik-API an

[1] ein *Vertex* (Plural Vertices) beschreibt einen Punkt im drei-dimensionalen Raum; neben der Position kann ein Vertex auch weitere Eigenschaften, wie z.B. eine Farbe, haben

die Graphikkarte übergibt. Für jeden Vertex werden dabei neben der Position in der Regel auch Farbwerte, Textur-Koordinaten und ähnliches übergeben. Die Vertex Pipeline führt auf den Vertices folgende Operationen durch:

- Transformation der Vertices in den Clipping Space (transformation)
- Beleuchtungs-Berechnung für die Vertices (lighting)
- Erzeugung von Textur-Koordinaten für die Vertices (texture coordinate generation)
- Wieder-Zusammensetzen der einzelnen Vertices zu Primitiven (primitive assembly)
- Elimination nicht sichtbarer Primitiven, z.B. eine Seite der Primitive sichtbare wäre, welche nicht gezeichnet werden soll (culling)
- Division der 3D-Koordinaten x, y und z durch die (vierte) homogene Komponente w (perspective division)
- Abbildung der drei-dimensionalen Vertex-Koordinaten auf Pixel-Koordinaten des zwei-dimensionalen Frame Buffers (viewport mapping)

Diese verschiedenen Operationen sind zugleich auch die Pipeline-Stufen, in die sich dieser Abschnitt der Graphik-Pipeline gliedern lässt. Die Ausgabe dieses Abschnittes sind Primitiven[2], die auf den zwei-dimensionalen Frame Buffer abgebildet wurden und für deren Eckpunkte (die Vertices) jeweils die Beleuchtungs-Berechnungen durchgeführt wurden.

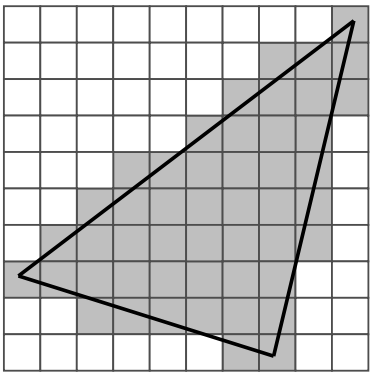
1.2 Rasterisierung

Die Pipeline-Stufe der Rasterisierung (rasterization) hat die Aufgabe, die auf den zwei-dimensionalen Frame Buffer abgebildeten Primitiven in Fragmente zu zerlegen. Jedes Fragment korrespondiert mit einem Bildpunkt (Pixel) des Frame Buffers und wird nach Durchlaufen der Fragment Pipeline mit dem entsprechenden Farbwert des Bildpunktes kombiniert. Die Bearbeitung der Gesamtheit aller Fragmente, in die eine Primitive zerlegt wurde, bewirkt schließlich das „Zeichnen“ der Primitive selbst.

In Abbildung 2 sehen Sie exemplarisch die Rasterisierung eines Dreieckes. Die grau umrandeten Quadrate repräsentieren die Flächen der einzelnen Bildpunkte im Frame Buffer. Sei nun also das abgebildete Dreieck zu zeichnen. Dann könnten wie in der Abbildung dargestellt 49 Fragmente bei der Rasterisierung erzeugt werden, von denen jedes einem der markierten Bildpunkte zugeordnet wäre. Im Randbereich des Dreieckes, gibt es Bildpunkte, die nicht mehr vollständig *innerhalb* des Dreieckes liegen. Die Entscheidung, für welche Bildpunkte in diesem Bereich noch Fragmente erzeugt werden, obliegt der Graphikkarte und kann von Graphikkarte zu Graphikkarte anders ausfallen (Stichwort Aliasing-Problem).

[2] eine *Primitive* ist eine einfache geometrische Figur, die effizient von der Graphikkarte gezeichnet werden kann (z.B. Dreieck); eine Primitive wird definiert durch ihre Eckpunkte in Form von Vertices

Abbildung 2 Beispiel einer Rasterisierung



Wie zuvor bereits erwähnt, wird jede Primitive durch ihre Eckpunkte definiert, wobei jeder Eckpunkt neben seiner Position weitere Eigenschaften wie z.B. Farben und Textur-Koordinaten haben kann. Desweiteren wurden im der Pipeline-Stufe „lighting“ für jeden Eckpunkt Beleuchtungswerte berechnet.

Zunächst haben die erzeugten Fragmente außer ihrer Position keine weiteren Eigenschaften. Die zusätzlichen Eigenschaften stehen bisher nur in den Eckpunkten zur Verfügung, sollen aber nach der Rasterisierung für jedes Fragment zur Verfügung stehen.

Die zusätzlichen Eigenschaften der Vertices lassen sich nicht direkt auf die Fragmente übertragen, da es offensichtlich wesentlich weniger Vertices als Fragmente gibt. Eine individuelle Berechnung der zusätzlichen Eigenschaften für jedes Fragment

- wäre im Falle der in der Pipeline-Stufe „lighting“ berechneten Beleuchtungswerte grundsätzlich möglich, ist aber im Allgemeinen aufgrund des enormen Rechenzeitaufwandes praktisch nicht realisierbar
- ist im Falle der anderen zusätzlichen Eigenschaften grundsätzlich nicht möglich, da ausschließlich die Anwendung Kenntnisse über den Algorithmus zur Berechnung dieser Werte haben kann

Aus diesem Grund wird die Annahme gemacht, dass sich die zusätzlichen Eigenschaften im Bereich *zwischen* den Eckpunkten nur linear ändern. Mit dieser Annahme ist zum Beispiel der Farbwert eines Fragmentes, das exakt in der Mitte zwischen zwei Eckpunkten einer Primitive liegt, gleich dem arithmetischen Mittel der Farbwerte für diese beiden Eckpunkte. Dieses Konzept der Interpolation wird analog auf alle Eckpunkte erweitert, so dass auf Basis der Zusatz-Eigenschaften der Eckpunkte für jedes Fragment Zusatz-Eigenschaften berechnet werden können. Die Interpolation wird dabei perspektivisch korrekt durchgeführt, so dass durch die perspektivische Sicht bedingten Verzerrungen ausgeglichen werden.

Diese Strategie liefert im Allgemeinen sehr gute Resultate, da sich viele denkbare Zusatz-Eigenschaften, wie zum Beispiel Farbwerte und Textur-Koordinaten

in der Regel nur linear ändern und die Interpolation somit tatsächlich den korrekten Wert für jedes Fragment berechnet. Andere Zusatz-Eigenschaften, wie zum Beispiel die in der Pipeline-Stufe „lighting“ berechneten Beleuchtungswerte, ändern sich jedoch *nicht* linear. Die Interpolation liefert in diesem Fall für jedes Fragment lediglich eine Annäherung für den richtigen Wert. Je nach Situation kann diese Annäherung so ungenügend sein, dass die Darstellungsqualität spürbar sinkt. Die einzige Möglichkeit, diese Folge zu vermeiden, ist die entsprechende Situation zu umgehen. Weitere Möglichkeiten eröffnen sich erst durch Einführung der Programmierbarkeit der Graphik-Pipeline.

1.3 Fragment Pipeline

Die Fragment Pipeline arbeitet auf den Fragmenten, die bei der Rasterisierung erzeugt werden. Auf Basis der bei der Rasterisierung interpolierten Werte, werden für jedes Fragment folgende Operationen durchgeführt:

- Ermittlung der entsprechenden Textur-Farbwerte anhand der interpolierten Textur-Koordinaten (texturing)
- Kombination der ermittelten Textur-Farbwerte mit den interpolierten Farbwerten und Beleuchtungs-Werten (color sum)
- Nebel-Berechnungen (fog)
- Durchführung verschiedener Tests, die die endgültige Sichtbarkeit des Fragmentes bestimmen (alpha test, depth test, stencil test)
- Kombination / Mischung des für das Fragment berechneten Farbwertes mit dem bisherigen Farbwert des entsprechenden Bildpunktes im Frame Buffer (frame buffer blending)

Auch hier sind diese verschiedenen Operationen zugleich die Pipeline-Stufen, in die sich dieser Abschnitt der Graphik-Pipeline gliedern lässt.

Nach Bearbeitung aller Vertices und der aus ihnen entstehenden Fragmente, ist die zwei-dimensionale Abbildung der von der Anwendung übermittelten dreidimensionalen Welt komplett berechnet und kann auf dem Bildschirm angezeigt werden.

Wie sich die einzelnen Pipeline-Stufen im Detail verhalten, wird über Modi festgelegt. Oft gibt es viele Modi, die das Verhalten *einer* Stufe steuern. Ferner gibt es auch Modi, die mehrere Stufen gleichzeitig beeinflussen.

In Abbildung 1 sehen Sie zusammenfassend die einzelnen Pipeline-Stufen der traditionellen Graphik-Pipeline im Überblick.

2 Bisherige Hardware-Implementierungen

2.1 Multi-Chip-Lösungen

Zu Beginn der Entwicklung hardware-beschleunigter Graphik teilten sich mehrere eigenständige, spezialisierte Prozessoren die Arbeit. Die Zwischenergebnisse wurden zwischen den Prozessoren über externe Datenbusse ausgetauscht. Dies war zugleich ein Leistungs-begrenzender Faktor, da

- bei externen Datenbussen gegenüber Chip-internen Datenbussen keine so hohen Taktraten möglich sind und
- der Bandbreite externer Busse (äquivalent zur Anzahl der parallelen Datenleitungen) Grenzen auferlegt sind (durch die nicht beliebig erhöhbare Pin-Anzahl der Chips).

Ein interessantes Beispiel aus dieser Zeit ist die –für damalige Verhältnisse– sehr leistungsfähige Reality Engine [1].

In dieser Zeit waren häufig einzelne Prozessoren der Pipeline auch in gewissem Rahmen programmierbar. Diese Programmierbarkeit wurde jedoch nicht über APIs zur Verfügung gestellt, so dass sie keine Bedeutung für die Anwendungsentwicklung hatte.

2.2 Ein-Chip-Lösungen

Im Rahmen der Fortschritte in der Halbleiter-Industrie konnten immer mehr Transistoren –und damit auch mehr Funktionalität– in einem Chip realisiert werden. Dadurch wurde es schließlich möglich, die gesamte Graphik-Pipeline in einem Chip zu realisieren, anstatt wie bisher die Funktionalität auf mehrere Chips aufteilen zu müssen.

Der Vorteil dieser Entwicklung war, dass aufgrund der kürzeren Signalwege und dem Wegfallen vieler externer Datenbusse höhere Taktraten und damit auch höhere Berechnungsgeschwindigkeiten möglich wurden. Ein Seiteneffekt dieser Entwicklung war, dass die (wenn auch geringe) Programmierbarkeit der Miniaturisierung und Chip-Optimierung zum Opfer fiel.

2.3 Modi-Problem

Wie bereits oben erwähnt wurde, verwendete die traditionelle Graphik-Pipeline diverse Modi, um das Verhalten der einzelnen Pipeline-Stufen zu beeinflussen. Dies hat sich auch in den APIs zur Steuerung der Graphik-Hardware niedergeschlagen, welche dieses Konzept ebenfalls übernommen haben.

Durch die beständig fortschreitende Entwicklung der Graphik-Hardware wurden immer mehr Funktionalitäten in die Graphikkarten integriert, welche durch immer neue Modi den Anwendungen zugänglich gemacht wurden. Immer häufiger trat auch das Problem auf, dass sich die Funktionalitäten nicht beliebig in Modi faktorisieren ließen und es somit zu einer regelrechten Explosion der Anzahl der Modi kam.

2.4 Programmierbarkeit

Durch die Einführung der programmierbaren Graphik-Pipeline wird unter anderem auch dieses Problem gelöst. Anstatt das Verhalten von Abschnitten der Pipeline mit Modi zu beeinflussen, wird das Verhalten der Abschnitt direkt *programmiert*. Dies gibt der Anwendung *wesentlich* mehr Kontrolle über das Verhalten der Pipeline, als es mit einer (zwangsläufig) begrenzten Anzahl von Modi jemals möglich wäre. Die Modi, die früher das Verhalten der jetzt programmierbaren Pipeline-Abschnitte steuerten, werden somit überflüssig.

2.5 Kompatibilität

Sofern sich die Anwendung jedoch entscheidet, die Programmierbarkeit der Graphik-Pipeline *nicht* zu nutzen, gibt es zwei Möglichkeiten.

Eine Möglichkeit ist, dass die Graphik-Hardware sowohl die neue programmierbare Pipeline als auch die traditionelle Pipeline realisiert. Wenn eine Anwendung die Programmierbarkeit nicht nutzen will, wird anstatt der programmierbaren Pipeline einfach die traditionelle verwendet. Diese Lösung hat den Vorteil, dass auch ältere Programme von einer optimierten Pipeline profitieren, wie sie auf den letzten Generationen der nicht-programmierbaren Graphikkarten existierten. Der große Nachteil dieser Lösung ist, dass statt einer Pipeline immerhin zwei Pipelines auf dem Chip integriert werden müssen.

Eine andere Möglichkeit ist, dass die Graphik-Hardware ausschließlich die neue programmierbare Pipeline realisiert. Wenn eine Anwendung die Programmierbarkeit nicht nutzen will, werden automatisch entsprechende Shader-Programme für die programmierbaren Pipeline-Abschnitte geladen, die die Funktionalität der traditionellen Pipeline *emulieren*. Nachteilig ist bei dieser Variante die potentiell geringere Ausführungsgeschwindigkeit der emulierten traditionellen Pipeline. Ein großer Vorteil dieser Variante ist jedoch der geringere Hardware-Implementierungsaufwand, da nur eine Pipeline und nicht zwei auf dem Chip untergebracht werden müssen.

Zur Zeit geben die meisten Graphikkarten-Hersteller der ersten Variante den Vorzug, da die traditionelle Pipeline jahrelang entwickelt und optimiert wurde und nun einfach die programmierbare Pipeline ergänzt wird. Es ist jedoch zu vermuten, dass die Weiterentwicklung der traditionellen Pipeline in absehbarer Zeit eingestellt wird. Dann wird unter anderem auch im Hinblick auf die geringeren Hardware-Herstellungskosten voraussichtlich die zweite Variante zunehmend Oberhand gewinnen.

Ältere Anwendungen, die noch keine Kenntnis von der programmierbaren Pipeline haben, können bei beiden Varianten ohne Änderung ausgeführt werden, da aus Kompatibilitätsgründen die traditionelle Pipeline weiterhin vollkommen transparent zur Verfügung gestellt wird.

3 Die programmierbare Graphik-Pipeline

3.1 Gemeinsamkeiten & Unterschiede

Die programmierbare Graphik-Pipeline hat exakt die selbe Struktur, die selben Eingaben und die selben Ausgabe wie die traditionelle Graphik-Pipeline. Dies begründet sich vor allem in der Tatsache, dass sich diese Struktur bereits bewährt hat.

Der Unterschied zwischen der programmierbaren und traditionellen Graphik-Pipeline ist also nicht darin zu suchen, *was* die Pipeline tut, sondern darin, *wie* die Anwendung das Verhalten der Pipeline steuert.

3.2 programmierbare Abschnitte

Zur Einführung der Programmierbarkeit wurden zwei Abschnitte –bestehend aus jeweils mehreren Pipeline-Stufen– der Graphik-Pipeline ausgewählt. Zum einen werden die Pipeline-Stufen

- Transformation der Vertices in den Clipping Space (transformation)
- Beleuchtungs-Berechnung für die Vertices (lighting)
- Erzeugung von Textur-Koordinaten für die Vertices (texture coordinate generation)

in Form eines sogenannten „Vertex Shaders“ programmierbar. Zum anderen werden die Pipeline-Stufen

- Ermittlung der entsprechenden Textur-Farbwerte anhand der interpolierten Textur-Koordinaten (texturing)
- Kombination der ermittelten Textur-Farbwerte mit den interpolierten Farbwerten und Beleuchtungs-Werten (color sum)
- Nebel-Berechnungen (fog)

in Form eines sogenannten „Pixel Shaders“ programmierbar.

3.3 Ersetzung der alten Pipeline-Stufen

Ein Shader ist ein Programmcode, welches durch die Anwendung auf die Graphikkarte übertragen wird. Je nachdem, ob das Programm als Vertex Shader oder als Pixel Shader deklariert wurde, *ersetzt* es die jeweiligen oben aufgezählten Pipeline-Stufen.

Angenommen eine Anwendung wollte lediglich die Funktionalität der Beleuchtung in Form eines Shaders selbst programmieren. Dann müsste die Anwendung in dem Shader *zusätzlich* auch die Arbeit der Pipeline-Stufen „Transformation“

und „Erzeugung von Textur-Koordinaten“ übernehmen, da ein Vertex Shader diese Pipeline-Stufen komplett ersetzt.

Dadurch, dass die Shader die entsprechenden Pipeline-Stufen der traditionellen Pipeline *ersetzen*, entfallen auch die Modi, die früher das Verhalten *dieser* Pipeline-Stufen gesteuert haben. Die Modi werden lediglich noch aus Kompatibilitätsgründen für den Fall unterstützt, dass eine Anwendung die Programmierbarkeit *nicht* verwendet.

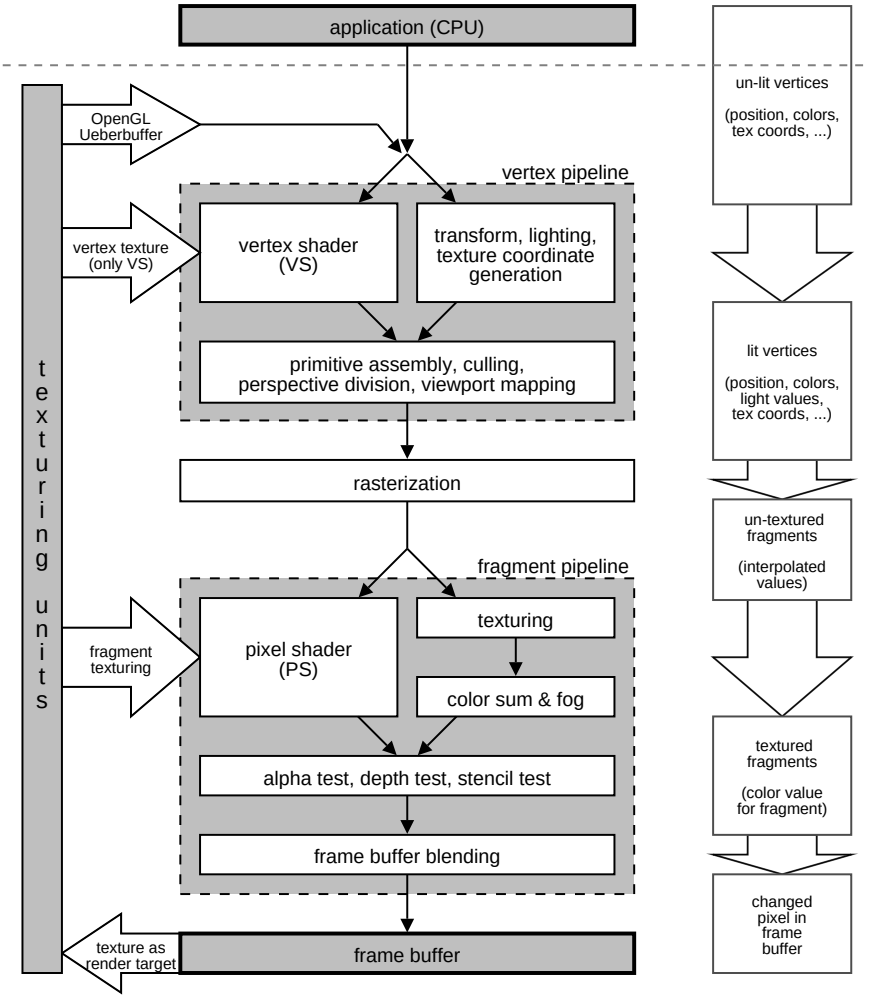
Die Shader-Programme für Vertex Shader und für Pixel Shader haben grundsätzlich einen ähnlichen Aufbau und ähnliche Möglichkeiten. Bedingt durch die Tatsache, dass Vertex Shader und Pixel Shader grundverschiedene Eingaben und Ausgaben haben, sind sie logischerweise nicht austauschbar.

3.4 Kommunikation

Verwendet eine Anwendung *gleichzeitig* Vertex Shader und Pixel Shader (dies wird der Regelfall sein) dann können die beiden Shader-Programme in einem gewissen Rahmen direkt miteinander kommunizieren, da die sich Eingabe des Pixel Shaders aus den Ausgaben des Vertex Shaders ableitet. Weitere Möglichkeiten der Kommunikation zwischen Vertex Shader und Pixel Shader werden in Kapitel 4 vorgestellt.

Einen Überblick über die Kombination von programmierbarer und traditioneller Pipeline in aktuellen Graphikkarten erhalten Sie durch das Blockdiagramm in Abbildung 3.

Abbildung 3 Kombination programmierbare & traditionelle Pipeline



4 Rückkopplung über den Texturspeicher

Mit steigender Komplexität der darzustellenden Szenen, lässt sich ein Einzelbild oft nicht mehr in einem Render-Durchlauf (single pass), bestehend aus

- Löschung des Frame Buffers,
- Setzen von Graphikkarten-Parametern und
- Übertragung der zu rendernden Welt in Form von Vertices an die Graphikkarte

vollständig berechnen. Dieser Fall tritt zum Beispiel beim Rendern von spiegelnden Flächen auf.

4.1 multi-pass-Strategien

Die Berechnung muss dann in mehrere Render-Durchläufe aufgeteilt werden (multi pass). Für die Aufteilung gibt es grundsätzlich zwei Strategien:

- die einzelnen Durchläufe rendern in den Frame Buffer, allerdings kombinieren sie jeweils ihr berechnetes Bild mit dem vorherigen Inhalt des Frame Buffers anstatt diesen zu löschen, das endgültige Bild ergibt sich somit durch Überlagerung und Mischung der berechneten Einzelbilder
- die einzelnen Durchläufe rendern in den Texturspeicher anstatt in den Frame Buffer, der letzte Durchlauf berechnet auf Basis der im Texturspeicher abgelegten Informationen das endgültige Bild und speichert dieses im Frame Buffer

Diese Strategien werden oft auch kombiniert. Für die zweite Strategie ergeben sich einige Neuerungen, die im folgenden vorgestellt werden.

4.2 bisherige Möglichkeiten

Vor der Einführung der programmierbaren Graphik-Pipeline konnten im Texturspeicher abgelegte Daten ausschließlich bei der Texturierung von Fragmenten *direkt* verwendet werden. Sollten die Daten aus dem Texturspeicher an anderer Stelle verwendet werden, war die einzige Möglichkeit, den Texturspeicher mittels der CPU in den Systemspeicher zu kopieren und dann wieder mittels der CPU an diese Stelle zu übertragen. Offensichtlich hatte diese „Notlösung“ zwei Nachteile:

- erhöhte CPU-Last und somit weniger Zeit für andere CPU-Aufgaben (wie zum Beispiel die „Künstliche Intelligenz“ in Spielen)
- zweimalige Übertragung der gesamten Daten über den im Vergleich zur Graphikkarte langsamen Systembus (in der Regel AGP-Bus)

Trotz der immer weiter steigenden Geschwindigkeiten von CPU und Systembus möchte man den Umweg über die CPU weitestgehend vermeiden.

4.3 neue Möglichkeiten

Mit Einführung der Vertex Shader 3.0 gibt es mittlerweile auch die Möglichkeit im Vertex Shader auf den Texturspeicher zuzugreifen. Der Vertex Shader kann also pro Vertex mehrere Male auf den Texturspeicher (lesend) zugreifen und auf Basis der gelesenen Daten zum Beispiel die Position des Vertex verändern.

Durch die geplante Einführung der sogenannten Überbuffer bzw. Superbuffer, wird es möglich werden, Daten aus dem Texturspeicher sogar als Eingabe für die gesamte Graphik-Pipeline zu verwenden, also als Reihe von Vertices zu interpretieren. Dies bedeutet, dass es möglich sein wird, Szenen-Geometrie auf der Graphikkarte selbst zu berechnen. Im Gegensatz dazu musste bisher immer die komplette Geometrie durch die CPU berechnet und an die Graphikkarte übertragen werden (auf der Graphikkarte konnten keine *neuen* Vertices erzeugt werden, sondern nur bereits existierende verändert werden).

Das bemerkenswerte an dieser Entwicklung ist, dass somit eine Rückkopplung über die gesamte Graphik-Pipeline möglich wird, ohne die CPU und den Systembus zur Datenübertragung zu verwenden. Denn wenn die letzte Pipeline-Stufe Daten an die erste Pipeline-Stufe übermitteln kann, ist die praktische Konsequenz, dass jede Pipeline-Stufen an jede andere Daten übertragen kann.

4.4 Einschränkungen

Es ist prinzipiell *nicht* möglich, innerhalb *eines* Render-Durchlaufs Daten zwischen tieferen und höheren Pipeline-Stufen auszutauschen. Es ist zum Beispiel *nicht* zulässig, eine Textur in einer beliebigen Pipeline-Stufe als Eingabe zu verwenden, die im *selben* Render-Durchlauf ein Render-Ziel ist. Der Datenaustausch ist immer nur zwischen *verschiedenen* Durchläufen möglich. Näheres zu den Gründen für diese Einschränkung erfahren Sie im Kapitel 6.

5 Funktionalitäten in den Shader-Programmen

5.1 Datenfluss & Register

Bereits in Kapitel 3 wurden die beiden Typen von Shader-Programmen

- Vertex Shader und
- Pixel Shader

vorgestellt und in die Graphik-Pipeline eingeordnet. Jeder der beiden Shadertypen ersetzt mehrere Pipeline-Stufen. Die Shader erhalten die Eingabedaten der jeweils ersten ersetzten Pipeline-Stufe und geben die selbe Art von Ausgabedaten zurück, die die jeweils letzte ersetzte Pipeline-Stufe zurückgeben würde.

Die Shader kommunizieren mit dem Rest der Graphik-Pipeline über Register[3]. Es gibt mehrere verschiedene Arten von Registern mit unterschiedlichen Eigenschaften, wobei jede Registerart einem bestimmten Verwendungzweck zugeordnet ist.

Sofern nicht anders angegeben sind alle Register Vektoren bestehend aus jeweils vier unabhängigen Komponenten. Diese Komponenten heißen in beiden Shadertypen `x`, `y`, `z` und `w` bzw. äquivalent `r`, `g`, `b` und `a`. Die Komponenten der Vektoren speichern dabei sofern nicht anders angegeben Fließkommazahlen.

5.2 Eingabe- und Ausgaberegister

Die Eingabedaten werden dem Shader in speziellen (nur lesbaren) Eingaberegistern zur Verfügung gestellt. Die berechneten Ergebnisse müssen vom Shader in spezielle (nur schreibbare) Ausgaberegister geschrieben werden. Die Anzahl der Eingabe- und Ausgaberegister unterscheidet sich zwischen Vertex Shader und Pixel Shader.

5.3 Deklaration

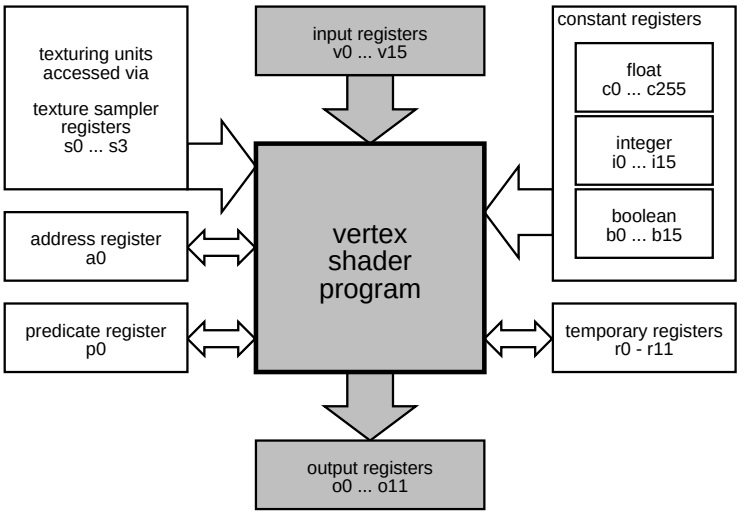
Die einzelnen Eingabe- und Ausgaberegister haben dabei *keine* feste Semantik, d.h. es ist nicht fest vorgegeben, in welchem Register welche Eingabe bzw. Ausgabe steht. Diese Verbindung wird erst durch Deklarationen hergestellt. Das Shader-Programm muss vor Verwendung der jeweiligen Register mit Deklarationen festlegen, in welchem Eingaberegister es welche Eingabe erwartet und in welchem Ausgaberegister die Graphikkarte welche Ausgabe erwarten kann.

5.4 konstante Register

Da die Anzahl der Eingaberegister sehr stark beschränkt ist und diese Register in der Regel nicht ausreichen, um alle Eingaben zu transportieren, die ein Shader

[3] *Register* sind vereinfacht ausgedrückt Speicherzellen, auf die das Programm zugreifen kann

Abbildung 4 Vertex Shader Datenfluss



benötigt, gibt es eine große Anzahl von konstanten Registern. Wie der Name bereits deutlich macht, können konstante Register ausschließlich gelesen werden.

Die Eingaberegister der Shader ändern in der Regel mit jedem weiteren Vertex bzw. Fragment, da sie die Eingabedaten der jeweils ersten ersetzten Pipeline-Stufe enthalten. Die konstanten Register werden hingegen *nicht* selbständig geändert, sondern müssen explizit von der Anwendung geändert werden. Es ist somit offensichtlich, dass sich der Inhalt dieser Register *nicht* mit jedem weiteren Vertex bzw. Fragment ändern kann. Da die kleinste Einheit, die die Anwendung zum Zeichnen an die Graphikkarte übertragen kann, eine Primitive ist, können sich die konstanten Register *höchstens* von Primitive zu Primitive ändern.

Bei den konstanten Registern stehen drei verschiedene Datentypen zur Verfügung. Es gibt konstante Register, die Vektoren aus Fließkommazahlen sind, konstante Register, die Vektoren aus ganzen Zahlen sind, und konstante Register, die skalare[4] boole'sche Werte[5] sind. Die Anzahl der verfügbaren konstanten Register ist je nach Shadertyp und Datentyp unterschiedlich.

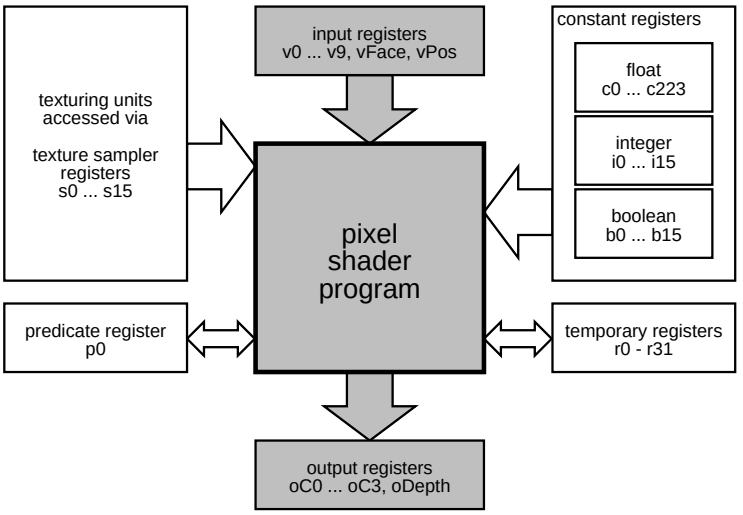
5.5 temporäre Register

Um temporäre Zwischenergebnisse speichern und später wieder laden zu können, stehen den Shadern eine begrenzte Anzahl von temporären Registern zur Verfügung, die sowohl gelesen als auch geschrieben werden können. Zu Beginn des Shader-Programms sind die Inhalte dieser Register undefiniert.

[4] ein *Skalar* ist ein einzelner Wert eines bestimmten Typs, im Gegensatz zu Vektoren, die sich aus mehreren Werten eines Typs zusammensetzen

[5] ein *boole'scher Wert* ist ein Wahrheitswert, der nur die Zustände wahr oder falsch annehmen kann

Abbildung 5 Pixel Shader Datenfluss



5.6 Spezialregister

Das Adress-Register ist ein Vektor aus ganzen Zahlen und dient der relativen Adressierung, welche in Kapitel 5.14 beschrieben wird.

Das Predicate-Register ist ein Vektor aus boole'schen Werten und dient einer besonderen Form der Flußsteuerung, die in Kapitel 5.21 beschrieben wird.

Die Textur-Sampler-Register dienen dem Zugriff auf den Texturspeicher. Dies sind keine Register wie die anderen Register, da sie lediglich Referenzen auf die zu verwendende Texturing Unit darstellen. Analog zu den Eingabe- und Ausgaberegistern müssen die Textur-Sampler-Register vor Verwendung deklariert werden, um den Typ der Textur (zwei-dimensional, drei-dimensional, ...) festzulegen.

5.7 Modifikatoren für Register und Instruktionen

Beim Lesen und Schreiben von Registern sind verschiedene Modifikationen des Lese- bzw. Schreibverhaltens möglich. Mit diesen Modifikatoren ist es oft möglich, komplizierte Operationen, für die man normalerweise mehrere Instruktionen bräuchte, in einer Instruktion auszudrücken. Ferner gibt es auch Modifikatoren, die das Verhalten von Befehlen ändern können.

Nicht alle der Modifikatoren können auf jedem Register bzw. jeder Instruktion angewendet werden und nicht alle Modifikatoren können zusammen verwendet werden. Für genaue Angaben zu den erlaubten Konstellationen konsultieren Sie bitte eine entsprechende Shader-Referenz [2].

5.8 Swizzling

Das Swizzling ist eine sehr leistungsfähige Möglichkeit, das Verhalten beim Lesen von Registern zu verändern. Die häufigsten Instruktionen arbeiten mit Vektoren und lesen als Eingabe somit auch Vektor-Register, bestehend aus jeweils vier Komponenten `x`, `y`, `z`, `w` bzw. äquivalent `r`, `g`, `b`, `a` ein.

Normalerweise wird die Komponente `x` des Eingaberegisters von der Instruktion auch als erste Komponente `x` behandelt, die Komponente `y` als zweite Komponente `y` usw. Mit Swizzling können beim Lesen des Vektors die einzelnen Komponenten vertauscht werden oder durch den Inhalt anderer Komponenten ersetzt werden. Dabei wird jedoch der Inhalt des gelesenen Registers *nicht* verändert.

Um Swizzling zu verwenden wird an den Registernamen getrennt durch einen Punkt eine Aufzählung von vier hintereinanderstehenden Buchstaben angehängt, wobei jeder dieser Buchstaben eine Komponente identifiziert. Der Buchstabe an der ersten Stelle gibt an, welche Komponente nach dem Lesen als erste Komponente verwendet werden soll, der Buchstabe an der zweiten Stelle, welche Komponente als zweite verwendet werden soll usw.

Die Angabe `r0.wzyx` würde zum Beispiel bewirken, dass die Komponente `w` als erste Komponente des Registers `r0` interpretiert würde, die Komponente `z` als zweite, die Komponente `y` als dritte und die Komponente `x` als vierte. Beim Lesen würde also die Reihenfolge der Vektor-Komponenten gespiegelt werden.

Die Angabe `r0.rbbb` würde zum Beispiel bewirken, dass die Komponente `r` als erste Komponente des Registers `r0` interpretiert würde und die Komponente `b` als zweite, dritte und vierte. Die Komponenten `g` und `a` würden dabei nicht verwendet werden.

Wenn am Ende der Komponentenliste die selbe Komponente mehrfach aufgeführt ist, können die wiederholten Ausführungen dieser Komponente weggelassen werden. Somit kann die Komponentenliste auch kürzer als vier Zeichen werden. So kann zum Beispiel `r0.rbbb` auf `r0.rb` und `r0.xyxx` auf `r0.xyx` verkürzt werden.

Das Swizzling ist eine Funktionalität von Graphikkarten, die CPUs nicht zur Verfügung stellen. Auf einer CPU muss eine solche Funktionalität durch mehrere Instruktionen mit einem entsprechenden Rechenzeit-Mehrbedarf nachgebildet werden, während auf Graphikkarten Swizzling ohne zusätzliche Rechenzeit durchgeführt werden kann.

5.9 Negierung & Absolutwert

Beim Lesen von Registern kann deren Inhalt auch direkt negiert werden. Um die Negierung zu verwenden wird dem Registernamen ein Minuszeichen vorangestellt. Alle Komponenten des Registers werden dabei gleichermaßen negiert.

Ferner kann beim Lesen von Registers deren Absolutwert gebildet werden. Dazu wird dem Registernamen die Zeichenkette `abs` vorangestellt. Der Absolutwert wird für jede Komponente einzeln gebildet.

5.10 Masking

Beim Schreiben von Registern kann das Schreiben einzelner Komponenten unterdrückt werden. Analog zum Swizzling wird dabei dem Registernamen getrennt durch einen Punkt eine Liste von bis zu vier Buchstaben angehängt, die angeben, welche Komponenten im Zielregister *tatsächlich* geschrieben werden sollen. Ohne die Verwendung von Masking wird immer die Maske `xyzw` vorausgesetzt.

Im Gegensatz zum Swizzling von Quellregistern ist beim Masking von Zielregistern *keine* Vertauschung von Komponenten und *keine* Vervielfältigung von Komponenten möglich. Die Komponentenangaben *müssen* immer in der Reihenfolge `x` vor `y` vor `z` vor `w` stehen (bzw. äquivalent `r` vor `g` vor `b` vor `a`).

5.11 Sättigung

Das Verhalten einer Instruktion kann mit dem Sättigungs-Modifikator verändert werden. Wird dem Namen einer Instruktion die Zeichenkette `_sat` angehängt, dann wird das Resultat der Instruktion *vor* dem Schreiben komponentenweise auf den Bereich 0.0 bis 1.0 beschränkt. Somit wird jede Komponente, die kleiner als 0.0 ist, durch den Wert 0.0 ersetzt und jede Komponente, die größere als 1.0 ist, durch den Wert 1.0.

5.12 arithmetische Operationen

Die arithmetischen Operationen arbeiten praktisch ausschließlich auf Fließkomma-Zahlen. Die meisten Instruktionen sind SIMD-Instruktionen[6] und arbeiten gleichzeitig auf allen vier Komponenten der Register.

Somit sind die verfügbaren arithmetischen Operationen optimal für die Anwendung in Graphik-Berechnung, da diese Berechnung praktisch nur aus Fließkomma-Berechnungen bestehen und fast immer für alle vier Vektor-Komponenten die selbe Operation ausgeführt werden muss.

Desweiteren gibt es für alle arithmetischen Operationen, die üblicherweise im Graphik-Bereich Verwendung finden, entsprechende Instruktionen. Während auf normalen Mehrzweck-Prozessoren solche Operationen häufig durch die Kombination mehrerer Instruktionen nachgebildet werden müssen, stehen sie auf spezialisierten Graphik-Prozessoren direkt zur Verfügung. Dies kann sich auch in zum Teil erheblichen Geschwindigkeits-Unterschieden manifestieren.

Vertex Shader und Pixel Shader unterstützen im Wesentlichen die gleichen arithmetischen Instruktionen. Für eine komplette Liste aller unterstützten Instruktionen konsultieren Sie bitte eine entsprechende Shader-Referenz [2].

[6] eine *SIMD-Instruktion* (Single Instruction Multiple Data) führt die selbe Operation gleichzeitig auf mehreren Eingabedaten aus

5.13 Zugriff auf den Texturspeicher

Sowohl im Pixel Shader als auch im Vertex Shader gibt es die Möglichkeit, auf Basis von Textur-Koordinaten einen Lesezugriff in den Texturspeicher durchzuführen. Im Pixel Shader steht dazu eine Reihe von verschiedenen Instruktionen zur Verfügung, während im Vertex Shader nur eine Instruktion für einen einfachen Texturzugriff zur Verfügung steht.

Weitere Informationen zur den verfügbaren Textur-Instruktionen und deren Funktionsweise entnehmen Sie bitte einer entsprechenden Shader-Referenz [2].

5.14 relative Adressierung

Normalerweise werden Register direkt über ihren Namen angesprochen, also zum Beispiel das temporäre Register mit dem Index 2 über `r2`. Es besteht jedoch auch die Möglichkeit, Register in Abhängigkeit von einem Adressregister auszuwählen. Die Indexnummer des Registers ist dann die Summe aus dem Inhalt des Adressregisters und einer konstanten Zahl.

Die Registerangabe `r[aL+2]` greift zum Beispiel für `aL=0` auf `r2` und für `aL=5` auf `r7` zu. Der ganzzahlige, skalare Schleifenzähler `aL`, steht in beiden Shadertypen zur Verfügung, jedoch nur innerhalb einer Schleife.

Im Vertex Shader gibt es zusätzlich auch einen Adressvektor (Register `a0`). Dieser ganzzahlige Vektor kann mit einer speziellen Instruktion auf Basis eines Fließkommavektors geladen werden, wobei jede Komponente des Fließkommavektors mathematisch genau gerundet wird, ehe sie in die entsprechende Komponente des Adressvektors geladen wird. Zur Adressierung auf Basis des Adressvektors muss immer *eine* Komponente des Vektors ausgewählt werden. Soll zum Beispiel die zweiten Komponente des Adressvektors `a0` anstatt des Schleifenzählers `aL` verwendet werden, so ist `aL` durch `a0.y` zu ersetzen.

5.15 Flusskontrolle

Bei den Instruktionen zur Flusskontrolle muss beachtet werden, dass sich diese nicht beliebig tief ineinander schachteln lassen. Welche maximale Schachtelungstiefe erlaubt ist, entnehmen Sie bitte einer entsprechenden Shader-Referenz [2].

Wenn die Instruktionen zur Flusskontrolle ihre Entscheidungen und ihre Anzahl von Schleifendurchläufen *ausschließlich* auf der Basis von konstanten Registern bestimmen, spricht man von statischer Flusskontrolle, da der Programmablauf des Shaders bereits beim Start des Shader-Programms bekannt ist. Sobald ein nicht-konstantes Register in die Entscheidungen einbezogen wird, spricht man von dynamischer Flusskontrolle.

5.16 Konditional

Blöcke in einem Shader-Programm können in Abhängigkeit von dem Ergebnis eines Vergleichs zweier skalarer Größen ausgeführt oder nicht ausgeführt

werden. Dazu wird aus zwei (nicht notwendigerweise verschiedenen) Eingaberegistern jeweils eine Komponente ausgewählt und diese beiden Komponenten miteinander verglichen. Dabei stehen die üblichen Vergleichsoperationen[7] zur Verfügung. Anstatt des Vergleiches kann im Konditional auch eine boole'sche Konstante als Bedingung dienen.

Bei jedem Konditional kann ein Block von Instruktionen angegeben werden, der ausgeführt wird, wenn die Bedingung eintritt, und ein Block, der ausgeführt wird, wenn die Bedingung nicht eintritt.

5.17 Wiederholungsblock

Ein einfacher Wiederholungsblock führt den eingeschlossenen Block von Instruktionen mehrfach aus. Dazu muss aus einem ganzzahligen Vektor eine Komponente ausgewählt werden, deren Wert die Anzahl der Wiederholungen des Blocks angibt.

Die derzeit maximal zulässige Anzahl von Wiederholungen ist 255. Ein Wiederholungsblock kann durch eine spezielle Instruktion auch vorzeitig verlassen werden. Das vorzeitige Verlassen kann bedingt durch einen Vergleich, bedingt durch eine boole'sche Konstante oder unbedingt sein. Die Schachtelung von Wiederholungsblöcken ist *nicht* erlaubt.

5.18 Schleife

Ebenso wie der einfache Wiederholungsblock führt die Schleife den eingeschlossenen Block von Instruktionen mehrfach aus. Ein ganzzahliger Vektor bestimmt dabei den Startwert des Schleifenzählers (erste Komponente), die Anzahl der Wiederholungen (zweite Komponente) und die vorzeichenbehaftete Schrittweite (dritte Komponente), die nach jeder Wiederholung zum Schleifenzähler addiert wird. Innerhalb der Schleife steht der Schleifenzähler im Register `aL` zur Verfügung.

Ebenso wie beim Wiederholungsblock ist die Anzahl der Wiederholungen bei der Schleife derzeit auf 255 begrenzt. Ferner ist auch bei der Schleife ein vorzeitiges Verlassen durch eine spezielle Instruktion möglich. Das vorzeitige Verlassen kann bedingt durch einen Vergleich, bedingt durch eine boole'sche Konstante oder unbedingt sein.

5.19 Unterprogramme

In Shader-Programmen können außerdem Unterprogramme definiert werden. Jedes Unterprogramm wird durch ein Label begonnen und durch eine Rückkehr-Anweisung abgeschlossen. Dabei ist in einem Unterprogramm nur eine Rückkehr-Anweisung erlaubt. Zur vorzeitigen Rückkehr aus einem Unterprogramm müssen

[7] *übliche Vergleichsoperationen* sind: größer, größer gleich, kleiner, kleiner gleich, gleich, ungleich

die Instruktionen bis zur Rückkehr-Anweisung z.B. durch ein Konditional übersprungen werden. Die Deklaration des Unterprogramms muss *nach* dem letzten Aufruf erfolgen.

Unterprogramme haben keine Aufrufparameter oder Rückgabewerte, so dass der Datenaustausch zwischen Unterprogramm und Aufrufer über Register erfolgen muss, die sowohl Unterprogramms als auch Aufrufer bekannt sind. Unterprogramme können selbst auch andere Unterprogramm aufrufen, die Schachtelungstiefe ist jedoch begrenzt. Der Aufruf eines Unterprogramms kann bedingt durch einen Vergleich, bedingt durch eine boole'sche Konstante oder unbedingt erfolgen.

5.20 Abbruch eines Pixel Shaders

Beim Pixel Shader gibt es noch eine besondere Möglichkeit der Flußsteuerung. Mit einer speziellen Instruktion ist es möglich, die Bearbeitung des aktuellen Fragments abzubrechen. Das Shader-Programm wird dann für dieses Fragment nicht weiter ausgeführt und das entsprechende Fragment aus der Pipeline *gelöscht* und somit auch nicht mehr gezeichnet.

5.21 Predication

Eine besondere Form der Beeinflussung der Verhaltens von Instruktionen ist die sogenannte Predication. Auf Basis des Predication Registers, welches ein Vektor aus vier boole'schen Werten ist, kann bestimmt werden, welche Komponenten des Ergebnisses einer arithmetischen Instruktion oder Textur-Instruktion in das Zielregister geschrieben werden sollen. Die Predication funktioniert somit analog zum Masking (siehe Kapitel 5.7). Während jedoch beim Masking *statisch* festgelegt wird, welche Komponenten des Zielregisters tatsächlich geschrieben werden sollen, geschieht dies bei der Predication *dynamisch* anhand des Predication Registers.

Bei der Anwendung des Predication Registers auf eine Instruktion ist es erlaubt Swizzling (siehe Kapitel 5.7) und Negation (alle Komponenten werden logisch negiert) auf das Predication Register anzuwenden.

Gesetzt wird das Predication Register durch eine spezielle Vergleichs-Instruktion, die komponentenweise die Eingaberegister vergleicht und die Ergebnisse des Vergleichs im Predication Register ablegt. Das Ergebnis des Vergleichs der beiden x-Komponenten der Eingaberegister wird in der x-Komponente des Predication Registers abgelegt, das Ergebnis des Vergleich der beiden y-Komponenten in der y-Komponente usw.

Die einzelnen (boole'schen) Komponenten können ebenso wie die boole'schen Konstantenregister in Konditionalen usw. anstatt des Vergleiches als Bedingung verwendet werden.

6 Pipeline & Parallelität

6.1 Parallelität

Bei der Programmierung der Vertex Shader fällt besonders auf, dass jeder Vertex und jedes Fragment vollkommen *unabhängig* von allen anderen bearbeitet wird. Dies wird deutlich, wenn man betrachtet

- dass die temporären Register für jeden neuen Vertex für jedes neue Fragment neu initialisiert werden,
- dass es keine statischen Variablen gibt und
- dass es nicht möglich ist, Texturen zu ändern, von denen im aktuellen Durchlauf auch gelesen wird.

Die Bearbeitung der einzelnen Vertices und Fragmente unabhängig voneinander durchzuführen ist eine bewusste Design-Entscheidung bei der Entwicklung der Graphikkarten. Dadurch ist es theoretisch möglich, beliebig viele Vertices bzw. Fragmente gleichzeitig auszuführen, ohne dass die Bearbeitung eines Vertex bzw. Fragmentes auf die Fertigstellung der Bearbeitung eines anderen warten muss.

Um dies in der Praxis auszunutzen, verfügen die GPUs über mehrere unabhängige Einheiten, die Vertices verarbeiten können (im folgenden Vertex Unit genannt), und mehrere unabhängige Einheiten, die Fragmente verarbeiten können (im folgenden Fragment Unit genannt). Letztere sind in der Regel in sehr großer Anzahl vorhanden, da wesentlich mehr Fragmente als Vertices bearbeitet werden müssen.

6.2 Verbergen von Speicher-Latenzen

Vertex Units und Fragment Units werden so entworfen, dass sich eine bestimmte Anzahl von Units Chip-Resourcen teilen, und nicht jede dieser Units eine eigene „Kopie“ der entsprechenden Resource hat. Bedingt durch die reduzierten Kosten[8] pro Unit können wesentlich mehr Units in einem Chip untergebracht werden.

Die größere Anzahl von Vertex Units und Fragment Units bringt zunächst keinen großen Vorteil, da die Units, die sich eine Resource teilen, niemals gleichzeitig diese Resource in Anspruch nehmen können. Solange also eine Unit die geteilte Resource verwendet, müssen alle anderen Unit dieser Gruppe, die auch diese Resource verwenden wollen, warten.

In der Praxis warten viele Units jedoch häufig auf das Ende von Textur-Zugriffen. Während dieser Wartezeit können die anderen Units auf die geteilte Resource ohne Konflikte mit der wartenden Unit zugreifen. Dadurch werden die verfügbaren Resourcen besser ausgelastet und es steht zu gleichen Kosten mehr Rechenleistung zur Verfügung. Ein ähnliches Konzept wird von Intel mit der „Hyperthreading Technologie“ auch für Mehrzweck-Prozessoren realisiert.

[8] mit Kosten sind hier nicht nur die Herstellungskosten sondern auch zum Beispiel der Platzbedarf der Schaltkreise im Chip gemeint, die für die Realisierung der Funktionen im Chip erforderlich sind

6.3 Synchronität

Die einzelnen Einheiten arbeiten dabei synchron, d.h. alle Vertex Units beginnen gleichzeitig mit der Bearbeitung je eines Vertex und schließen die Bearbeitung auch gleichzeitig ab. Analog gilt dies auch für die Fragment Units. Dies hat den Vorteil, dass die Ergebnisse der jeweils vorherigen Pipeline-Stufe gleichzeitig als Eingabe entgegengenommen werden können und dass die Ergebnisse der jeweils aktuellen Pipeline-Stufe gleichzeitig an die nächste Pipeline-Stufe weitergegeben werden können.

6.4 Flußkontrolle & Parallelität

Daraus, dass alle Vertex Units und Fragment Units vollständig parallel arbeiten, ergeben sich in Verbindung mit dynamischer Flußkontrolle jedoch Probleme.

Dabei stellt sich nämlich die Frage, was passiert, wenn gleichzeitig Vertices bzw. Fragmente verarbeitet werden, die im Shader-Programm bedingt durch dynamische Flußkontrolle unterschiedliche Programmpfade durchlaufen. In der Regel wird dabei die Situation eintreten, dass einige Vertex Units bzw. Fragment Units weniger Instruktionen als die anderen ausführen müssen und somit eher fertig sind. Um die Synchronität zu erhalten müssten diese Units warten bis auch alle anderen parallel verarbeiteten Vertices bzw. Fragmente fertig bearbeitet sind. Dies führt insgesamt zu einer Performance-Einbuße, da Teile der Graphik-Hardware in dieser Zeit ungenutzt bleiben.

Ein besondere Dimension erhält dieses Programm im Rahmen von Wiederholungsblöcken und Schleifen, da durch eine häufige Wiederholung eines Blocks von Instruktionen sehr viel Rechenzeit in Anspruch genommen wird. Wird dann die Schleife zum Beispiel bedingt durch ein Konditional für einige Vertices bzw. Fragmente ausgeführt und für einige nicht, ergeben sich große Diskrepanzen bei den Rechenzeiten und somit eine sehr schlechte Auslastung der Graphik-Hardware.

Das Problem tritt im übrigen ebenso auf, wenn im Pixel Shader die Bearbeitung eines Fragmentes abgebrochen wird. Der Abbruch der Bearbeitung eines Fragmentes führt also im Allgemeinen nicht zu einer Leistungssteigerung.

Diese Probleme treten bei statischer Flußkontrolle nicht auf, da alle Entscheidungen hinsichtlich des Programmablaufs ausschließlich auf Basis von Konstanten getroffen werden, die sich nicht mit jedem Vertex bzw. Fragment ändern. Somit beschreiten alle gleichzeitig bearbeiteten Vertices bzw. Fragmente den selben Programmpfad im Shader-Programm.

6.5 Predication & Parallelität

Die Predication ist im Sinne der Parallelität gewissermaßen die „natürliche“ Flußkontrolle, da bei der Predication *alle* Befehle ausgeführt werden, aber *nicht alle* Ergebnisse tatsächlich geschrieben werden. Somit tritt das oben beschriebene Problem in Verbindung mit Predication nicht auf.

7 Ausblick

7.1 Erweiterung Pixel Shader

Beim Versionswechsel von Pixel Shader 2.0 auf Pixel Shader 3.0, wurden bereits die Nebel-Berechnungen zusätzlich mit in den Aufgabenbereich des Pixel Shaders verschoben.

Die nachfolgenden Pipeline-Stufen, die die verschiedenen Sichtbarkeitstest durchführen (alpha test, depth test, stencil test), ebenso wie die Kombination der für die Fragmente berechneten Farbwerte mit dem Inhalt des Frame Buffers (frame buffer blending) sind potentielle Kandidaten, um in einer kommenden Version der Pixel Shader ebenfalls programmierbar zu werden.

Um dies zu realisieren, müssten nur wenige neue Instruktionen eingeführt werden:

- Lesen und Schreiben des Depth Buffers an der dem Fragment entsprechenden Stelle,
- Lesen und Schreiben des Stencil Buffers an der dem Fragment entsprechenden Stelle und
- Lesen und Schreiben des Frame Buffers an der dem Fragment entsprechenden Stelle.

Mit einer Einführung der Programmierbarkeit dieser Pipeline-Stufen würden sich dem Anwender wiederum neue Möglichkeiten zur Realisierung von Effekten eröffnen.

7.2 Allgemeine Entwicklung

Selbstredend werden sich die technischen Daten wie Registeranzahl, maximale Anzahl von Befehlen usw. im Rahmen der Weiterentwicklung verbessern. Ferner bleibt abzuwarten, inwieweit die Graphikkarten in der Zukunft die Konflikte zwischen der Flußsteuerung und der starken Pipeline-Struktur lösen können.

Literaturverzeichnis

[1] Kurt Akeley. Reality engine graphics. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 109–116. ACM Press, 1993. URL: http://doi.acm.org/10.1145/166117.166131.

[2] Microsoft Corporation. *Microsoft DirectX 9.0 Reference.* Microsoft Corporation. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/shaders/shaders.asp.

Sven Buente
Advisor: Jörg Schmittler

Renderman and its Shading Language

Renderman is a Scene Description Language (SDL) for 3D imagery. Like Postscript describes the layout of documents, Renderman is supposed to describe 3D imagery and therefore 3D animations as well. The idea is great but not easy to instanciate because the discription of 3D images differs in a way that not only results like layouts are described but problems like the illumination process as well. Anyway, Renderman has made it to a great SDL and is therefore widely distributed today, especially in the motion picture industry. This document deals with the problems stated above and gives some examples to use Renderman.

1 Introduction

Because this paper mainly deals with the shading language provided by Renderman, the meaning of the word shading has to be clarified at first. Objects, appearing in computer generated images are defined on the one hand by their shape and on the other hand by their shading. The shape results from the object's geometry, whereas shading contains color, surface and material behavior, as well as illumination attributes. Renderman offers a language to program shading compuations which are then used by an extern rendering system. This seperation is a great approach and implicates a lot of advantages which will be discussed later in detail.

2 An Overview of Renderman

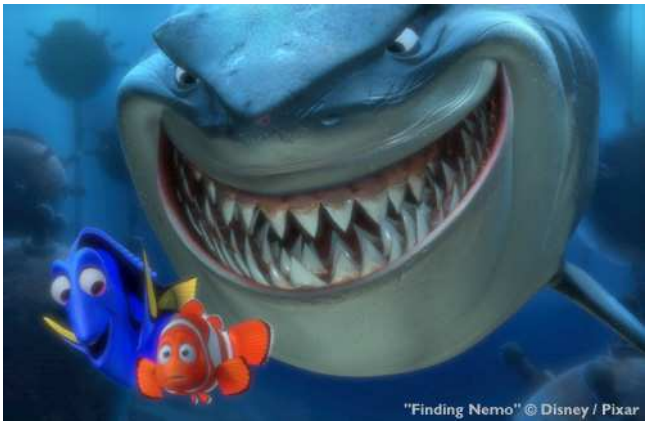
2.1 Motivation

In the mid-eighties, each rendering system had its own shading model. The problem was that they differed at least in some little ways. The result was that

the designers have been fixed to one rendering system. Then in 1987, the idea came up to develop an interface which provides shading relevant data from a seperated system to the rendering machine. Pixar was the first company who did so, but there was no feedback from other reserchers. The first specification of the Renderman standard remained at Pixar for internal use then. Later, in the early nineties, the interest in Renderman growed rapidely, since the power and potential of Renderman has been appreciated.

Today, Renderman is still a powerfull tool and used by many companies in computer graphics, especially for creating motion pictures with a high amount of photorealistic visual effects. At this point Pixar has to be singled out. They pushed the progress of Renderman enormously and still do great work with Renderman, which can be seen in motion pictures like "Finding Nemo" (Abbildung 1), "Monsters, Inc." and a lot more.

Figure 1 Scene from "Finding Nemo", Disney/Pixar



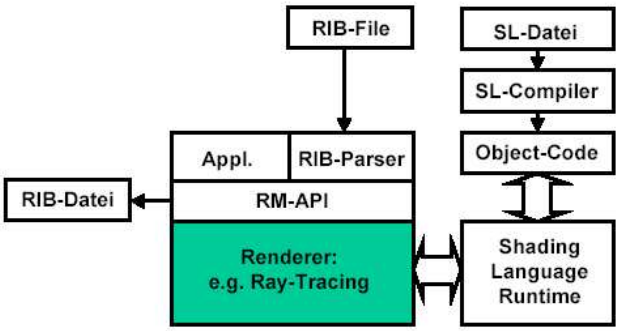
2.2 Structure of the Renderman Interface

As mentioned before, Renderman is more than just the language to describe shading calculations. It consists of three independent parts:

- The Renderman Interface Bytestream (RIB), a file format
- The Renderman Application Programming Interface (API)
- The Shading Language (SL)

A RIB file contains a huge amount of the scene description, realized by so called "requests". In addition to camera, image or light source parameters, the whole geometry of scene elements is defined here as well. The API is the part of Renderman which combines the SL- and RIB files and then allocates the optimized information data to an extern rendering system in a way it can be used dynamically and efficiently. The interaction of these components can be seen in Abbildung 2.

Figure 2 The Renderman Structure, source: [7]



The syntax of RIB requests is similar to shell commands:

<request> {options}* [name-value pairs]*

To link a Phong Shader for example, the phrase whould look like this:

Shader "phong" "Ka" 0.2 "Kd" 0.5 "Ks" 0.5 "Ke" 50

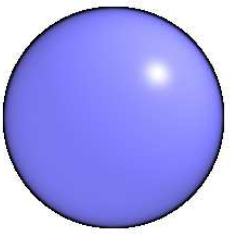
It is not important for now to understand the declaration, I just want to show an example of how a request is structured. Shaders are discussed later.

The RIB-requests can be classified into three parts:

- Options
  - Camera settings (e.g. shutter speed, kind of projection, position, etc.)
  - Image resolution
  - File names, etc.
- Attributes
  - Color and transperency
  - Light sources
  - Shader
  - Transformations
- Geometry
  - Polygons
  - Free form sufaces
  - Quadrics
  - Torus

Instead of describing all kinds of requests in detail, I think an example is sufficient to understand the structure and usage of a RIB file. More information about the interface is available here [6].

Figure 3 Plastic Sphere, rendered with BMRT [1]



The sphere in Abbildung 3 can be modeled like this:

```
Declare "Ke" "uniform float"
Display "simple.tif" "file" "rgba"
Format 400 400 1
FrameBegin 1

# Perspective projection, opening angle of 37°
Projection "perspective" "fov" 37

# Shifts the world to a visible region
Translate 0 0 3

WorldBegin
  LightSource "pointlight" 1 "intensity" 1.3
              "from" [1 1 -1] "lightcolor" [1 1 1]
  LightSource "ambientlight" 2 "intensity" 1
  Color [.5 .5 1]
  Surface "shineyplastic"
  Sphere .5 -.5 .5 360
WorldEnd
FrameEnd
```

One more thing to mention is that all transformations are based on a left handed coordinate system. Now it is time to work with the Renderman Shading Language, the most interesting and critical part of the Renderman standard.

3 The Renderman Language

Attaching Shaders to geometric primitives via the RIB was already covered in section 2.2. The next step would be to take a look at the role of shaders and how they can be described via a shading language. The Renderman Shading Language is based on the C-syntax and therefore easy to understand and to work with.

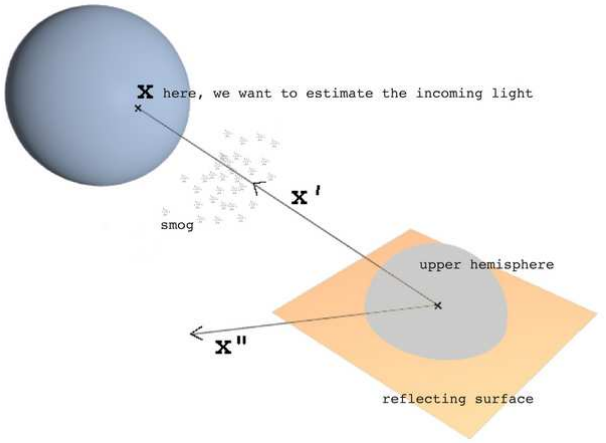
3.1 Model of the Shading Process

The first thing to know when programming shaders is how light is transported through the environment. You need to know that since every time you want to estimate the color at a certain point on a surface, the amount of light which brings it to that point is of high importance when calculating the resulting color. James T. Kajiya [5] pointed out that this process during rendering can be modeled like this:

$$i(x,x') = v(x,x') \cdot \left[l(x,x') + \int r(x,x',x'') \cdot i(x',x'') \cdot dx'' \right]$$

Of course, it is not easy to understand the interaction of the functions. Abbildung 4 might help to get a feeling of what is going on during shading.

Figure 4 Transport of light through the environment



$i(x,x')$ This function determines the intensity of light at x, coming from x'

$r(x,x',x'')$ This is the Bidirectional Reflectance Distribution Function (BRDF). Given a point x', an incoming light direction x'' and an outgoing direction from x' to x, the term gives us information about how much light is reflected from x'' to x. This term depends on the material of the reflected surface.

$i(x',x'')$ A recursive use of the light intensity distribution function, which provides the light intensity to the BRDF.

$\int r(x,x',x'') \cdot i(x',x'') \cdot dx''$ Here the BRDF and the light intensity distribution function are integrated over x'', which means that all incoming directions to x' on the surface are taken into account to estimate the correct radiance at x'. If the material of the surface is nontransparent, only the upper hemisphere over x' has to be considered.

$l(x, x')$ If there are light sources at x', they might emit light towards x.

$v(x, x')$ At last, light which travels from x' to x might be scattered or blocked by fog or some other volume which is placed in between. v gives the percentage of light that makes it to x.

As seen, various shaders are responsible during rendering to compute the different terms. There are three major types of shaders:

Light Source Shaders This shader determines the value of $l(x, x')$. That includes color as well as the intensity of light emitted from a light source in a dedicated direction. The shader does not care about whether the light is blocked by another surface in between, this part is managed by the renderer. Light Source Shaders might be linked to a geometric primitive, for example when it characterizes an area light source. Nevertheless, the shader itself makes no assuption about whether it is bounded or not.

Surface Reflectance Shaders The integral $\int r(x, x', x'') \cdot i(x', x'') \cdot dx''$ is calculated by this shader. All Surface Reflectance Shaders are bound to geometric primitives. In that kind of way, they determine all local surface properties which deal with color or light intensity. A surface might absorb, reflect or refract light, each in several ways. If an object has a red surface for example, only the red part of the incoming light is reflected (towards the observer's eye).

Volume or Atmosphere Shaders A Volume or Atmosphere Shader handles $v(x, x')$, namely scattering effects between x and x'.

The Renderman Shading Language is based on the abstraction shown above. Every kind of shaders are defined seperately and independent. The model makes no assumption about how the renderer calculates the light equation from James T. Kajiya. Therefore, all models defined by the Shading Language can easily be used by several rendering systems. This is a great approach, since in former times all descriptions of surfaces and light sources were bound to a specific rendering system.

3.2 Colors

There are different models for colors in Computer Graphics, since the physical basis of color, which is a continous function of wavelength, is not easy to implement and to work with. Therefore the function is sampled differently in various existing models. I will not discuss color models here, since that would fill up a prosemiar of its own. The shading language provides colors with different samples. The most favoured model consists of three samples and can be, for example, interpreted as an RGB-Color. One sample would describe a monochrome color space. Opacity can be modeled as a three component color, describing the opacity in each component. The shading language does not provide resampling, for example converting a color from a three dimensional color

space to a monochrome one. This has to be done by the renderer or another system on demand.

Operations, according to colors are based on two occurences which can happen in shading calculations. On the one hand interference of light, which is an additive light combination and on the other hand filtering, which occures for example when light interacts with some material. The operators in the language are "+" for additive correlation and "$*$" for a filtered interaction. These operations are independent of the number of samples.

3.3 Points

As colors, points are represented via vectors. Since a scene is defined normally in three dimensions, the shading language uses three component vectors. All standard operations are overloaded, so that it is simple to add or multiply points and directions respectively. The " . " operation defines the dot product, " ^ " the cross product.

3.4 Built-In Functions

A lot of useful functions which are often used during shading are added to the language. This has the adventage that one does not have to define the functions over and over again. The functions are implemented very efficiently, nevertheless it is possible for the user to modify them on demand. Here are some built-in functions:

Standard mathematical functions like $sin()$, $cos()$, etc.

$clamp(a, min, max)$

Maps a into the interval $[min, max]$, abiding the proportion

$step(min, v)$

Determines whether $v < min$ or not

$random()$

$length(v)$

$distance(p1, p2)$

$normalize(v)$

$faceforward(N, I)$

Returns N so that $N.I < 0$, which means that N might be inverted to be oriented to the upper hemisphere.

$reflect(V, N)$

Returnes the reflected vector V on N

$refract(V, N, index)$

Returnes the refracted vector V on N according to the refraction-$index$

$texture(texname, s, t)$

Returns the color of a texture specified by $texname$ at position (s,t)

$bump(texname, s, t)$

Returns the perturbation of the normal at (s, t) provided by the height-map $texname$

$environment(texname, p)$

Returns floats or colors as a function of a direction p in space to the environment map $texname$

3.5 Writing Surface Shaders

The shading language alows us to model very complex and detailed surfaces. When a shader is instantiated, a lot of attributes might get passed to the shader via arguments, like for example "Kd" which gives us the amount of diffuse reflected light. As seen in section 2.2 there are no limitations in the number of arguments. But this information is not sufficient to determine a satisfying result for the integral containing the BRDF. The rendering system is responsible to provide additional attributes which are essential for good shading. For a simple diffuse reflection the point where the surface's color should be estimated, a normal to reflect on and information about incoming light have to be known. For specular reflections one might need to know a direction to the vantage point. Bump mapping requires information about the surface derivatives in addition. Everytime a shader is executed the variables may change because they depend on the state the rendering process is situated in. Abbildung 5 describes an example of such a state.

Figure 5 A surface shader state, source: [7]

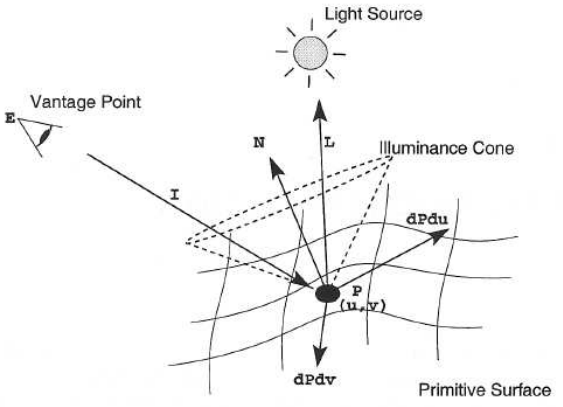
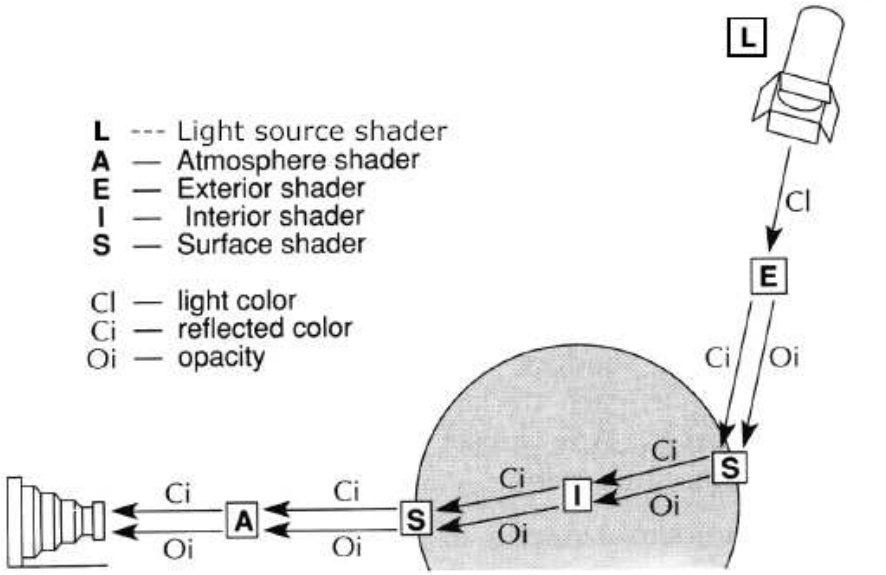


Figure 6 Shader execution, source: [7]



Now, the question arises how to use the variables since they are not passed to the shader as arguments. The answer is, they can simply be treated like global variables. Here are the most popular ones for surface shaders.

- L - Vector to light source
- I - Viewing direction (at surface)
- N, P - Normal and point
- Cs, Os - Color and opacity
- u, v - Surface coordinates in [0,1]x[0,1]
- $time$
- Ci, Oi - Reflected color and opacity

Till now only one light source was considered. Dealing with several light sources requires an additional construct. The surface shaders' task is to compute the integral containing the BRDF as realistic as possible, however an analytical solution is not feasible. That is why the solution is often approximated by only taking directions to light sources into account. There are methods to estimate even indirect light but it is a bit more complicated. To approximate the integral, by iterating a shader operation over all light sources, Renderman offers the $illuminance()$ statement. The construct supported by renderman is an

illuminance() statement. Everything within this procedure is called individually for each light source which acts in a specified region. Thus, it is an iteration over all light sources. The construct can be used like this:

illuminance(*point* P, *point* N, *float angle*),

where P, N and *angle* define a cone. Only light sources from these directions are taken into account. To consider all light sources the statement looks this way:

illuminance(*point* P)

There can be multiple uses of *illuminace*-constructs within one shader deklaration.

Since we got now all information to write simple shaders, it is time to take a look at an example. The resulting surface can be seen in Abbildung 7. For detailed information in writing complex shaders, [2, 3, 8] are good references.

```
/* Ks: Amount of specular reflected light */
/* Kd: Amount of diffuse reflected light */
/* Ka: Amount of ambient light */
/* roughness: Size of dents */
/* dent: intensity of dents */

surface
dent( float Ks=.4, Kd=.05, Ka=.1, roughness=.25, dent=.2 )
{
  float turbulence;
  point Nf, V;
  float i, freq;

  /* Transform to solid texture coordinate system */
  V = transform("shader",P);


  /* Forms turbulence, to displace the normal later on */
  turbulence = 0; freq = 1.0;
  for(i=0; i<6; i+=1)
  {
    turbulence += 1/freq * abs( 0.5 - noise(4*freq*V));
    freq *= 2;
  }

  /* To intensify turbulence
  turbulence *= turbulence * turbulence;

  /* For adaption to the dent argument */
  turbulence *= dent;

  /* Displace Surface and compute new normal */
  P -= turbulence * normalize(N);
```

```
  Nf = faceforward( normalize(calculatenormal(P)),I);
  V = normalize(-I);

  /* Perform shading calculation */
  Oi = 1 - smoothstep( 0.03, 0.05, turbulence);
  Ci = Oi * Cs * (Ka*ambient()
       + Ks*specular(Nf,V,roughness))
       + Kd*diffuse(Nf);
}
```

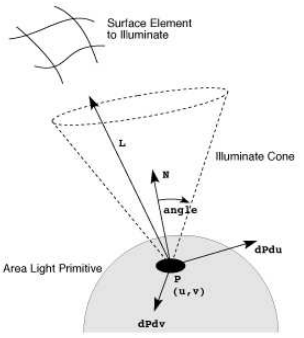
Figure 7 A corroded teapot, rendered with BMRT [1]

3.6 Writing Light Shaders

A general light source shader can be modeled as a fuction of position and direction which determines the emitted light according to the arguments (see section 3.1). Here are several classes of light source shaders:

- *Ambient light source.* Ambient light is independent of directions, but may vary as a function of position.
- *Point light source.* Light is distributed to all directions equally.
- *Spot light source.* Like a point light source, but the intensity of light is maximum at a specified direction and falls off in other directions.
- *Shadowed light source.* Like a point light source, but the intensity varies according to a texture or shadow map.
- *Distant light source.* The light source is not bound to a finite position, the light rays travel parallel. Sunlight for example, could be modeled like this.

Figure 8 A light source shader state, source [7]



As seen in section 3.5, each state offers several variables. Abbildung 8 shows some specific ones for light source shaders. The following example is an implementation of a simple light source.

```
/* A simple declaration of a point light source */

light
pointlight (float intensity = 1;
            color lightcolor = 1;
            /* Sets from to the point where the shader is executed */
            point from = point "shader" (0,0,0);)
{
    illuminate (from) {
        /* neglect directions, intensity falls of with distance */
        Cl = intensity * lightcolor / (L . L);
    }
}
```

4 Conclusion

The Renderman stardard provides a SDL which is well structured, easy to use and very powerfull. As seen in section 3.5 a nice looking suface can be modeled with a few lines of code. Once a shader is defined it can be reused on every rendering system again, in case the system supports the Renderman interface. Unfortunately, it seems that there are not that much enterprises that use renderers together with Renderman professionally. However, there are several free distributions of renderers supporting Renderman. I used BMRT [4], which is free but not available anymore. In my opinion it is very powerfull and everyone who got motivated by Renderman should try to play a bit with this tool.

Bibliography

[1] Blue Moon Rendering Tools, not available anymore. `http://www.bmrt.org/`.

[2] Apodaca and Gritz. Advanced RenderMan: Creating CGI for Motion Pictures. *Morgan Kaufmann*, 1999.

[3] Apodaca and Peachey. Writing RenderMan Shaders. *SIGGRAPH*, 1992.

[4] Pat Hanrahan and Jim Lawson. A Language for Shading and Lighting. *SIGGRAPH*, 1990.

[5] James T. Kajiya. The Rendering Equation. *Computer Graphics 20(4)*.

[6] Pixar. The RenderMan Interface v3.2. 2000.

[7] Hans Peter Seidel and Karol Myszkowski. Computer Graphics - RenderMan and the Shading Language. *Lecture at the "Universität des Saarlandes"*.

[8] Steve Upstill. The RenderMan Companion. *SIGGRAPH*, 1990.

Holger Abt
Betreuer: Nicolas Fritz

Superword Level Parallelism

In dieser Seminararbeit wird Superword Level Parallelismus [4], eine Möglichkeit, Parallelismus in Code zu finden und in Superwords auszuführen, vorgestellt. SLP bietet dabei eine robuste und einfache Möglichkeit, Code für normale Desktopprozessoren zu beschleunigen. Dabei benutzt SLP die von den Hardwareherstellern eingeführten Multimedia–Erweiterungen.

1 Einführung

In dieser Seminararbeit werden die Konzepte hinter Superword Level Parallelismus [4] (SLP) erläutert. SLP ist eine Compileroptimierung, deren Ziel die Beschleunigung von multimedialen Anwendungen ist. Eine Besonderheit von SLP ist, dass SLP Programme für normale Desktop–Prozessoren optimiert und nicht für hoch spezialisierte Graphikchips. Viele Multimediaprogramme wie etwa die Videokodierung und -dekodierung umfassen Berechnungen, die vom höheren Parallelitätsgrad des von SLP erzeugten Codes profitieren. Dieser wird durch den Einsatz von Multimedia–Erweiterungen, wie Intels SSE, 3DNow! von AMD oder AltiVec [3] von Motorola, erzielt.

Die Verwendung solcher herstellerspezifischen Instruktionssätze bedeutet üblicherweise einen enormen Aufwand für den Programmierer, wenn er sein Programm für verschiedene Prozessoren optimieren will.

Um dieser Vielfalt von Erweiterungen zu entgehen, kam die Idee auf, den Code schon beim Kompilieren so zu optimieren, dass er auf all diesen Plattformen schnell abläuft. Dies ist möglich, da alle diese Multimedia Erweiterungen gewisse Ähnlichkeiten aufweisen. Sie arbeiten alle mit Superwords (siehe Kapitel 1.3), ermöglichen die Benutzung von SIMD Operatoren (siehe Kapitel 1.2) und die meisten unterstützen auch Floating–Point Operationen. Einhergehend mit der Einführung von Superwords wurde die Datenpfadbreite auf die Größe der Superwords erhöht. Bereits vor der Einführung der Multimedia Erweiterungen gab es Ansätze zur Optimierung von Vektoroperationen. Nach der Einführung von Multimedia-Erweiterungen wurde versucht die Optimierung von Programmen zu automatisieren. Ein Ansatz war der Einsatz von Vektorcompiler, die von

Abbildung 1 Multimedia Erweiterung

Instruction Sets	Architektur	SIMD Weite	Floating Point
AltiVec	PowerPC	128	ja
MMX/SSE/SSE2	Intel	64/128	ja
3DNow!/SSE2	AMD	64/128	ja
VIS	Sun	64	nein
MAX2	HP	64	nein
MVI	Alpha	64	nein
MDMX	MIPS V	64	ja

den Vektor Supercomputern stammen. Diese Compiler haben aber den Nachteil, dass sie sehr komplex und komplieziert sind. SLP hingegen benutzt einfache und robuste Methoden zur Code-Optimierung.

1.1 Multimedia Erweiterungen

Der Hauptanwendungsbereich von Computern verändert sich dahingehend, dass mehr und mehr Multimedia-Informationen verarbeitet werden müssen. Unter Multimedia-Informationen versteht man dabei z.B. Bilder, Videos, Musik, 2D- und 3DGrafiken und Animationen. Multimedia Erweiterungen wurden von den Prozessor–Herstellern entwickelt, um multimediale Inhalte zu beschleunigen. Bei der Analyse der typischen Anwendungen trat die Relevanz von SIMD Instruktionen (Kapitel 1.2), Superword Register(Kapitel 1.3) und einer schnellen Gleitkommazahlenberechnung zu Tage. Die drei Hersteller für den Massenmarkt AMD (K6.2), Intel (Pentium 3) und Motorola (PowerPC) führten SIMD Wörter, Superword Register und eine schnelleren Gleitkommazahlenberechnung ein.

1.2 Single Instruction Multiple Data (SIMD)

SIMD Technologie arbeitet unter Benutzung des Konzeptes von Datenparallelismus. Datenparallelismus tritt auf, wenn dieselbe Instruktion auf eine Menge von Daten angewendet wird. Durch Zusammenfassung der Daten und Einführung eines SIMD Operators wird eine parallele Abarbeitung ermöglicht. Ein SIMD Operator ist in der Funktion äquivalent zum normalen Operator, er wird nur auf n Datenwörter angewendet. Ein Beispiel für eine solche Konvertierung ist in Abbildung 4 dargestellt.

SIMD Technologie findet zum Beispiel in MPEG–Videokodierung, die auf Matrizenberechnungen beruht, ihre Verwendung. Wenn zwei Matrizen addiert werden, haben alle Berechnungen die gleiche Form nämlich $c_{x,y} = a_{x,y} + b_{x,y}$ und lassen sich parallel berechnen. Abbildung 2 zeigt diese parallele Verarbeitung von Daten, wenn vier Instruktionen in einem Schritt bearbeitet werden können. Eine spürbare Geschwindigkeitssteigerung in 3D Spielen und Multimedia Applikationen ist dabei offensichtlich.

Abbildung 2 SIMD Instruktionsfluss

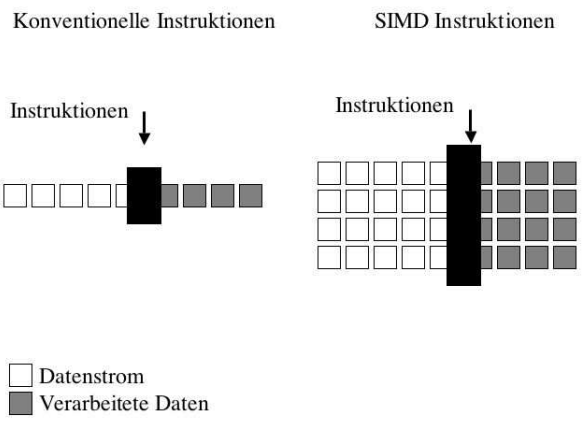
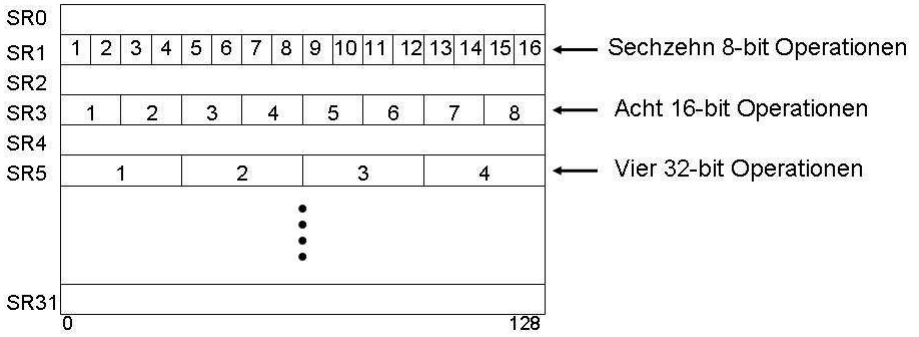


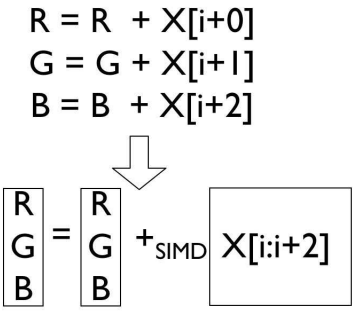
Abbildung 3 Superword Register in der AltiVec Erweiterung



1.3 Superwords

Superwords [6] ist eine Erweiterung die Prozessorhersteller eingeführt haben, um Anweisungen schneller abarbeiten zu können. Superwords sind spezielle Register in denen sich mehr Daten als in normalen Registern speichern lassen. Im Fall der AltiVec Erweiterung von Motorola sind dies 32 Register mit 128–Bit Breite, bei Intel acht 128–Bit Register. Es können also sechzehn 8–Bit, acht 16–Bit oder vier 32–Bit Anweisungen in ein Register geschrieben werden (siehe Abbildung 3). Der Vorteil dieser Register liegt in der Reduzierung der Lade- und Speicherzugriffe, da nur noch pro Block und nicht wie vorher pro Wert ein Zugriff nötig ist.

Abbildung 4 Isomorphe Anweisungen die gepackt werden und parallel ausgeführt werden können



2 Superword Level Parallelismus (SLP)

In diesem Kapitel werden die Techniken die hinter SLP stehen beschrieben. Dabei werden die wichtigen Termini dieser Thematik eingeführt. Das nächste Kapitel liefert eine Beschreibung der Umsetzung der Techniken in Verarbeitungsschritte.

2.1 Beschreibung von Superword Level Parallelismus

Superword Level Parallelismus kann man als kurze SIMD Parallelismen beschreiben, wobei die Operanden und das Resultat einer SIMD Berechnung in demselben Paket gespeichert werden. Die Suche nach geeigneten Instruktionen geschieht mit Hilfe einer einfachen Analyse, in der isomorphe Operationen im selben Basisblock gesucht werden. Unter isomorphen Operationen versteht man Operationen, die dieselben Operanden in der gleichen Reihenfolge benötigen. Solche Operationen können, falls keine Datenabhängigkeiten bestehen, parallel ausgeführt werden. Ein Beispiel für eine solche Abhängigkeit sind die Ausdrücke $b = a+1$ und $a = c+1$. Die hierzu verwendete Technik heißt Statement packing. In Abbildung 8 wird dieser Prozess beschrieben.

Offensichtlich haben alle Operationen die gleiche Struktur und es bestehen keine Nebeneffekte. Aus diesem Grund kann man diese 4 Operationen in Blöcke zusammenfassen. Diese Operationen können parallel ausgeführt werden, ohne das eine Beeinflussung auftritt. Die Operatoren werden durch ihre äquivalenten SIMD Operatoren ersetzt.

Ob die Operationen in dieser Blöcken verbleiben, hängt davon ab, ob die Ergebnisse zu einem späteren Zeitpunkt gebraucht werden. Die Geschwindigkeitssteigerung, die sich aus solchen Operationen ergibt, berechnet sich aus dem Gewinn an Geschwindigkeit durch die Parallelisierung abzüglich der Kosten für das Entpacken und Verpacken der Anweisungen.

Ein weiterer Faktor für die Performanz ist die Hardware. Wenn z.B. das Packen und Entpacken eine relativ teure Operation in Bezug zu ALU (Arithmetik

Logik Unit)–Operationen darstellt, kann dies die Performanz entscheidend reduzieren. Deshalb wurde der SLP Detektionsalgorithmus so entworfen, dass er die Packoperationen minimiert. Man sucht Fälle, in denen die Ergebnisse von Berechnungen gepackter Daten in die Berechnungen anderer Daten eingehen.

In dem Fall, dass gepackte Anweisungen mit angrenzenden Speicherreferenzen existieren und zusätzlich Datenabhängigkeiten zwischen den Operationen bestehen, ist SLP besonders gut geeignet, da es die Operationen in aufeinander folgende Speicherplätze legt. Da sich die Ergebnisse der Berechnung dann noch in den Registern des Prozessors befinden. Dadurch wird das Neuladen von Ergebnissen vorhergehender Berechnungen aus dem Speicher vermieden. Es muss nur noch einmal für einen Block eine Speicheradresse berechnet und ein Lade- und Speichervorgang durchführt werden und nicht wie sonst pro Anweisung. In den Experimenten, die von den Entwicklern des Compilers durchgeführt wurden, hatte die Anordnung der Anweisungen im Speicher den größten Einfluss auf die Steigerung der Performanz. Diese Steigerung war zu erwarten, da dieses Verfahren bei Vektorisierung ebenfalls zum Einsatz kommt.

2.2 Vektor Parallelismus

Um den Unterschied zwischen Superword Level Parallelismus und Vektor Parallelismus besser erklären zu können, wird an einem Beispiel erklärt wie Parallelismen erkannt wird. Das Beispiel (Abbildung 5) lässt sich leicht parallelisieren und dient zur Verdeutlichung des Vorgangs. Die Verfahren, die bei der Vektorisierung zum Einsatz kommen sind komplex und kompliziert und liefern deshalb nicht immer ein Ergebnis. Der SLP Algorithmus hingegen ist einfacher und robuster und liefert auch bei komplexen Problemen ein Ergebnis.

Das Beispiel in Abbildung 5(a) stellt die innere Schleife des Motion Estimation Algorithmus, dar welcher in MPEG–Komprimierung verwendet wird. Vektorisierung wird durch das Auftreten von Abhängigkeiten in der Schleife erschwert. Um das zu Vermeiden, verwendet der Vektor–Compiler eine Reihe von Transformationen, um die Schleife in eine vektorisierbare Form zu bringen. Die erste Transformation ist „Scalar Expansion“, welche bei jedem Schleifendurchlauf neue generierte Elemente, in ein temporäres Array schreibt [1]. Die Schleife wird dann mittels „Loop fission“ in zwei unabhängige Schleifen unterteilt. Das Ergebnis dieser Transformation ist in Abbildung 5(b) zu sehen. Diese Schleife ist vektorisierbar, aber die zweite Schleife muss sequenziell mit der ersten ausgeführt werden.

In Abbildung 5(c) wird die Schleife nach der Vektorisierung mittels SLP gezeigt. Nachdem die Schleife aufgerollt wurde, werden die vier korrespondierenden Anweisungspaare umgeordnet. Dieser Packprozess gruppiert die Anweisung in einer Reihenfolge, die eine Ausführung in SIMD Blöcken ermöglicht. Diese Umordnung ist nach einer Umbenennung der Variablen gültig weil keine Abhängigkeiten mehr bestehen. Das Ergebnis ist in Abbildung 5(d) zu sehen.

Abbildung 5 Vergleich zwischen SLP und Vektor–Parallelisations–Techniken

```
for (i=0;i<16;i++){
    localdiff = ref[i] - cur[i];
    diff += abs(localdiff);
}
```

(a) Original Schleife

```
for (i=0;i<16;i++){
    T[i] = ref[i] - cur[i];
}
for (i=0;i<16;i++){
    diff += abs(T[i] );
}
```

(b)Schleife nach skalarer Expansion und Schleifen Aufteilung

```
for (i=0;i<16;i+=4){
    localdiff0 = ref[i + 0] - cur[i + 0];
    diff += abs(localdiff0);
    localdiff1 = ref[i + 1] - cur[i + 1];
    diff += abs(localdiff1);
    localdiff2 = ref[i + 2] - cur[i + 2];
    diff += abs(localdiff2);
    localdiff3 = ref[i + 3] - cur[i + 3];
    diff += abs(localdiff3);
}
```

(c) Schleife nach dem Abrollen durch SLP

```
for (i=0;i<16;i+=4){
    localdiff0 = ref[i + 0] - cur[i + 0];
    localdiff1 = ref[i + 1] - cur[i + 1];
    localdiff2 = ref[i + 2] - cur[i + 2];
    localdiff3 = ref[i + 3] - cur[i + 3];

    diff += abs(localdiff0);
    diff += abs(localdiff1);
    diff += abs(localdiff2);
    diff += abs(localdiff3);
}
```

(d) Die Anweisungen werden gruppiert und zusammen gepackt

Abbildung 6 Eine einfache Schleife vor und nach dem Ausrollen

```
for(i=0;i<8;i++){
   a[i]= b[i] + c[i]
}

for(i=0;i<8;i+=4){
   a[i + 0] = b[i + 0] + c[i + 0]
   a[i + 1] = b[i + 1] + c[i + 1]
   a[i + 2] = b[i + 2] + c[i + 2]
   a[i + 3] = b[i + 3] + c[i + 3]
}
```

3 Der SLP Algorithmus

Dieses Kapitel beschreibt den SLP Algorithmus. Der Algorithmus kann in sechs Phasen unterteilt werden: *Schleifen Abrollen* (Loop Unrolling), *Finden angrenzender Speicherreferenzen*, *Erweitern der Paketliste*, *Kombination*, *Datenabgleich* und *Zyklus Identifizieren*. In *Schleifen Abrollen* werden die Schleifen für die Extraktion von Parallelität vorbereitet. Der restliche Code wird in *Finde angrenzende Speicherreferenzen* an das SLP Schema angepasst. Die gefundenen Anweisungen werden in *Erweitern der Paketliste* in Basisblöcke zusammengefasst. In *Kombination* werden diese Basisblöcke soweit wie möglich kombiniert. *Datenabgleich* passt die Blöcke an die zugrunde liegende Hardware an und *Zyklus Identifizieren* stellt sicher, dass die Anweisungen parallel ausgeführt werden können.

3.1 Schleifen Abrollen

In SLP hat Schleifen Abrollen die Aufgabe vektorisierbare Schleifen in Basisblöcke mit parallelen SIMD Anweisungen umzuwandeln.

Abbildung 6 oben zeigt eine einfache vektorisierbare Schleife. Die aufgerollte Version zeigt Abbildung 6 unten. Sie zeigt vier isomorphe (Definition 5) Anweisungen mit aufeinander folgenden Speicherzugriffen. Die Operatoren dieser Anweisungen können durch SIMD Operatoren ersetzt werden, was eine parallele Ausführung ermöglicht.

3.2 Finden angrenzender Speicherreferenzen

Im Allgemeinen sind Anweisungen, die Referenzen auf angrenzende Speicheradressen haben, Kandidaten für das Packen. In diesem Schritt werden Variablen gesucht, die an einer anderen Stelle im Programm weiterverwendet werden. Dies können zum Beispiel Ergebnisse einer Berechnung sein. Es kann zu Problemen führen, wenn Abhängigkeiten im Programm bestehen. Dies kann so weit gehen,

dass sie eine Parallelisierung unmöglich machen oder die Kosten für das Packen in Pakete höher sind als die Beschleunigung.

In Experimenten hat sich ergeben, dass die Ergebnisse die durch das Neusortieren des Codes entstehen das größte Potential auf Beschleunigung des Programms haben. Bei der Analyse werden im ersten Schritt diese Anweisungen gesucht. Dann werden diese Anweisungen kombiniert, um eine effektive Verarbeitung zu ermöglichen. Im nächsten Schritt werden sie dann in eine Paketliste geschrieben.

Definition 1. Ein Paket $= \langle a_1, \ldots, a_n \rangle$ ist eine Folge von unabhängigen isomorphen Anweisungen $a_1, \ldots, a_n$ in einem Basic Blocks.

Definition 2. Eine PaketListe $= \{p_1, \ldots, p_n\}$ ist eine List von Paketen $p_1, \ldots, p_n \in Paket$

In dieser Phase des Algorithmus werden nur Paare von den Anweisungen erstellt. Diese Paare werden in darauf folgenden Phasen noch zusammengefasst. Später wird immer Bezug auf diese Paare (rechtes und linkes Element) genommen.

Definition 3. Paar $= \langle a_{links}, a_{rechts} \rangle$ wo a_{links}, a_{rechts} unabhängige isomorphe Anweisungen in Basic Blocks sind.

Definition 4. Unabhängig bedeutet in diesem Fall, dass keine Datenabhängigkeiten in den Ausdrücken bestehen dürfen.

Definition 5. Zwei Ausdrücke sind isomorph, wenn sie den gleichen Aufbau haben. Das heißt, beide Anweisungen haben die gleichen Operationen und die Operanden kann man durch einfache Umbenennung ineinander überführen.

Es muss aber sichergestellt sein, dass jede Anweisung aus dem Originalprogramm höchstens einmal in der linken und einmal in der rechten Position der Paketliste auftritt. Diese Regel stellt sicher, dass später diese Anweisung höchstens einmal ausgeführt wird.

Ein Beispiel für eine solche Umformung ist im Schritt zwischen Abbildung 8a und Abbildung 8b zu sehen. In dem Kasten P sind Pakete gebildet worden. In dem ersten Paket sind Ausdrücke, in denen die Variablen a und b kombiniert sind. Im zweiten Paket werden die Variablen b und c kombiniert. Wie vorher beschrieben, kommt keine Variable zweimal auf der linken oder zweimal auf der rechten Seite des Ausdrucks vor. In Abbildung 8b ist außerdem noch die Wunschliste zu erkennen. Diese wird im nächsten Abschnitt beschrieben.

3.3 Erweitern der Paketliste

Nachdem die Paketliste mit den initialen Werten gefüllt wurde, können weitere Gruppen zur Paketliste hinzufügt werden. Dies macht der Algorithmus, indem er die anderen Anweisungen daraufhin analysiert, ob es profitabel ist, sie mit der Paketliste zu vereinigen. Mit dem Finden einer solchen Anweisung, wird ein neuer Eintrag in der Paketliste erzeugt.

Definition 6. Ein Wunsch $= \langle\langle a_1, a_2\rangle, n\rangle$ ist ein Tuple zweier a_1, a_2 unabhängige isomorphe Anweisungen in einen *BasicBlock* und n ist die Anzahl der Gruppen die $\langle a_1, a_2\rangle$ als Operanden liefern.

Definition 7. Eine WunschListe $= \{w_1, \ldots w_n\}$ ist eine Liste von Wünschen mit $w_i \in$ Wunsch.

In dem Beispiel (Abbildung 8b) werden das Paar q und r und das Paar r und s zusammengelegt. Im Schritt von c nach d wird dieser Vorgang noch deutlicher. Die Variablen q und r aus der Wunschliste, werden in Anweisungen 2 und 5 verwendet. Aus diesem Grund kann man sie profitabel in ein neues Paar zusammenfassen. Das liegt daran, dass die Resultate aus dem Paket mit a und b in den Anweisungen 2 und 5 gebraucht werden.

Natürlich erschöpft das Beispiel nicht alle Möglichkeiten. Es kann zum Beispiel der Fall auftreten, dass mehrere Kombinationen möglich sind. Dies ist der Fall wenn ein Operand an mehr als einer Stelle verwendet wird. In dieser Situation übernimmt man die, die am profitabelsten sind. Danach wird dann jede andere Gruppe auf ihre Profitabilität untersucht. Dieser Prozess läuft solange ab, bis keine weitere Variable mehr in die Liste eingetragen werden kann.

Des Weiteren wird noch eine Punkteliste eingeführt. In dieser Punkteliste werden die Einsparungen, die sich durch eine Aktion ergeben eingetragen. Ein Punkt kommt in diese Punkteliste, wenn sich aus einem Eintrag in der Paketliste ein anderer ergibt. Diese Punkteliste befindet sich in Abbildung 8 in der Wunschliste. Der Index in der oberen rechten Ecke gibt an, an wie vielen Stellen dieser Operand gebraucht wird.

Variablen aus der Wunschliste, die eine gewisse Anzahl von Punkten erreichen, werden in die Paketliste übernommen. In Abbildung 8 geschieht dies im Schritt d nach e. In unserem Beispiel reichen 2 Punkte.

3.4 Kombination

Nachdem alle profitablen Paare gewählt wurden, kann man sie in größere Gruppen kombinieren. Eine Kombination von Paaren ist möglich, wenn die rechte Anweisung des einen Paares die linke Anweisung des anderen Paares ist. Im Allgemeinen kann man zwei Gruppen unterschiedlicher Größe kombinieren, solange sie dieselbe Anweisung in den Ecken beinhalten. Die Gruppen müssen sogar kombiniert werden um ein Mehrfachauftreten von Anweisungen in verschiedenen Gruppen zu verhindern. In dieser Phase wird jede Gruppe mit jeder anderen verglichen. Dabei wird Überprüft ob Anweisungen mehrfach in den Ecken auftreten. Das wird solange gemacht, bis kein Mehrfachauftreten mehr vorhanden ist. In Abbildung 8e sieht man das Ergebnis dieses Schrittes.

Es ist möglich, dass zwei Gruppen dieselbe Anweisung an anderer Position als den Ecken enthalten. In diesem Fall ist eine sinnvolle Kombination schwierig oder sogar unmöglich. Tritt dieser Fall auf, wird die Gruppe, die am wenigsten profitabel ist aufgelöst. In der Praxis soll dieser Fall sehr selten auftreten und hat nur wenig Einfluss auf die Performanz.

3.5 Datenabgleich

Mit dem Abschluss der Kombinationsphase wurde eine Liste mit SLP Anweisungen, die profitabel gepackt werden können, konstruiert. Nur kann es vorkommen, dass die Bits die zur Repräsentation der Gruppen von Anweisungen benötigt werden, die Superword–Größe der zugrunde liegenden Architektur überschreitet. Als Folge müssen große Gruppen so aufgeteilt werden, dass ihre Größe der Kapazität der Architektur entspricht. Es wurde der einfachste Ansatz, des Abschneidens an der höchstmöglichen Bitgrenze gewählt. Diese Bitgrenze ist die größtmögliche Anzahl der Anweisungen deren Größe kleiner oder gleich der Größe der Datenpfade (siehe Kapitel 1.1) ist. Dieser Vorgang wird solange wiederholt bis alle Gruppen in einer verarbeitbaren Größe sind. In dem Beispiel in Abbildung 8f wurde angenommen, dass ein Packen von höchstens zwei Anweisungen in ein Superword möglich ist.

Dem aufmerksamen Leser mag aufgefallen sein, dass die Möglichkeit besteht, Gruppen so aufzuspalten, dass sie nur noch suboptimal verarbeitet werden. Es kann sogar vorkommen, dass die Daten so ineffektiv gepackt werden, dass das Programm langsamer abläuft als vor der Optimierung. In den Testreihen der Entwickler ist ein solcher Fall nicht aufgetreten, aber ausschließen lässt er sich nicht. Aus diesem Grund wird an diesem Punkt weiter geforscht.

3.6 Zyklus Identifizieren

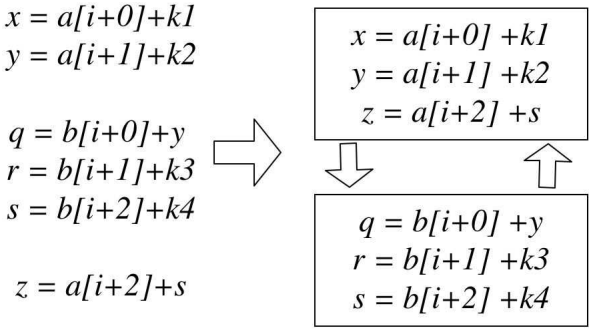
Abhängigkeitsanalyse vor dem Packen stellt die semantische Equivalenz zum Quellprogramm sicher. Es mußgarantiert werden, dass es keine Datenabhängigkeiten innerhalb einer Gruppe gibt. Dennoch kann der Fall auftreten, dass das Abarbeiten von zwei Gruppen zu einer Abhängigkeitsverletzung führt. Ein Beispiel für diesen Fall ist in Abbildung 7 zu sehen. Hier wurden Abhängigkeitskanten zwischen zwei Gruppen eingezeichnet. Diese Kanten zeigen an, dass Anweisungen in der einen Gruppe von den Anweisungen in der anderen Gruppe abhängig sind. Solange sich kein Zyklus in diesem Abhängigkeits-Graphen befindet, können die Gruppen so sortiert werden, dass keine Fehler auftreten. Ein Zyklus in den Gruppen ist ein Indikator dafür, dass die gewählten Gruppen ungültig sind.

Um die Korrektheit sicherzustellen, eliminiert der SLP Algorithmus alle Gruppen die zu einem Zyklus gehören. Um einen Zyklus zu finden, wird der Tarjan Algorithmus für „Strongly Connected Components“ verwendet [7].

4 Ergebnisse

SLP wurde auf zwei Testsets getestet, dem SPEC95fp Benchmark Set [2] und UTDSP Kernels [5]. Das SPEC95fp Benchmark Set ist eine Zusammenstellung der Standard Performance Evaluation Corporation von Programmen aus den unterschiedlichsten Bereichen um eine Vergleichbarkeit von verschiedenen Compilern auf verschiedenen Computern zu ermöglichen. Die UTDSP Benchmark Suite wurde 1992 zusammengestellt, um die Qualität der Code Generierung von high-level Sprachen (wie C) vergleichen zu können.

Abbildung 7 Beispiel für Datenabhängigkeiten



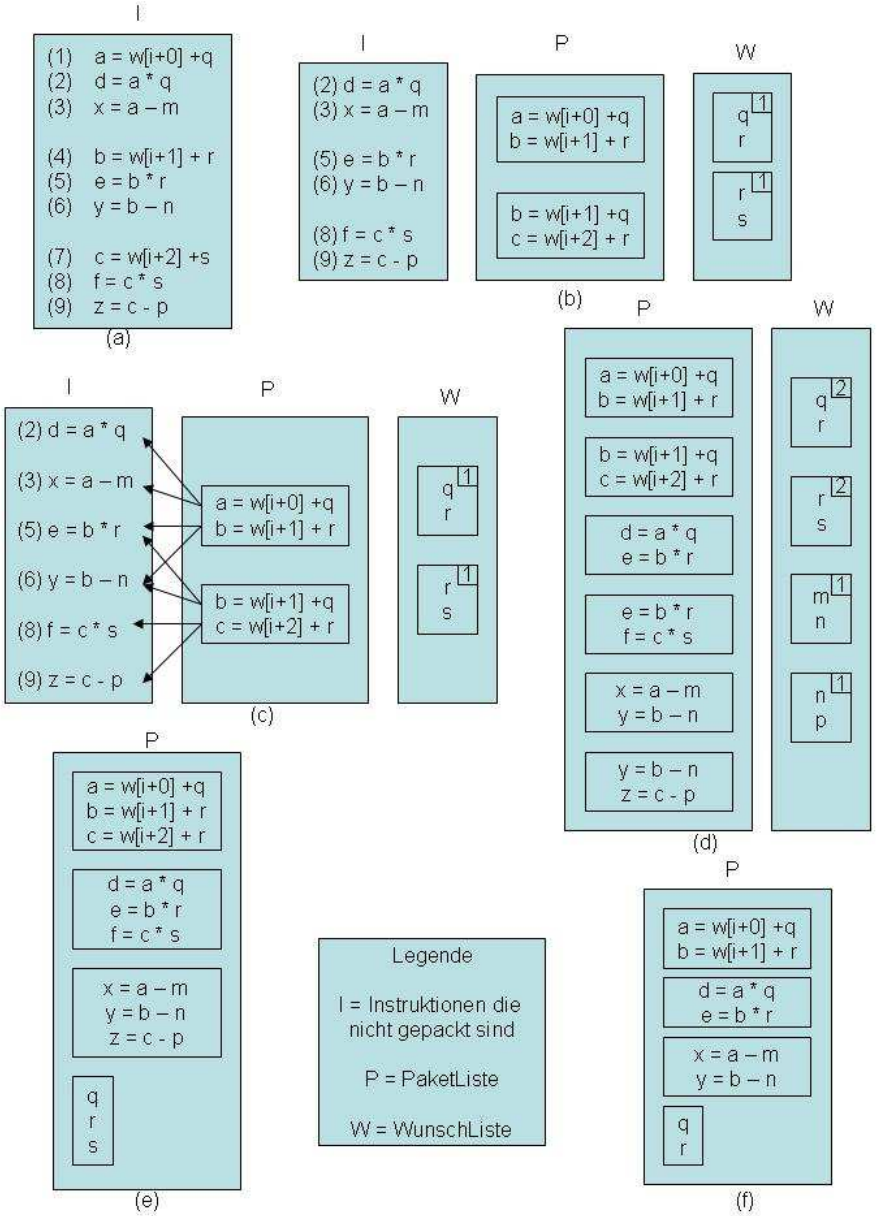
In Tabelle 1 sind die Ergebnisse der Benchmark Tests, die mit SLP durchgeführt wurden zusammengefasst. Die Tests wurden mit 3 verschiedenen Datenpfadgrössen durchgeführt. Das SPEC95fp Benchmark Set arbeitet schlecht auf 64 Bit. Die einzige Ausnahme bildet hier das swim Set. Mit einem 128 Bit Datenpfad ist die Beschleunigung schon erheblich größer, weil hier die Möglichkeit bestand, zwei Operanden in eine Anweisung zu packen. Die Verwendung von 256-Bit Datenpfaden brachte noch bessere Ergebnisse. Das liegt daran, dass hier Parallelität noch besser ausgenutzt werden konnte. Im Schnitt konnte eine Steigerung der Performanz um 1.71 über alle Testsets erzielt werden.

Das UTDSP Benchmark Set beruht auf 32-Bit Datentypen, deshalb kommt es schon bei 64-Bit Datenpfaden zu solch einer Steigerung. Einige dieser Benchmarks profitierten von den angrenzenden Speicherreferenzen in Schleifen. Bei der Steigerung der Datenpfadgrösse auf 128 bzw. 256-Bit brachte das Schleifen Abrollen 3.1 keine neue Parallelität und nur begrenzte Steigerung der Performanz.

5 Zusammenfassung

In dieser Seminararbeit wurde Superword Level Parallelismus [4], eine Möglichkeit Parallelismus in Code zu finden und in Superwords auszuführen, vorgestellt. SLP bietet dabei eine robuste und einfache Möglichkeit, Code für normale Desktopprozessoren zu beschleunigen. Dabei benutzt SLP die von den Hardwareherstellern eingeführten Multimedia–Erweiterungen. Auf den beiden zur Evaluation des Algorithmus verwendeten Benchmarks führte SLP zu einer durchschnittliche Steigerung von 170%.

Abbildung 8 Arbeitsweise des SLP–Algorithmus. (a) Initiale Sequenz von Instruktionen (b) Nach dem Anweisungen mit Angrenzenden Speicherreferenzen gepackt wurden (c) Suche nach Datenabhängigkeiten (d) Gepackte Anweisungen die von der Paketliste stammen (e) Nach der Kombination (f) Nach dem Datenabgleich zu einer Architektur mit Zweiwege Parallelismus



	Daten path size					
Benchmark	256 bits		128 bits		64 bits	
bench	%saved	speedup	%saved	speedup	%saved	speedup
swim	71.66%	3.529	68.96%	3.222	48.73%	1.950
tomcatv	66.41%	2.977	47.12%	1.891	10.60%	1.119
mgrid	59.22%	2.452	41.70%	1.715	1.91%	1.019
su2cor	44.57%	1.804	34.68%	1.531	10.80%	1.121
hydro2d	25.65%	1.345	19.21%	1.238	5.10%	1.054
apsi	25.44%	1.341	21.52%	1.274	7.06%	1.054
wave5	23.54%	1.308	19.30%	1.237	5.56%	1.059
applu	21.87 %	1.280	16.26%	1.194	6.36%	1.068
turb3d	21.46%	1.273	17.14%	1.207	2.75%	1.028
fpppp*	15.53%	1.184	10.24%	1.114	0.00%	1.000
fir	58.08%	2.385	50.88%	2.036	39.82%	1.662
lmsfir	52.99%	2.127	49.58%	1.983	38.90%	1.637
latnrm	49.96%	1.998	49.80%	1.992	37.55%	1.601
fft	40.90%	1.692	40.90 %	1.692	32.32%	1.478
iir	33.09%	1.495	33.09%	1.495	29.12%	1.411
mult	20.71%	1.261	19.19%	1.237	14.07%	1.164

Tabelle 1: Ergebnisse des SPEC95fp Benchmarks und der UTDSP Kernels

Literaturverzeichnis

[1] D. Callahan and P. Havlak. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 5, 1986.

[2] Standard Performance Evaluation Corporation. Spec95fp. `http://www.specbench.org/`. 11. 02. 2004.

[3] L. Gwennap. Altivec vectorizes powerpc. *Microprocessor Report*, 1998.

[4] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *ACM SIGPLAN Notices*, 35(5):145–156, 2000.

[5] C. Lee and M. Stoodley. Utdsp benchmark suite. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`. 11. 02. 2004.

[6] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 45–55. IEEE Computer Society, 2002.

[7] R.E. Tarjan. Scalar expansion in pfc: Modifications for parallelization. *Supercomputer Software Newsletter*, 1972.

Georg Eckert
Betreuer: Nicolas Fritz

Vektorisierung

Um die Programmausführung zu beschleunigen, können mehrere Programmteile parallel ausgeführt werden. Wie teilt man aber das Programm sinnvoll auf? Bleibt dabei die Semantik des Programms erhalten? Mit diesen Fragen beschäftigt sich die Vektorisierung.

1 Einführung

Seitdem die modernen Graphikkarten programmierbare Prozessoren haben, spielt natürlich die Leistungssteigerung der Programme, die auf diesen Prozessoren laufen, eine relevante Rolle. Eine Möglichkeit der Leistungssteigerung besteht darin, dass mehrere Programmteile parallel ausgeführt werden können. Dabei ist zu beachten, dass bei der Programmaufteilung die Semantik des ursprünglichen Programms erhalten bleibt. Dieser Frage ist unter anderem *Vektorisierung* gewidmet.

Vektorisierung ist die Umwandlung des Programms in den Vecktorcode.

Betrachten wir folgendes Programm:

```
DO I=1,100
  A(I-2)=C(I)
END DO I
```

Die Vektorcode für dieses Programm ist:
A[-1:98:]=B[1:100], wobei A und B eindimensionale Vektoren mit jeweils 100 Elementen sind. **A[-1:98]** repräsentiert dabei den Vektor mit Elementen (A(-1), A(0),..., A(98)) und B - den Vektor mit Elementen (B(1),...,B(100))). Alle Anweisungen von diesem Vektorcode können nun gleichzeitg ausgeführt werden, ohne dass die Semantik des Programms verletzt wird.

Notationsvereinbarung: Sei (x_1, x_2) ein Paar und A eindimensionaler Vektor. $A(x_1, x_2) = \{A(i) : x_1 \leq i \leq x_2\}$. Diese Notation ist auch auf mehrdimensionale

Vektoren erweiterbar.

A(I,1:M) präsentiert beispielsweise die I-te Zeile vom zweidimensionalen Vektor. $A(T_1 : U_1, T_2 : U_2)$ besteht aus allen Elementen von A(I,J), für die gilt: $T_1 \leq I \leq U_1, T_2 \leq J \leq U_2$

2 Grundbegriffe

Bei der Umwandlung eines Programms muss natührlich die Semantik erhalten bleiben. Wenn das Programm in mehrere Teile aufgeteilt wurde, die gleichzeitig ausgeführt werden, so stellt sich die Frage, ob dabei die Semantik beibehalten wurde. Diese Frage kann nun mit Hilfe von der Datenabhängigkeitsanalyse beantwortet werden.

Zuerst werden einige Definionen benötigt.

Definition 1: Sei P ein Programm, dass aus einem Schleifennest $L_1, ..., L_n$ und einer Anweisung S besteht, wobei $L_i : DO\ I_i = t_i, u_i$ eine Schleife ist. Bei einer bestimmten Ausführung von S ist $\mathbf{i} = \{i_j : \forall j\ aus[1, n]\ und\ t_j \leq i_j \leq u_j\}$ ein *Iterationsvektor* und S(**i**) eine *Instanz* von S. Es seien **i** und **i'** Iterationsvektoren. Die normale Ausführungsreihenfolge S(**i**) $\ll$ S(**i'**) gilt genau dann, wenn **i** $\leq$ **i'**

Definition 2: Sei S eine Programmanweisung. Die *Eingabe- und Ausgabemenge* von S (input and output sets of S) *DEF(S)* und *USE(S)* sind folgendermaßen definiert.

DEF(S) := {v $\in$ VARX: v wird in S definiert }
USE(S) := {v $\in$ VARX: v wird in S verwendet },

wobei *VARX* alle Variablen von S beinhaltet.

In Abbildung 1 ist ein Programm mit zwei Anweisung S und S' dargestellt. Die Eingabe- und Ausgabemengen von S uns S' sind:
DEF(S)={A(2*I)}, USE(S)={I,B(I)}
DEF(S')={D(I)}, USE(S')={I,A(2*I+1)}

Für einen Iterationsvektor **i**={i}, $1 \leq i \leq 100$ gilt:
DEF(S(**i**))={A(2*i)}, USE(S**i**))={I, B(i)}
DEF(S'(**i**))={D(i)}, USE(S'(**i**))={I, A(2*i+1)}

```
DO I=1,100
  S: A(2*I)=B(I)+1
  S': D(I)=A(2*I+1)
END DO I
```

Abb. 1: Beispiel für DEF und USE

Definition 3: Es seien S und S' Anweisungen in einem Schleifennest und **i**,**i'** ihre Iterationsvektoren. Dann ist S'(**i'**) von S(**i**) datenabhängig, wenn folgende Bedingungen erfüllt sind:

1. S'(**i'**) $\ll$ S(**i**)
2. $\exists v \in$ VARX:
   v $\in DEF(S(\mathbf{i})) \bigcap USE(S'(\mathbf{i}'))$ $\vee$ (2.1)
   v $\in USE(S(\mathbf{i})) \bigcap DEF(S'(\mathbf{i}'))$ $\vee$ (2.2)
   v $\in DEF(S(\mathbf{i})) \bigcap USE(S'(\mathbf{i}'))$ (2.3)
3. Es gibt keine andere Instanz SI, so daß:
   S(**i**) $\ll$ SI $\ll$ S'(**i'**) und $v \in DEF(SI)$

Ist S'(**i'**) von S(**i**) datenabhängig, so schreibt man S(**i**) δ S'(**i'**)

Dabei unterscheidet man drei Arten von der Datenabhängigkeit: *true, anti* und *output* dependence.

- ist die Bedingung 2.1 erfüllt, so gilt S(**i**) δ^t S'(**i'**) (true dependence)
- ist die Bedingung 2.2 erfüllt, so gilt S(**i**) δ^a S'(**i'**) (anti dependence)
- ist die Bedingung 2.3 erfüllt, so gilt S(**i**) δ^o S'(**i'**) (output dependence)

```
DO I=1,100
  S: X(I+1)=X(I)+Y(I)
END DO I
```

Abb. 2: Beispiel für Datenabhängigkeit

Betrachten wir das Programm aus Abbildung 2. Die Eingabe- und Ausgabemenge von der Anweisung S sind DEF(S)={X(I+1)} und USE(S)={X(I),Y(I),I}. Für einen Iterationsvektor **i**={i}, $i \in [1, 100]$ gilt: DEF(S(**i**))={X(i+1)}, USE(S(**i**))={X(i),Y(i),i}. Um die Datenabhängigkeit zu ermitteln, sollen zwei Iterationsvektoren **i**, **i'** gefunden werden, die der Bedingungen 1,2 der Definition 3 entsprechen. Damit die Bedingung 1 erfüllt ist, muss $1 \leq \mathbf{i} \leq \mathbf{i}' \leq 100$ gelten. Um die Bedingung 2 zu erfüllen, muss die geeignete Variable v gefunden werden, für die gilt:

$v \in \{X(i+1)\} \bigcap \{X(i'), Y(i'), I\}$ oder
$v \in \{X(i), Y(i), I\} \bigcap \{X(i'+1)\}$ oder
$v \in \{X(i+1)\} \bigcap \{X(i'+1)\}$

Wählen wir i'=i+1, so gilt die obere Bedingung $v \in \{X(i+1)\} \bigcap \{X(i'), Y(i'), I\}$. Offensichtlich gilt auch die Bedingung 3, da es zwischen S(**i**) und S'(**i'**) keine andere Instanz geben kann. Somit gilt: S(**i**) δ^t S'(**i'**)

Definition 4: Es seien $x, y \in \mathbb{Z}^n$ zwei Vektoren und $c \in [1, n]$. $x <_c y$ gilt genau dann, wenn $(\forall j \in [1, c-1] : x_i = y_i) \wedge x_c < y_c$

Zur Verdeutlichung betrachten wir ein Beispiel. Seiec $c \in [1,3]$ und $x, y \in \mathbb{Z}^3$. Dann gilt:

$(1,2,3) <_1 (2,5,9)$
$(2,5,7) <_1 (3,1,1)$
$(3,3,3) <_2 (3,4,1)$
$(5,5,5) <_3 (5,5,8)$

Es gilt aber nicht:
$(1,2,3) <_2 (2,5,9)$

Man unterscheidet zwei weitere Arten von Datenabhängigkeit: *loop-carried dependence* und *loop independent dependence.*

Definition 5: Es seien S und S' die Anweisungen und **i**,**i'** ihre Iterationsvektoren, so daß $S(\mathbf{i}) \ll S'(\mathbf{i}')$ gilt. Es gilt:

$S(\mathbf{i}) \ll_c S'(\mathbf{i}')$, wenn $\mathbf{i} <_c \mathbf{i}'$ $S(\mathbf{i}) \ll_\infty S'(\mathbf{i}')$, wenn $\mathbf{i} = \mathbf{i}'$ und S' im Programm nach S vorkommt.

Weiterhin, wenn wir annehmen, daß S'(**i'**) von S(**i**) datenabhängig ist, also S(**i**) δ S'(**i'**), dann gilt:
S(**i**) δ_c S'(**i'**) genau dann, wenn $S(\mathbf{i}) \ll_c S'(\mathbf{i}')$ (loop-carried dependence)
S(**i**) δ_∞ S'(**i'**) genau dann, wenn $S(\mathbf{i}) \ll_\infty S'(\mathbf{i}')$ (loop-independent dependence)

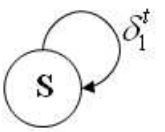
Wenn S(**i**) δ_k S'(**i'**) gilt, wobei $k \in [1,n] \cup \infty$ dann ist k die *Abhängigkeitsebene* (level of the dependence)

Bei der Vektorisierung spielt der *Abhängigkeitsgraph* eine wichtige Rolle. Mit Hilfe vom Abhängigkeitsgraph können die Datenabhängigkeiten veranschaunlicht werden.

Definition 6: Der Abhängigkeitsgraph ist ein gerichteter Graph, wessen Knoten die Programmanweisungen und wessen Kanten die Datenabhängigkeiten darstellen. Dabei wird eine Kante von einem Knoten S zu einem Knoten S' eingefügt, falls $S \delta S'$.

In Fig. 1 ist ein Abhängigkeitsgraph für Programm aus Abbildung 2 dargestellt.

Abbildung 1 Abhängigkeitsgraph für das Programm aus Abb. 2



3 Grundlegende Umwandlungen

Hier werden grundlegende Umwandlungen vorgestellt. Außerdem werden die Bedingungen abgeleitet, unter denen diese Umwandlungen semantisch korrekt sind.

3.1 Vertausch von Anweisungen in einer Schleife (statement reordering)

Bei diesem Ansatz werden zwei Programmanweisungen, die benachbart sind, vertauscht. Dies kann auf diejenigen Schleifen angewandt werden, die mindestens zwei Anweisungen beinhalten.

Vertausch von Anweisungen:

Eingabe: Schleife, die zwei benachbarte Anweisungen S und S' beinhaltet. S' kommt in der Schleife nach S vor.

Ausgabe: modifizierte Schleife, in der S und S' vertauscht wurden.

```
DO I=1,100                          DO I=1,100
  S: A(I)=B(I)*2                      S': C(I)=A(I-1)-4
  S': C(I)=A(I-1)-4          →        S: A(I)=B(I)*2
END DO I                            END DO I
```

Abb. 3: gültige Umwandlung

```
DO I=1,100                          DO I=1,100
  S: A(I)=B(I)*2                      S': C(I)=A(I)-4
  S': C(I)=A(I)-4            →        S: A(I)=B(I)*2
END DO I                            END DO I
```

Abb. 4: ungültige Umwandlung

In Abbildung 3 ist ein Beispiel für die Umwandlung dargestellt. In diesem Programm besteht eine loop-carried dependence zwischen S und S'. Nach der Umwandlung bleibt diese Abhängigkeit erhalten, also ist die Semantik unverändert geblieben.

Im nächsten Beispiel aus Abbildung 4 ist die Umwandlung hingegen ungültig. Im ursprünglichen Programm besteht eine loop-independent dependence zwischen S und S'. Nach der Umwandlung geht diese Abhängigkeit verloren, also ist die Umwandlung ungültig.

Zima hat gezeigt, daß der Vertausch von zwei Anweisungen S und S' nur dann gültig ist, wenn keine loop-independent dependence zwsischen S und S' besteht [1].

Der Vertausch von zwei Anweisungen wird bei der Schleifenaufteilung verwendet.

3.2 Aufteilung einer Schleife (loop distribution)

Ein weiterer Ansatz besteht darin, daß die Schleife aufgeteilt wird, d.h. im Endeffekt für jede Programmanweisung ein perfektes Schleifennest gebildet wird. Dieser Ansatz heißt *Schleifenaufteilung* (loop distribution).
Definition 3.1: Es sei ein Programm P, das aus einem Schleifennest $L_1, ..., L_n$ und einer Anweisung S besteht, wobei L_i: DO $I_i = t_i, u_i$ eine Schleife ist. Man sagt, daß $L_1, ..., L_n$ ein *perfektes Schleifennest* ist, wenn für alle $j \in [1, n-1]$ gilt: L_{j+1} ist der Schleifenkörper von L_j.

Definition 3.2: Die Schleifenaufteilung ist *distributiv.* wenn das modifizierte Programm semantisch identisch mit dem alten ist.

In Abbildung 5 ist ein Beispiel für die Schleifenaufteilung dargestellt. Wie man sieht, ist die Schleifenaufteilung in dem Fall distributiv.

```
DO I=1,100                                  DO I=1,100
  DO J=1,100                                  DO J=1,100
    S: C(I,J)=0                                 S': C(I,J)=0
    DO K=1,100                                  END DO J
      S': C(I,J)=C(I,J)+A(I,K)                END DO I
    END DO K                  →
  END DO J                                  DO I=1,100
END DO I                                      DO J=1,100
                                                DO K=1,100
                                                  S': C(I,J)=C(I,J)+A(I,K)
                                                END DO K
                                              END DO J
                                            END DO I
```

Abb. 5: Beispiel für die Schleifenaufteilung

Zima hat gezeigt, daß die Schleifenaufteilung nur dann distributiv ist, wenn es keine loop-carried dependence im Programm gibt. In dem anderen Fall ist es jedoch möglich, mit Hilfe vom geeignten Anweisungsvertausch die Schleifenaufteilung teilweise druchzuführen, was im nachfolgenden Beispiel präsentiert wird. Dies kann aber nur dann durchgeführt werden, wenn der zugehörige Abhängigkeitsgraph azyklisch ist [1].

Schleifenaufteilung:

Eingabe: Ein Schleifennest mit Anweisungen $S_1, ..., S_n$. Die Anweisungen sind in Blöcke $B_1, ..., B_n$ aufgeteilt, so daß jeder Block diejenige benachbarte Anweisungen enthält, die im zugehörigen Abhängigkeitsgraph einen Zyklus bilden.

Ausgabe: eine Schleifensequenz $DISTR(B_1), ..., DISTR(B_n)$, wobei $DISTR(B_j)$ ein perfektes Schleifennest für den Block B_j ist.

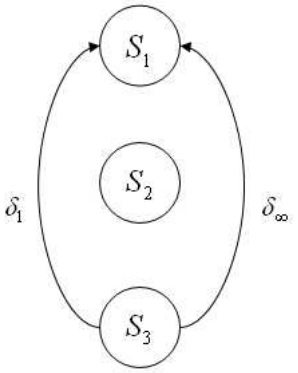
Betrachten wir zur Verdeutlichung ein Beispiel. In Abb. 6 ist ein Programm, in Fig. 2 sein Abhängigkeitsgraph dargestellt. Zuerst sollen die Blöcke ermittelt werden. Im Graph gibt es zwar einen Zyklus, doch er besteht aus den Anweisungen, die nicht benachbart sind. Es kann aber ein Vertausch von Anweisungen S_2 und S_3 durchgeführt werden, diese Umwandlung ist gültig, da keine Datenabhängigkeit zwischen S_2 und S_3 besteht. Nun können die Blöcke gebildet werden, und zwar $B_1 = \{S_1, S_3\}$ und $B_2 = \{S_2\}$. Daraufhin wird das Schleifennest aufgeteilt, das Ergebnis ist in Abbildung 7 gezeigt.

Erfolgt eine Schleifenaufteilung teilweise, so können später nur die Schleifen vektorisiert werden, die genau eine Anweisung erhalten.

```
DO I=1,100
  DO J=1,100
    S1: C(I)=A(I-2)*B(I)
    S2: D(I)=B(I)+B(I-1)
    S3: A(I)=C(I)+2
  END DO J
END DO I
```

Abb. 6: Beispiel für die Schleifenaufteilung

Abbildung 2 Abhängigkeitsgraph für das Programm aus Abb. 6




```
DO I=1,100
  DO J=1,100
    S1: C(I)=A(I-2)*B(I)
    S3: A(I)=C(I)+2
  END DO J
END DO I
```

```
DO I=1,100
  DO J=1,100
    S2: D(I)=B(I)+B(I-1)

  END DO J
END DO I
```

Abb. 7: Schleifenaufteilung

3.3 Vektorisierung

Wenn der Vertausch von Anweisungen und die Schleifenaufteilung auf ein Programm erfolgreich angewandt wurden, kann das modifizierte Programm vektorisiert werden.

Vektorisierung:

Eingabe: Ein perfektes Schleifennest $L_1, ..., L_n$ mit genau einer Anweisung S.

Ausgabe: Ein Vektor $[T_1 : U_1, ..., T_n : U_n]$,wobei T_j, U_j die Werte der entsprechenden Schleife für alle $j \in [1, n]$ sind.

Die Vektorisierung ist gültig, wenn $S \not\delta S$ gilt. Im anderen Fall kann die Vektorisierung jedoch möglich sein, wenn andere Ansätze (z.B. variable copying) zusätzlich verwendet werden. Auf diese Ansätze wird im Abschnitt 5 eingegangen.

Definition 3.3: Sei G=(V,E) ein gerichteter Graph, wobei V die Knotenmenge und E die Kantenmenge sind.

1. G heißt *stark zusammenhängend*, falls von jedem Knoten jeder Knoten erreicht werden kann.Eine*stark zusammenhängende Komponente*(SCC) ist ein maximaler stark zusammenhängender Teilgraph von G
2. Der *zugehörige azyklische Graph* $AC(G) = (V', E')$ ist ein derichteter Graph für den gilt: $V' = \{C : \text{C ist SCC von G}\}$, $E' = \{(v_1', v_2') : v_1', v_2' \in V' \text{ und } \exists (v, w) \in E \text{ mit } v \in v_1' \text{ und } w \in v_2'\}$

Im folgenden wird der Vektorisierungsansatz beschrieben, der die Umwandlungen von 3.1 und 3.2 beinhaltet.

Vektorisierungsansatz:

Es seien P ein Programm mit dem Schleifennest und Anweisungen $S_1, ..., S_n$ und G der zugehörige Abhängigkeitsgraph.

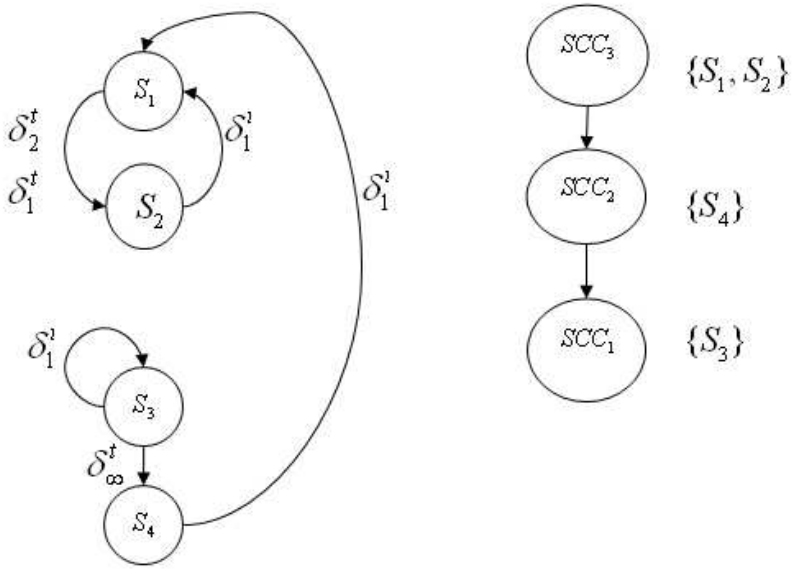
1. Konstruiere stark zusammenhängende Komponenten SCCs und den zugehörigen azyklischen Graph AC(G).
2. Sortiere SCCs in topologischer Ordnung
3. Führe gültige Vertauschoperationen von Anweisungen durch, so daß:
   (a) alle Anweisungen in jeder SCC benachbart sind.
   (b) alle Anweisungen bzgl. alles SCCs in topologischer Ordnung sind.
4. das modifizierte Programm ist distributiv. Ermittle alle Blöcke $B_1, ..., B_n$ und führe die Schleifenaufteilung durch.
5. Diejenige Blöcke, die genau eine Anweisung enthalten, können nun vektorisiert werden.

Betrachten wir dazu ein Beispiel. Abbildung 8 zeigt hierfür ein Programm.

```
DO I=1,100
  DO J=1,100
    S1: C(I)=A(I,J)*B(I,J)
    S2: A(I+1,J+1)=C(I,J-2)+C(I-1,J)
    S3: D(I,J)=D(I-1,J-1)
    S4: B(I,J+4)=D(I,J)
  END DO J
END DO I
```

Abb. 8: Ein Beispielsprogramm

Abbildung 3 a) Abhängigkeitsgraph für das Programm aus Abb. 8 und b) der zugehörige azyklische Graph



1. In Fig. 3 sind sein Abhängigkeitsgraph und azyklischer Graph dargestellt. Die SCCs sind: $SCC_1 = \{S_3\}, SCC_2 = \{S_4\}, SCC_3 = \{S_1, S_2\}$.

2. Die Sortierung nach topologischer Ordnung ergibt die folgende Reihenfolge: SCC_1, SCC_2, SCC_3

3. (a) in jeder SCC sind die Anweisungen schon benachbart.

   (b) die Sortierung nach topologischer Ordnung bzgl. aller SCCs ergibt folgende Reihenfolge von Anweisungen im Programm: S_1, S_2, S_3, S_4 . D.h., es wurden S_1 mit S_3 und S_2 mit S_4 vertauscht. Alle diese Transformationen sind gültig, da zwischen S_1, S_3 und zwischen S_2, S_4 keine loop-independent dependence besteht.

4. Nun werden die Blöcke ermittelt. Es gibt insgesamt drei Blöcke: $B_1 = \{S_3\}, B_2 = \{S_4\}, B_3 = \{S_1, S_2\}$. Die Schleifenaufteilung kann nun durchgeführt werden. Daraus erhält man:

DO I=1,100	DO I=1,100	DO I=1,100
DO J=1,100	DO J=1,100	DO J=1,100
S_3: ...	S_4: ...	S_1: ...
		S_2: ...
END DO J	END DO J	END DO J
END DO I	END DO I	END DO I
Schleife 1	Schleife 2	Schleife 3

5. Von den obigen drei Schleifen kann nur die zweite vektorisiert werden. Bei der ersten ist die Vektorisierung nicht möglich, da $S_3 \delta S_3$ gilt und in der dritten gilt: $S_1 \delta_1^t S_2$ und $S_2 \delta_1^t S_1$. Nun haben wir das endgültige Ergebnis:

   Schleife 1
   B(1:100,5:104)=D(1:100,1:100)
   Schleife 2

4 Allen-Kennedy Algorithmus

Hier wird ein Ansatz von Allen/Kennedy vorgestellt, der weitere Vektorisierung ermöglicht.

Zuerst wird eine Bedingung abgeleitet, unter welcher die gleichzeitige Ausführung aller Instanzen an einer bestimmten Schleifenebene semantisch korrekt ist.

Definition 4.1: Sei G=(V,E) ein Abhängigkeitsgrpah.

1. levels(e) ist die Menge aller Abhängigkeitsebenen, die zur $e \in E$ gehören.
2. minlevel(e)=min(levels(e))
3. maxlevel(e)=max(levels(e))
4. $constrain(G, k) = (V, E_k)$, wobei $k \in \mathbb{N} \bigcup \{\infty\}$, eine bestimmte Ebene ist und $E_k = \{e \in E : maxlevel(e) \geq k\}$

Betrachten wir den Graph aus Abbildung 11 a): $level(S_1, S_2) = \{1, 2\}, level(S_3, S_4) = \{\infty\}$, D(G,2) ist der obige Graph, bei dem die Kanten (S_1, S_2) und (S_3, S_3) wegfallen.

Definition 4.2: Sei π ein Pfad in einem Graph.

1. $levels(\pi) = \{k \in (N) \bigcup \{\infty\} : \forall e \in \pi : maxlevel(e) \geq k$ und $\exists e \in \pi : k \in levels(e)\}$

2. $minlevel(\pi) = min(levels(\pi))$
3. $maxlevel(\pi) = max(levels(\pi))$

Sei π beispielsweise ein Pfad aus e_1, e_2, e_3 mit $levels(e_1) = \{1, 4\}, levels(e_2) = \{2, 5\}$ und $levels(e_3) = \{\infty\}$. Dann gilt: $levels(\pi) = \{1, 2, 4\}$

Definition 4.3: Seien S eine Anweisung und G ein Graph. $CP(S) = \{\pi : \pi$ ist der Pfad $S_0, ..., S_n$ aus $G, n \geq 1$ und $S_0 = S_n = S\}$.
Mit anderen Worten ist CP(S) die Menge aller zyklischen Pfaden von S nach S.

Definition 4.4: Seien P ein Programm mit dem Schleifennest $L_1, ..., L_n$ und S eine Anweisung in der Schleife $j \in [1, n]$. S ist an der Ebene $c \in [1, n]$ *serial*, wenn c>j oder $\exists \pi \in CP(S) : c \in levels(\pi)$. Ansonsten ist S an der Ebene c *concurrent*.
Wenn S concurrent an der Ebene c ist, dann bedeutet dies, daß die Instanzen von S in der Schleife L_c gleichzeitg ausgeführt werden können.

Wenn ein Programm im Form eines perfekten Schleifennestes sich nicht vollständig vektorisieren lässt, so wird danach angestrebt, so viel, wie möglich davon zu vektorisieren. Eine Anweisung S im perfekten Schleifennest kann nur dann vektorisiert werden, wenn S $\not\delta$S gilt, wenn es also keinen Abhängigkeitszyklus gibt. Im anderen Fall soll die größte Anzahl der äußeren Schleifen gefunden werden, die den Abhängigkeitszyklus verursachen. Den Rest vom Programm kann dann vektorisiert werden.

Definition 4.5: Seien P ein Programm mit dem Schleifennest $L_1, ..., L_n$ und S eine Anweisung.

1. $maxcycle(S) = \begin{cases} max(maxlevel(\pi) : \pi \in CP(S) & CP(S) \neq \oslash \\ 0 & sonst \end{cases}$
2. $\alpha(S) = n - maxcycle(S)$

$\alpha(S)$ heißt Index der Vektorisierung von S. Im Schleifennest können $\alpha(S)$ innere Schleifen zusammen mit S vektorisiert werden. Mit anderen Worten: Sei L Schleifennest $L_1, ..., L_n$ mit der Anweisung S. L kann nun folgendermaßen vektorisiert werden:

```
DO I_1 = t_1, s_1
  DO I_2 = t_2, s_2
    ...
      DO I_c = t_c, s_c
        S(I_1, ..., I_c, T_{c+1} : U_{c+1}, ..., T_n : U_n)
      END DO I_c
    ...
  END DO I_2
END DO I_1
```

wobei $c = maxcycle(S)$ ist.

Diese Eigenschaft heißt *partielle Vektorisierung*, die Vektorisierung erfolgt dabei in n-c inneren Schleifen. Darauf ist Allen-Kennedy Algorithmus basiert.

Allen-Kennedy Algorithmus:

Eingabe: Schleifennest L mit zugehörigem Abhängigkeitsgraph G und die Anweisungsmenge STAT. **Ausgabe:** modifiziertes Programm L', ind er Form wie oben

procedure vectorize(R,c)
/* R ist die Menge von Anweisungen in L, und $c \in \mathbb{N}$ ist die Ebene. Diese Prozedur generiert den Code für die Anweisungen an der Ebene c von L. Wenn c>1, dann repräsentiert R eine stakr zusammenhängende Komponente SCC von constrain(D,c-1) un der Code wurde bereits für äußeren Schleifen (in denen sich R befindet) generiert. */
begin
  $DD = constrain(G, c)/R$
  $G' = (N', E') = AC(DD)$
  for every $n' \in N'$ (in topologischer Ordnung)
    if $n' \in SCCG_i = (N_i, E_i)$ von DD
      then
        /* Es gibt einen zyklischen Pfad π in DD, in dem alle Knoten von N_i enthalten sind und für den gilt: $maxlevel(\pi) \geq c$ (S kann an der Ebene c concurrent sein, aber es gibt eine Ebene $c' \geq c$, in der S serial ist). In dem Fall wird die Schleife an der Ebene c für alle Anweisungen von N_i sequentiell ausgeführt. */
        generate('DO $I_c = T_c, U_c$');
        vectorize('DO $N_i, c+1$');
        generate('END DO I_c');
      else
        /* n' gehört zu SCC $G_i = (\{S\}, \oslash\}$ von DD, wobei S eine Anweisung ist, Dann gilt: $c = maxcycle(S) + 1, \alpha(S) = n - (c-1)$. S kann in dem Fall vektorisiert werden */
        generate('$S(I_1, ..., I_{c-1},\ T_c : U_c, ..., T_n : U_n)$')
    fi
  end for
end;

begin vectorize(STAT,1)
end

Wenden wir den Algorithmus auf das Programm aus Abbildung 10 an, so erhält man folgendes Ergebnis:

DO I=1,100
  S_{vect3} : D(I,1:100)=D(I-1,0:99)
END DO I
B(1:100, 5:104)=D(1:100,1:100)
DO I=1,100
  S_{vect1} : C(I,1:100)=A(I,1:100)*B(I,1:100))

S_{vect2} : A(I+1,2:101)=C(I,-1:98)*C(I-1,1:100)
END DO I

5 Weitere Transformationen

Es gibt eine Reihe von Ansätzen, die weitere Vektorisierung ermöglichen. Sie, sowie ihre Effekte, sind im Einzeln aufgelistet. Für die Details wird auf [1] verwiesen.

Loop interchange - Reduzierung von maxcycle
Scalar expansion - Zerstörung des Zyklus im Abhängigkeitsgraph
Variable copying - ermöglicht die Vektorisierung, in dem die Datenabhängigkeiten eliminiert werden
Index set splitting - kann den Zyklus im Abhängigkeitsgraph brechen *Node splitting* - Reduzierung des Zyklus im Abhängigkeitsgraph.

6 Kontrollabhängigkeit

Bisher wurden nur sequentielle Programme untersucht, d.h. die Programme, in denen die Anweisungen immer nacheinander ausgeführt werden. Ist das Programm nicht sequentiell, so sind die Datenabhängigkeiten für die Programmanalyse unzureichend. Für die vollständige Programmanalyse werden die *Kontrollabhängigkeiten* benötigt.

Definition 6.1: Seien S_1 und S_2 Anweisungen in einem Programm. Wenn während der Ausführung von S_1 entschieden wird, ob S_2 unmittelbar danach ausgeführt wird, so ist S_2 kontrollabhängig von S_1.

DO I=1,100
  S_1: IF A(I)=0 THEN GOTO S_3 FI
  S_2: B(I)=1
  S_3: C(I)=A(I)=3
END DO I

Abb. 9: Beispiel für Kontrollabhängigkeit

So ist S_3 im Programm aus Abbildung 9 kontrollabhängig von S_1.

Wir fügen zwei Sprunganweisungen ein:

- GOTO Anweisung
- If Anweisung im Form:
  IF Bedingung THEN S_1 [ELSE S_2] FI, falls S_1 [und S_2] GOTO-Anweisungen sind.

Definition 6.2: Eine *Quelle* ist die Sprunganweisung. Ein *Ziel* ist die Anweisung, die unmittelbar nach der Quelle ausgeführt wird.

Definition 6.3: Sei L ein Schleifennest mit der Quelle S an der Ebene n, und S' das Ziel an der Ebene n'. Der Sprung ist:

- *vorwärts*, falls S vor S' liegt und n=n'
- *rückwärts*, falls S' vor S liegt und n=n'
- ein *Endsprung*, falls n'<n

```
DO I=1,100
  S1 : ···
  DO J=1,100
    S2: IF A(I)=0 THEN GOTO S1 FI
    S3: IF B(I)=0 THEN GOTO S4 FI
    S5: GOTO S7
    S6: IF C(I)≠0 THEN GOTO S2 FI
    S7 : ···
   END DO J
END DO I
S4 : ···
```

Abb. 10: Vorwärts-, rückwärts- und Endsprünge

Betrachten wir das Programm aus Abbildung 10. Die Sprünge an den Stellen S_2 und S_3 sind Endsprünge. Der Sprung an der Stelle S_5 ist vorwärts und an der Stelle S_6 rückwärts.

Definition 6.4 Sei S eine Anweisung, mask(S) ist ein boolescher Ausdruck, der besagt, unter welcher Bedingung S ausgeführt wird.

Betrachten wir Programm aus Abbildung 9. mask$(S_2) = \{A(I) \neq 0\}$, mask$(S_3) = \{A(I) = 0\}$.

6.1 If-conversion

Dieser Ansatz konvertiert Kontrollabhängigkeit in Datenabhängigkeit, so dass anschließend die Vektorisierung durchgeführt werden kann. Dabei ist der Ansatz nur für Programme mit Vorwärtssprüngen tauglich. Für die weitere Ansätze, die auf Programme mit Rückwärts-, Endsprüngen und Prozeduren anwendbar sind, wird auf [1] verwiesen.

If-conversion Algorithmus:

Eingabe: Eine Sequenz von Anweisungen $SEQ = S_1, ..., S_n, n \geq 1$, die folgende Bedingungen erfüllt:

- es gibt keine Rückwärts- und Endsprünge in SEQ
- alle Anweisungen befinden sich in einem Schleifennest
- $mask(S_1) = C_1$
- wenn $S \in SEQ$ das Ziel ist, dann befindet sich die Quelle in SEQ

/* $source\_conds(S)$ ist eine globale Variable für eine Anweisung S. Wenn S das Ziel ist, dann ist $source_cond(S)$ die Menge aller Masken von Quellen von S. Ansonsten ist $source\_cond(S)$ leer. $code(cond)$ spezifiziert den Ausgabecode für die Bedingung cond */

function generate_masked_statement(S, AC): boolean;
begin
$curr\_cond = AC \vee (\vee\{sc : cs \in source\_cond(S)\}$
case S **in**:

$S =' IF\ cond\ THEN\ SEQ_1\ ELSE\ SEQ_2\ FI'$
begin
$c\_true := generated\_masked\_sequence(SEQ_1, curr\_cond \wedge cond)$;
$c\_false := generated\_masked\_sequence(SEQ_1, curr\_cond \wedge \neg cond)$;
$curr\_cond := c\_true \vee c\_false$;
end

$S =' IF\ cond\ THEN\ SEQ_1\ FI'$
begin
$c\_true := generated\_masked\_sequence(SEQ_1, curr\_cond \wedge cond)$;
$c\_false := generated\_masked\_sequence(SEQ_1, curr\_cond \wedge \neg cond)$;
$curr\_cond := c\_true \vee \neg cond$;
end

$S =' GOTO\ S'$
begin
$source\_conds(S) := source\_conds(S) \bigcup \{curr\_cond\}$
$curr\_cond := false$
end

$S =' default'$
generate($'IF\ code(curr\_cond)\ THEN\ S\ FI'$);
end case

$generate\_masked\_statement := curr\_cond$;

end function

function generate_masked_sequence(SQ, AC): boolean;
begin
$curr\_cond := AC$;
for every S in SQ **do** $source\_conds(S) := \oslash$ **end for**
for every S in SQ in textual order **do**

$curr\_cond := generate\_masked\_statement(S, curr\_cond)$;
end for
end function

/* Main programm */
begin generate_masked_sequence(SEQ, C)
end

If-conversion Algorithmus funktioniert folgendermaßen:
Wenn S keine Sprunganweisung ist, so wird S im Form: *IF(cond) THEN S FI* überschrieben, wobei $cond = mask(S)$. Ist $cond = true$, so bleibt S unverändert. Alle Sprunganweisungen werden dabei einfach eliminiert.

Wenden wir den Ansatz auf Programm aus Abbildung 9 an:
Wir modifizieren das Programm, in dem eine zusätzliche Anweisung S' eingefügt wird. In S' wird eine Hilfsvariable V für die Bedingung $A(I) = 0$ eingefügt. Nach der Anwendung von if-conversion erhält man folgendes Ergebnis:

DO I=1,100
  S' : V=(A(I)=0)
  S_2 : IF (NOT(V)) THEN B(I)=1 FI
  S_3 : IF(V) THEN C(I)=A(I)+3 FI
END DO II

Somit sind alle Kontrollabhängigkeiten in Datenabhängigkeiten umgewandelt.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurden verschiedene Ansätze vorgestellt, die Vektorisierung eines Programms ermöglichen. Außerdem wurden die Bedingungen, unter welchen die Vektorisierung die Semantik des Programms nicht verletzt, erläutert. Auch wenn ein Programm sich nur teilweise vektorisieren lässt, trägt die Vektorisierung auf entsprechenden Architekturen zur Leistungssteigerung bei.

Literaturverzeichnis

[1] Zima/Chapmann. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.

Martin Thielen
Betreuer: Nicolas Fritz

Cg & GLslang: Programmiersprachen für Graphikkarten

Die Fähigkeiten moderner Graphikhardware sind in den letzten Jahren stark gestiegen. Dabei ist der Trend, Hardware-Bausteine mit fester Funktionalität durch programmierbare zu ersetzen. Cg und GLslang sind zwei Programmiersprachen für Graphikhardware, die als Reaktion auf diese neuen Entwicklungen entstanden sind und dem Programmierer deutlich mehr Komfort und Flexibilität als die bisherige Assembler-Programmierung bieten. In diesem Artikel sollen die grundlegenden Konzepte und Möglichkeiten dieser Sprachen erläutert werden.

Einführung

Bisher wurden die programmierbaren Teile der Graphikpipeline mit Assembler programmiert. Die steigenden Möglichkeiten heutiger GPU's und die sich daraus ergebende Komplexität machen ein effizientes Programmieren in Assembler unmöglich. Daraus resultierend ist es erforderlich, daß Abstraktionsniveau durch Einführung höherer Programmiersprachen zu steigern. Die Vorteile eines solchen Vorgehens liegen auf der Hand:

- An einem Programm vorgenommene Veränderungen können schneller getestet werden
- Der Code wird vom Compiler automatisch optimiert und Fehler, die bei „low-lewel"-Programmierung leicht auftreten, können vermieden werden.
- Das Verständnis für ein Programm, das in einer höheren Programmiersprache geschrieben wurde ist wesentlich besser, wodurch Anpassungen leichter vorgenommen werden können.
- Die Kompatibilität zu verschiedenen Systemen wird gesteigert.

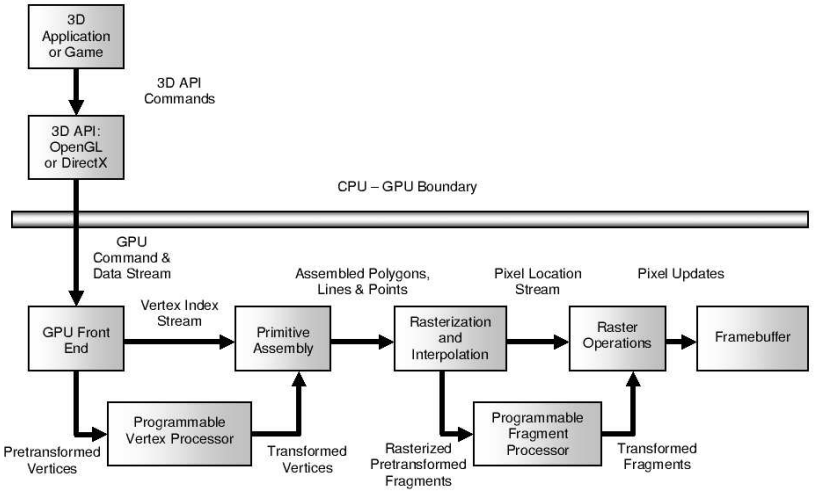
Im Rahmen dieses Arbeit werden nun zwei höhere Sprachen zur Programmierung von Grafikkarten vorgestellt. Beide Konzepte orientieren sich an den Programmiersprachen C/C++.

Abbildung 1 Echtzeit-Demo auf einer NVIDIA GeForce^{TM}FX, der Shader läßt die Oberfläche des Autos mit der Zeit rostiger werden. Quelle: [6]

Grundlagen

Heutige GPU's haben zwei programmierbare Prozessoren: vertex processor und fragment processor. Die Programme, die auf diesen Prozessoren laufen nennt man „vertex shader" und „fragment/pixel shader". Die Hardware arbeitet auf Basis von Datenströmen, d.h. der vertex shader wird für jeden Eckpunkt und der fragment shader für jedes Fragment im Datenstrom ausgeführt. Die Applikation muß dem vertex shader also ein Array von Eckpunkten bereitstellen, welches Daten enthält, die sich von Eckpunkt zu Eckpunkt ändern, wie z.B. Position und Normale. Die Ausgabe des vertex-shaders gelangt über den Rasterisierer, der die Polygone durch Pixel approximiert und interpoliert, zum fragment shader, der z.B. Position und Farbe eines Fragments benötigt. Der fragment shader muß die berechnete Farbe ausgeben, die dann an den „Framebuffer" weitergegeben wird.

Abbildung 2 Vereinfachte Darstellung des Datenflußes in einer GPU. Quelle: [5]



Cg (C for graphics)

1 Motivation

Die bisherige Programmierung mit Assembler kann die Anforderungen, die an moderne Graphikanwendungen gestellt werden, nicht mehr erfüllen. Cg ist eine neu entwickelte Programmiersprache, die versucht, den steigenden Möglichkeiten der programmierbaren Prozessoren gerecht zu werden. Sie orientiert sich an der Programmiersprache C, im Bezug auf Syntax und Semantik. Da die meisten Programmierer C kennen, bedeutet dies einen erheblichen Vorteil, da die Syntax nicht neu erlernt zu werden braucht. Cg folgt auch der Philosophie von C, hardware-nah zu sein. Dies ermöglicht das Schreiben von effizientem Code, da Datentypen und Operatoren eine direkte und offensichtliche Beziehung zu Hardware-Operationen haben. Der Vergleich(s.u.) verdeutlicht auf Anhieb, welche Vorteile Cg gegnüber herkömmlicher Assembler-Programmierung hat.

Cg wurde von NVIDIA in Zusammenarbeit mit Microsoft entwickelt. Die aktuelle Version ist 1.1 und wurde im März 2003 veröffentlicht. Cg kann zusammen mit den beiden verbreitetsten API's auf Graphikebene, DirectX und OpenGL, verwendet werden.

Assembler

```
...
DP3 R0, c[11].xyzx, c[11].xyzx;
RSQ R0, R0.x;
MUL R0, R0.x, c[11].xyzx;
MOV R1, c[3];
MUL R1, R1.x, c[0].xyzx;
DP3 R2, R1.xyzx, R1.xyzx;
RSQ R2, R2.x;
MUL R1, R2.x, R1.xyzx;
ADD R2, R0.xyzx, R1.xyzx;
DP3 R3, R2.xyzx, R2.xyzx;
RSQ R3, R3.x;
MUL R2, R3.x, R2.xyzx;
DP3 R2, R1.xyzx, R2.xyzx;
MAX R2, c[3].z, R2.x;
MOV R2.z, c[3].y;
MOV R2.w, c[3].y;
LIT R2, R2;
...
```

Cg

```
...
float4 cPlastic =
Cd * dot(Nf, normalize(L)) +
Cs * pow(max(0,
dot(Nf, normalize(H))), phongExp);
...
```

Vergleich: Assember ↔ Cg.
Quelle: [1]

2 Cg im Überblick

Anhand des folgenden einfachen Beispiels eines vertex shaders[1] soll der elementare Aufbau der Sprache erläutert werden:

```
  // Definiert die Eingaben der Applikation
  struct appin
  {
    float4 Position : POSITION;
    float4 Normal : NORMAL;
  }

  // Definiert die Ausgaben des vertex shaders
  struct vertout
  {
    float4 HPosition : POSITION;
    float4 Color0 : COLOR0;
  }


vertout main(appin IN,
             uniform float4x4 ModelViewProj,
             uniform float4x4 ModelViewIT,
             uniform float4 LightVec)

  {
    // Transformation der Position in den „Clip-Space“
    OUT.HPosition = mul(ModelViewProj, IN.Position);

    // Transformation der Normalen von „model-space“ zu „view-space“
    float3 normalVec = normalize(mul(ModelViewIT, IN.Normal).xyz);

    float3 lightVec = normalize(LightVec.xyz);

    // Berechnen des Vektors in Richtung des halben Winkels
    float3 eyeVec = float3(0.0, 0.0, 1.0);
    float3 halfVec = normalize(lightVec + eyeVec);

    // Berechnen des diffusen Anteils
    float diffuse = dot(normalVec, lightVec);

    // Berechnen des spiegelnden Anteils
    float specular = dot(normalVec, halfVec);
```

[1] übernommen aus [2]

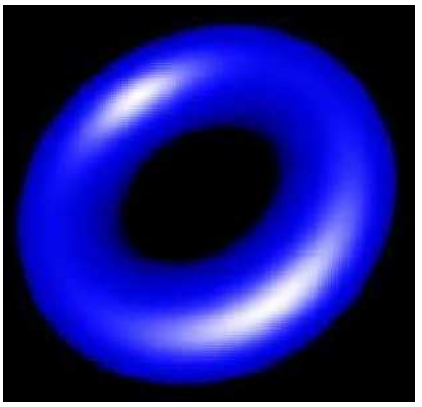
```
    // Die Funktion lit() ist eine Standardfunktion die zur
    // schnellen Lichtberechnung verwendet werden kann.
    // Die y-Komponente des resultierenden 4-dimensionalen Vektors
    // enthält den diffusen Koeffizient, die z-Komponente den
    // spiegelnden. x- und w-Komponente sind beide auf 1.0 gesetzt.
    float4 lighting = lit(diffuse, specular, 32);

    // Diffus: blau, spiegelnd: weiß
    float3 diffuseMaterial = float3(0.0, 0.0, 1.0);
    float3 specularMaterial = float3(1.0, 1.0, 1.0);

    // Kombination der diffusen und spiegelnden Anteile
    // zur endgültigen Farbe des Eckpunktes
    OUT.Color0.rgb = lighting.y * diffuseMaterial
    + lighting.z * specularMaterial;
    OUT.Color0.a = 1.0;

    return OUT;
}
```

Abbildung 3 Beispielergebnis unter Verwendung des vertex-shaders. Quelle: [2]



An diesem Beispiel wird deutlich, daß die Syntax von Cg an C angelehnt ist. Allerdings treten auch hier schon einige Besonderheiten auf, auf die im folgenden eingegangen wird.

2.1 Schnittstellen in der Graphikpipeline

Die Struktur am Anfang des Beispiels dient zur Kommunikation der Applikation mit dem vertex shader:

```
// Definiert Eingaben der Applikation
struct appin
{
  float4 Position : POSITION;
  float4 Normal : NORMAL;
}
```

Sie enthält Position und Normale eines Vektors. Da sich diese Daten naturgemäß pro Eckpunkt ändern, müssen sie von der Applikation mit jedem Eckpunkt im Eingabestrom übergeben werden. Die Schnittstelle zwischen Anwendung und vertex shader wird über vordefinierte Namen („binding semantics“) wie zum Beispiel „POSITION“, „TANGENT“, „NORMAL“, „TEXCOORD2“ realisiert. Diese sind direkt gebunden an physikalische Register in der Graphikkarte oder API-Resourcen. Eine solche Struktur wird auch „connector“ genannt.

Die zweite Struktur stellt das Interface zwischen vertex-shader und fragment shader dar:

```
// Definiert die Ausgaben des vertex shaders
struct vertout
{
  float4 HPosition : POSITION;
  float4 Color0 : COLOR0;
}
```

Sie enthält die Position (in homogenen Koordinaten) und Farbe eines Eckpunktes nach der Transformation durch den vertex shader.

2.2 Datentypen

Als Standarddatentypen in Cg sind `int`[2], `float`, `half`[2], `fixed`[2] und `bool`. Der Datentyp `int` ist dabei nur aus Kompatibilitätsgründen zu C integriert, denn heutige Graphikprozessoren unterstützen keine Integer-Datentypen. Mit den Typen `half` und `fixed` können Operationen, die weniger genaue Präzision erfordern effizienter ausgeführt werden. Allerdings muß die Hardware diese Typen unterstützen.

Desweiteren können Strukturen (structs) und Arrays wie in C definiert werden, jedoch muß bei diesen die Array-Schreibweise verwendet werden. Die aktuelle Version von Cg unterstützt noch keine Pointer, da Indirekte Adressierung (Dereferenzierung von Pointern) bei heutigen Graphikkarten nur in sehr beschränktem Maße verfügbar ist. Deshalb gibt es, anders als C, eine strikte Unterscheidung zwischen Arrays und Pointern. Dies wird auch daran deutlich, daß Arrays immer „per value“ übergeben werden, d.h. bei jeder Übergabe/Zuweisung komplett kopiert werden, sofern der Compiler keine Optimierung vornehmen kann.

[2]nicht in allen Profilen voll unterstützt (vgl. Abschnitt 3.2 Seite 84)

Wie in dem folgendem Codeausschnitt deutlich wird, besitzt Cg graphik-spezifische Datentypen:

```
vertout main(appin IN,
             uniform float4x4 ModelViewProj,
             uniform float4x4 ModelViewIT,
             uniform float4 LightVec)
```

Dies sind Vektoren und Matrizen bis zur Dimension 4 sowie „sampler“:

- floatN (1 <= N <= 4)
  N-dimensionaler Vektor, der Fließkommazahlen enthält,
- boolN (1 <= N <= 4)
  N-dimensionaler Vektor, der Boole'sche Werte enthält,
- floatNxM (1 <= N,M <= 4)
  Matrix mit N Spalten und M Zeilen, die Fließkommazahlen enthält
- sampler, sampler1D-3D, samplerCube, samplerRect[3]
  Sampler stellen eine Schnittstelle für Texturen dar; über eine Vielzahl von eingebauten Funktionen kann hiermit der Zugriff auf Texturen erfolgen.

Diese Datentypen wurden integriert, um dem Hauptanwendungsgebiet (Graphikprogrammierung) gerecht zu werde, und um dem Programmierkomfort zu erhöhen.
Typumwandlungen werden implizit wie in C vorgenommen. Ausnahme: Bei Konstanten, die keine explizite Typangabe über ein Suffix[4] haben, erfolgt keine Typumwandlung.
Eine explizite Typumwandlung ist mit Hilfe des cast-operators „(<type>)“ möglich.

2.3 Schlüsselwörter

Die GPU arbeitet anders als die CPU auf Eingabeströmen, d.h. der Vertex-Shader wird für jeden Vertex im Strom einmal ausgeführt und der Fragment-Shader für jedes Fragment. Parameter von Funktionen, oder globale Variablen die sich nicht mit jedem Element im Datenstrom ändern, können aus Effizienzgründen mit dem Schlüsselwort *uniform* gekennzeichnet werden. Die entsprechenden Variablen werden dann in einem anderen Bereich des Speichers der Graphikkarte gesichert, als die veränderlichen Daten. Dies ist im letzten Codeausschnitt für Parameter erfolgt, die für alle Eckpunkte identisch sind:
ModelViewProj (Kombination aus „modelview-“ und Projektions-Matrix),
ModelViewIT (invertierte transponierte der Matrix *ModelViewProj*) und
Lightvec (Vektor, der die Position der Lichtquelle angibt)

Da indirekte Adressierung nur in sehr begrenztem Maße auf aktueller Graphikhardware verfügbar ist, unterstützt Cg einen speziellen Mechanismus zur Spezifizierung von Parametern als Ein- oder Ausgabe, genannt „call-by-value-result“.

[3] Wird nur in OpenGL-Profilen unterstützt (vgl. Abschnitt 3.2 Seite 84)
[4] f für float, h für half, x für fixed

Dazu gibt es folgende Bezeichner:

- in (default) bezeichnet einen Eingabe-Parameter.
- inout / in out bezeichnet ein Argument, das sowohl Ein- als auch Ausgabe-Parameter ist.
- out bezeichnet einen Ausgabe-Parameter

Die Definition der Parameter des vertex Shaders hätte also auch folgende Form haben können[5]:

```
void main(appin IN, out vertout OUT
          uniform float4x4 ModelViewProj,
          uniform float4x4 ModelViewIT,
          uniform float4 LightVec)
```

2.4 Operatoren

Standardoperatoren

Cg unterstützt arithmetische Operatoren (+, -, *, /), logische Operatoren (&&, ||, !), Vergleichsoperatoren (<, <=, >, >=, !=, ==), Zuweisungsausdrücke wie (+=, -=), Increment/Dekrement (++, - -) und den Komma-Operator (,) Arithmetische und logische Operatoren sowie der Konditional-Operator (?:) können auch auf Vektoren angewendet werden und werden dabei Komponentenweise ausgeführt. Dies wird leicht an den folgenden Beispielen deutlich:

```
float4(x1,y1,z1,w1) + float4(x2,y2,z2,w2)
⟶ float(x1+x2,y1+y2,z1+z2,w1+w2)

a * float4(x,y,z,w)
⟶ float4(a*x,a*y,a*z,a*w)

bool4(true,false,false,true) || bool4(false,true,false,true)
⟶ bool4(true,true,false,true)

( float3(1.0,2.0,3.0) < float3(3.0,2.0,1.0) )
? float3(1.0,1.0,1.0) : float3(0.0,0.0,0.0)
⟶ float3(1.0,0.0,0.0)
```

Hierbei müssen natürlich die Dimensionen der Vektoren passend sein. Desweiteren ist zu beachten, daß, anders als in C/C++, Seiteneffekte immer auftreten. Bei einem Ausdruck A && B werden z.B. Seiteneffekte des Ausdrucks A und des Ausdrucks B auftreten, selbst wenn A zu „false“ auswertet.

[5] wobei „`vertout OUT;`“ und „`return OUT;`“ im Funktionsrumpf entfallen

Swizzle-Operator

Zusätzlich besitz Cg noch den praktischen „Swizzle-Operator“ (.), der in folgendem Codeausschnitt verwendet wird:

```
// Transformation der Normalen von „model-space“ zu „view-space“
float3 normalVec = normalize(mul(ModelViewIT, IN.Normal).xyz);
```

Er stellt eine effiziente Möglichkeit dar, Vektoren umzuordnen und neue Vektoren zu erzeugen. Die Komponenten des Vektors werden durch x,y,z,w oder r,g,b,a repräsentiert. Dabei dürfen dei einzelnen Komponenten auch öfters vorkommen, oder wegfallen. Mischformen zwischen x,y,z,w und r,g,b,a sind allerdings nicht erlaubt. Einige weitere Beispiele, die die Möglichkeiten verdeutlichen:

```
float4(1.0,2.0,3.0,4.0).xxz
⟶ float3(1.0,1.0,3.0)

float4(1.0,2.0,3.0,4.0).w
⟶ 4.0

float2(1.0,2.0).rrg
⟶ float3(1.0,1.0,2.0)

float d = 1.5; d.xxx;
⟶ float3(1.5,1.5,1.5)
```

Write-Mask-Operator

Der „Write-Mask-Operator“ (.) ist das Gegenstück zum Swizzle-Operator, er dient zum Überschreiben einer oder mehrerer Komponenten eines Vektors:

```
// Kombination der diffusen und spiegelnden Anteile
// zur endgültigen Farbe des Eckpunktes
OUT.Color0.rgb = lighting.y * diffuseMaterial
+ lighting.z * specularMaterial;
OUT.Color0.a = 1.0;
```

2.5 Standardfunktionen

Durch die Bereitstellung einer Vielzahl von Standardfunktion ist für „shading“ zugeschnittene Funktionalität in CG integriert, ohne die Universalität (vgl. Abschnitt 3.1 Seite 83) der Sprache an sich zu beeinträchtigen. Im vorgestellten Beispiel werden folgende Funktionen verwendet:

- *mul*: Multiplikation von Matrix mit Vektor oder Matrix mit Matrix
- *normalize*: Normalisieren eines Vektors
- *dot*: Berechnen des Skalarprodukts zweier Vektoren
- *lit*: Berechnet Beleuchtungsvektor(Blinn-ähnlich)

Weitere in Cg eingebaute Funktionen sind zum Beispiel:

- *determinant*: Berechnen der Determinante einer Matrix
- *all*: Gibt „true“ zurück, falls jeder Wert des Vektors x „true“ ist, ansonsten „false“;
- *reflect(i,n)*: Berechnet für einen Eingangsstrahl in Richtung i und einer Normalen n den Reflektionsstrahl.

Standardfunktionen sollten nicht nur aus Bequemlichkeit sondern auch aus Effizienzgründen benutzt werden, da sie „hardware-nah“ implementiert sind und für zukünftige Hardware-Architekturen optimiert werden können. Dies kann dann für ein älteres Programm von Vorteil sein, wenn es zur Laufzeit kompiliert wird. (vgl. Abschnitt 3.3 Seite 85)

2.6 Weitere Eigenschaften und Unterschiede zu C

Die Konstrukte **for, do, while, if/else, break, continue**[6] werden von Cg unterstützt, **goto** und **switch** dagegen nicht. Präprozessordirektiven wie **#include #define #ifdef** sind verfügbar. Bitoperationen werden in der aktuellen Version noch nicht unterstützt. Es ist möglich eigene Funktionen zu implementieren, allerdings muß gegenwärtig noch auf rekursive Funktionsaufrufe verzichtet werden.
Um zukünftigen Hardware-Architekturen gerecht werden zu können, sind für Cg alle Schlüsselwörter von C und C++ reserviert. So können später Konzepte wie Pointer leicht integriert werden, wenn die entsprechene Hardwarefunktionalität vorhanden ist.

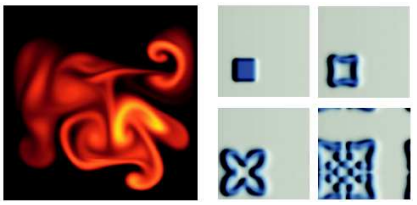
[6] in einigen Profilen muß die Anzahl der Iterationen während der Kompilierung ermittelbar sein

3 Design-Entscheidungen und Konzepte

3.1 Universalität

Bei der Entwicklung einer Programmiersprache stellt sich zwangsläufig die Frage, ob die Sprache universell, oder für einen speziellen Bereich konzipiert sein soll. Bei einer spezialisierten Sprache ist es möglich, Optimierungen vorzunehmen, die zum Beispiel die Effizienz steigern. Der Hauptanwendungsbereich von Cg ist sicherlich die Graphikprogrammierung. Effizienz ist in dieser Domäne natürlich sehr entscheidend. Dennoch haben sich die Entwickler von Cg dazu entschlossen, eine universelle Sprache zu kreieren und damit die Möglichkeit eröffnet, auch Anwendungen zu schreiben, die normalerweise der CPU vorbehalten sind (vgl. Abbildung 4). Dabei ist es ihnen gelungen, den vermeintlichen Nachteil, keine Optimierungen vornehmen zu können, dadurch auszugleichen, daß Optimierungen ausgelagert in den mitgelieferten Standardfunktionen integriert sind.

Abbildung 4 links: Strömungssimulation, rechts: reaction-diffusion Simulation auf einer GPU unter Verwendung von Cg. Quelle: [6]; weitere universelle Anwendungen von GPU's können auf http://www.gpgpu.org/ gefunden werden.



Ein weiterer Vorteil von Cg ist außerdem die klare Verbindung zur Hardware. Nur so ist für den Programmierer ersichtlich, wie effizient sein Code ist. Desweiteren kann die für eine bereichs-spezifische Sprache notwendige Abstraktion dazu führen, daß sie den Anforderungen des Benutzers nicht entsprechen kann. Zum Beispiel ist es nicht ohne Umstände möglich, mit „RenderMan" das Lichtmodell von OpenGL zu unterstützen, da OpenGL verschiedene Farben für spiegelnd- und diffus-reflektierendes Licht verwendet. Auch kann zusätzlicher Aufwand durch den Compiler und Laufzeit-Umgebung nötig werden, wenn die verwendete Hardware nicht zur Abstraktion der Sprache paßt.
Cg besitzt eine weitere universelle Funktionalität[7], die der einer virtuellen Basisklasse in C++ ähnelt: Der Programmierer erhält die Möglichkeit, ein „interface" zu definieren, welches die gemeinsamen Eigenschaften verschiedener Strukturen zusammenfaßt. Wie dieses Beispiel[8] zeigt, können mit einem solchen interface separate Oberflächen- und Licht-Shader implementiert werden:

[7] befindet sich noch in der Entwicklung und ist in der aktuellen Version noch nicht integriert
[8] übernommen aus [6]

```
// Deklaration des Light-Interface
interface Light
{
  float3 direction(float3 from);
  float4 illuminate(float3 p, out float3 lv);
}

// Deklaration für punktförmige Lichtquellen
struct PointLight : Light
{
  float3 pos, color; float3 direction(float3 p)
  { return pos - p; }
  float3 illuminate(float3 p, out float3 lv)
  { lv = normalize(direction(p)); return color; };
}

// Deklaration für gerichtete Lichtquellen
struct DirectionalLight : Light
{
  float3 dir, color;
  float3 direction(float3 p)
  { return dir; }
  float3 illuminate(float3 p, out float3 lv)
  { lv = normalize(dir); return color; };
}

// Hauptprogramm (Oberflächen-Shader)
float4 main(appin IN, out float4 COUT, uniform Light lights[])
{
  ...
  for (int i=0; i < lights.Length; i++)
  {
    Cl = lights[i].illuminate(IN.pos, L);
    color += Cl * Plastic(texcolor, L, Nn, In, 30);
  }
  COUT = color;
}
```

3.2 Profile

Cg kann sowohl mit OpenGL als auch DirectX verwendet werden. Um die unterschiedlichen Ansätze dieser API's und die verschiedenen Möglichkeiten ihrer Versionen (es unterstützen zum Beispiel nicht alles DirectX Versionen if/else im vertex processor) nutzen zu können, verwendet Cg Profile. Da die Fähigkeiten des vertex processors und die des fragment processors unterschiedlich sind, kommen auch hier verschiedene Profile je nach Prozessor zum Einsatz. In einem Profil wird nur der Teil der gesamten Sprache unterstützt, den die Hardware und API zulassen. Cg unterstützt momentan 18 verschiedene Profile. Natürlich schränkt das Konzept der Profile die Portabilität ein, dafür können

so aber spezielle Fähigkeiten der Hardware ausgenutzt werden und somit zum Beispiel die Leistung in Graphikanwendungen erhöht werden. Außerdem gibt es in Cg die Möglichkeit, Funktionen nicht nur nach Typ und Anzahl der Parameter zu überladen, sondern auch nach Profilen. Dadurch kann die Portabilität eines Programms erreicht und zugleich Optimierungen auf bestimmten Hardware-Architekturen ermöglicht werden. Einige Beschränkungen die Profile zum Beispiel aufgrund von begrenzten Fähigkeiten der Hardware auferlegen können:

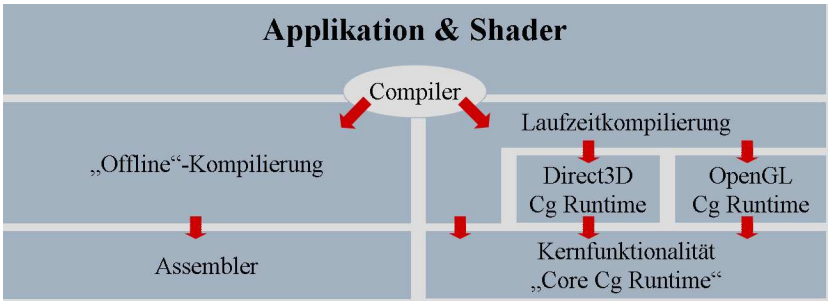
- Eingschränkte Benutzung von Arrays
- Keine Unterstützung von Samplern
- Implementierung des Typs *int* als *float*

Ein weiterer Vorteil der Profil-Struktur ist, daß Restriktionen in der Zukunft leicht aufgehoben werden können, sobald Hardware und API es zulassen.

3.3 Cg-Runtime und Assembler Interface

Das Kompilieren und „Linken“ eines Cg-Programms an eine Applikation kann entweder zur „Compile-Zeit“ oder zur Laufzeit der Anwendung (mit der „Cg-Runtime“) geschehen. Beide Möglichkeiten haben verschiedene Vor- und Nachteile. Die Ausgabe des Cg-Compilers in Assembler-Code stellt eine Schnittstelle (Assembler Interface) dar, über die der Programmierer die Möglichkeit hat, Code von Hand zu optimieren. Sie kann nur dann verwendet werden, wenn die Einbindung des Cg-Programms zur Zeit der Kompilierung der Applikation erfolgt. Desweiteren erhöht sich die Ladezeit der Applikation bei einer Laufzeit-Compilierung. Dafür kann die Anwendung aber von zukünftigen Verbesserungen des Compilers und der Hardware profitieren, ohne das dafür Änderungen am Cg-Programm notwendig sind. Auch ist die Übergabe der Parameter mit Cg-Semantik wesentlich leichter und weniger fehleranfällig als über Assembler.

Abbildung 5 Laufzeit- und „Offline“- Kompilierung, Struktur der Cg Runtime.



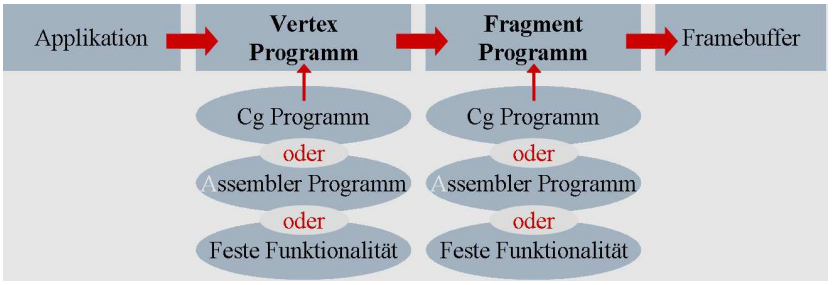
Die Cg-Runtime stellt eine API dar, und ist in drei Module aufgeteilt (vgl. Abbildung 5):

- Kernfunktionalität
- Spezielle OpenGL Funktionalität
- spezielle Direct3D Funktionalität

3.4 Modularität

In Cg sind die jeweiligen Programme für vertex und fragment processor eigenständig. Dieses Modell ermöglicht aufgrund seiner direkten Abbildung auf die Hardware eine bessere Abschätzung der Performance und eine partielle Integration: Ein in Cg implementierter vertex-shader kann unter gleichzeitigem Gebrauch der festen Funktionalität der Graphikpipeline für den fragment shader verwendet werden. Genauso ist es natürlich auch möglich, einen Cg-fragment-shader, und einen in Assembler geschriebenen vertex-shader zu kombinieren. Dies gestattet eine stufenweise Anpassung bestehender Anwendungen und schnelleres Testen eines neuen Shaders.

Abbildung 6 Partielle Integration von vertex und fragment shader.



GLslang (The OpenGL Shading Language)

1 Motivation

Im folgenden soll GLslang vorgestellt werden, eine weitere Sprache, die entwickelt wurde, um Graphikkartenprogrammierung zu vereinfachen und zu verbessern. Auch hier ist das Konzept, womit dieses erreicht werden soll, die Assemblerprogrammierung im Bereich der programmierbaren Teile der Graphikhardware (vertex und fragment processor) durch „High-level-Programmierung“ zu ersetzen. Sie hat viele Gemeinsamkeiten mit Cg und basiert ebenfalls auf C. Dennoch gibt es Unterschiede in verschiedenen Bereichen, auf die im folgenen eingegangen werden soll.

2 Einführendes Beispiel

Dieses Beipiel[9] mit einem einfachen vertex und fragment shader, zeigt wie mit einer Farbverlaufstextur verschiedene Texturen weich ineinander übergeblendet werden können. (vgl. Abbildung 7)

Vertex shader:

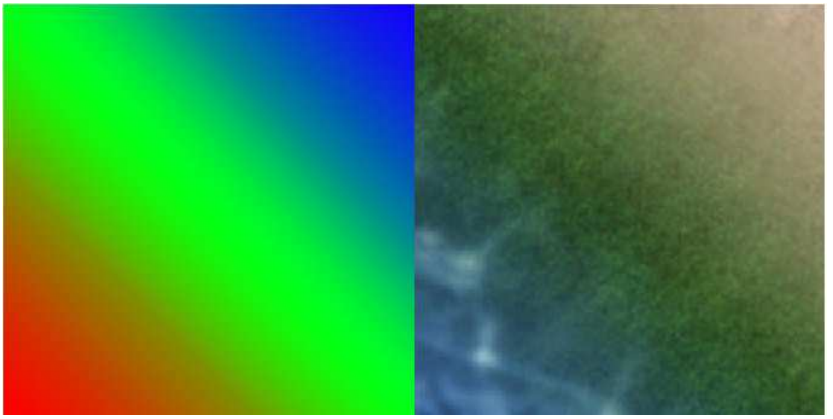
```
uniform vec4 GlobalColor;

void main(void)
{
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  gl_FrontColor = gl_Color * GlobalColor;
  gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

Im vertex shader wird als erstes die globale Variable „GlobalColor“ definiert. Das *uniform* Schlüsselwort wird in GLslang etwas anders verwendet als in Cg: Es

[9] übernommen aus [4]

Abbildung 7 Weiche Überblendung von Texturen mit Hilfe von GLslang-shadern. Quelle: [4]



bildet die Schnittstelle zwischen Anwendung, OpenGL und einem Shader. Der Wert einer Variable, die als *uniform* gekennzeichnet wird, bleibt dabei konstant. Die Kommunikation zwischen Shadern untereinander und mit der Graphikpipeline erfolgt, anders als in Cg über eingebaute Variablen(gl_Position, gl_Color, gl_FrontColor, gl_TexCoord[0] und Attribute(gl_MultiTexCoord). Dabei gibt es Variablen, die nur in einem der shader verfügbar sind
(z.B. gl_Position: der vertex shader muß die homogenen Koordinaten des Eckpunktes zuweisen) oder Attribute, auf die jeder Shader nur lesend zugreifen kann (z.B. gl_MultiTexCoord0: enthält die Texturkoordinaten aus der Graphikpipeline). Weitere Schüsselwörter sind *attribute* zur Übergabe von sich pro Eckpunkt verändernden Attributen aus OpenGL an den vertex shader, und *varying* als Schnittstelle zwischen vertex und fragment shader für interpolierte Daten.

Fragment shader:

```
uniform sampler2D Texture0;
uniform sampler2D Texture1;
uniform sampler2D Texture2;
uniform sampler2D Texture3;

void main(void)
{
  vec2 TexCoord = vec2( gl_TexCoord[0] );
  vec4 RGB = texture2D( Texture0, TexCoord );
  gl_FragColor = texture2D(Texture1, TexCoord) * RGB.r
  + texture2D(Texture2, TexCoord) * RGB.g
  + texture2D(Texture3, TexCoord) * RGB.b;
}
```

Über die Standardfunktion *texture2D* werden *RGB* die Koordinaten des Eckpunktes auf einer mit einem Farbverlauf gefüllten Textur zugewiesen. Die end-

gültige Farbe (gl_FragColor) wird durch Gewichtung dreier anderer Texturen nach den Farbanteilen an einer bestimmten Position ermittelt (vgl. Abbildung 7).
In der Zeile `vec2 TexCoord = vec2( gl_TexCoord[0] );` erfolgt eine explizite Typumwandlung über einen sogenannten Konstruktor. Hier werden zum Beispiel der x- und y-Wert des 4-dimensionalen Vektors *gl_ TexCoord[0]* an den 2-dimensionalen vektor *TexCoord* übergeben. Impliziete Typumwandlung wie in Cg oder C gibt es nämlich in GLslang nicht.

3 Unterschiede zu Cg

Sofern noch nicht im letzten Abschnitt behandelt, werden in diesem Abschnitt Unterschiede zwischen Cg und GLslang erläutert.

3.1 Unterstützte API und Profile

Einer der größten Gegensätze ist sicherlich die unterstützte API: GLslang kann „nur“ mit OpenGL verwendet werden, wohingegen Cg bekannterweise auch mit DirectX zusammenarbeitet. OpenGL besitzt deshalb auch keine Profile, wie Cg. GLslang besteht allerdings strenggenommen aus zwei Sprachen, einer für den vertex processor und einer für den fragment processor. Dies ist zum Beispiel an den eingebauten Variablen bemerkbar, die für vertex und fragment shader unterschiedlich sind.

3.2 Maschinencode

Anders als Cg produziert der Compiler von GLslang keinen Assembler-Code sondern Maschinencode. Außerdem ist er komplett in OpenGL integriert, was den Vorteil hat, daß auch ältere Anwendungen bei zukünftigen Treibern den neuesten Compiler benutzen. Dafür hat der Programmierer allerdings nicht die Möglichkeit, wie bei Cg den Code von Hand zu optimieren und kann nicht garantieren, daß eine bestimmte Version eines Compilers benutzt wird.

3.3 Flexibilität

GLslang ist nicht so vielseitig und universell ausgerichtet, wie Cg. Zum Beispiel gibt es keine Funktionalität die dem interface-Konzept aus Cg ähnelt. Außerdem bietet Cg bei der Compilierung mehr Flexibilität, da hier sowohl die Möglichkeit der Compilierung zur Laufzeit besteht, als auch die zur Entwicklungzeit mit manueller Optimierung.

Zusammenfassung und Ausblick

In diesm Artikel wurden die beiden Programmiersprachen Cg und GLslang vorgestellt. Mit ihnen lassen sich effizient und übersichtlich Shader für Echtzeitanwendnungen schreiben. Sie verfolgen einen sehr ähnlichen Ansatz, haben aber im Detail unterschiedliche Konzepte. Es bleibt abzuwarten welche der beiden Sprachen sich im Endeffekt durchsetzen wird, oder ob beide koexistieren können. Ich denke, daß sich das Konzept der Programmierbarkeit von Graphikkarten in Zukunft noch weiter durchsetzen wird, und das Sprachen wie Cg oder GLslang immer mehr Bedeutung gewinnen, gerade in der Spieleindustrie. Es scheint jedenfalls so, als ob die klassische Assemblerprogrammierung im Bereich der Graphikhardware mit Einführung dieser Sprachen in Zukunft ausgedient hat.

Literaturverzeichnis

[1] NVIDIA Corp. *Cg - Getting Started with Cg*, 2002.

[2] NVIDIA Corp. *Cg Toolkit - User's Manual*, 2003.

[3] NVIDIA Corp. *Cg Whitepaper*, 2003.

[4] John Kessenich, Dave Baldwin, and Randi Rost. *The OpenGL Shading Language*, 2003.

[5] Christopher Lux. Hauptseminar Cg. *TU Ilmenau*, 2003.

[6] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *Siggraph*, 2003.

[7] Sascha Willems. glSlang - OpenGL Shading Language. `http://www.delphigl.de/dgl/glSlang.html`. . . .

Robert Müller
Betreuer: Philipp Lucas

Shader Metaprogramming

Shader Metaprogramming ermöglicht das Schreiben von Shader-Programmen direkt in bereits bekannten Hochsprachen, hier konkret C++. Durch die Verwendung der syntaktischen Konstrukte von C++ wird die Entwicklung der Sprache stark beschleunigt sowie der Programmierer entlastet, da er keine weitere Sprache erlernen muß.

1 Einführung

Im Laufe der letzten Jahre ist die in Standard-PCs eingesetzte Grafikhardware immer leistungsfähiger geworden. Die Entwicklung hat nun einen Punkt erreicht, an dem aus einfachen Hardware-Beschleunigern mit fest verdrahteter Logik programmierbare, flexible Coprozessoren werden. Derzeit existieren verschiedene Low- und High-Level-Ansätze, diese Prozessoren zu programmieren. Während die assemblerähnlichen Low-Level-Sprachen lediglich relativ einfache Compiler erfordern, ist die Entwicklung einer neuen High-Level-Shading-Sprache eine Aufgabe, die viel Zeit und Ressourcen in Anspruch nimmt.
Ein neuer Ansatz, das Shader Metaprogramming, kann die Entwicklungszeit einer solchen Sprache extrem verkürzen, indem eine bereits vorhandene Hochsprache mitsamt ihrer syntaktischen Konstrukte genutzt wird, um Shaderprogramme zu erstellen. Darüberhinaus wird dem Programmierer die Arbeit erleichtert, da dieser seine Shaderprogramme in einer bereits erlernten Hochsprache formulieren kann und keine weitere Sprache lernen muß.
Am Computer Graphics Lab der School of Computer Science an der Universität von Waterloo in Kanada wird derzeit eine Bibliothek entwickelt, die genau dies unter C++ ermöglichen soll. [1] Im Folgenden soll untersucht werden, welche Möglichkeiten diese Bibliothek bietet, wie sie diese umsetzt und wo sich ihre Grenzen befinden. Dabei verwenden wir die LibSH in der Version vom 19.11.2003.

2 Das Konzept

Die Idee beim Shader-Metaprogramming mit der LibSH ist folgende: Anstatt in einer dedizierten Shading-Sprache werden Shader-Programme direkt in C++ geschrieben. Hierzu stellt die Library einen Satz API-Aufrufe bereit. Jeder dieser Aufrufe wird als Befehl in der Shader-Sprache interpretiert. So kann man zum Beispiel mit

```
SH_FOR((i=0.0),(i<5.0),(i=i+1.0))
```

eine for-Schleife einleiten, die i von 0.0 bis 4.0 in ganzzahligen Schritten hochzählt. Der syntaktische Aufbau des Befehls lehnt sich dabei ganz bewußt an die aus C/C++ bekannte Syntax an, die dem Programmierer in der Regel bereits vertraut ist.
Bei SH_FOR handelt es sich um eine Präprozessordirektive, die die Statements in den Klammern an die entsprechenden LibSH-Funktionen weiterreicht.

Der Vorteil dieser Vorgehensweise ist vor allem, daß der Parser der LibSH wesentlich einfacher aufgebaut ist, als dies bei einem dedizierten Compiler der Fall wäre. Auf die Verwendung von Stringmanipulationen kann komplett verzichtet werden, stattdessen kann man die API-Aufrufe als einen Strom von Tokens interpretieren, aus denen sich direkt ein Syntaxbaum erstellen läßt.
Durch die Verwendung von Operatorüberladung auf C++-Ebene lassen sich arithmetische und boolesche Ausdrücke der Shading-Sprache schon zur Übersetzungszeit des Hauptprogramms sowohl parsen als auch auf Typkorrektheit prüfen. Die Prüfung der Typkorrektheit wird dabei vom C++-Compiler selbst vorgenommen, während das Parsen durch die Generierung von Syntaxbäumen aus den Ausdrücken bewerkstelligt wird.

Für den (C++-)Programmierer, der sich der LibSH bedient, um seine Shader-Programme zu erstellen, ergeben sich ebenfalls Vorteile:
Zunächst einmal kann er bei der Erstellung der Shader in seiner bereits vertrauten Programmierumgebung bleiben. Die externe Entwicklung und Übersetzung von Shader-Programmen ist nicht notwendig, somit ist der Shader quasi Teil des Hauptprogramms.
Zum zweiten werden die eigentlichen Shader-Programme erst zur Laufzeit des Hauptprogramms on-the-fly aus der LibSH-Syntax in den eigentlichen Maschinencode übersetzt. Das ermöglicht die flexible Verwendung ein und desselben Shader-Programms auf verschiedenen Hardware-Plattformen, die unter Umständen sogar zur Laufzeit des Programms gewechselt werden können. Die einzige Bedingung ist, daß ein geeignetes Backend zur Verfügung steht, das die LibSH-Programme für die Zielhardware übersetzen kann.

3 Ein erster Shader in der LibSH

Die Erstellung von Shader-Programmen mit der LibSH ist nicht sonderlich kompliziert. Die Vorgehensweise unterscheidet sich nicht sehr von der beim Erstellen normaler Funktionen in C++. Zur Verdeutlichung der Verwendung von Shadern in der LibSH folgt hier ein kurzes Beispiel:

```
 1 void DiffuseShader::init()
   {
     ShShader diffuse_vertex = SH_BEGIN_VERTEX_PROGRAM {
       ShInputNormal3f normal;
 5     ShInputPosition4f p;

       ShOutputPoint4f ov;
       ShOutputNormal3f on;
       ShOutputVector3f lvv;
10     ShOutputPosition4f opd;

       opd = ( mv | projection ) | p;
       on = normalize( mv | normal );
       ov = -normalize( mv | p );
15     lvv = normalize( lightPos - ( mv | p )(0,1,2) );
     } SH_END_PROGRAM;


     ShShader diffuse_fragment = SH_BEGIN_FRAGMENT_PROGRAM {
20     ShInputVector4f v;
       ShInputNormal3f n;
       ShInputVector3f lvv;
       ShInputPosition4f p;

25     ShOutputColor3f out;
       out = color(0,1,2) * dot(normalize(n), normalize(lvv));
     } SH_END_PROGRAM;
28 }
```

Hier werden ein Vertex Shader und ein Fragment Shader definiert, die ein diffuses Shading einer Oberfläche berechnen. Bei den kursiv hervorgehobenen Wörtern handelt es sich um globale Parameter. (Siehe hierzu Abschnitt 3.3.)

3.1 Die Form eines LibSH-Shaders

Eingeleitet wird ein Shader-Programm mit `SH_BEGIN_VERTEX_PROGRAM` bzw. `SH_BEGIN_FRAGMENT_PROGRAM`. *(Zeilen 3, 19)*
Direkt am Anfang eines Shaders werden seine Ein- und Ausgabeattribute spezifiziert. *(Zeilen 4-10, 20-25)* Dies entspricht der Signatur einer Funktion in C++. Auffallend ist, daß es kein `return`-Statement gibt. Zur Rückgabe bzw. Weiterreichung von Berechnungsergebnissen werden die Ausgabeattribute verwendet, indem ihnen einfach Werte zugewiesen werden. *(Zeilen 12-15, 26)*
Auf die Deklaration der Ein- und Ausgabeattribute folgt das eigentliche Shader-Programm. In beiden Fällen werden nur ein paar einfache mathematische Operationen durchgeführt. *(Zeilen 12-15, 26)*
Beendet wird das Shader-Programm durch ein `SH_END_PROGRAM`-Statement. *(Zeilen 16, 27)* Dieses bewirkt unter anderem auch, daß das Programm jetzt in den für die Zielplattform erforderlichen Code übersetzt wird. Der C++-Scope, der das jeweilige Programm umschließt, ist nicht wirklich notwendig,

da die Makros `SH_BEGIN_VERTEX_PROGRAM` bzw. `SH_BEGIN_FRAGMENT_PROGRAM` und `SH_END_PROGRAM` selbst einen Scope erzeugen, erhöht aber die Lesbarkeit und Übersicht.

3.2 Die Verwendung der Ein- und Ausgabe-Attribute

Der Vertex Shader und der Fragment Shader sind hintereinander in einer Pipeline angeordnet. Die Daten werden den Shadern als Pakete zugeführt, und die Shader versenden ihre Ausgabedaten auch wieder als Pakete. Der Fragment Shader muß daher Pakete in genau der Größe akzeptieren, die der Vertex Shader ausgibt. Wie er die Pakete interpretiert, ist dabei ihm überlassen. Beispielsweise kann ein vom Vertex Shader ausgegebener dreidimensionaler Punkt durchaus als eine Sequenz aus drei eindimensionalen Koordinaten eingelesen werden.
Darüberhinaus gelten für die Ausgabeattribute der Vertex und Fragment Shader ein paar Regeln. So muß das letzte Ausgabeattribut des Vertex Shader die Position des Vertex auf der Anzeige in homogenen Koordinaten sein. Der Fragment Shader darf als einziges Ausgabeattribut einen Farbwert vom Typ *ShColor3f* oder *ShColor4f* ausgeben.

3.3 Attribute und Parameter

Außer auf seine lokalen Attribute kann ein Shader auch auf globale Parameter zugreifen. Im Gegensatz zu den lokalen Attributen werden diese jedoch bereits beim Binden und Laden des Shaders und nicht zur Laufzeit des Shaders ausgewertet. Somit kann man diese Attribute nutzen, um mehrere Instanzen desselben Shader-Programms mit unterschiedlichen Eigenschaften zu erstellen. Nachdem das Shader-Programm gebunden und ins Grafiksystem geladen wurde, haben Änderungen der globalen Parameter keine Auswirkungen mehr auf das Shader-Programm. Wird das Programm hingegen erneut gebunden, werden die neuen Werte der Parameter berücksichtigt.
Im Beispielcode sind globale Parameter *kursiv* hervorgehoben.
Globale Parameter können vom Programmierer einfach erzeugt werden, indem eine Variable entsprechenden Typs außerhalb eines Shader-Programms deklariert wird. So wird beispielsweise mit

```
ShPoint3f u;
```

ein Parameter des Typs *ShPoint3f* mit Namen u angelegt. In einem später definierten Shader-Programm kann dieser dann verwendet werden. Hierbei gelten die Scope-Regeln von C++. Im folgenden Code-Beispiel würde keineswegs der dort deklarierte Parameter u verwendet, da er sich nicht im Scope des Shaders befindet:

```
{
  ShPoint3f u(3.0,1.5,6.3);
}

ShShader shader = SH_BEGIN_VERTEX_PROGRAM {
  ShInputPoint3f i;
```

```
  ShOutputPoint3f o;

  o = u;

} SH_END_PROGRAM;
```

Entweder gäbe es beim Compilieren des Hauptprogramms einen Fehler, da u nicht bekannt ist, oder eine in einem umfassenderen Scope deklarierte Variable u würde verwendet, auf keinen Fall aber die Variable u, die im ersten gezeigten Scope deklariert wird.

3.4 Temporäre Variablen

Werden Sh-Variablen innerhalb eines Shader-Programms deklariert, so handelt es sich um lokale Variablen innerhalb dieses Programms. Diese sind im Gegensatz zu den globalen Parametern nicht statisch, sondern können manupuliert werden. Somit können sie verwendet werden, um Zwischenergebnisse komplexer Berechnungen abzuspeichern und später wieder zu verwenden.

3.5 Typumwandlungen

Grundsätzlich lassen sich alle Datentypen, die die LibSH bereitstellt, ineinander umwandeln, da es sich eigentlich nur um Repräsentationen von float-Tupeln handelt. Beachtet werden muß lediglich, daß beide Typen die gleiche Dimension besitzen. Folgende Zuweisung wäre damit durchaus typkorrekt:

```
ShPoint3f p(0.0,1.0,2.0);
ShColor3f c = p;
```

Nicht typkorrekt wäre folgende Zuweisung:

```
ShColor4f c1(1.0,0.0,0.0,0.5);
ShColor3f c2 = c1;
```

Man kann jedoch einzelne Komponenten der Tupel herausgreifen und so zum Beispiel aus einem vierdimensionalen Farbwert die ersten drei Komponenten extrahieren und in einem dreidimensionalen Farbwert speichern:

```
ShColor4f c1(1.0,0.0,0.0,0.5);
ShColor3f c2 = c1(0,1,2);
```

Genau so kann man aus einem dreidimensionalen Vektor alle drei Komponenten wahlfrei in einen vierdimensionalen Vektor schreiben:

```
ShColor4f c1(1.0,0.0,0.0,0.5);
ShColor3f c2 = c1(0,1,2);
ShColor4f c3;
c3(0,1,3) = c2;
```

Die Variable c3 bekommt hier nicht direkt bei ihrer Deklaration einen Wert zugewiesen, da der C++-Compiler die Angabe der zu belegenden Komponenten als Wert, mit dem die Variable zu belegen wäre, mißinterpretieren würde, und das Programm wegen verletzter Typkorrektheit nicht übersetzen würde. Zu beachten ist, daß bei solchen Komponentenangaben auf der linken Seite einer Zuweisung diese nur in aufsteigender Reihenfolge erfolgen dürfen. Die Zuweisung

```
c3(1,3,0) = c1;
```

wäre demnach nicht zulässig, wohl aber folgende Zuweisung:

```
c3 = c1(1,3,0);
```

Auf diese Weise können Komponenten in Vektoren umsortiert werden.
Das oben vorgestellte Verfahren wird als **Swizzling** bezeichnet.
Im oben angegebenen Beispiel-Shader wird die Typumwandlung von einer höheren Dimension in eine niedrigere zweimal verwendet. Unter anderem wird damit der vierdimensionale Parameter color auf drei Dimensionen gebracht, indem die Alpha-Komponente weggelassen wird.

3.6 Mischen von C++- und LibSH-Code

Innerhalb von Shader-Programmen kann auch C++-Code verwendet werden. So können zum Beispiel in einer For-Schleife nacheinander die Komponenten eines Vektors bearbeitet werden:

```
for (int i=0; i<3; i++)
  outputcolor(i) = inputcolor(i)*1.5;
```

Dabei wird nicht etwa eine Schleife auf der GPU ausgeführt, vielmehr werden vier einzelne Befehle für die GPU zur Laufzeit des Hauptprogramms erzeugt. (Siehe hierzu auch Abschnitt 4.4 und Abschnitt 4.5.)
Durch die Verwendung von C++-Funktionen kann man den Shader-Code übersichtlicher gestalten:

```
ShColor3f killGreen(ShColor3f c)
{
  ShColor3f out(c);
  out(1) = 0.0;
  return out;
}

ShColor3f mycolor1(0.75,0.6,1.0);
ShColor3f mycolor2 = killGreen(mycolor1);
```

Ähnlich wie bei der For-Schleife wird hier nicht wirklich eine Subroutine erzeugt, stattdessen wirkt diese Funktion als eine Art Makro, das an beliebigen Stellen aufgerufen werden kann und dann dort Code erzeugt, der die in der Funktion definierten Berechnungen ausführt.

3.7 Verwendung von Texturen

Texturen können ähnlich wie bei normalem OpenGL verwendet werden: Zunächst wird ein Objekt vom Typ *ShTexture* erstellt, in das dann die gewünschte Textur hineingeladen wird:

```
// declare image
ShImage image;

// load image file
image.loadPng("meinetextur.png");

// create Texture object with width and height of image
ShTexture2D<ShColor4f> texture(image.width(), image.height());

// load image into texture object
texture.load(image);
```

Auf die Textur zugreifen kann man einfach durch Übergabe der Texturkoordinaten. Diese sollten zwischen 0.0 und 1.0 liegen:

```
ShColor4f texpixel = texture(0.3,0.5);
```

3.8 Den Shader verwenden

Bisher wurde der Shader lediglich definiert. Um ihn zu verwenden, muß er gebunden und ins Grafiksystem übertragen werden. Dies geschieht mit Hilfe eines Aufrufs der Funktion *shBindShader()*. Ihr muß das jeweilige Shader-Programm als Parameter übergeben werden. Da wir zwei Programme definiert haben (einen Vertex- und einen Fragment-Shader), muß *shBindShader()* zwei mal aufgerufen werden:

```
shBindShader(diffuse_vertex);
shBindShader(diffuse_fragment);
```

Jetzt können Geometriedaten ans Grafiksystem geschickt werden. Dies geschieht mittels normaler OpenGL-Befehle:

```
glBegin(GL_TRIANGLES);
  glNormal3f(0.0,1.0,1.0);
  glTexCoord2f(0.0,0.0);
  glVertex3f(0.0,2.0,4.0);
  glNormal3f(1.0,0.0,1.0);
  glTexCoord2f(1.0,1.0);
  glVertex3f(2.5,3.3,6.7);
  glNormal3f(1.0,1.0,0.0);
  glTexCoord2f(1.0,0.0);
  glVertex3f(0.2,1.8,3.5);
glEnd();
```

Mit dem Absetzen eines *glVertex*-Befehls wird das Datenpaket ans Grafiksystem geschickt.

4 Hinter den Kulissen: Wie funktioniert das alles?

Nachdem wir uns mit der Verwendung der LibSH zum Erstellen von Shader-Programmen befaßt haben, wollen wir untersuchen, wie das Ganze überhaupt funktioniert.

4.1 Ein erster Compilerlauf

Wie in Abschnitt 2 bereits erwähnt, werden Aufrufe in die LibSH-API als ein Strom symbolischer Tokens, aus denen ein Syntaxbaum erstellt werden kann, interpretiert. Um die Verwendung gewohnter syntaktischer Konstrukte zu ermöglichen, wurden die arithmetischen Operatoren der Sh-Objekte überladen, so daß sie jetzt internen LibSH-Code erzeugen und nicht direkt in C++ ausgewertet werden.
Die Übersetzung beginnt, sobald das Shader-Programm durch das Statement *SH_END_SHADER* abgeschlossen wird. Wir betrachten hier die Übersetzung eines einfachen Fragment Shaders, der eine eindimensionale Textur auf ein Objekt anwendet, in ARB-GPU-Code.
*(Die Syntax des ARB-GPU-Codes findet sich in der Beschreibung der von nVidia eingebrachten OpenGL-Extensions **ARB_vertex_program** [3] und **ARB_fragment_program** [2].)*
Bei *tex* handelt es sich um eine Textur vom Typ ShTexture1D<ShColor4f>.

LibSH-Code	*ARB-Code*

`ShShader f =`
`SH_BEGIN_FRAGMENT_PROGRAM`
`{`
	`!!ARBfp1.0`

`ShInputVector3f hv;`
`ShInputTexCoord2f u;`
`ShInputNormal3f nv;`
`ShInputColor3f ec;`
`ShInputPosition3f pdxy;`
`ShOutputColor3f fc;`
	`ATTRIB i3 = fragment.texcoord[1];`
	`ATTRIB i2 = fragment.texcoord[0];`
	`ATTRIB i4 = fragment.texcoord[2];`
	`ATTRIB i1 = fragment.color;`
	`ATTRIB i0 = fragment.position;`
	`OUTPUT o0 = result.color;`

	`PARAM c0 = {3, 0, 0, 0};` `TEMP t0, t1;`

`fc`
`= tex(u(0)/3.0)(0,1,2);`
	`RCP t0.x, c0.x;`
	`MUL t1.x, i2.x, t0.x;`
	`TEX t0.xyzw, t1.x, texture[0], 1D;`
	`MOV o0.xyz, t0.xyzw;`

`} SH_END_PROGRAM;`	`END`

Gut zu erkennen ist die direkte Übernahme der Deklaration der Ein- und Ausgabeattribute in den ARB-Code.

Die darauf folgenden Deklarationen *PARAM* und *TEMP* haben zunächst keine Entsprechung im LibSH-Code. *PARAM* deklariert eine Konstante, die später im Programm benötigt wird, *TEMP* reserviert zwei Register für temporäre Zwischenergebnisse.
Die jetzt folgenden vier Instruktionen führen die Berechnung von fc durch.

```
RCP t0.x, c0.x;
```

berechnet den Kehrwert der Konstante c0 und speichert ihn im Zwischenspeicher t0. Man sieht, daß hier nur auf die x-Komponente der eigentlich vierdimensionalen Konstante c0 zugegriffen wird, ebenso bei t0. Dies liegt daran, daß die Register der GPU, die zur Speicherung der Konstante und zur Aufnahme der Zwischenergebnisse verwendet werden, festverdrahtet für vierdimensionale Werte sind, hier aber mit Skalaren gerechnet wird.

```
MUL t1.x, i2.x, t0.x;
```

berechnet nun den Term *u(0)/3.0* und speichert das Ergebnis in t1.x. Jetzt endlich kann auch der Textur-Lookup erfolgen. Es fällt auf, daß

```
TEX t0.xyzw, t1.x, texture[0], 1D;
```

als Zwischenspeicher für das Ergebnis t0 verwendet. Offensichtlich führt der Compiler bereits eine Optimierung der Registerbelegung durch und hat an dieser Stelle festfestellt, daß das mit t0 bezeichnete Register wiederverwendet werden kann, da der in ihm gespeicherte Wert nicht mehr benötigt wird.

```
MOV o0.xyz, t0.xyzw;
```

kopiert das Ergebnis schließlich ins Ausgabeattribut. Dabei wird es durch Swizzling in einen dreidimensionalen Typ umgewandelt. (Zur Erklärung von Swizzling siehe Abschnitt 3.5.)
Die Assemblersyntax für Swizzling ist übrigens recht einfach. Beim Zielregister werden die Komponenten angegeben, in die geschrieben werden soll, beim Quellregister die Komponenten, aus denen gelesen werden soll. Dabei ist die Reihenfolge entscheidend. Man sieht, daß beim Quellregister vier Komponenten angegeben sind, obwohl eigentlich nur drei benötigt werden. Im Code werden beim Quellregister jedoch immer vier Komponenten angegeben. Die überzähligen Komponenten werden dabei einfach ignoriert. Ein einfaches Beispiel mit Swizzling auf beiden Seiten könnte so aussehen:

LibSH-Code	*ARB-Code*
	`PARAM u0 = program.local[0];`
`fc(0,2)` `  =Globals::mycolor(3,1);`	`MOV o0.xz, u0.wyyw;`

4.2 Handling von Parametern

Wir wollen nun prüfen, wie der Compiler mit globalen Parametern umgeht. Dazu ersetzen wir die letzte Zeile des LibSH-Programms durch

```
fc = Globals::color;
```

Globals::color ist ein globaler Parameter vom Typ *ShColor3f*. Nun ergibt sich folgendes Programm (nach der Deklaration der Attribute):

LibSH-Code	*ARB-Code*
	`PARAM u0 = program.local[0];`
`fc = Globals::color;`	`MOV o0.xyz, u0.xyzw;`

(Wir werden bei allen Listings im folgenden Abschnitt die Deklaration der Ein- und Ausgabeattribute weglassen, da sie sich nicht ändern werden.)

Der Compiler reserviert also ein Register für den externen Parameter. Beim Binden und Laden des Shaders wird der aktuelle Wert des Parameters in dieses Register geschrieben, das daraufhin wie jedes andere Konstantenregister verwendet werden kann.

4.3 Registerbelegung und Optimierung

Angesichts der noch vergleichweise knappen Ressourcen aktueller programmierbarer Shader-Einheiten ist es natürlich interessant, wie gut der Compiler optimiert. Wir wollen zunächst überprüfen, ob ungenutzte Variablen erkannt und wegoptimiert werden.

LibSH-Code	*ARB-Code*
	`PARAM c0 = {3, 0, 0, 0};` `TEMP t0, t1;`

`ShColor3f d;`
`fc`
`  = tex(u(0)/3.0)(0,1,2);`
	`RCP t0.x, c0.x;`
	`MUL t1.x, i2.x, t0.x;`
	`TEX t0.xyzw, t1.x, texture[0], 1D;`
	`MOV o0.xyz, t0.xyzw;`

Wie man sieht, erzeugt der Compiler den gleichen Output wie im vorhergehenden Test. Die ungenutzte Variable d wurde also erkannt und wegoptimiert. Ein weiterer Test soll zeigen, ob der Compiler eine ungenutzte Variable auch dann noch erkennt, wenn eine Zuweisung an sie erfolgt.

LibSH-Code	*ARB-Code*
	`PARAM c0 = {3, 0, 0, 0};` `TEMP t0, t1;`

`ShColor3f`
`d(0.0,0.0,0.0);`
`fc`
`= tex(u(0)/3.0)(0,1,2);`
	
	`RCP t0.x, c0.x;`
	`MUL t1.x, i2.x, t0.x;`
	`TEX t0.xyzw, t1.x, texture[0], 1D;`
	`MOV o0.xyz, t0.xyzw;`

Auch hier wurde die ungenutzte Variable erkannt und wegoptimiert. Auch eine kompliziertere Zuweisung wie im folgenden Test wird wegoptimiert:

LibSH-Code	*ARB-Code*
	`PARAM c0 = {3, 0, 0, 0};` `TEMP t0, t1;`

`ShColor3f`
`d=tex(u(0))(0,1,2);`
`fc`
`= tex(u(0)/3.0)(0,1,2);`
	
	`RCP t0.x, c0.x;`
	`MUL t1.x, i2.x, t0.x;`
	`TEX t0.xyzw, t1.x, texture[0], 1D;`
	`MOV o0.xyz, t0.xyzw;`

Wir wollen jetzt untersuchen, wie gut der Compiler unnötige Berechnungen wegoptimieren kann. Dazu berechnen wir zweimal den selben Texture-Lookup und addieren die beiden Ergebnisse:

LibSH-Code	*ARB-Code*
	`TEMP t0, t1;`

`fc = tex(u(0))(0,1,2)`
`+ tex(u(0))(0,1,2);`
	`TEX t0.xyzw, i2.x, texture[0], 1D;`
	`TEX t1.xyzw, i2.x, texture[0], 1D;`
	`ADD o0.xyz, t1.xyzw, t0.xyzw;`

Hier hat die Optimierung versagt. Anstatt den Lookup einmal auszuführen und dann zu sich selbst zu addieren übersetzt der Compiler den Lookup tatsächlich zweimal.
Beim Versuch, diese Optimierung von Hand vorzunehmen, ergibt sich folgendes:

LibSH-Code	*ARB-Code*
	`TEMP t0, t1;`

`ShColor3f d`
`= tex(u(0))(0,1,2);`
`fc = d+d;`
	`TEX t0.xyzw, i2.x, texture[0], 1D;`
	`MOV t1.xyz, t0.xyzw;`
	`ADD o0.xyz, t1.xyzw, t0.xyzw;`

Dies widerspricht der Erwartung. Zwar wird der Texture-Lookup jetzt nur noch einmal ausgeführt, dafür wird jetzt die Typkonvertierung in einem extra Schritt durchgeführt. Das verwundert umso mehr, als sie doch im vorhergehenden Beispiel offensichtlich in einem Schritt mit der Addition erfolgte.

Wir versuchen, statt beim Texture-Lookup bei der Addition den Typ umzuwandeln:

LibSH-Code	*ARB-Code*
	`TEMP t0, t1;`

`ShColor4f d = tex(u(0));`
`fc = d+d;`
	`TEX t0.xyzw, i2.x, texture[0], 1D;`
	`ADD t1.xyzw, t0.xyzw, t0.xyzw;`
	`MOV o0.xyz, t1.xyzw;`

Die Umwandlung wird allerdings immer noch in einem separaten Schritt ausgeführt. Die Optimierung ist an dieser Stelle offensichtlich noch nicht sehr ausgereift.

Eine weitere Art der Optimierung ist die Konstantenfaltung. Wir wollen prüfen, ob und wie gut der Compiler dies beherrscht. Bereits im Beispiel in Abschnitt 4.1 hätte eine Konstantenfaltung erfolgen können: Da die ARB-Assemblersprache keine Division zur Verfügung stellt, erzeugt der Compiler stattdessen Code zur Berechnung des Kehrwertes des Divisors und zur Multiplikation des Dividenden mit diesem Kehrwert:

LibSH-Code	*ARB-Code*
	`PARAM c0 = {3, 0, 0, 0};` `TEMP t0, t1;`

`fc`
`  = tex(u(0)/3.0)(0,1,2);`
	`RCP t0.x, c0.x;`
	`MUL t1.x, i2.x, t0.x;`
	`TEX t0.xyzw, t1.x, texture[0], 1D;`
	`MOV o0.xyz, t0.xyzw;`

Da es sich bei 3.0 ja um eine Konstante handelt, ist auch ihr Kehrwert zur Compilezeit zu berechnen. Der Compiler könnte daher die Konstante 3.0 durch die Konstante 0.333333333 ersetzen und damit den RCP-Befehl einsparen. Eine solche Optimierung ist allerdings offensichtlich noch nicht implementiert.
In einem weiteren Test wollen wir prüfen, was passiert, wenn zwei Konstanten an einer Berechnung beteiligt sind, und nicht wie oben ein Argument eine Variable ist:

LibSH-Code	*ARB-Code*
	`PARAM c0 = {0.5, 0.5, 0.5, 0};` `PARAM c1 = {0.2, 0.2, 0.2, 0};`
`fc=ShColor3f(0.5,0.5,0.5)` `  +ShColor3f(0.2,0.2,0.2);`	`ADD o0.xyz, c0.xyzw, c1.xyzw;`

Offensichtlich ist auch hier noch keinerlei Optimierung implementiert.

Wir wollen noch untersuchen, ob und wie der Compiler Swizzling optimiert. Dazu swizzlen wir eine Variable zwei mal, und zwar so, daß nach der Ausführung beider Operationen wieder die ursprüngliche Belegung herauskommt:

LibSH-Code	*ARB-Code*
	`PARAM c0 = {0.9, 0.6, 0.3, 0};` `TEMP t0;`

`ShColor3f d(0.9,0.6,0.3);`
`ShColor3f e;`
`e = d(2,1,0);`
`fc = e(2,1,0);`
	
	`MOV t0.xyz, c0.zyxw;`
	`MOV o0.xyz, t0.zyxw;`

Man sieht sofort, daß keinerlei Optimierung stattfindet. Die Anweisungen aus dem LibSH-Code werden 1:1 in ARB-Code übersetzt.

4.4 Konditionale und Schleifen

Die LibSH wurde von vornherein mit Support für bedingte Verzweigungen und Schleifen entworfen. Allerdings stand zum Zeitpunkt des Entwicklungsbeginns noch keine Hardware zur Verfügung, die diese Konstrukte unterstützte. Von daher ist es interessant zu sehen, wie die gut derzeitige Implementierung der LibSH damit umgehen kann, und ob die Entwickler eventuell Workarounds für bestimmte Situationen eingebaut haben, die auch ohne Verwendung von Konditionalen und Schleifen zu bewältigen sind.
In einem ersten Test wollen wir abhängig von einer Texturkoordinate die Farbe des Pixels auf rot (1,0,0) oder grün (0,1,0) setzen:

LibSH-Code	*ARB-Code*

	`PARAM c0 = {1, 0, 0, 0};`
	`PARAM c1 = {1, 0, 0, 0};`
	`PARAM c2 = {0, 1, 0, 0};`
	`TEMP t0;`

`ShColor3f d;`
`SH_IF (u(0) < 1.0) {`
`d=ShColor3f(1.0,0.0,0.0);`
`} SH_ELSE {`
`d=ShColor3f(0.0,1.0,0.0);`
`} SH_ENDIF;`
`fc = d;`
	
	`SLT t0.x, i2.x, c0.x;`
	
	`MOV t0.xyz, c2.xyzw;`
	
	`MOV o0.xyz, t0.xyzw;`

Zu bemerken ist hier zunächst das Token `SH_IF`, mit dem eine if-Konstruktion beginnt. Es handelt sich hier um ein Präprozessor-Makro, das Code für den LibSH-Compiler erzeugt. Somit wird diese Konstruktion später im GPU-Programm ausgeführt, nicht im Hauptprogramm. Gleiches gilt für `SH_ELSE` und `SH_ENDIF`.
Der Befehl

```
SLT t0.x, i2.x, c0.x
```

prüft, ob der Inhalt von i2.x (u(0)) kleiner als der Inhalt von c0.x (1.0) ist, und speichert das Ergebnis dieses Vergleichs in t0.x. Dabei bedeutet ein Ergebnis von 1.0 true, ein Ergebnis von 0.0 bedeutet false.

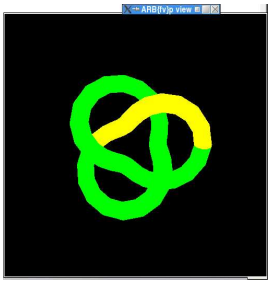
Der Compiler übersetzt hier zwar Code für einen Vergleich, erzeugt jedoch keinen Code für eine Verzweigung, sondern den Code für die Anweisung, die im ELSE-Teil steht.
Ganz nebenbei fällt noch auf, daß auch die Optimierung im Bereich Konstanten noch nicht sehr weit gediehen ist: abgesehen davon, daß die Konstante c1 im fertigen Code nirgendwo verwendet wird, ist eine Konstante gleichen Wertes (c0) bereits definiert, so daß c1 selbst wenn es im Code verwendet würde überflüssig wäre. Fairerweise muß man hier aber anmerken, daß die Eliminierung identischer Konstanten normalerweise vom ARB-Assembler vorgenommen wird, so daß diese Optimierung im Compiler nicht unbedingt notwendig ist.
Das eigentliche Problem ist aber, daß der Compiler derzeit offensichtlich keine Verzweigungsanweisungen generiert. Die Frage, die sich stellt, ist die, warum ausgerechnet der Code des ELSE-Teils übersetzt wurde. Eine Möglichkeit wäre, daß der Compiler derzeit aufgrund mangelnder Unterstützung von Konditionalen einfach grundsätzlich den ELSE-Teil übersetzt und den THEN-Fall komplett ignoriert. Eine andere Theorie ergibt sich nach einer Überlegung zu der in Abschnitt 4.3 beobachteten Optimierung bei der Verwendung der Register. Danach wäre es nämlich auch möglich, daß der Compiler sehr wohl beide Fälle berücksichtigt und einfach nacheinander übersetzt. Der nicht übersetzte Code aus dem THEN-Fall würde sich dann dadurch erklären, daß die erste Zuweisung an `d` vom Compiler wegoptimiert würde, da die zweite Zuweisung sie eh unwirksam machen würde. Um dies zu klären, ändern wir den Code so ab, daß auch die erste Zuweisung an d noch eine Auswirkung hätte, so daß der Compiler sie nicht wegoptimieren dürfte.

LibSH-Code	*ARB-Code*

	`PARAM c0 = {1, 0, 0, 0};`
	`PARAM c1 = {1, 0, 0, 0};`
	`PARAM c2 = {0, 1, 0, 0};`
	`TEMP t0;`

`ShColor3f d;`
`SH_IF (u(0) < 1.0) {`
`d=ShColor3f(1.0,0.0,0.0);`
`} SH_ELSE {`
`d=d`
`+ShColor3f(0.0,1.0,0.0);`
`} SH_ENDIF;`
`fc = d;`
	`SLT t0.x, i2.x, c0.x;`
	
	`ADD t0.xyz, t0.xyzw, c2.xyzw;`
	
	`MOV o0.xyz, t0.xyzw;`

Tatsächlich scheint der Compiler nur den ELSE-Fall zu berücksichtigen. Weiterhin stellen wir fest, daß der Compiler einen schwerwiegenden Fehler bei der Allokation der temporären Register macht: Nachdem das Ergebnis des Vergleichs (u(0) $<$ 1.0) in t0.x geschrieben wurde, wird t0 ohne erneute Initialisierung als Register zur Speicherung von d verwendet und einfach c2 dazuaddiert. Dies muß zwangsläufig zu Fehlern in der Berechnung führen. Und tatsächlich sind in der Bild-



schirmausgabe einige Pixel nicht grün, sondern gelb.
Der Fehler könnte daher rühren, daß der Compiler die erste Zuweisung an d erst hinter dem Vergleich sieht und daher t0 so verwendet, als habe es zwischen dem Vergleich und der Addition einen Befehl gegeben, der t0 initialisiert. Um diese Vermutung zu testen, weisen wir d vor dem IF-Block einen Wert zu:

```
ShColor3f d(0.0,0.0,0.0);

SH_IF (u(0) < 1.0) {
  d = ShColor3f(1.0,0.0,0.0);
} SH_ELSE {
  d = d + ShColor3f(0.0,1.0,0.0);
} SH_ENDIF;

fc = d;
```

Leider stürzt der Compiler der LibSH beim Versuch, dieses Programm zu übersetzen, mit einem Speicherzugriffsfehler ab, was darauf schließen läßt, daß der Compiler den gleichen Fehler wie vorher immer noch macht und hier aufgrund der vorher bereits erfolgten Zuweisung durcheinander kommt.

Wir wollen noch die Behandlung von Schleifen betrachten, hier am Beispiel einer While-Schleife. Dazu übersetzen wir folgendes Programm:

LibSH-Code	*ARB-Code*

	`PARAM c0 = {0.3, 0, 0, 0};`
	`PARAM c1 = {0, 0, 0, 0};`
	`PARAM c2 = {3, 0, 0, 0};`
	`PARAM c3 = {1, 0, 0, 0};`
	`PARAM c4 = {0, 0, 0.1, 0};`
	`TEMP t0, t1, t2;`

`ShColor3f d(0.3,0.0,0.0);`
`ShAttrib1f c(0.0);`
`SH_WHILE(c < 3.0) {`
`c=c+1.0;`
`d=d`
`+ShColor3f(0.0,0.0,0.1);`
`} SH_ENDWHILE;`
`fc=d;`
	`MOV t0.xyz, c0.xyzw;`
	`MOV t1.x, c1.x;`
	`SLT t2.x, t1.x, c2.x;`
	
	`MOV o0.xyz, t0.xyzw;`

Hier verhält sich der Compiler ähnlich wie bei der Übersetzung von IF-Blöcken: Es wird Code für den Vergleich erzeugt, aber das Resultat bleibt ungenutzt. Der Code in der Schleife wird komplett ignoriert. Der Compiler fährt mit der Übersetzung nach dem Ende des Schleifenblocks fort. Dementsprechend werden die Pixel nicht dunkelviolett eingefärbt, wie es nach diesem Programm eigentlich der Fall sein sollte, sondern dunkelrot (da die Blau-Komponente nicht erhöht wird).

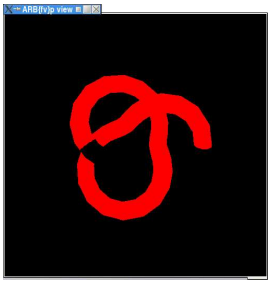
Wir werden jetzt noch untersuchen, ob der Compiler bei Schleifen zu ähnlichen Fehlern bei der Registerbelegung provoziert werden kann, wie es bei IF-Blöcken der Fall war. Hierzu werden wir wieder eine Variable deklarieren, aber nicht initialisieren. Danach werden wir dieser Variable innerhalb des Schleifenkörpers einen Wert zuweisen, und diesen Wert nachher ins Ausgabeattribut kopieren. *(Das folgende Programm ist natürlich Unsinn, da wir auf diese Weise eine Endlosschleife erzeugen. Da Schleifen aber derzeit sowieso nicht ausgeführt werden, kann es für diesen speziellen Test verwendet werden.)*

LibSH-Code	*ARB-Code*

	`PARAM c0 = {3, 0, 0, 0};`
	`PARAM c1 = {0, 0, 0.1, 0};`
	`TEMP t0;`

`ShColor3f d;`
`SH_WHILE(u(0)<3.0) {`
`d=ShColor3f(0.0,0.0,0.1);`
`} SH_ENDWHILE;`
`fc=d;`
	`SLT t0.x, i2.x, c0.x;`
	
	`MOV o0.xyz, t0.xyzw;`

Tatsächlich macht der Compiler den gleichen Fehler wie bei der Übersetzung des IF-Blocks. Eigentlich sollten alle Pixel schwarz sein (da der Schleifenkörper niemals ausgeführt wird), aber bei allen Pixeln, bei denen die Vergleichsbedingung zutrifft, ist die Rot-Komponente auf 1.0 gesetzt. Allerdings agiert der Compiler hier besser, wenn man d vor der Schleife einen Wert zuweist. Jetzt wird Code erzeugt wie erwartet (wenn man von der nicht übersetzten While-Schleife absieht), und auch die Ausgabe entspricht den Erwartungen und enthält nur schwarze Pixel:



LibSH-Code	*ARB-Code*

	`PARAM c0 = {0, 0, 0, 0};`
	`PARAM c1 = {3, 0, 0, 0};`
	`PARAM c2 = {0, 0, 0.1, 0};`
	`TEMP t0, t1;`

`ShColor3f d(0.0,0.0,0.0);`
`SH_WHILE(u(0)<3.0) {`
`d=ShColor3f(0.0,0.0,0.1);`
`} SH_ENDWHILE;`
`fc=d;`
	`MOV t0.xyz, c0.xyzw;`
	`SLT t1.x, i2.x, c1.x;`
	
	`MOV o0.xyz, t0.xyzw;`

4.5 Workarounds für Konditionale und Schleifen

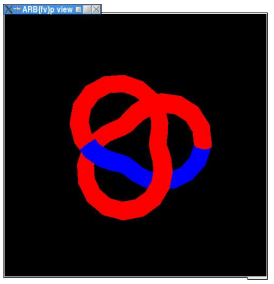
In bestimmten Fällen kann man die gleiche Funktionalität wie man sie durch IF-Blöcke und Schleifen erhalten würde durch andere Mittel erreichen. Im

Falle eines IF-Blocks kann man beide Ergebnisse berechnen, um dann das nicht gewünschte Ergebnis mit einer Maske auszublenden. Da keine bitweisen Verknüpfungen möglich sind, muß dies durch Multiplikation mit dem Ergebnis des Vergleichs bewerkstelligt werden, der praktischerweise bereits als Float-Wert vorliegt. Im folgenden Beispiel färben wir alle Pixel, bei denen die erste Komponente der Texturkoordinate kleiner als 3 ist, rot ein, die übrigen blau:

LibSH-Code	*ARB-Code*

	`PARAM c0 = {3, 0, 0, 0};`
	`PARAM c1 = {1, 0, 0, 0};`
	`PARAM c2 = {0, 0, 1, 0};`
	`PARAM c3 = {1, 0, 0, 0};`
	`TEMP t0, t1, t2;`

`ShAttrib1f s=(u(0)<3.0);`
`fc=ShColor3f(1.0,0.0,0.0)*s+`
`ShColor3f(0.0,0.0,1.0)*(1.0-s);`
	`SLT t0.x, i2.x, c0.x;`
	`ADD t1.x, c1.x, -t0.x;`
	`MUL t2.xyz, c2.xyzw, t1.xxxw;`
	`MUL t1.xyz, c3.xyzw, t0.xxxw;`
	`ADD o0.xyz, t1.xyzw, t2.xyzw;`



Hier wird zunächst das Ergebnis des Vergleichs in der Variablen s gespeichert. War das Ergebnis *true*, so enthält s den Wert 1.0, war das Ergebnis *false*, so enthält s den Wert 0.0. Nun kann man beide Varianten (THEN- und ELSE-Fall) berechnen und durch Multiplikation mit s (THEN-Fall) bzw. (1.0-s) (ELSE-Fall) sowie anschließender Addition der beiden Ergebnisse das Resultat erhalten, das man mit einer IF-Konstruktion auch erhalten hätte. Allerdings ist dieses Vorgehen nur bei einfachen Berechnungen sinnvoll, da sonst zu viele Ressourcen für nicht verwendete Berechnungen aufgewendet werden müssen.

Schleifen lassen sich leider nicht in jedem Fall durch eine andere Konstruktion ersetzen. Steht allerdings die Anzahl der Durchläufe schon im Voraus fest (zur Laufzeit des Hauptprogramms), so kann man die Schleife im Shader durch eine C++-Schleife ersetzen, die dann entsprechenden LibSH-Code erzeugt. (Siehe auch Abschnitt 3.6.)

5 Zusammenfassung und Ausblick

Die LibSH soll dem Programmierer die Erstellung von Shader-Programmen vereinfachen. Wenn auch noch nicht alles funktioniert und es noch viele Unzulänglichkeiten im Compiler speziell im Bereich Optimierung sowie beim Umgang mit derzeit noch nicht in Hardware verfügbaren Features (hier würde man sich eine Fehlermeldung wünschen anstatt fehlerhaftem Code) gibt, so zeigt die LibSH derzeit doch schon das Potential, das im Metaprogramming von Shader-Einheiten steckt.

Mit der nicht zuletzt durch die Spieleindustrie angetriebenen ständigen Weiterentwicklung der für PCs verfügbaren Grafikhardware ist es nur noch eine Frage der Zeit, bis die Hardware alle von der LibSH angebotenen Features unterstützt. Wenn die Library bis zu diesem Zeitpunkt entsprechend ausgereift ist, kann sie ein mächtiges Werkzeug zur Erschließung der Rechenkapazitäten der Grafikkarten sein, nicht zuletzt auch für allgemeine Berechnungsaufgaben, bei denen Vektor-CPUs benötigt werden.

Mittlerweile (Februar 2004) ist übrigens eine neue Version der LibSH erschienen, die jedoch aus zeitlichen Gründen nicht mehr getestet werden konnte.

Literaturverzeichnis

[1] Zheng Qin Michael D. McCool and Tiberiu S. Popa. Shader metaprogramming, revised version from http://www.cgl.uwaterloo.ca/projects/rendering/papers/metaapipaper.pdf. *SIGGRAPH/Eurographics Graphics Hardware Workshop*, September 2-3, 2002, Saarbrücken, Germany:57–68.

[2] NVIDIA Corporation. *ARB_fragment_program, http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt*, 2002-2003.

[3] NVIDIA Corporation. *ARB_vertex_program, http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt*, 2002-2003.

Hans-Joachim Haas
Betreuer: Philipp Lucas

Nutzung nicht programmierbarer Graphikkarten

Die Graphikhardware in einem PC kann mehr, als nur Bilder anzeigen. Moderne Graphikkarten sind auch in der Lage, allgemeine mathematische Berechnungen mit der GPU (Graphic Process Unit) durchzuführen, die nicht direkt mit der Bilderzeugung zusammen hängen. Unter Verwendung von Blending und Texturing in OpenGL ist es möglich, mathematische Operationen zu berechnen, wie z. B. Addition und Multiplikation. Die GPU besitzt bei diesen Berechnungen die gleichen Performance-Eigenschaften einer CPU und in manchen Fällen werden sie sogar übertroffen.

1 Einführung

Nicht programmierbare Graphikkarten kann man für sehr viele Zwecke verwenden, auch für vieles, das nicht direkt mit der Bilderzeugung zusammen hängt. Im folgenden werden wir sehen, wie es möglich ist, mit Graphikkarten allgemeine mathematische Berechnungen durchzuführen. Nach einer kurzen Beschreibung der Graphikhardware und deren Entwicklung in Kapitel 2 werden in Kapitel 3 die einzelnen mathematischen Operationen genauer erläutert. Hier erfahren wir, wie man mit der GPU z. B. addiert oder multipliziert. In Kapitel 4 und 5 werden dann zwei Beispiele dargestellt. Das erste Beispiel zeigt, wie man schnell und effizient allgemeine Voronoi-Diagramme mit der GPU berechnet [5]. Dies geschieht durch die Berechnung eines Abstands-Netzes und Abstandsvergleiche im Z-Buffer. Im zweiten Beispiel werden mit der GPU Matrizen multipliziert unter der Verwendung von Texturing und Blending [6]. Die Operationen werden auf Farbwerte angewendet, die die einzelnen Elemente in den Matrizen repräsentieren.

Nicht programmierbare Graphikkarten sind für diese Art Berechnungen sehr gut geeignet. Sie liefern eine sehr gute Performance vergleichbar mit der CPU.

Graphikkarten zählen heute schon lange nicht mehr zur Spezial-Hardware. Im Gegenteil, in jedem Rechner ist heute eine Graphikkarte vorhanden, so dass diese Technik auch für Heimanwender interessant sein wird.

2 Graphikhardware und deren Verwendung

2.1 Übersicht: Graphikkarten und historische Entwicklung

Noch vor etwa 10 Jahren waren Graphikkarten nicht mehr als Adapter, die aus dem Framebuffer ein analoges Ausgangssignal bildeten, welches mit einem Monitor angezeigt werden konnte. Der Framebuffer enthielt dabei die Bildinformationen, die beispielsweise aus RGB-Werten für ein Farbbild bestanden. Diesen Buffer gibt es auch heute noch, jedoch wird heute dieser auf eine andere Weise mit Daten gefüllt. Früher geschah das Füllen des Framebuffers allein durch die CPU, was im Bezug auf polygonale 3D-Graphik bedeutete, dass die gesamte 3D-Pipeline auf der CPU sequentiell abgearbeitet werden musste. Man hat sich schnell Gedanken gemacht über eine spezialisierte Hardware, die dies bewältigen konnte. Daraufhin wurden Spezial-Graphikrechner entwickelt, welche sogar Texture-Mapping ausführen konnten.

Diese Technik wurde schnell auch für den Heimanwender interessant und verwendbar, als 1997 die ersten Graphikkarten von nVidia, wie die TNT oder TNT2, die von ATI und 3dfx auf den Markt kamen. Diese Graphikkarten waren in der Lage, transformierte Dreiecksdaten mit mehreren Texturen (meist 2) zu verzieren. Für das Kombinieren der Texturen standen wenige Operationen zur Verfügung, aber schon bilineare und zum Teil trilineare Texturfilter. Bilineare Filter sorgen dafür, dass die Texturen nicht mehr so grob verpixelt wirken, sondern erzeugen weiche Farbverläufe, indem aus den Farbwerten eines 2×2-Texelfeldes ein Zwischenwert berechnet wird. Dies entspricht im Grunde einem Blur-Verfahren und gehört zu den Standardtechniken jedes 3D-Renderers, da der Graphikchip diesen Filter mit sehr geringen Performanceverlusten berechnen kann. Es erhöht sich lediglich der Speicherzugriff, da mehr Elemente betrachtet werden müssen. Wenn dieser Filter nicht ausreicht und die Texturen immer noch etwas verpixelt wirken, werden trilineare Texturfilter angewendet. Diese sind eine Kombination aus zwei bilinearen und einem linearen Filter.

1999 kamen dann Graphikkarten, von nVidia die GeForce und von ATI die Radeon 7500, auf den Markt, mit der Eigenschaft, auch Vertex-Transformationen durchführen zu können. Diese mussten nun nicht mehr von der CPU verarbeitet werden. Daher war es möglich, eine sehr hohe Anzahl an Vertices in kurzer Zeit zu transformieren.

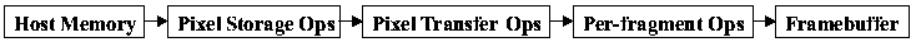
Dann folgten 2001 Graphikkarten mit erweiterter Konfigurierbarkeit, wie die GeForce3 von nVidia oder die ATI Radeon 8500. Dadurch ist man nun in der Lage, kurze Programme zu definieren, die dann auf jeden Vertex angewendet werden. Jedoch besitzen diese Graphikkarten noch nicht die Programmierbarkeit wie man sie bei heutigen Graphikkarten findet, besitzen aber schon eine sehr hohe Funktionalität. In den folgenden Beispielen wurde genau diese Generation von Graphikkarten benutzt. Wie wir sehen werden, sind diese für die im

Folgenden dargestellten Beispiele von mathematischen Berechnungen sehr gut geeignet.

2.2 Die Graphikpipeline

Dieses Kapitel enthält eine kurze Übersicht über die Graphikpipeline, wie sie in einer GPU benutzt wird. Eine genauere Erläuterung findet man in [9] und [3].

Abbildung 1 : Graphikpipeline. Übersicht der verwendeten Operationen [8].

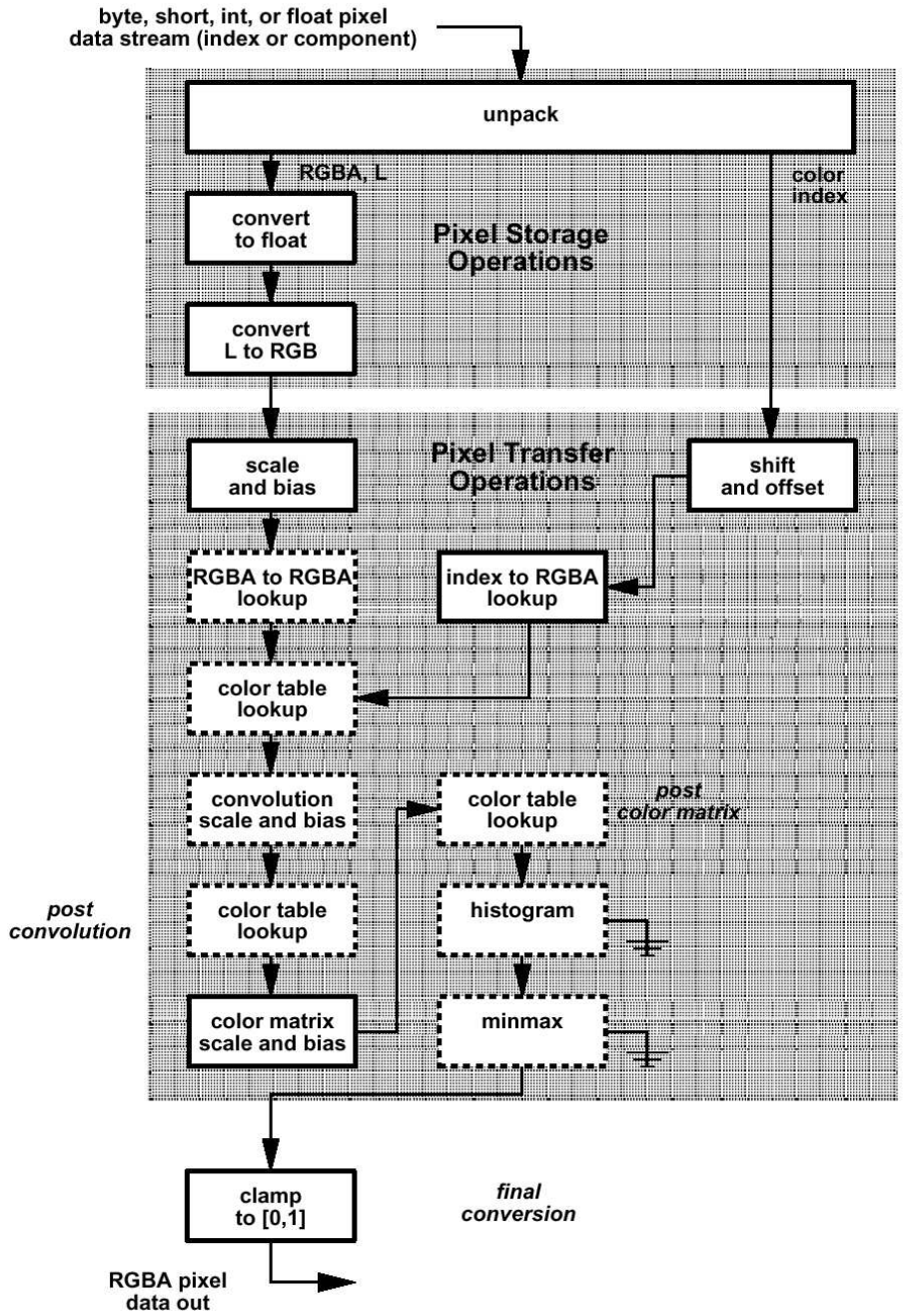


Die GPU ist die Hauptkomponente einer jeden Graphikkarte. Dieser Chip ist es, der die Berechnungen vollzieht, die wir dann als dreidimensionales Bild wahrnehmen. Bis es jedoch soweit ist, bevor ein einziges fertiges Bild berechnet ist, müssen erst Millionen einzelne mathematische Anweisungen vollzogen werden. Durch die heutige Technik geht dies sehr schnell und es ist möglich, mehrere von diesen Bildern in einer Sekunde auf den Bildschirm zu bringen. In einer Bildszene bestehen alle Objektoberflächen aus Polygonen. In der GPU werden diese Polygone durch Rasterisierung in Pixelfragmente umgewandelt. Mit der Funktion *DrawPixels* in OpenGL können die Fragmente anschließend in den Framebuffer abgespeichert und angezeigt werden. In Abbildung 2 sehen wir einen Teil der Graphikpipeline, und zwar den Teil mit den Operationen für die Funktion *DrawPixels* [7]. Es folgt eine kurze Beschreibung der einzelnen Schritte der Pixel Transfer Operationen in der Pipeline. Genaueres kann in der OpenGL Spezifikation 1.5 [7] nachgelesen werden.

Nach dem Auslesen der Daten aus dem Host Memory werden die Pixel Storage Operationen ausgeführt. Dort werden die einzelnen Daten zuerst in Gleitkommazahlen umgewandelt. Z.B. werden Integer Zahlen durch die Formel $(2c+1)/(2^{32}-1)$ umgerechnet. (Siehe [7]). Abhängig von dem Format der Daten werden anschließend die Daten in RGB- oder RGBA-Werte umgewandelt. Danach werden die Pixel Transfer Operationen ausgeführt. Wir betrachten hier allerdings nur die Operationen auf RGBA-Daten:

- Scale and Bias
  Beim Skalieren bzw. Biasing wird jedes Element mit einem bestimmten Skalierungsfaktor multipliziert bzw. zu einem Wert addiert. Dabei werden die R-,G-,B- und A-Werte einzeln betrachtet.
- RGBA to RGBA Lookup
  Hier werden die RGBA-Daten den entsprechenden Werten zugeordnet. Die Farben stehen in der jeweiligen Tabelle, der Pixelmap, für die einzelnen Komponenten R, G, B und A. Für das Zuordnen der Werte müssen die Daten in den Wertebereich der Tabelle skaliert werden. Dies geschieht, indem man zuerst die Daten in den Bereich $[0, 1]$ skaliert und dann mit der Länge der jeweiligen Tabelle multipliziert. So erhält man einen Index, dem

Abbildung 2 : Graphik-Pipeline. Quelle: OpenGL Specification 1.5 [7]. Operationen der *DrawPixels* Funktion.



man nun einen Wert aus der Tabelle zuordnen kann. Der Programmierer kann somit bestimmen, dass einzelne Farbwerte ersetzt werden.
Enthalten die Daten keine RGBA-Werte, sondern nur einen Color-Index, müssen die Werte aus anderen Tabellen, den Pixelindexmaps, ausgelesen werden. Jede dieser Tabelle hat genau 2^n Einträge. Der Index muss nicht skaliert werden, so dass direkt ein Wert aus der jeweiligen Tabelle zugeordnet werden kann.

- Color Table Lookup
  Hier werden nun den obigen Werten die entsprechenden Farbwerte aus der Color Table zugeordnet. Für alle Farben wird die selbe Tabelle verwendet. Die Werte müssen jedoch zuerst wieder in den Wertebereich der Color Table skaliert werden. Anschließend kann man den Farbwert aus der Tabelle auslesen.

- Convolution
  Hier wird ein Filter auf die Daten angewendet. Pixelfarben können anhand einer Vorschrift in Abhängigkeit der umliegenden Pixel verändert werden. Möchte man z. B. einen 3×3 Smoothing-Filter anwenden, wird einem Pixel der Mittelwert aus den acht umliegenden Pixel und sich selbst zugewiesen. Seien W_f die Breite und H_f die Länge des Filters (im Beispiel des Smoothing-Filters $= 3$), i, j die Koordinaten des Pixels und C_s bzw. C_f die Pixelfarbe der Quelle bzw. des Filters. Dann erhalten wir folgende Formel:
  $$C[i,j] = \sum_{n=0}^{W_f-1} \sum_{m=0}^{H_f-1} C_s[i+n, j+m] * C_f[n,m]$$
  Die Farbwerte der Quelldaten werden mit den Werten des Filters komponentenweise multipliziert. Dabei ist es möglich, alle Werte einzeln zu verändern, oder aber auch z. B. nur den Alpha-Wert, oder alle Werte mit einer Konstanten zu multiplizieren. Pixel, die an Rändern des Bildes liegen, können anders behandelt werden, als Pixel im Bildinneren.

- Color Matrix Transformation
  Jedes Element wird anhand der Color-Matrix transformiert. Alle zu skalierenden Komponenten werden hierfür mit einem Skalierungsfaktor multipliziert und anschließend zu einem Biasingwert addiert. Sei M_c die Color-Matrix, die Komponenten mit Index s seien die Skalierungsfaktoren und die Komponenten mit Index b die Biasingwerte. Dann erhalten wir folgende Color Matrix Transformation:
  $$\begin{pmatrix} R' \\ G' \\ B' \\ A' \end{pmatrix} = \begin{pmatrix} R_s & 0 & 0 & 0 \\ 0 & G_s & 0 & 0 \\ 0 & 0 & B_s & 0 \\ 0 & 0 & 0 & A_s \end{pmatrix} M_c \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} + \begin{pmatrix} R_b \\ G_b \\ B_b \\ A_b \end{pmatrix}$$

- Histogramm
  In diesem Teil der Pipeline wird eine Histogramm-Tabelle erstellt und abgespeichert. Jede R-, G-, B-, A-Komponente eines Pixels wird untersucht und der entsprechende Wert in der Histogramm-Tabelle um eins erhöht. Übersteigt ein Wert den Maximalwert der Tabelle, wird dieser als undefiniert angenommen.

- Min-Max
  Hier wird das Minimum und das Maximum einer jeden Komponente bestimmt. Die Werte werden als Gleitkommazahlen in einer Tabelle abgespeichert. Jede Komponente eines Farbwerts wird mit dem Minimum bzw. Maximum in der Tabelle verglichen. Bei einem niedrigerem bzw. höherem Farbwert der Komponente, wird der Wert in der Tabelle ersetzt.

Weitere wichtige Funktionen sind *ReadPixels* und *CopyPixels*. *ReadPixels* wird benötigt, um Pixel-Werte aus dem Framebuffer auszulesen. Mit *CopyPixels* kann man Pixel-Werte von einer Position im Framebuffer in eine andere kopieren. Beide Funktionen führen die gleichen Pixel Transfer Operationen durch wie die Funktion *DrawPixels*.

Nach den Pixel Transfer Operationen finden die Per-Fragment Operationen statt. Dort werden die wichtigen Operationen des Blending, des Pixel-Texturing und der Tiefenvergleiche im Z-Buffer ausgeführt. Im nachfolgenden Beispiel 2 wird zur Multiplikation Texturing verwendet. Mit Texturing kann ein Bild oder ein Teil eines Bildes auf eine Oberfläche eines Objektes oder Fragments abgebildet werden. Dabei wird die Farbe des Bildes benutzt um die RGBA-Werte des Fragments an der entsprechenden XY-Koordinate zu verändern. Dies kann auch mit mehreren Bildern gleichzeitig geschehen. Bei Pixel-Texturing wird einem Pixel explizit eine XY-Koordinate der Bildes zugewiesen. Es gibt verschiedene Arten, wie Texturing angewendet werden kann. In den verschiedenen Modi kann dies festgelegt werden. In dem Modus *REPLACE* wird die Farbe des Bildes einfach übernommen. Der Modus *MODULATE* dient dazu, beide Farbwerte zu multiplizieren. Diesen Modus nutzen wir in unserem Beispiel der Matrixmultiplikation um die Elemente zu berechnen. Es gibt darüberhinaus noch weitere Modi, wie *ADD*, *SUBTRACT* oder *INTERPOLATE*.

Eine weitere wichtige Funktion sind die Tiefenvergleiche im Z-Buffer. Im Z-Buffer werden normalerweise die Z-Koordinaten, also die Tiefe jedes sichtbaren Pixels gespeichert. Es können Werte gespeichert werden von 24 bis 32 Bit. Dies entspricht zwar einer höheren Anzahl an Bits pro Farbwert eines Pixel als in einem normalen Color-Buffer, ist im allgemeinen jedoch nicht ausreichend für genaue mathematische Berechnungen. Dies sollte bei Berechnungen mit dem Z-Buffer berücksichtigt werden. Diese Funktion wird in unserem ersten Beispiel, der Berechnung der Voronoi-Diagramme benötigt. Hier werden jedoch keine Tiefenwerte gespeichert, sondern die Abstände zwischen den einzelnen Voronoi-Formen.

Logische Operationen gehören ebenfalls zu den Per-Fragment Operationen. Die Operation wird zwischen dem Quell-Element und dem Element aus dem Framebuffer ausgeführt. Als Logische Operationen stehen z. B. *CLEAR, AND, COPY, NOOP, OR, XOR, EQUIV, INVERT* oder *NAND* zur Verfügung.

3 Mathematische Operationen mit Graphikhardware

In diesem Kapitel sehen wir, wie man mathematische Funktionen auf der GPU ausführen kann [8]. Rasterbilder bestehen aus Pixeln. Diese Pixel haben be-

stimmte Farbwerte im RGB- oder RGBA-Schema. Mit diesem Schema kann man Rasterbilder als drei- oder vierdimensionale Vektorfelder definieren, mit denen sich dann Berechnungen sehr schnell durchführen lassen. Allerdings gibt es hier ein paar Probleme mit der Vorzeichenarithmetik, da es wohl leider nur Farbwerte zwischen 0 und 1 gibt und daher auch keine negativen Farbwerte existieren. Dieses Problem wurde auf eine andere Art und Weise gelöst und wird später genauer erläutert.

Eine der Hauptfunktionen um mathematische Operationen auf der GPU durchzuführen ist Blending. Blending kombiniert die RGBA-Werte des Quell-Fragments mit denen des Ziel-Fragments, das im Framebuffer gespeichert ist. Dies geschieht entsprechend der Blending-Gleichung. Die Blending-Gleichungen sind z. B. Addition, Subtraktion, Minimum und Maximum. Desweitern kann auch eine Gewichtung des Quell- oder Zielpixels angegeben werden, womit man eine Multiplikation erreichen kann. Sei C_s der RGBA-Wert des Quell-Fragment, C_d der Wert des Ziel-Fragments, C der neue Farbwert, so erhält man z. B. folgende Blending-Gleichung für die Addition:

$$C = C_s S + C_d D.$$

C und D sind Faktoren zur Gewichtung der Quell- und Zielfarbwerte. Bei Subtraktion wird entsprechend das Additionszeichen getauscht. Für Minimum und Maximum erhält man folgende Gleichungen:

$$C = min(C_s, C_d) \text{ bzw. } C = max(C_s, C_d).$$

Um nun die Vorzeichenarithmetik zu implementieren, müssen alle Funktionen zuerst in das Intervall [0,1] skaliert und angepasst werden. Dies geschieht in der Graphikpipeline in den Per-Fragment Operationen nach dem Blending durch Scaling und Biasing.

Sei $\widetilde{f}$ eine Funktion f, die skaliert und angepasst wurde. Dann ist:

$$\widetilde{f} = \alpha f + \beta.$$

Die Konstanten α und β können einfach berechnet werden, da nur endlich viele Funktionswerte existiern. Eine bereits skalierte Funktion $\widetilde{f}$ kann man ebenfalls mit einer nicht skalierten Konstante c addieren. Dabei muss die Konstante mit dem gleichen Skalierungsfaktor multipliziert werden, wie die Funktion selbst:

$$\widetilde{f + c} = \alpha(f + c) + \beta = \alpha f + \beta + \alpha c = \widetilde{f} + \alpha c.$$

Wir erhalten die selbe Gleichung, wenn wir eine skalierte Funktion $\widetilde{f}$ und eine nicht skalierte Funktion g addieren:

$$\widetilde{f + g} = \widetilde{f} + \alpha g.$$

Um diese Operationen auszuführen, müssen wir g um α skalieren und additives Blending benutzen. Wenn beide Funktionen $\widetilde{f}$ und $\widetilde{g}$ bereits skaliert sind, kann man sie folgendermaßen addieren:

$$\widetilde{f + g} = \alpha(f + g) + \beta = \alpha f + \beta + \alpha g + \beta - \beta = \widetilde{f} + \widetilde{g} - \beta.$$

Multiplikation einer skalierten Funktion $\widetilde{f}$ mit einer Konstanten c funktioniert folgendermaßen:

$$\widetilde{cf} = \alpha(cf) + \beta = \alpha cf + \beta c + \beta - \beta c = c\widetilde{f} + (1-c)\beta.$$

Multiplikation einer skalierten Funktion $\widetilde{f}$ mit einer nicht skalierten Funktion g:

$$\widetilde{fg} = g\widetilde{f} + (1-g)\beta.$$

Die Multiplikation zweier skalierter Funktionen $\widetilde{f}$ und $\widetilde{g}$ ist ein wenig komplizierter:

$$\begin{aligned}
\widetilde{fg} &= \alpha fg + \beta \\
&= \alpha fg + \beta(f+g) + \frac{\beta^2}{\alpha} - \beta(f+g) - \frac{\beta^2}{\alpha} + \beta \\
&= \frac{1}{\alpha}(\alpha^2 fg + \alpha\beta(f+g) + \beta^2) - \frac{\beta}{\alpha}(\alpha \cdot (f+g) + \beta - \alpha) \\
&= \frac{1}{\alpha}(\widetilde{f} \cdot \widetilde{g}) - \frac{\beta}{\alpha}(\widetilde{(f+g)} - \alpha).
\end{aligned}$$

4 Beispiel 1: Voronoi-Diagramme

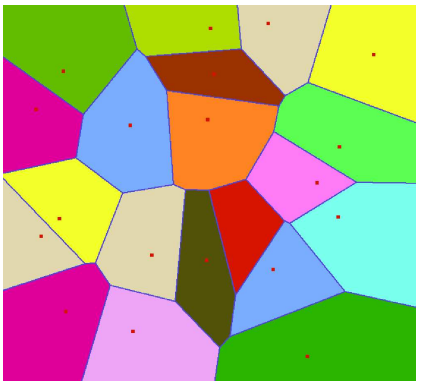
In unserem ersten Beispiel werden wir ein Verfahren zeigen, wie man schnell und effizient allgemeine Voronoi-Diagramme berechnet [5]. In einem Bild sind Punkte, Linien oder Kurven gegeben, die Voronoi-Formen. Ein Voronoi-Diagramm teilt dieses Bild in verschiedene Bereiche ein, derart dass in einem Bereich alle Punkte liegen, die am nächsten zu einem der gegebenen Voronoi-Formen sind. Abbildung 3) zeigt ein solches Diagramm mit Punkten als Voronoi-Formen. Jeder Punkt besitzt seinen Bereich, für den der Abstand zum diesem am geringsten ist. Genauere Erläuterungen kann man in [1] nachlesen.

4.1 Allgemeine Voronoi-Diagramme

Im folgenden werden ein paar Definitionen gegeben, um Voronoi-Diagramme zu beschreiben. Seien $A_1, A_2, \ldots, A_k$ Formen in einem Voronoi-Diagramm. Die Voronoi-Form A_i kann aus einem Punkt, einer Linie oder auch einer Kurve bestehen. Für jeden Punkt p im Raum definieren wir die Distanz von p zur Form A_i als $dist(p, A_i)$. Den Bereich, in dem die Punkte näher an A_i liegen als an A_j definieren wir als $Dom(A_i, A_j) = \{p \mid dist(p, A_i) \leq dist(p, A_j)\}$. Für eine Form A_i definieren wir den Voronoi-Bereich als $V(A_i) = \bigcap_{j \neq i} Dom(A_i, A_j)$. Die Einteilung des Raums in $V(A_1), V(A_2), \ldots, V(A_k)$ nennen wir allgemeines Voronoi-Diagramm.

Der einfachste Weg ein Voronoi-Diagramm zu berechnen ist wohl, für jeden Pixel im Raum die Distanz zu jeder einzelnen Form zu berechnen und anschließend die Pixel entsprechend einzufärben. Dies wäre ein sehr langwieriger Brute-Force Algorithmus, der iterativ über alle Formen und Pixel laufen müsste. Wir betrachten hier einen anderen Ansatz und verwenden für unsere Zwecke den Z-Buffer der Raster-Graphikhardware und definieren uns Distanzfunktionen. Im

Abbildung 3 : Voronoi-Diagramm mit Punkten als Voronoi-Formen. Quelle: [4]



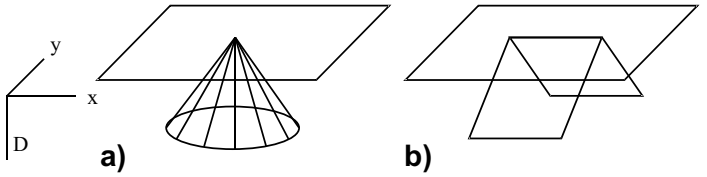
Z-Buffer wird normalerweise die Z-Koordinate, also die Tiefe jedes sichtbaren Pixels gespeichert. Wir verwenden diesen Buffer zur Berechnung und Speicherung der Distanz eines Punktes zur nächstliegenden Voronoi-Form. Mit den Distanzfunktionen können wir für jede einzelne Form ein polygonales Netz rendern. Durch die Tiefenvergleiche im Z-Buffer erhalten wir immer den geringsten Abstand jedes Pixels zu einer Voronoi-Form. Jede Voronoi-Form erhält eine eigene Farbe, so dass im Color-Buffer für jeden Pixel die Farbe von der Voronoi-Form gespeichert wird, an der es am nächsten liegt.

4.2 Die Distanz-Funktionen

Es gibt verschiedene Distanz-Funktionen für die einzelnen Arten von Voronoi-Formen wie Punkt, Linie oder Fläche. Dabei muss unterschieden werden, ob wir uns im zwei- oder dreidimensionalen Raum befinden. Zuerst betrachten wir den zweidimensionalen Raum und die dort verwendeten Funktionen.

- Punkt in 2D

Abbildung 4 : Distanzfunktion für Punkt (a) und Linie (b) mit $XY-$Ebene, die die Voronoi-Form enthält

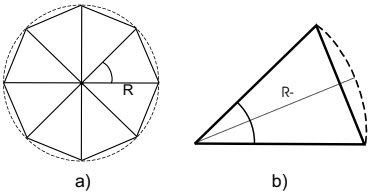


Die Distanz-Funktion für einen Punkt in einer Ebene ist ein Kegel (Abbildung 4 (a)). Der Kegel wird dabei durch Dreiecke approximiert. An der Kreisfläche kann dadurch ein Rundungsfehler entstehen (Abbildung 5 (a)). Um diesen Fehler so gering wie möglich zu halten, müssen wir genügend viele Dreiecke verwenden. Sei R der maximale Radius des Kegels, also der größte Abstand zwischen Beispielpunkt und Voronoi-Figur. Den maximalen Fehler bezeichnen wir mit ϵ. Nun können wir den maximalen Winkel α folgendermaßen berechnen (Abbildung 5 (b)):

$$\cos\left(\frac{\alpha}{2}\right) = \frac{R-\epsilon}{R} \Longrightarrow \alpha = 2cos^{-1}\left(\frac{R-\epsilon}{R}\right)$$

Wenn wir den Fehler ϵ bis auf einen Pixel reduzieren wollen, sollte man bei einem Netz der Größe 512×512 mindestens 60 Dreiecke verwenden und bei einem Netz der Größe 1024×1024 mindestens 85.

Abbildung 5 : Approximationsfehler ϵ an einer Kreisfläche (a) Berechnung von ϵ (b) [5]



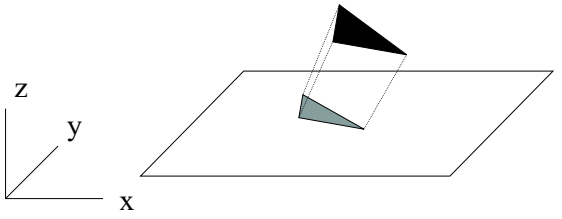
- Liniensegment in 2D
  Für ein Liniensegment besteht die Distanz-Funktion aus drei Teilen: der Linie selbst und aus zwei Eckpunkten. Die Funktion für die Linie selbst ohne die Endpunkte ist ein aufgespanntes Zelt (Abbildung 4 (b)). Diese Funktion muss nicht approximiert werden und gibt die Distanzen genau an. Die Eckpunkte hingegen werden wie normale Punkte behandelt, so dass die Zeltfunktion an beiden Seiten durch einen Kegel abgeschlossen wird. An den Eckpunkten entstehen, wie oben beschrieben, entsprechende Fehler bei der Approximation.

- Polygone oder Vielflächer
  Ein Polygon kann man auffassen als eine Komposition aus mehreren Liniensegmenten, es besteht also aus Linien und Punkten. An den Punkten wird nur ein Teil des Kegels gerendert, so dass man viele Dreiecke bei der Berechnung spart und einen Dreieckfächer erhält.

- Polygone in 3D
  Bei Voronoi-Diagramme in 3D werden die Ebenen in Z-Richtung, also die Ebenen parallel zur XY-Ebene einzeln nacheinander gerendert. Ein Polygon besteht wiederum aus sieben einzelnen Teilen, drei Punkten, drei Linien und dem Polygon selbst. Die Funktion für das Polygon selbst erhalten wir, wenn wir die Polygonfläche orthogonal durch den Raum schieben. An jedem Punkt der Polygonfläche ist so der Abstand zu dessen projizierten Punkt auf der XY-Ebene minimal. Für jede XY-Ebene erhalten wir

so eine neue Voronoi-Fläche, auf die wir unsere obigen 2D-Funktionen anwenden können.

Abbildung 6 : Polygon als Voronoi-Figur wird orthogonal durch den Raum verschoben

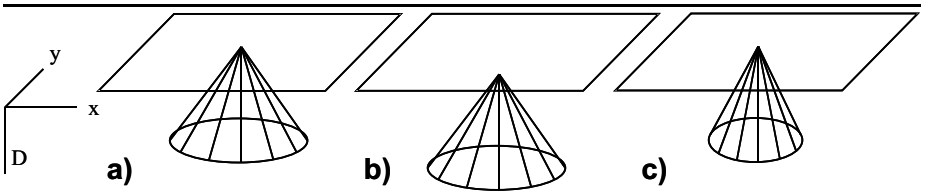


Natürlich gibt es auch Kurven wie Splines als Voronoi-Formen. Diese müssen dann als Linien approximiert werden, wobei wiederum Fehler entstehen können. Mit diesen Funktionen lassen sich nun allgemeine Voronoi-Diagramme berechnen. Die Ergebnisse werden in den Color-Buffer geschrieben und können von dort ausgegeben werden.

4.3 Gewichtete Voronoi-Diagramme

In einem gewichteten Voronoi-Diagramm sind die Distanzfunktionen additiv oder multiplikativ gewichtet. Verschiebung des Distanznetzes entlang der Distanzaxe führt zu additiver Gewichtung. Lineare Skalierung entlang der Distanzaxe führt zu multiplikativer Gewichtung. Dies ist gleich bedeutend mit der Vergrößerung oder Verkleinerung des Winkels des Kegels oder Zeltes in 2D.

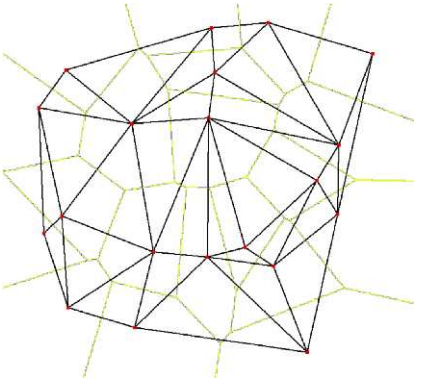
Abbildung 7 : Gewichtete Distanzfunktion für Punkt in 2D. (a) Funktion ohne Gewichtung, (b) additiv gewichtete Funktion als verschobener Kegel, (c) multiplikativ gewichtete Funktion als Kegel mit kleinerem Winkel



4.4 Ränder und Nachbarn (Delaunay)

Um nun die Ränder zwischen den Voronoi-Bereichen zu finden, braucht man nur noch zwei nebeneinander liegende Pixel zu vergleichen, ob diese unterschiedlich gefärbt sind. Dies kann leicht mit der Graphikhardware untersucht werden mit der Min-Max Funktion in OpenGL. Wenn ein solches Pixelpaar gefunden wurde, ist es sinnvoll, nur noch die daneben liegenden Pixelpaare zu untersuchen.

Abbildung 8 : Delaunay-Triangulierung zu obigem Voronoi-Diagramm [4]



Allerdings gilt es darauf zu achten, ob alle Ränder miteinander verbunden sind. Nach Auskunft der Autoren von [5] sind die Ränder in einem Voronoi-Diagramm mit konvexen Formen immer alle verbunden, so dass auch alle Pixelpaare auf diese Art gefunden werden können. Diese Methode könnte fehlschlagen bei gekrümmten Flächen, wo das Voronoi-Diagramm isolierte Komponenten enthalten könnte.

Der Algorithmus findet ebenfalls alle Paare von Nachbarn. Somit kann man ebenfalls die Delauny-Triangulierung berechnen. Hierbei werden alle Voronoi-Nachbarpaare mit Linien zu einem Graphen verbunden. In Abbildung 8 sehen wir die Delaunay-Triangulierung des obigen Voronoi-Diagramms. Die hellen Linien stellen dabei die Ränder der Voronoi-Bereiche dar und die dunklen Linien zeigen die zugehörige Delaunay-Triangulierung.

4.5 Fehlerabschätzung

Wie schon mehrmals erwähnt entstehen durch die Approximation der einzelnen Funktionen viele Fehler. Diese gilt es nun abzuschätzen und so gering wie möglich zu halten. Es gibt zwei Arten von Fehlern: Zum einen den bei der Approximation der Distanzfunktion und zum anderen die Fehler in der Berechnung.

Die Fehler bei der Distanzberechnung können nur durch eine höhere Anzahl von Dreiecken reduziert werden. Genauso verhält es sich bei der Approximierung von Kurven in Liniensegmente. Die evtl. vorkommenden Fehler, verursacht durch die Hardware können nicht behoben werden. Sie entstehen während der Rasterisierung durch die Benutzung von Integer-Datentypen, sie sind jedoch sehr gering gegenüber den zuvor genannten Fehlerquellen.

Bei der Berechnung können drei Arten von Fehlern auftreten. Approximierungsfehler kann man wie oben beschrieben verbessern. Außerdem können auch noch Fehler auftreten, wenn die Auflösung nicht hoch genug gewählt wird. Dies kommt bei diskreten Diagrammen, wie wir sie hier betrachten, desöfteren vor. Dadurch könnten einzelne Voronoi-Bereiche zu klein oder gar nicht angezeigt werden. Als letzte Fehlerquelle sollte man noch den Z-Buffer berücksichtigen.

Dieser ist in seiner Präzision begrenzt auf 24 oder 32 Bit. Wenn die Distanz zweier Pixel niedriger ist als die Präzision des Z-Buffers, kann unter Umständen die falsche Farbe ausgewählt werden. Dies ist zwar im Vergleich nur ein kleiner Fehler, sollte aber bei vielen nebeneinander liegenden Formen berücksichtigt werden. Bei Bedarf kann dieser Z-Buffer jedoch softwaremäßig eine höhere Präzision erhalten, z. B. kann der Treiber den Z-Buffer emulieren. Dies verlangsamt die gesamte Berechnung aber erheblich und ist daher nicht sehr sinnvoll.

4.6 Anwendungsbeispiele

Ein typisches Anwendungsbeispiel für Voronoi-Diagramme ist das Problem des Motion-Planning. Es tritt sehr häufig in der Robotik auf. Dort begegnet man sehr oft der Aufgabe einen Weg durch einen Raum mit Gegenständen zu finden, ohne irgendwo anzustoßen. Hier können die Gegenstände als Voronoi-Formen angesehen werden. Die Ränder um die Voronoi-Bereiche bilden dann den Weg für den Roboter durch den Raum. Auf diesem Weg hält der Roboter die maximale Distanz zu allen Gegenständen. So lässt sich auch überprüfen, ob der Roboter zwischen zwei Gegenständen hindurch passt oder nicht, wenn die Distanz zwischen zwei Gegenständen kleiner ist als die Größe des Roboters.

Auch das Konzept der Delaunay-Triangulierung ist für viele Arten von Applikationen nützlich, wie z. B. in der Kristallographie, in der Geographie und in der Metallurgie. Der Kristallograph Delaunay verwendete diese Struktur für die Simulation von Kristallwachstum sowie zur Beschreibung und Untersuchung von Kristallstrukturen. Weitere geographische Anwendungen finden sich in der Kartographie und in der Stadtplanung. In der Computer-Graphik eignet sich die Delaunay-Triangulierung besonders gut für die Visualisierung und Modellierung von geometrischen Objekten, wie z. B. Freiformflächen.

5 Beispiel 2: Schnelle Matrixmultiplikation

Das zweite Beispiel beschäftigt sich mit der Multiplikation von Matrizen. Hierbei werden die einzelnen Elemente einer Matrix als Farbwerte gespeichert. So kann man die Matrizen als Pixelbild darstellen.

5.1 Der Algorithmus

Zur Matrixmultiplikation werden oft viele Prozessoren parallel benutzt um Teilmatrizen zu berechnen und diese dann zum eigentlichen Ergebnis aufzusummieren. Unser Ansatz funktioniert mit dem selben Prinzip. Angenommen wir wollen zwei $n \times n$-Matrizen multiplizieren. Zuerst wird die erste Matrix in ein Pixelbild umgewandelt und im Color-Buffer gespeichert. Danach legen wir das Bild horizontal in eine Ebene, kopieren die Matrix $n-1$-mal und legen sie ebenfalls horizontal über die erste Matrix. Als Ergebnis erhalten wir einen Würfel (siehe Abbildung 9). Danach wird die zweite Matrix transponiert und ebenfalls umgewandelt. Diese Matrix wird nun vertikal seitlich $n-1$-mal kopiert und

Abbildung 9 : Matrix 1 wird horizontal kopiert

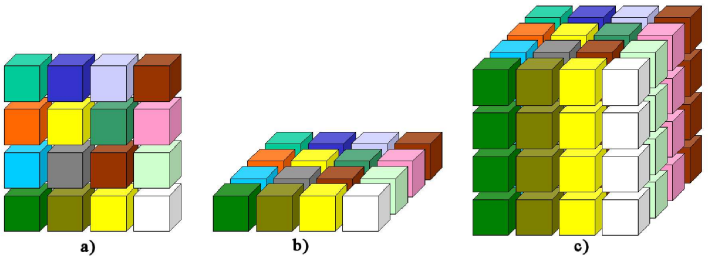
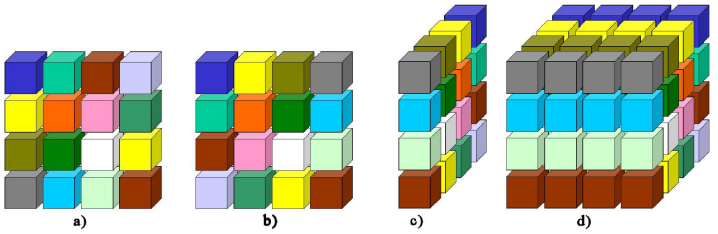


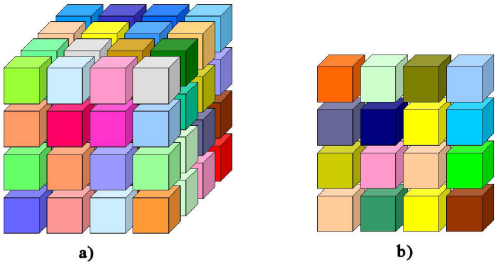
Abbildung 10 : Matrix 2 (a) wird nach dem Transponieren (b) seitlich (c) nebeneinander kopiert (d)



als Würfel dargestellt (siehe Abbildung 10). Nun werden die Würfel übereinander gelegt und die Elemente komponentenweise multipliziert. Das Multiplizieren geschieht wie oben beschrieben durch Multi-Texturing (Abbildung 11 (a)). Danach werden alle Elemente in Z-Richtung zur Front-Ebene durch Blending aufaddiert. Die Front-Ebene, die Ebene, die wir am Bildschirm sehen, zeigt uns das Ergebnis (Abbildung 11 (b)). Das erhaltene Bild muss jetzt nur noch den Farbwerten entsprechend in eine Matrix umgewandelt werden. Durch Texturing- und Blending-Operationen werden die Daten im Color-Buffer verändert und anschließend in den Frame-Buffer ausgegeben.

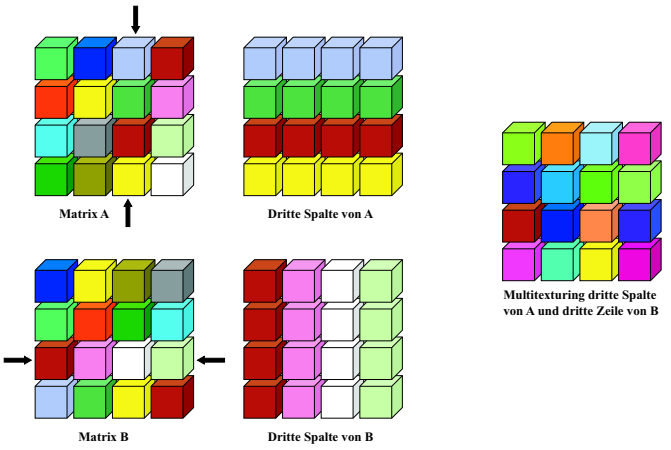
Eine Verdeutlichung der Vorgehensweise zeigt Abbildung 12. Hier sehen wir zwei Matrizen A und B. Die Pixel-Daten der Matrizen stehen im Color-Buffer. Als Beispiel werden nun die dritte Spalte von A und die dritte Zeile von B

Abbildung 11 : Ergebnis nach Multiplikation (a) und Addition (b)



verwendet. Aus der Spalte von A erhalten wir durch waagerechtes nebeneinander kopieren, und aus Zeile B durch senkrechtes kopieren, Texturen für ein Rechteck. Nun wird eine Fläche gerendert, die beide Texturen erhält und mittels Multi-Texturing im Modus *modulate* kombiniert werden. In der Abbildung rechts sehen wir das Ergebnis, nach dem Kombinieren der Texturen. Diesen Vorgang führen wir z mal aus und rendern z parallele Rechtecke hintereinander, so dass wir nur das vordere am Bildschirm sehen können. Nach dem Aufaddieren der Rechtecke sehen wir das Ergebnis.

Abbildung 12 : Matrixmultiplikation mit Graphikhardware. Zwei Matrizen A und B. Die dritte Spalte von A wird waagerecht kopiert, und die dritte Zeile von B wird senkrecht kopiert. Rechts sehen wir das Ergebnis nach dem Multiplizieren durch Multitexturing.



5.2 Ergebnisse und Performance

Nun gilt es, die Geschwindigkeit und Effizienz zu beurteilen und mit einer normalen Berechnung von der CPU zu vergleichen. In mehreren Testläufen wurde eine nVidia GeForce3 benutzt. Diese besitzt eine Präzision von 16 Bit. Dies ist jedoch für viele Arten von Anwendungen keinesfalls ausreichend. Angenommen, wir müssen z Werte addieren. k sei die Anzahl der Bits um z zu repräsentieren. Dann benötigt die korrekte Addition dieser Werte $k + 16$ Bit, wobei jedoch nur 16 Bit gegeben sind. In der Computergraphik stellt dies kein Problem dar. Dort werden einfach die maximalen Werte übernommen. Ein Versuch dieses Problem zu lösen wäre, die Polygonfarbe auf $1/z$ zu setzen. Damit werden die letzten k Bit herausgeschiftet. Diese Lösung verringert jedoch die Genauigkeit um genau diese k Bit und ist damit eine sehr schlechte Lösung. Daher können nur Anwendungen berechnet werden, für die 16 Bit genügen, um reelle Vergleiche mit der CPU zu erhalten.

Als Maßeinheit zum Vergleichen eignen sich Bops (byte operations per second) am besten, da mit den verwendeten Graphikkarten noch nicht mit Gleitkomma-

zahlen gearbeitet werden kann. Bei der nächsten Generation von Graphikkarten wird dies der Fall sein und die Werte, die hier in Bops erreicht wurden, können ebenso in Flops (floatingpoint operations per second) angenommen werden. Nun müssen alle Schritte, die die Graphikkarte durchläuft zusammengezählt werden. Diese sind Konvertieren der Matrix in eine Texture-Map, Senden der Textur zur Graphikhardware, die eigentliche Berechnung und das Auslesen des Framebuffers also das Kopieren der Daten in den Hauptspeicher. Mit einer GeForce3 erhalten wir eine Geschwindigkeit von 4.4GBobs. Dieses Ergebnis übertrifft die Geschwindigkeit eines P4 mit SSE2 Erweiterung. Dieser erreicht 4.0 Gflops [2].

Die Gründe für die höhere Geschwindigkeit liegen hauptsächlich an dem Speichersystem und der spezialisierten Bearbeitung in der Graphikhardware. Größere Matrizen können nicht mehr im 2nd Level Cache der CPU gespeichert werden. Deshalb können solche großen Matrizen nur nach und nach aus dem Speicher gelesen werden. Dies funktioniert bei Graphikkarten viel schneller, da diese einen für dieses Problem spezialisierten Streaming-Speicher besitzen. Dort beträgt die Speicher-Bandbreite 128 Bit und kann mit doppelter Geschwindigkeit pro Taktrate ausgelesen werden. Bei einer GeForce3 beträgt die Taktrate 233 MHz, was eine Übertragung von 7.4 GB pro Sekunde bedeutet. Eine AMD Athlon CPU mit 64 Bit und 133 MHz erlaubt nur eine vergleichsweise geringe Datenübertragung von 2.1 GB pro Sekunde. Ein P4 mit PC800 schafft sogar nur 1.5 GB/s [8].

Beim Auslesen des Speichers sind Graphikkarten überlegen. Jedoch besitzt eine CPU eine deutlich höhere Taktrate. Die Taktrate einer CPU, wie sie hier verwendet wurde, beträgt über 1,4GHz, die einer Graphikkarte allerdings nur 200MHz. Dieser Umstand ist jedoch nicht ausschlaggebend, da das oben genannte Problem, das des Auslesens des Speichers, dieses Problem dominiert. Daher kann die CPU ihre hohe Taktrate nicht voll ausnutzen.

6 Programmierbare Graphikkarten

In den vorherigen Kapiteln wurde beschrieben, wie man nicht programmierbare Graphikkarten benutzen kann und wurden Anwendungen gezeigt, die nicht nur mit der Bilderzeugung zusammen hängen.

Seit der neuesten Generation von Graphikkarten ist es allerdings nun möglich die GPU zu programmieren. Dies geschieht zum Beispiel mit einer C-ähnlichen Sprache Cg. Cg bildet eine Art Schnittstelle zwischen OpenGL bzw. DirectX und der Graphikhardware und vereinfacht die Graphikprogrammierung erheblich. Durch diese Technik ist es möglich, kleine Programme, Vertex- oder Pixelshader, zu schreiben, und diese dann auf der GPU ablaufen zu lassen. Dadurch erhalten wir auch eine variable Graphikpipeline. Desweiteren besitzen moderne Graphikkarten nun Gleitkommazahlen.

7 Zusammenfassung und Ausblick

Wie wir gesehen haben, können Graphikkarten mehr, als nur Bilder erzeugen. Sie lassen sich auch hervorragend zur Berechnung mathematischer Funktionen

nutzen. Dabei wurde sogar, bei einigen Einschränkungen, ein Geschwindigkeitsvorteil gegenüber der CPU erzielt. Aber die Einschränkungen sind immer noch viel zu hoch. Mit der Graphikkarte können keine sehr genauen Berechnungen angestellt werden, da der Z-Buffer und die Farbtiefe eine viel zu geringe Präzision aufweisen. Auch die Taktrate der CPU ist viel höher als die der GPU. Einen Geschwindigkeitsvorteil erhält die Graphikhardware nur durch die Beschaffenheit des Speichers und dessen viel höheren Geschwindigkeit beim Auslesen der Daten, da diese für solche Zwecke spezialisiert wurde. Erst wenn all diese Einschränkungen verbessert werden, könnte die GPU konkurenzfähig werden.

Literaturverzeichnis

[1] F. Aurenhammer. Voronoi-diagrams: A survey of a fundamental geometric data structure. *Computing Surveys*, 23:345–405, 1991.

[2] Jack Dongarra. An update of a couple of tools:atlas and papi. In *DOE Salishan Meeting April 2001*, 2001.

[3] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice.* Addison Wesley, second edition, 1996.

[4] Christian Icking, Rolf Klein, Peter Köllner, and Lihong Ma. Voro glide. `http://wwwpi6.fernuni-hagen.de/GeomLab/VoroGlide/`. 22. 01. 2004.

[5] Kenneth E. Hoff III, Tim Culver, John Keyser, Ming Lin, and Dinesh Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH 1999*, pages 277–286, 1999.

[6] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing 2001, Conference on High Performance Networking and Computing*, pages 55–60, 2001.

[7] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.5)*, 2003.

[8] Chris Trendall and A. James Stewart. General calculations using graphics hardware, with applications to interactive caustics. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 287–298, 2000.

[9] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide.* Addison-Wesley Longman, Inc., third edition, 1999.

Thomas Schultz
Betreuer: Philipp Lucas

GPUs für spezielle Berechnungen

Die auf modernen Grafikkarten eingesetzten GPUs (graphics processing units) sind programmierbar und besitzen eine hohe Rechenleistung bei Gleitkomma-Operationen mit bis zu 32 Bit Genauigkeit. Diese Ausarbeitung stellt zwei Ansätze vor, diese Leistung der allgemeinen Programmierung zugänglich zu machen: Zum einen erlaubt es ein spezielles Framework, die GPU als Vektorrechner zu betrachten; zum anderen führen zwei Bibliotheken auf der GPU Operationen der Linearen Algebra aus. Die Geschwindigkeit der Berechnungen übertrifft z. T. die auf modernen CPUs erreichbare und lässt eine weitere Forschung in diesem Bereich lohnenswert erscheinen. Wir vergleichen die einzelnen Ansätze und nehmen kritisch zu den Ergebnissen Stellung. Außerdem gehen wir kurz auf Weiterentwicklungen der Hardware ein, die für allgemeine Berechnungen relevant sind.

1 Einführung

Moderne GPUs verfügen mittlerweile über eine enorme Rechenleistung. [8] weist darauf hin, dass der Chip der NVIDIA GeForce3 mehr Transistoren enthält als ein Intel Pentium 4. Es ist zu erwarten, dass dieser Unterschied in Zukunft noch zunehmen wird: Laut [7] übersteigt der Leistungs-Zuwachs bei Grafikkarten derzeit mit einem jährlichen Faktor von 2,4 Moores Gesetz.

Schon früh kamen Ideen auf, diese Leistung nicht nur für die Grafikausgabe zu verwenden: Die Autoren von [1] umreißen die Geschichte der Nutzung von Grafikhardware zu allgemeinen Zwecken – zunächst für Anwendungen, die noch in einem Zusammenhang zur Computergrafik standen, dann jedoch auch ganz davon unabhängig für numerische Berechnungen. Ein Beispiel dafür ist der in [4] vorgeschlagene Einsatz von Grafikkarten zur Matrixmultiplikation.

Zwei Neuerungen aktueller Grafik-Hardware bieten diesem Forschungsbereich neue Möglichkeiten: [8] hebt hervor, dass die Funktionalität moderner GPUs nicht mehr starr vorgegeben ist. Statt dessen können sie über sogenannte Vertex und Fragment (bzw. Pixel) Shader in weitem Rahmen programmiert werden.

Zudem stellt die neueste Generation von GPUs Farben nicht mehr mit Festkommawerten sehr begrenzter Genauigkeit dar, sondern ermöglicht Rechnungen mit bis zu 4 Byte langen Gleitkommazahlen (single float precision). [6] weist darauf hin, dass diese Genauigkeit für viele allgemeine Zwecke ausreichend ist.

Die vorliegende Ausarbeitung beschäftigt sich mit zwei speziellen Ansätzen, diese Möglichkeiten zur allgemeinen Programmierung auszunutzen: In Abschnitt 2 gehen wir auf die Arbeit [8] von Thompson et al. ein. Sie stellen ein Framework bereit, das es dem Programmierer erlaubt, die Vertex-Einheit einer GPU als Vektorrechner zu betrachten. Abschnitt 3 beschreibt die Arbeiten [3] von Krüger und Westermann sowie [6] von Moravánszky; sie verwenden die GPU, um eine Bibliothek zu implementieren, die allgemeine Operationen der Linearen Algebra zur Verfügung stellt.

Die Autoren haben zum Teil sehr beeindruckende Ergebnisse vorzuweisen: In [8] wird berichtet, dass große Matrizen (1500^2 Elemente) auf einer GeForce4 Ti4600 um den Faktor 3,2 schneller multipliziert werden können als auf einem Pentium 4 mit 1,5 GHz. Sowohl [3] als auch [6] führen auf der GPU Algorithmen zur Lösung linearer Gleichungssysteme aus, die es ihnen ermöglichen, komplexe physikalische Simulationen ablaufen zu lassen und in Echtzeit zu visualisieren.

Im weiteren Verlauf der Ausarbeitung werden wir die Ansätze vergleichen (Abschnitt 4), kurz auf mögliche Verbesserungen an der Hardware sowie den aktuellen Stand der Technik eingehen (Abschnitt 5) und in Abschnitt 6 schließlich die beschriebenen Ergebnisse zusammenfassen und kritisch reflektieren.

2 Die GPU als Vektorrechner

Als erste praktische Anwendung betrachten wir das in der Arbeit [8] beschriebene Framework von Thompson et al.; Ausgangspunkt ihrer Forschung ist die Beobachtung, dass moderne Grafikchips Fähigkeiten besitzen, die denen allgemeiner Vektorprozessoren sehr ähnlich sind – programmierbare Vertex und Fragment Shader bieten einen Befehlssatz, der mächtig genug ist, um auch allgemeine Programmierung zu erlauben.

Wie auch in [1] beschrieben, besitzen GPUs dabei in der Tat zwei Merkmale, die sie als SIMD-Architektur (single instruction, multiple data) charakterisieren: Erstens werden dieselben Vertex bzw. Fragment Shader auf einen ganzen Strom von Eingabedaten angewandt. Zweitens bieten die GPUs Maschinenbefehle, die auf Vektoren (in der Regel bestehend aus 4 Gleitkommawerten) arbeiten.

Bislang stehen in Rechnern des Desktop-Bereichs außer den SIMD-Erweiterungen moderner CPUs (MMX/SSE von Intel, 3DNow von AMD, AltiVec von Motorola) keine Vektorprozessoren zur Verfügung und es ist unwahrscheinlich, dass in diesem Bereich ein hinreichend großer Markt entstehen könnte, der die Entwicklung von Spezialchips wirtschaftlich erscheinen ließe.

Das übergeordnete Ziel der Arbeit ist daher der Versuch, die GPU als Koprozessor für allgemeine Zwecke nutzbar zu machen; im Idealfall könnte sie überall dort eingesetzt werden, wo ein traditioneller Vektorrechner geeignet wäre. Als ersten Schritt dazu entwickeln Thompson et al. ein Framework, das dem Anwender die Vertex-Programme moderner GPUs als Mittel zur Berechnung von Vektorfunktionen präsentiert.

Die Entscheidung, sich auf die Nutzung von Vertex-Programmen zu beschränken, begründen die Autoren damit, dass OpenGL keine Assemblersprache zur Programmierung von Fragment Shadern spezifiziert.[1] Ein weiterer Grund für ihre Entscheidung ist die Genauigkeit der Berechnungen: Die Fragment Shader der von ihnen eingesetzten Hardware verarbeiten lediglich 10-bit Festkommazahlen, während die Vertex Shader für 16-bit Gleitkommazahlen geeignet sind.

Als langfristiges Ziel betrachten Thompson et al. die Entwicklung moderner Compiler für GPUs. Als Beispiel solcher Bemühungen nennen sie die bereits existierende Programmiersprache Cg, befürworten jedoch auch komplexere Compiler, die sich bestehende Ergebnisse der Vektorrechner-Forschung zunutze machen und die Gelegenheit bieten, diese Forschung zu vertiefen.

Zudem sind sie der Ansicht, dass GPUs dazu beitragen können, den Bedarf nach kostengünstiger Rechenleistung zu decken, der in der zunehmenden Beliebtheit von Rechner-Clustern zum Ausdruck kommt. Sie halten es daher für sinnvoll, die Möglichkeiten des allgemeinen Einsatzes von GPUs in Netzwerk-Clustern zu untersuchen und Hardware zu entwerfen, die mehrere GPUs aufnehmen kann.[2]

2.1 Das Framework und seine Komponenten

Nun sehen wir uns an, wie das von Thompson et al. bereitgestellte Framework aufgebaut ist und wie seine praktische Anwendung aussehen kann. Die Abstraktion von der Hardware ist sehr gering – vor dem Anwender verborgen wird lediglich die Kommunikation mit der Grafikhardware (sie erfolgt intern über OpenGL) und die Verwaltung des Grafikspeichers.

Zur Darstellung von Vektoren, die im Grafikspeicher abgelegt werden sollen, definiert das Framework die Klasse **DVector.** Größe und Typ (wahlweise Gleitkomma-Zahlen einfacher Genauigkeit oder vorzeichenlose Bytes) können bei ihrer Initialisierung festgelegt werden.

Die eigentlichen Assembler-Programme werden in der Klasse **DProgram** gekapselt. Beim Schreiben des Codes kann der Anwender davon ausgehen, dass die Argumente der Funktion in den Eingaberegistern bereitgestellt sind und dass es genügt, das Ergebnis in ein Ausgaberegister zu schreiben. Das Laden der Register und Auslesen des Ergebnisses erledigt das Framework für ihn.

Die Klasse **DFunction** stellt eine vollständige, auf der GPU ausführbare Vektorfunktion dar. Ihre Objekte enthalten neben dem DProgram mit dem benötigten Programmcode auch Belegungen für die konstanten Register; außerdem wird hier die Semantik der Funktion festgelegt, d. h. welche Zahl von Parametern sie besitzt (unär, binär oder ternär) und ob ihre Auswertung einen Vektor oder einen skalaren Wert ergibt.

Ihre Methode execute() führt die Funktion auf der Grafikkarte aus. Je nach gewählter Semantik besitzt sie ein bis drei Parameter des Typs DVector und liefert einen DVector oder einen Skalar zurück. Da die von der Grafik-Hardware verarbeiteten Vektoren immer 4 Komponenten besitzen (ursprünglich sind sie zur Darstellung von RGBA-Farbwerten bzw. homogenen Koordinaten bestimmt),

[1] Diese Situation hat sich seit Einreichung ihres Papers geändert.
[2] Diesbezüglich geht Abschnitt 5 auf den aktuellen Stand der Technik ein.

muss das Assembler-Programm unter Umständen mehrfach ausgeführt werden, um das Ergebnis zu errechnen.

Bei vektorwertigen Funktionen werden in jedem Durchlauf (je nach Semantik) ein bis drei der Attributregister v[1]–v[3] mit jeweils vier aufeinanderfolgenden Werten aus den übergebenen DVector-Objekten gefüllt. Das Assembler-Programm liest sie von dort ein und schreibt das Ergebnis in das Ausgaberegister o[1]. Der Rückgabewert wird automatisch aus den $\lceil \frac{n}{4} \rceil$ Teilergebnissen zusammengesetzt, so dass der Anwender beim Schreiben des Assembler-Programms diese Einschränkung der Hardware nicht berücksichtigen muss. Eine Ausnahme bilden hier die Opcodes, deren Eingabe ein Skalar ist (wie etwa `LOG` und `EXP`).

Da die Hardware es gegenwärtig nicht erlaubt, zwischen zwei Aufrufen des Assembler-Codes den Zustand der Register beizubehalten, müssen die Teilergebnisse bei der Berechnung skalarwertiger Funktionen mit der CPU aufsummiert werden. In diesem Fall werden möglichst viele der Eingaberegister ausgenutzt, z. B. verarbeitet ein einziger Durchlauf des Assembler-Programms bei der Berechnung einer unären Funktion 60 ($= 4 \cdot 15$) Komponenten des Eingabevektors, die in den Registern v[1] bis v[15] abgelegt werden.

Ein Vorteil der Nutzung von Grafik-Prozessoren als Koprozessor besteht darin, dass sie während ihrer Berechnungen keine Kontrolle durch die CPU benötigen und daher beide Prozessoren gleichzeitig verwendet werden können. Die Klasse **DSemaphore** ermöglicht in diesem Fall die Synchronisation zwischen CPU und GPU. Sie bietet eine Methode, die vom CPU-Code aus aufgerufen werden kann, wenn er die Ergebnisse einer GPU-Berechnung benötigt und die erst dann zurückkehrt, wenn diese zur Verfügung stehen.

Ein Codebeispiel der Autoren

Um die Arbeitsweise mit dem Framework von Thompson et al. anschaulicher zu machen, betrachten wir in Abbildung 1 ein Codefragment aus ihrem Paper [8].[3] Sie stellen darin eine unäre, vektorwertige Funktion vor, die die Fakultät der Komponenten eines Vektors berechnet. In dem Beispiel wird sie auf einen 5 Komponenten umfassenden Vektor angewandt.

Bemerkenswert ist der Aufwand, der getrieben werden muss, weil der Befehlssatz der verwendeten Grafikhardware keine bedingten Sprünge oder Schleifen ermöglicht (zum aktuellen Stand der Technik siehe Abschnitt 5). Deshalb ist der Inhalt der Schleife zur Berechnung der Fakultät achtmal ausgeschrieben. Dies hat zur Folge, dass einerseits nur die Fakultäten einstelliger Zahlen berechnet werden können und die GPU andererseits in vielen Fällen weitere (unnötige) Rechenschritte ausführt, obwohl das Ergebnis bereits vorliegt.

Um die Komponenten, in denen das Endergebnis bereits vorliegt, von denen unterscheiden zu können, die weiter multipliziert werden müssen, werden in den Registern R3 und R4 komplementäre Bitmasken erzeugt (Zeilen 19 und 20). Mit ihrer Hilfe werden die Komponenten berechnet, die in diesem Schritt geändert wurden bzw. gleich geblieben sind und ab Zeile 27 in die Register R7 und R8 geschrieben. Schließlich können die beiden Vektoren zum Ergebnis des jeweiligen Schritts aufaddiert werden.

[3] Aus Layout-Gründen weicht die hier wiedergegebene Form leicht vom Original ab.

Die verwendeten Opcodes sind `MOV` (move), `SGE` (set on greater/equal than), `SLT` (set on less than), `MUL` (komponentenweise Multiplikation) und `ADD` (Addition). In das erste angegebene Register wird das Ergebnis der Operation geschrieben, aus den folgenden werden die benötigten Argumente gelesen.

Abbildung 1 Codebeispiel zur Verwendung des Frameworks von Thompson et al.: Berechnung der Fakultät einstelliger Zahlen.

```
    DVector* inputVec  =
       allocDVector(5, DV_UCHAR);
    DVector* outputVec =
       allocDVector(5, DV_UCHAR);

    // Fill the test input vector.
    (*inputVec)[0] = 6;
    (*inputVec)[1] = 2;
    (*inputVec)[2] = 3;
    (*inputVec)[3] = 1;
    (*inputVec)[4] = 4;

    char* program[] = {
      "MOV R1, v[1];",
      "MOV R2, v[1];",

      // If R2>=2 decrement R2.
      // If R2< 2 leave unchanged.
      "SGE R3, R2,  c[12];",
      "SLT R4, R2,  c[12];",
      "MUL R5, R3, -c[11];",
      "ADD R2, R2,  R5;",

      // Multiply the updated values
      // into R1.
      "MUL R6, R3, R2;",
      "MUL R7, R1, R6;",
      "MUL R8, R1, R4;",
      "ADD R1, R7, R8;",

      // Unroll once.
      "SGE R3, R2,  c[12];",
      "SLT R4, R2,  c[12];",
      "MUL R5, R3, -c[11];",
      "ADD R2, R2,  R5;",
      "MUL R6, R3, R2;",
      "MUL R7, R1, R6;",
      "MUL R8, R1, R4;",
      "ADD R1, R7, R8;",
      ...
      // Unroll six more times.

      // Store the result in
      // the output register.
      "MOV o[COL0], R1;",
      0 };

    DFunctionUnary programFunc
       ( makeProgram( program ) );
    programFunc.setConstant( 11, 1 );
    programFunc.setConstant( 12, 2 );

    programFunc.execute( inputVec,
                         outputVec );
```

2.2 Anwendungsbeispiele

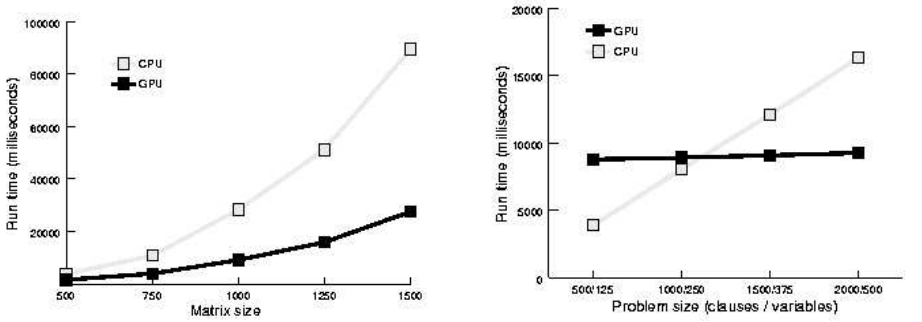
Um die Rechenleistung der GPU für praktische Anwendungen abzuschätzen, haben Thompson et al. die von einer Reihe von Testprogrammen benötigte Zeit mittels der Systemuhr gemessen (Auflösung von 10 ms). Ihre Messungen beinhalten die zur Datenübertragung notwendige Zeit. Zum Vergleich wurden ähnliche Messungen auch für entsprechende auf der CPU implementierte Algorithmen durchgeführt. Dabei kam als GPU eine GeForce4 Ti4600[4] zum Einsatz; als CPU wurde ein mit 1,5 GHz getakteter Pentium 4 verwendet. Die Programme für die GPU wurden in Assembler, die für die CPU in C++ geschrieben.

Näher eingehen werden wir an dieser Stelle auf die beiden Tests, die realistischen Anwendungen des Frameworks am nächsten kommen: Zum einen ein Algorithmus zur Multiplikation dichter Matrizen, zum anderen ein Programm zur Suche nach Lösungen des 3-SAT-Problems.

Zur Matrixmultiplikation multipliziert eine binäre, skalare DFunction einen Zeilenvektor der ersten mit einem Spaltenvektor der zweiten Matrix; sie wird für

[4] Taktrate vermutlich 300 MHz (Kernsystem) und 650 MHz (Speicher); Quelle: `http://www.3dgpu.com/modules/wfsection/article.php?articleid=60&page=4` (13.2.2004)

Abbildung 2 Die Messergebnise der realistischen Tests aus [8]: Links die Matrix-Multiplikation, rechts der 3-SAT-Löser (100 000 Iterationen).



jedes Element der Ausgabematrix aufgerufen. Die Berechnungen auf der CPU folgen dem gleichen Schema. Wie die linke Seite von Abbildung 2 zeigt, ist die GPU-Implementierung wesentlich schneller als die CPU-Variante. Der Vorsprung wächst überdies mit der Größe der beteiligten Matrizen: Für eine Matrix der Größe 1500×1500 ist die GPU um den Faktor 3,2 schneller als die CPU.

Zur Lösung des 3-SAT-Problems kommt ein inkrementeller Algorithmus für Terme zum Einsatz, die so viele Variablen enthalten, dass ein Test aller möglichen Belegungen nicht durchführbar ist: Dabei wird zunächst für eine zufällige Belegung festgestellt, wie viele Klauseln sie erfüllt. Dann wird der Wert einer zufälligen Variablen invertiert und es werden erneut die erfüllten Klauseln gezählt. Die Änderung wird nur beibehalten, falls sie eine Verbesserung bewirkt hat. Erzielen einige Iterationen in Folge keinen Fortschritt, so wird die Belegung vollständig verworfen und eine neue gewählt. Wenn nach einer festen Zahl von Iterationen keine Lösung gefunden werden konnte, wird die Suche als erfolglos abgebrochen.

Die GPU kann hierbei zur Ermittlung der erfüllten Klauseln eingesetzt werden. Dazu werden die Klauseln als Elemente eines DVectors kodiert; die Belegung der Variablen wird in den konstanten Registern abgelegt. Nun kann eine unäre, skalare DFunction die Anzahl der erfüllten Klauseln feststellen. Abbildung 2 zeigt auf der rechten Seite die Ergebnisse der Messung mit und ohne Verwendung der GPU. Dabei sind alle Probleme unerfüllbar gewählt, um die maximale Zahl von Iterationen zu erzwingen. Die CPU-Implementierung ist bei kleinen Problemen performanter, ihr Zeitaufwand wächst jedoch linear mit der Problemgröße an. Der Zeitbedarf der GPU ist innerhalb des untersuchten Bereichs noch von der Datenübertragung von und zur GPU dominiert und bleibt somit annähernd konstant, so dass ihr Einsatz für große Probleme effizienter ist.

3 Bibliotheken für Lineare Algebra auf der GPU

In diesem Abschnitt beschäftigen wir uns mit einem zweiten Ansatz, moderne GPUs zur allgemeinen Programmierung zu nutzen. Er wird durch die Artikel [3] von Krüger und Westermann sowie [6] von Moravánszky repräsentiert. Ziel beider Arbeiten ist es, allgemeine Operationen aus der Linearen Algebra auf der

GPU zu implementieren und sie dem Benutzer als Bibliothek zur Verfügung zu stellen. Wie wir in Abschnitt 3.3 sehen werden, kommen diese Operationen als Bausteine physikalischer Simulationen zum Einsatz.

Krüger und Westermann nennen einige Beispiele, in denen Algorithmen für spezielle Anwendungen in diesem Bereich auf der GPU implementiert wurden. Die Ergebnisse dieser Arbeiten sind insbesondere dann überzeugend, wenn der simulierte Vorgang zeitgleich visualisiert werden soll: Verwendet man in diesem Fall die GPU zur Ausführung der Berechnungen, liegen die Ergebnisse bereits im Grafikspeicher vor und können unmittelbar verwendet werden.

Ihre Arbeit ist auch softwaretechnisch motiviert: Die Nutzung einer fertigen Bibliothek, wie die Autoren sie vorschlagen, bietet dem Anwender die Möglichkeit, neue Lösungen in kürzerer Zeit umzusetzen. Zudem kann er so die Rechenleistung der GPU nutzen, ohne sich selbst mit ihrer Programmierung auseinandersetzen zu müssen. Dabei nimmt er in Kauf, dass eine vollständig selbst geschriebene Speziallösung möglicherweise effizienter wäre.

Als Vorbild nennen beide Autoren die Bibliothek BLAS (Basic Linear Algebra Subprogram) für Lineare Algebra. Zwar bietet noch keine der Arbeiten den vollen Funktionsumfang dieser Bibliothek, der Ansatz ist jedoch erkennbar der gleiche und Krüger/Westermann betrachten es als langfristiges Ziel, eine GPU-basierte Bibliothek zur Verfügung zu stellen, die zu BLAS kompatibel ist.

3.1 Die Bibliothek von Krüger und Westermann

Um einen Überblick über die Funktionalität zu bekommen, die Krüger und Westermann bereitstellen, sehen wir uns zunächst die C++-Schnittstelle ihrer Bibliothek an:

- **Vektorarithmetik**

  `void clVecOp (CL_enum op, float` α`, float` β`, clVec x, clVec y, clVec res)`

  Diese Methode führt eine Berechnung der Form $\mathbf{res} := \alpha\mathbf{x}\ \mathtt{op}\ \beta\mathbf{y}$ aus. Die Operation `op` ist hierbei CL_ADD, CL_MULT oder CL_SUB (komponentenweise Addition, Multiplikation bzw. Subtraktion).

- **Matrix-Vektor-Produkt**

  `void clMatVec (CL_enum op, clMat A, clVec x, clVec y, clVec res)`

  Hier wird die Matrix $\mathbf{A}$ mit dem Vektor $\mathbf{x}$ multipliziert und das Ergebnis mit Vektor $\mathbf{y}$ verknüpft. Die Berechnung hat also die Form $\mathbf{res} := \mathbf{A}\mathbf{x}\mathtt{op}\mathbf{y}$, wobei `op` eine der für die Vektorarithmetik definierten Operationen ist.

- **Reduktion von Vektoren**

  `float clVecReduce (CL_enum cmb, clVec x, clVec y)`

  Diese Methode verrechnet die Vektoren $\mathbf{x}$ und $\mathbf{y}$ zu einem einzigen Skalar. Die ausgeführte Rechnung lässt sich als `cmb` $(\mathbf{x} * \mathbf{y})$ beschreiben, wobei $*$ das komponentenweise Produkt bezeichnet und `cmb` eine skalarwertige Vektorfunktion ist, die aus CL_ADD, CL_MULT, CL_MAX, CL_MIN und CL_ABS ausgewählt werden kann.

  Ein typisches Beispiel für den Einsatz dieser Methode ist das Skalarprodukt zweier Vektoren: Hierzu wird CL_ADD als Wert von `cmb` verwendet.

Interne Darstellung der Matrizen

Krüger und Westermann haben nach einer Darstellung von Matrizen auf der GPU gesucht, die zum einen wenig Speicherplatz benötigt, es zum anderen aber auch ermöglicht, die genannten Berechnungen effizient auszuführen.

Da sie insbesondere die Visualisierung physikalischer Simulationen unterstützen möchten, ist es darüber hinaus wichtig, dass die Ergebnisse mit möglichst wenig Aufwand am Bildschirm dargestellt werden können. Außerdem treten bei derartigen Simulationen häufig große, dünn besetzte Matrizen auf – es ist möglich, die Effizienz weiter zu steigern, wenn es gelingt, diese Eigenschaft bei der Darstellung und Verarbeitung von Matrizen auszunutzen.

Die Grundentscheidung ihres Ansatzes besteht darin, Vektoren und Matrizen als Texturen darzustellen und die Rechenoperationen mit Pixel Shadern auszuführen. Das hat den Vorteil, dass die Ergebnisse wiederum als Textur gebunden werden und somit wahlweise zur Darstellung am Bildschirm oder unmittelbar als Operand einer weiteren Berechnung dienen können.

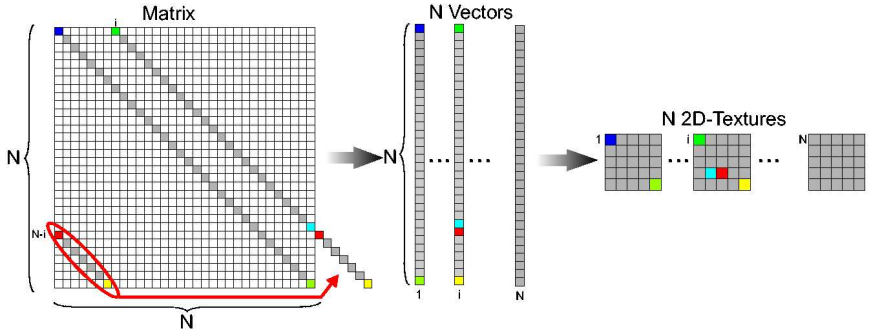
Die nahe liegendste Möglichkeit wäre es, Vektoren als eindimensionale und Matrizen als zweidimensionale Texturen darzustellen. Diese Idee bringt jedoch wesentliche Nachteile mit sich: Die beschränkte Größe eindimensionaler Texturen würde beispielsweise die maximal mögliche Anzahl von Komponenten in einem Vektor einschränken. Experimente zeigen außerdem, dass die Verarbeitung eindimensionaler Texturen bei gleicher Anzahl der Elemente etwa um den Faktor 2 langsamer ist als die zweidimensionaler Texturen. Krüger und Westermann vermeiden diese Probleme, indem sie Vektoren als quadratische Texturen darstellen. Falls die Länge des Vektors keine Quadratzahl ist, werden die übrigen Elemente der Textur mit Nullen aufgefüllt.

Quadratische ($N \times N$ große) Matrizen betrachten sie als Menge von Vektoren, die als jeweils eine 2D-Textur dargestellt werden. Zunächst erscheint es etwas ungewöhnlich, dass sie die Matrix dabei weder in Zeilen- noch in Spaltenvektoren aufteilen, sondern in Diagonalenvektoren. Das Prinzip ist in Abbildung 3 illustriert:[5] Um nicht unnötig viel Speicher zu verbrauchen, legen sie nach jeder Diagonalen, die mit dem i-ten Element der ersten Zeile beginnt, zusätzlich die Elemente der Diagonalen ab, die mit dem $(N-i)$-ten Element der ersten Spalte anfängt. Die Zählung der Variablen i beginnt – auch im Folgenden – bei 0.

Der Grund für diese etwas unintuitive Darstellung liegt in dem oben begründeten Bemühen, dünn besetzte Matrizen effizient zu speichern und zu verarbeiten. Bei den Anwendungen (siehe Abschnitt 3.3) treten beispielsweise Matrizen auf, die ausschließlich in der Hauptdiagonalen und einigen darüber liegenden Diagonalen Werte enthalten; es handelt sich also um sogenannte Bandmatrizen. Betrachtet man eine solche Matrix als Menge von Diagonalenvektoren, müssen die Nullvektoren weder im Speicher abgelegt noch für Berechnungen berücksichtigt werden. Das Produkt der Matrix mit einem Vektor ist auf diese Weise auf wenige Vektor-Vektor-Produkte zurückzuführen. Die gewählte Darstellung erlaubt überdies eine einfache Transposition der Matrizen; Abschnitt 4.2 wird auf diese Tatsache genauer eingehen.

[5] Die Abbildung ist [3] entnommen. Bei genauem Hinsehen fällt auf, dass die Vektoren 32 Komponenten enthalten – das wirft die Frage auf, wie sie in 5×5 großen Texturen gespeichert werden können. Ich halte das schlicht für einen Fehler in der Zeichnung, der das Verständnis des Grundprinzips jedoch nicht beeinträchtigen sollte.

Abbildung 3 Die Aufteilung einer Matrix in Diagonalenvektoren und deren Darstellung als zweidimensionale Texturen.



Um auch solche dünn besetzte Matrizen effizient verarbeiten zu können, die keine Bandmatrizen sind, stellen Krüger und Westermann ein weiteres Konzept vor. Dabei speichern sie die Werte der Matrix nicht mehr in Texturen; statt dessen existiert nun für jeden Spaltenvektor ein Vertex-Array, das ebenfalls im Grafikspeicher vorgehalten werden kann. Es enthält für jeden vorhandenen Wert einen Vertex, in dessen Farbregister er abgelegt wird. Die Koordinaten des Vertex sind so gewählt, dass er an der gleichen Stelle gerendert wird wie das Texel, das bei der bereits bekannten Darstellung des Vektors als Textur verwendet würde. Um auch die Multiplikation der so repräsentierten Matrix mit einem Vektor zu unterstützen, verweist die Texturkoordinate jedes Vertex schließlich auf die Komponente des Vektors, mit der sein Farbwert multipliziert werden muss.

Umsetzung der zur Verfügung gestellten Operationen

Nun, da wir die Darstellung von Matrizen und Vektoren kennen, können wir die Funktionsweise der Pixel Shader genauer betrachten, mit denen Krüger und Westermann die bereits vorgestellten Operationen auf der GPU umsetzen.

Die verwendeten **Datentypen** sind clVec für Vektoren und clMat für Matrizen. Sie zerlegen C++-Arrays in einen oder mehrere Vektoren und übernehmen die Verwaltung der zugehörigen Texturen.

Für die **Vektorarithmetik** ($\mathbf{res} := \alpha\mathbf{x}\,\mathrm{op}\,\beta\mathbf{y}$) werden die Texturen gleichzeitig geladen, in denen $\mathbf{x}$ und $\mathbf{y}$ abgelegt sind. Nun wird ein Quadrat gerendert, das genau die Größe der Texturen abdeckt; dabei kommt ein Fragment Shader zum Einsatz, der die beiden Texturen entsprechend der mit `op` ausgewählten Operation kombiniert. Zur Skalierung werden dem Fragment Shader die Faktoren α und β als Konstanten zur Verfügung gestellt.

Beim **Matrix-Vektor-Produkt** ($\mathbf{res} := \mathbf{A}\mathbf{x}\,\mathrm{op}\,\mathbf{y}$) wird zunächst die Matrix $\mathbf{A}$ mit dem Vektor $\mathbf{x}$ multipliziert. Das Zwischenergebnis kann dann wie im vorhergehenden Absatz beschrieben mit $\mathbf{y}$ verknüpft werden.

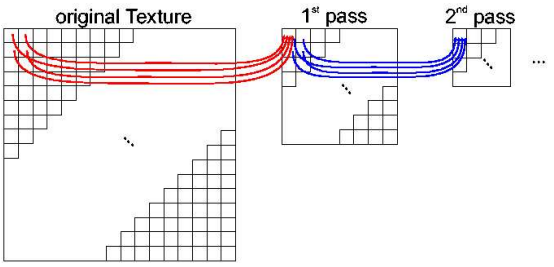
Da die Matrix als Menge von Vektoren vorliegt, lässt sich auch die Multiplikation einer $N \times N$-Matrix mit einem Vektor auf N Vektor-Vektor-Multiplikationen zurückführen: Im einfachsten Fall wird in N Rendering-Durchläufen jeweils eine

Diagonale mit dem Vektor multipliziert. Nachdem die erste Diagonale verarbeitet wurde, wird das Ergebnis in den folgenden Schritten immer zugleich als Textur (also als Eingabe des Fragment Shaders) und als Render Target (Ausgabe des Shaders) verwendet. Das Ergebnis wird also in jedem Schritt aktualisiert, so dass nach dem N-ten Schritt das fertige Endergebnis vorliegt.

Wichtig für den Entwurf des Fragment Shaders ist es, dass er für jede Komponente der aktuellen Diagonale die zugehörige Komponente des Vektors selbst ermitteln muss: Wie man sich etwa anhand von Abbildung 3 klar machen kann, gehört dabei zur j-ten Komponente einer Diagonalen, die in der i-ten Spalte beginnt, die $((i+j) \mod N)$-te Komponente des Vektors. Der Modulo-Operator ist dabei wegen der oben beschriebenen Platz sparenden Darstellung nötig, bei der im Anschluss an jede Diagonale aus der oberen Hälfte der Matrix noch ihr Gegenstück aus der unteren Hälfte abgespeichert wird. i und N bleiben innerhalb eines Rendering-Durchlaufs gleich und können dem Fragment Shader daher als Konstanten übergeben werden. Den Wert von j muss der Shader selbst aus den Koordinaten des Ausgabe-Pixels errechnen.

Um eine möglichst hohe Verarbeitungsgeschwindigkeit zu erzielen, nutzen Krüger und Westermann alle zur Verfügung stehenden Texturierungseinheiten. Sofern möglich verwenden sie mehrere benachbarte Diagonalen gleichzeitig als Textur und können auf diese Weise mehrere Multiplikationen in einem einzigen Rendering-Durchlauf ausführen. Dabei bekommt der Shader i und N weiterhin als Konstanten übergeben; der Wert von j ist für eine feste Komponente des Ausgabevektors in jeder Diagonalen gleich und muss daher nur einmal berechnet werden. Nachdem der Index für die erste Diagonale ermittelt wurde, kann er für die folgenden also einfach um jeweils eins erhöht werden.

Abbildung 4 Illustration des zur Vektorreduktion eingesetzten rekursiven Verfahrens, entnommen aus [3].



Auch die bei der **Reduktion von Vektoren** (`cmb` $(\mathbf{x} * \mathbf{y})$) mögliche Multiplikation $\mathbf{x} * \mathbf{y}$ läuft wie bei der Vektorarithmetik beschrieben ab.

Die skalarwertige Funktion `cmb` selbst wird in mehreren rekursiven Schritten ausgewertet; eine schematische Darstellung des Vorgangs findet sich in Abbildung 4. Hierzu wird in jedem Schritt ein Quadrat gerendert, dessen Kanten nur noch halb so lang sind wie die des Quadrats im vorherigen Schritt. Der Fragment Shader kombiniert für jedes Pixel des Ausgabepuffers vier benachbarte Texel der Eingabe. Bei einem Vektor, der durch eine Textur mit einer Kantenlänge von 2^n Texeln dargestellt wird, sind somit $n-1$ Rendering-Durchläufe nötig, bis auf diese Weise nur noch ein einzelnes Pixel übrig bleibt, dessen Wert ausgelesen und zurückgegeben werden kann.

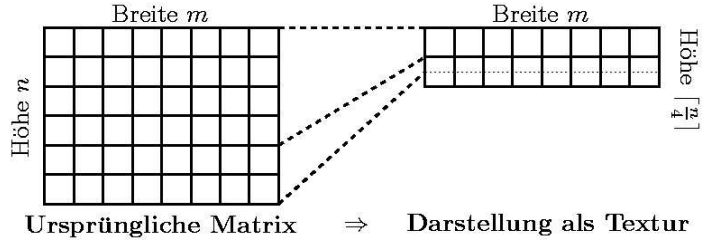
3.2 Alternative Ansätze bei Moravánszky

Auch Moravánszky stellt mit seiner Arbeit [6] eine Bibliothek von Operationen der Linearen Algebra zur Verfügung, die auf der GPU implementiert ist. Der Grundgedanke ist dem von Krüger und Westermann sehr ähnlich. Daher konzentrieren wir uns im Folgenden auf zwei spezielle Aspekte seiner Arbeit: Zum einen die interne Darstellung der Matrizen, die von der bereits vorgestellten Methode abweicht. Zum anderen betrachten wir seine Implementierung der Matrixmultiplikation, da diese Operation bei Krüger und Westermann fehlt.

Interne Darstellung dichter Matrizen

Bei der Suche nach einem geeigneten Datenformat trifft Moravánszky aus den gleichen Gründen wie Krüger und Westermann die Entscheidung, zur Darstellung von Matrizen Texturen zu verwenden und diese mit Pixel Shadern zu bearbeiten. Im Gegensatz zum gerade vorgestellten Ansatz versucht er jedoch nicht, sich die Eigenschaften dünn besetzter Matrizen zunutze zu machen, um sie schneller verarbeiten zu können. Für Vektoren definiert er keinen eigenen Datentyp, sondern betrachtet sie als Spezialfall einer Matrix mit Breite eins; er stellt sie also als Spaltenvektoren dar.

Abbildung 5 Eine $n \times m$-Matrix und ihre interne Darstellung als Textur.



Diese abweichende Zielsetzung führt Moravánszky zur Verwendung eines anderen Datenformats: Er stellt Matrizen als je eine einzige zweidimensionale Textur dar. Um die SIMD-Fähigkeiten der Hardware nutzen zu können, enthält jedes Texel vier aufeinander folgende Werte aus einer Spalte der Matrix, so dass die resultierende Textur zur internen Darstellung einer $n \times m$ großen Matrix die Breite m, aber die Höhe $\lceil \frac{n}{4} \rceil$ hat. Soll eine Matrix dargestellt werden, deren Höhe kein Vielfaches von vier ist, so werden die übrigen Farbwerte mit Nullen aufgefüllt (in Abbildung 5 der Bereich unterhalb der gepunkteten Linie).

Wie wir sehen werden, erschwert diese Darstellung die Multiplikation mit der Transponierten einer Matrix: Das Datenformat erlaubt hier keine einfache Vertauschung der Indizes von Zeile und Spalte, sondern macht eine völlig andere und komplexere Befehlsfolge notwendig.

Die bereitgestellten Operationen und ihre Umsetzung

Moravánszky geht in [6] mit ausführlichen Codebeispielen auf die Implementierung der bereitgestellten Operationen ein; wir beschränken uns an dieser Stelle auf ein Verständnis der wesentlichen Ideen und Funktionsprinzipien.

Generell verwendet Moravánszky (wie auch schon Krüger und Westermann) Point Sampling, da eine Interpolation zwischen benachbarten Werten, wie sie für grafische Anwendungen häufig sinnvoll ist, bei arithmetischen Berechnungen das Ergebnis verfälschen würde. Außerdem überprüfen alle bereitgestellten Methoden, dass die ausgeführten Berechnungen sinnvoll sind und sorgen dafür, dass eine Textur nie zugleich als Ein- und Ausgabe einer Operation dient – obwohl einige Grafikkarten dies unterstützen,[6] ist es in der Direct3D-Spezifikation nicht vorgesehen.

Die **Zuweisung** (Kopie) einer Matrix ist die einfachste Operation: Hierbei wird lediglich ein Viereck gerendert, das die Größe der zu kopierenden Textur genau abdeckt – der Pixel Shader braucht die Werte nur auszulesen und in das Ausgaberegister zu kopieren.

Bei der **Addition** zweier Matrizen müssen zwei Fälle unterschieden werden: Bei der Akkumulation (entspräche in C/C++ dem Operator +=) soll in einem der Operanden zugleich das Ergebnis abgelegt werden. Dies ist als Zuweisung implementiert, bei der additives Blending mit dem Zielpuffer eingeschaltet wird; in diesem Modus addiert die GPU die Ausgabe zu den bestehenden Werten hinzu, statt sie zu ersetzen. Auch der binäre Fall ist der Zuweisung sehr ähnlich: Nun werden beide Eingabematrizen als Textur gebunden; der Pixel Shader liest aus beiden jeweils einen Wert und addiert sie auf.

Die **Multiplikation** zweier Matrizen ($\mathbf{C} := \mathbf{AB}$) ist weniger einfach: Um das Ergebnis eines Elements der Ausgabematrix zu ermitteln, muss das Skalarprodukt eines Zeilenvektors aus $\mathbf{A}$ mit einem Spaltenvektor aus $\mathbf{B}$ gebildet werden. Allerdings ist sowohl die Zahl der Texturzugriffe als auch die der insgesamt in einem Fragment Shader ausführbaren Befehle beschränkt, so dass dieses Skalarprodukt bei großen Matrizen nicht in einem einzigen Pipeline-Durchlauf ermittelt werden kann.

Aus diesem Grund muss die Rechnung in elementare Operationen aufgeteilt werden, die nötigenfalls auf mehrere Durchläufe verteilt werden können. Aufgrund der internen Darstellung der Matrix, bei der vier benachbarte Werte in einem Texel gespeichert werden, bietet es sich an, das Produkt einer 4×4-Matrix (aus dem linken Operanden) mit einem 4×1-Vektor (aus dem rechten Operanden) als eine solche Elementaroperation zu verwenden – Moravánszky bezeichnet sie als MOP (matrix operation).

Bezeichnen wir die innere Dimension (d. h. die Breite der Matrix $\mathbf{A}$, per Definition gleich der Höhe der Matrix $\mathbf{B}$) als q, so sind zur Berechnung jedes Ausgabeelements $\lceil \frac{q}{4} \rceil$ solcher MOPs nötig; ihre Ausgaben werden aufsummiert, um das Endergebnis zu erhalten. In der Regel können in einem Pipeline-Durchlauf mehrere MOPs berechnet werden; die genaue Zahl hängt davon ab, wie viele Befehle, insbesondere Texturzugriffe, der Fragment Shader ausführen kann.

Aus der Zahl der benötigten MOPs und der Anzahl der im Fragment Shader untergebrachten MOPs lässt sich die Zahl der benötigten Pipeline-Durchläufe errechnen. Die Methode zur Matrixmultiplikation rendert eine entsprechende Zahl übereinander liegender Vierecke, die die Ergebnismatrix abdecken und akkumuliert die Einzelergebnisse mittels additivem Blending.

[6] Krüger und Westermann nutzen dies bei ihrer Implementierung des Matrix-Vektor-Produkts aus; siehe Abschnitt 3.1.

Bemerkenswert ist die Arbeitsteilung, die Moravánszky zwischen Vertex und Fragment Shader vornimmt. Die Eckpunkte zur Definition der gerenderten Vierecke sind mit einem Index versehen, der die Nummer des aktuellen Durchlaufs enthält. Der Vertex Shader liest ihn aus und berechnet daraus die 5 Texturkoordinaten (vier für die linke, eine für die rechte Matrix), die der Fragment Shader für die erste MOP dieses Durchlaufs benötigt.

Der Pixel Shader kann sofort mit dem Auslesen der Texturdaten beginnen und anschließend das gewünschte Produkt berechnen. Falls er mehrere MOPs ausführt, muss er erst ab der zweiten MOP selbst einen Offset auf die Texturkoordinaten addieren, um auf die passenden Elemente zugreifen zu können.

Auf eine ausführliche Erklärung der **Multiplikation mit einer Transponierten** soll an dieser Stelle aus Platzgründen verzichtet werden. Es sei lediglich nochmals darauf hingewiesen, dass diese Operation aufgrund des gewählten Datenformats einen komplexere Fragment Shader benötigt. Dem Diagramm in Abbildung 7 ist zu entnehmen, dass diese Operation in der vorliegenden GPU-Implementierung etwa doppelt so viel Zeit benötigt wie die gewöhnliche Matrix-Multiplikation, obwohl beide aus mathematischer Sicht gleich aufwändig sind.

Weitere Operationen, die Moravánszky bereitstellt, beinhalten das Skalarprodukt zweier Vektoren, die Skalierung einer Matrix (Multiplikation mit einer Konstanten), eine Maximum-Funktion, mittels der Elemente auf eine gegebene Zahl beschränkt werden können, sowie verschiedene Methoden, die in einem einzigen Aufruf eine oder mehrere Additionen und eine Multiplikation ausführen.

3.3 Anwendungsbeispiele

Die Anwendungsbeispiele, die Krüger/Westermann und Moravánszky nennen, sind einander sehr ähnlich. Beide implementieren numerische Lösungsverfahren, unter anderem die Konjugierte Gradienten-Methode, auf die wir hier kurz eingehen werden. Für Details zu dieser Methode sei auf die Ausarbeitung von Johannes Wender verwiesen, in der er Thema genauer behandelt.

Die Methode der Konjugierten Gradienten ist ein iteratives Verfahren zur näherungsweisen Lösung linearer Gleichungssysteme der Form $\mathbf{Ax} = \mathbf{b}$, wobei $\mathbf{A}$ und $\mathbf{b}$ vorgegeben sind und der Lösungsvektor $\mathbf{x}$ ermittelt werden soll. Sie ist insbesondere dann gut geeignet, wenn die Matrix $\mathbf{A}$ groß, aber dünn besetzt ist, weil die einzige Operation, die der Algorithmus auf $\mathbf{A}$ ausführt, eine Multiplikation mit einem Vektor ist. Wie wir in Abschnitt 3.1 gesehen haben, kann diese auch bei großen Matrizen effizient ausgeführt werden, wenn deren interne Darstellung für dünn besetzte Matrizen optimiert wurde.

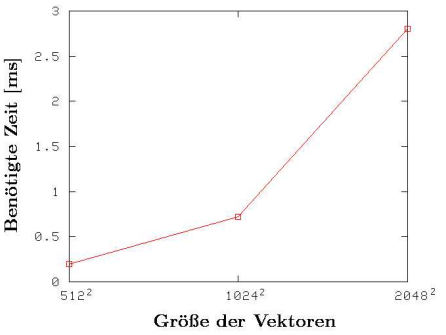
Die Konjugierte Gradienten-Methode kann als Hilfsmittel zur interaktiven Visualisierung physikalischer Simulationen dienen. Wie bereits erwähnt, treten in diesem Zusammenhang große Bandmatrizen auf. Außerdem ist es möglich, dem Verfahren die Lösung des vorhergehenden Zeitschritts als erste Näherung für $\mathbf{x}$ zur Verfügung zu stellen. Da der simulierte Vorgang meist kontinuierlich abläuft, d. h. die Lösung des vorhergehenden derjenigen des aktuellen Zeitschritts meist sehr ähnlich ist, erreicht das Verfahren auf diese Weise oft schon nach wenigen Iterationen die benötigte Genauigkeit und kann abgebrochen werden.

Die konkrete GPU-basierte Anwendung, die Krüger und Westermann mit Hilfe dieses Verfahrens realisiert haben, ist eine interaktive Simulation einer Wasseroberfläche. Der Benutzer kann mit der Maus Kräfte auf die Oberfläche ausüben und die Reaktion darauf beobachten. Auf einem 1024^2 großen Gitter läuft die Anwendung mit 13 Frames pro Sekunde bei 5 Iterationen pro Zeitschritt.

3.4 Einschätzung der erzielten Geschwindigkeit

Auch die von Krüger und Westermann bzw. Moravánszky durchgeführten Messungen zeigen, dass bei Einsatz der GPU für numerische Berechnungen erstaunliche Geschwindigkeiten erreicht werden. Bei der Bewertung der Ergebnisse ist zu beachten, dass die Fragment Shader auf den von ATI hergestellten GPUs derzeit intern nur mit einer Genauigkeit von 24 Bit arbeiten.

Abbildung 6 Geschwindigkeit der Multiplikation zweier Vektoren mit dem Framework von Krüger und Westermann.



Bei den Experimenten von Krüger und Westermann kamen als Hardware ein Pentium 4-Prozessor mit 2,8 GHz und eine ATI 9800-Grafikkarte[7] zum Einsatz. Um die Effizienz ihrer Lösung zu belegen, führen sie eine Messreihe an, in der sie jeweils zwei Vektoren unterschiedlicher Länge miteinander multipliziert haben. Die Ergebnisse sind in Abbildung 6 aufgetragen. Weiterhin benötigten sie für die Multiplikation einer 4096^2 großen Matrix mit einem Vektor 0,23 s; bei einer Bandmatrix der gleichen Größe und Bandbreite 10 sogar nur 0,72 ms.

Sie behaupten, dass ihre Lösung damit um einen Faktor von 12–15 schneller sei als eine als „hoch optimiert“ bezeichnete Softwarelösung, zu der sie keine genauen Angaben machen. Sie räumen jedoch ein, dass diese Software die SSE-Funktionalität des verwendeten Prozessors nicht ausnutzt. Zur Einschätzung dieser Ergebnisse ist wichtig, dass sie bei ihren Messungen die Zeit vernachlässigen, die zur Datenübertragung von und zur GPU notwendig ist.

Moravánszky führt vergleichende Messungen auf zwei aktuellen ATI Radeon-Karten,[8] sowie einem Pentium 4 mit 1,6 GHz Taktrate durch. Als Software

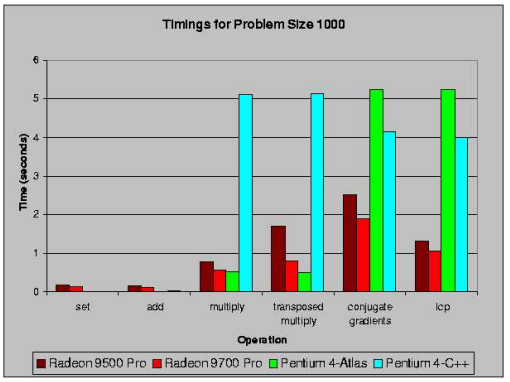
[7] Taktrate vermutlich 325 MHz (Kernsystem) und 580 MHz (Speicher); Quelle: `http://www.ati.com/products/radeon9800/radeon9800pro/compare.html` (13.2.2004)

[8] Geschätzte Taktraten: 275 MHz (Kern) und 540 MHz (Speicher) für die 9500 Pro, 325 MHz (Kern) und 620 MHz (Speicher) für die 9700 Pro; Quellen: `http://www.ati.com/products/radeon9500/radeon9500pro/specs.html` und `http://www.digit-life.com/articles2/digest3d/0203/itogi-video-radeon9700pro.html` (13.2.2004)

kommen dabei sowohl selbst geschriebene C-Programme als auch die Bibliothek ATLAS (Automatically Tuned Linear Algebra Software) zum Einsatz, die besonders effizienten Code zur Multiplikation von Matrizen enthält.

Moravánszky berücksichtigt bei seinen Messungen auch die zur Datenübertragung zur GPU notwendige Zeit. Wie seine Ergebnisse (siehe Abbildung 7) zeigen, dominiert diese bei einfachen Operationen wie Zuweisungen oder Additionen die Laufzeit, so dass die CPU in diesen Fällen wesentlich schneller ist. Bei der Multiplikation zweier 1000^2 großer Matrizen ist die Geschwindigkeit der GPU-Lösung dagegen bereits derjenigen der ATLAS-Bibliothek zu vergleichen.

Abbildung 7 Die Messergebnisse von Moravánszky, entnommen aus [6]: Geschwindigkeit der GPU und CPU beim Rechnen mit 1000^2 großen Matrizen.



Bei den praktischen Anwendungen, wie etwa der beschriebenen Konjugierten Gradienten-Methode, erweist sich die GPU schließlich als effizienter. Insbesondere die ATLAS-Bibliothek ist hier viel langsamer, da diese Algorithmen ihre besondere Stärke, die Matrixmultiplikation, nicht nutzen. Das im Diagramm aufgeführte LCP (linear complementary problem) ist mittels der GPU noch effizienter zu lösen als die Konjugierte Gradienten-Methode, weil die Ausführung dieses Algorithmus nur sehr wenig Kommunikation mit der CPU erfordert.

4 Vergleich der Ansätze

Beim Vergleich der vorgestellten Ansätze sollte man meines Erachtens die unterschiedlichen Zielsetzungen der Arbeiten berücksichtigen. So sind die Bibliotheken von Krüger/Westermann und Moravánszky von einem softwaretechnischen Standpunkt aus betrachtet sicherlich weiter fortgeschritten als das Framework von Thompson et al. – erstere bieten eine einfache C++-Schnittstelle, die ein Anwender nutzen kann, ohne Kenntnisse der Grafikhardware zu besitzen; bei letzterem muss der Anwender sich dagegen selbst mit den Eigenheiten der Hardware beschäftigen und Programmcode in Assembler verfassen.

Ich betrachte dies jedoch nicht als Defizit der Arbeit von Thompson et al.: Sie verfolgen eine wesentlich allgemeinere Zielsetzung als die beiden anderen, nämlich den Einsatz der GPU als Koprozessor zu beliebigen Zwecken. Daher würde die Bereitstellung einer höheren Abstraktionsebene in ihrem Fall die Entwicklung eines Compilers beinhalten – ich halte es für verständlich, dass diese den Rahmen einer ersten Arbeit zu dem Thema gesprengt hätte.

4.1 Vertex oder Fragment Shader?

Es ist bemerkenswert, dass die verschiedenen Autoren für ihre Arbeit unterschiedliche Einheiten der GPU bevorzugen: Thompson et al. führen ihre Berechnungen in Vertex Shadern durch, während die beiden anderen hauptsächlich Fragment Shader nutzen.

Thompson et al. hatten sich gegen den Einsatz von Fragment Shadern entschieden, weil OpenGL keine Assembler-Sprache zu ihrer Programmierung spezifizierte – dies hat sich inzwischen geändert. Auch die Präzision der Fragment Shader, die in ihrer Arbeit noch als unzureichend bemängelt wurde, wurde inzwischen gesteigert: Es existieren bereits GPUs, die auch an dieser Stelle mit einfacher Gleitkomma-Genauigkeit rechnen. Für den Einsatz von Fragment Shadern spricht, dass sie den Zugriff auf Texturen ermöglichen. Diese sind oft die kompakteste Alternative zur Darstellung von Daten im Grafikspeicher.

Insbesondere Moravánszky bemüht sich, eine Arbeitsteilung zwischen Vertex und Fragment Shadern herzustellen; Krüger und Westermann verfolgen bei der Darstellung allgemeiner dünn besetzter Matrizen einen ähnlichen Ansatz. Diese Art von Arbeitsteilung ist vorteilhaft, da Vertex und Fragment Shader in einer Pipeline angeordnet sind, die prinzipiell dann am effizientesten betrieben wird, wenn die beteiligten Stationen möglichst gleichmäßig ausgelastet sind.

4.2 Die gewählten Datenformate für Matrizen

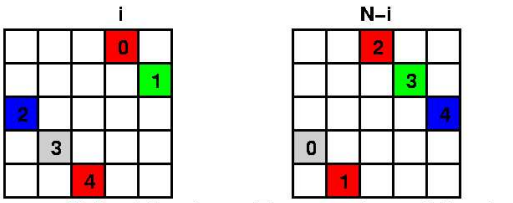
Sowohl Krüger/Westermann als auch Moravánszky verwenden Texturen zur Darstellung von Matrizen. Dennoch kommen sie zu sehr unterschiedlichen Datenformaten: Krüger und Westermann entscheiden sich für ein komplexeres und weniger intuitives Format, dessen wesentlicher Vorteil in der effizienten Darstellbarkeit von Bandmatrizen liegt – ein Problem, auf das Moravánszky in seiner Arbeit ebenfalls stößt, für das er jedoch keinen Lösungsansatz bereithält.

Beide Autoren nutzen die SIMD-Fähigkeit der GPU aus, indem sie alle Farbkanäle der Texel mit Werten füllen. Moravánszky muss auf diese Art der Optimierung bei der Multiplikation mit einer Transponierten besondere Rücksicht nehmen; die Effizienz dieser Operation leidet bei ihm deutlich darunter.

Krüger und Westermann weisen darauf hin, dass bei der Darstellung einer Matrix als Menge von Diagonalenvektoren die Transposition aus einer einfachen Umordnung der Diagonalen und Verschiebung der Indizes besteht: Wie in Abbildung 8 zu sehen, wird dabei die i-te obere Nebendiagonale (mit ihrem Gegenstück aus der unteren Hälfte der Matrix) zur $(N-i)$-ten oberen Nebendiagonalen. Außerdem sind die Indizes um $(N-i)$ verschoben: Man erhält das j-te Element der Transponierten durch Zugriff auf die ursprüngliche Diagonale an Position $(N-i+j \mod N)$. Die Autoren argumentieren, dass der Fragment Shader eine solche Indexverschiebung ohne weiteren Aufwand durchführen könne. Somit wären Operationen mit transponierten Matrizen unter Verwendung ihres Formats nicht aufwendiger als mit den ursprünglichen Matrizen.

Krüger und Westermann nutzen ebenfalls alle Farbkanäle, betrachten dies aber als Implementierungsdetail und gehen im Hauptteil ihres Artikels nicht darauf

Abbildung 8 Transposition einer als Menge von Diagonalenvektoren dargestellten Matrix.



ein. Erst bei der Präsentation der Ergebnisse findet diese Tatsache Erwähnung und ist mit dem kurzen Hinweis versehen, dass dabei das Funktionsprinzip aller vorgestellten Operationen unverändert bleibe.

Ich halte es jedoch nicht für offensichtlich, dass die einfache Art der Transposition von Matrizen davon unberührt bleibt: Wie in Abbildung 8 angedeutet, würden in der ursprünglichen Matrix die Elemente $\langle 0, 1, 2, 3 \rangle$ in die Kanäle eines Texels gepackt; in der Transponierten müssten dagegen die Elemente $\langle 2, 3, 4, 0 \rangle$ zusammengefasst sein, um die vorgestellten Operationen zu ermöglichen. Im allgemeinen Fall reicht eine Verschiebung der Indizes bei Nutzung aller Farbkanäle also nicht mehr aus. Falls Krüger und Westermann tatsächlich wie angekündigt weitere Matrixoperationen auf der Grundlage ihrer Darstellung entwerfen, bleibt abzuwarten, wie sie dieses Problem lösen werden.

5 Weiterentwicklung der Hardware

Nun betrachten wir noch kurz vorgeschlagene sowie bereits umgesetzte Erweiterungen der Grafikhardware und die Vorteile, die sich daraus für allgemeine Anwendungen ergeben.

Thompson et al. schlagen in [8] einige Modifikationen vor, die aktuelle GPUs für allgemeine Anwendungen noch attraktiver machen würden: An erster Stelle nennen sie eine schnellere Schnittstelle für die Datenübertragung zur GPU. Wie wir etwa am Beispiel des 3-SAT-Lösers gesehen haben, stellt diese bislang einen Engpass dar, der den Einsatz von Grafikhardware bei geringen Problemgrößen ineffizient werden lässt. Weitere Vorschläge betreffen die Bereitstellung eines DMA-Controllers zur Übertragung von Daten aus dem Grafik- in den Hauptspeicher des Rechners, die Möglichkeit, den Zustand der Register zwischen zwei Aufrufen eines Vertex Shaders beizubehalten, die Einführung logischer Operatoren, sowie eine höhere Präzision der Berechnungen.

Als Beispiele moderner GPUs, die den aktuellen Stand der Technik repräsentieren, seien an dieser Stelle die Radeon 9800 von ATI[9] sowie die GeForce FX 5900 von NVIDIA[10] genannt. Im Vergleich zu den Modellen, die bei den hier behandelten Arbeiten zum Einsatz kamen, erlauben sie längere Vertex und Fragment Shader. Zudem wurden die Befehlssätze um Sprünge, Schleifen und Subroutinen erweitert – auf diese Weise ließe sich beispielsweise der in Abschnitt 2.1

[9] `http://www.ati.com/products/radeon9800/radeon9800pro/specs.html` (18.1.2004)
[10] `http://www.nvidia.com/page/fx_5900.html` (18.1.2004)

zitierte Code wesentlich einfacher und effizienter formulieren. Schließlich stehen inzwischen Farbformate zur Verfügung, deren einzelnen Kanäle durch Gleitkommazahlen einfacher Genauigkeit dargestellt werden.

Stand der Technik sind inzwischen auch die von Thompson et al. vorhergesagten Karten, die mehrere GPUs nutzen. Laut Aussage von ATI ist es möglich, bis zu 256 ihrer R300-Chips parallel zu betreiben.[11] Tatsächlich bietet Evans & Sutherland, ein Hersteller professioneller Grafikanwendungen, bereits Lösungen an, die diese Möglichkeit nutzen.

Bedeutung des F-Buffers

Die Radeon 9800 von ATI ist die erste Grafikkarte, auf der ein F-Buffer realisiert ist. Die Idee für diese Erweiterung stammt aus dem Artikel [5]; darin argumentieren Mark und Proudfoot, dass für komplexe Fragment Shader ein einziger Pipeline-Durchlauf oftmals nicht ausreicht. In diesem Fall mussten die Zwischenergebnisse bisher (wie wir es auch im Rahmen von Abschnitt 3 gesehen haben) in einem Framebuffer gespeichert und später wieder eingelesen werden. Dies brachte in einigen Fällen Nachteile mit sich, z. B. erschwerte es stark die korrekten Darstellung transparenter Flächen.

Mit dem F-Buffer steht nun ein FIFO-Puffer zur Verfügung, der Ergebnisse zwischen zwei Pipeline-Durchläufen aufnehmen kann. Er stellt für jedes Fragment (statt für jedes Pixel) einen Speicherplatz zur Verfügung; anders als bei Verwendung des Framebuffers können zwei Fragmente also auch dann unterschiedliche Zwischenergebnisse speichern, wenn sie im gleichen Pixel dargestellt werden.

Effektiv ermöglicht es dieses Konzept, komplexe Pixel Shader auf Ebene der Treibersoftware auf mehrere Durchläufe zu verteilen. Diese Umsetzung geschieht ohne Mitwirkung des Programmierers, der somit wie von der CPU gewohnt beliebig lange Shader-Programme schreiben kann.

6 Zusammenfassung und persönliche Bewertung

Im Verlauf dieser Ausarbeitung haben wir zwei Ansätze kennen gelernt, moderne GPUs für allgemeine Anwendungen zu nutzen: In Abschnitt 2 ein Framework, um die Vertex-Einheit der GPU auf Assembler-Ebene als Vertexrechner zu verwenden; in Abschnitt 3 dann zwei Bibliotheken, die auf der GPU allgemeine Operationen der Linearen Algebra ausführen. Wir haben einige Anwendungsbeispiele betrachtet und die von den jeweiligen Autoren gemessenen Ergebnisse genannt; im folgenden Abschnitt gehen wir auf diese nochmals genauer ein.

Unser Vergleich der drei Arbeiten konzentrierte sich auf die Art und Weise, in der die Grafikhardware verwendet wird und insbesondere die zum Einsatz kommenden Datenformate mit ihren jeweiligen Vor- und Nachteilen. Zudem haben wir kurz denkbare und z. T. bereits umgesetzte Änderungen an der Hardware auf ihre Relevanz für unser Thema hin untersucht.

[11] `http://www.techreport.com/etc/2002q3/nextgen-gpus/index.x?pg=3` (30.1.2004)

6.1 Kritische Würdigung der zitierten Ergebnisse

Die in den Abschnitten 2 und 3 genannten Geschwindigkeitsvorteile, die beim Einsatz der GPU gegenüber der CPU auftreten, sind meines Erachtens mit Vorbehalten zu betrachten:

Thompson et al. erwecken den Eindruck, als benachteiligten sie bei dem Geschwindigkeitsvergleich der Matrixmultiplikation geradezu die GPU: In einer Fußnote weisen sie darauf hin, dass der C++-Code für die CPU mit der höchsten Optimierungsstufe übersetzt wurde, die der verwendete Compiler anbot, während der Assembler-Code auf Seiten der GPU ohne Optimierung auskomme. Hier bestanden jedoch schon beim Schreiben des Codes mehr Optimierungsmöglichkeiten als bei Code, der in einer Hochsprache geschriebenen wurde.

Die bei großen Matrizen gemessene deutlich höhere Geschwindigkeit der GPU führen sie ganz auf die hohe Rechenleistung ihrer Gleitkomma-Einheiten zurück. Die Autoren erwähnen nicht, ob die von ihnen geschriebene Software die SSE-Fähigkeiten des eingesetzten Pentium 4-Prozessors nutzt. Noch gravierender ist vermutlich, dass sie unberücksichtigt lassen, dass bei der Verarbeitung derart großer Datenmengen auf heutigen Prozessoren häufig nicht die Rechenleistung, sondern der Zugriff auf den Hauptspeicher den begrenzenden Faktor darstellt.

Eine optimale Nutzung des Prozessors erfordert es daher, schon beim Entwurf der verwendeten Datenformate die Cachekohärenz zu berücksichtigen – eine Aufgabe, die der Compiler in diesem Fall nicht allein übernehmen kann. Berücksichtigt man, dass die ATLAS-Bibliothek in den Messungen von Moravánszky durch Ausnutzen der SSE-Einheit und der Cachekohärenz um den Faktor 10 schneller ist als sein eigener Code, liegt die Vermutung nahe, dass sie auch die Geschwindigkeit der GPU-Implementierung von Thompson et al. überträfe.

Ein direkter Vergleich der Ergebnisse lässt einige Fragen offen: Beide Autoren implementieren den gleichen Algorithmus zur Matrixmultiplikation in C – dennoch benötigt Moravánszky für die Multiplikation zweier 1000^2 großer Matrizen gut 5 Sekunden, Thompson et al. brauchen dagegen etwa 30 Sekunden. Auch die von der GPU-Implementierung benötigte Zeit unterscheidet sich deutlich: Moravánszky gibt hier unter eine Sekunde an, Thompson et al. knapp 10 Sekunden.

Erstaunlich finde ich insbesondere die Tatsache, dass die Softwarelösungen auf fast identischer Hardware so unterschiedliche Ergebnisse zeigen – ich halte daher einen Darstellungs- oder systematischen Messfehler bei wenigstens einem der Autoren für möglich. Die Geschwindigkeit der GPU-Lösungen weichen noch stärker voneinander ab; in diesem Fall unterscheiden sich jedoch die eingesetzte Hardware und die Art ihre Verwendung sehr stark, so dass hier unterschiedliche Ergebnisse zu erwarten waren.

Bei der Einschätzung der Ergebnisse von Krüger und Westermann ist zu beachten, dass sie bei ihren Messungen die Zeit zur Datenübertragung von und zur GPU nicht berücksichtigen. Sie argumentieren damit, dass diese Zeit bei den vorgestellten iterativen Verfahren vernachlässigt werden könne, nennen jedoch Messergebnisse für kleinere Berechnungen, etwa zur Multiplikation zweier Vektoren. Die Messungen von Moravánszky zeigen, dass der Kommunikationsaufwand bei solchen weniger rechenintensiven Aufgaben die Gesamtlaufzeit dominiert.

Der von ihnen genannte Faktor 12–15, um den die GPU bei solchen Aufgaben schneller sei, wird damit zu einer rechnerischen Größe, die praktisch nur noch dann bedeutsam ist, wenn die Daten nach der Berechnung ohnehin zur GPU übertragen werden müssten. Leider stellen sie bei den iterativen Verfahren, die sie als eigentliche Anwendung ihrer Arbeit betrachten, keinerlei vergleichende Messungen an.

Die Arbeit von Moravánszky erscheint mir bei der Präsentation der Ergebnisse am ehrlichsten, kann dafür bei Addition und Matrizenmultiplikation jedoch auch keinen Geschwindigkeitsvorteil der GPU gegenüber der optimierten CPU-Variante vorweisen. In den beiden Anwendungsfällen, in denen seine GPU-Lösungen schneller sind, muss Moravánszky selbst einräumen, dass auf Seiten der CPU suboptimale Algorithmen zum Einsatz kamen, die z. B. die Eigenschaften dünn besetzter Matrizen nicht ausnutzen. Hier stellt sich für mich die Frage, warum Moravánszky beim Entwurf des Datenformats dünn besetzte Matrizen außer Acht lässt und statt dessen auf die leichte Implementierbarkeit der Matrixmultiplikation Wert legt, die er in keinem seiner Anwendungsbeispiele nutzt.

6.2 Persönliche Bewertung

Wie im vorhergehenden Abschnitt begründet, stehe ich den Messergebnissen kritisch gegenüber, mit denen die Autoren zu zeigen versuchen, dass GPUs bereits heute für allgemeine Berechnungen besser geeignet seien als moderne CPUs.

Ein bleibendes Problem sehe ich überdies in der zur Verfügung stehenden Präzision: Sicherlich gibt es Anwendungen, bei denen einfache Gleitkomma-Genauigkeit ausreicht – insbesondere dann, wenn physikalische Simulationen in erster Linie ihres visuellen Effekts wegen ausgeführt werden. Wissenschaftliche Simulationen, die unter Umständen eine höhere Präzision erfordern, stoßen hier jedoch an die Grenzen der Hardware.

Moravánszky weist darauf hin, dass eine Erweiterung der Grafik-Pipeline auf doppelte Genauigkeit unwahrscheinlich ist, da die überwältigende Mehrheit der Hardware-Anwender nach wie vor an grafischen Anwendungen interessiert ist. Für diese böte doppelte Genauigkeit jedoch keine wesentlichen Vorteile.

Aus diesen Gründen betrachte ich die von Thompson et al. in Erwägung gezogene Möglichkeit etwas zurückhaltend, dass in Zukunft Spezialrechner für wissenschaftliche Berechnungen durch Cluster von mit GPUs bestückten PCs ersetzt werden könnten. Dennoch halte ich es für lohnenswert, den allgemeinen Einsatz moderner GPUs weiter zu erforschen und neu erscheinende Hardware auf die neuen Möglichkeiten hin zu untersuchen, die sie für allgemeine Zwecke bietet.

Ich denke, dass allgemeine Berechnungen mittels der GPU insbesondere dann schon in naher Zukunft massive Vorteile bieten werden, wenn die beteiligten Daten ohnehin auf der Grafikkarte vorliegen oder dorthin transferiert werden sollen. Als denkbare Einsatzgebiete sind bereits jetzt eindrucksvolle Simulationen oder die in [7] angekündigte Echtzeit-Filterung von Bildern mittels einer GPU-gestützten FFT (Fast Fourier Transformation) absehbar.

Bei der Ausführung allgemeiner Anwendungen liegt die immense Rechenleistung heutiger GPUs derzeit brach. In diesen Fällen liegt es nahe, die GPU als Koprozessor einzusetzen. Damit sich diese Art der Nutzung etablieren kann, muss

meiner Ansicht nach ein Framework geschaffen werden, das eine hinreichende Abstraktionsstufe bietet, um sowohl dem Programmierer als auch dem Anwender die unmittelbare Auseinandersetzung mit der Grafikhardware zu ersparen.

Einen Ansatz hierzu sehe ich in dem Projekt BrookGPU [2]. Die Programmiersprache Brook erweitert ANSI C um zusätzliche Sprachkonstrukte, die datenparallele und algorithmisch intensive Programmierung erlauben. BrookGPU implementiert einen Teil dieser Erweiterungen – für den Programmierer transparent – auf der GPU. Die resultierenden Programme sind hardwareunabhängig; das Laufzeitsystem bietet über verschiedene Backends prinzipiell die Möglichkeit, eine beim Anwender evtl. vorhandene GPU automatisch zu nutzen und die Berechnungen sonst wie bisher auf der CPU auszuführen.

Schließlich lässt sich anführen, dass die rasante Entwicklung der Hardware in den letzten Jahren immer wieder bestehende Erwartungen übertroffen hat. Es ist anzunehmen, dass sich dieser Trend fortsetzen wird, weil die von GPUs erfüllten Aufgaben in hohem Maße parallelisierbar sind und die Rechenleistung daher – effektiver als dies bei CPUs möglich ist – durch Hinzufügen weiterer Funktionseinheiten erhöht werden kann.

Man darf also gespannt sein, welche weiteren Möglichkeiten des Einsatzes sich in Zukunft ergeben: Obwohl die hier behandelten Arbeiten erst vor kurzem erschienen sind, wurden einige der Hindernisse, die den zitierten Autoren noch im Weg standen, in der Zwischenzeit bereits behoben. Beispiele sind die Einführung bedingter Sprünge in Shader-Programmen, die Bereitstellung eines F-Buffers auf der Radeon 9800 und die erhöhte Genauigkeit der Gleitkommaberechnungen.

Literaturverzeichnis

[1] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transaction on Graphics (TOG)*, 22(3):917–924, 2003.

[2] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Pat Hanrahan, Mike Houston, and Kayvon Fatahalian. BrookGPU: Introduction. `http://graphics.stanford.edu/projects/brookgpu/intro.html`. 2004/01/18.

[3] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003.

[4] E. Scott Larsen and David McAllister. Fast Matrix Multiplies using Graphics Hardware. In *Proceedings Supercomputing 2001*, 2001.

[5] William R. Mark and Kekoa Proudfoot. The F-Buffer: A Rasterization-Order FIFO Buffer for Multi-Pass Rendering. In *SIGGRAPH / Eurographics Graphics Hardware Workshop 2001*, pages 57–63, 2001.

[6] Ádám Moravánszky. *Shaderx2: Shader Programming Tips & Tricks With Directx 9*, chapter Dense Matrix Algebra on the GPU. Wordware Publishing, 2004.

[7] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *SIGGRAPH / Eurographics Workshop on Graphics Hardware 2003 Proceedings*, pages 112–119, July 2003.

[8] Chris J. Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317. IEEE Computer Society Press, 2002.

Johannes Wender
Betreuer: Prof. Philipp Slusallek

Multigrid Solver for Boundary Value Problems and Sparse Matrix Solvers for Conjugate Gradients

Durch das Aufkommen von GPUs sind Grafikkarten programmierbar geworden. Auf ihnen können kleine Programme, sogenannte Shader, Berechnungen durchführen. Hierdurch können GPUs als Co-Prozessor von CPUs verwendet werden. Die Vorteile liegen klar auf der Hand: Heutige Grafikkarten sind leistungsstark und relativ günstig.
Die Eignung von GPUs für komplexe Anwendungen wird anhand zweier konkreter Implementierungen untersucht. Die Implementierung des Multigrid Solvers wird zeigen, dass Leistungssteigerungen um den Faktor 15 gegenüber der reinen CPU-Variante möglich sind.
Bei der Implentierung des konjugierten Gradientenverfahrens wird jedoch deutlich werden, dass es noch einige Schwierigkeiten gibt. Ursache hierfür ist eine Vielzahl von Einschränkungen, denen die Grafikkartenprogrammierung unterworfen ist. Zukünftige Grafikkarten- und Treiber-Generationen werden hier sicherlich Verbesserungen bringen.

1 Einführung

Grafikkarten werden immer leistungsfähiger. Nicht zuletzt durch die Einführung einer programmierbaren GPU stellt sich daher die Frage, ob sich die enorme Rechnenleistung heutiger Grafikkarten auch für nicht-graphische, komplexe und rechenintensive Anwendungen ausnutzen lässt. Wirtschaftlich interessant ist sicher auch das günstige Preis-Leistungs-Verhältnis. Außerdem sind leistungsstarke Grafikkarten heute in beinahe jeden neuen Rechner bereits integiert.
Die enorme Leistung im Vergleich zu CPUs erreichen Grafikkarten dank ihrer hohen Parallelität, einem festen Ablaufschema und einer eingeschränkten Programmierbarkeit.

Andererseits sind dies gleichzeitig die Nachteile der GPU. Der Programmfluss muss mangels bedingter Sprünge durch die CPU gesteuert werden. Um nun die gesamte Leistung der GPU nutzen zu können, ist es notwendig, dass Berechnungen die Parallelität der Grafikkarte möglichst gut ausschöpfen. Die SIMD-Struktur (single instruction, multiple data) besagt, dass jeder Befehl immer auf ein ganzes Datenbündel angewendet wird. Wird eine bestimmte Berechnung ausschließlich für ein einziges Datum benötigt, fallen trotzdem die selben Kosten an. Die parallele Struktur wurde nicht ausgenutzt. Nur strikt seriell lösbare Problem sind folglich weniger gut geeignet.
Viele Berechnungen der linearen Algebra, bspw. Matrixmultiplikationen, bieten sich da schon eher an. Nicht zuletzt ist eine gute Abbildung auf die gegebene Speicherstruktur wichtig. Denn auch der Speicherzugriff ist verschiedenen Beschränkungen unterworfen. Wegen diesen Einschränkungen besteht ein erhöhter Entwicklungsaufwand.
Nun mögen Zweifel an der Brauchbarkeit von GPUs aufkommen. Dass es trotzdem machbar ist und gegebenfalls durch hohe Leistungssteigerungen belohnt wird (Multigrid Solver erreicht Steigerung um den Faktor 15 [3]), wird im folgenden dargestellt.

2 Besonderheiten der GPU

Um Berechnungen von der CPU auf die GPU zu übertragen, müssen zuerst die Besonderheiten der GPU betrachtet werden. Für eine tiefergehende Betrachtung siehe [2].

2.1 Parallelität: SIMD und Pipelines

Grafikkarten besitzen eine Menge von Pipelines. Diese stellen die Daten zur Verfügung, auf welche die Shader-Programme angewendet werden sollen.
SIMD (single instruction multiple data) steht für die parallele Struktur der Grafikkarte. Ein Befehl wird immer auf sämtliche Pipelines gleichermaßen angewendet. Damit die GPU effizient ausgenutzt werden kann, ist es sinnvoll, die Berechnungen möglichst passend auf die gegebene SIMD-Struktur abzubilden. Diese bedeutet, dass Elemente, für welche die selbe Berechnung durchzuführen sind, zu Gruppen zusammengefasst werden. So wird mit den gleichen Kosten die selbe Berechnung nicht für ein, sondern gleich für eine ganze Menge von Werten, abhängig von der Pipeline-Anzahl, durchgeführt.

2.2 Ablauf, Vertex- und Pixel Shader

Die Eingabe der Vertex Engine erfolgt über den Vertex Stream. Nach der Ausführung des Vertex Shaders werden dessen Ergebnisse rasterisiert (3D $\rightarrow$ 2D) und gelangen danach als Eingabe in die Pixel Pipeline. Hier wird ein Pixel-Shader-Programm ausgeführt, welches gleichzeitig auf Daten mehrerer Texturen zugreifen kann (Multitexturing). Beim Texture-Zugriff kann es zu Latenzzeiten kommen.

Das Ergebnis des Pixel Shader gelangt durch Blending schließlich in den Framebuffer. Um dieses Ergebnis in einem späteren Pixel-Shader-Aufruf verwenden zu können, muss es vom Framebuffer in eine Texture kopiert werden. Shader können keine Framebuffer-Inhalten in Texturen kopieren. Die Anweisung erfolgt durch die CPU.
Eingabedaten können als Texturen auf der Grafikkarte gespeichert werden. Um Schleifen zu ermöglichen, welche solange wiederholt werden, bis eine gegebene Bedingung erfüllt ist, muss die CPU den Programmfluss steuern. Hierfür wiederrum ist der Austausch von Daten zwischen der Grafikkarte und der CPU nötig. Die sogenannte Occlusion-Query erweist sich dabei als hilfreich. Zur Steuerung des Programmflusses durch die CPU gehört auch, dass die CPU bestimmt, welcher Shader als nächstes von der GPU ausgeführt wird.
Die Interaktionen zwischen CPU und Grafikkarte wird durch Treiber geregelt, welche Daten puffern, also nicht zwangsläufig sofort weiterleiten. Es ist zwar möglich dem Treiber mitzuteilen, bestimmte Daten sofort zusenden, letztlich bleibt dies doch allein dem Treiber überlassen. Als Folge können Latenzen auftreten, welche sich negativ auf die Leistung niederschlagen.

3 Multigrid Solver for Boundary Value Problems

3.1 Einführung

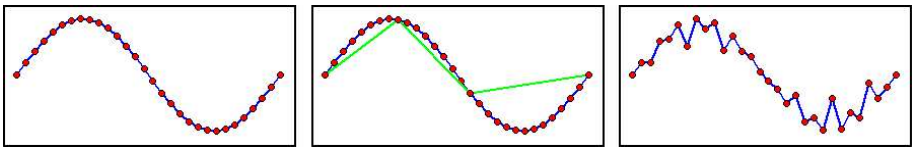
Abbildung 1 Fluidmechanik

Multigrid Solver finden Anwendung bei der Simulation physikalischer Prozesse. Dargestellt ist ein Beispiel der Fluidmechanik. (Abb.: [1])

Lineare Gleichungssysteme können durch den Gauß-Algorithmus gelöst werden. Handelt es sich um große Gleichungssystem, so wird dies bedingt durch Resourcenbeschränkung, z.B. Zeit, quasi unmöglich. In vielen Anwendungsfällen genügen meist gute Näherungen, welche einer festgelegten Fehlertoleranz genügen. Erwähnenswert ist sicher auch, dass beim praktischen, 'exakten' Lösen von Gleichungssystemen, z.B. mit Hilfe des Gauß-Algorithmus, bedingt durch die endliche Darstellung von Gleitkommawerten auf Rechnerarchitekturen, ein Rundungsfehler auftritt, der sich während der Berechnung noch verstärken kann (es wird mit gerundeten Werten weitergerechnet). Bei dem nun vorzustellenden iterativen Verfahren werden solche Fehler dank der Iteration immer wieder herausgerechnet.
Dies ist durch numerische Iterationsverfahren wie dem Gauß-Seidel-Algorithmus möglich. Hierbei wird eine Lösung x_r geraten, welche nach mehreren Iterationen gegen die eigentliche Lösung konvergiert (s. [4]). Die Interation wird solange durchgeführt, bis die Fehlertoleranz nicht mehr überschritten wird.
Das Verfahren eignet sich sehr gut zum Eliminieren von Fehlern hoher Frequenz, nicht aber von Fehlern niedriger. So ergibt sich insgesamt eine Laufzeit

von $O(n^2)$. Es stellt sich die Frage, ob es ein Iterationsverfahren gibt, welches schneller konvergiert.

Abbildung 2 Frequenzumwandlung



Links Ursprüngliche Darstellung mit hohen Frequenzen.
Mitte Reduzieren der Auflösung. Niedrige Frequenzen werden zu hohen Frequenzen (blaue Kurve → grüne Kurve) (Restriktion).
Rechts Die grobe Kurve wurde geglättet und das Ergebnis in die feinere Kurve eingearbeitet (Interpolation). Durch Anwendung des Gauß-Seidel-Algorithmus auf die feinere Kurve können die Unebenheiten in der hohen Frequenz wieder herausgearbeitet werden. (Abb.: [5])

Mit dem Multigrid Solver ist solch ein Verfahren gegeben. Hierbei gibt es nicht ein Gitter, sondern mehrere verschiedener Auflösung. Das feinste Gitter ist das ursprüngliche. Die gröberen Gitter sind durch Restriktion des ursprüngliche entstanden, besitzen weniger Details. Niedrige Frequenzen werden dabei zu hohen Frequenzen (s. Abb. 2). Somit sind nun auch Fehler ursprünglich niedriger Frequenz durch den Gauß-Seidel-Algorithmus effizient lösbar. Durch Interpolation wird das Ergebnis eines gröberen Gitters in das nächst feinere eingearbeitet. Nach jedem Durchlauf wird der Residual berechnet, welcher ein Maß für den derzeitigen Fehler ist. Je kleiner der Residual, desto geringer der Fehler.
Konvergenz wird nun in O(nlogn) erreicht.
Zusammenfassung der Schritte eines Iterationsdurchlaufes:

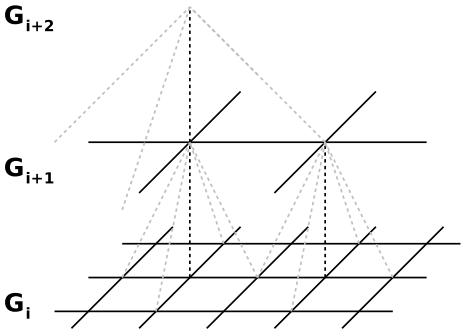
- Glätten des Gitters durch Gauß-Seidel-Algorithmus
- Prüfen, ob jetztige Näherung der Fehlertoleranz genügt (durch Residual), falls ja: Abbruch der Iteration
- Restriktion des gegebenen Gitters auf eines geringerer Auflösung
- Das Problem wird für das grobe Gitter gelöst (Anwendung von Rekursion)
- Durch Interpolation wird die Lösung des groben Gitters in das ursprüngliche eingearbeitet

Zur Veranschaulichung werden die Zusammenhänge zwischen den verschiedenen Gittern, sowie die einzelnen Schritte des Algorithmus noch einmal grafisch dargestellt:

3.2 Anwendung von Multigrid Solver

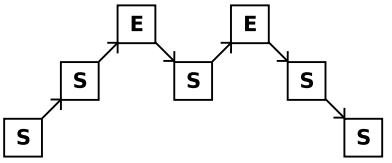
Multigrid Solver ist ein bedeutendes Näherungsverfahren zum Lösen von Gleichungssystemen, welche bei der Simulation dynamischer Prozesse der Strömungs-

Abbildung 3 Ein Gitter wirkt immer auf das nächst gröbere und umgekehrt



Durch Restriktion wird Gitter G_i auf das nächst gröbere Gitter G_{i+1} abgebildet. Dabei gehen Informationen vorloren. Dank der geringeren Knotenanzahl ist das gegebene Problem nun einfacher zulösen. Änderungen im gröberen Gitter wirken auf das feinere Gitter durch die Interpolation zurück. Die Gitter scheinen miteinander elatisch verbunden zu sein.

Abbildung 4 Multigrid Iteration



Dargestellt ist der Iterationsdurchlauf eines Multigird Solvers [4]. Aufstrebende Pfeile geben eine Restriktion, absteigende eine Interpolation an. Je weiter oben, desto gröber das Gitter. S steht für Smoothing, das gegebene Gitter wird geglättet. Ist kein gröberes Gitter mehr dargestellt, so wird das gegeben Gleichungssystem ohne weitere Rekursion gelöst. (Abb.: vgl. [4])

lehre, des Wärmeflusses, der Quantenmechanik und von vielen weiteren physikalischen Erscheinungen auftreten.

3.3 Implementierung der GPU-Variante

Die Berechnungen auf der GPU können mit Hilfe von Vertex Engine und Pixel Pipeline durchgeführt werden. Große Datenmengen können als Texturen gespeichert werden. Wie bereits erwähnt, hat nur die Pixel Pipeline, nicht aber die Vertex-Engine, Zugriff auf Texturen.
Die Bandbreite zum Datenaustausch zwischen CPU und Grafikkarte ist begrenzt (Flaschenhals). Außerdem kann es zu Verzögerungen durch den Treiber kommen. Um dem zu entgehen sollte der Datenaustausch so gering wie möglich bleiben.

Speicher: Texturen

Um den Datenaustausch zwischen CPU und Grafikkarte gering zu halten, werden alle benötigten Daten auf der Grafikkarte in Texturen abgelegt. Das Gitter, auf welches der Algorithmus angewendet werden soll, wir zu Beginn auf der Grafikkarte als Texture gespeichert. Die vom Multigrid Solver verwendeten gröberen Gitter werden ebenso in Texturen gespeichert, je Gitter, zwei Texture.
In der ersten Texture befindet sich das Gitter. In der zweiten Texture ist für jede Gitterzelle der L Operator aus der Grenzwertgleichung als 5-Punkt-Stempel angegeben. Der 5-Punkt-Stempel gibt die Gewichtung der Nachbarpunkte (oben, unten, links, rechts) beim Glätten an (Erläuterung weiter unten).

Ablauf

Die Gitterdaten werden auf der Grafikkarte als Texture gespeichert. Dazu werden die Daten nur einmal hochgeladen. Die Algorithmen werden als Pixel-Shader-Programme implementiert. Diese bieten Texture-Zugriffe und einen höheren Datendurchsatz als bei Vertex-Shader-Programmen.
Schleife:

1. Smoothing: Laplace-Operator wird angewendet (5-Punkt-Stempel, siehe Formel) Hierbei wird die Näherung verbessert.

2. Berechnung des Residual: Jede n-te Iteration wird ein Pixel Shader gestartet, welcher die Abbruchbedinung testet. Die CPU überprüft, ob die Fehlertoleranz erreicht wurde. Gegebenenfalls Abbruch der Iteration.

3. Restriktion des Gitters auf ein Gitter geringerer Auflösung. Rekursion.

4. Interpolation des Ergebnisses vom göberen Gitter in das ursprüngliche.

Im Folgenden werden die einzelnen Schritte mit ihren Shadern näher vorgestellt:

Smoothing

Das Gitter wird im wesentlichen durch Anwendungen eines 5-Punkt-Stempel geglättet. Hierbei wird ein Gitterknoten in Abhängigkeit seiner direkten Nachbarn verändert. Der 5-Punkt-Stempel gibt dabei die Gewichtung der einzelnen Knoten an. Dies wird in der folgenden Gleichung formalisiert (∇^2 ist der Laplace'sche Operator L):

$$\nabla^2 U_{ij} \approx U_{i-1,j} + U_{i+1,j} + U_{i,j-1} + U_{i,j+1} - 4U_{i,j} \tag{1}$$

(weitere Details s. [3])

Berechnung des Residual

In diesem zweiten Schritt wird der Residual-Wert durch Anwendung des Laplace-Operators auf die derzeitige Näherung berechnet. Der Residual-Wert zeigt, wie gut die Näherung bis jetzt ist. Je gringer der Residual-Wert, desto besser die Näherung. Der Residual wird auch bei der späteren Restriktion Verwendung finden.
Jede n-te Iteration wird ein Pixel Shader gestartet. Dieser testet, ob die Fehlertoleranz bereits erreicht ist. Entsprechend wird etwas in eine Texture geschrieben, oder eben nicht. Mit Hilfe der *Occlusion Query* ist es möglich zu prüfen, ob wenigstens ein Pixel geschrieben wurde. Anhand dessen kann die CPU feststellen, ob die Fehlertoleranz denn bereits reicht ist.
Sollte dem so sein, so kann das Iterationsverfahren an dieser Stelle abgebrochen werden und der momentane Nährungswert für x wird ausgegeben. Hierfür wird der aktuelle Wert von x aus der entsprechenden Texture durch die CPU ausgelesen. Ansonsten wird die Iteration fortgesetzt.
Dadurch, dass der Test auf Erreichen der geforderten Toleranzgrenze nur alle n mal erfolgt, kann die GPU mehr produktive Berechnungen durchführen.

Restriktion des Residual

Wenn gerade das i. Gitter G_i betrachtet wird, so findet in diesem Schritt eine Restriktion des Residual statt. Hierbei wird G_i auf das nächst gröbere Gitter G_{i+1} abgebildet. Bei dieser Abbildung gehen Details verloren. Das neue Gitter besitzt nun weniger Knoten. Es wird ein Pixel Shader gestartet, welcher die Residual-Werte des Gitters G_i durch bilineare Interpolation und durch Restriktion der Abtastungen in das gröbere Gitter G_{i+1} überführt. Herauskommt die Beschreibung des Problems, welches es nun für das gröbere Gitter zu lösen gilt, in Form eines inhomogenen Termes f. Die Ergebnisse werden zur weiteren Verwendungen in die entsprechenden Texturen von Gitter G_{i+1} kopiert.

Interpolation der Verbesserung

Die Ergebnisse des nächst gröberen Gitters werden als Korrektur in das eigene Gitter durch lineare Interpolation eingearbeitet. Dies geschieht durch Anwendung eines Pixel Shaders. Als Eingabe werden die Texturen des eigenen Gitters und die des gröberen Gitters verwendet. Das Ergebnis wird wieder in die Texturen des eigenen Gitters geschrieben. Somit wurde für das eigene Gitter ein Update durchgeführt.

3.4 Ergebnisse

Gitterauflösung	CPU Iter./s	GPU Iter./s	Leistungssteigerung
256 x 256 Pixel	4,7	65,8	14,00
512 x 512 Pixel	1,1	17,4	15,82
1024 x 1024 Pixel	0,3	4,1	13,67

Die Simulation der Wärmeleitung mit Multigrid Poisson Solver ergibt eine Leistungssteigerung um den Faktor 15, gegenüber der reinen CPU-Variante, was ein beachtliches Ergebnis ist (s. Tabelle). Die theoretisch zu erwartende Leistungssteigung wird leider nicht genannt(vgl. Ergebnisse zur Implementierung des konjugierten Gradientenverfahrens). Es ist aber zu erwarten, dass es nahezu optimal ist, da alle nötigen Daten der GPU direkt zur Verfügung stehen und die GPU soweit es geht von der CPU unabhängig ist. Außerdem kann auf die benötigten Daten direkt zugegriffen werden, ohne einen Umweg über verschachtelte Texture-Koordinaten nehmen zu müssen (vgl. Implementierung des konjugierten Gradientenverfahrens für schwach besetzte Matrizen).
Zum Leistungsvergleich wurde ein AMD Athlon 1600 mit 1GB RAM und einer ATI Radeon 9700 verwendet, obwohl die Anwendung eigentlich auf einer nVidia NV30 laufen sollte, welche den Autoren jedoch noch nicht zur Verfügung stand.
Laut Autoren liegt die Fehlertoleranz in der durch Floating-Point vorgegeben Genauigkeit. Um eine höhere Genauigkeit zu erzielen, müsste also die derzeitige 32-Bit-Floating-Point-Architektur der Grafikkarten entsprechend erweitert werden. Da dies eine Frage der Hardware ist, ist dies nicht beliebig veränderbar. Daher wäre gegebnenfalls eine komplexere Abbildung auf die gegebene Struktur von Nöten. So könnten für einen Wert mehrere Speicherzellen verwendet werden, was jedoch sicherlich erhebliche Leistungseinbußen mitsichbringen würde.
Ebenso halten die Autoren die Speicherplatzbeschränkung bei den Texturen für hinderlich, was durch Texture-Kompression behebbar sei.
Meiner Meinung ist diese Lösung mehr ein Aufschieben des Problems. Sinnvoll wäre eher, den Algorithmus so umzugestalten, dass sich nicht sämtliche Daten auf der Grafikkarte befinden müssen. Dabei ist selbstverständlich zu beachten, dass Daten, welche einmal auf der Grafikkarte sind, möchlichst lange verwendet werden und ein Datenaustausch nur erfolgt, wenn es wirklich notwenig ist. Hierdurch würden mögliche Verzögerungen durch unnötiges Hin- und Herschieben der Daten vermieden.

4 Sparse Matrix Solvers for Conjugate Gradients

4.1 Einführung

Unstructured sparce matrix solver taucht häufig bei der Bestimmung von Nährungslösungen von linearen aber auch nicht linearen partiell differenzierbaren Gleichungen beliebiger Gitter auf. Hierbei lässt sich das konjugierten Gradienten-Verfahren gut anwenden.
Beim konjugierten Gradienten-Verfahren wird durch den Gradienten bestimmt, in welche Richtung die jetztige Näherung verändert werden muss, sodass das Verfahren gegen die gesuchte Lösung konvergiert. Um ein solches Verfahren auf die Grafikkarte zu übertragen, muss eine Abbildung der schwach besetzten Matrizen (Sparse Matrix) auf Texturen der Grafikkarte, sowie die eigentlichen Algorithmen als Vertex- oder Pixelshader bereitgestellt werden.
Um eine optimale Leistung zuerhalten, müssen die gegebenen Voraussetzungen beachtet werden. So handelt es sich um schwach besetzte Matrizen, also Matrizen, bei denen die meisten Einträge null sind. Dies lässt sich bei Matrizenmultiplikationen ausnutzen: Das Produkt zweier reeller Zahlen, wobei ein Faktor

null ist, wird immer null ergeben. Daher können viele Berechnungen eingespart werden.
Aus besagten Gründen lassen sich schwach besetzte Matrizen auch kompakt abspeichern. Nur die Werte und Positionen von Einträge ungleich null werden hinterlegt. Ein nicht abgespeicherter Eintrag der schwach besetzten Matrix ist folglich null.
Die Abbildung der Matrix auf den gegebenen Texturenspeicher ist dabei nicht mehr direkt möglich. Da nur noch Einträge ungleich null abgespeichert werden, ist durch diese kompakte Speicherung zwar eine kleinere Texture ausreichend, es muss allerdings eine bijektive Abbildung zwischen Speicherung und abgespeicherter Matrix existieren, weshalb zusätzlich eine Zeigerstruktur nötig ist. Da die GPU allerdings Zeiger, wie sie bei CPUs selbstverständlich sind, nicht kennt, muss der selbe Effekt über einen Umweg erreicht werden. Es wird eine Zeigerstruktur *simuliert*.
Nicht nur die kompakte Abbildung der schwach besetzten Matrizen ist wichtig, sondern auch die effiziente Abbildung der Algorithmen auf die parallele Struktur der GPU. Denn beim Ausführen eines Shaders soll die GPU Berechnungen nicht nur für einen einzelnen Wert, sondern, abhängig von der Pipeline-Größe, gleich für eine ganze Menge von Werten durchführen.
Ebenso wichtig sind mögliche Latenzzeiten, die beim Datenaustausch oder Speicherzugriff auftreten können.

4.2 Umsetzung

Gesucht wird ein *Sparse Matrix Solver for Conjugate Gradients*. Dass es sich aussschließlich um schwach besetzte Matrizen handelt, lässt sich bei der Implentierung ausnutzen. Dies ist jedoch wegen den Einschränkungen, denen die Programmierung von Grafikkarten unterworfen ist, nicht trivial. Es geht um die Frage, wie man eine schwach besetzte Matrix am geschicktesten auf Texturen abbildet und wie die Shader auf diesen Speicher zugreifen können.
Moderne Grafikkarten unterstützen Multitexturing. Dies bedeutet, ein Pixelshader kann für jede Pipepline Daten aus mehreren Texturen beziehen. Solange die Bandbreite nicht ausgelastet ist, sollten dabei keine zusätzlichen Verzögerungen entstehen.
Für jede Pipeline, welche in eine Pixel Pipeline hineinführt, geht auch wieder genau eine hinaus. Dabei kann das Ergebnis für eine Pipeline nur in ihr entsprechendes Ausgaberegister geschrieben werden. Es ist also nicht möglich, ein Ergebnis auf mehrere Register zu kopieren ($1 \rightarrow n$).
Ebenso ist es nicht möglich, auf mehrere Pipelines zuzugreifen und daraus ein Ergebnis zu berechnen. Bei der Implementierung des Skalarproduktes wird dies interessant werden. Es ist auf einfache Weise möglich, die einzelnen Komponenten der beteiligten Vektoren paarweise zu multiplizieren. Jeder Vektor ist dabei in einer eigenen Texture abgespeichert. Dies ist dank Multitexturing machbar. Probleme gibt es allerdings beim Aufaddieren der berechneten Produkte. N Produkte können nicht in einem Zuge zu einer Summe aufaddiert werden. Dies ist nicht mit der SIMD-Struktur vereinbar, welche vorgibt, dass alle Befehle immer gleichermaßen auf alle Pipelines angewendet werden und das diese unabhängig von einander sind.
Letztlich werden im Grafikkartenspeicher immer Vektoren von vier Floating

Points gespeichert. Da aber nur einfache Floating Points benötigt werden, können in jedem Vektor vier Floating Points untergebracht werden. Dadurch wird der Speicher optimal ausgenutzt.

4.3 Anwendung

Anwendung findet das Konjugierten-Gradienten-Verfahren für schwach besetzte Matrizen, ebenso wie der Multigrid Solver, als effektives Verfahren zum nährungsweise Lösen von Gleichungssystemen der Form $Ax = \mathrm{b}$ mit $A \in \mathbb{R}^n \mathrm{x} \mathbb{R}^n$. Diese treten vorallem bei der Simulation physikalischer Prozesse auf.
Weitere Anwendung findet das Verfahren beim der Entfernung von Rauschen in 3D-Bildern, welche mit einem 3D-Scanner aufgenommen wurden.

4.4 Implementierung auf GPU

Zuerst ist zu klären, was, auf welche Weie abgebildet wird. Hierzu wird der *DOF* (*degrees of freedom*) untersucht. Dieser Freiheitsgrad umfasst alles, was bei einem bestimmten Problem nicht festgelegt ist, sondern sich ändern kann. Als Beispiel wäre hier die Position oder die Geschwindigkeit von Teilchen zu nennen, welche sich im Raum bewegen, wobei die Simulation dieser Bewegung berechnet werden sollen. Diese werden in 2D-Gittern bestehend aus Dreiecken oder Vierecken, bspw. in 3D-Gittern bestehend aus Tetraeder o.ä. dargestellt.
In dieser Implementierung wird ein Dreiecksgitter für die Darstellung der schwach besetzten Matrizen verwendet, da Grafikkarten auf Berechnungen mit Dreiecken optimiert sind. Nachfolgend werden einige Mengen definiert:
V:=$\{v_i \mid 1 \leq i \leq n \wedge n \in N\}$ ist die Knotenmenge
E:=$\{e_{ij} \mid$ Es gibt Kante, welche v_i mit v_j verbindet$\}$.
N(i):=$\{v_j \mid e_{ij} \in E\}$ Menge der benachbarten Knoten von v_i.
Ein generisch erzeugter DOF im Zusammenhang mit Knoten v_i wird als x_i bezeichnet.
Ein Beispiel DOF, welcher eine 3D-Position enthält wäre mit $x_i := (x, y, z)_i$ oder Texture-Koordinaten durch $x_i := (s, t)_i$ gegeben.
Ziel ist es nun, das Gleichungssystem $Ax = f$ zulösen, also:

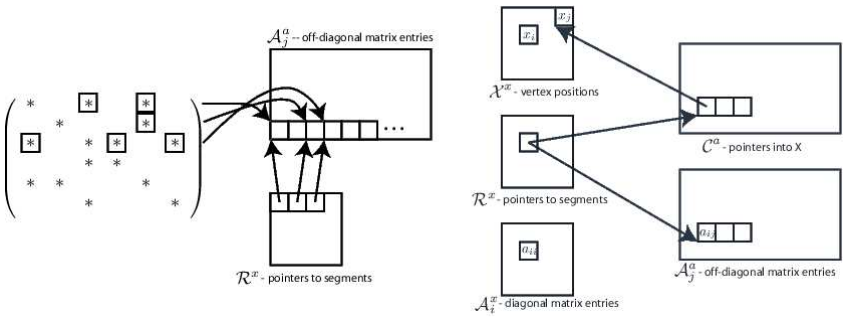
$$\forall i = 1..n : a_{ii}x_i + \sum_{j \in N(i)} a_{ij}x_j = f_i \tag{2}$$

.

Die gesuchte Lösung ist hierbei x. Nach dem obigen Beispielen könnte dies die Positionen von Teilchem im Raum sein oder aber auch Texture-Koordinaten.
f_i legt die Grenzwertbedingungen fest. Der Koeffizient a_{ij} hängt vom Zustand der beiden Dreiecke, welche die Kante e_{ij} bilden, ab. Die Koeffizienten werden also als Dreieckskanten gespeichert, stellen also die Matrixeinträge dar.
Sollte das linerare Gleichungssystem symmetrisch positiv definit sein, so kann dieses konjugierten Gradientenverfahren verwendet werden. Im Folgenden wird nur dieser Fall betrachtet. Zum Lösen allgemeinerer Gleichungssysteme wäre eine allgemeine Variante des konjugierten Gradientenverfahrens notwendig.

Speicher: Die Texturen

Abbildung 5 Texturezusammenhang



Links ist dargestellt, wie die Nicht-Diagonaleinträge der Matrix A in der Texture A_j^a abgelegt werden. Die Einträge einer Zeile kommen nacheinander in ein Segment. Die Startpositionen, ab denen die einzelnen Segmente beginnen, werden in der Texture R^x gespeichert.
Rechts wird die Zeigerstruktur dargestellt. Die Quelle einer Kante zeigt an, dass der Wert des Pixels als Texture-Koordinate benutzt wird. Die Texture-Koordinate fungiert als Zeiger auf Pixel anderer Texturen. Im Beispiel sind dies die Pixel, auf welche die Pfeile verweisen. Erläuterungen folgen im Text. (Abb.: [1])

Alle Daten werden in Texturen auf der Grafikkarte gespeichert. Dabei werden die folgenden Texturen unterschieden:

- A_i^x Diagonalemente der Matrix A
- A_j^a Übrige Matrixeinträge von A, welche ungleich null sind
- X^x Der Vektor x, welche die Nährungslösung darstellt
- C^a Zeigertexture
- R^a Zeigertexture

Texturen mit gleichem Exponent besitzen die selbe Größe und jedes Element einer Texture steht im logischen Zusammenhang mit dem einer anderen Texture, wenn es sich dort an derselben Stelle befindet.
Die Texturen hängen wie folgt miteinander zusammen. In der Texture A_j^a sind die Matrixeinträge, welche nicht zur Diagonalen gehören und ungleich null sind, in Segmentabschnitten zusammengefasst: Die Matrixeinträge einer gleichen Zeile werden immer in das gleiche Segment von A_j^a gespeichert.
Um später zu wissen, wo welche Segment innerhalb dieser Texture beginnt, werden Texture-Koordinaten in der Texture R^x gespeichert. R_t^x gibt dazu die Startposition des Segmentes an, in der die Einträge der t. Zeile von A gespeichert sind. Zu beachten ist: Texture-Koordinaten sollten eigentlich vom Typ Integer sein. Dieser wird aber von Grafikkarten nicht unterstützt, weshalb der Floating-Point-Datentyp dazu quasi missbraucht wird.

In der Texture C^a ist nun passend zu Texture A_j^a für jedes Pixel aus A_j^a gespeichert, welchem Matrixeintrag er in der Matrix A entspricht.
Zusätzlich zu den hier vorgestellten Matrizen gibt es noch zwei weitere. In diesen werden die Matrixeinträge von A^T auf ähnliche Weise wie die von A gespeichert (Diagonaleinträge getrennt von übrigen, nur Einträge ungleich null werden gespeichert).

Multiplikation schwach besetzter Matrizen

Hierfür ist das Skalarprodukt nötig. Es werden die bereits beschriebenen Texturen verwendet:

$$j = R^x[j] \tag{3}$$

$$Y^x[i] = A_i^x[i] * X^x[i] + \sum_{c=0}^{k_i-1} A_j^a[j+c] * X^x[C^a[j+c]] \tag{4}$$

Mit $R^x[j]$ wird die Startposition des Segmentes bestimmt, welches die Matrixeinträge der Zeile j enthält. $Y^x[i]$ bekommt das Skalarprodukt der j. Matrixzeile von A mit dem Vektor X^x zugewiesen. Die Berechnung der Komponentenprodukte erfolgt getrennt. Erst das Diagonalelement. Für die übrigen Elemente wird eine Zeigerstruktur simuliert. Mit $C^a[j+c]$ wird die zugehörige Komponente von X^x zur Multiplikation mit $A_j^a[j+c]$ bestimmt. $A_j^a[j+c]$ ist der c. Nichtdiagonaleintrag ungleich null der j. Matrixzeile. Nichtdiagonaleinträge, welche gleich null sind, werden nicht betrachtet. Hierdurch wird das Ergebnis nicht verändert.
Die getrennte Speicherung der Diagonalen von den übrigen Matrixeinträgen erweist sich als vorteilhaft. Die Diagonaleinträge werden für die Diagonalvoraussetzung, Division der Redisuals durch die Diagonaleinträge, benötigt.
Um die Summe der einzelen Produkte ausrechnen zu können, wird ein sogenannter Reduktionsoperator verwendet. Da die Addition kommutativ ist, dürfen die Summanden in beliebiger Reihenfolge aufaddiert werden. Die gilt für reelle Zahlen. Bei endlicher Gleitkommadarstellung, wie es nunmal bei konkreten Rechnerarchitekturen der Fall ist, trifft dies natürlich nicht mehr zu. Dieser Fehler wird von den Autoren jedoch als vernachlässigbar angesehen.
Bei der Summenreduktion sind die Summanden in einem Rechteck angeordnet. In jedem Durchlauf des Summenreduktionsoperators wird immer die Summe von vier benachbarten Werten gebildet. Nach jedem Schritt bleiben folgisch nur noch etwa ein viertel der Summanden übrig. Somit sind $\lceil log_4 n \rceil$ Schritte nötig, bis das Ergebnis von $\sum_{i=1}^{n} x_i$ tatsächlich vorliegt.

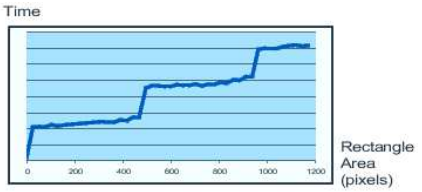
Berechnung von Matrixeinträgen

Werden Flussprobleme betrachtet, so ändert sich im Laufe der Zeit die Matrix A. Die Einträge sind abhängig von x. Daher muss die Matrix immer wieder in Abhängigkeit von x neu berechnet werden. Hierfür werden zwei neue Kernel benötigt. Der eine für die Berechnung der Diagonaleinträge, der zweitere für die

übrigen Einträge.
Es werden lokale Steifigkeitsmatrizen berechnet, welche zu einer einzelnen großen zusammengefasst werden sollen (Details siehe [1]).
Was hierbei wichtig ist: Wie bei der Berechnung des Skalarproduktes, bei dem die Einzelprodukte aufaddiert werden müssen, kommt es auch hier zu Problemen. Diesmal ist jedoch eine Streuungsoperation notwendig, welche wie bereits erwähnt, wegen der SIMD-Struktur der GPUs nicht unterstützt wird. Wie dies gelöst werden kann, ist in [1] genau beschrieben.

4.5 Optimierungen

Abbildung 6 Speicherlatenzen



Ergebnis der experimentellen Bestimmung der Speicherlatenzen beim Texture-Zugriff in Abhängigkeit der Flächengröße. Es sind Sprünge erkennbar, welche die Speicherlatenzen aufzeigen. An diesen Stellen muss vor Anwendung eines Befehles auf einen Datensatz auf die nötigen Texture-Daten gewartet werden. Für eine Fläche von 100 Pixel ist eine nahezu ebenso große Zeitspanne von nöten, wie für eine Fläche von 500 Pixel. (Abb.: vgl. [1])

Müssen zum Ausführen eines Befehles Daten aus einer oder mehreren Texturen gelesen werden, so können Latenzen auftreten. Durch die Multi-Thread-Struktur ist es möglich, dass diese nicht zur Geltung kommen. Seien q Datensätzen und p Befehlen gegeben. Ein Datensatz habe dabei die gleich Anzahl an Elementen, wie die Grafikkarte Pipelines hat. In einem Schritt wird also immer ein Befehl auf einen ganzen Datensatz angewendet. Bei q Datensätzen vergehen also q-1 Schritte, bevor der nächste Befehl auf einen Datensatz angewendet wird. Ist die Zeitspanne lange genug, um die Speicherzugriffe für den Folgebefehl durchzuführen, kommt es zu keinen speicherbedingten Verzögerungen.
Um herauszufinden, wie viele Datensätze nötig sind, damit keine speicherbedingten Verzögerungen auftreten, haben die Autoren Befehlsfolgen auf verschieden große Pixelflächen angewendet und die jeweils verstrichene Zeit bestimmt (s. Grafik, Details s. [3]). Zum Abarbeiten dieser Flächen fasst die GPU entsprechend der Pipeline-Anzahl mehrere Pixel zu Datensätzen zusammen. Es zeigt sich, dass alle 512 Pixel Sprünge auftreten. Dies bedeutet, es ist egal, ob die Befehlsfolge auf 10, 100 oder 500 Pixel angewendet wird, es wird immer nahezu die selbe Zeit benötigt. Erst ab 512 Pixel steigt die Zeitdauer sprunghaft an, bleibt dann aber wieder bis 1024 nahezu konstant.
Diese Rechtecke, welche hierbei betrachtet wurden, setzen sich immer aus zwei rechtwinkligen Dreiecken zusammen, deren Hypotenusen deckungsgleich sind. Für die Hypotenusen beider Dreiecke werden somit Rasterisierungen für die gleichen Pixel durchgeführt. Dies bringt allerdings nichts, vielmehr gehen hier-

bei Berechnungen verloren. Ein Recheck mit den Maßen 32x16 Pixel=512 Pixel sollte nicht verwendet werden, da hierbei für 32 Pixel die Rasterisierung doppelt durchgeführt wird, also insgesamt für 544 Pixel (bedingt durch Hypotenusenüberdeckung). Dies liegt aber schon im nächsten Interval und bräuchte somit erhelblich länger (s. Abb. 6). Die Autoren wählten daher Flächen der Größe, 26x18 Pixel = 468 Pixel und welche beim Hinzuaddieren von 26 überflüssig berechneten Pixel mit 494 Pixel noch gerade unterhalb der Grenze von 512 Pixel bleiben. Diese Wahl lieferte den Autoren die besten Ergebnisse, wobei relativ wenige Berechnung doppelt durchgeführt wurden und Verzögerungen durch Speicherzugriffe ebenso nicht zum tragen kamen.
Eine letzte Optimierung betrifft die Wahl der Programme, welche zur Berechnung herangezogen werden. Wegen der schach besetzten Matrizen sind die Matrix in eine Diagonale und die übrigen Einträge ungleich null getrennt. Die Nichtdiagonaleinträge ungleich null sind zeilenweise in Segmente einer Texture abgespeichert. Da nicht in jeder Zeile gleichviele Nullen enthalten sind, ergeben sich zwangsläufig verschieden lange Segmente. Um eine Zeile mit dem Vektor x multiplizieren zu können, sind, da eine Zeile unterschiedlich viele Nullen enthalten kann, verschieden viele Einzel-Multiplikationen nötig. Es wird daher eine Klasse von Programmen P_k bereitgestellt. Mit k wird angegeben, wieviele Einträge ungleich null eine Zeile höchstens haben darf, damit dieses Programm noch anwendbar ist.
Durch eine richtige Sortierung wird erreicht, dass jede Zeile gerade von dem Programm multipliziert wird, mit dem diese Multiplikation gerade noch durchführbar ist und damit die wenigstens überflüssigen Einzel-Multiplikationen aufweist.

4.6 Ergebnis

Alle Teile eines allgemeinen konjugierten Gradienten-Verfahrens wurden von den Autoren umgesetzt. Zusätzlich wurde eine spezielle Matrix für den geometische Fluss erstellt, welche nach jedem Iterationschritt neu berechnet werden musste (Matrixeinträge von A abhängig von x).
Die Programme wurden auf einem 500MHz mit einer GeForce FX getestet. Dabei ergab sich im praktischen Durchlauf ein wesentlich schlechterer Wert, als theoretisch zu erwarten wäre. Bei 37000 Knoten waren nicht 500, sondern nur 120 Matrixmultiplikationen pro Sekunde möglich. Die Autoren führen dies auf eine Cacheüberlastung beim Speicherzugriff zurück(beim Multitexturing).
Für die Leistung spielt der verwendete Reduktionsoperator auch eine sehr wichtig Rolle, macht er doch einen Großteil bei der Skalarberechnung aus. So werden beim Zusammenfassen eines 200x200 Feldes zu einem 100x100 Feldes 10000 Befehlsaufrufe benötigt. Laut Autoren müssten theoretisch 15000 dieser Aufrufe pro Sekunde abarbeitbar sein, ihr Summenreduktionsoperator schafft jedoch nur 3400, weniger als 1/4. Hier besteht also noch die Möglichkeit zu einigen Verbesserungen, welche sich positiv auf die Leistung auswirken.
Pro Sekunde werden 110 Schleifendurchläufe erreicht, was zu 20 Smoothing-Schritten je Sekunde reicht. Jedoch werden mehrere Smoothing-Schritte benötig, um eine brauchbare Lösung für ein Gleichungssystem zu approximieren. Daher kann diese Implementierung noch nicht als Echtzeitanwendung verwendet werden.

Die größten Leistungseinbußen werden sicherlich durch die vielen verschiedenen Texture-Zugriffe zur Realisierung der kompakten Darstellung der schwach besetzten Matrizen (Zeigerstruktur) und durch den Summenreduktionsoperator hervorgerufen. Für ersteres würde die Einführung von Zeigern, wie sie bei CPUs üblich sind, Abhilfe schaffen. Der Umweg über Texturekoordinaten in Form von Floating-Points in dritten Texturen abzuspeichern, würde entfallen.
Für letzteres, der Summenreduktionsoperator, stellt sich die Frage, ob dies überhaupt, bedingt durch den parallelen Aufbau, zuverbessern ist.
Letztlich bleiben noch Treiberprobleme übrig. So geben die Autoren an, dass z.Z. der *pbuffer* nur 200 mal pro Sekunde gewechselt werden kann, was eine obere Schranke für ihre Implementierung darstellt, da diese auf den Wechsel des pbuffer angewiesen ist. Diese Beschränkung der Grafikkarten soll jedoch in Zukunft entfallen.

5 Zusammenfassung und Ausblick

Beim Vergleich der beiden Implementierungen fällt auf, dass bei der ersten eine signifikante Leistungssteigerung gegenüber der reinen CPU-Variante erreicht wurde: Eine Steigerung um den Faktor 15. Die zweite Implentierung hinkt hingegen beträchtlich den theoretischen Erwartungen hinterher. Hierbei darf man allerdings nicht den Fehler machen, Äpfel mit Birnen zu vergleichen. Während bei der Implentierung des konjugierten Gradientenverfahrens keine Vergleich mit einer reinen CPU-Variante angestellt wurden, fehlt bei der ersten die Überlegung, welche Werte denn theoretisch mit der GPU-Variante erreichbar sein müssten.
Letztlich kann man allerdings folgern, dass sich der Mutligrid Solver besser, im Gegensatz zum konjugierten Gradientenverfahren, auf die GPU-Architektur übertragen lässt. Die schwach besetzten Matrizen lassen sich nicht sowohl speichereffizient als auch optimal für den späteren Zugriff auf die Texturespeicher abbilden. Entweder werden die Texturen riesig groß sein und unnötige Berechnungen wegen Nulleinträgen durchgeführt werden. Oder aber die Verwendung eines einzigen Matrixeintrages zieht, bedingt durch die Simulation der Zeigerarithmetik, gleich mehrere Texture-Zugriffe nach sich.
Daher ist es erforderlich eine effiziente Zeigerstruktur auf GPUs bereitzustellen, sowie die Treiber in punkto Latenzzeiten zu verbessern.
Um die Nutzbarkeit der Grafikkarte weiter zu erhöhen, wären Erweiterungen wie if-then-else-Konstruktionen, Schleifen mit Schleifenabbruchbedingungen, gegebenenfalls Sprünge wünschenswert. Es stellt sich allerdings die Frage, ob diese Neuerungen nicht einen Bruch mit dem eigentlich Prinzip der Grafikkarte darstellen. Eine derartige parallele Struktur wird nunmal erst unter gewissen Voraussetzungen konstruierbar. Wird zuviel Freiraum gelassen, verkompliziert sich die Angelegenheit zunehmens, so dass die Komplexität, letztlich auch die Produktionskosten der Grafikkarte steigen werden, wohingegen die Leistung eher abnimmt.
Vielleicht ist es aber auch sinnvoll, für derartige Anwendungen kostengünstige und leistungsstarke Streamprozessoren zu entwicklen, welche nicht so sehr auf Grafik ausgerichtet sind. Z.Z. müssen die Algorithmen auf Vertex- und Pixelshader abgebildet werden, welche sich doch eher auf Grafikanwendungen beziehen.

Als letztes möchte ich die Programmiersprachen betrachten. CG bietet zwar schon eine gewisse Abstraktion von der Grafikkartenarchitektur. Jedoch muss ein Programmierer, welcher sich die Grafikkarte für Simulationsberechnungen von physikalischen Prozessen nutzbar machen möchte, sich erst mit gewissen Eigenheiten vertraut machen, damit die Algorithmen sinnvoll auf die gegebene Architektur abgebildet werden können.
Hilfreich wäre es, eine bekannte Programmiersprache verwenden zu können. Programme könnten wie gewöhnlich entworfen werden. Erst der Compiler würde sich um die Umsetzung auf die Grafikkarte kümmern und dabei selbständig die Passagen finden, welche sich von einer GPU gut berechnen lassen und diese durch Shader-Programme und eine gute Speicherabbildung auf die Architektur der Grafikkarte übertragen.

Literaturverzeichnis

[1] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *SIGGRAPH 2003*, 2003.

[2] Michael Doggett. Programmability features of graphics hardware. *SIGGRAPH 2003*, 2003.

[3] Nolan Goodnight, Gregory Lewin, David Luebke, and Kevin Skadron. A multigrid solver for boundary value problems using programmable graphics hardware. 2003.

[4] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing - Second Edition*, 2002.

[5] Nick Wyman. How multigrid solver acceleration works. `http://www.cfdreview.com/features/01/11/28/2217256.shtml`. 30. 11. 2001.